TcpChd: Improved coexistence and loss tolerance for delay based TCP congestion control
A report on CSE322 Term Project

Towhidul Islam (1705017)
Supervised By
Md. Toufikuzzaman
Lecturer

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

February 2022

1 Introduction

In this project we have done the following tasks:

- In Task A, we measure throughput, end to end delay. packet delivery and drop ratio for an experimental topology using the existing congestion control algorithm in ns3.35.
- In Task B, we implement our new congestion control algorithm and compare the results with an existing one.

2 Experimental Topologies

We used a total of three kinds of topology. One of them is wired topology, one is mesh for wifi low rate 802.15.4 devices and the other is hybrid (Wifi high rate 802.11 devices with Csma devives).

2.1 Topology Diagrams

- **Wired**

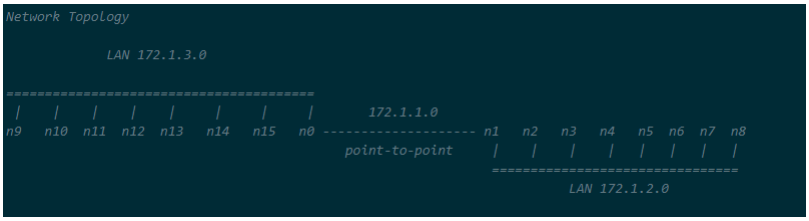

Figure 1: Fully Wired Topology

- **Mesh**

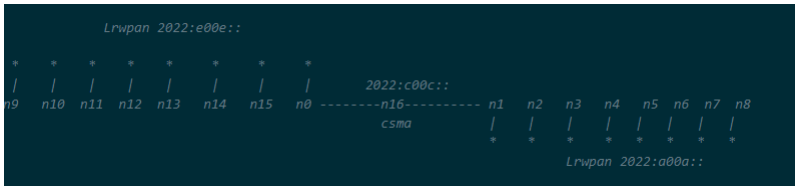

Figure 2: Mesh Topology for Wireless low rate(mobile)

- **Hybrid**

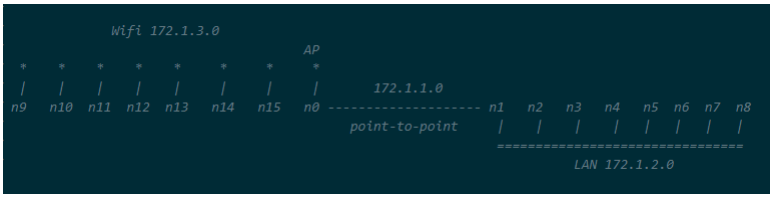

Figure 3: Hybrid Topology (Wireless high rate with Csma)

3 Parameters Under Variation

In this task, we determined five metrics

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

We varied total four different kinds of parameters while determining those metrics

- Number of nodes(20, 40, 60, 80, 100)
- Number of flows(10, 20, 30, 40, 50)
- Speed of nodes (5 m/s, 10 m/s, 15 m/s, 20 m/s, and 25 m/s) [Only in case of having mobility]
- Number of packets per second(100,200,300,400,500)

We determined these metrics except goodput using flowmonitor. We calculated goodput in application layer. Since flowmonitor gives transport layer information, all these metrics except goodput was calculated in transport layer.

4 Overview of the proposed algorithm

TcpChd is a delay based algorithm which extends work by Budzisz et al. to provide tolerance to non-congestion related losses, and better coexistence with loss-based TCP in lightly multiplexed environments.

It improves the throughput when there are 1% packet losses by about 150%, and gives more than 50% improvement in the ability to share capacity with NewReno in lightly multiplexed environments.

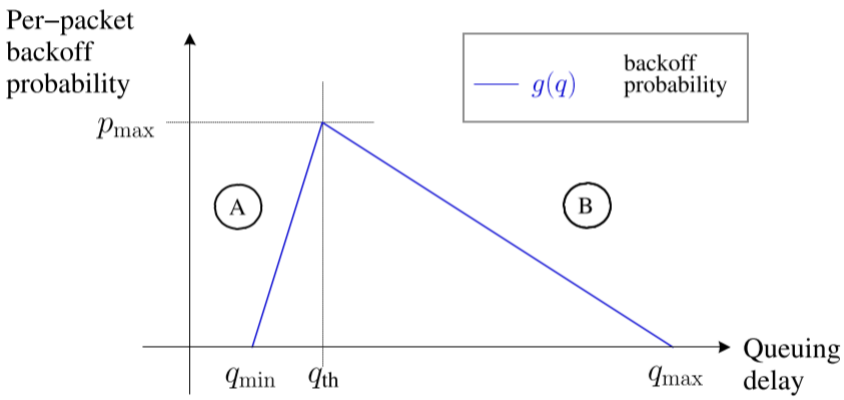

Figure 4: Per-packet backoff probability as a function estimated queueing delay

•

TcpChd uses a probability backoff funtion. Using a threshold queuing delay it divides congestion avoidance into two different region A and B.

In congestion avoidance, the windows changes as follows:

when $X > g(q)$:

$$w_{i+1} = \frac{w_i}{2} \tag{1}$$

else:

$$w_{i+1} = w_i + \frac{1}{w_i} \tag{2}$$

When a packet loss occurs, TcpChd checks if it is in region A or in region B. If it is in region A, TcpChd doesn't consider the packet loss as congestion related loss, that's why it doesn't change windows. But if the region is B which is determined by the backoff probability, it updates the window size as follows:

$$w_{i+1} = \frac{max(w_i, s_i)}{2} \tag{3}$$

where s_i is the TcpNewReno congestion window.

5 Modifications made in the simulator

First of all, we needed to one cc and one header file for our new algorithm.

- tcp-chd.cc
- tcp-chd.h

To add these two files in the simulator, we also needed to include them in wscript file(src>internet) and in internet-module.h(build>ns3) **The modified functions are:**

- **CongestionAvoidance**
- **dupAck**(in tcp-socket-base.cc)

The newly added functions are:

- **CalcBackoffProbability**
- **PktsAcked**
- **PckLossWindowUpdate**
- **CongestionAvoidanceChd**
- **CongestionAvoidanceNewReno**

5.1 Code Snippets:

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

  if (tcb->m_cWnd < tcb->m_ssThresh)
    {
      segmentsAcked = SlowStart (tcb, segmentsAcked);
    }

  if (tcb->m_cWnd >= tcb->m_ssThresh)
    {

    if(m_isPckLoss==0){
      if(m_qDelay<=m_minQD){
        uint32_t wind = CongestionAvoidanceNewReno (tcb, segmentsAcked);
        tcb->m_cWnd = wind;
      }
      else{
        uint32_t wind = CongestionAvoidanceChd (tcb, segmentsAcked);
        tcb->m_cWnd = wind;

        uint32_t Swind = CongestionAvoidanceNewReno (tcb, segmentsAcked);
        m_cWndNR = std::max(wind, Swind);
      }
    }
    else{
      m_isPckLoss = 0;
    }

    }
}
```

Figure 5: Increasewindow

```
if ((m_dupAckCount == m_retxThresh) && ((m_highRxAckMark >= m_recover) || (!m_recoverActive)))
  {

    if(m_congestionControl->GetName() == "TcpChd"){
      m_congestionControl->PckLossWindowUpdate(m_tcb);
    }

    EnterRecovery (currentDelivered);
    NS_ASSERT (m_tcb->m_congState == TcpSocketState::CA_RECOVERY);
  }
```

Figure 6: dupAck in Tcp-socket-base.cc

```
void
TcpChd::PckLossWindowUpdate(Ptr<TcpSocketState> tcb)
{
  if(m_qDelay > m_threshQD){
    uint32_t wind = static_cast<uint32_t>(tcb->m_cWnd);
    wind = std::max(m_cWndNR, wind);
    tcb->m_cWnd = static_cast<uint32_t> (wind/2);
  }
}
```

Figure 7: PckLossWindowUpdate

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

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

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

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

  ++m_RttCount;
  NS_LOG_DEBUG ("Updated m_cntRtt = " << m_RttCount);

  m_sumRtt += rtt;
  NS_LOG_DEBUG ("Updated m_sumRtt = " << m_sumRtt);

  m_qDelay = m_sumRtt/m_RttCount - m_minRtt;
  NS_LOG_DEBUG ("Updated m_qDelay = " << m_maxQDelay);

  m_maxQDelay = m_maxRtt - m_maxRtt;
  NS_LOG_DEBUG ("Updated m_maxQDelay = " << m_maxQDelay);

}
```

Figure 8: PktsAcked

```
uint32_t
TcpChd::CongestionAvoidanceNewReno (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked)
{
  NS_LOG_FUNCTION (this << tcb << segmentsAcked);

  if (segmentsAcked > 0)
    {
      double adder = static_cast<double> (tcb->m_segmentSize * tcb->m_segmentSize) / m_cWndNR;
      adder = std::max (1.0, adder);
      m_cWndNR += static_cast<uint32_t> (adder);
      NS_LOG_INFO ("In New Reno CongAvoid, updated to cwnd " << tcb->m_cWnd <<
                   " ssthresh " << tcb->m_ssThresh);

    return m_cWndNR;
    }

  return 0;
}
```

Figure 9: CongestionAvoidanceNewReno

```
uint32_t
TcpChd::CongestionAvoidanceChd (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked)
{
  NS_LOG_FUNCTION (this << tcb << segmentsAcked);

  if (segmentsAcked > 0)
    {
    double pb = CalcBackoffProbability();

    double x = (std::rand()%10)/10;

    if(x<pb){
      return (static_cast<uint32_t>(tcb->m_cWnd/2));
    }
    else {
      return (tcb->m_cWnd+static_cast<uint32_t>(1/tcb->m_cWnd));
    }

    }

  return 0;
}
```

Figure 10: CongestionAvoidanceChd

```
double
TcpChd::CalcBackoffProbability() const
{

  double qDelayVal = static_cast<double> (m_qDelay.GetMilliSeconds());
  double maxQDelayVal = static_cast<double> (m_maxQDelay.GetMilliSeconds());
  double minQDelayVal = static_cast<double> (m_minQD.GetMilliSeconds());
  double threshVal = static_cast<double> (m_threshQD.GetMilliSeconds());

  if(qDelayVal==m_threshQD.GetMilliSeconds()){
    return m_maxBP;
  }
  else if(qDelayVal<m_threshQD.GetMilliSeconds()){
    double slope = m_maxBP/(threshVal-minQDelayVal);
    double c = (-1)*(slope*minQDelayVal);

    double prob = slope*qDelayVal + c;

    return prob;
  }
  else {
    double slope = (-1)*(m_maxBP/(maxQDelayVal-threshVal));
    double c = (-1)*(slope*maxQDelayVal);

    double prob = slope*qDelayVal + c;

    return prob;
  }
}
```

Figure 11: CalcBackoffProbability

```
private:
  Time m_minRtt;
  Time m_maxRtt;
  uint32_t m_RttCount;
  Time m_sumRtt;
  Time m_minQD;
  Time m_threshQD;
  double m_maxBP;
  uint32_t m_cWndNR;
  Time m_qDelay;
  Time m_maxQDelay;
  uint32_t m_isPckLoss;
};
```

Figure 12: All attributes

```
TcpChd::TcpChd (void)
: TcpNewReno (),
    m_minRtt (Time::Max ()),
    m_maxRtt (Time::Min ()),
  m_RttCount (0),
  m_sumRtt (Time(0)),
  m_minQD (Time(0.005)),
  m_threshQD (Time(0.030)),
  m_maxBP (0.25),
  m_cWndNR (1),
  m_qDelay (Time(0)),
  m_maxQDelay (Time(0)),
  m_isPckLoss (0)
{
  NS_LOG_FUNCTION (this);
}

TcpChd::TcpChd (const TcpChd& sock)
  : TcpNewReno (sock),
    m_minRtt (sock.m_minRtt),
    m_maxRtt (sock.m_maxRtt),
  m_RttCount (sock.m_RttCount),
  m_minQD (sock.m_minQD),
  m_threshQD (sock.m_threshQD),
  m_maxBP (sock.m_maxBP),
  m_cWndNR (sock.m_cWndNR),
  m_qDelay (sock.m_qDelay),
  m_maxQDelay (sock.m_maxQDelay),
  m_isPckLoss (sock.m_isPckLoss)
{
  NS_LOG_FUNCTION (this);
}
```

Figure 13: Attribute Initialization

6 Results with graphs

6.1 Task A(Wired)

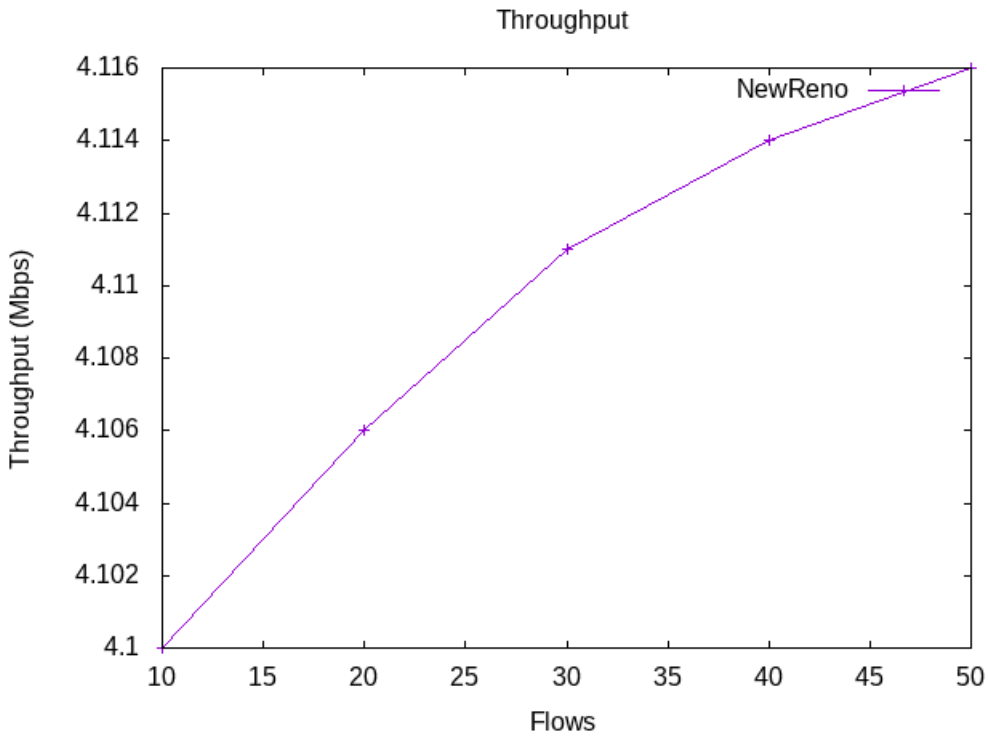

Figure 14: Throughput(varying flow)

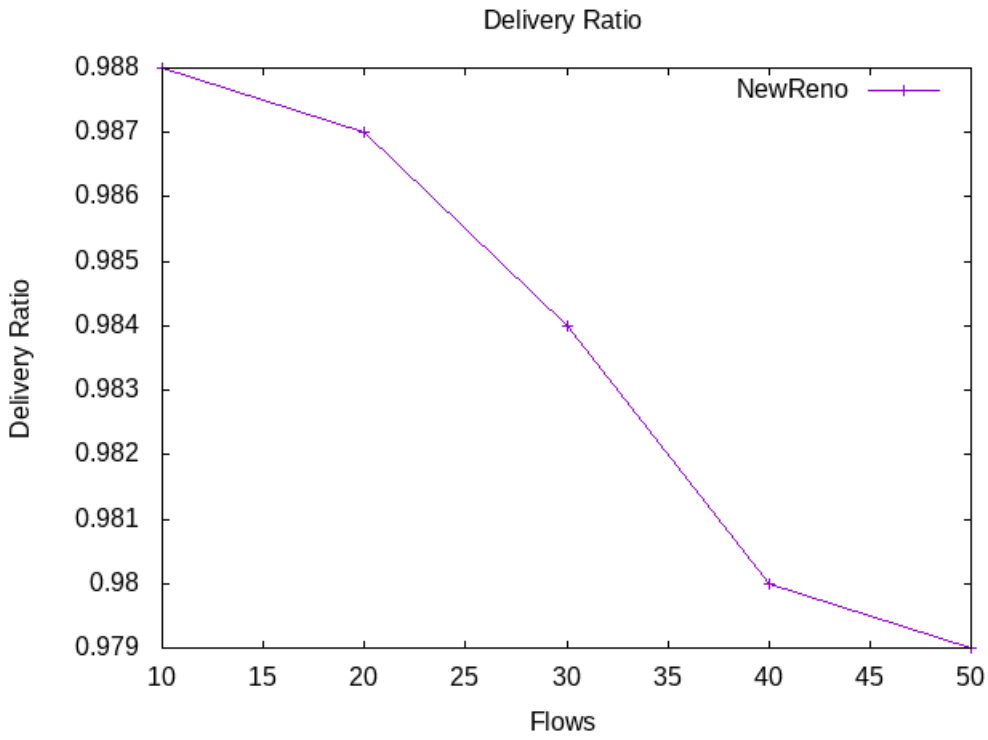

Figure 15: Delivery Ratio(varying flow)

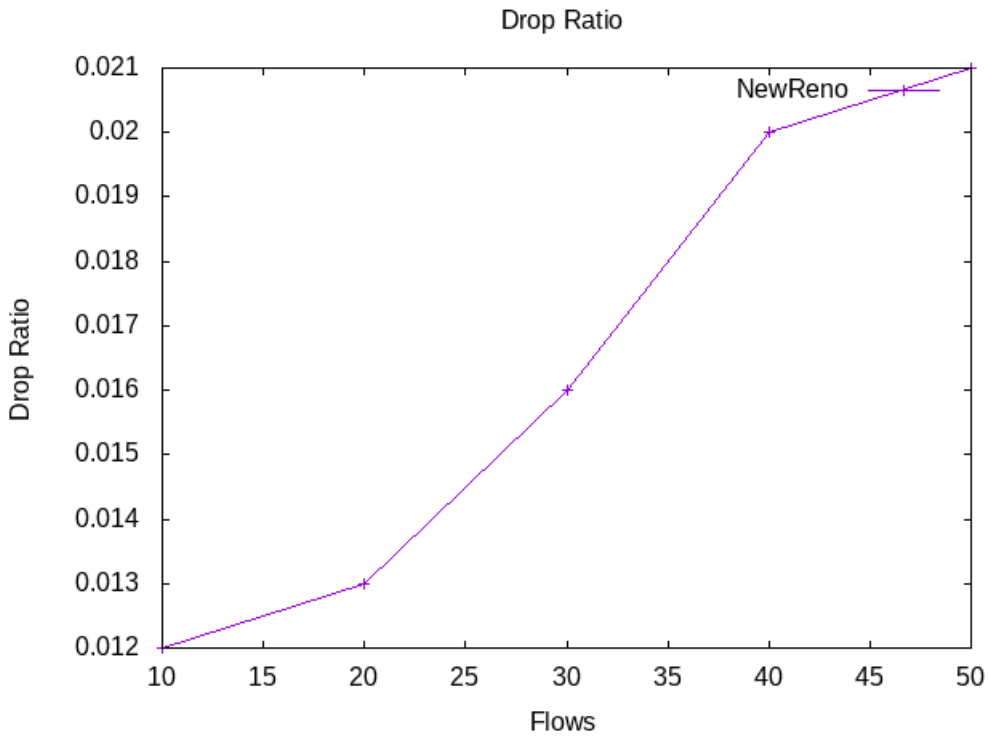

Figure 16: Drop Ratio(varying flow)

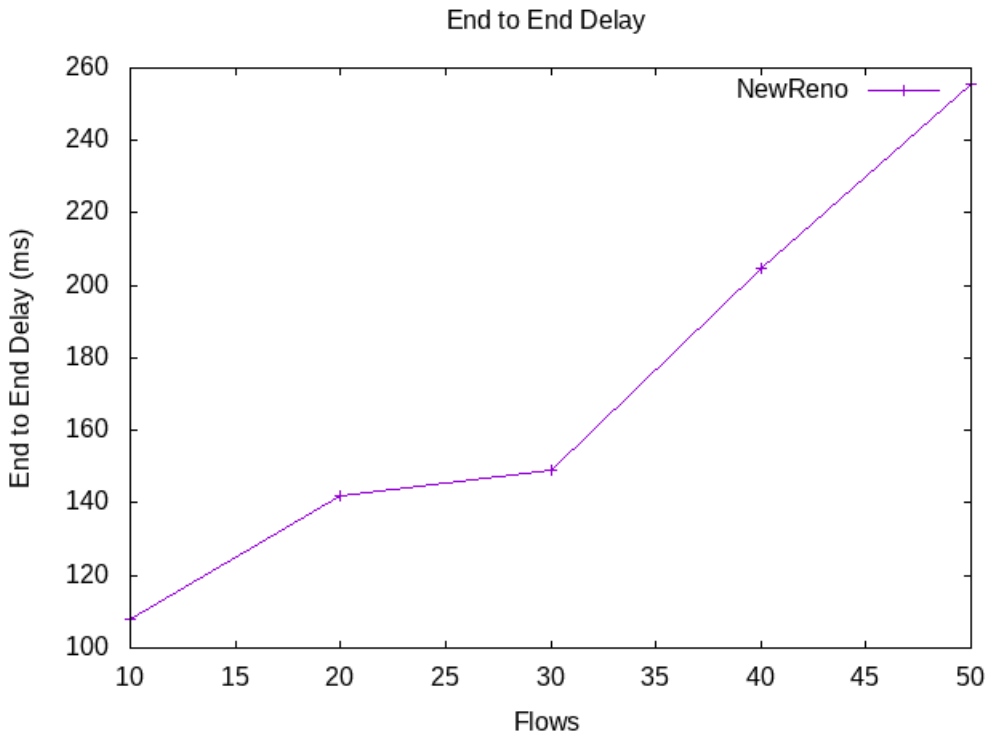

Figure 17: End to end delay(varying flow)

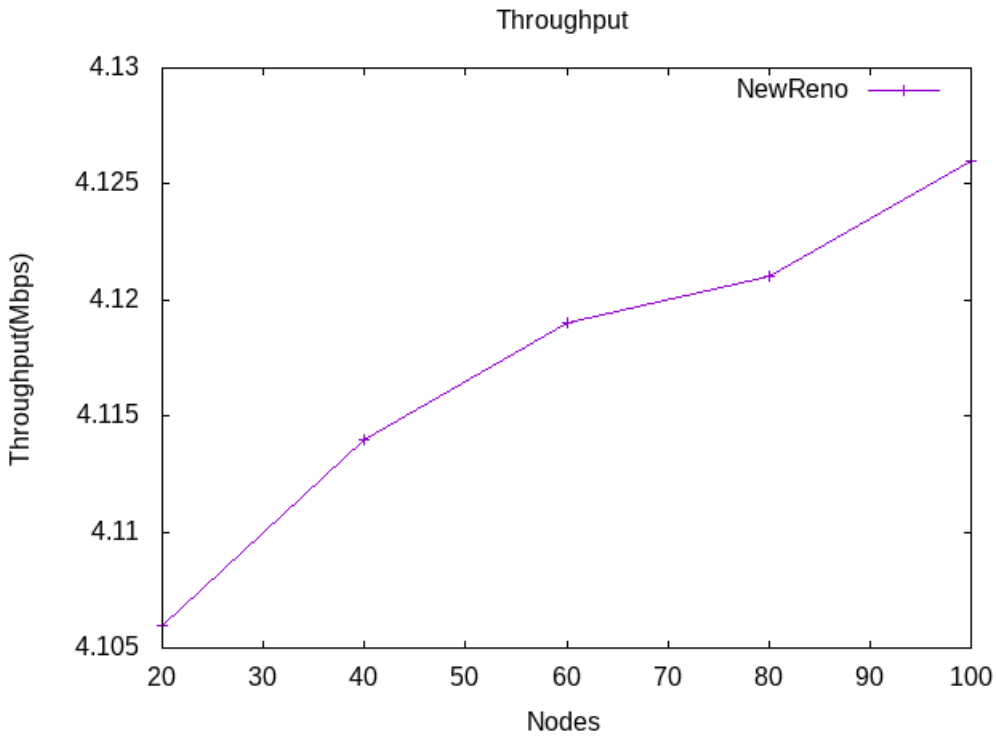

Figure 18: Throughput(Varying Node)

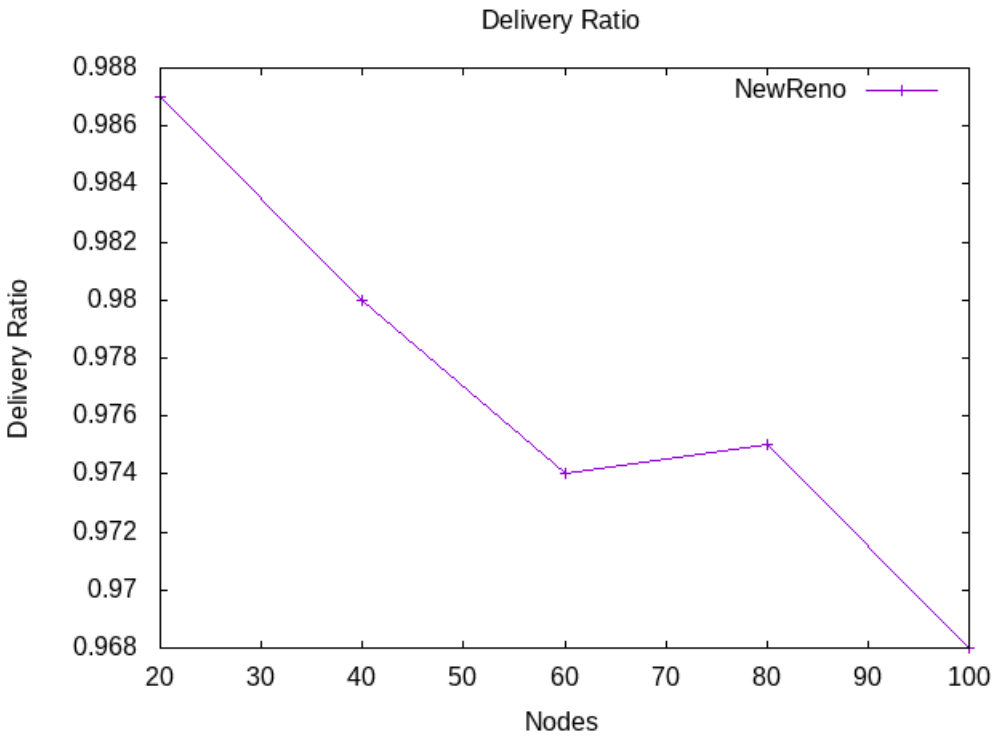

Figure 19: Delivery Ratio(Varying Node)

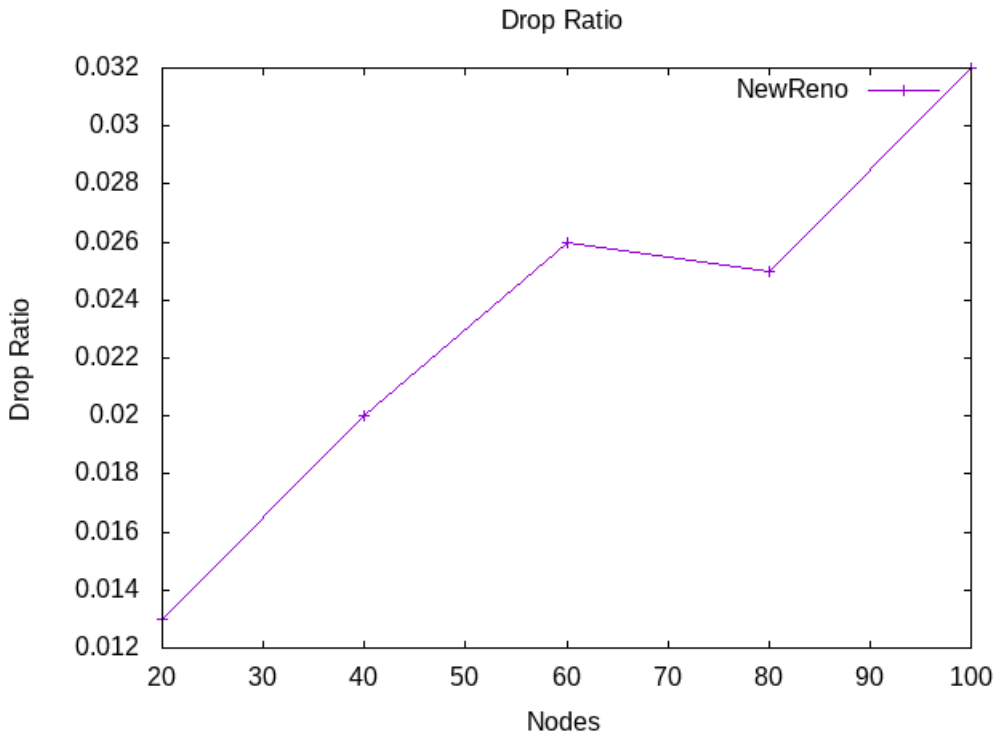

Figure 20: Drop Ratio(Varying Node)

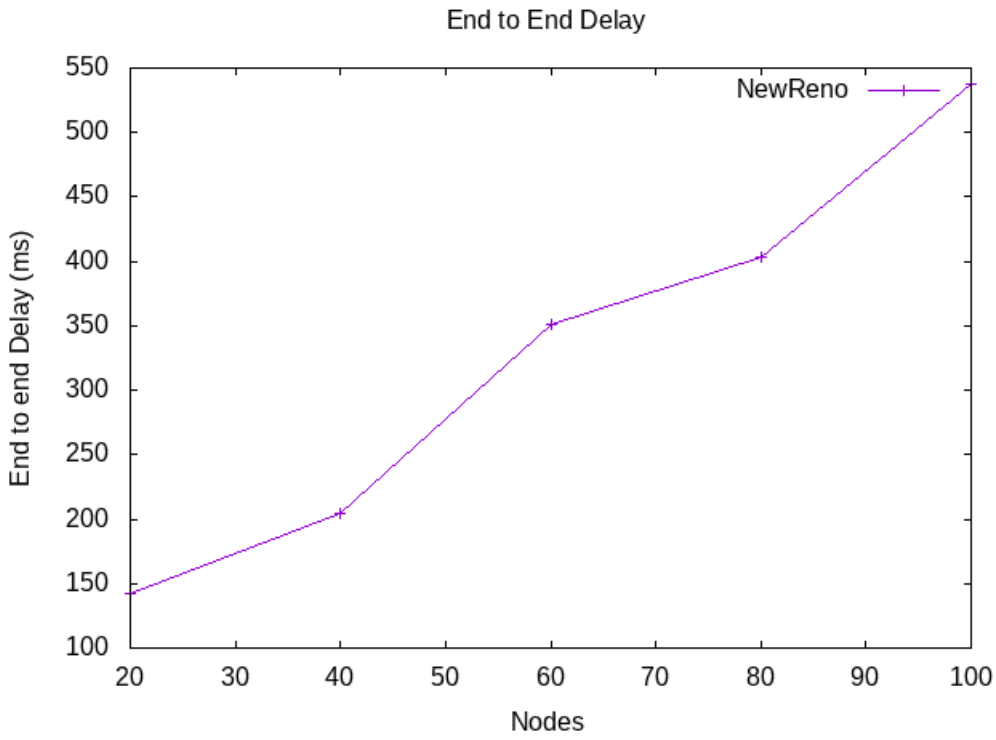

Figure 21: End to end delay(Varying Node)

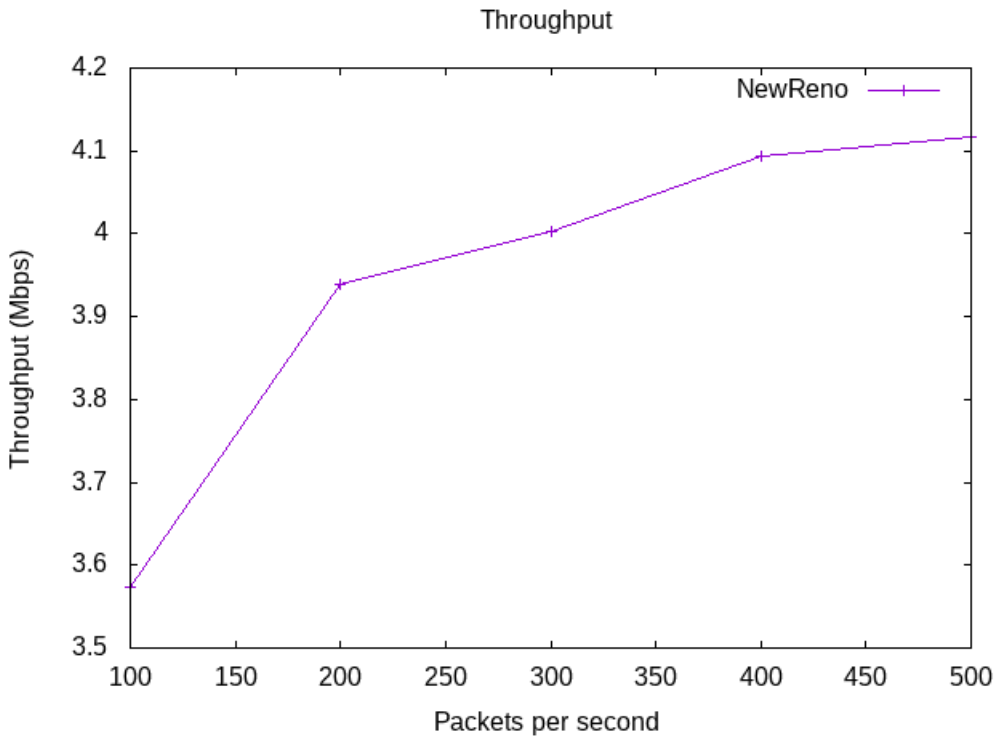

Figure 22: Throughput(Varying Packet per sec)

Figure 23: Delivery Ratio(Varying Packet per sec)

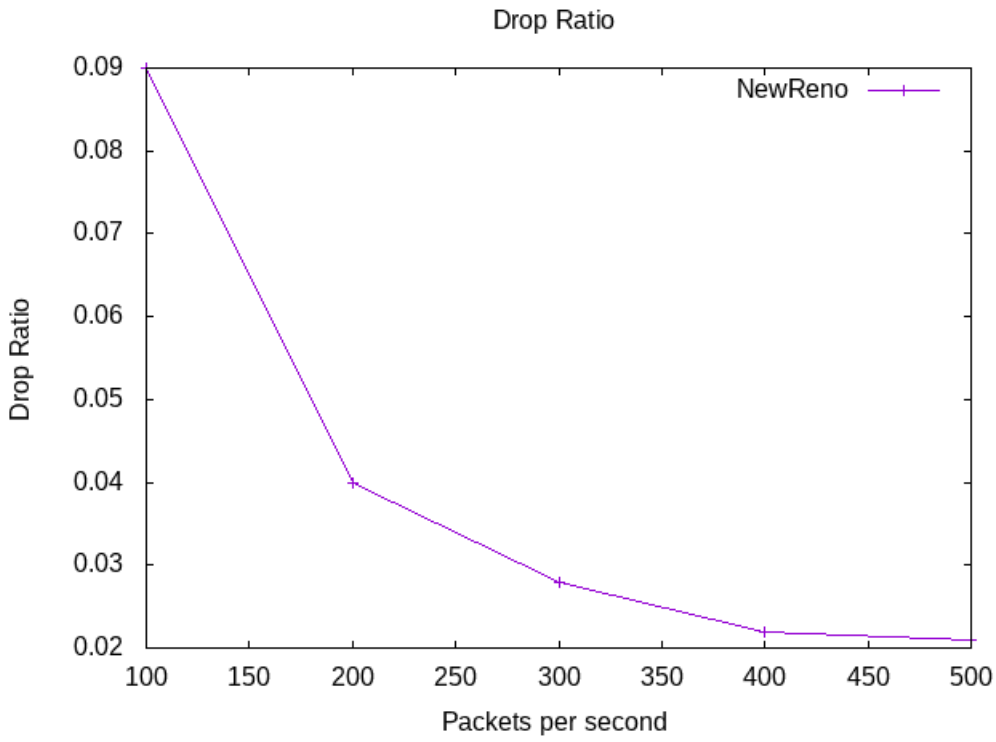

Figure 24: Drop Ratio(Varying Packet per sec)

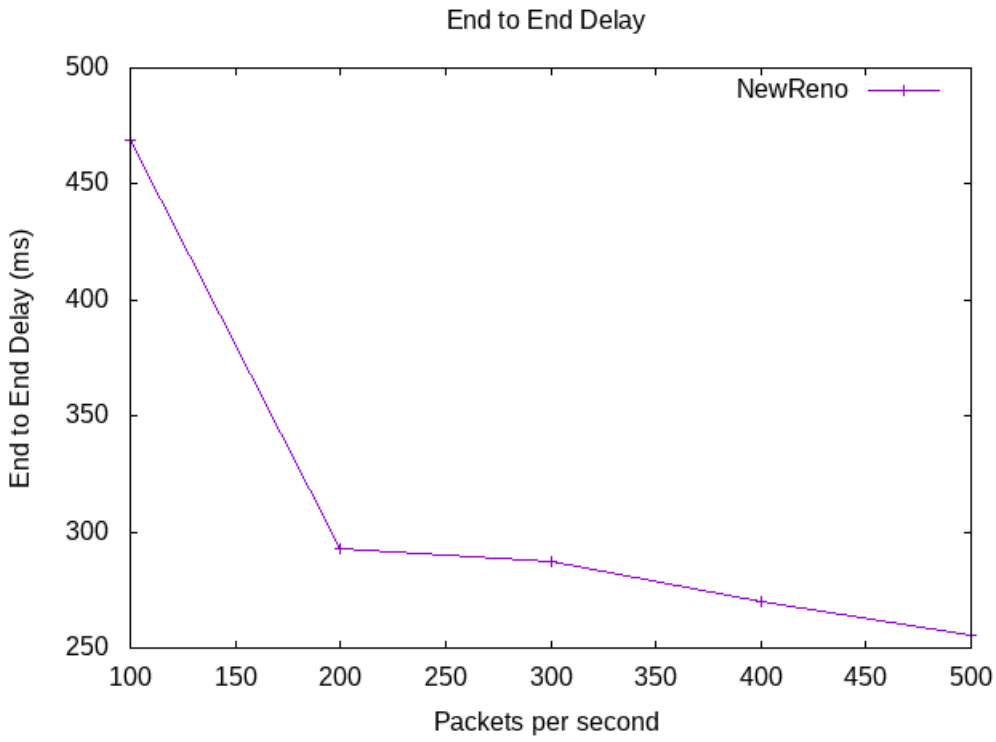

Figure 25: End to end delay(Varying Packet per sec)

6.2 Task A(Wireless: low rate 802.14.5)

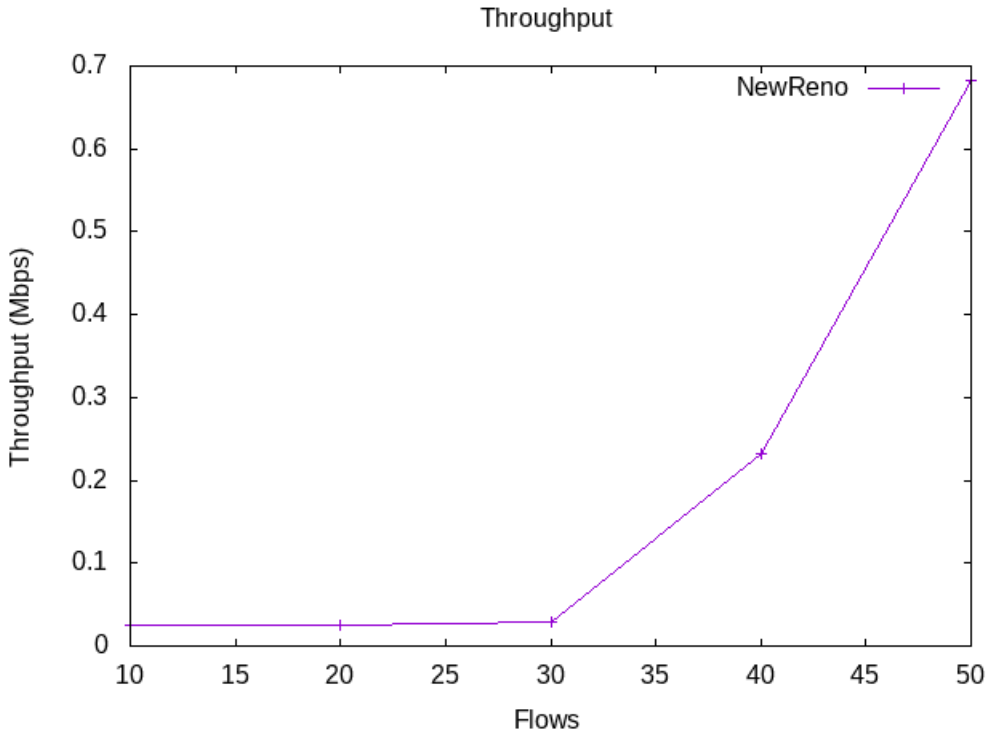

Figure 26: Throughput(varying flow)

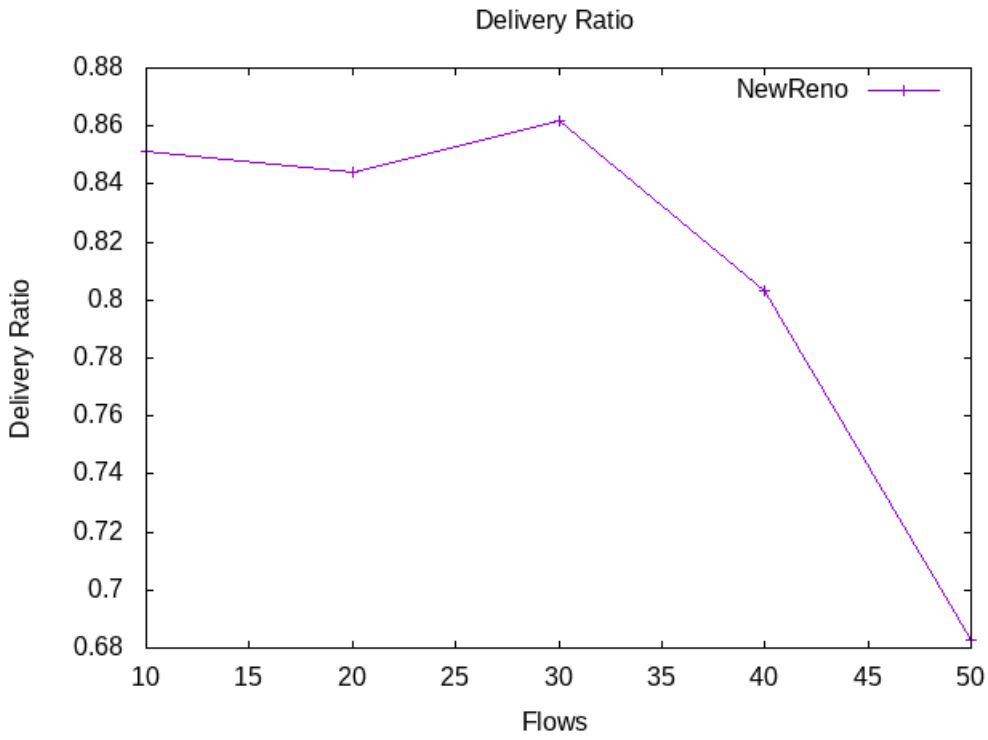

Figure 27: Delivery Ratio(varying flow)

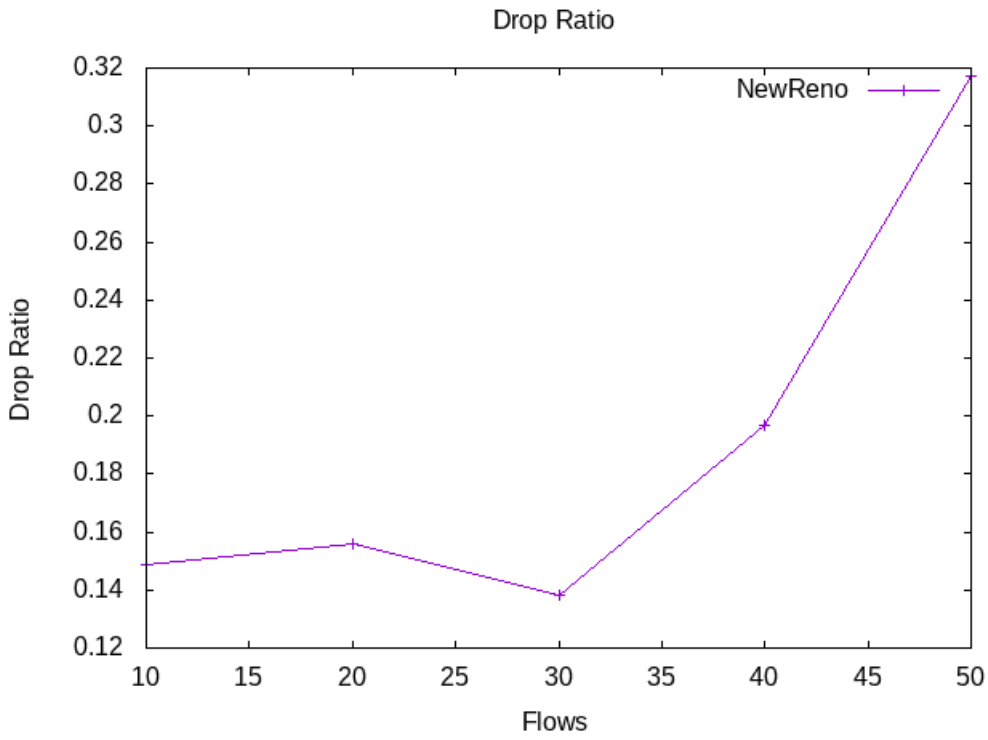

Figure 28: Drop Ratio(varying flow)

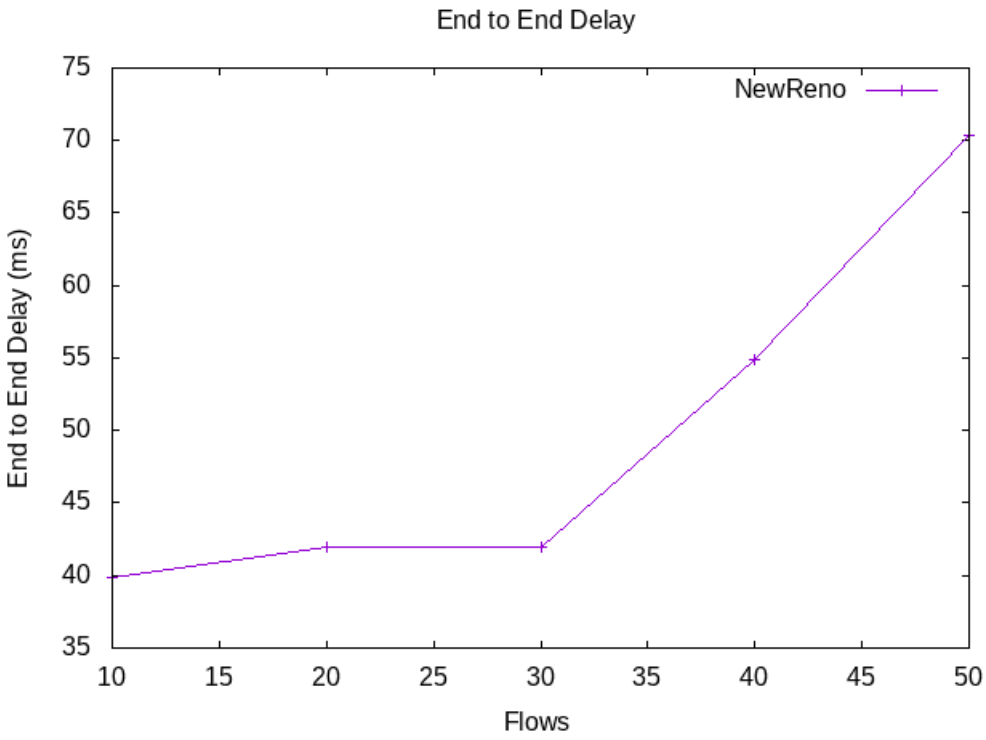

Figure 29: End to end delay(varying flow)

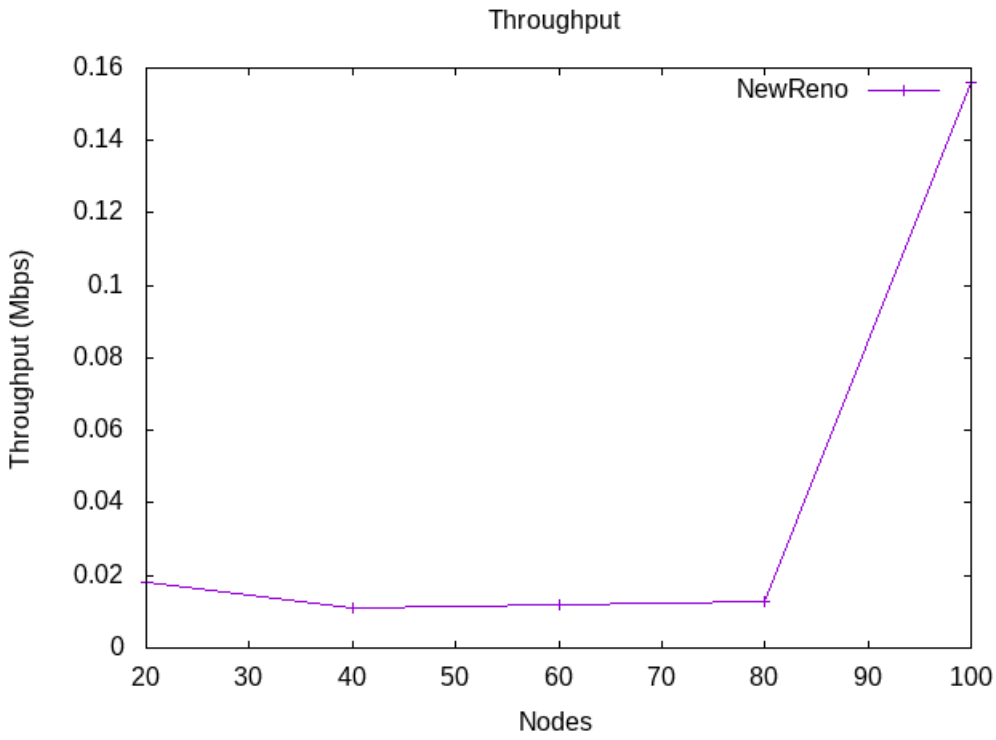

Figure 30: Throughput(Varying Node)

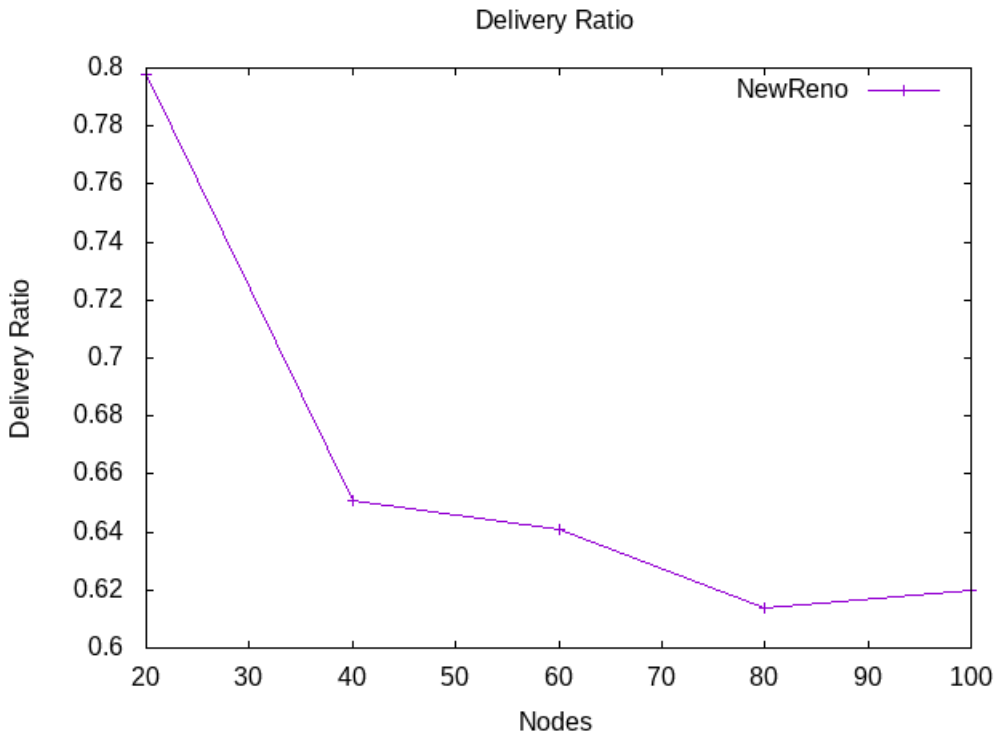

Figure 31: Delivery Ratio(Varying Node)

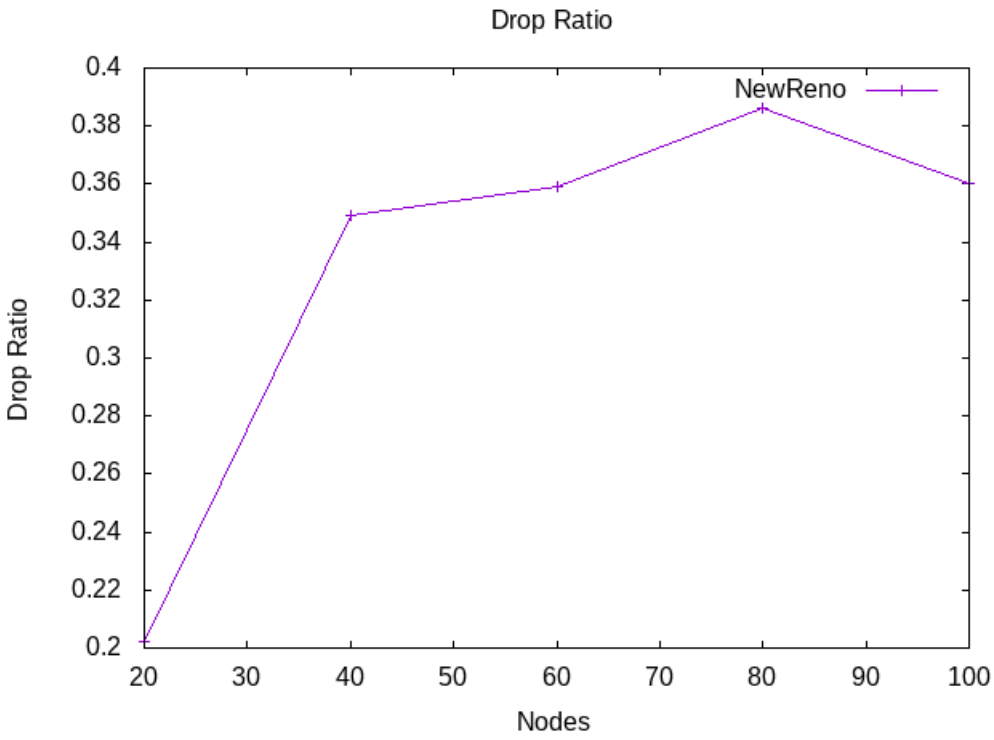

Figure 32: Drop Ratio(Varying Node)

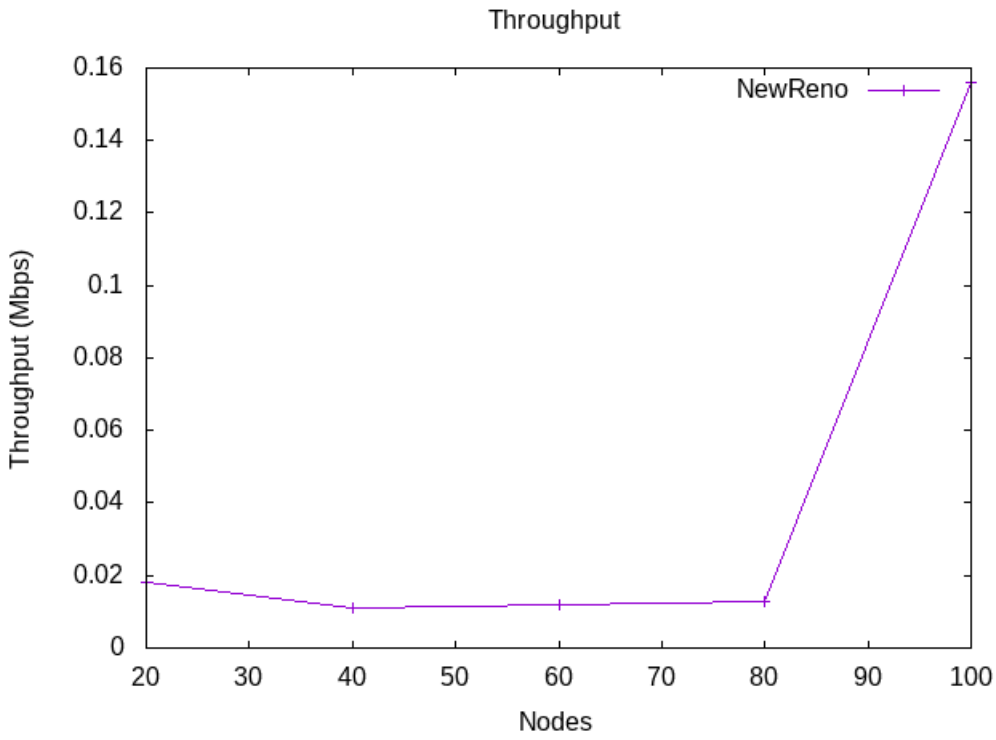

Figure 33: End to end delay(Varying Node)

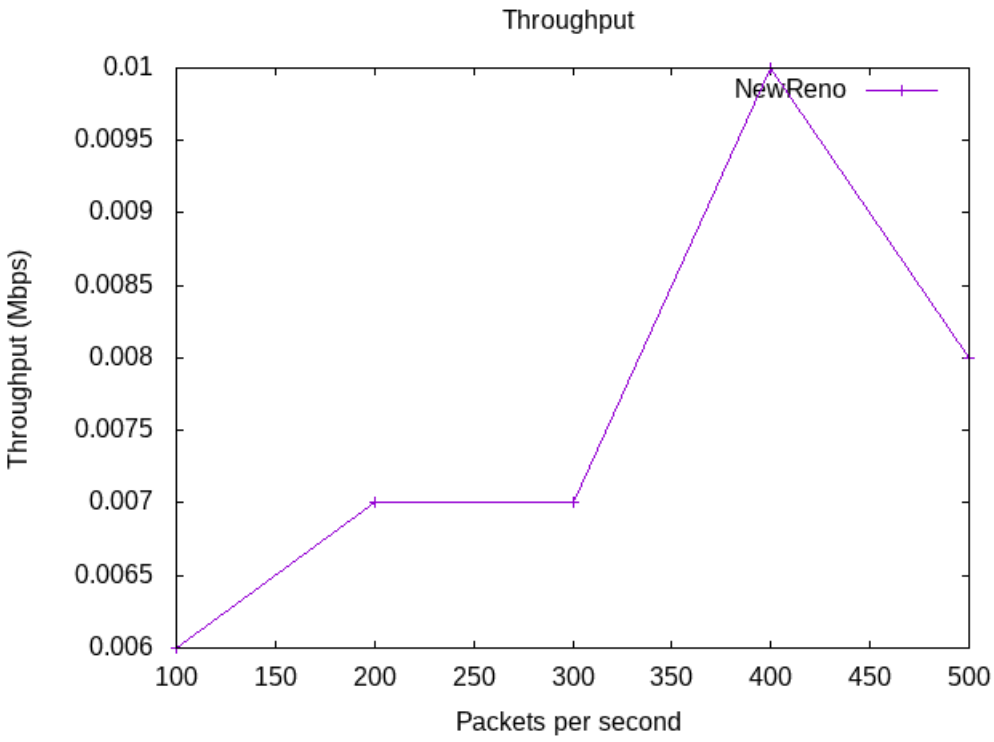

Figure 34: Throughput(Varying Packet per sec)

Figure 35: Delivery Ratio(Varying Packet per sec)

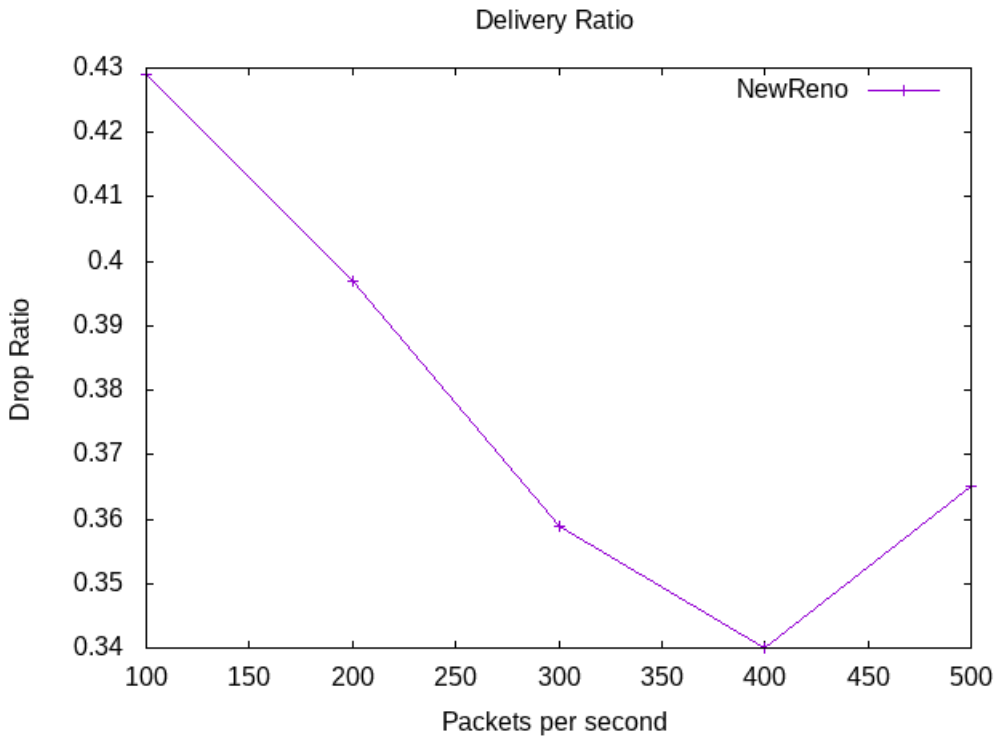

Figure 36: Drop Ratio(Varying Packet per sec)

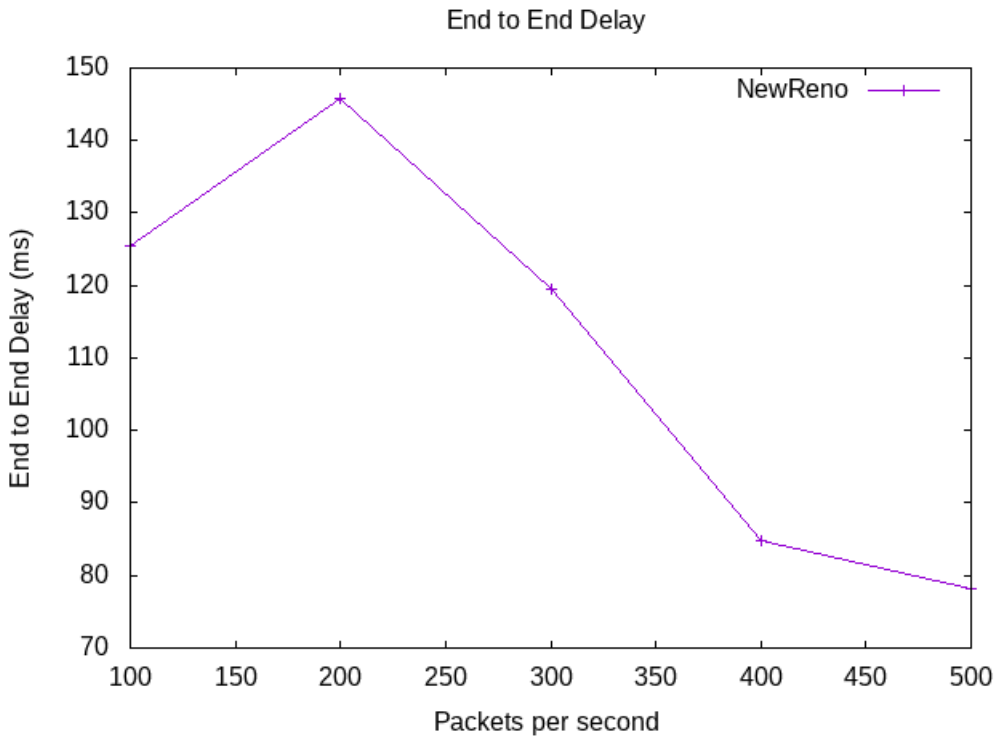

Figure 37: End to end delay(Varying Packet per sec)

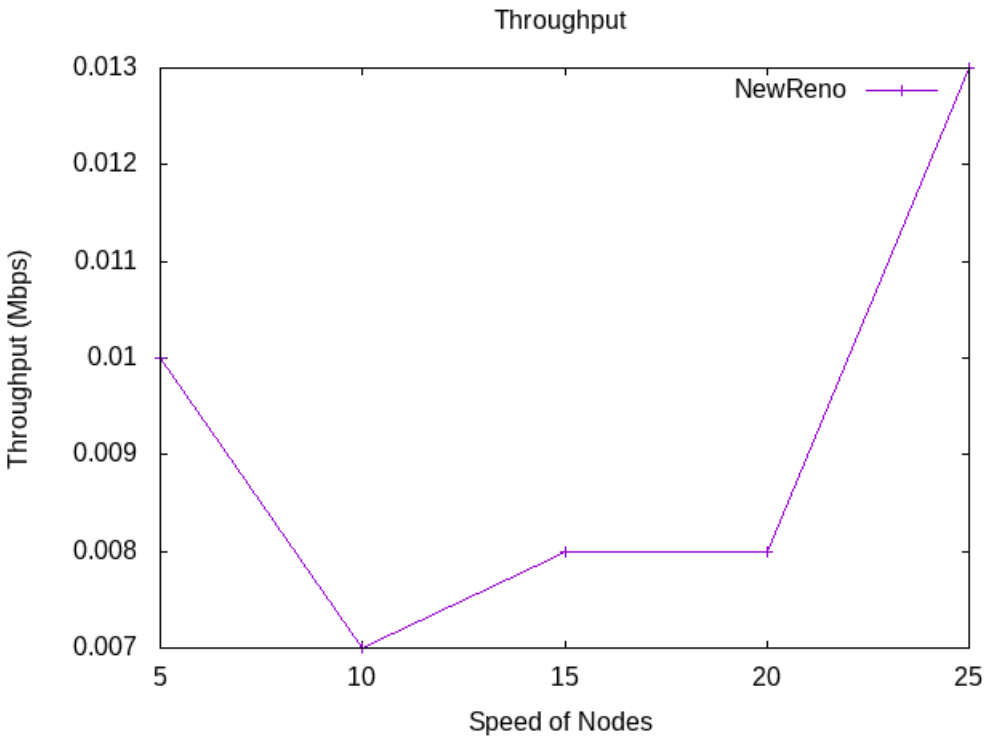

Figure 38: Throughput(Varying node's speed)

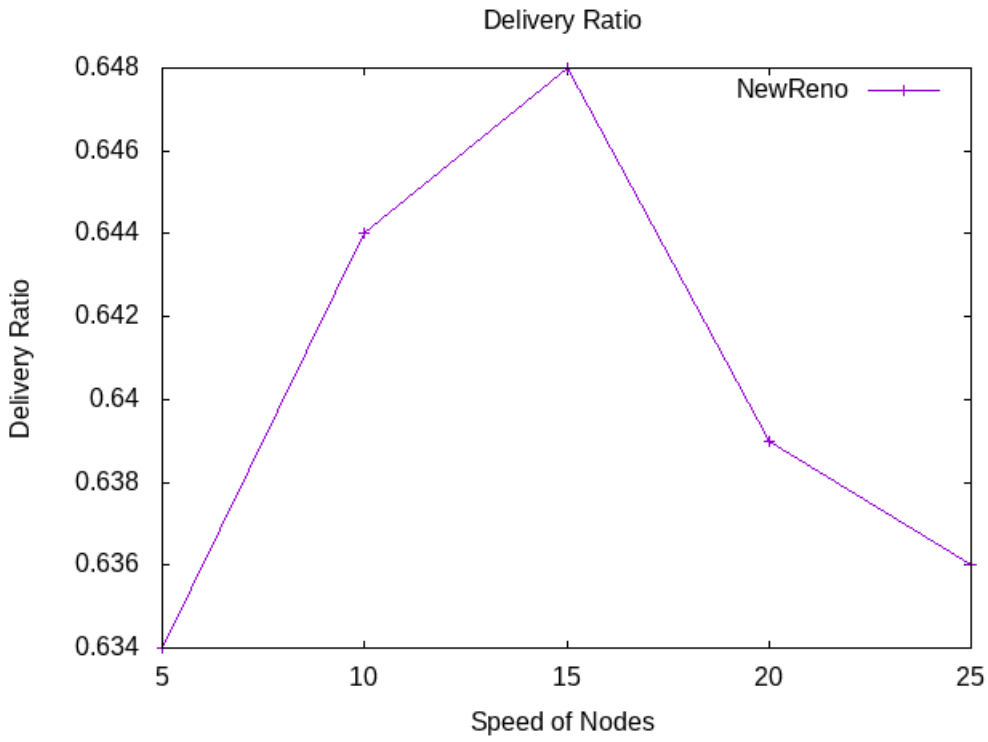

Figure 39: Delivery Ratio(Varying node's speed)

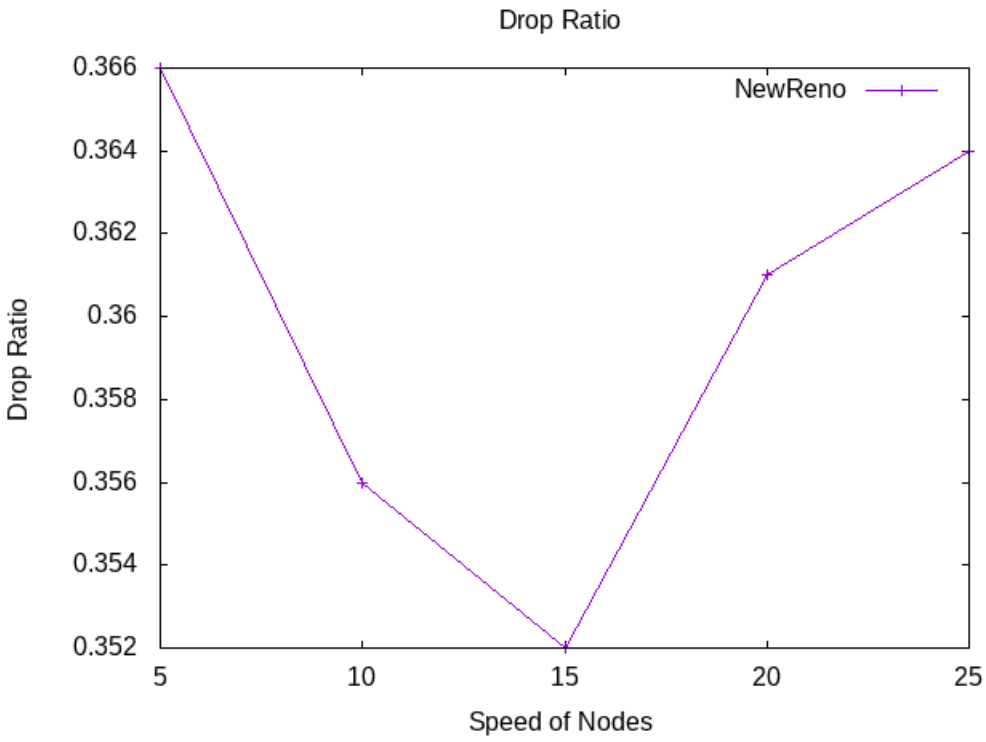

Figure 40: Drop Ratio(Varying node's speed)

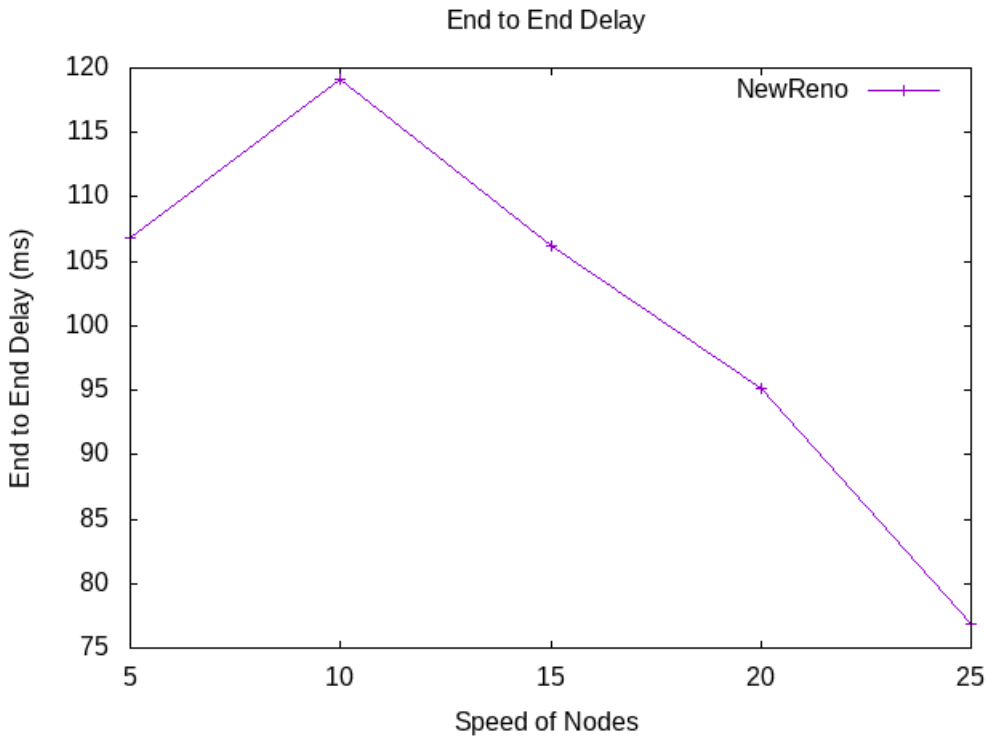

Figure 41: End to end delay(Varying node's speed)

6.3 Task B

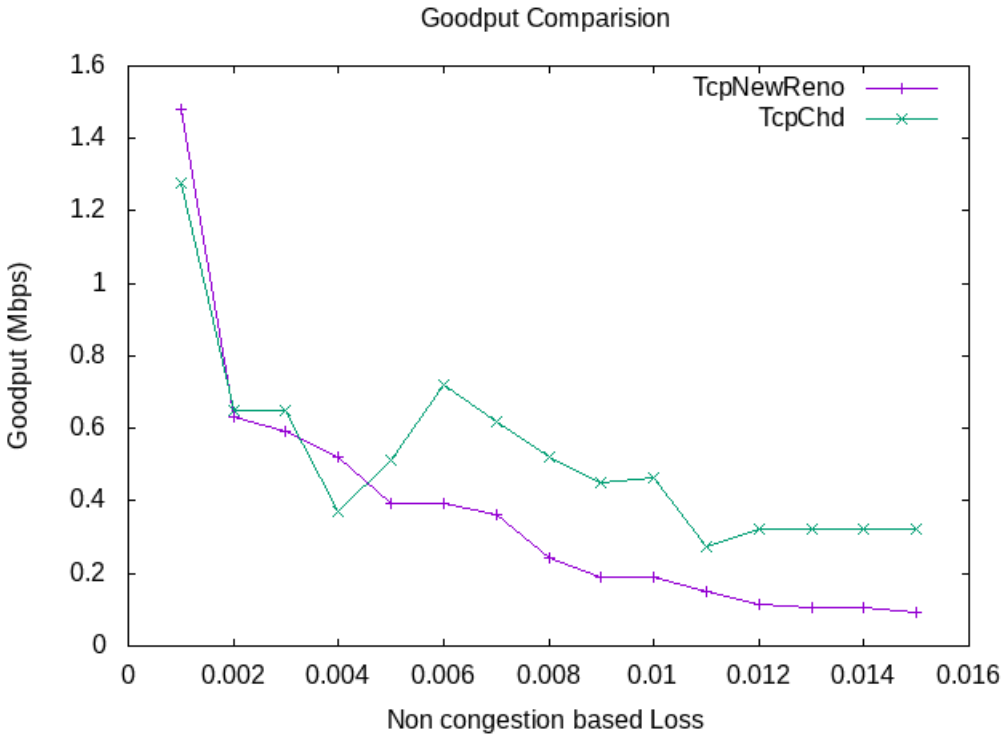

Figure 42: Goodput vs Non congestion based loss

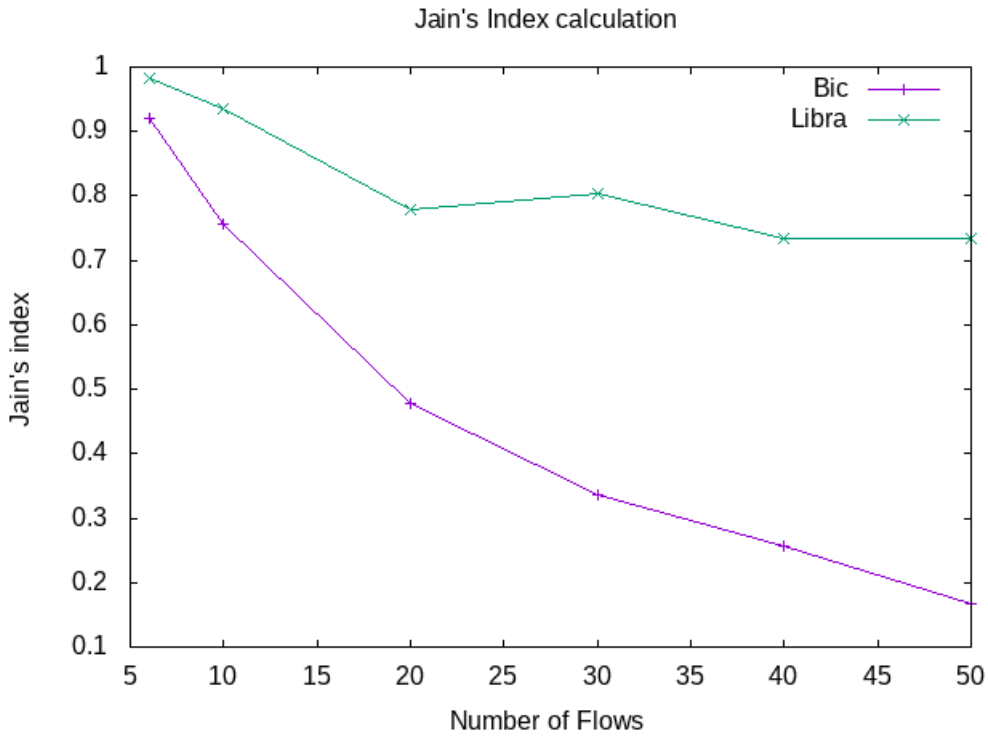

Figure 43: Fairness comparison between TcpNewReno and TcpChd

7 Summary

7.1 Findings of Task A

In task A, we have to simulate two different kinds of networks

- Wired
- Wireless low rate(mobile)

In wired network:
If number of nodes or flows increases:

- throughput increases
- End to end delay increases
- Drop ratio increases
- Delivery ratio decreases
- Goodput decreases

This is because traffic increases as the number of nodes and flows increases. As a result, total number of sending and receiving packets increases which increase the throughput. But so many packets create congestion situation, packets drop increases, that's why received packets in application layer decreases which makes a degradation in goodput value

In wireless low rate network: These metrics should follow the pattern what wired network has and it almost shows similar result. But these metrics don't behave regularly. There may be several reasons behind this

- We used TcpNewReno as our congestion control algorithm. This algorithm may not be the best suitable algorithm to handle low rate data transfer.
- Low rate data transmission increases the chance of packets drop.
- There seems to be a lot packet drop in case of low rate data transfer. They may be dropped due to excessive transmission retries or channel access failure which we found in ns3 Lrwpan documnetation.

7.2 Findings of Task B

We implemented a new Tcp congestion algorithm named TcpChd which actually improves Tcp NewReno since it discards the non congestion based loss from total packet loss when to change the congestion avoidance window. So it shows better result in "Gootput vs Non congestion base loss" than TcpNewReno which was almost similar to the paper's claim. We also calculated fairness of these two algorithm against different number of flows. As number of flows increases, fairness index decreases for both of the algorithms but fairness of the TcpChd is significantly better when specially flows number are higher.