REPORT ON NS3 TERM PROJECT

An Improved Adaptive Active Queue Management Algorithm Based on Nonlinear Smoothing

Reference:

- An Improved Active Queue Management Algorithm Based on Nonlinear Smoothing.
- Zhang, Jing & Xu, Wen & Wang, Li. (2011).
- Advanced Materials Research. 295-297. 1823-1828. 10.4028/www.scientific.net/AMR.295-297.1823.

Submitted By:

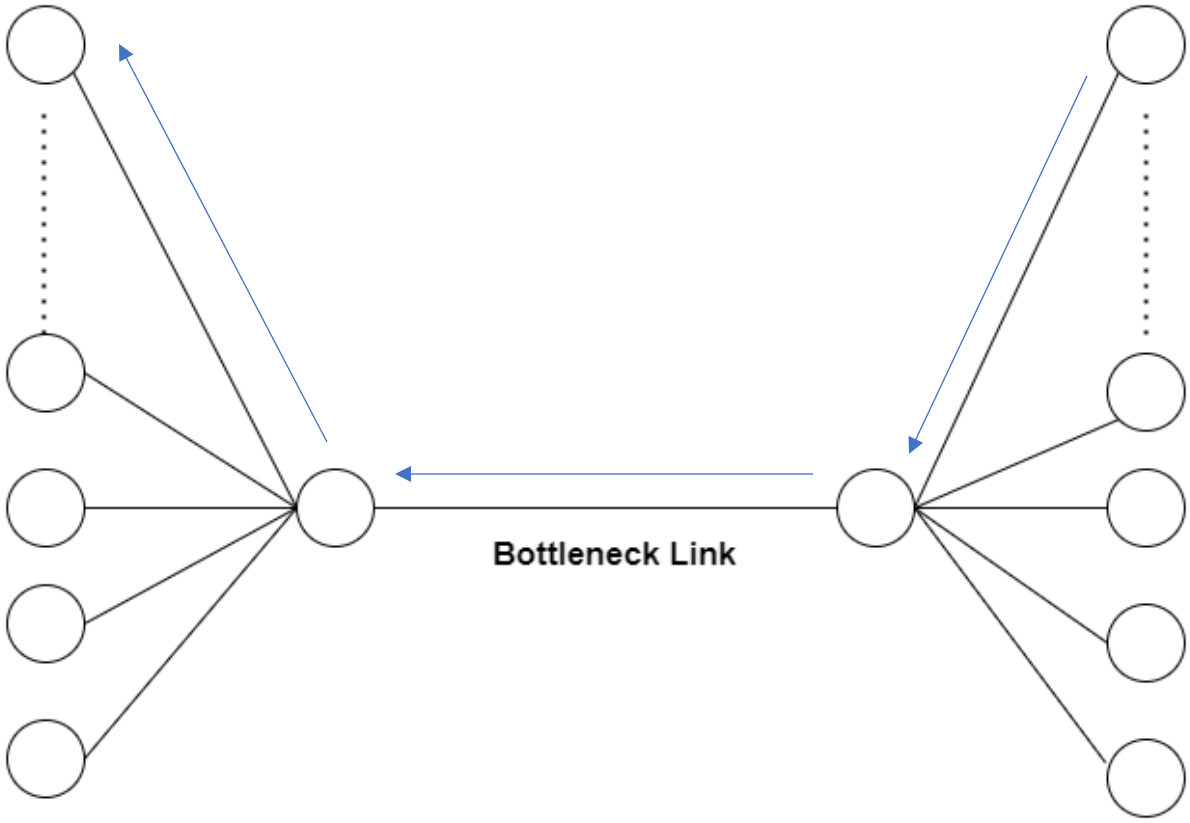
Student ID: 1705025.

Name: Swapnil Dey.

Student ID: 23 February, 2022.

NETWORK TOPOLOGIES

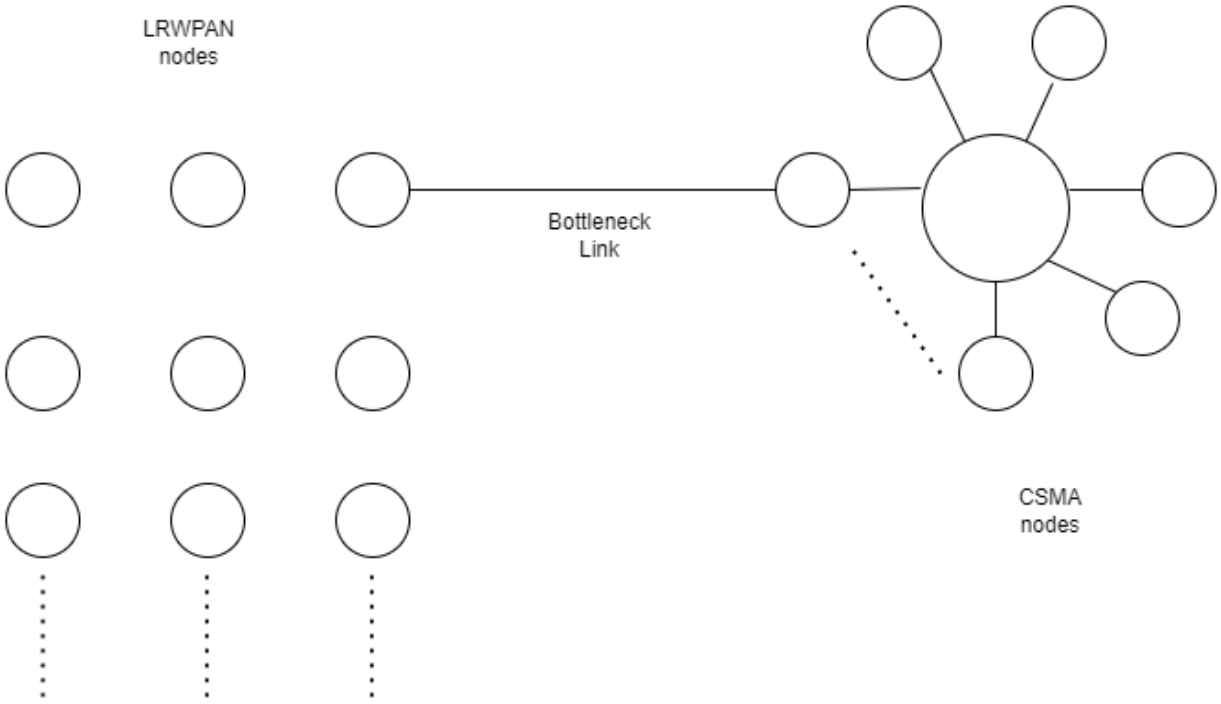
Wired:



In this wired topology, in the right sided nodes, client applications (“OnOffApplication”) are set-up and in the left sided nodes server applications (“PacketSinkApplication”) are set-up. The link in the middle work as bottleneck link as it’s bandwidth is set to low. The queue-disc algorithm is configured to RED/ARED in the bottleneck-link routers. This topology helps to simulate the RED and ARED active queue management algorithms.

NETWORK TOPOLOGIES

Wireless (802.15.4 Static):



In this topology, right sided nodes are CSMA nodes and left sided nodes are LRWPAN nodes. They are connected by a point-to-point network which is the bottleneck link here. Here, client applications are set-up at wireless nodes and the CSMA nodes contain packet sink applications. Here, as the wireless nodes are static in place, as mobility module, “ConstantPositionMobilityModel” is used. In the network layer, IPv6 is

used instead of IPv4. As AQM algorithm, RED queue disc is used here to simulate the queue management.

Parameters under variation:

Here, the parameters those are varied are:

1. **Number of Nodes:** In both wired and wireless topologies, the number of nodes on each side of the bottleneck link can be varied with the "nLeaf" parameter. For example: if nLeaf = 9 , there are total 20 nodes, 18 on each side and 2 in the middle.

2. **Number of Flows:** In both wired and wireless topologies, the number of flows can be varied with the "nFlows" parameter. There can be maximum 2*nLeaf number of flows in the network and minimum of 2 flows.

3. **Packet per Second:** We can also vary the data-rate of applications by varying the "pps" parameter. For example: if we set pps = 200, we mean that each client application will send 200packets/second.

4. **Coverage Area:** In the wireless topology, as the nodes are static, we can configure the range or coverage area of the WiFi channel. We can vary this with help of "range_mult" parameter. For example: if range_mult=2, the coverage area will be doubled of the initially set coverage area.

Overview of the proposed algorithm:

- an improved RED(Random early detection) algorithm which is NARED (Non-linear Adaptive RED).
- **NON-LINEAR:** The main idea is that when Q_{avg} exceeds Q_{max} , a part of queue is idle, so the probability should not be changed to 1 immediately like RED algorithm.
- **ADAPTIVE:** The value of P_{max} will be dynamically adjusted by using of the average queue length and the target of queue length.

Steps:

- Calculating the average Queue size,
  - $Qav = (1 - Wq) * Qav + q * Wq$
- If $Q_{av} < Q_{min}$,
  - P = 0
- If $Q_{min} \leq Q_{av} <$ Bufsize,
  - $P_{NARED} = \frac{Pmax(Qav - Qmin)^3}{(1-Pmax)(Bufsize - Qmax)^3 + Pmax(Qav - Qmax)^3}$
  - $P = \frac{P_{NARED}}{(1-count*P_{NARED})}$

- Adjusting the value of according to the relation of the length of average queue and the target of queue,
  - Pmax = Pmax * b , $Q_{av} < k1$
  - Pmax = Pmax + a , $Q_{av} > k2$

- If $Q_{av} \geq$ Bufsize,
  - P = 1

Modifications made in NS3:

Modifications were needed in two of the files in NS3 library.

1. Red-queue-disc.h
2. Red-queue-disc.cc

First of all, to add the modified algorithm, we need a way to enable this variant of RED algorithm. For this, some parameters and functions were added in the "Red-queue-disc.h" header file. One of the parameters is "m_isNARED" which is used to enable the NARED algorithm. Then, unlike RED and ARED algorithms, we need the device queue buffer size in the modified NARED algorithm. So, another parameter "m_bufferSize" was added. We also needed to add another function prototype CalculatePNared() to calculate the modified packet-dropping probability.

In the "Red-queue-disc.cc" file, we first needed to initialize the parameters needed for NARED in the function InitializeParams(). We also needed to implement the CalculatePNared() function, which uses

non-linear smoothing to calculate the packet drop probability. We needed to make some changes in the UpdateMaxP() function, which adjusts the maximum packet drop probability with respect to the calculated average queue size. We also needed to make some changes to the Estimator() function, which calculates the average queue size.

After making these changes, we can enable NARED from our simulation code and also set necessary parameters.

```
void
RedQueueDisc::InitializeParams (void)
{
  NS_LOG_FUNCTION (this);
  NS_LOG_INFO ("Initializing RED params.");

  m_cautious = 0;
  m_ptc = m_linkBandwidth.GetBitRate () / (8.0 * m_meanPktSize);

  if (m_isARED)
    {
      // Set m_minTh, m_maxTh and m_qW to zero for automatic setting
      m_minTh = 0;
      m_maxTh = 0;
      m_qW = 0;

      // Turn on m_isAdaptMaxP to adapt m_curMaxP
      m_isAdaptMaxP = true;
    }


  if(m_isNared)
  {
     // Set m_minTh, m_maxTh and m_qW to zero for automatic setting
      m_minTh = 0;
      m_maxTh = 0;
      m_qW = 0;

      // adaptive MaxP also needed in NARED
      m_isAdaptMaxP = true;
  }

```

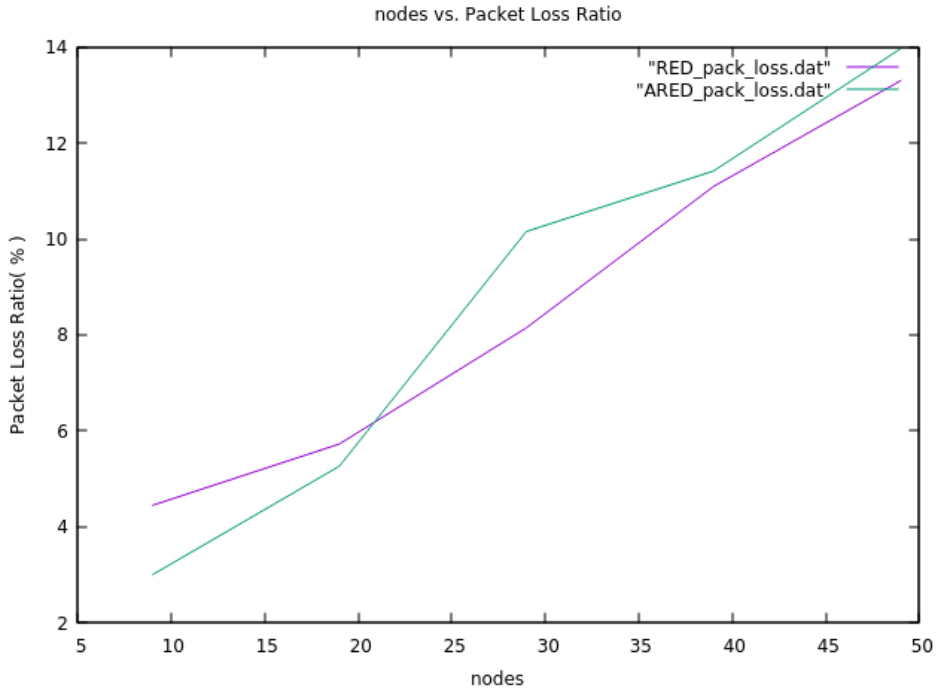
```
// Returns a probability for NARED using these function parameters for the DropEarly function
double
RedQueueDisc::CalculatePNared(void)
{
  NS_LOG_FUNCTION (this);
  double p;

  if(m_qAvg <= m_minTh)
  {
    p = 0.0;
  }

  else if(m_qAvg >= m_bufferSize)
  {
    p = 1.0;
  }

  else
  {
    //calculate the probability using demi-cauchy distribution
    p = (m_curMaxP*pow((m_qAvg-m_minTh),3))/((1-m_curMaxP)*pow((m_bufferSize-m_minTh),3) + m_curMaxP*(pow((m_qAvg-m_maxTh),3)));
  }

  if(p > 1.0)
  {
    p = 1.0;
  }

  return p;
}
```

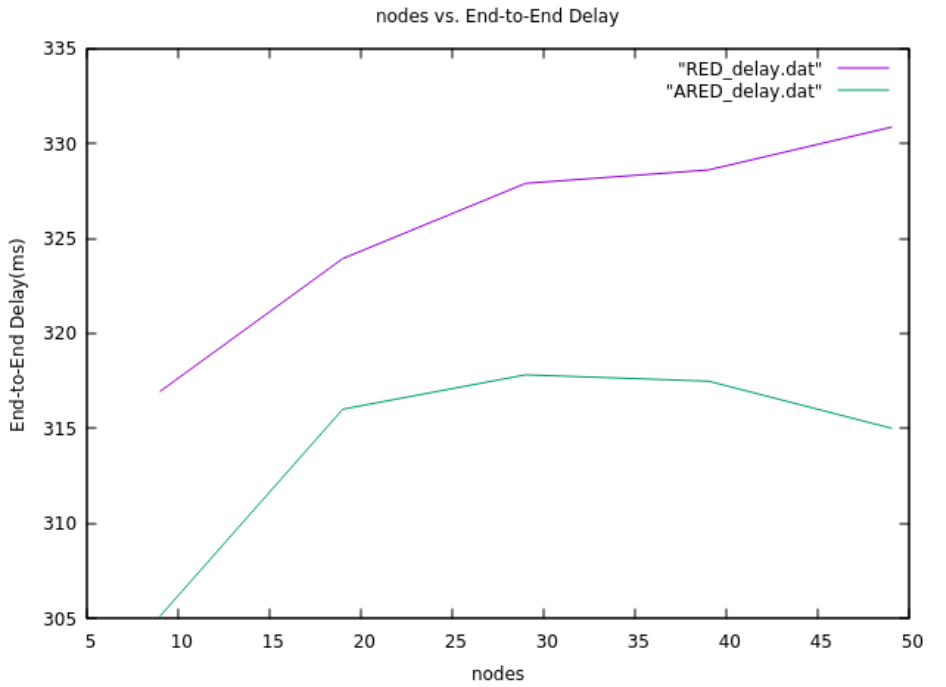
```
double
RedQueueDisc::Estimator (uint32_t nQueued, uint32_t m, double qAvg, double qW)
{
  NS_LOG_FUNCTION (this << nQueued << m << qAvg << qW);

  double newAve;
  if(m_isNared)
  {
    newAve = qAvg*(1-qW);
  }
  else
    newAve = qAvg * std::pow (1.0 - qW, m);

  newAve += qW * nQueued;

  Time now = Simulator::Now ();
  if (m_isAdaptMaxP && now > m_lastSet + m_interval)
    {
      UpdateMaxP (newAve);
    }
  else if (m_isFengAdaptive)
    {
      UpdateMaxPFeng (newAve);  // Update m_curMaxP in MIMD fashion.
    }

  return newAve;
}
```

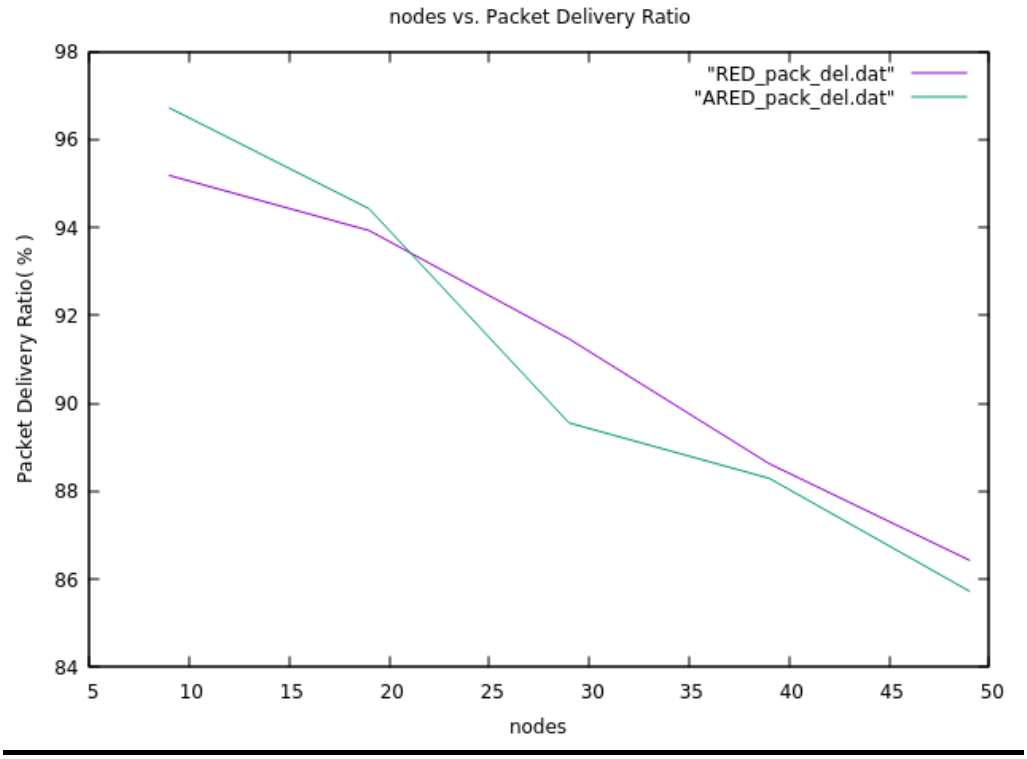
```
// Update m_curMaxP to keep the average queue length within the target range.
void
RedQueueDisc::UpdateMaxP (double newAve)
{
  NS_LOG_FUNCTION (this << newAve);

  Time now = Simulator::Now ();
  double m_part;
  if(m_isNared)
  {
    m_part = 0.4*(m_bufferSize-m_minTh);
  }
  else
    m_part = 0.4 * (m_maxTh - m_minTh);
  // AIMD rule to keep target Q~1/2(m_minTh + m_maxTh)
  if (newAve < m_minTh + m_part && m_curMaxP > m_bottom)
    {
      // we should increase the average queue size, so decrease m_curMaxP
      m_curMaxP = m_curMaxP * m_beta;
      m_lastSet = now;
    }

  double temp;
  temp = (m_maxTh-m_part);
  if(m_isNared)
  {
    temp = 0.6*(m_bufferSize-m_minTh);
  }
  else if (newAve > temp && m_top > m_curMaxP)
    {
      // we should decrease the average queue size, so increase m_curMaxP
      double alpha = m_alpha;
      if (alpha > 0.25 * m_curMaxP)
        {
          alpha = 0.25 * m_curMaxP;
        }
      m_curMaxP = m_curMaxP + alpha;
      m_lastSet = now;
    }
}
```

Results with Graphs (TASK A):

Wired:







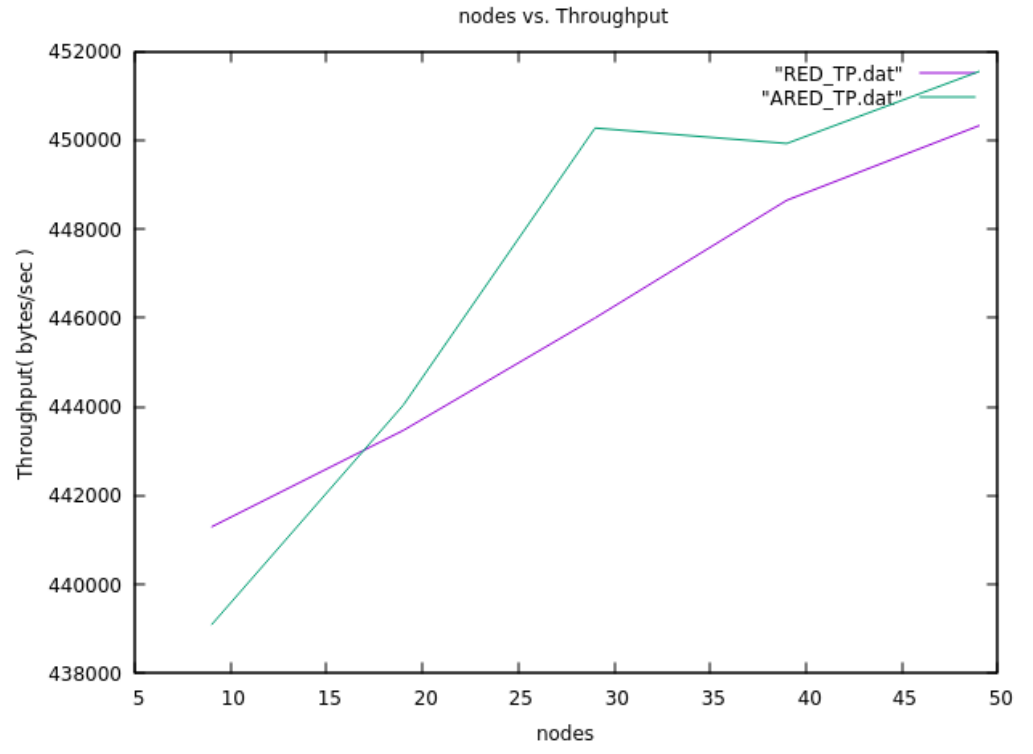
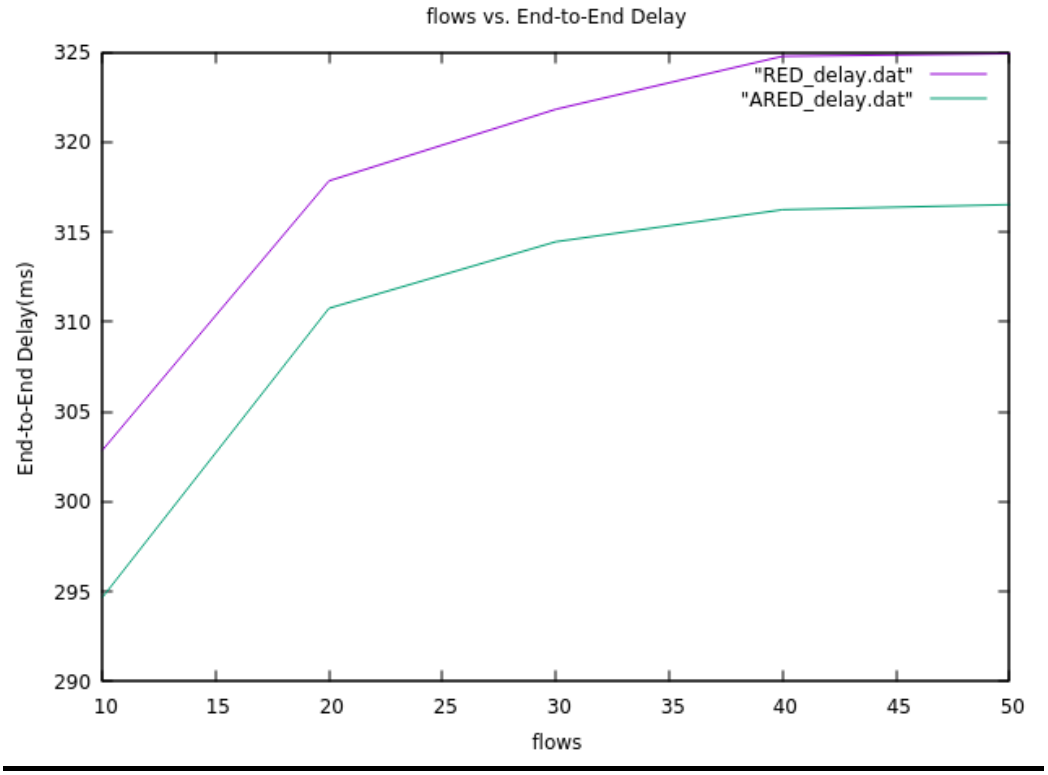
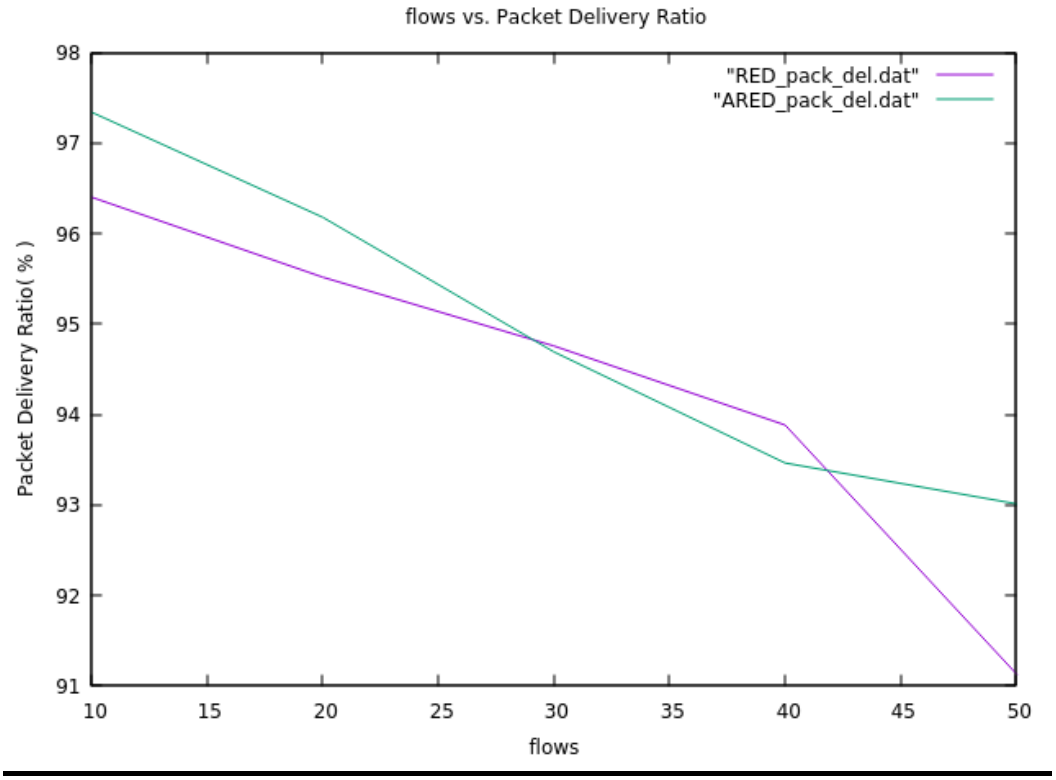
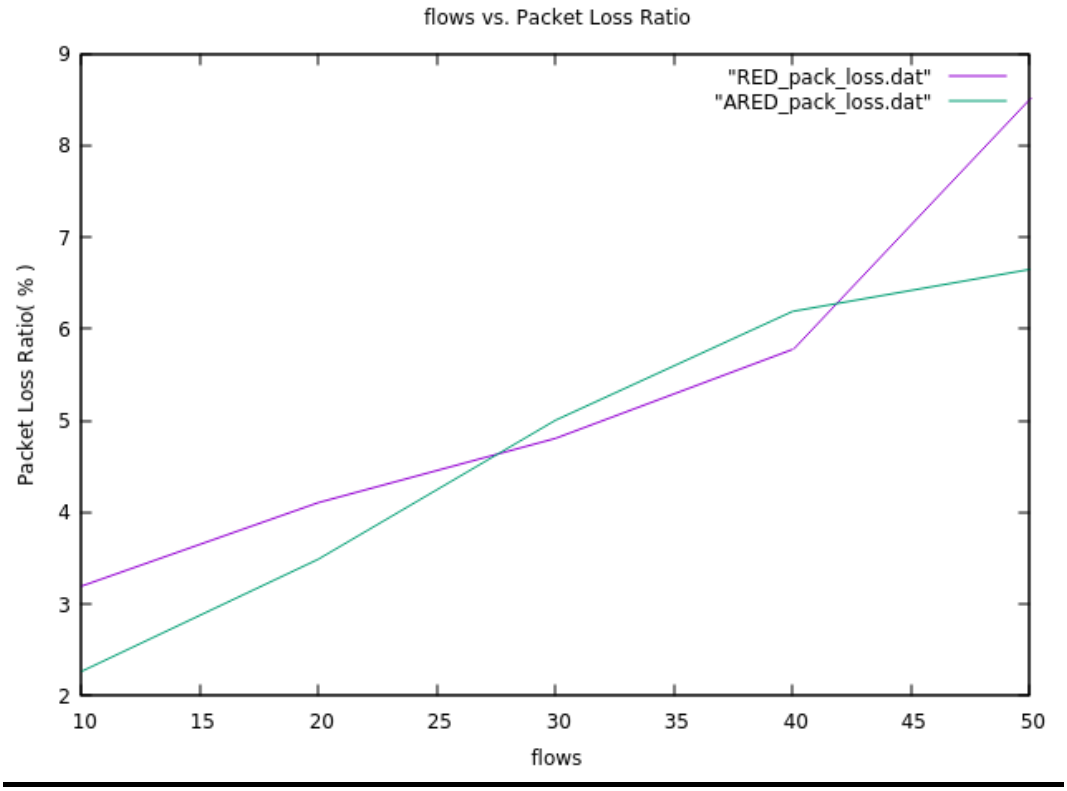


Fig: varying number of nodes(wired)


flows vs. End-to-End Delay
"RED_delay.dat"
"ARED_delay.dat"
End-to-End Delay(ms)
flows
325
320
315
310
305
300
295
290
10
15
20
25
30
35
40
45
50


flows vs. Packet Delivery Ratio
"RED_pack_del.dat"
"ARED_pack_del.dat"
Packet Delivery Ratio(%)
flows
98
97
96
95
94
93
92
91
10
15
20
25
30
35
40
45
50



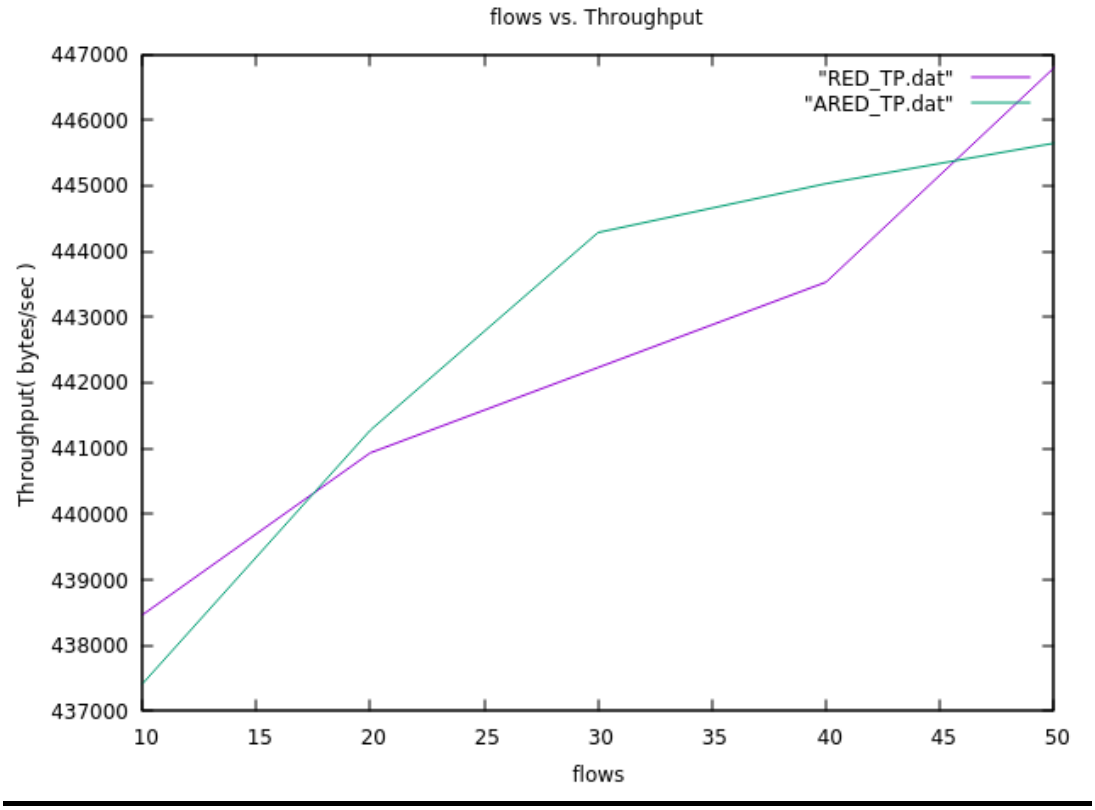
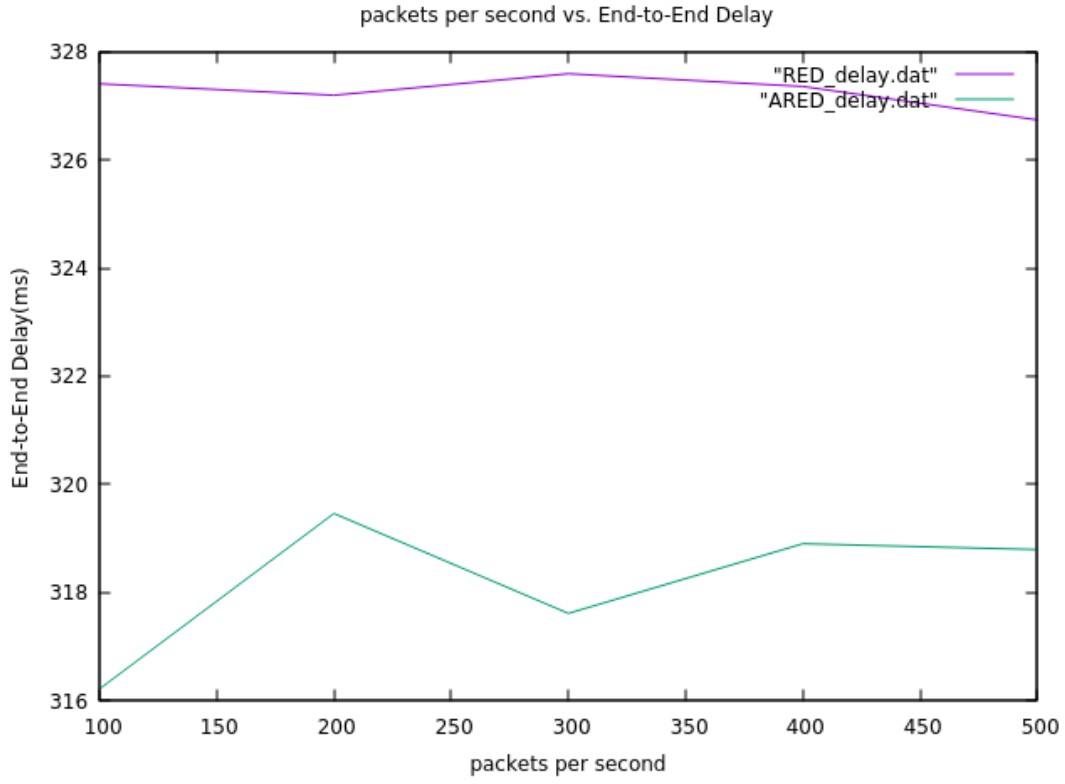
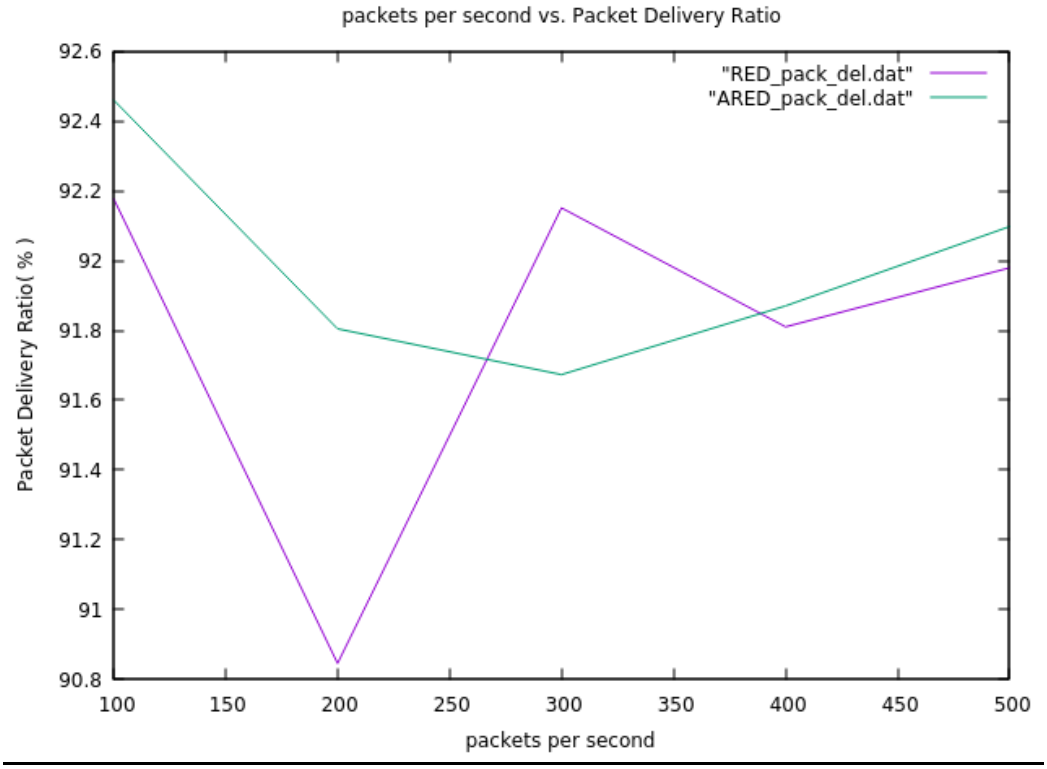
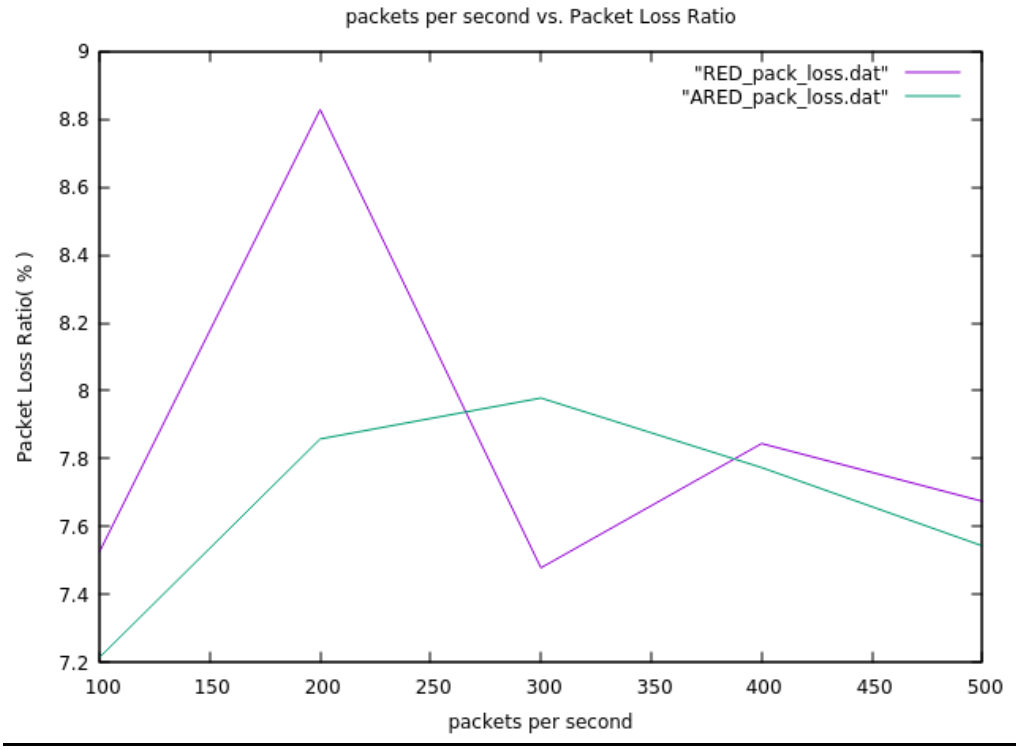


Fig: varying number of flows(wired)


packets per second vs. End-to-End Delay
"RED_delay.dat"
"ARED_delay.dat"
End-to-End Delay(ms)
packets per second
328
326
324
322
320
318
316
100
150
200
250
300
350
400
450
500


packets per second vs. Packet Delivery Ratio
"RED_pack_del.dat"
"ARED_pack_del.dat"
Packet Delivery Ratio(%)
packets per second
92.6
92.4
92.2
92
91.8
91.6
91.4
91.2
91
90.8
100
150
200
250
300
350
400
450
500



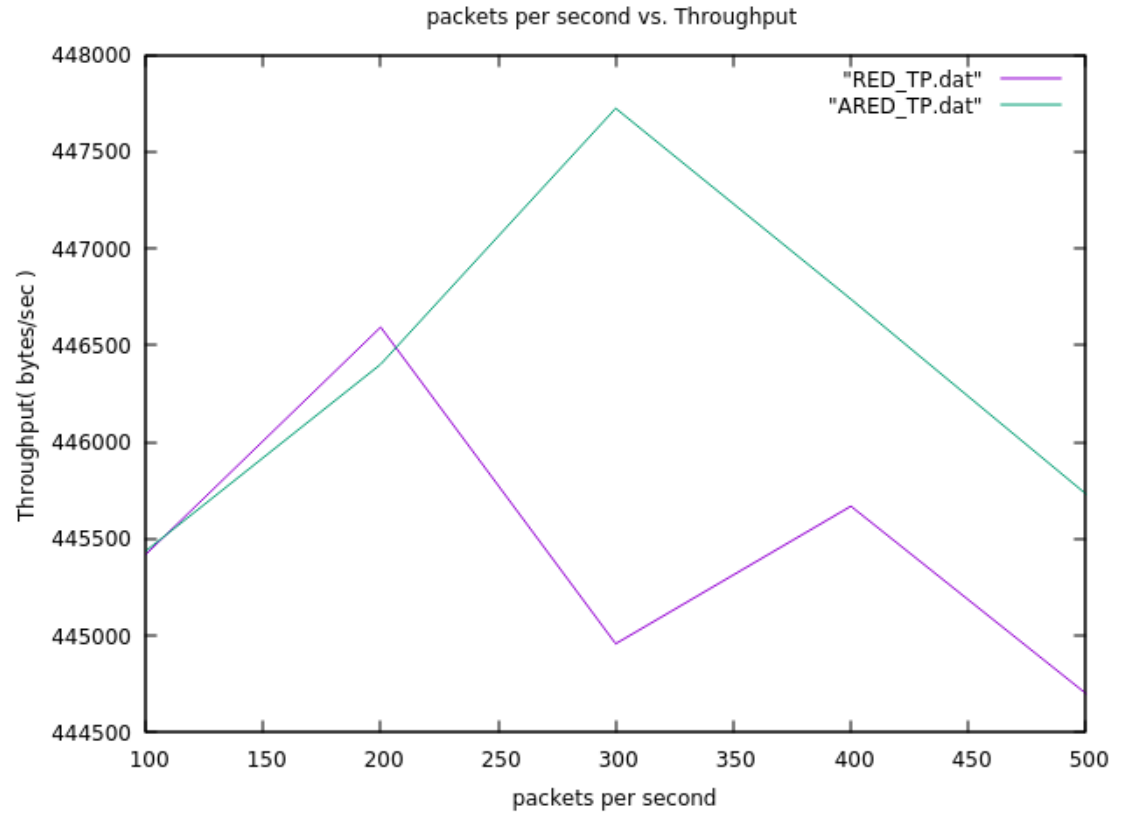
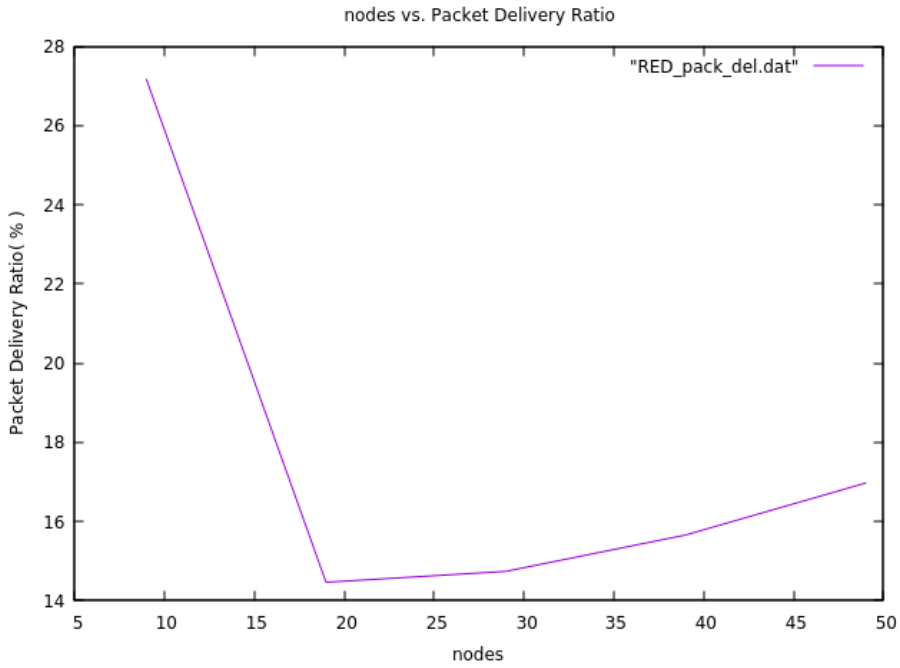
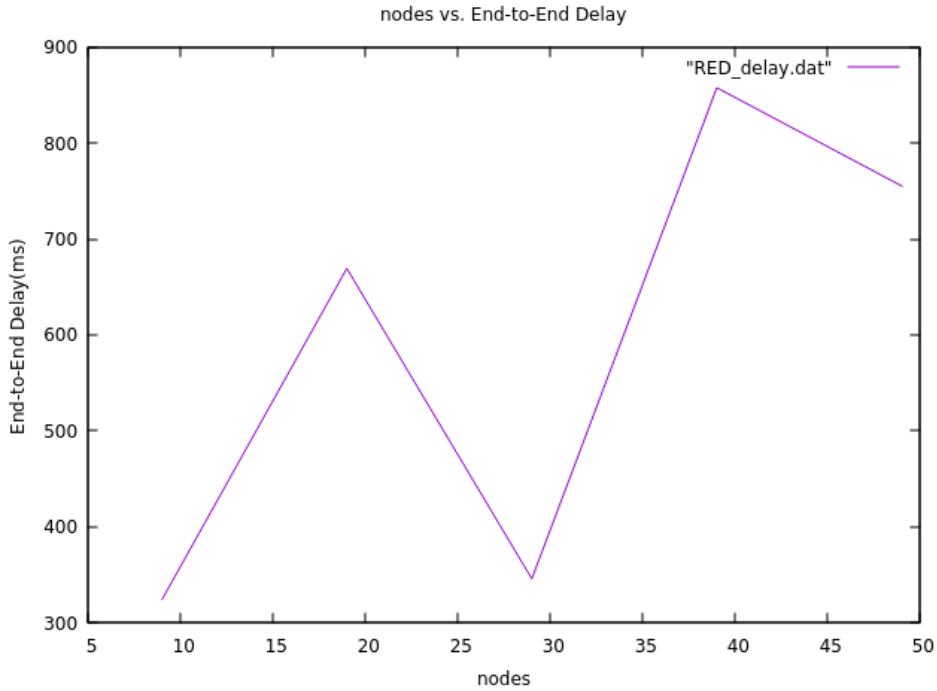
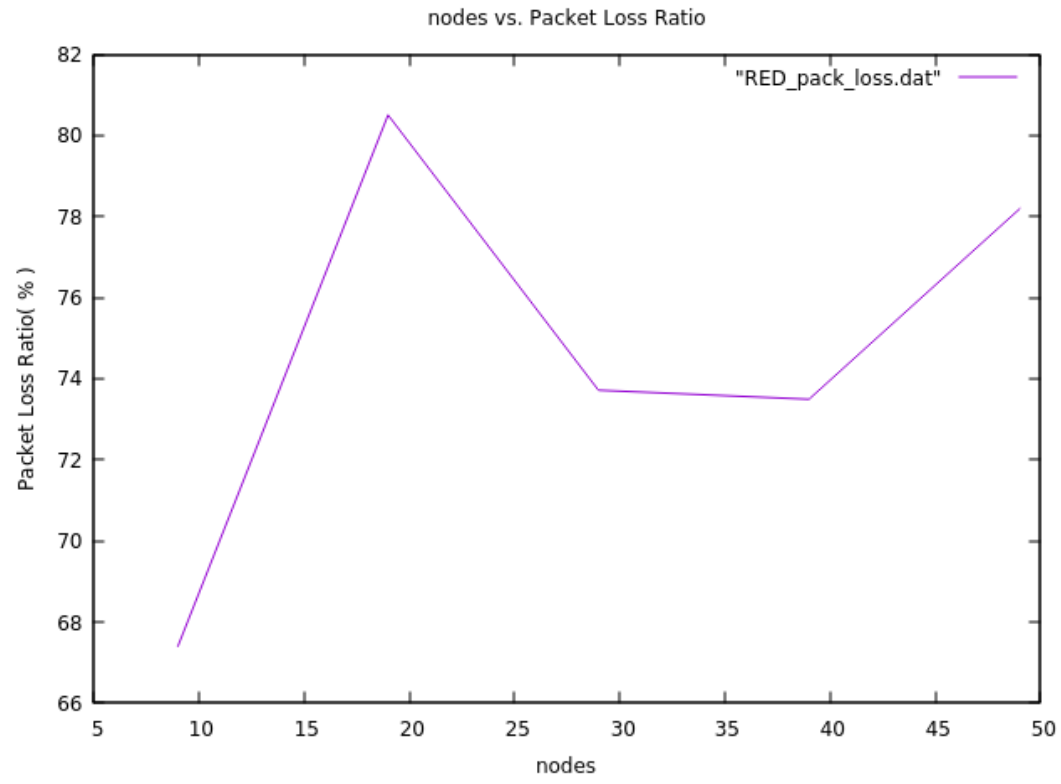


Fig: varying packets per second (wired)

Wireless:


nodes vs. Packet Delivery Ratio
"RED_pack_del.dat"
Packet Delivery Ratio(%)
nodes
28
26
24
22
20
18
16
14
5
10
15
20
25
30
35
40
45
50


nodes vs. End-to-End Delay
"RED_delay.dat"
End-to-End Delay(ms)
nodes
900
800
700
600
500
400
300
5
10
15
20
25
30
35
40
45
50



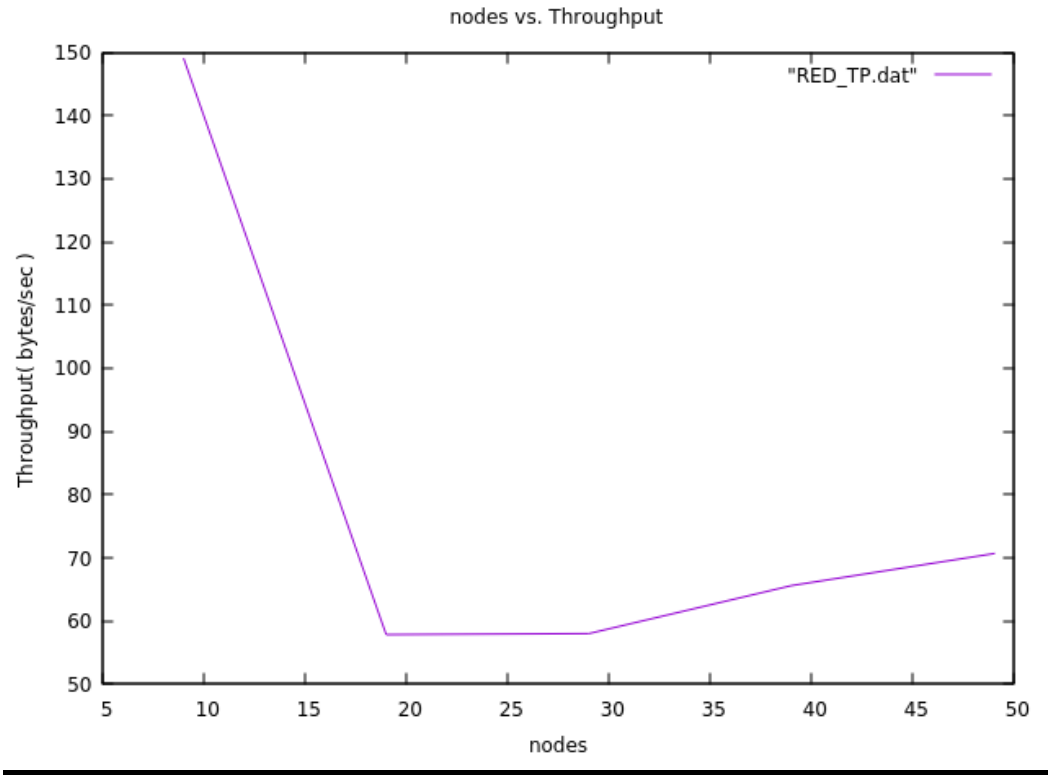
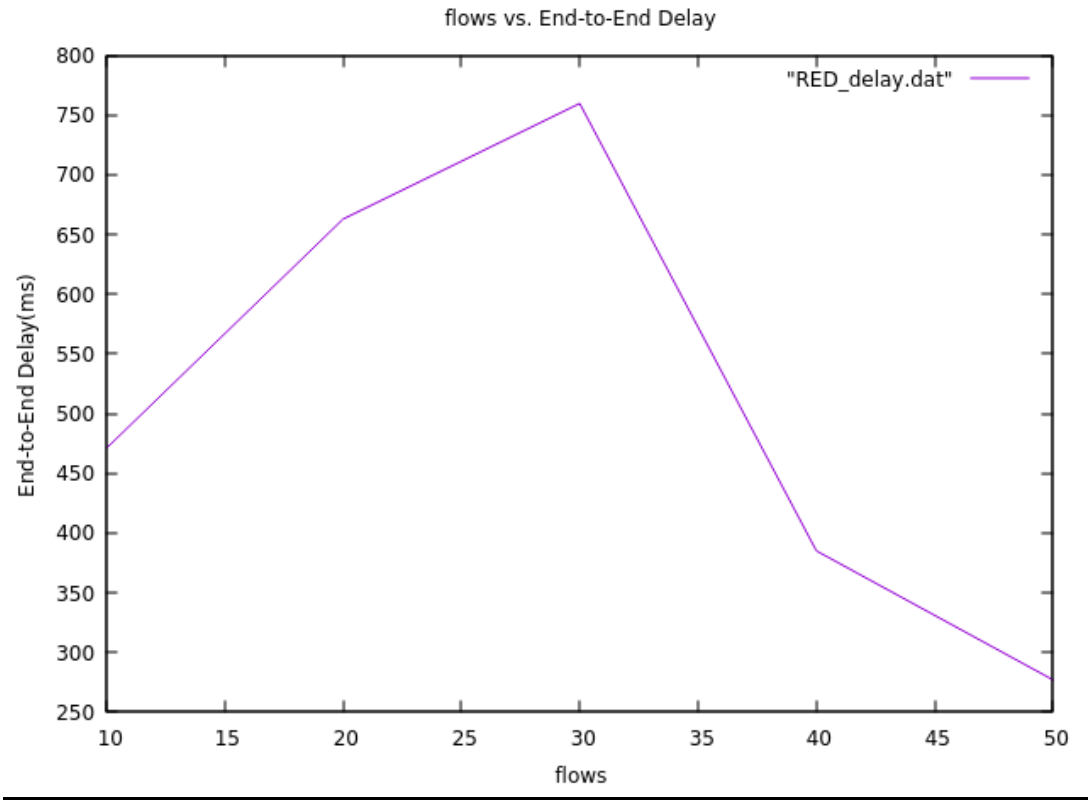
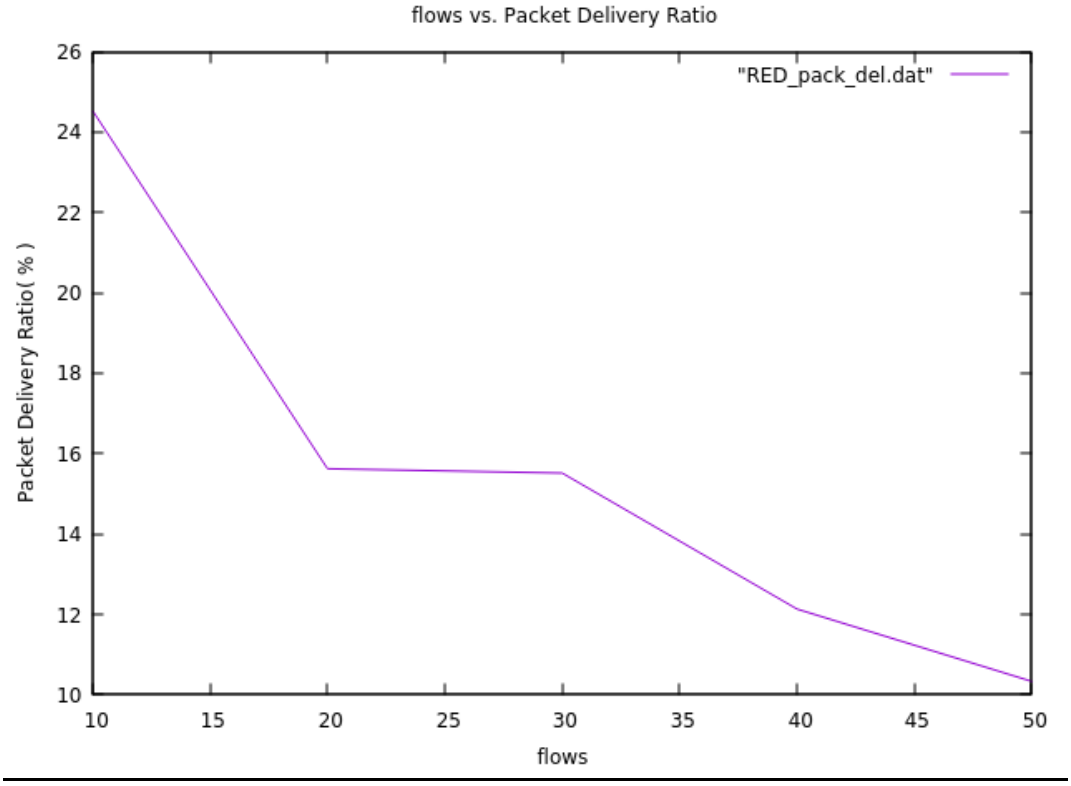
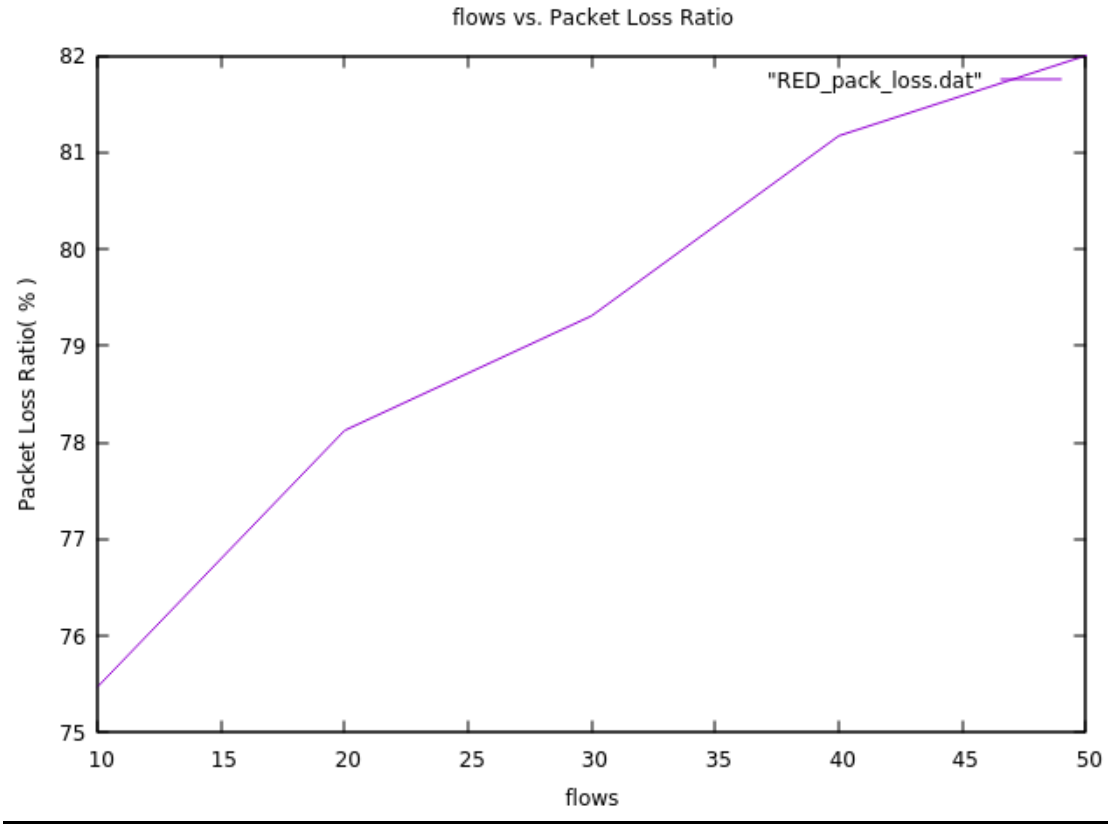


Fig: varying number of nodes(wireless)


flows vs. End-to-End Delay
"RED_delay.dat"
End-to-End Delay(ms)
flows
800
750
700
650
600
550
500
450
400
350
300
250
10
15
20
25
30
35
40
45
50


flows vs. Packet Delivery Ratio
"RED_pack_del.dat"
Packet Delivery Ratio(%)
flows
26
24
22
20
18
16
14
12
10
10
15
20
25
30
35
40
45
50



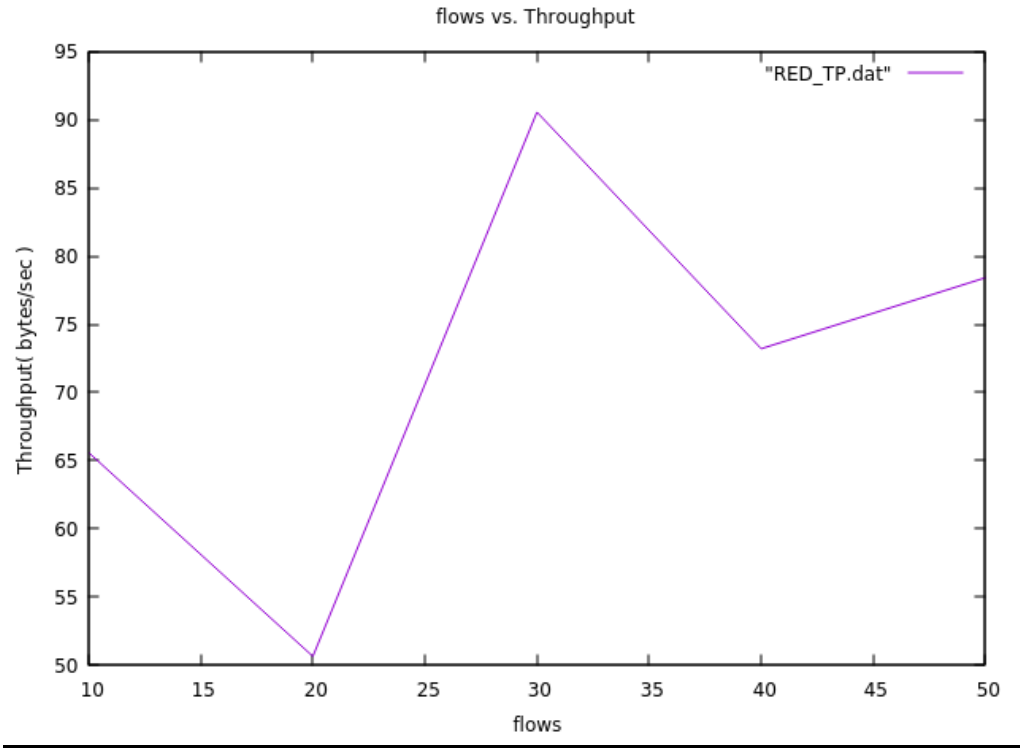
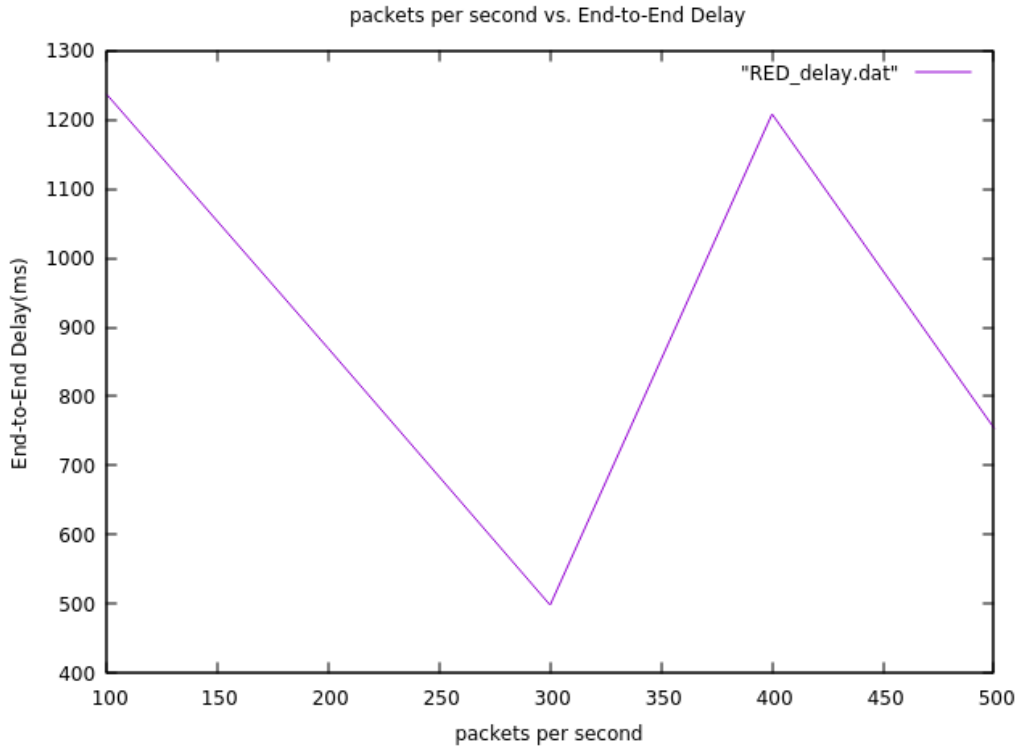
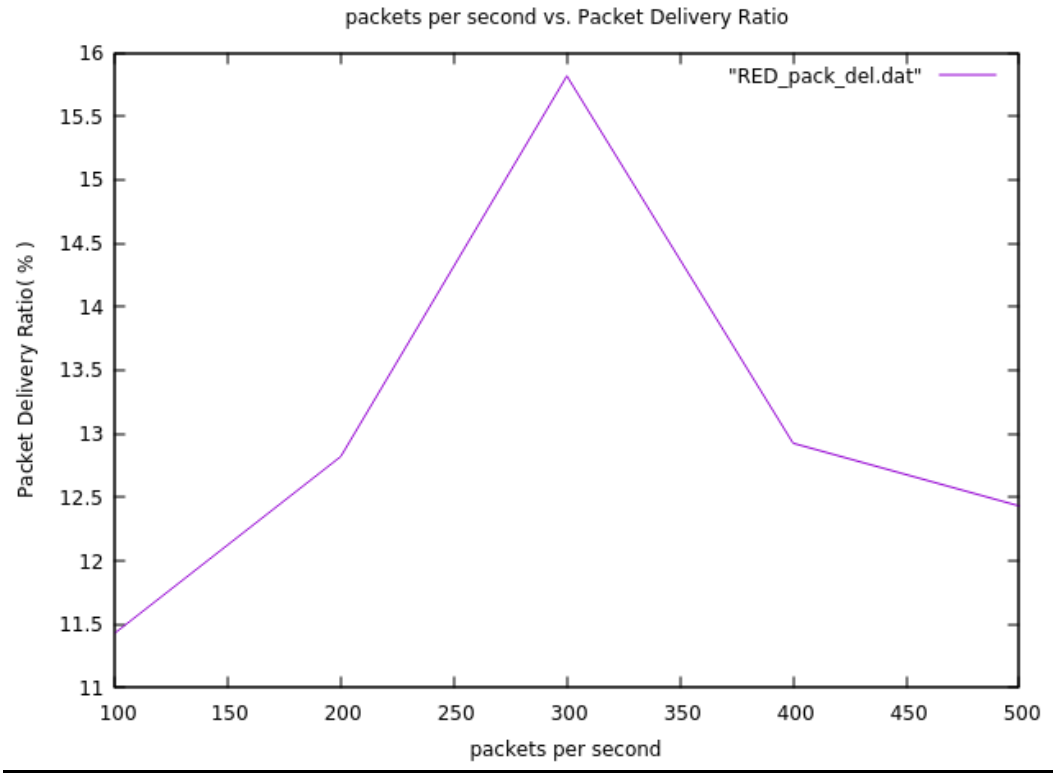
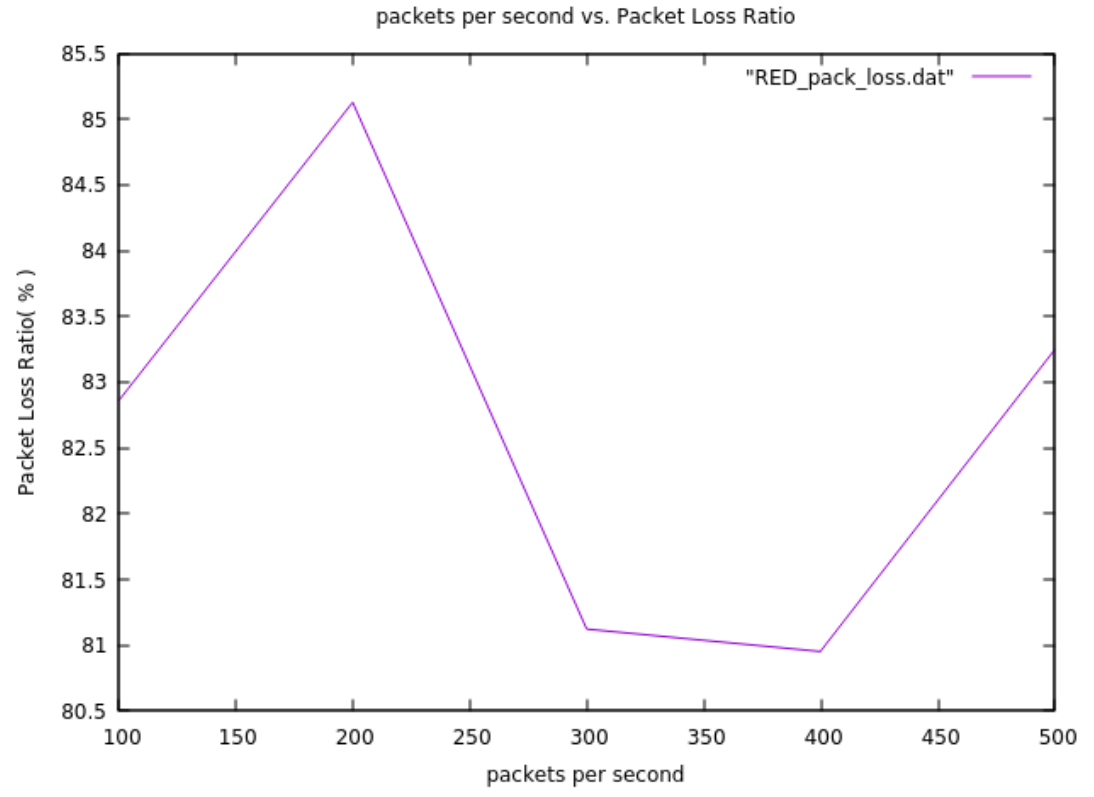


Fig: varying number of flows(wireless)


packets per second vs. End-to-End Delay
"RED_delay.dat"
End-to-End Delay(ms)
packets per second
1300
1200
1100
1000
900
800
700
600
500
400
100
150
200
250
300
350
400
450
500


packets per second vs. Packet Delivery Ratio
"RED_pack_del.dat"
Packet Delivery Ratio(%)
packets per second
16
15.5
15
14.5
14
13.5
13
12.5
12
11.5
11
100
150
200
250
300
350
400
450
500



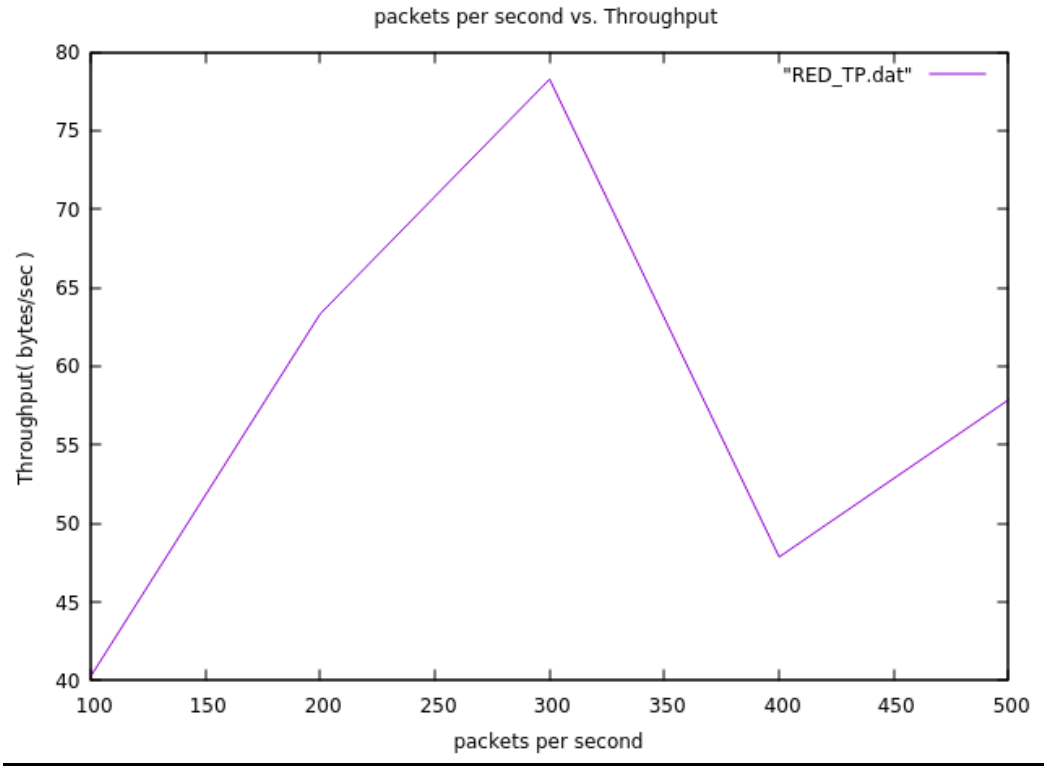
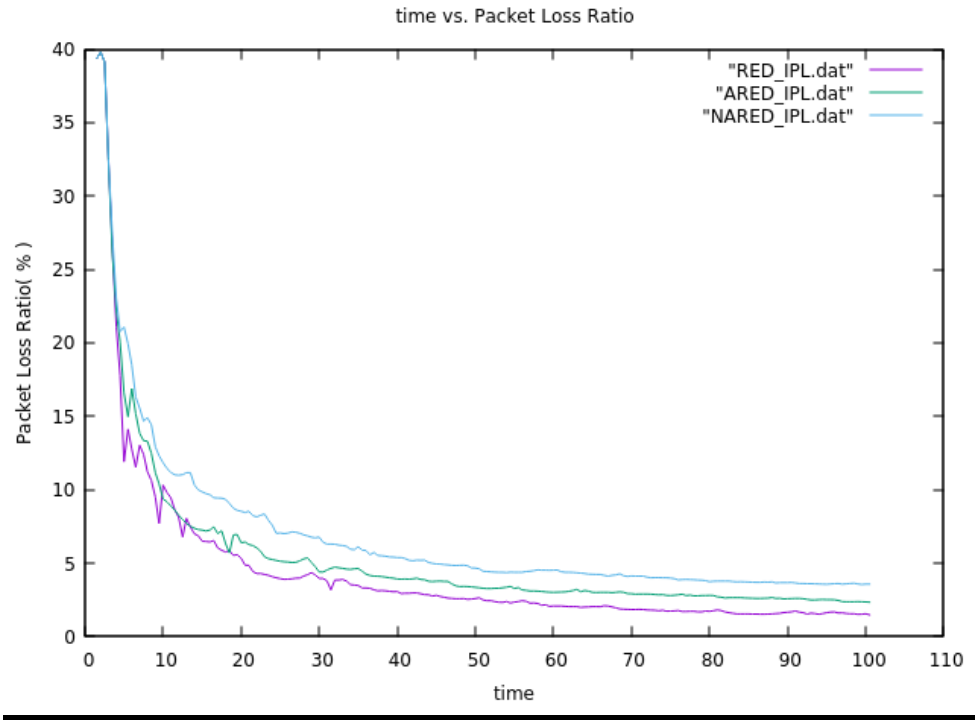


Fig: varying packets per second (wireless)

Results with Graphs (TASK B):



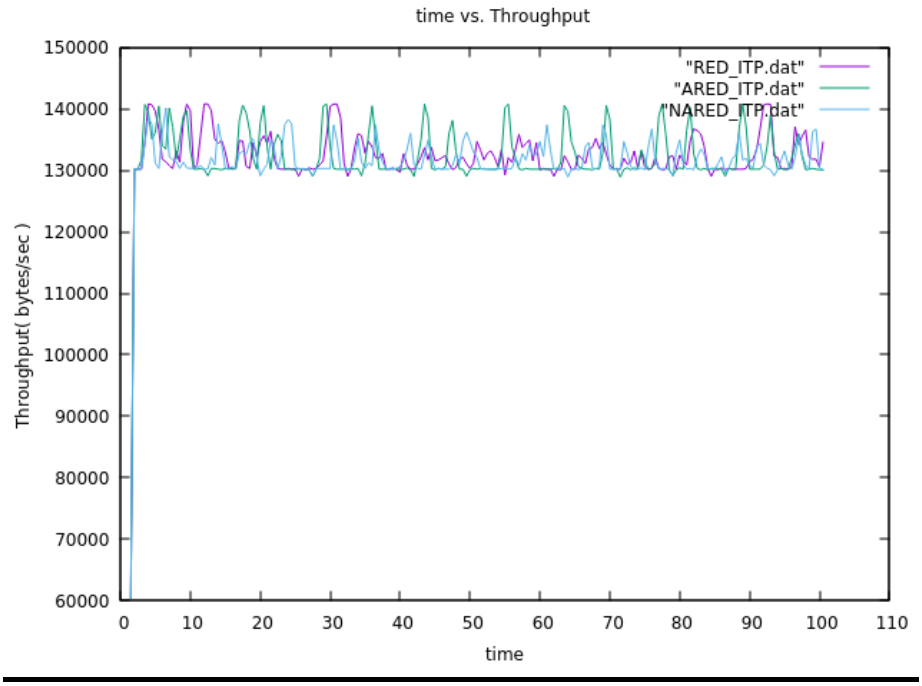
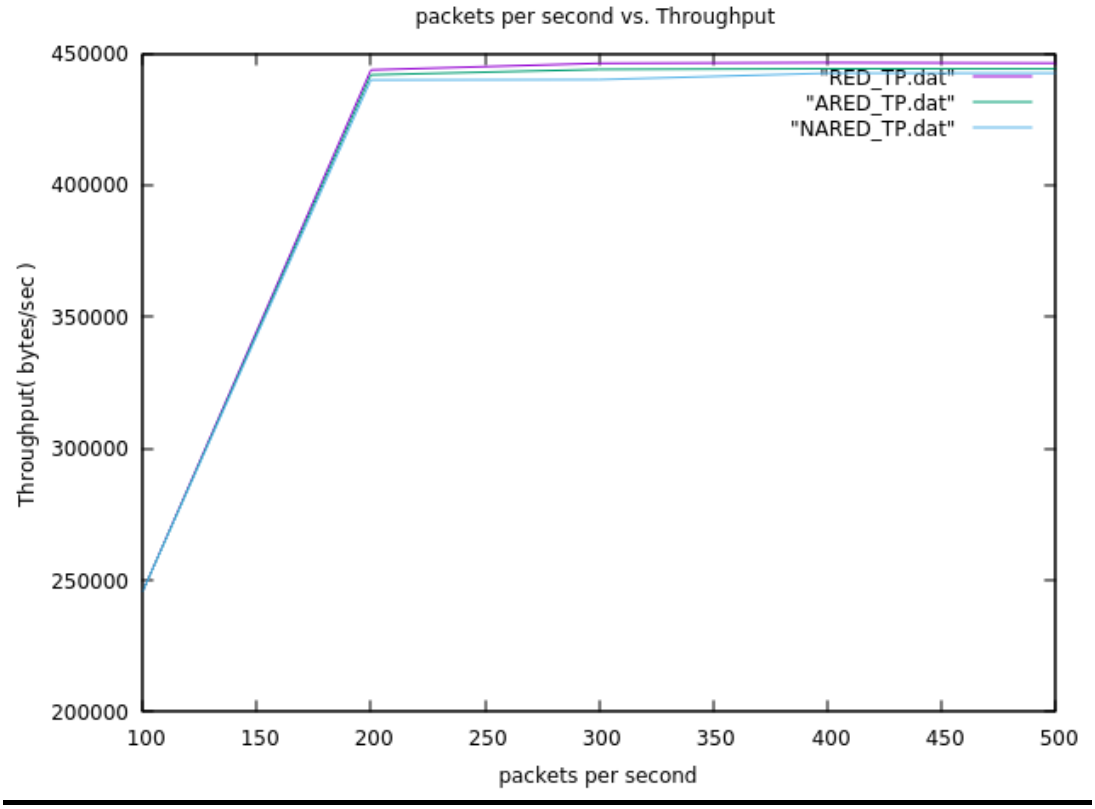
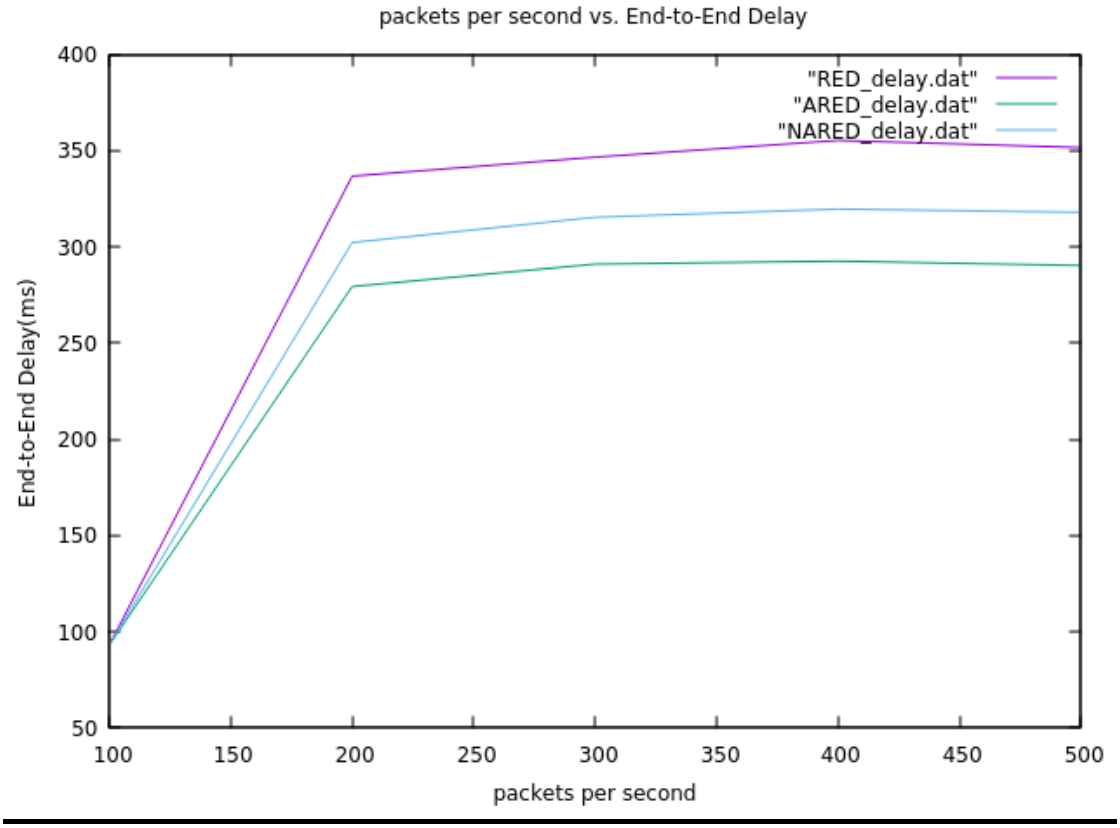
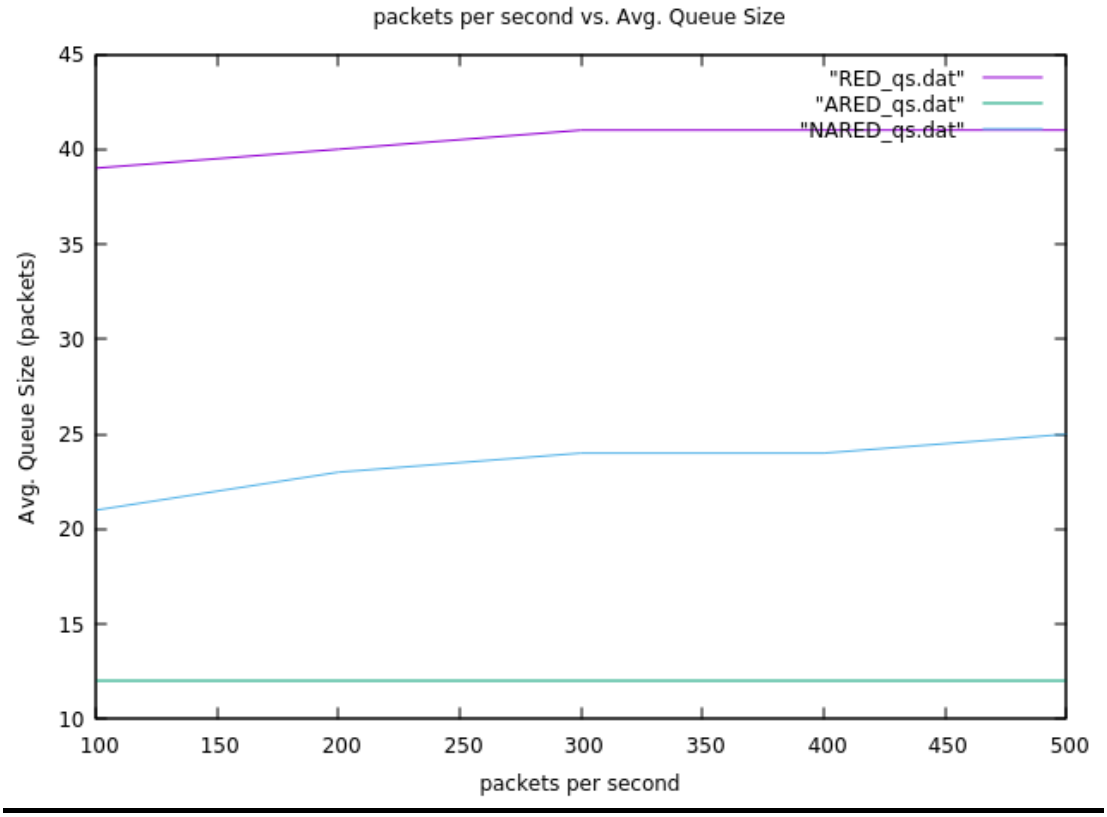
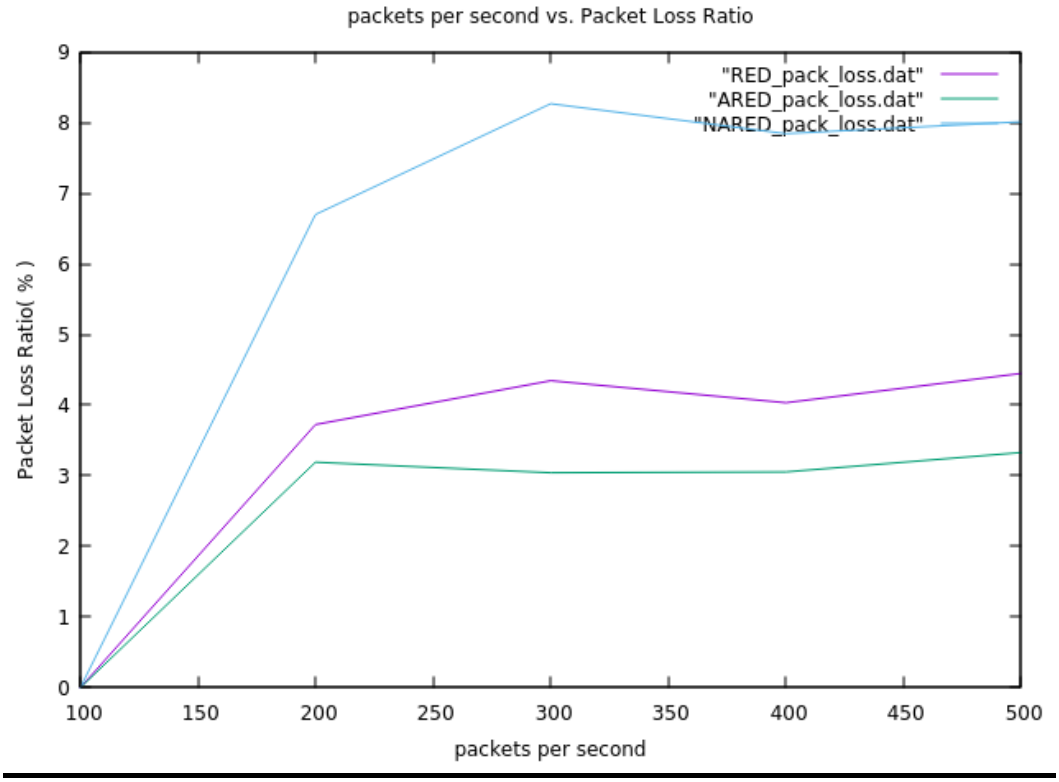


Fig: Instantaneous throughput and packet drop ratio


packets per second vs. Throughput
"RED_TP.dat"
"ARED_TP.dat"
"NARED_TP.dat"
Throughput(bytes/sec)
packets per second
450000
400000
350000
300000
250000
200000
100
150
200
250
300
350
400
450
500


packets per second vs. End-to-End Delay
"RED_delay.dat"
"ARED_delay.dat"
"NARED_delay.dat"
End-to-End Delay(ms)
packets per second
400
350
300
250
200
150
100
50
100
150
200
250
300
350
400
450
500


packets per second vs. Avg. Queue Size
"RED_qs.dat"
"ARED_qs.dat"
"NARED_qs.dat"
Avg. Queue Size (packets)
packets per second
45
40
35
30
25
20
15
10
100
150
200
250
300
350
400
450
500


packets per second vs. Packet Loss Ratio
"RED_pack_loss.dat"
"ARED_pack_loss.dat"
"NARED_pack_loss.dat"
Packet Loss Ratio(%)
packets per second
9
8
7
6
5
4
3
2
1
0
100
150
200
250
300
350
400
450
500

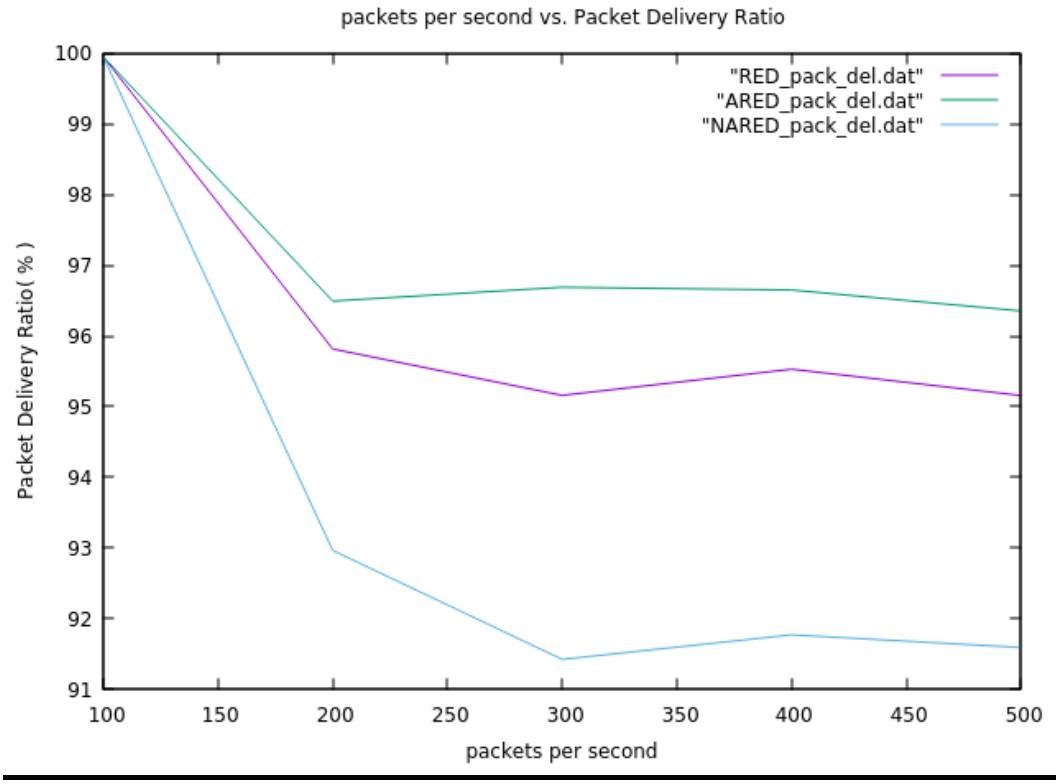


Fig: varying packets per second

Summary Findings:

TASK A:

From task A, we can see from the graphs generated in the wired topology that RED and ARED performs pretty similar, but in the metrics of average end to end delay, ARED performs a bit better with delay 10ms or so less than RED. I think that, the adjustment of maximum packet dropping probability Pmax plays a role here. Also, if we look at the metric of average network throughput, we can see that with the increase of flows in the inter-network, the throughput increases but it will reach a saturation point which will be equal to bandwidth of the bottleneck link. On the contrary, we can see a downward trend in the

metric of packet drop ratio. Therefore, we can see an inverse relationship between throughput and packet drop ration here. Again, in the wireless topology, we can see the packet drop ratio is significantly more in wireless than in wired network. The reason for it is I think down to the “RangePropagationLossModel” installed in the wireless channel which basically means the farther a node is, the more the loss. This channel behaviour may result in the randomness noticed in the generated plots. Also analyzing the flows generated by the flow monitor, it can be seen some nodes in the wifi network receive significantly less packets than other wifi nodes and that can be due to the distance of these nodes from the router node.

TASK B:

In Task B, we set all the parameters of RED queue disc according to the simulation shown in paper. Here, we compare three AQM variations of RED algorithm: the original RED algorithm, ARED and our modified algorithm NARED. Here, if we see the plots, we can see that though some of the claims made in the referenced paper go in parallel with the results, some clearly differ. If we look at the plot of the average network throughput, we can see that, all of these three shows pretty similar result. In the plot of average end-to-end delay of packets, we can see that ARED result in the least delay. NARED performs better than the original RED but still falls short of ARED. We know that, NARED does not increase the packet dropping probability right away rather increases in non-linearly. Therefore, it is expected that it will result in higher average queue size than the other algorithms. But from the simulation results, we can see that though it results in higher average queue size than the ARED algorithm, it still results in lower average queue size than the original RED algorithm. The instantaneous

throughput plot shows that, all of the three algorithms perform pretty similar again. But the aspect that the simulation result differs from the claims made in the paper is that the packet drop ratio is expected to be lower than the other two algorithms but in this case, it is actually a bit higher. One possible reason for this maybe packet retransmission. Packet retransmission is the re-sending of packets that have been damaged or lost during their initial transmission. NARED not showing expected packet drop ratio maybe be because in the other two algorithms, initial packet drop may cause packet retransmission, which lead to re-sending the packets and corrupting the packet drop ratio calculation.

These are all the findings I came across going through the simulation results.