CSE 322

NS3 Term Project Final Submission

Submitted by:

Simantika Bhattacharjee Dristi

1705029

Supervised by:

Syed Md. Mukit Rashid

Lecturer, Department of CSE, BUET

Task-A

Wired Network:

Used topology:

- PointTopoint Dumbbell Topology
- Bottleneck datarate:500Mbps
- Bottleneck delay:0.5ms
- Packet Size 200Byte

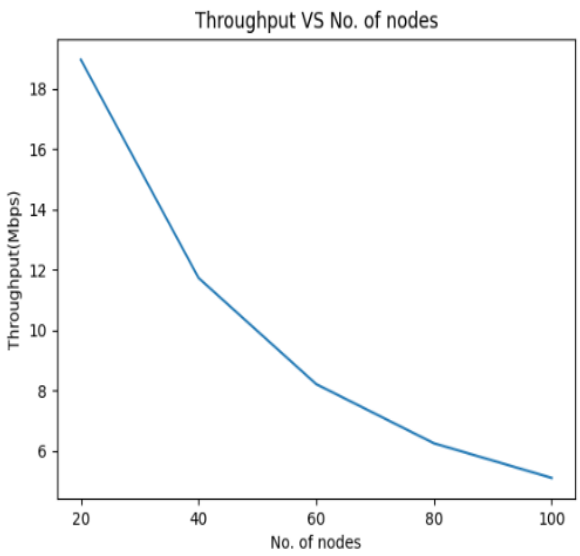
Varied Parameters:

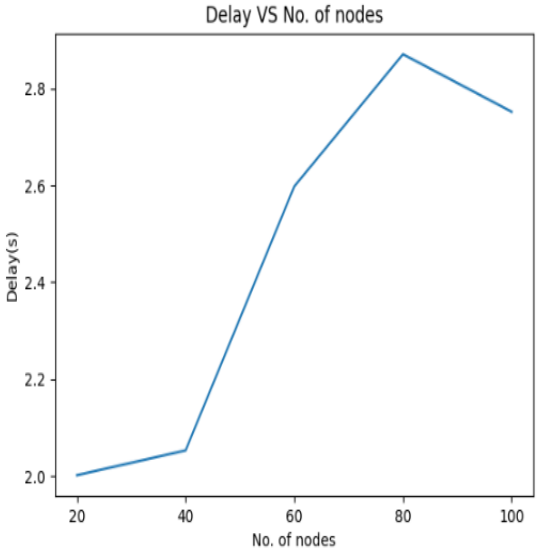
- No of Nodes
- No of Flows
- Packets per Second

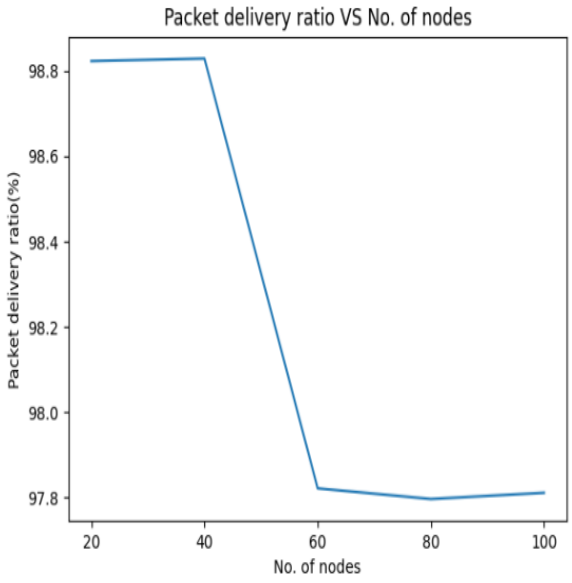
Calculated metrics:

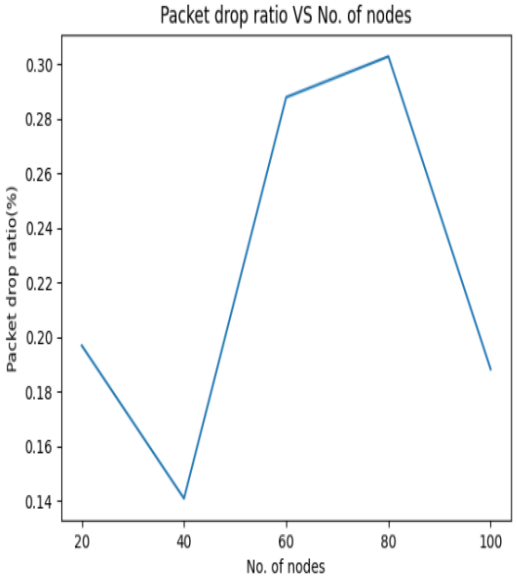
- Network Throughput
- End-to-end delay
- Packet delivery ratio
- Packet drop ratio

Generated Results:

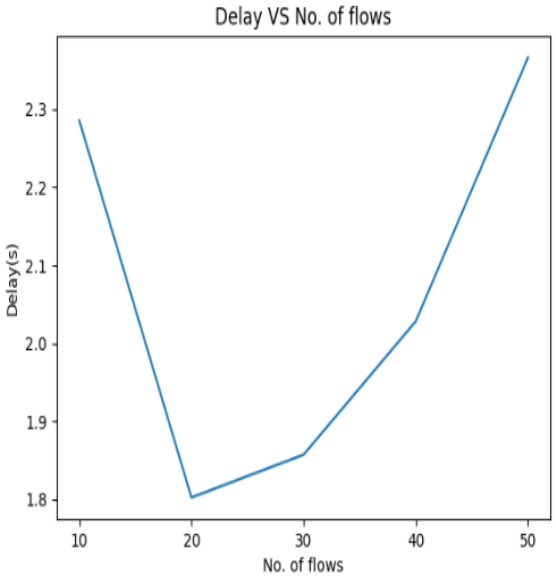


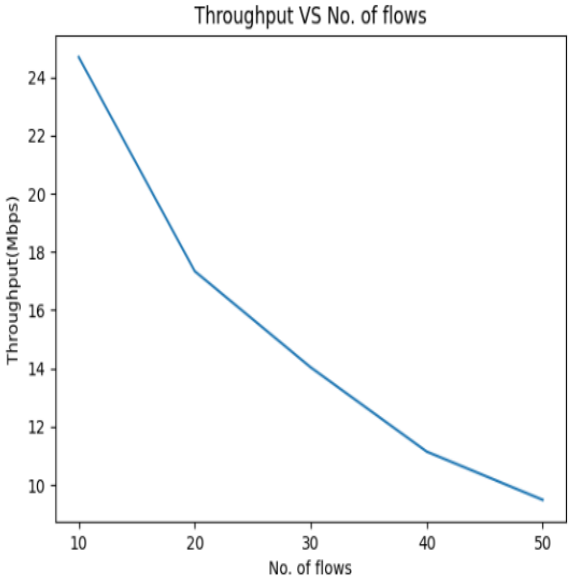


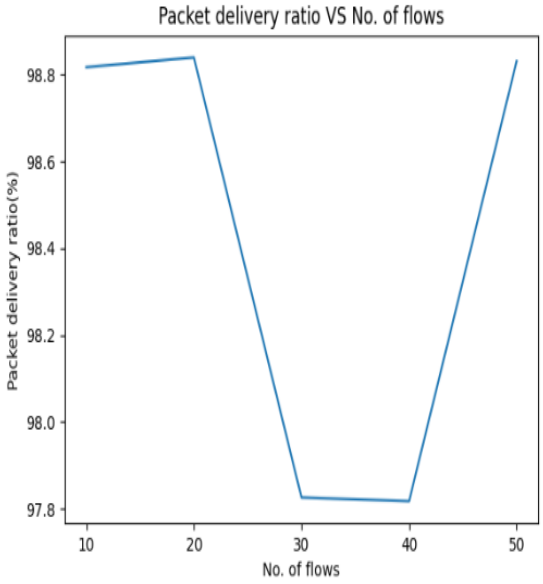


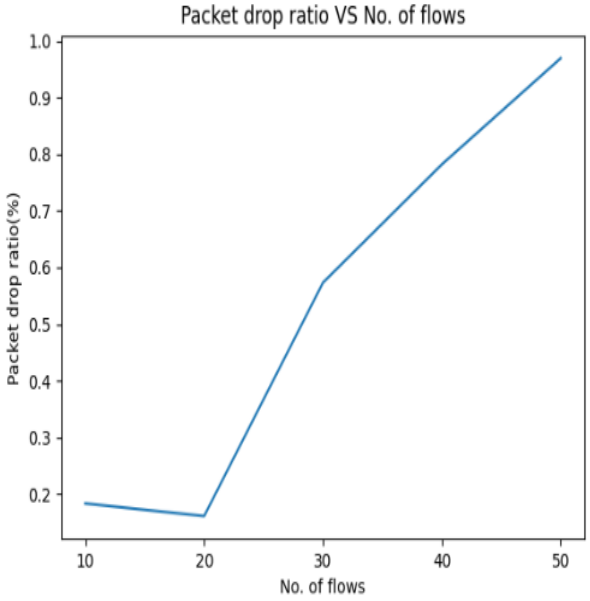


Varying nodes while keeping flow constant does not change much of the metrices and hence the number of flows were also increased proportionately with the number of nodes to get the attached results.

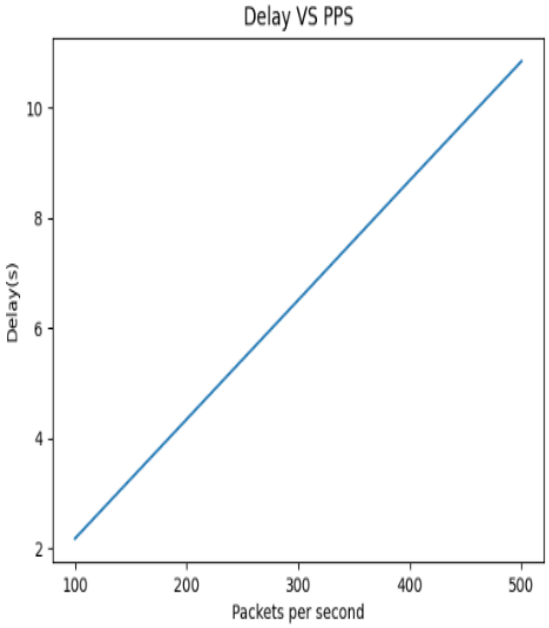


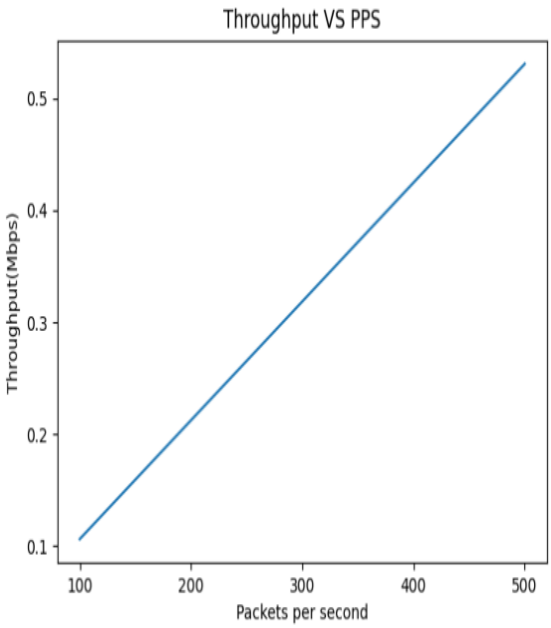


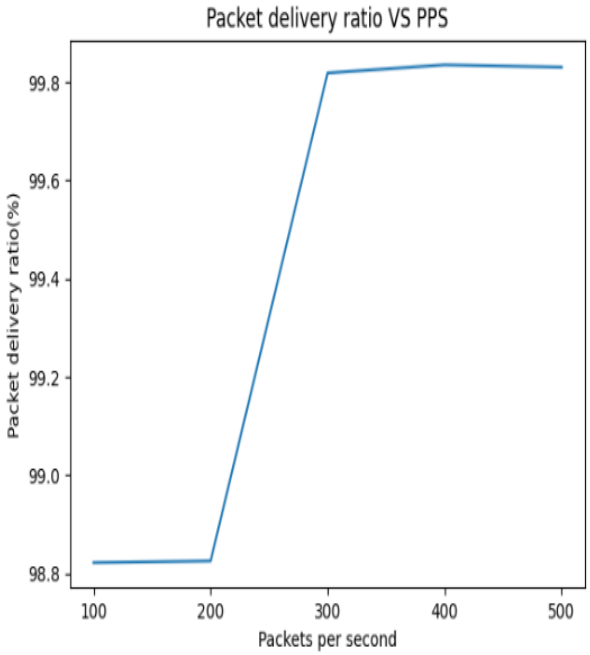


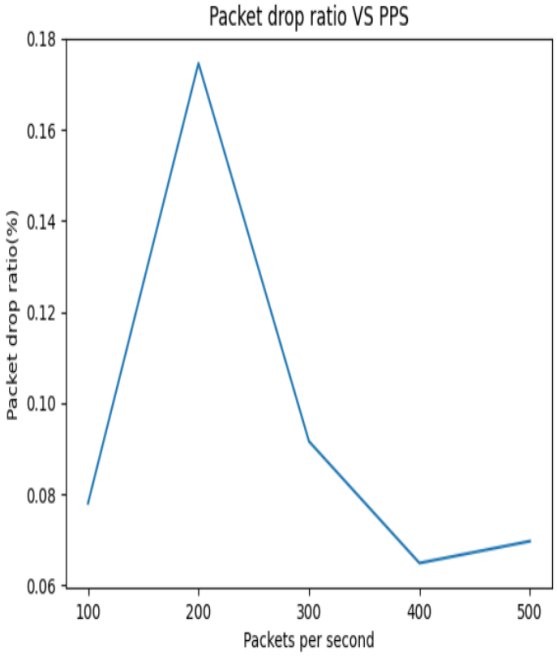


If number of flows is increased in a network while keeping other parameters fixed, the throughput of the network is bound to lessen as the network becomes congested then. This can exactly be seen from the graphs attached above.


Delay VS PPS
Delay(s)
Packets per second
10
8
6
4
2
100
200
300
400
500


Throughput VS PPS
Throughput(Mbps)
Packets per second
0.5
0.4
0.3
0.2
0.1
100
200
300
400
500


Packet delivery ratio VS PPS
Packet delivery ratio(%)
Packets per second
99.8
99.6
99.4
99.2
99.0
98.8
100
200
300
400
500


Packet drop ratio VS PPS
Packet drop ratio(%)
Packets per second
0.18
0.16
0.14
0.12
0.10
0.08
0.06
100
200
300
400
500

When packets sent per second of the network is increased, the throughput increases as expected as more bytes of data is received. From the packet drop ratio and packet delivery ratio graphs, the minimal packet drop characteristic of wired networks can be verified.

Wireless Network (Low rate, Mobile):

Used topology:

- Low-rate wireless personal access network
- Nodes with random-walk mobility module installed
- Packet size 200Byte
- Nodes laid out in a row-first grid.

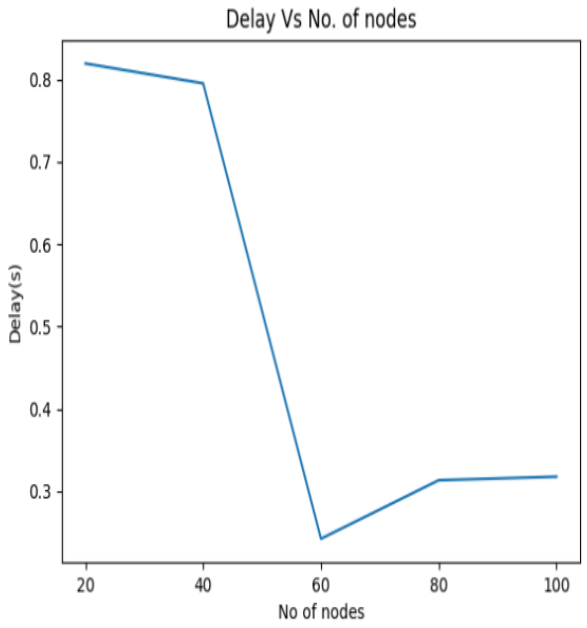
Varied Parameters:

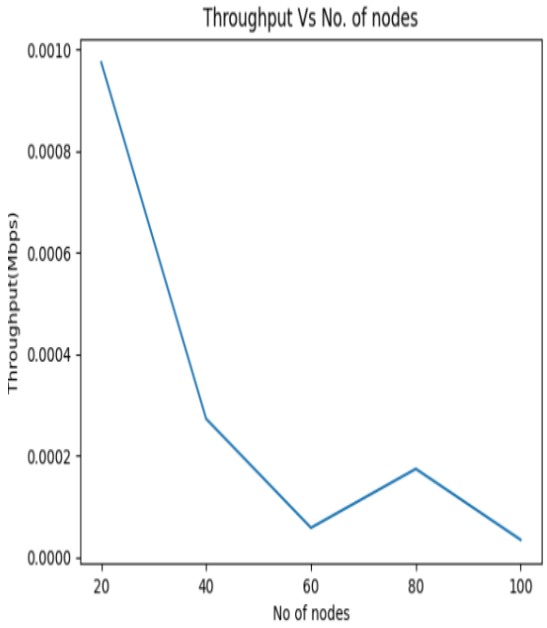
- No of Nodes
- No of Flows
- Packets per Second
- Speed of mobile nodes

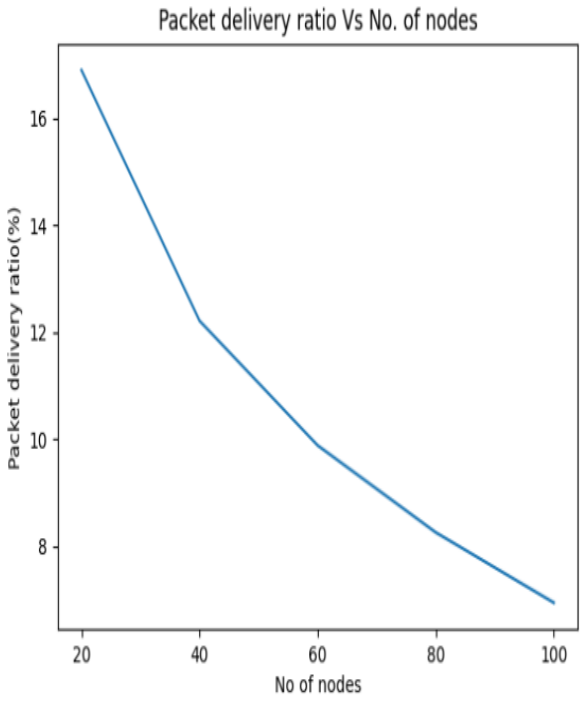
Calculated metrics:

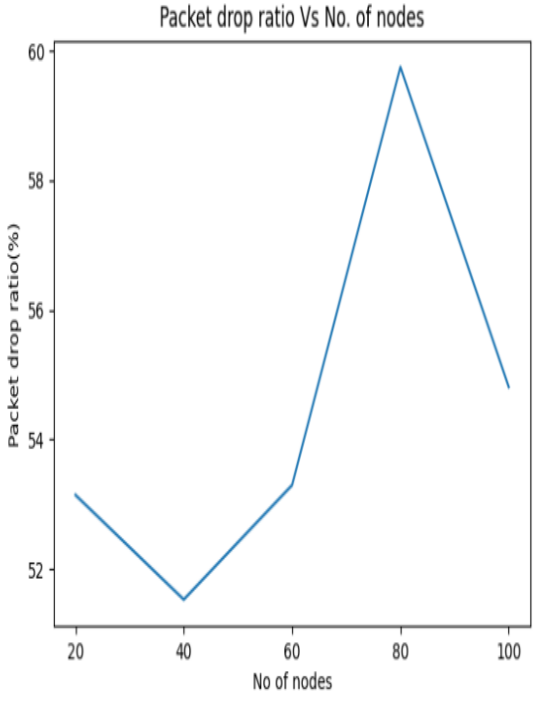
- Network Throughput
- End-to-end delay
- Packet delivery ratio
- Packet drop ratio

Generated Results:

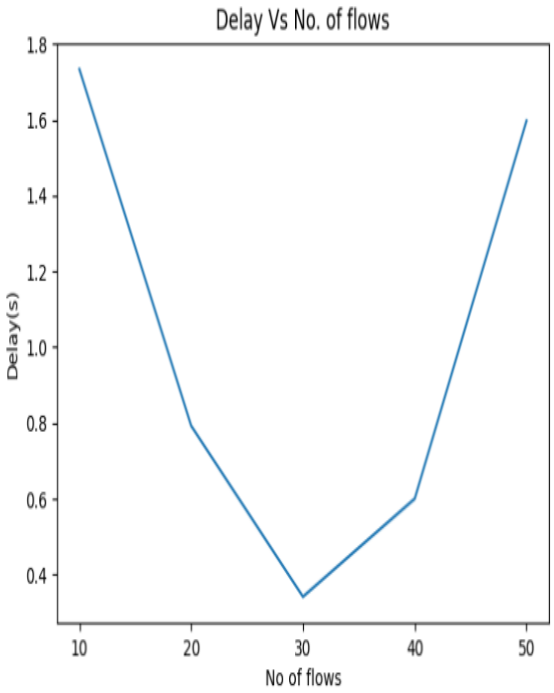


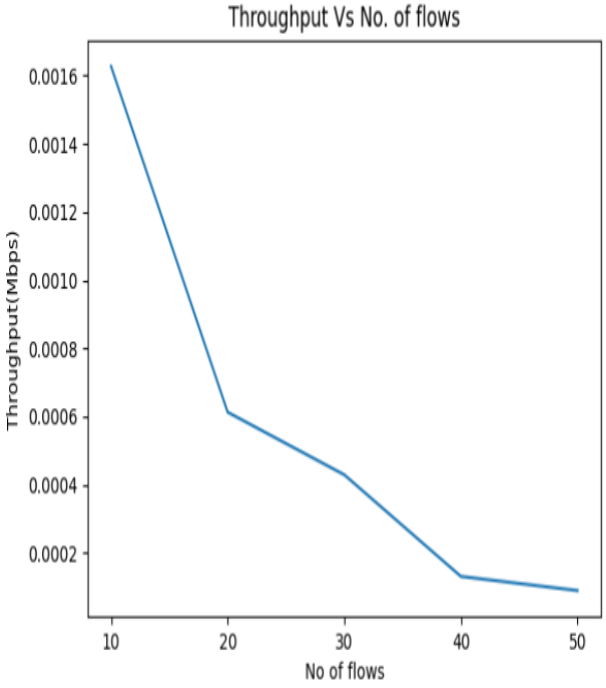


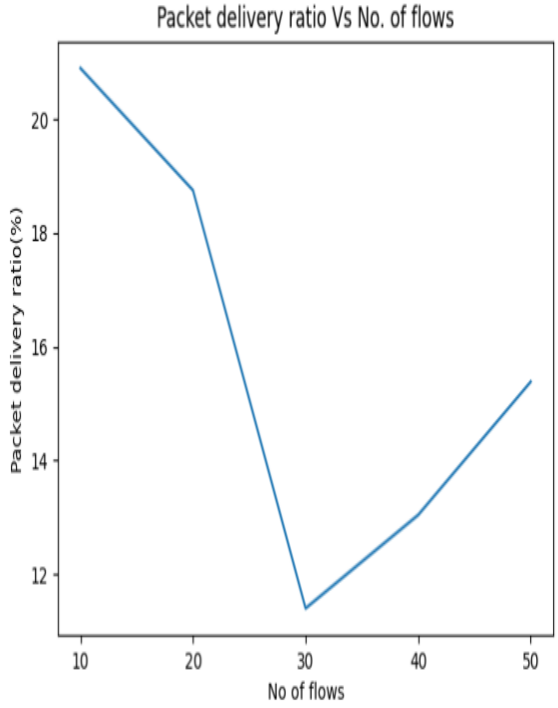


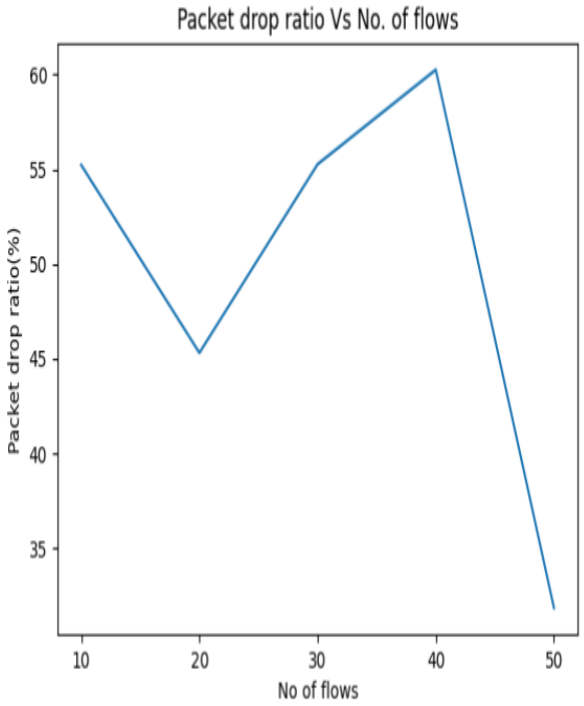


Throughput and packet delivery ratio drop as the number of nodes in the network increases.

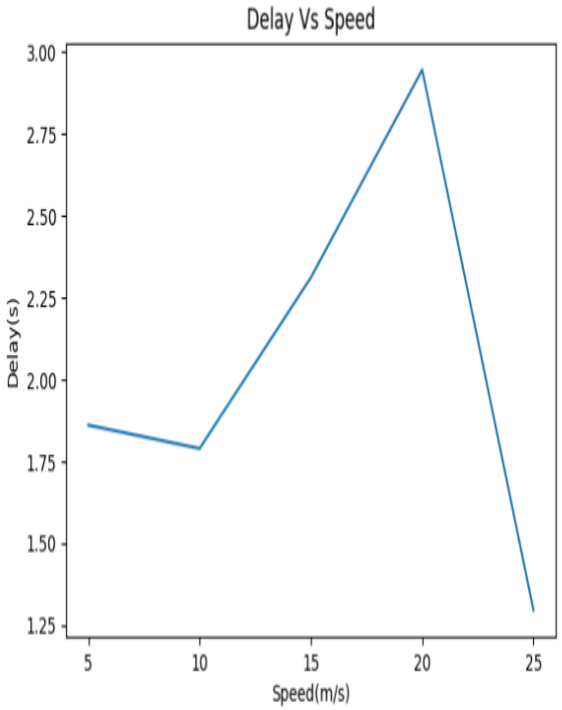


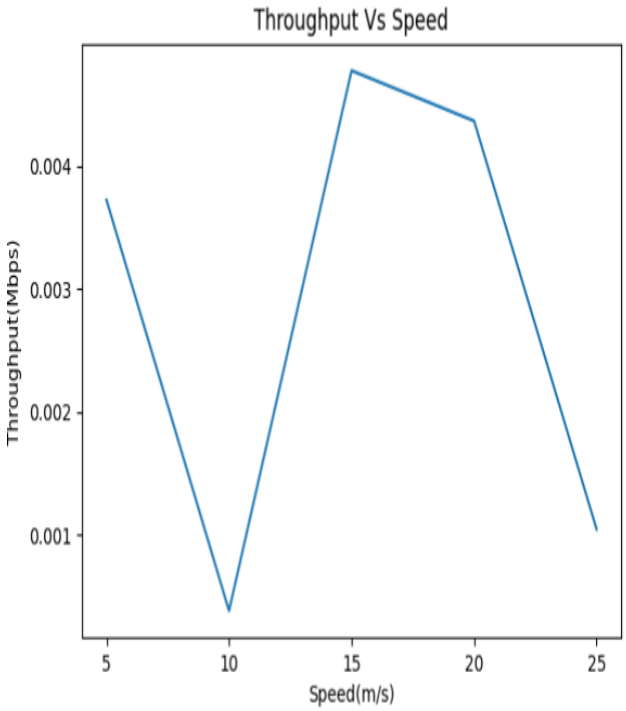


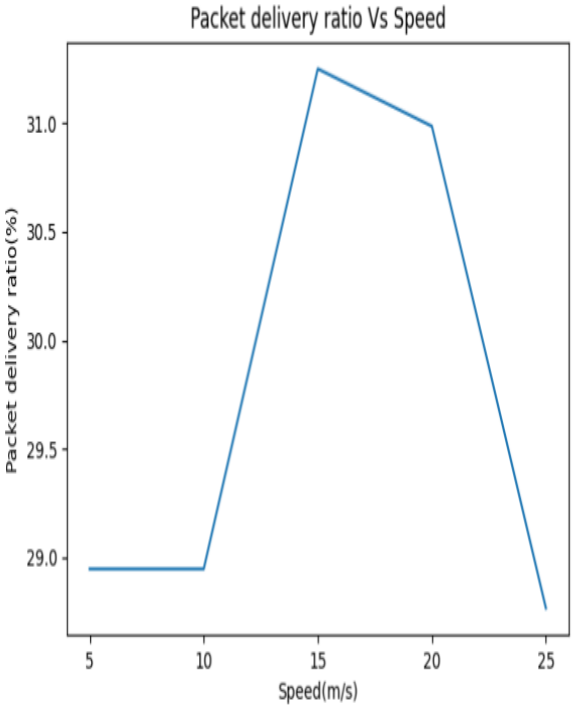


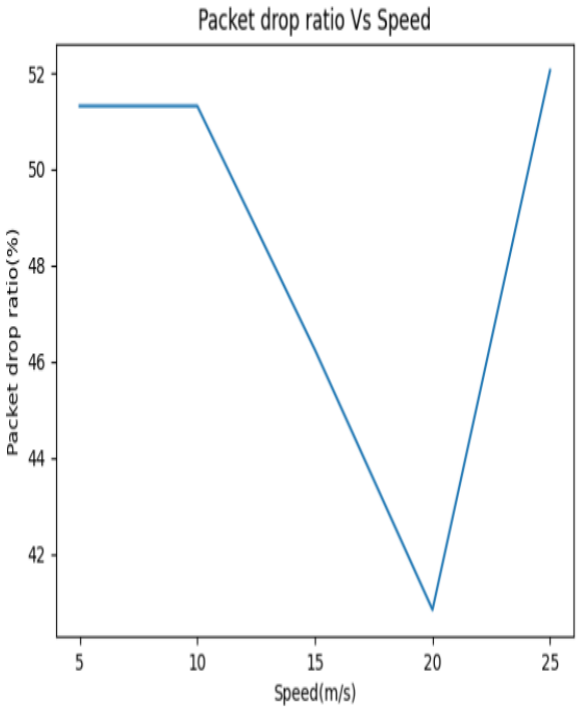


Throughput decreases as the number of flows increases in the network which should be the case because with increasing number of flows, amount of successfully received bytes decreases.

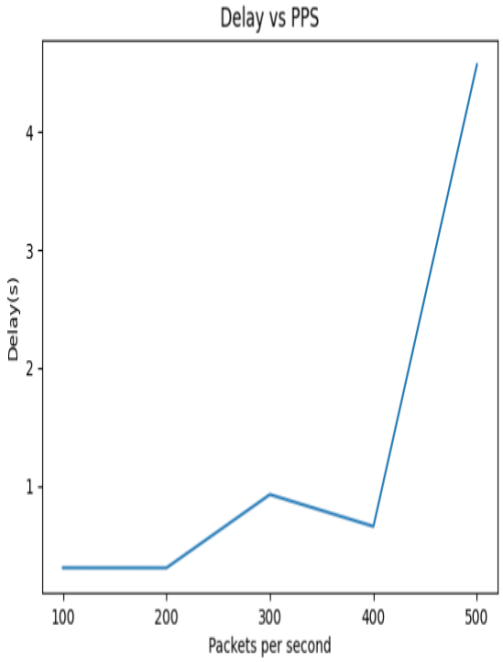


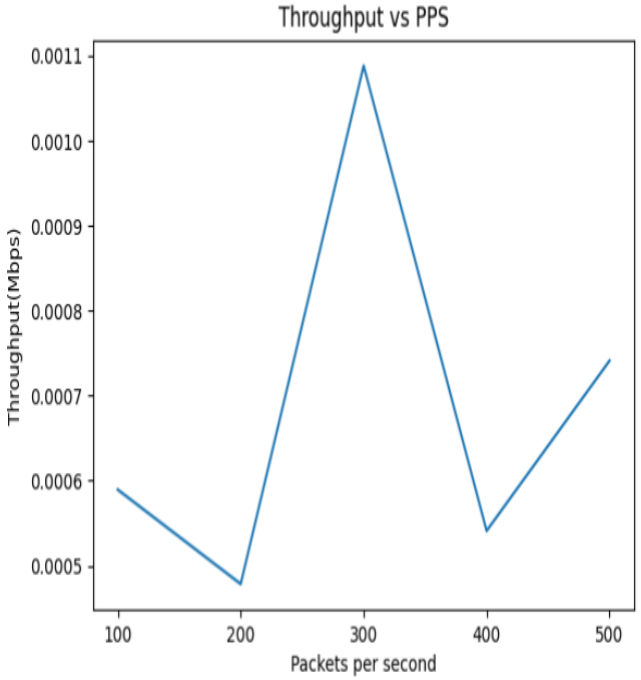


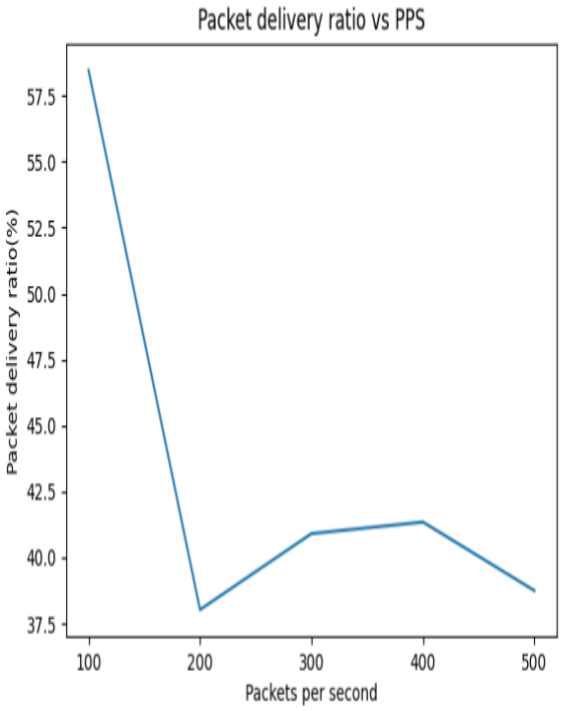


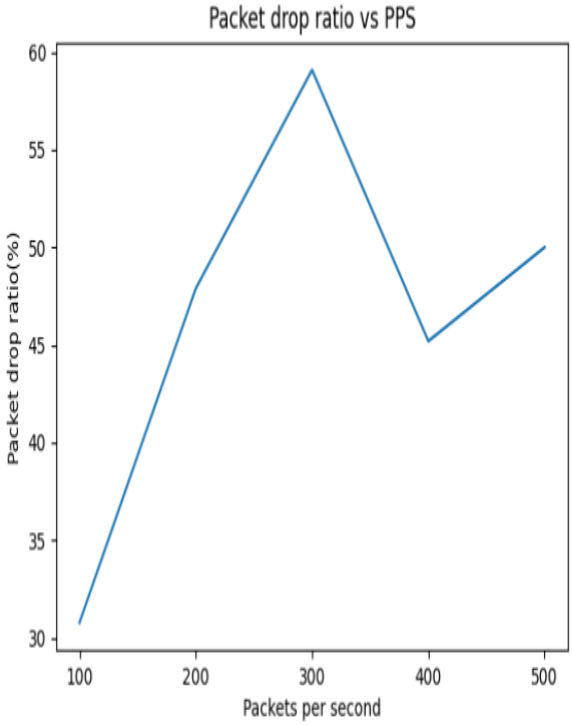


Because of high rate of packet loss, the graphs here don’t follow any certain pattern rather produce arbitrary outputs.









From the above generated graphs, the high packet drop rate of wireless networks can be verified. In case of my topology, packet drop ratio was greater than packet delivery ratio many a times. These frequent packet drops resulted in the not so systemic output of the generated graphs.

Task-B

Used Topology:

- PointTopoint dumbbell topology (as mentioned in the paper)
- Bottleneck datarate 500Mbps
- Access links datarate 1000Mbps

Varied Parameters:

- Random error rate
- Rtt
- Time

Calculated metrics:

- Rtt fairness
- Throughput

Overview of the proposed algorithm:

The proposed algorithm TCP-INVS mainly focuses on adapting to the path conditions while updating congestion window. It does so by incorporating a growth factor, k in the congestion window calculation part. K depends on estimated bandwidth and rtt of the path and as a result congestion window can cope with the path change efficiently. TCP-INVS also proposes a loss classification scheme based on buffer length to differentiate congestion-loss and non-congestion loss. Hence, TCP-INVS claims to perform better in lossy links and also claims to provide rtt fairness.

Modifications made in the simulator:

To implement the proposed algorithm, following changes were made in the simulator:

- A new variable in tcp socket-state class (m_cWndsp)
- A new class (tcp-invs.cc) and a new header file(tcp-invs.h) in the src/internet/model directory

```
protected:
  TracedValue<double>     m_currentBW;
  double                  m_lastSampleBW;
  double                  m_lastBW;
  double                  m_c;
  double                  m_beta;
  double                  m_gamma;
  Time                    m_last_rtt;
  double                  m_rtt_min;
  Time                    m_Minrtt;
  Time                    m_Maxrtt;
  double                  m_bwest;
  double                  m_buffest;
  double                  m_maxbuff;

  uint32_t                m_ackedSegments;
  bool                    m_IsCount;
  EventId                 m_bwEstimateEvent;
  Time                    m_lastAck;
  uint32_t                m_measure;
```

Fig: Variables defined in tcp-invs.h

```
void
TcpINVS::IncreaseWindow (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked)
{
  //std::cout<<"inval"<<tcb->m_cWndsp<<std::endl;
  //std::cout<<"cwnvallll"<<tcb->m_cWnd<<std::endl;
  uint32_t k=Update();
  if(tcb->m_cWnd<tcb->m_ssThresh) {
    tcb->m_cWnd+=tcb->m_segmentSize;
    //tcb->m_cWnd+=1;
  }
  else if(tcb->m_cWnd>tcb->m_ssThresh && tcb->m_cWnd<tcb->m_cWndsp) {
    //std::cout<<"firstif"<<std::endl;
    uint32_t val = ((tcb->m_cWndsp-tcb->m_cWnd)/(k*tcb->m_cWndsp))*tcb->m_segmentSize;
    //uint32_t val = ((tcb->m_cWndsp-tcb->m_cWnd)/(k*tcb->m_cWndsp));
    tcb->m_cWnd+=val;
  }
  else {
    //std::cout<<"secondif"<<std::endl;
    //std::cout<<k<<std::endl;
    //std::cout<<tcb->m_cWndsp<<std::endl;
    uint32_t val = ((tcb->m_cWnd-tcb->m_cWndsp)/(k*tcb->m_cWnd))*tcb->m_segmentSize;
    //uint32_t val = ((tcb->m_cWnd-tcb->m_cWndsp)/(k*tcb->m_cWnd));
    tcb->m_cWnd+=val;
  }
}
```

```
void
TcpINVS::EstimateRTT(const Time &rtt)
{
  m_Minrtt = std::min (m_Minrtt, rtt);
  m_Maxrtt = std:: max(m_Maxrtt,rtt);
}

uint32_t TcpINVS::Update() {
    //std::cout<<"cccc"<<m_c<<std::endl;
    //std::cout<<"bww"<<m_bwest<<std::endl;
    //std::cout<<"rtttt"<<m_rtt_min<<std::endl;
    //std::cout<<"gammm"<<m_gamma<<std::endl;
    return m_c*(log2(m_bwest*pow(m_rtt_min,m_gamma))+1);
     //return 1;
}
```

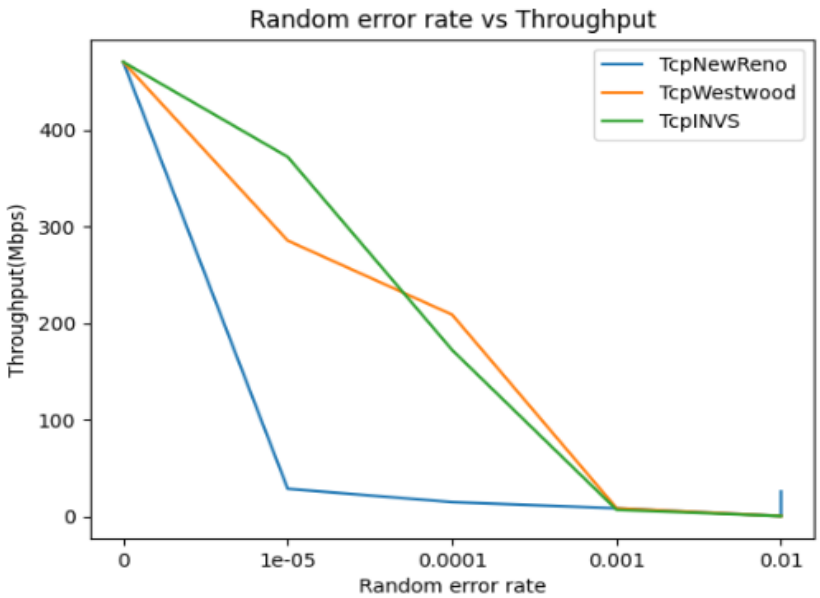
```
void
TcpINVS::PktsAcked (Ptr<TcpSocketState> tcb, uint32_t packetsAcked,
                    const Time& rtt)
{
  NS_LOG_FUNCTION (this << tcb << packetsAcked << rtt);

  if (rtt.IsZero ())
    {
      NS_LOG_WARN ("RTT measured is zero!");
      return;
    }
  m_last_rtt=rtt;
  m_ackedSegments += packetsAcked;
  EstimateBW (rtt, tcb);
  EstimateRTT(rtt);
  double val=500.0/(m_currentBW+1);
  //std::cout<<val<<std::endl;
  if(val>1) m_bwest=val;
  //m_bwest=std::max((500.0/m_currentBW),1.0);
  else m_bwest=1;
  m_rtt_min=std::max((0.5/m_Minrtt.GetSeconds()),1.0);

}
```

Fig: Functions implemented in tcp-invs.cc

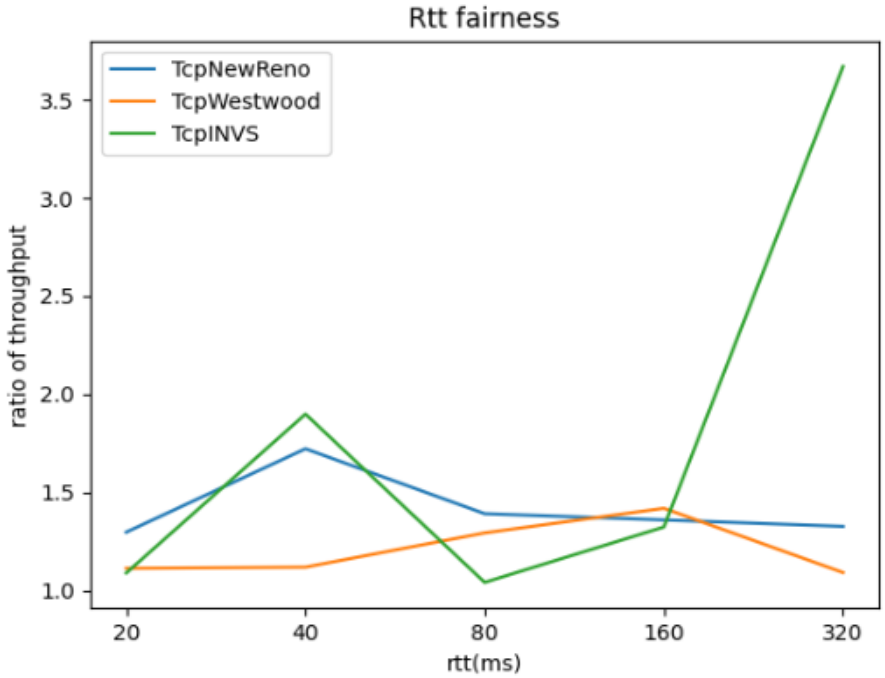
```
void
TcpINVS::EstimateBW (const Time &rtt, Ptr<TcpSocketState> tcb)
{
  uint32_t delta=Simulator::Now().GetSeconds()-m_measure;
  NS_LOG_FUNCTION (this);

  NS_ASSERT (!rtt.IsZero ());
  if(delta>std::max(rtt.GetSeconds(),20000.0)) {
  //m_currentBW = m_ackedSegments * tcb->m_segmentSize / rtt.GetSeconds ();
  m_currentBW=1*m_currentBW+1*(m_ackedSegments * tcb->m_segmentSize)/delta;
  m_ackedSegments = 0;
  }
}

uint32_t
TcpINVS::GetSsThresh (Ptr<const TcpSocketState> tcb,
                      uint32_t bytesInFlight)
{

  m_maxbuff=m_Maxrtt.GetSeconds()*m_currentBW;
  double val=std::min(5.0,m_maxbuff);
  if(val<((m_last_rtt.GetSeconds()-m_Minrtt.GetSeconds())*m_currentBW)) return tcb->m_cWnd;

  return (0.5 * tcb->m_cWnd);
}
```

Fig: Functions implemented in tcp-invs.cc

Generated Results:



As the paper claims, I got almost similar results when computing throughput of a single lossy link. TCPINVS outperforms TCPWestwood and TCPNewReno in some cases which validates the algorithm's claim to work better in lossy conditions. To generate the above graph, the simulation criteria mentioned in the paper was strictly followed.



Rtt fairness of the three congestion control algorithms were calculated by considering two competing flows. The rtt of the first flow was kept fixed at 80ms while the rtt of the second flow was varied from 20-320ms. When both the flows had same rtt(80ms), TcpINVS provided a throughput ratio of almost 1. Rtt was approximated using delay in the path.

To simulate the above graph, two new files were added in the simulator(p2p.cc and p2p.h).To generate two flows having different delay in the same dumbbell topology, I implemented a custom dumbbell helper that takes two different channels with different delays in the constructor and links half of the nodes in the topology with each of the channels. As a result, the resultant topology will have two types of links each type having different channel attributes(in this case, only the delay was varied).

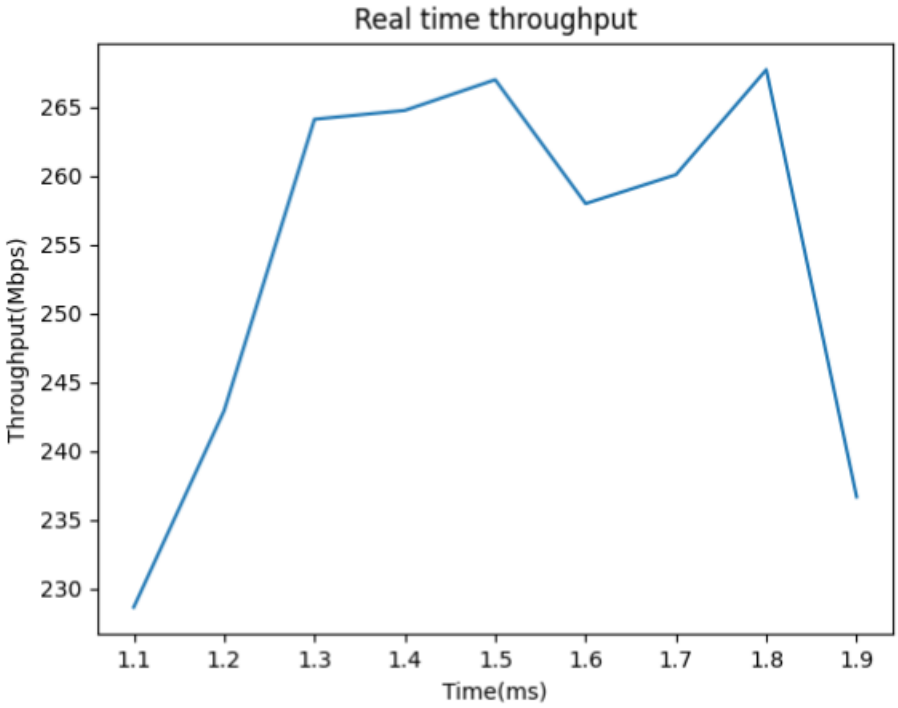


Fig: Real-time throughput of an INVS flow

Summary and Findings:

Although the paper claims that TcpINVS always outperforms TCP Westwood in case of lossy links, my simulation results don't reflect the same. In case of RTT fairness , INVS does moderately well but cannot ensure fairness when difference in RTT is significantly large. Hence , based on my simulation results, I would say, TcpINVS has a reasonable efficiency but is not efficient enough to replace the existing congestion control algorithms in the simulator(NS3).