NS3 Project Report

Sihat Afnan
Student Id. : 1705098

February , 2022

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
(BUET)
Dhaka - 1000

Contents

List of Figures

1 Introduction

NS3 is an open-source event-driven simulator designed specifically for research in computer communication networks. In this project, I have to vary different parameters such as number of nodes, number of flows, number of packets per second and measure some metrics and plot them in some graphs.
I also had to modify some mechanisms of Retransmission Timeout calculation and compare the modified version with the original implementation of NS3.The paper from which I have taken idea from is The Peak-Hopper: A New End-to-End Retransmission Timer for Reliable Unicast Transport

2 Simulated Networks

- Wireless 802.11 (static) network using AODV routing in Random topology
- Wireless 802.15.4 (static) network using Ripng routing in Random Topology
- Dumbbell topology consisting of same number of senders and receivers and two routers for showing RTT and RTO modification.

3 Parameters and Metrics

3.1 Parameters

1. Number of nodes (20, 40, 60, 80, and 100)
2. Number of flows (10, 20, 30, 40, and 50)
3. Number of packets per second (10, 20, 30, 40, and 50)
4. Coverage Area (1000 , 1500 , 2000 , 2500 ,2800 m^2)

3.2 Metrics

1. Network Throughput
2. End-to-End Delay
3. Packet Delivery Ratio
4. Packet Drop Ratio

4 Overview of the proposed algorithm

The PeakHopper algorithm essentially runs two RTO algorithms in parallel.One algorithm (Short-Term History RTO) monitors the present and short-term history in order to respond to RTT increases. The other algorithm(Long-Term History RTO) simply decays the current value of RTO, and can therefore be said to represent the long-term history.

$$\delta = \frac{RTT_{sample} - RTT_{previous}}{RTT_{previous}} \qquad \text{(Step 1)}$$

$$D = 1 - \frac{1}{F * S} \qquad \text{(Step 2)}$$

$$B \leftarrow \max(\delta, D * B) \qquad \text{(Step 3)}$$

$$RTT_{\max} = \max(RTT_{sample}, RTT_{previous}) \qquad \text{(Step 4)}$$

$$RTO \leftarrow \max(D * RTO, (1 + B) * RTT_{\max}) \qquad \text{(Step 5)}$$

$$RTO \leftarrow \max(RTO, RTO_{\min}) \qquad \text{(Step 6)}$$

Figure 1: Algorithm Overview

1. Having collected a new RTT sample, RTTsample, we compare this value to the previous RTT sample collected, RTTprevious, as shown in Step 1. We call the normalized change between these two samples delta. This is the measure of the short-term changes in RTT.

2. In Step 2, we further define a decay factor, D. D determines how rapidly the RTO is decayed. We also introduce a fader variable, F, which controls the speed of this decay (a high F gives a slow decay and vice versa).

3. In Step 3, we calculate a booster variable B. The booster variable determines how high the RTO should hop when a large RTT increase has been detected.

4. In Step 4, we set RTTmax to the maximum of the new RTT sample, RTTsample, and the previous RTT sample, RTTprevious. RTTmax is used to represent the short-term history of the RTT.

5. In step 5, We set RTO to the maximum of a long-term history (represented by the term D*RTO) and the short-term history (represented by the term ((1+B)*RTTmax).

6. In final step (Step 6), we ensure that the RTO does not fall below the minimum allowed RTO.

Metrics used in the paper

2) Waiting-Time

We call a timeout "necessary" if no 3rd DUPACK has arrived at the sender by the time that an acceptable ACK for the retransmitted segment arrives at the sender.

In order to assess the promptness of the retransmission timer, we define the normalized waiting time for the *i*:th necessary timeout in a simulation run as

$$W_i = \frac{RTO_i}{RTT_i}.$$

RTO_i is the value with which the retransmission timer was started for the retransmitted segment, and RTT_i is the last RTT measurement before the timeout. We calculate

Figure 2: Metric 1

the mean normalized waiting time for a necessary timeout as

$$W = \frac{1}{M} \sum_{i=1}^{M} W_i \cdot$$

M is the total number of necessary timeouts in a particular simulation run. Expressed in words, W measures the mean time that the retransmission timer waited until it expired, thereby producing a necessary timeout, expressed in multiples of the last RTT measurement prior to the timeout.

We define $W_{95\%}$ as the value for which 95% of all W_i values collected in a simulation run are smaller than $W_{95\%}$. This metric reveals how long the waiting time is for the retransmissions that had to wait the long for the retransmission timer to expire. In effect, this is a predictability metric: a high $W_{95\%}$ reveals that the retransmission timer in some cases waits excessively long to retransmit, while a low $W_{95\%}$ suggests that the retransmission timer made all its necessary retransmissions in a narrow spread of waiting time.

Figure 3: Metric 1

3) Other Metrics

For simulation runs where none or only a few timeout-events were observed, we cannot fairly judge the performance of the retransmission timers based on the accuracy and promptness metrics introduced above. Hence, we need another metric to judge the performance of the retransmission timer.

In this case, we chose use the proximity of the RTO to the RTT as a performance metric. This was calculated as RTO_N/RTT_L where RTO_N is the newly calculated RTO and RTT_L is the last RTT sample prior to the RTO update. We then calculated *P* as the mean value of this fraction over an entire simulation run.

Obviously, when comparing the performance of Retransmission Timer A with Retransmission Timer B for a certain simulation run, Retransmission Timer A can be said to have performed better than Retransmission Timer B if its P is lower (i.e. the RTO was in mean closer to the RTT), provided none of the timers caused any (or only a few) timeouts during that simulation.

Figure 4: Metric 2

5 Modifications

I have modified the RTT and RTO calculation algorithm in NS3.The relevant files were rtt-estimator.cc , rtt-estimator.h , tcp-socket-base.cc , tcp-socket-base.h Apart from these , I have written a simulation script simulate.cc to generate the topology and run my modified algorithm upon it.

1. **rtt-estimator.cc:** Inside RttMeanDeviation::Measurement (Time m) function,

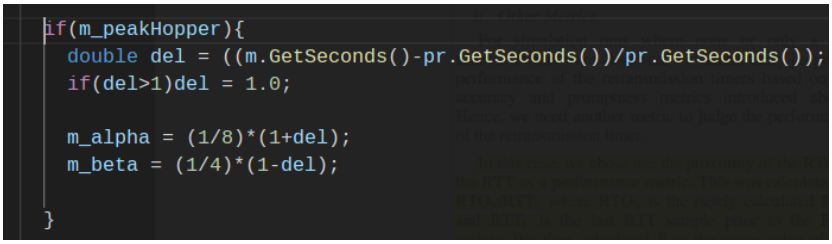


Figure 5: rtt-estimator.c modification

2. **tcp-socket-base.h** Declared necessary flag and trace values for calculation of metrics proposed in the paper.

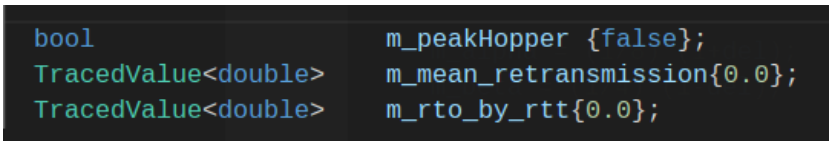


Figure 6: tcp-socket-base.h modification

2. **tcp-socket-base.cc** There are several modifications in this file.Following are the added attributes and trace sources added in tcp-socket-base class.

```
.AddAttribute ("m_peakHopper", "Enable or disable peakHopper option",
               BooleanValue (false),
               MakeBooleanAccessor (&TcpSocketBase::m_peakHopper),
               MakeBooleanChecker ())
.AddAttribute("no_of_retransmit" , "count of how many retransmit",
               UintegerValue (0),
               MakeUintegerAccessor (&TcpSocketBase::no_of_retransmit),
               MakeUintegerChecker<uint32_t> ())
.AddAttribute("rto_by_rtt" , "mean of RTO_n by RTT_l" ,
              DoubleValue(0.0) ,
              MakeDoubleAccessor (&TcpSocketBase::m_rto_by_rtt),
              MakeDoubleChecker<double> ())
.AddAttribute("m_mean_retransmission" , "mean of rto/rtt at retransmission" ,
              DoubleValue(0.0) ,
              MakeDoubleAccessor (&TcpSocketBase::m_mean_retransmission),
              MakeDoubleChecker<double> ())
```

Figure 7: Added attributes

```
.AddTraceSource ("m_mean_retransmisionTrace",
                 "mean of rto/rtt at retransmission time",
                 MakeTraceSourceAccessor (&TcpSocketBase::m_mean_retransmission),
                 "ns3::TracedValueCallback::Time")
.AddTraceSource ("rto_by_rttTrace",
                 "rto by rtt",
                 MakeTraceSourceAccessor (&TcpSocketBase::m_rto_by_rtt),
                 "ns3::TracedValueCallback::Time")
```

Figure 8: Added trace sources

In the TcpSocketBase::EstimateRtt (const TcpHeader tcpHeader) function , following modification has been made when peakHopper flag is enabled

```
if(m_peakHopper){
  m_rtt->Measurement (m);
  double del = ((m.GetSeconds() - lastRtt.GetSeconds())/lastRtt.GetSeconds());
  double S = 1;
  double F = 16;
  double D = 1-(1/F*S);
  if(B*D > del)B = B*D;
  else B = del;

  Time rttMax;
  if(m.GetSeconds() > lastRtt.GetSeconds())rttMax = m;
  else rttMax = lastRtt;

  Time tmp;
  if(D*(m_rto.Get()).GetSeconds() > (1+B)*rttMax.GetSeconds())tmp = D*m_rto;
  else tmp = (1+B)*rttMax;

  m_minRto = rttMax + 2*m_clockGranularity;
  lastRtt = m_rtt->GetEstimate ();
  m_rto = Max(D*tmp , (1+B)*rttMax);
  //m_rto = Max (m_rtt->GetEstimate () + Max (m_clockGranularity, m_rtt->GetVariation ()
  m_rto = Max(m_rto , m_minRto);
  cnt_m_rto_update++;
  ratio += (m_rto.Get().GetSeconds() / lastRtt.GetSeconds());
  m_rto_by_rtt = ratio / (cnt_m_rto_update*1.0); //change of this value will invoke rto_
  NS_LOG_UNCOND(m_rto_by_rtt);
```

Figure 9: estimateRtt method modification

The next piece of modification is common for both TcpSocketBase::NewAck (SequenceNumber32 const ack, bool resetRTO) and TcpSocketBase::SendEmptyPacket (uint8_t flags) functions.

```
//----------------added by afnan-----------------------------
if(m_peakHopper){
  m_rto = Max(m_rtt->GetEstimate ()
  +(Time::FromDouble (m_rtt->GetVariation ().ToDouble (Time::S) ,
   Time::S)), lastRtt + 2*m_clockGranularity);
  NS_LOG_INFO("Its peakHopper-----------------------------------------
}
else{
  m_rto = Max (m_rtt->GetEstimate ()
  + Max (m_clockGranularity, m_rtt->GetVariation () * 4), m_minRto);
  NS_LOG_INFO("Not peakHopper-----------------------------------------
}
```

Figure 10: NewAck and SendEmptypacket method modification

Following modification was done in TcpSocketBase::DoPeerClose (void) function while updating the lastRtt value

```
//---------------added by afnan-------------------------
Time lastRto;
if(m_peakHopper){
  lastRto = Max(m_rtt->GetEstimate ()
   +(Time::FromDouble (m_rtt->GetVariation ().ToDouble (Time::S),
    Time::S)), lastRtt + 2*m_clockGranularity);
  NS_LOG_INFO("Its peakHopper-----------------------------------
}
else{
  lastRto = m_rtt->GetEstimate ()
  + Max (m_clockGranularity, m_rtt->GetVariation () * 4);
  NS_LOG_INFO("Not peakHopper-----------------------------------
}
```

Figure 11: DoPeerClose method modification

Finally to calculate mean retransmission , following piece of code was added at the end of DoRetransmit() function.

```
//-----------------added by afnan-------------------------
total_retransmit+=1;
//get lastRtt and m_rto to generate sumof W/M
w += (m_rto.Get().GetDouble()/lastRtt.GetDouble());
//NS_LOG_UNCOND("No of Retransmitted packets : "<<total_retransmit);
m_mean_retransmission = w*1.0 / total_retransmit;
NS_ASSERT (sz > 0);
//------------------------------------------------------
```

Figure 12: mean retransmission calculation

6 Comparison

6.1 RTT RTO modification

Network Topology Recap

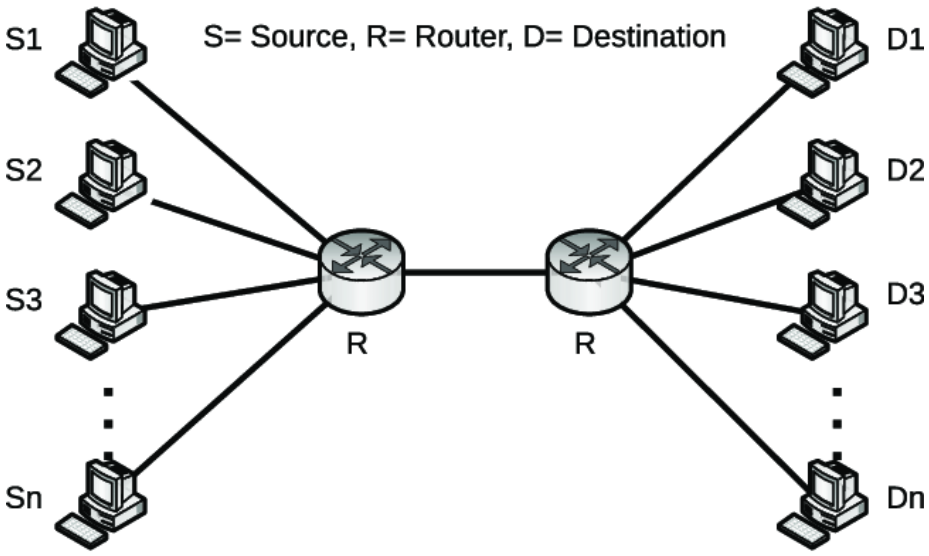


Figure 13: DumbBell Topology

6.1.1 Access delay of 130ms and 45ms , shared delay 35ms

RTT and RTO

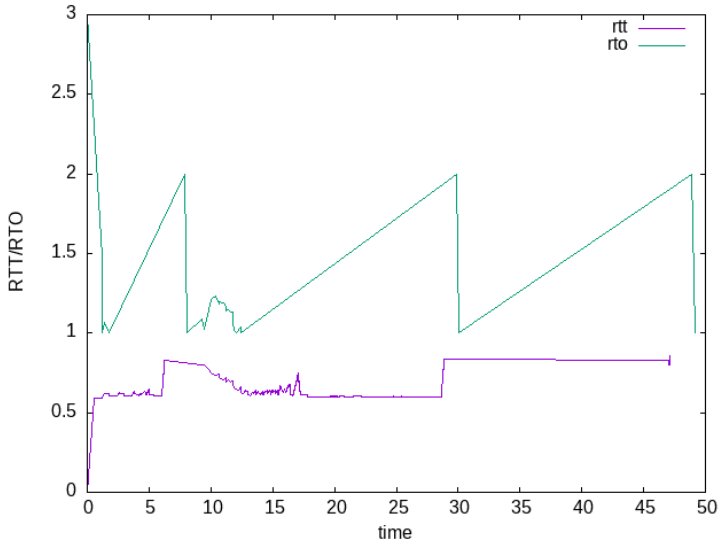


Figure 14: Default RTT RTO graph

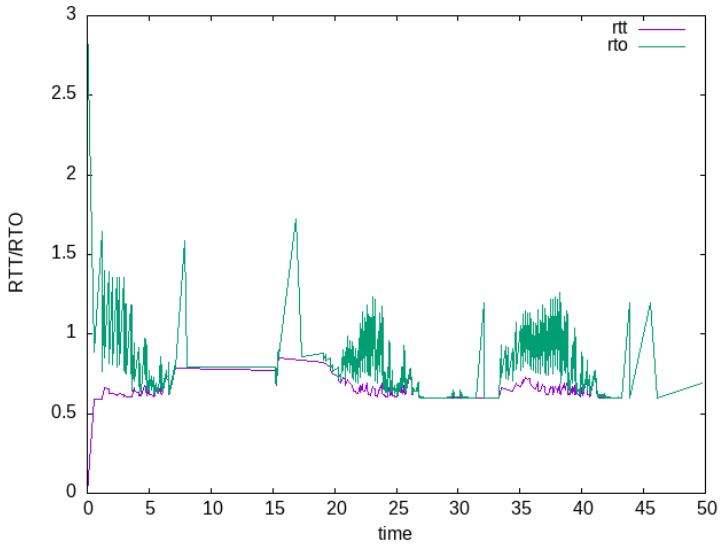


Figure 15: Modified RTT RTO graph

Mean Retransmission

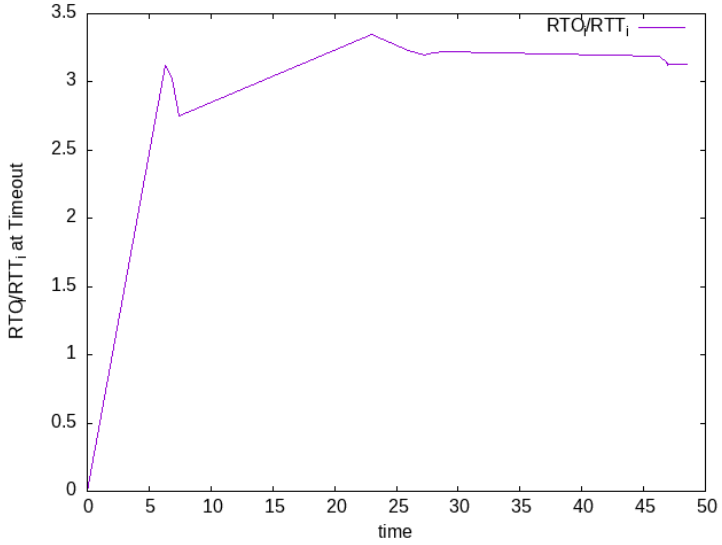


Figure 16: Default mean retransmission graph

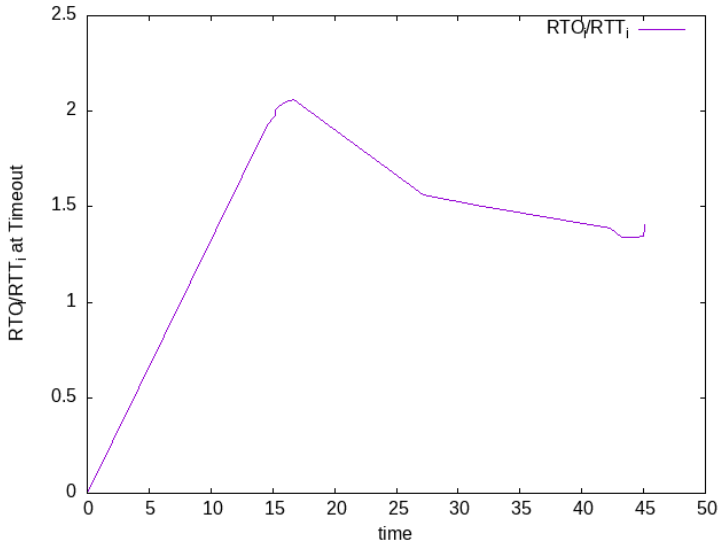


Figure 17: Modified mean retransmission graph

RTO_i/RTT_i **ratio**

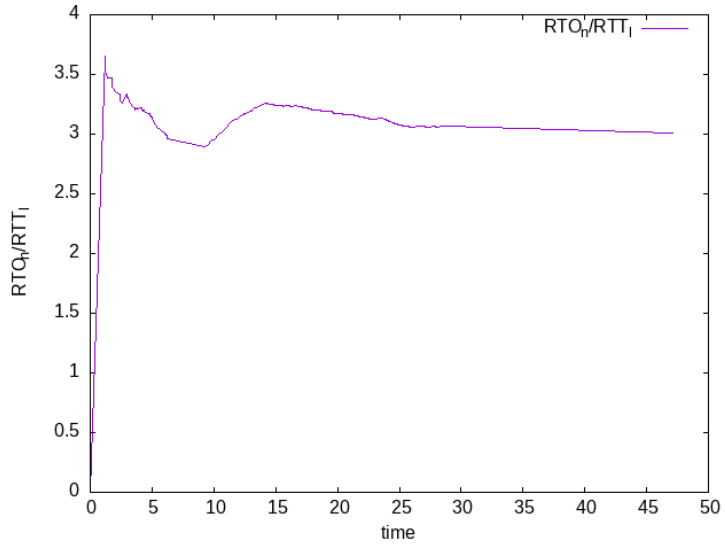


Figure 18: Default rto/rtt graph

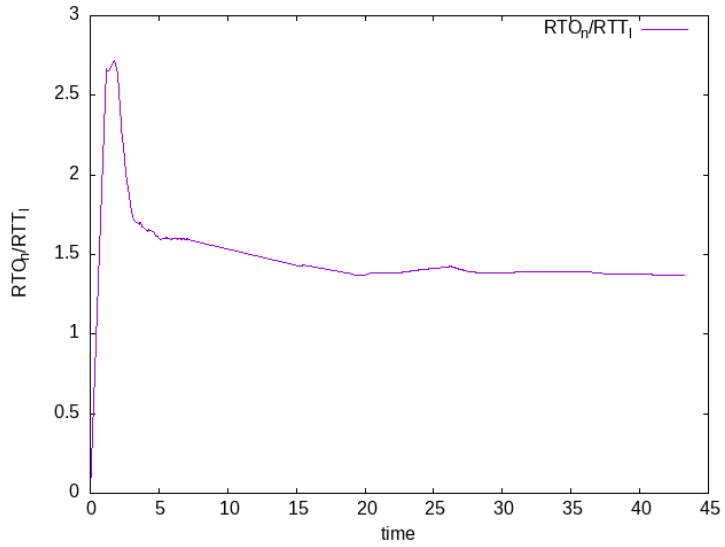


Figure 19: Modified rto/rtt graph

Congestion Window

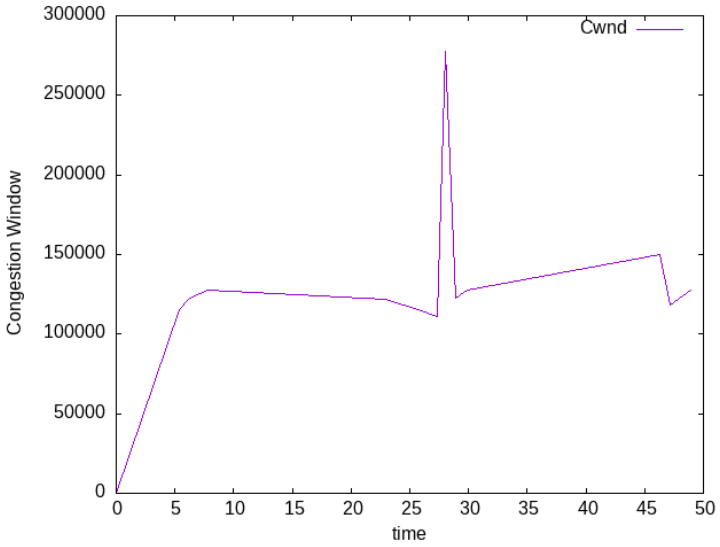


Figure 20: Default cwnd graph

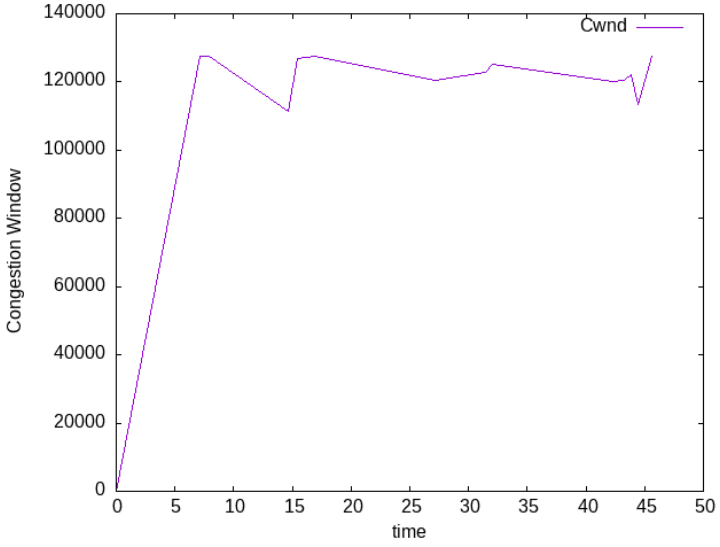


Figure 21: Modified cwnd graph

Slow Start Threshold

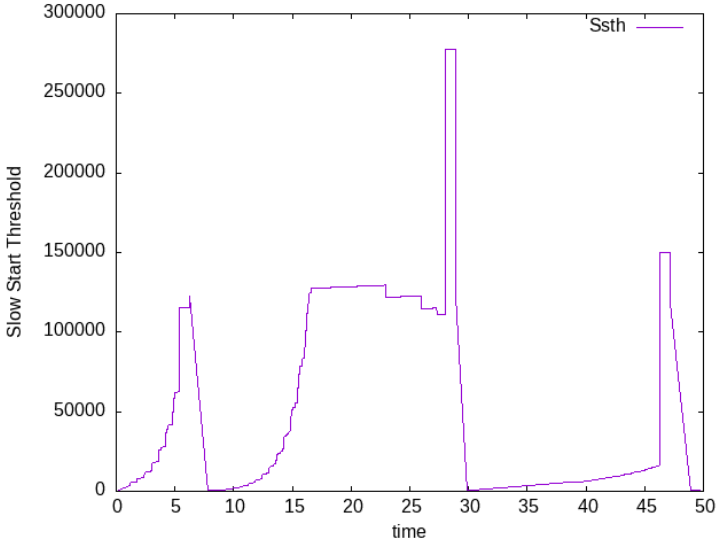


Figure 22: Default ssth graph

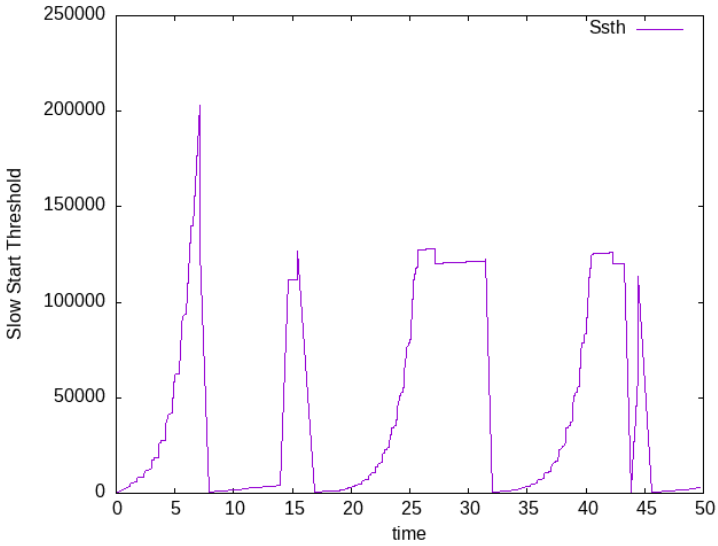


Figure 23: Modified ssth graph

6.1.2 access delay of 140 ms and 50 ms, shared delay 40 ms

RTT and RTO

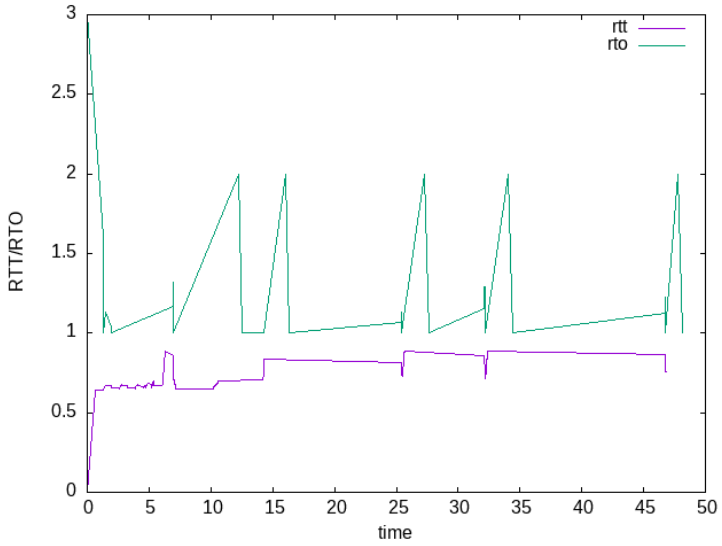


Figure 24: Default RTT RTO graph

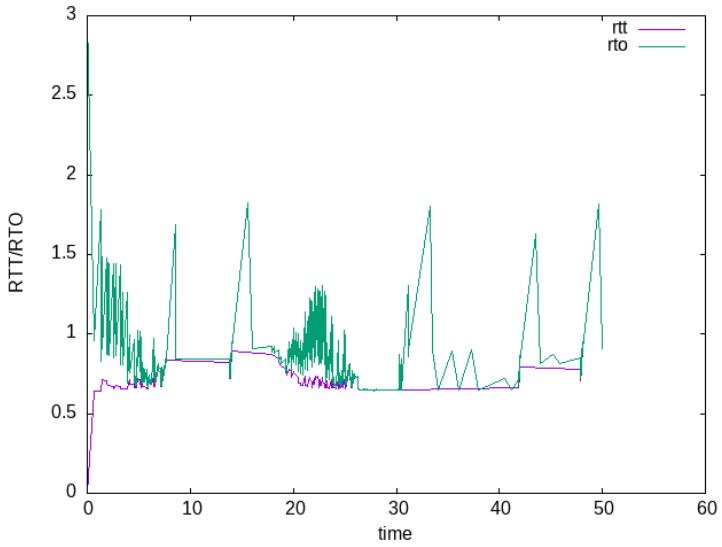


Figure 25: Modified RTT RTO graph

Mean Retransmission

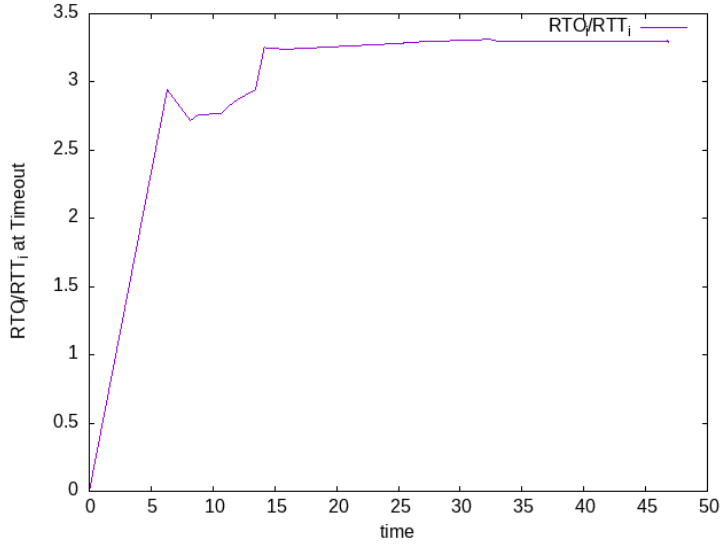


Figure 26: Default mean retransmission graph

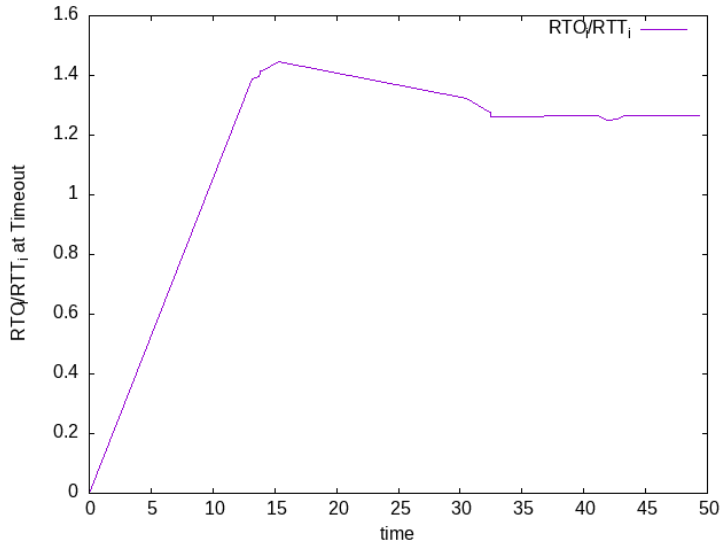


Figure 27: Modified mean retransmission graph

RTO_i/RTT_i **ratio**

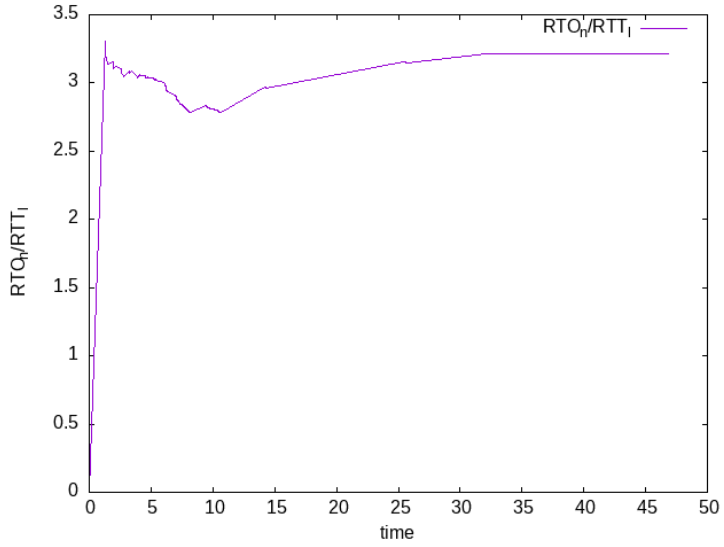


Figure 28: Default rto/rtt graph

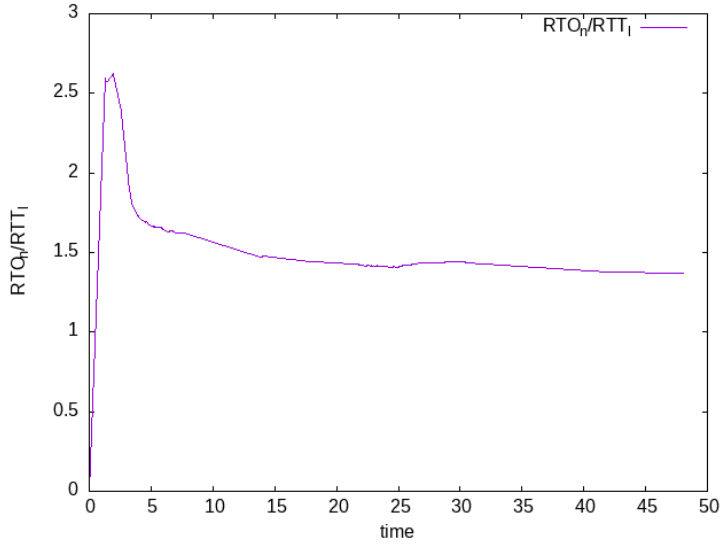


Figure 29: Modified rto/rtt graph

Congestion Window

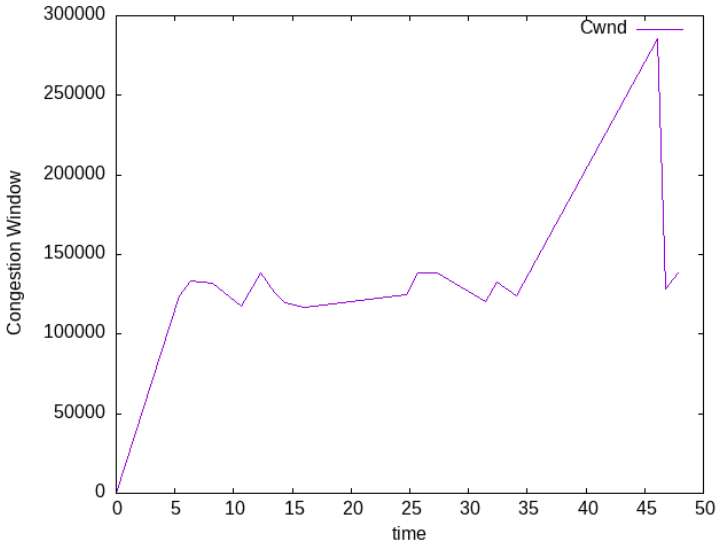


Figure 30: Default cwnd graph

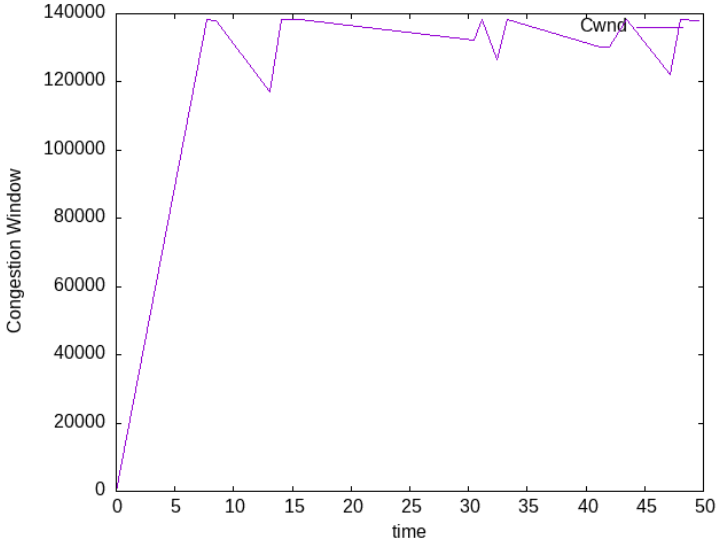


Figure 31: Modified cwnd graph

Slow Start Threshold

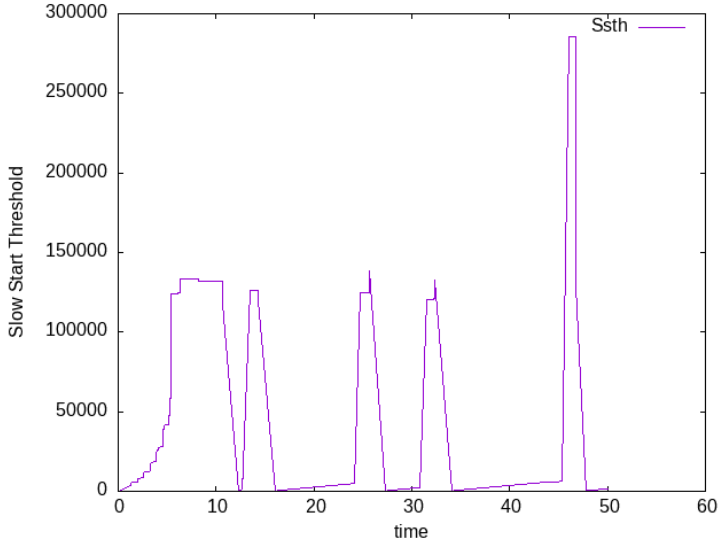


Figure 32: Default ssth graph

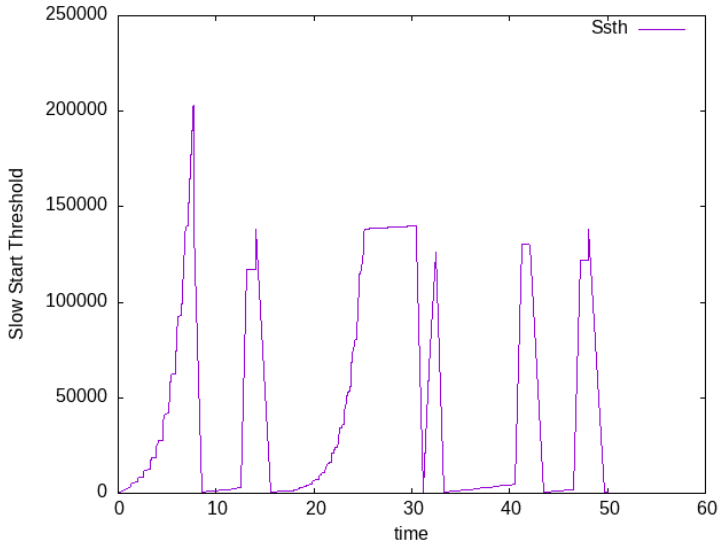


Figure 33: Modified ssth graph

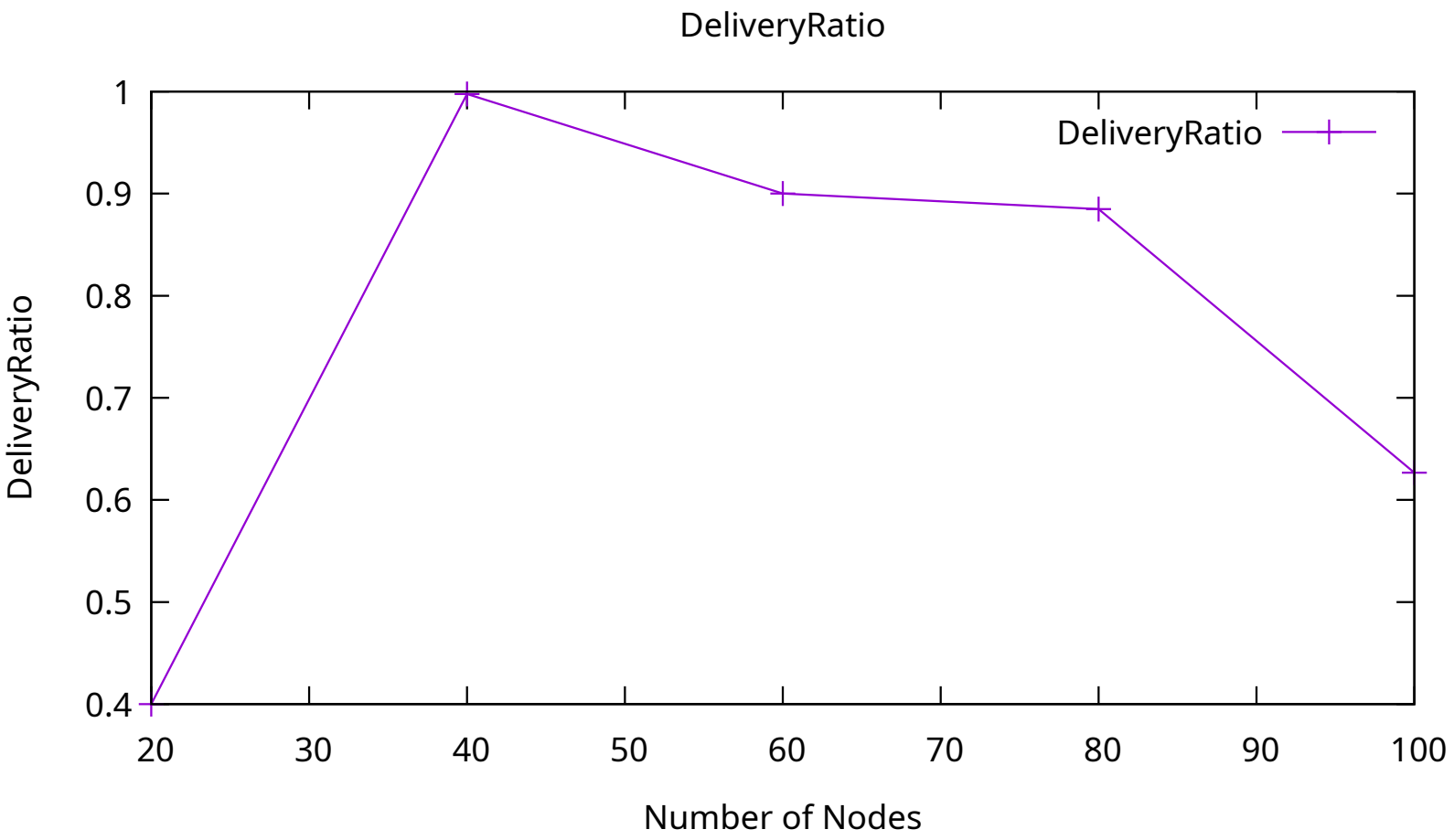
6.2 Graphs of 802.11 and 802.15.4 protocol

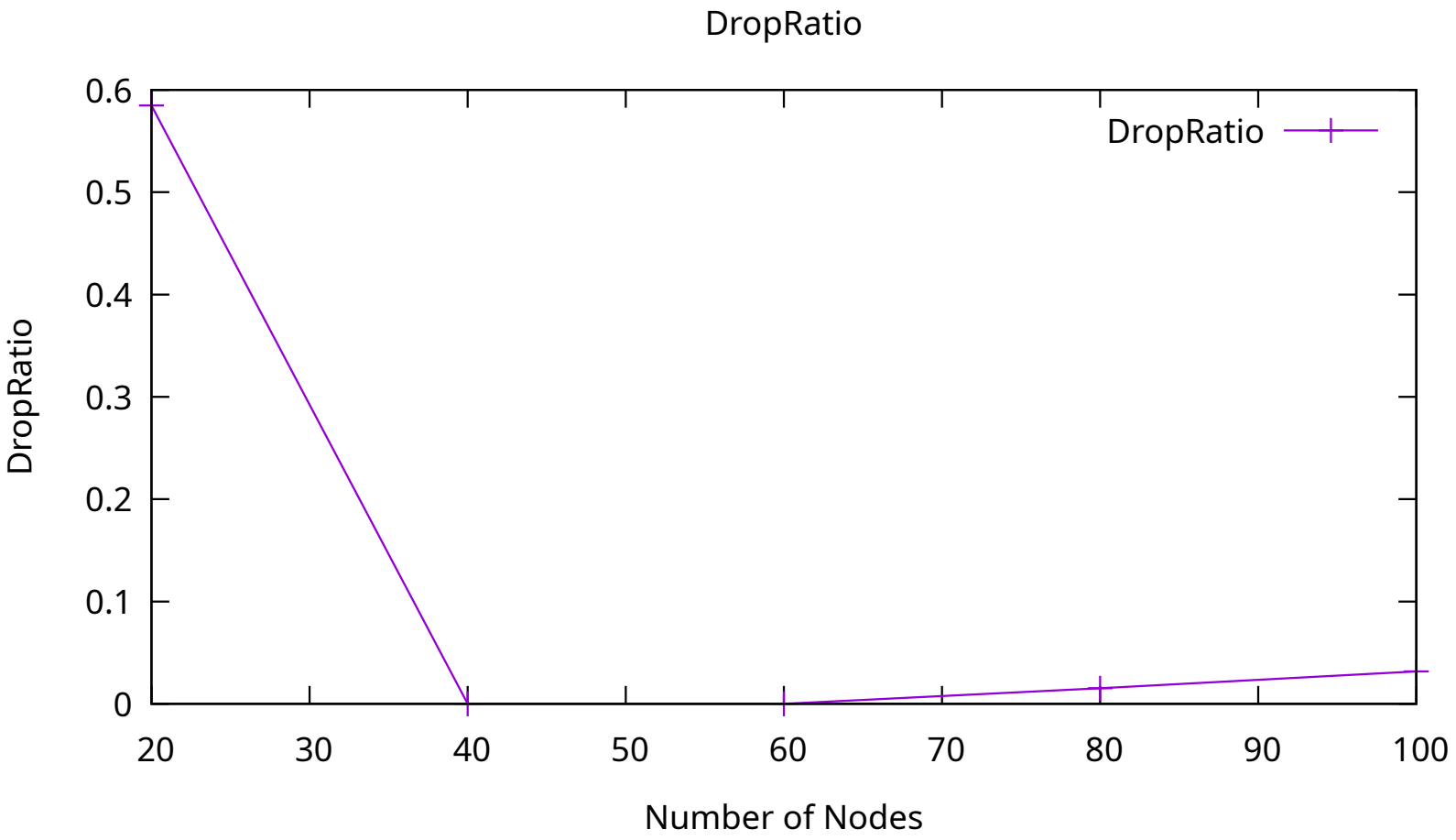
6.2.1 802.11 highrate static network

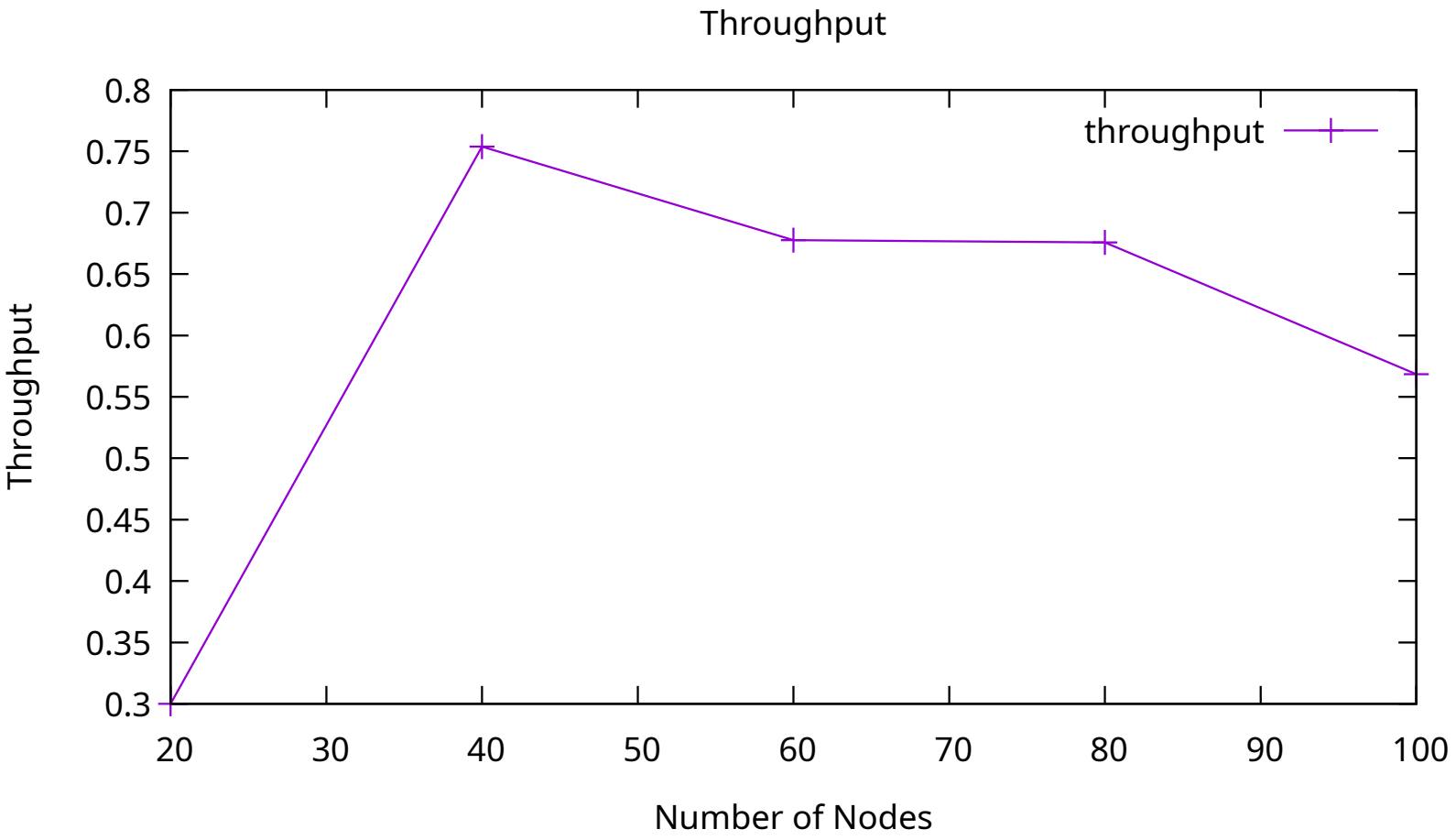
Network Topology Recap

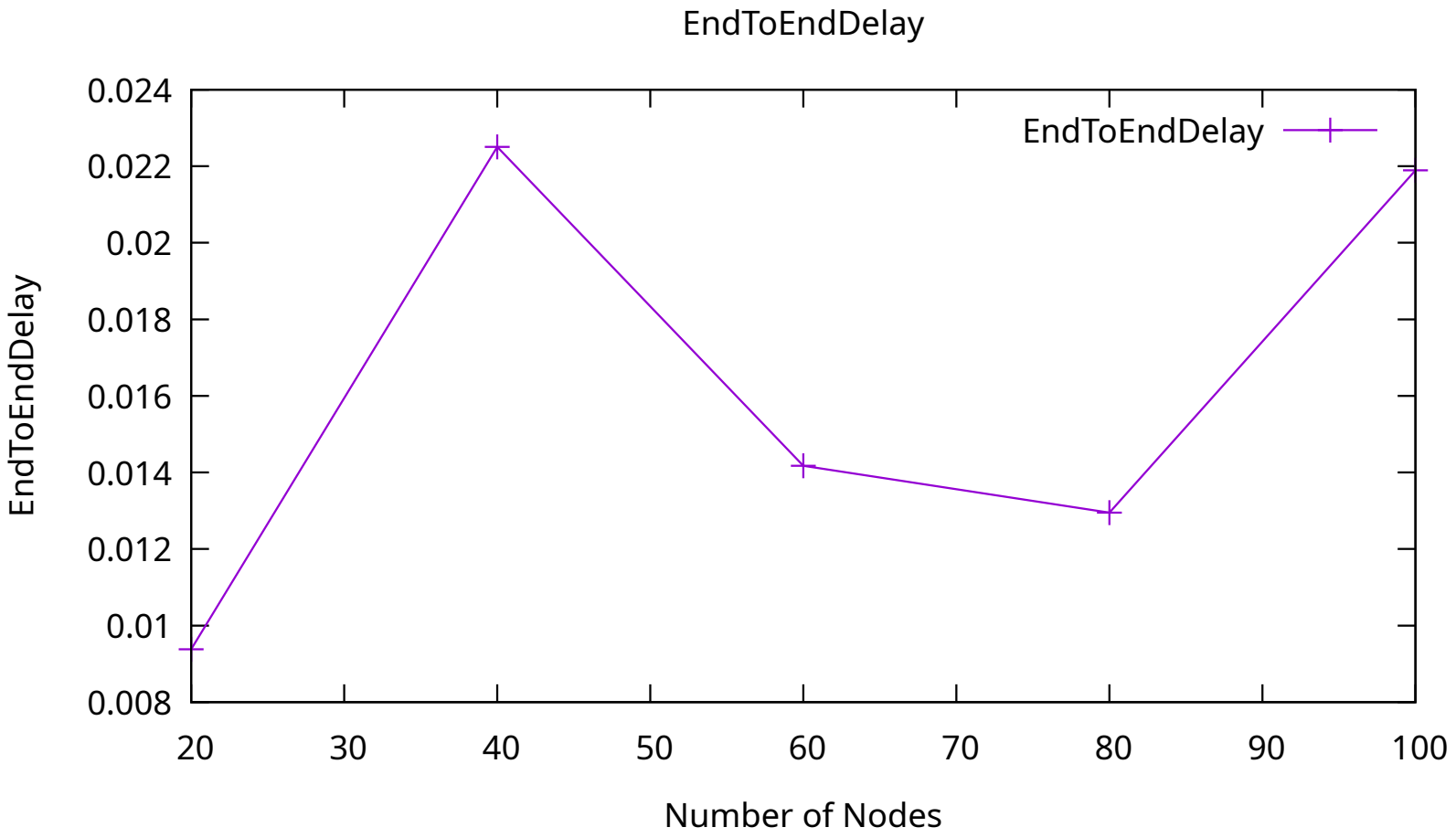
- **Propagation Model** ConstantSpeedPropagationDelayModel
- **Propagation Loss Model** RangePropagationLossModel
- **Mac**] AdhocWifiMac
- **Mac Standard** WIFI_STANDARD_80211b
- **Routing Protocol** AODV
- **Application Layer** OnoffHelper/ns3::UdpSocketFactory
- **Mobility Model** ConstantPositionMobilityModel

Varying Number of Nodes

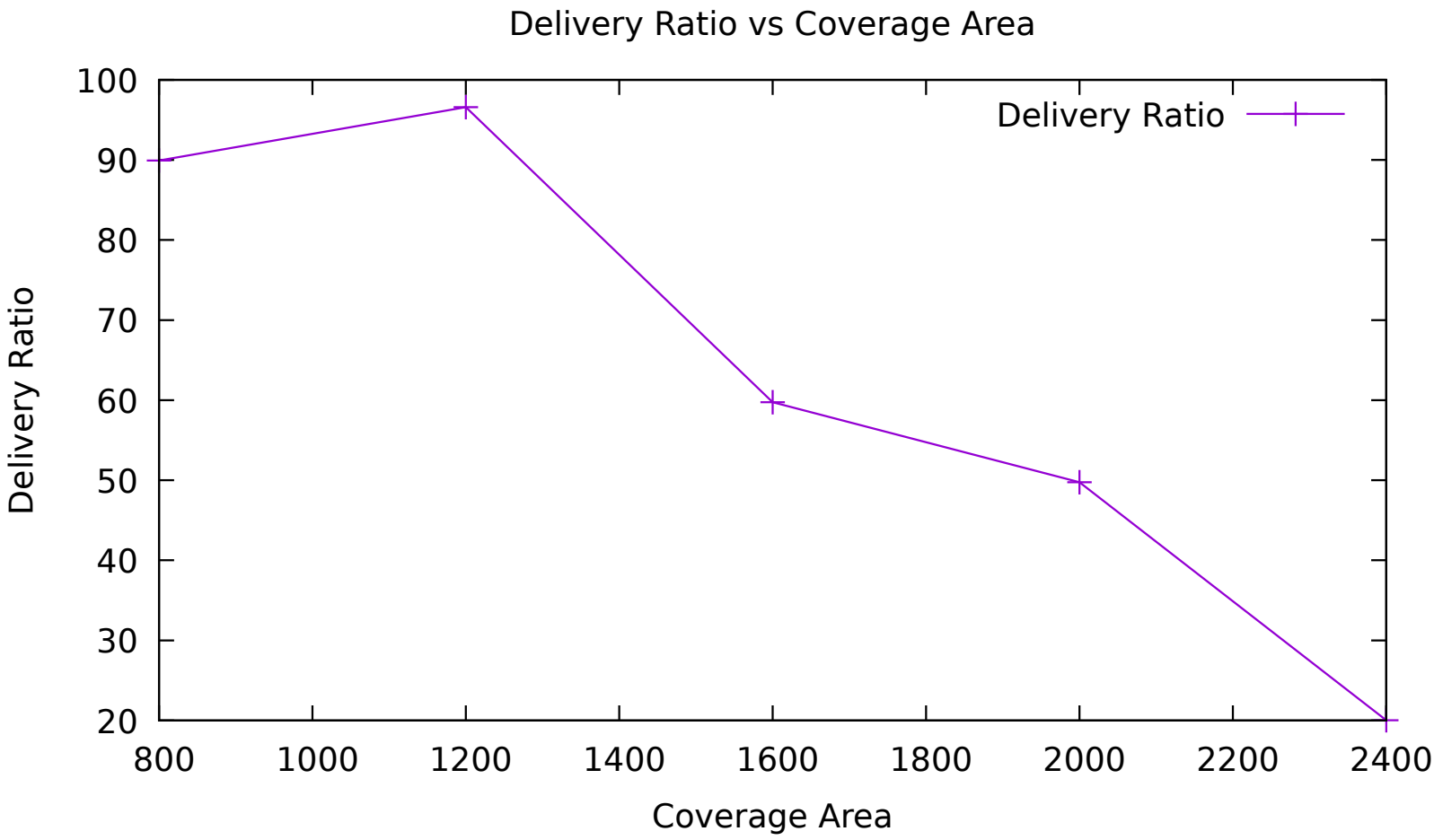

DeliveryRatio
DeliveryRatio
1
0.9
0.8
0.7
0.6
0.5
0.4
20
30
40
50
60
70
80
90
100
Number of Nodes
DeliveryRatio

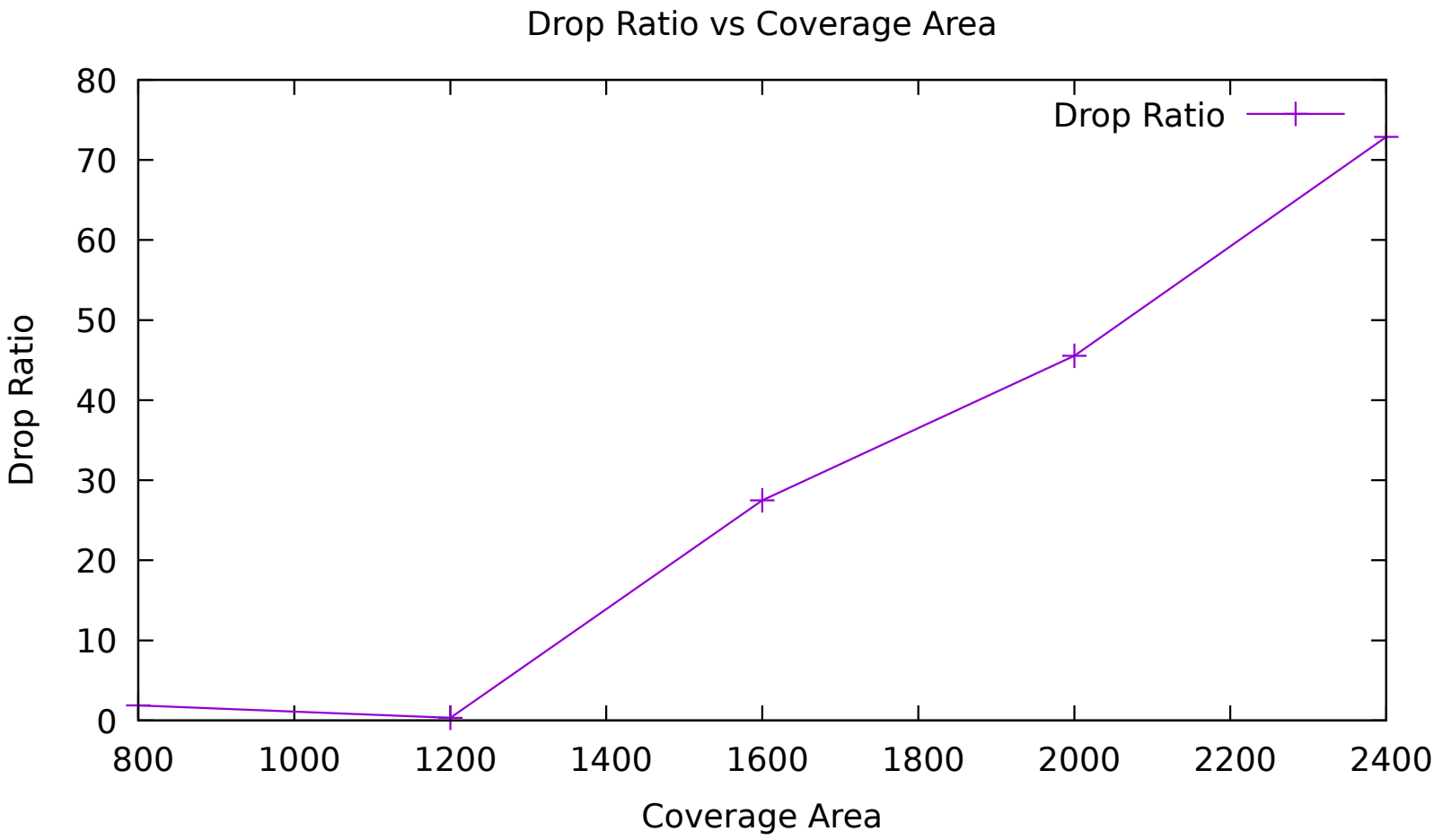

DropRatio
DropRatio
0.6
0.5
0.4
0.3
0.2
0.1
0
20
30
40
50
60
70
80
90
100
Number of Nodes
DropRatio

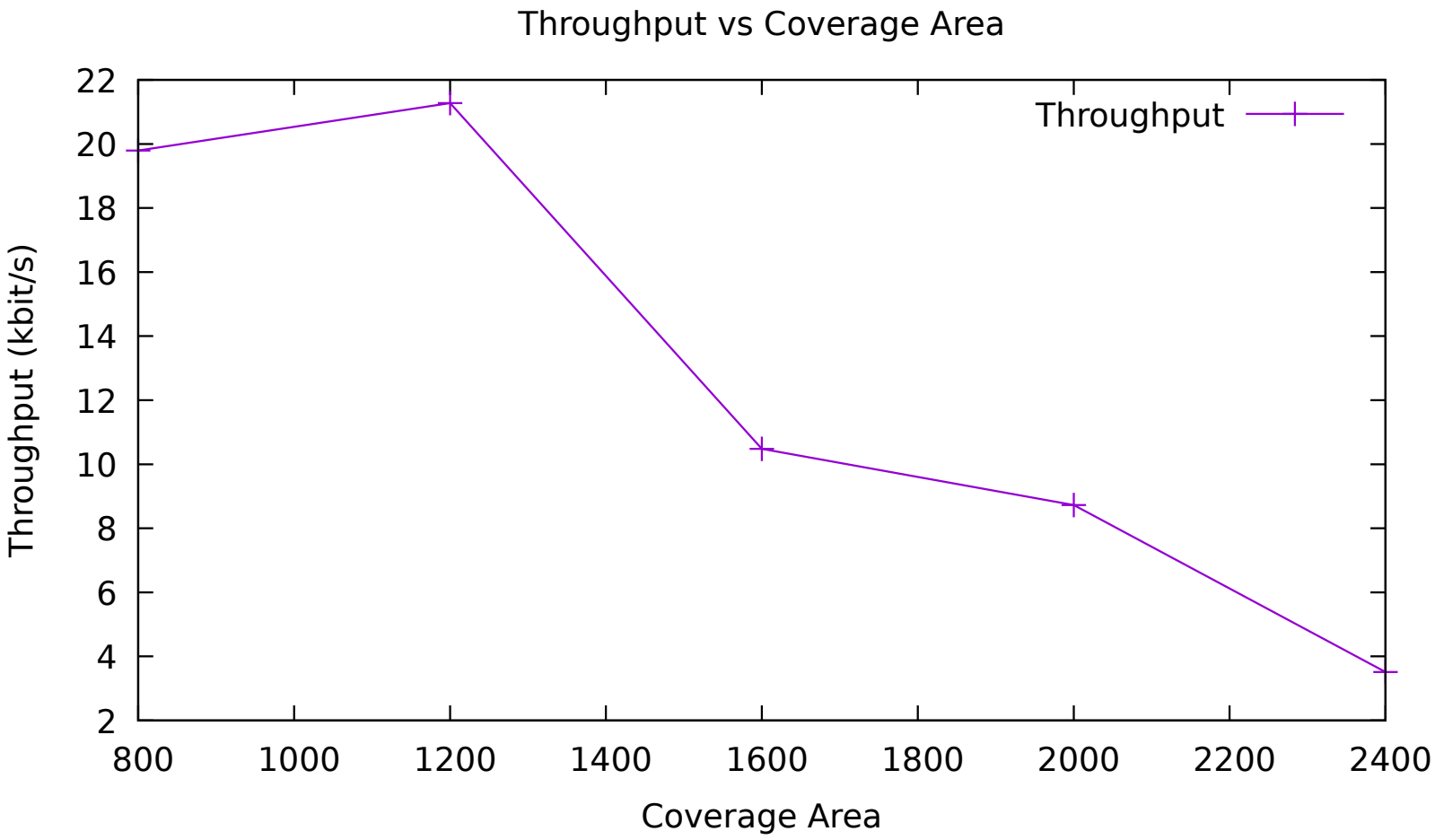

Throughput
throughput
Throughput
Number of Nodes
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
20
30
40
50
60
70
80
90
100

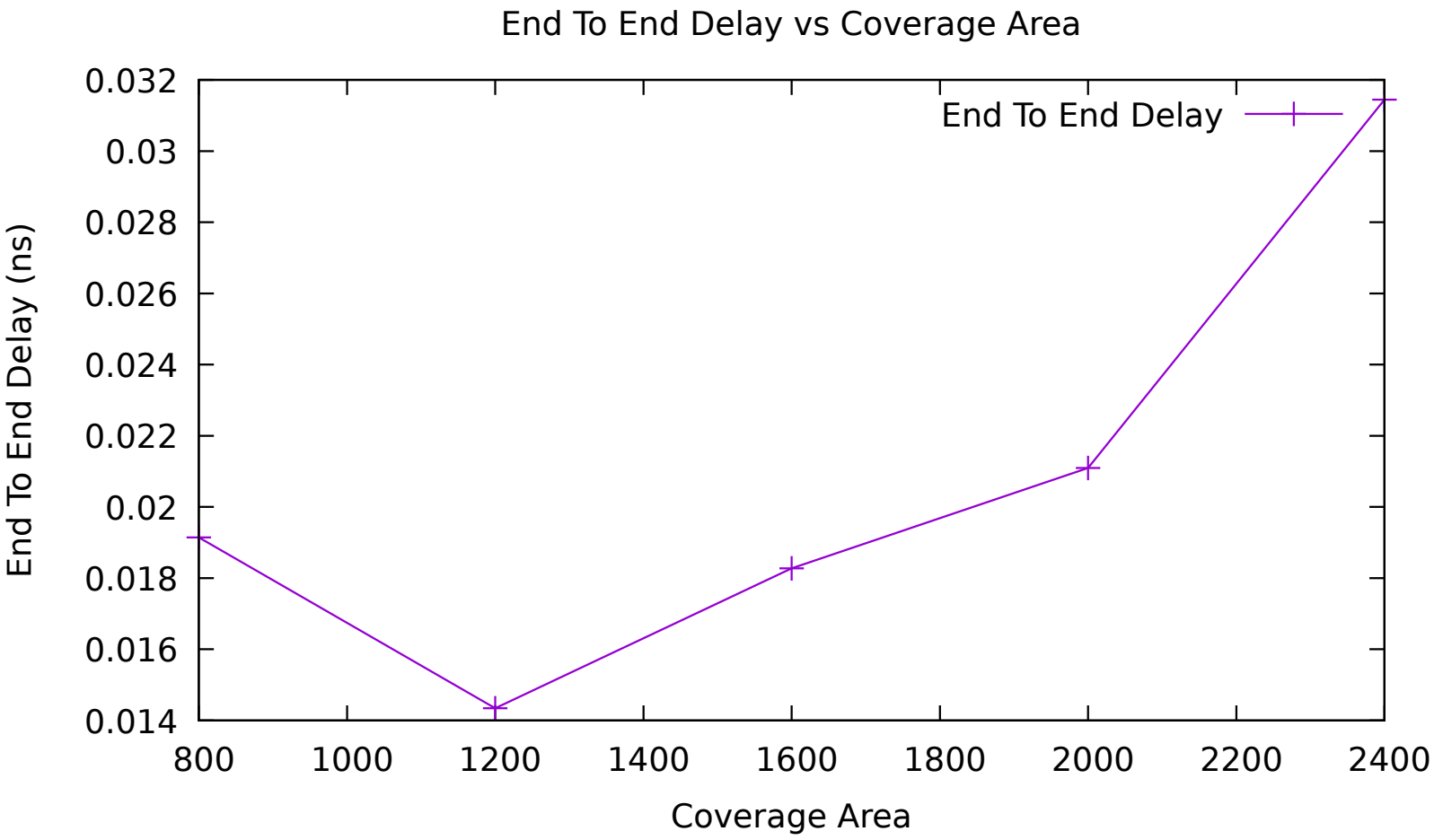

EndToEndDelay
EndToEndDelay
0.024
0.022
0.02
0.018
0.016
0.014
0.012
0.01
0.008
20
30
40
50
60
70
80
90
100
Number of Nodes

Varying Coverage Area

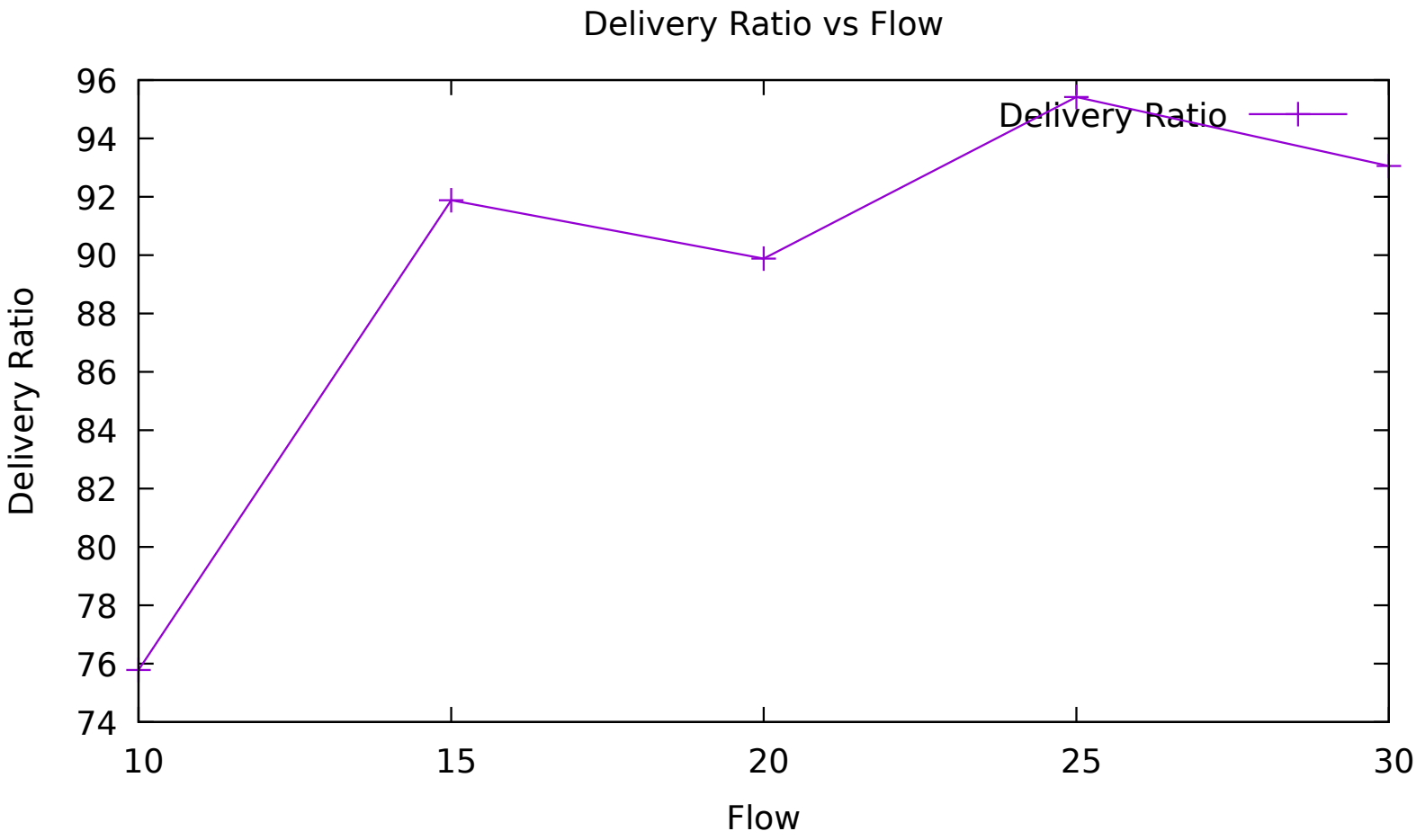

Delivery Ratio vs Coverage Area
Delivery Ratio
Coverage Area
100
90
80
70
60
50
40
30
20
800
1000
1200
1400
1600
1800
2000
2200
2400
Delivery Ratio

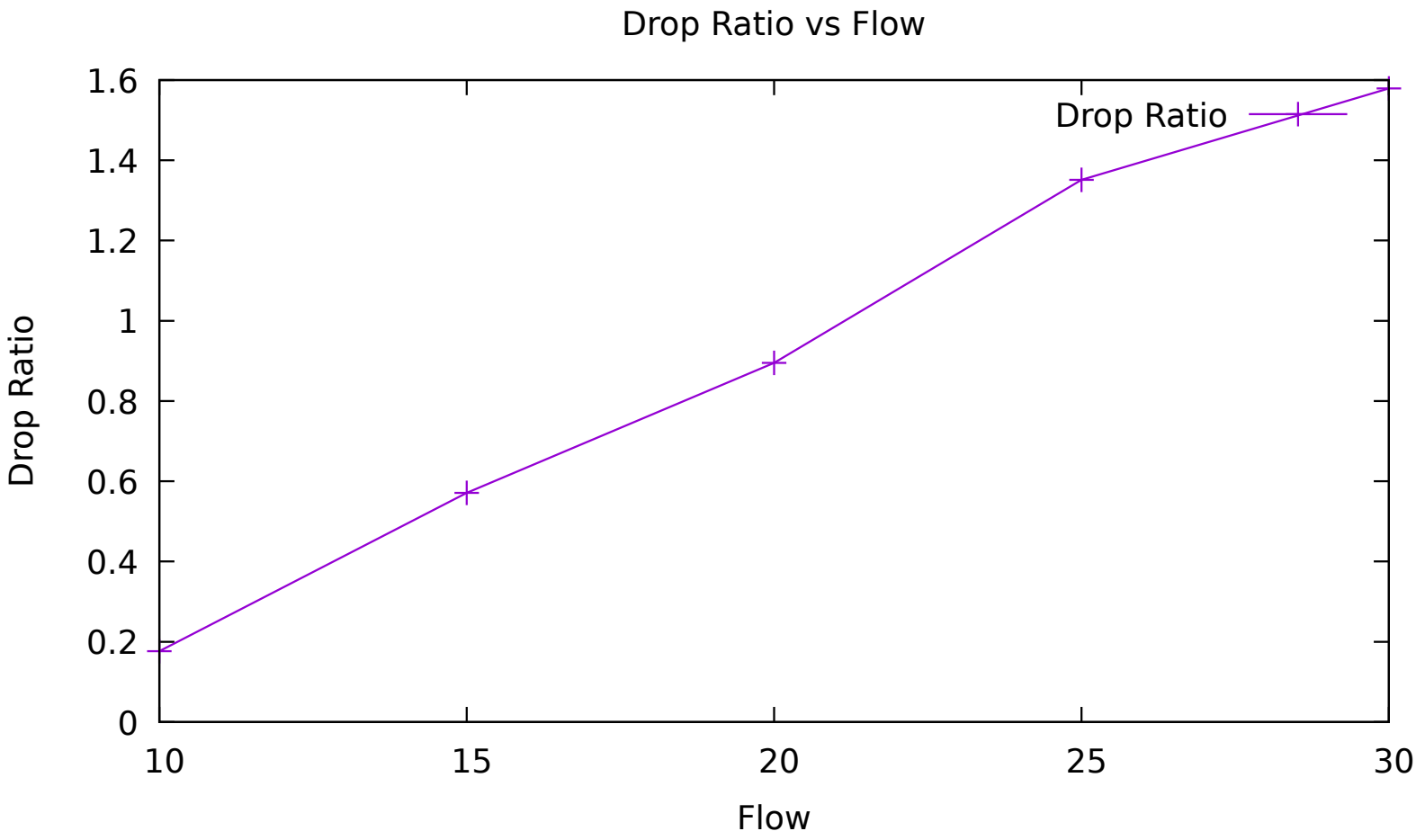

Drop Ratio vs Coverage Area
Drop Ratio
Coverage Area
0
10
20
30
40
50
60
70
80
800
1000
1200
1400
1600
1800
2000
2200
2400

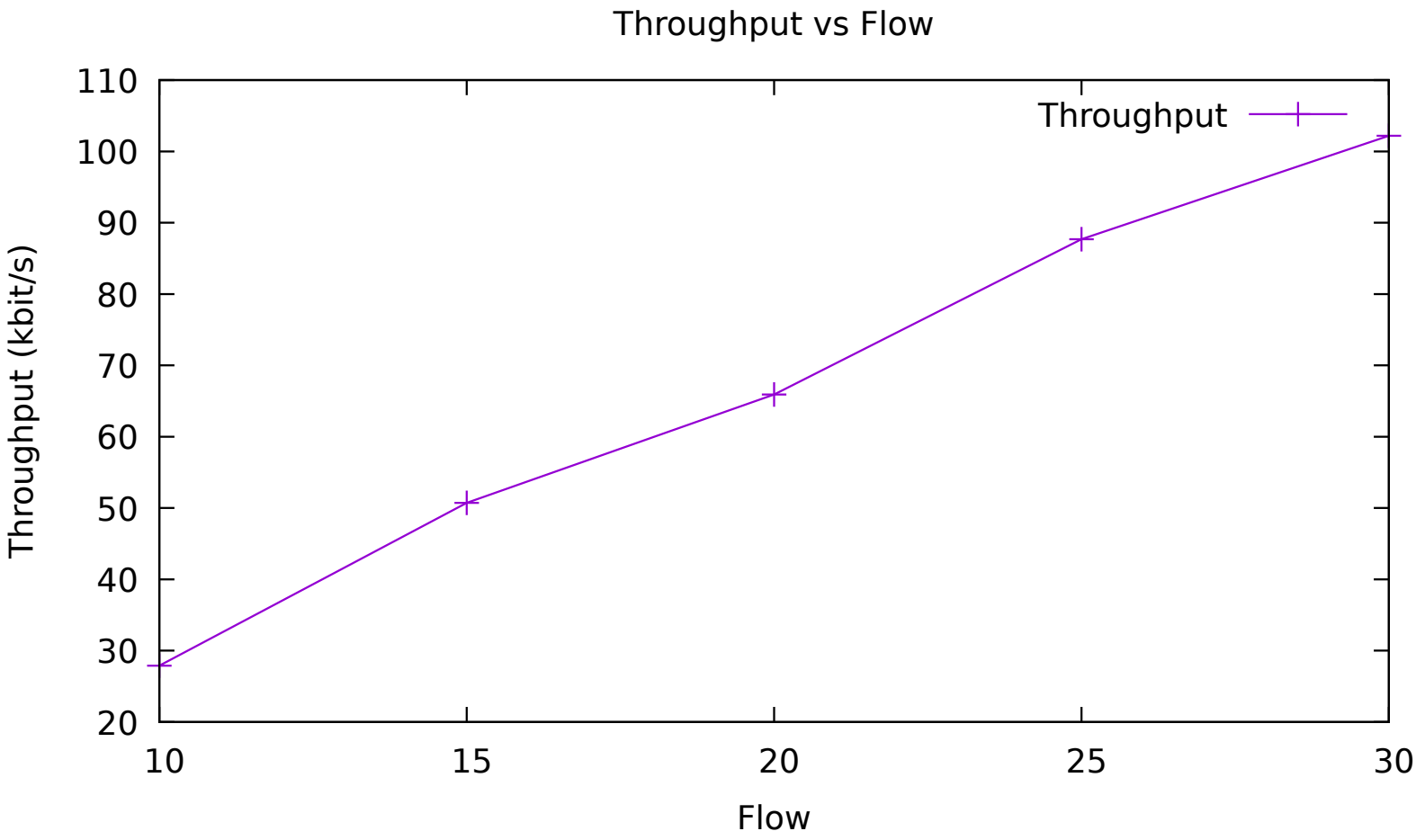

Throughput vs Coverage Area
Throughput
Throughput (kbit/s)
Coverage Area
22
20
18
16
14
12
10
8
6
4
2
800
1000
1200
1400
1600
1800
2000
2200
2400

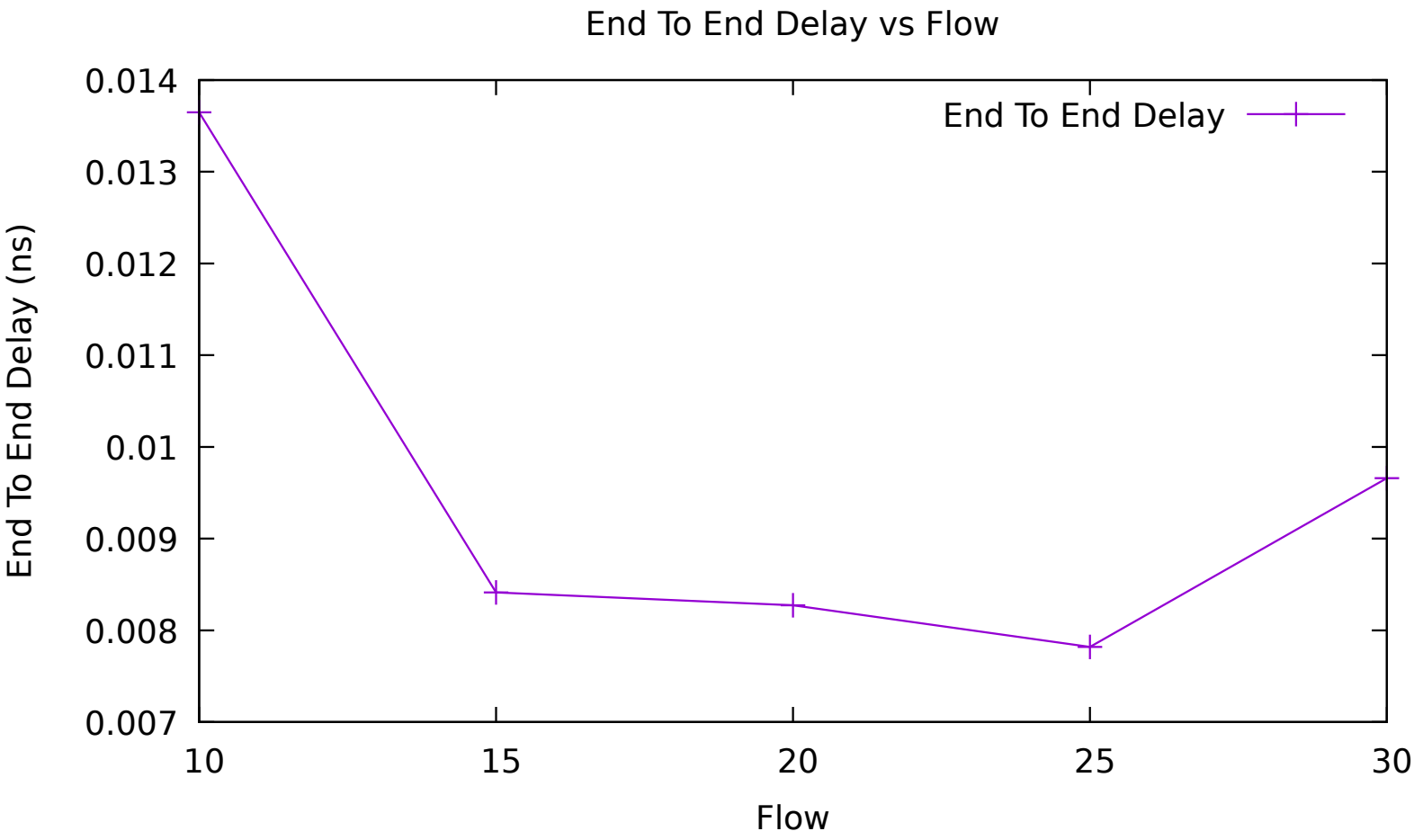

End To End Delay vs Coverage Area
End To End Delay
End To End Delay (ns)
0.032
0.03
0.028
0.026
0.024
0.022
0.02
0.018
0.016
0.014
800
1000
1200
1400
1600
1800
2000
2200
2400
Coverage Area

Varying Number of Flows

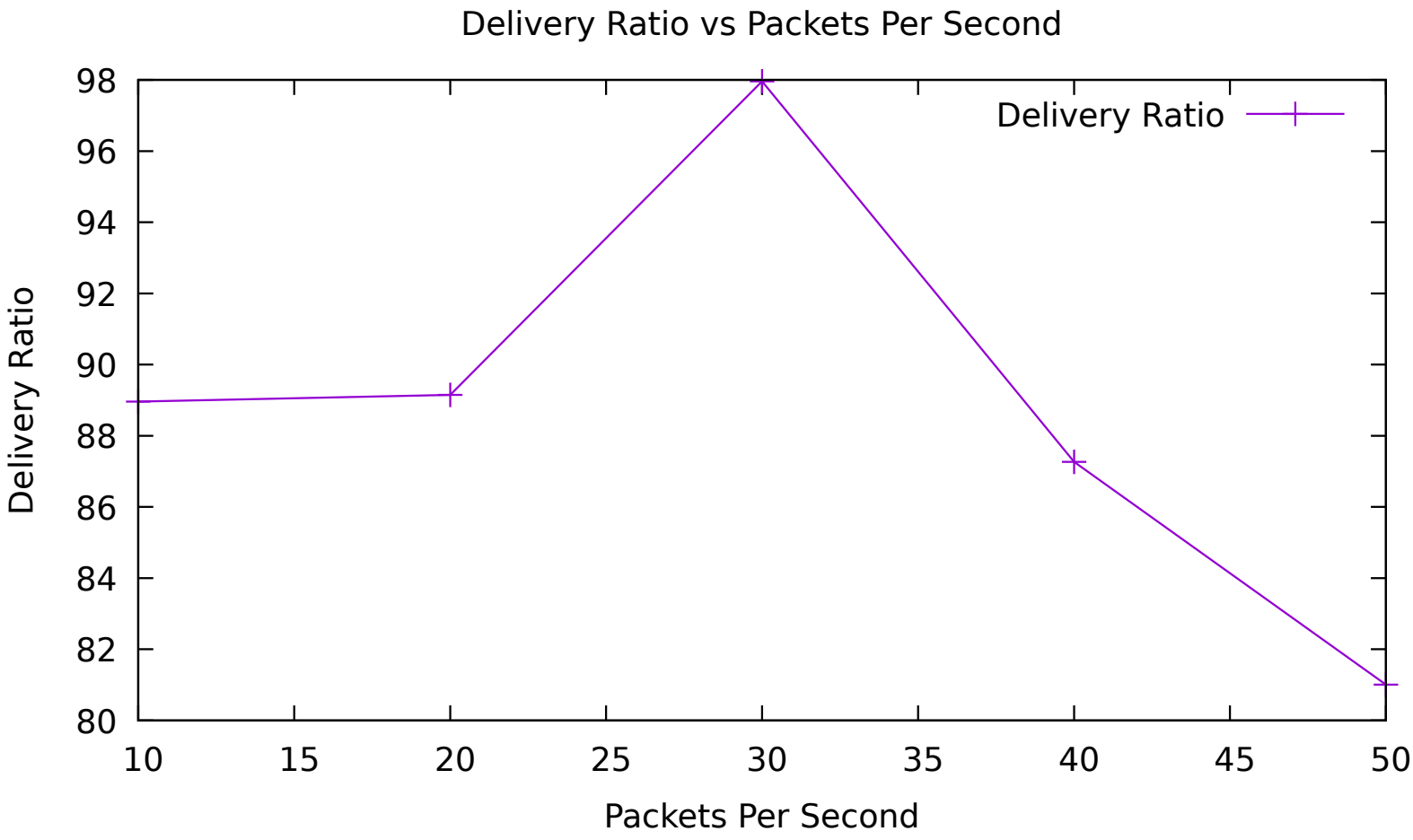

Delivery Ratio vs Flow
Delivery Ratio
Delivery Ratio
Flow
96
94
92
90
88
86
84
82
80
78
76
74
10
15
20
25
30

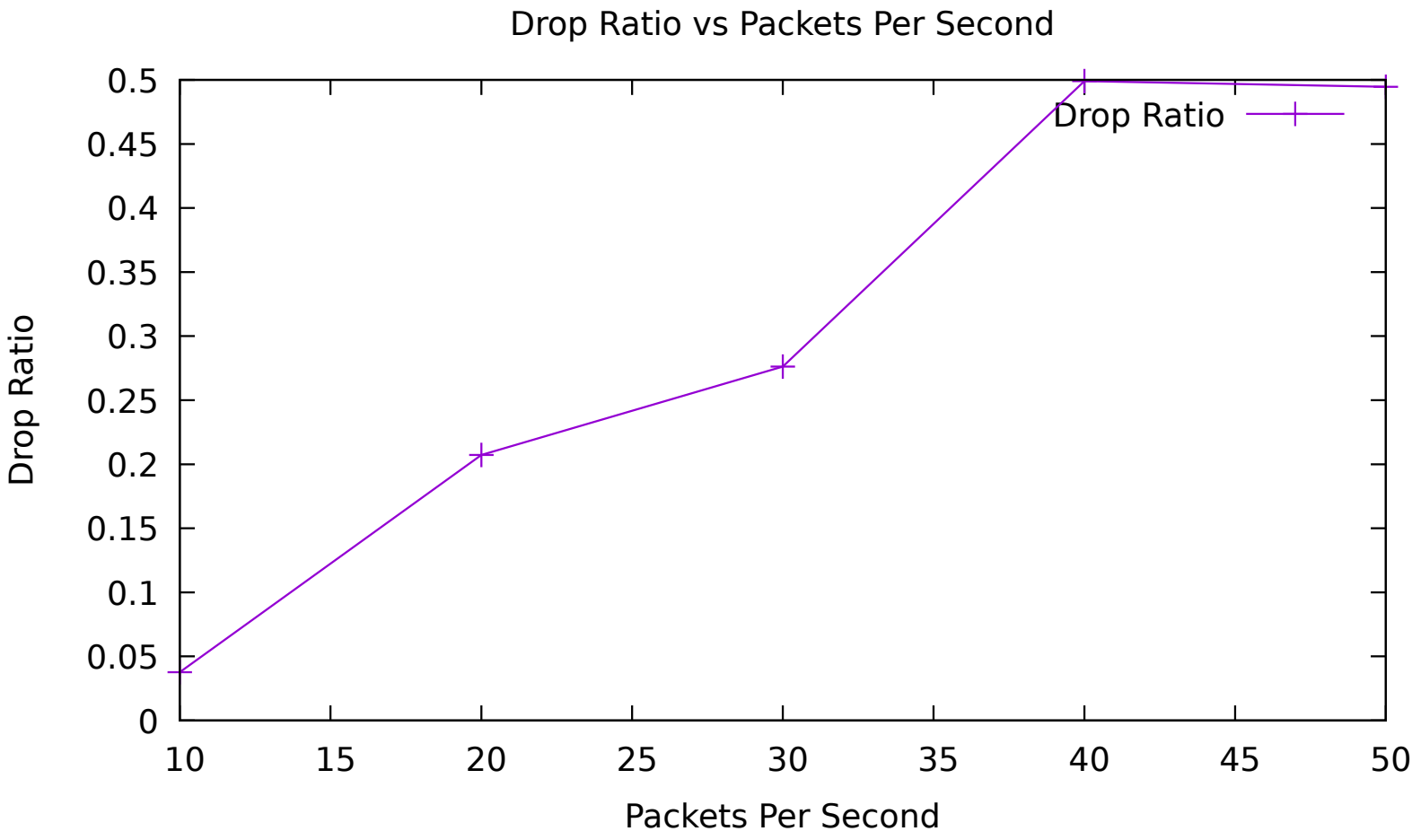

Drop Ratio vs Flow
Drop Ratio
Flow
Drop Ratio
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
10
15
20
25
30

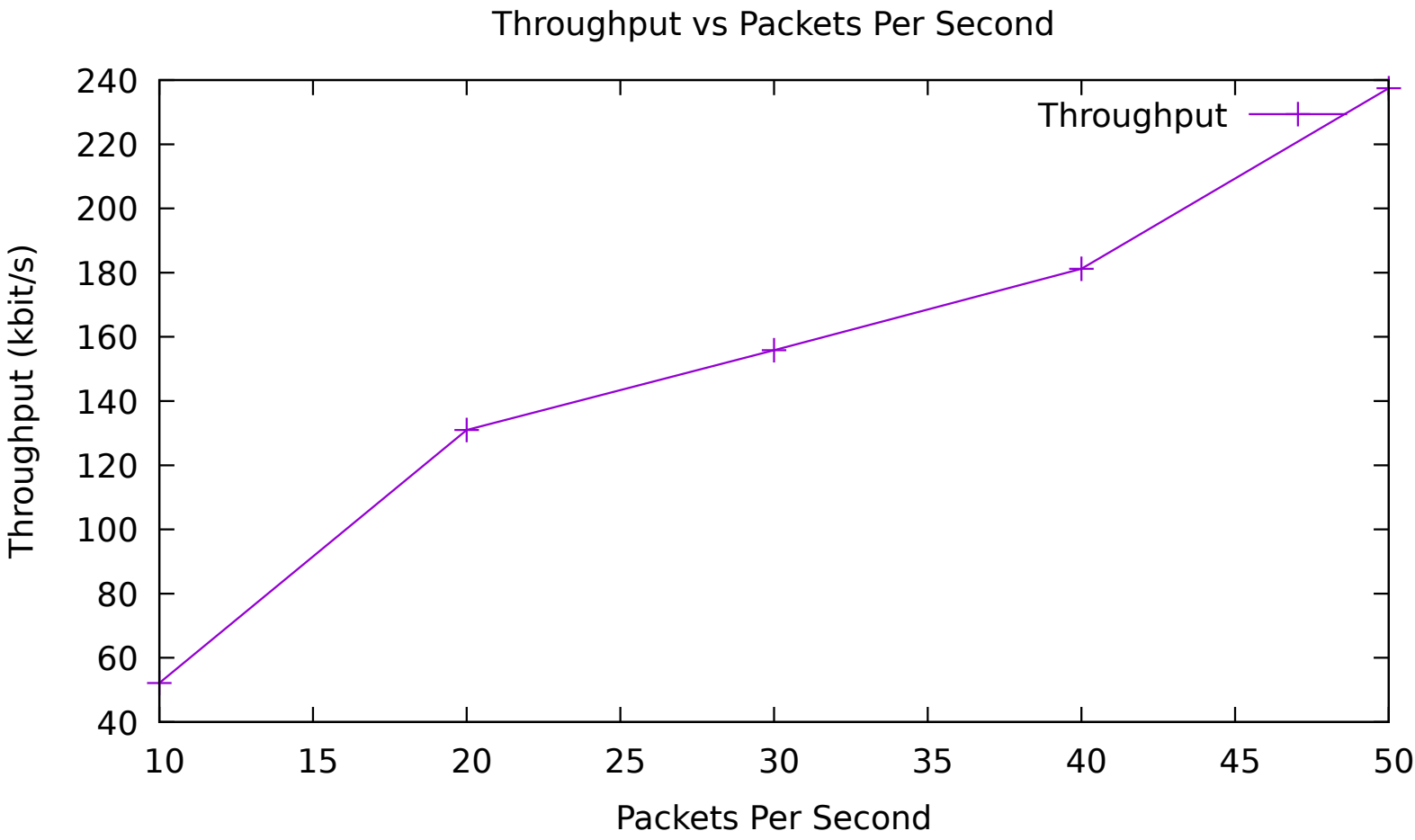

Throughput vs Flow
Throughput
Throughput (kbit/s)
Flow
110
100
90
80
70
60
50
40
30
20
10
15
20
25
30

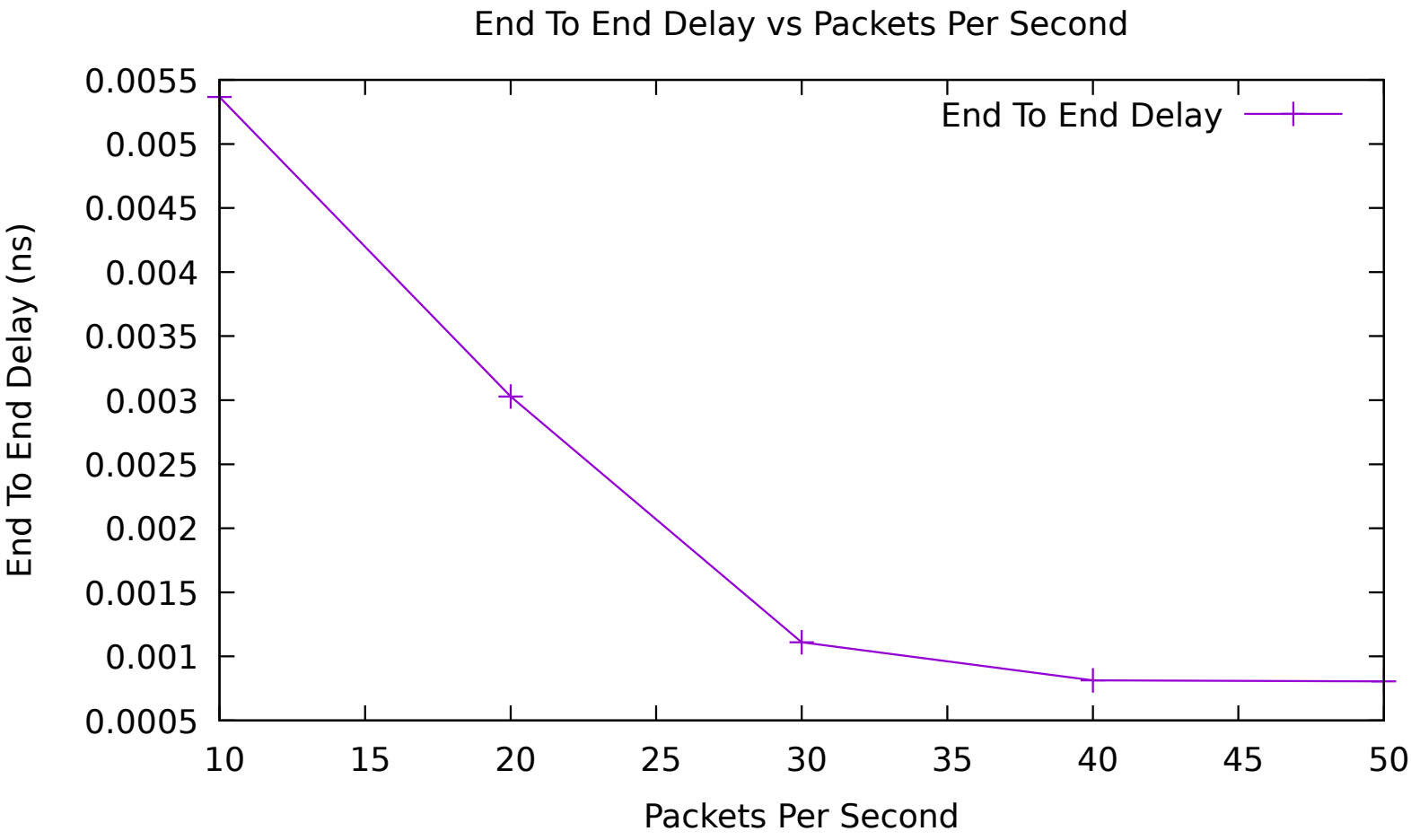

End To End Delay vs Flow
End To End Delay
End To End Delay (ns)
0.014
0.013
0.012
0.011
0.01
0.009
0.008
0.007
10
15
20
25
30
Flow

Varying Number of Packets/Second


Delivery Ratio vs Packets Per Second
Delivery Ratio
Delivery Ratio
98
96
94
92
90
88
86
84
82
80
10
15
20
25
30
35
40
45
50
Packets Per Second


Drop Ratio vs Packets Per Second
Drop Ratio
Packets Per Second
Drop Ratio
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
10
15
20
25
30
35
40
45
50


Throughput vs Packets Per Second
Throughput
Throughput (kbit/s)
Packets Per Second
240
220
200
180
160
140
120
100
80
60
40
10
15
20
25
30
35
40
45
50

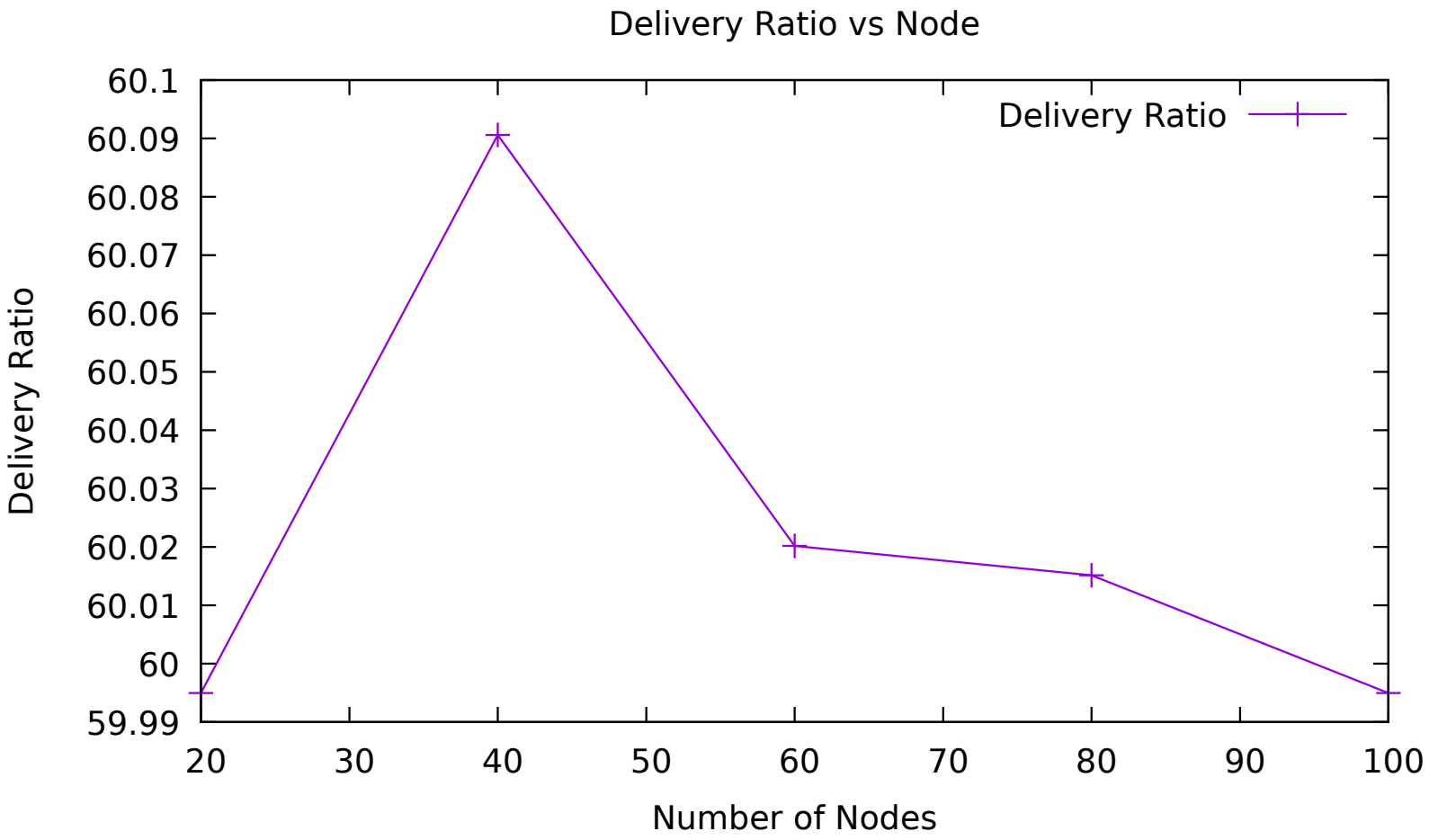

End To End Delay vs Packets Per Second
End To End Delay
End To End Delay (ns)
0.0055
0.005
0.0045
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
10
15
20
25
30
35
40
45
50
Packets Per Second

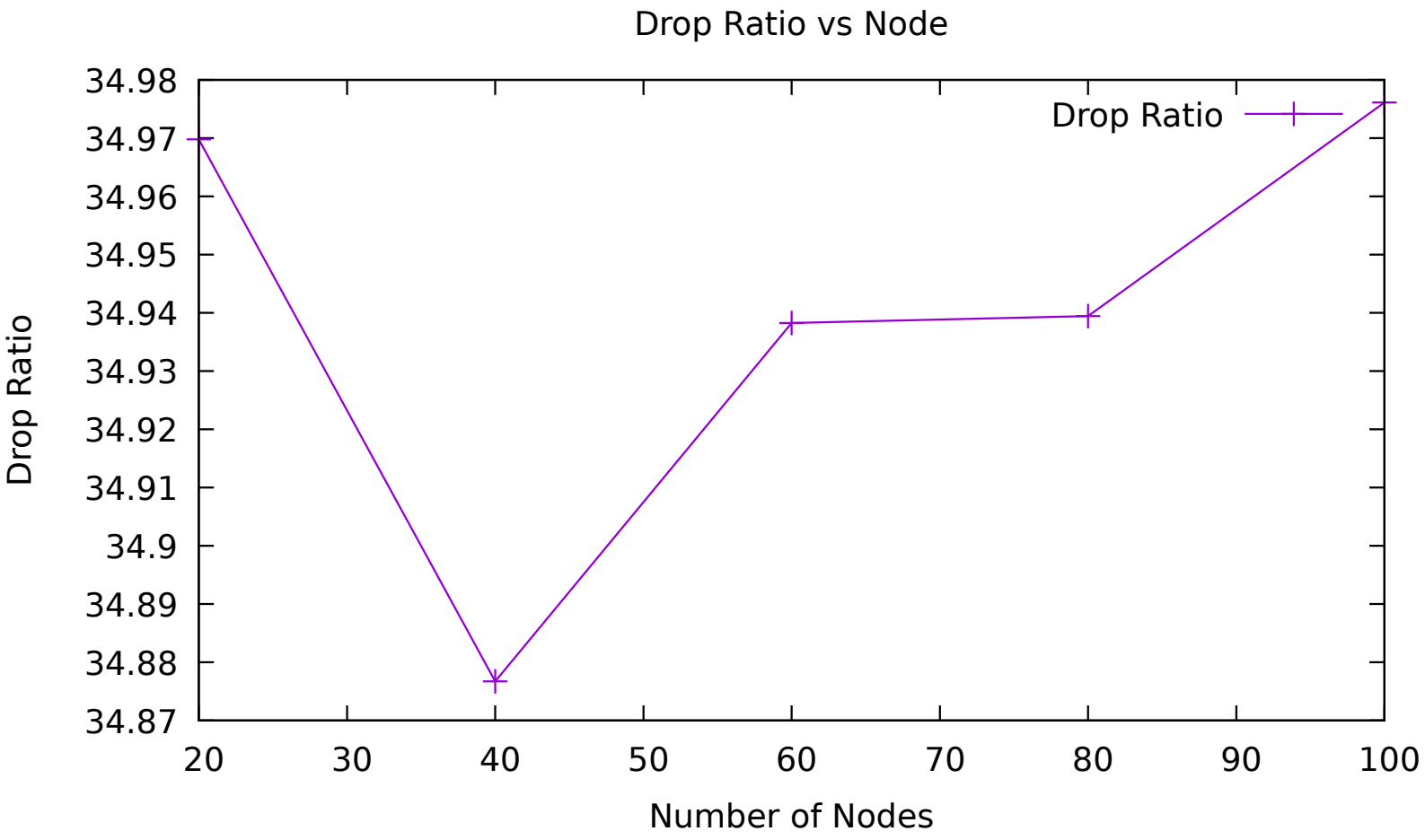
6.2.2 802.15.4 lowrate static network

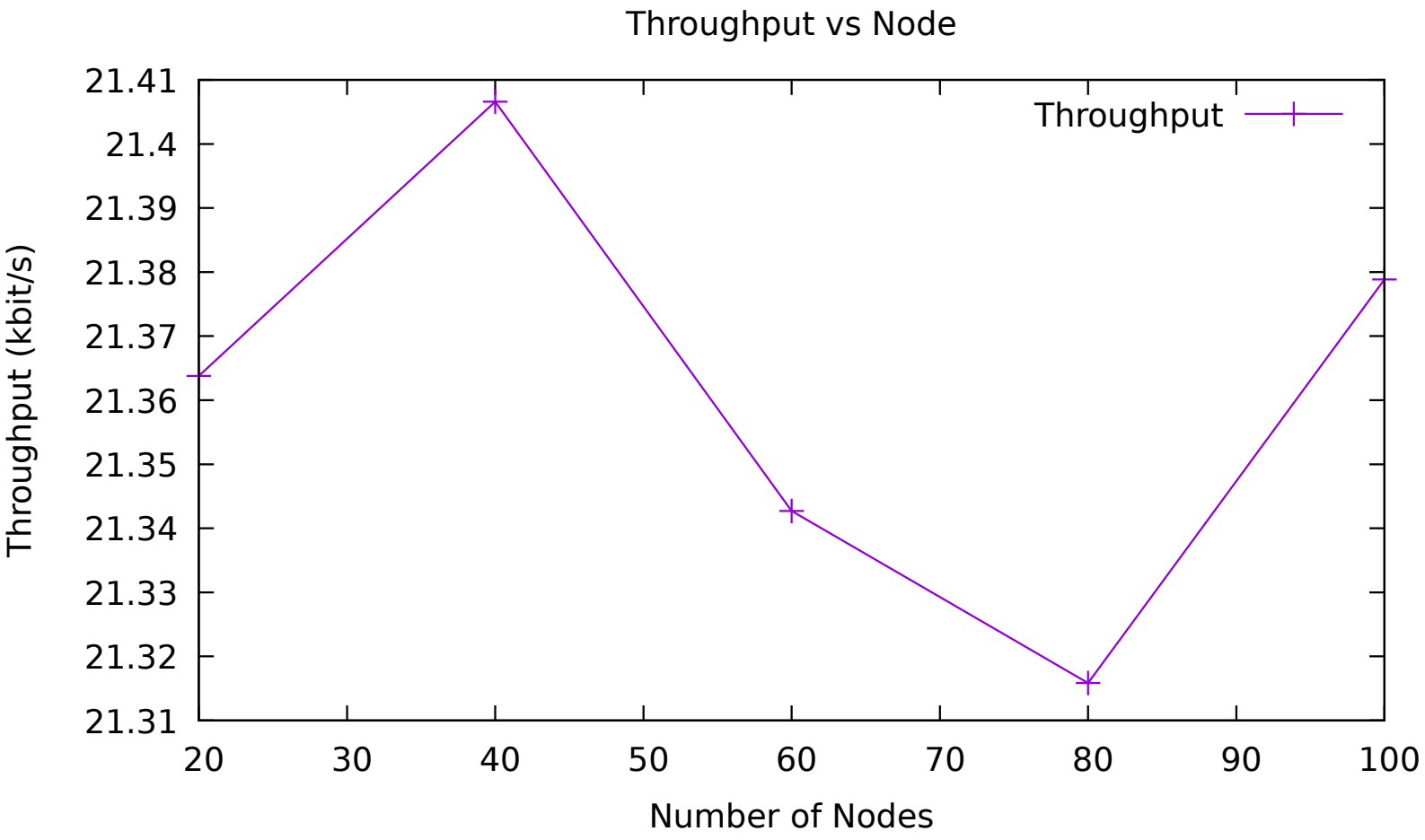
Network Topology Recap

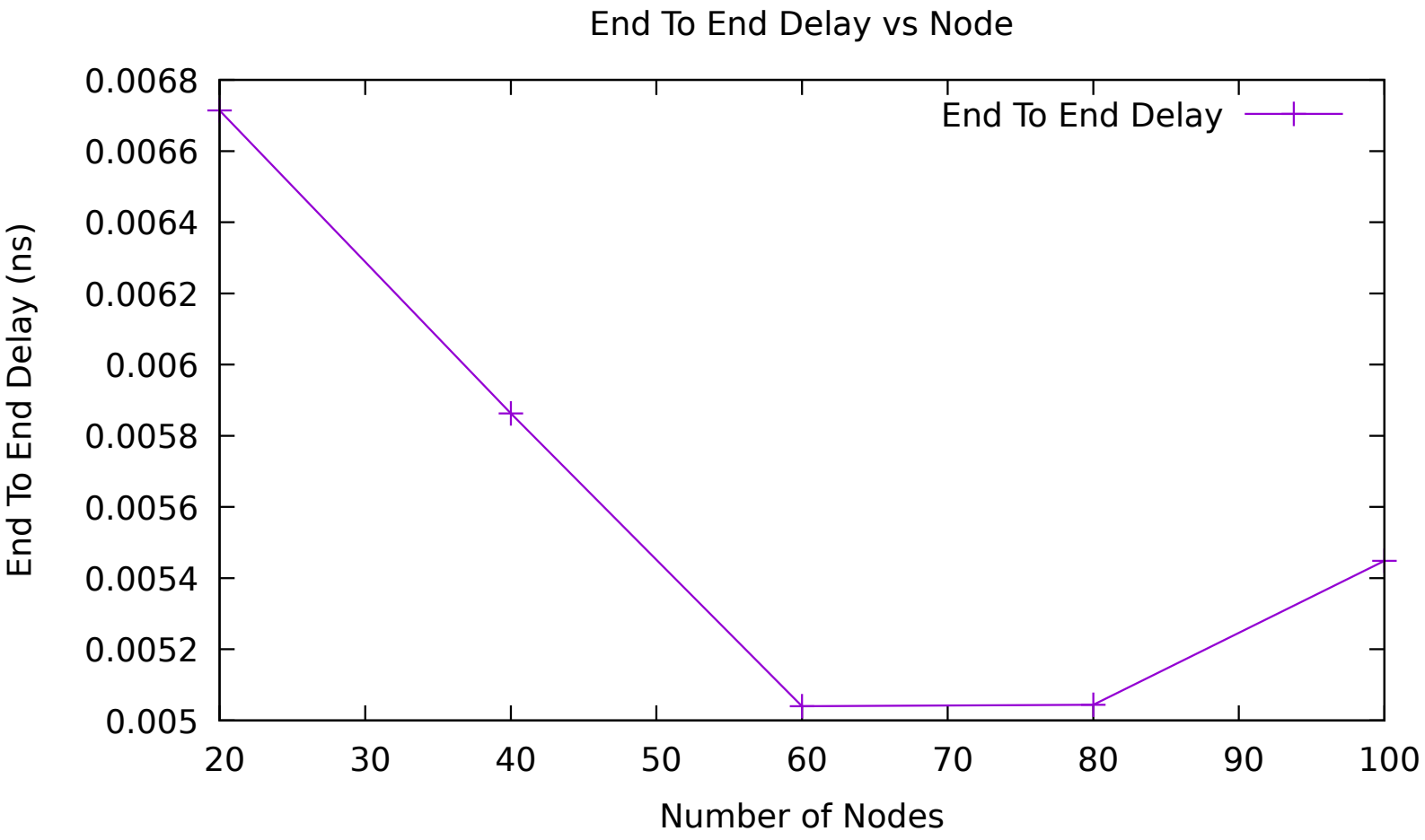
- **Propagation Loss Model** RangePropagationLossModel
- **Channel** LrwpanHelper
- Used **6LoWPAN** group that allows IPv6 packets to be sent and received over IEEE 802.15.4 based networks
- **Routing Protocol** Default Global Routing
- **Application Layer** OnoffHelper/ns3::UdpSocketFactory
- **Mobility Model** ConstantPositionMobilityModel

Varying Number of Nodes

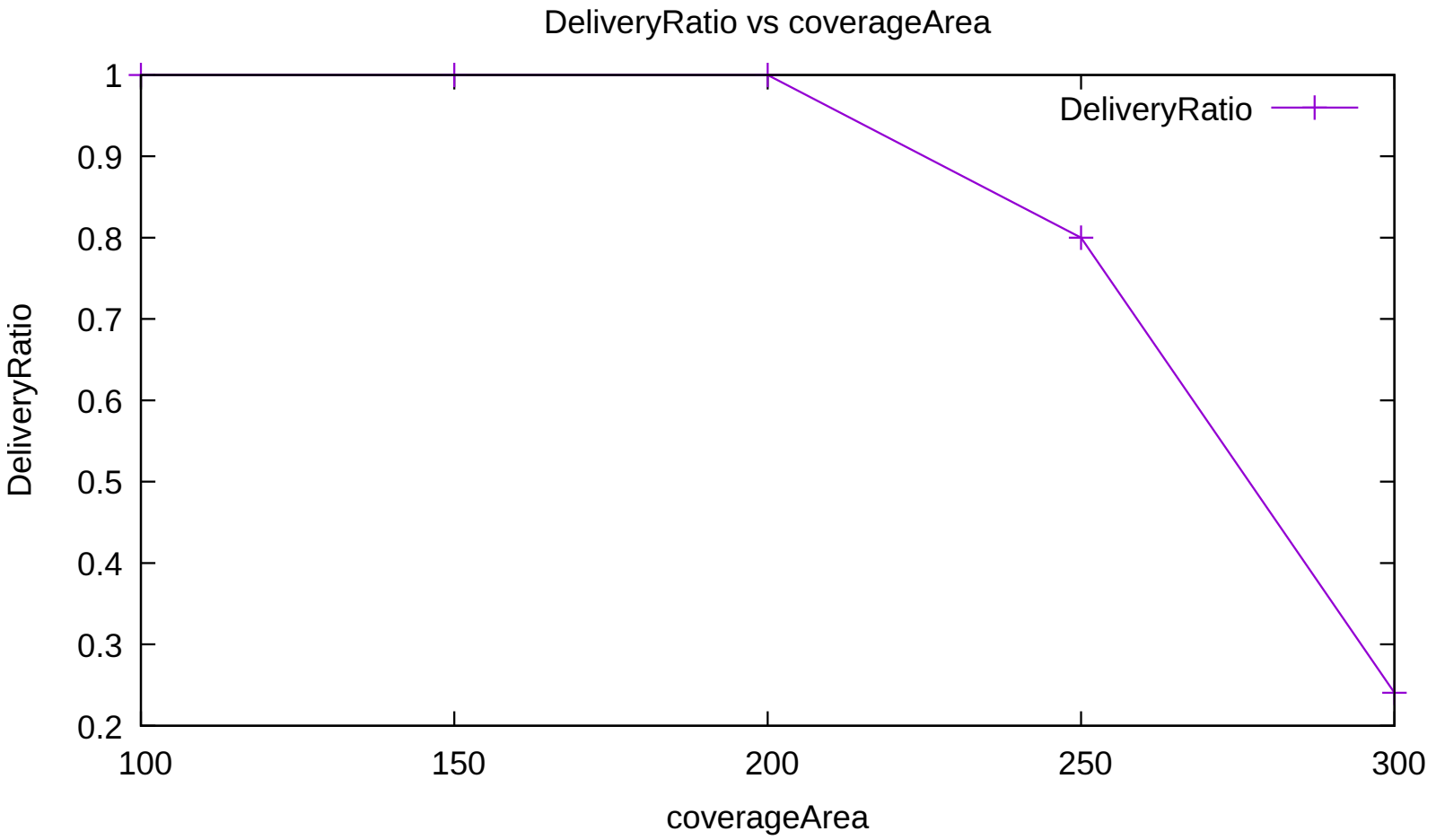

Delivery Ratio vs Node
Delivery Ratio
60.1
60.09
60.08
60.07
60.06
60.05
60.04
60.03
60.02
60.01
60
59.99
20
30
40
50
60
70
80
90
100
Delivery Ratio
Number of Nodes

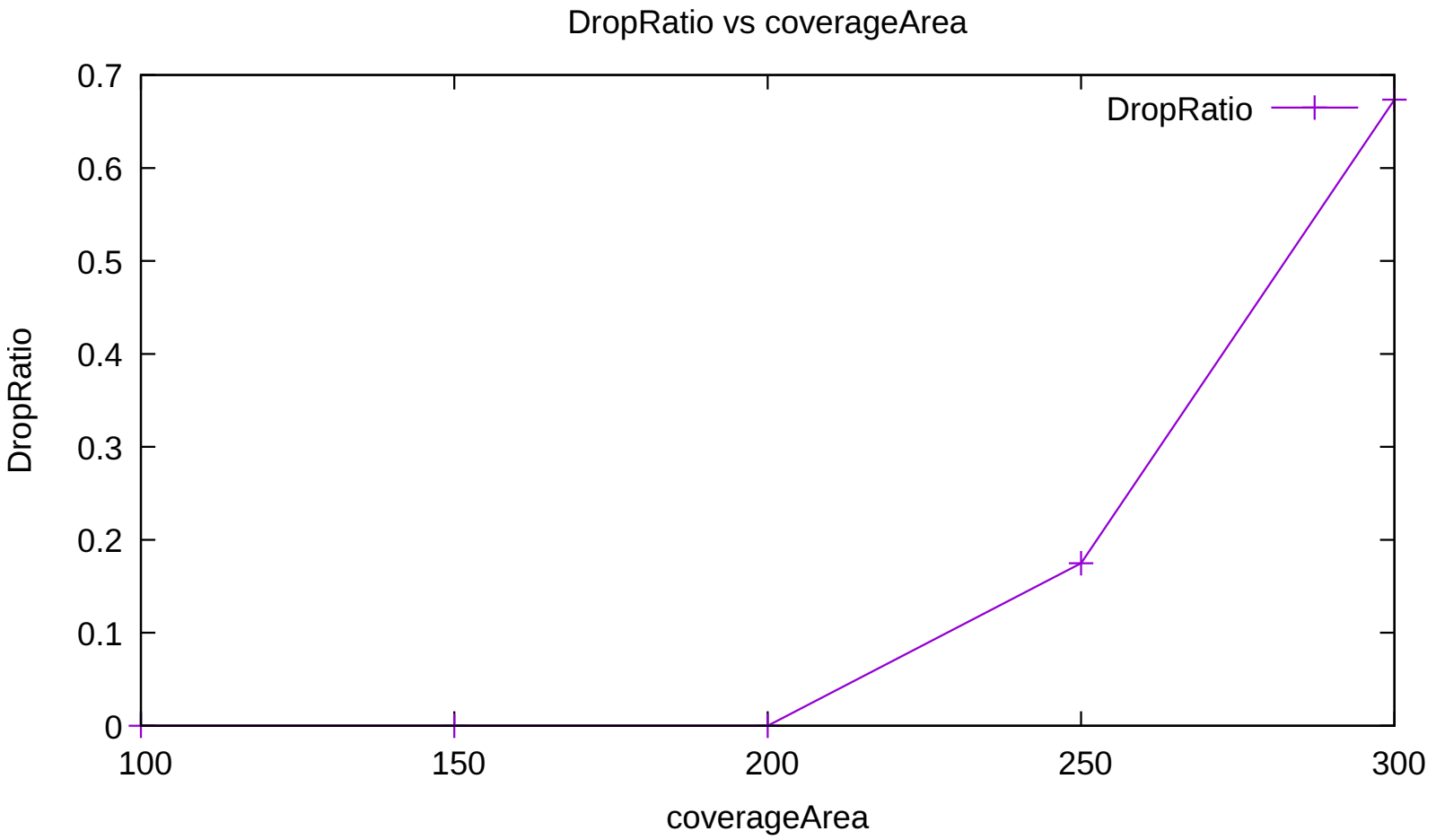

Drop Ratio vs Node
Drop Ratio
34.98
34.97
34.96
34.95
34.94
34.93
34.92
34.91
34.9
34.89
34.88
34.87
20
30
40
50
60
70
80
90
100
Number of Nodes

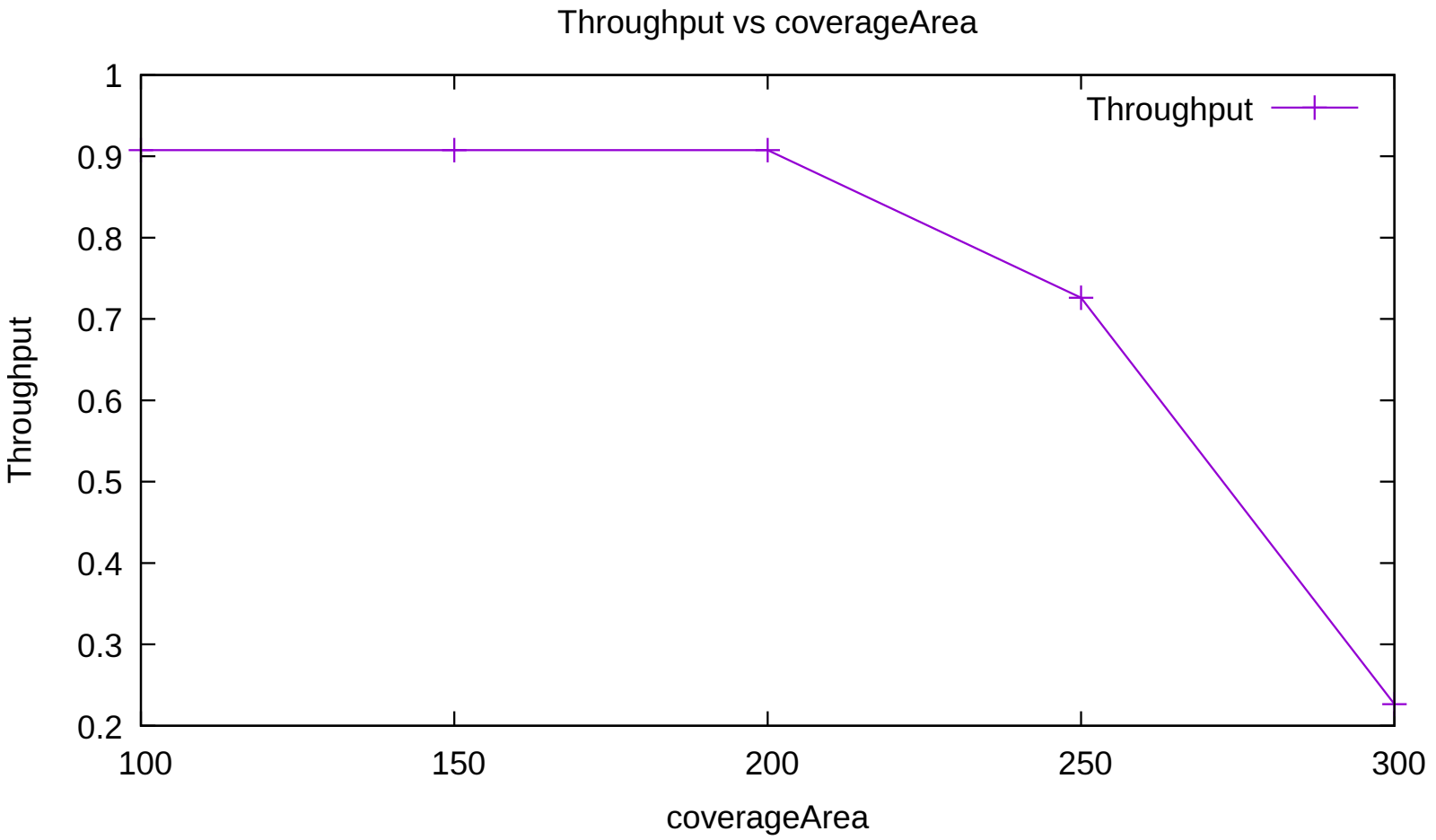

Throughput vs Node
Throughput
Throughput (kbit/s)
21.41
21.4
21.39
21.38
21.37
21.36
21.35
21.34
21.33
21.32
21.31
20
30
40
50
60
70
80
90
100
Number of Nodes

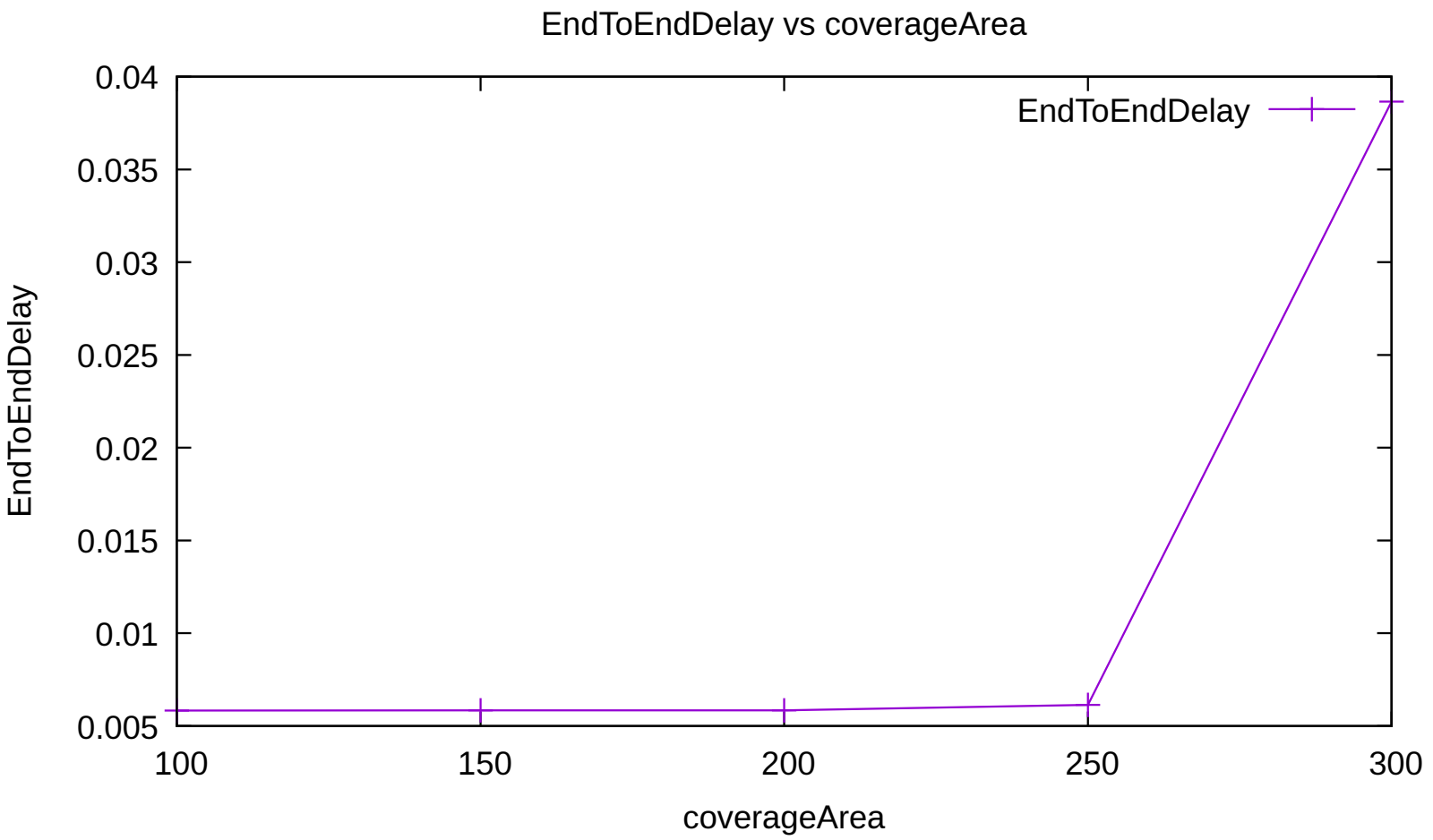

End To End Delay vs Node
End To End Delay
End To End Delay (ns)
0.0068
0.0066
0.0064
0.0062
0.006
0.0058
0.0056
0.0054
0.0052
0.005
20
30
40
50
60
70
80
90
100
Number of Nodes

Varying Coverage Area

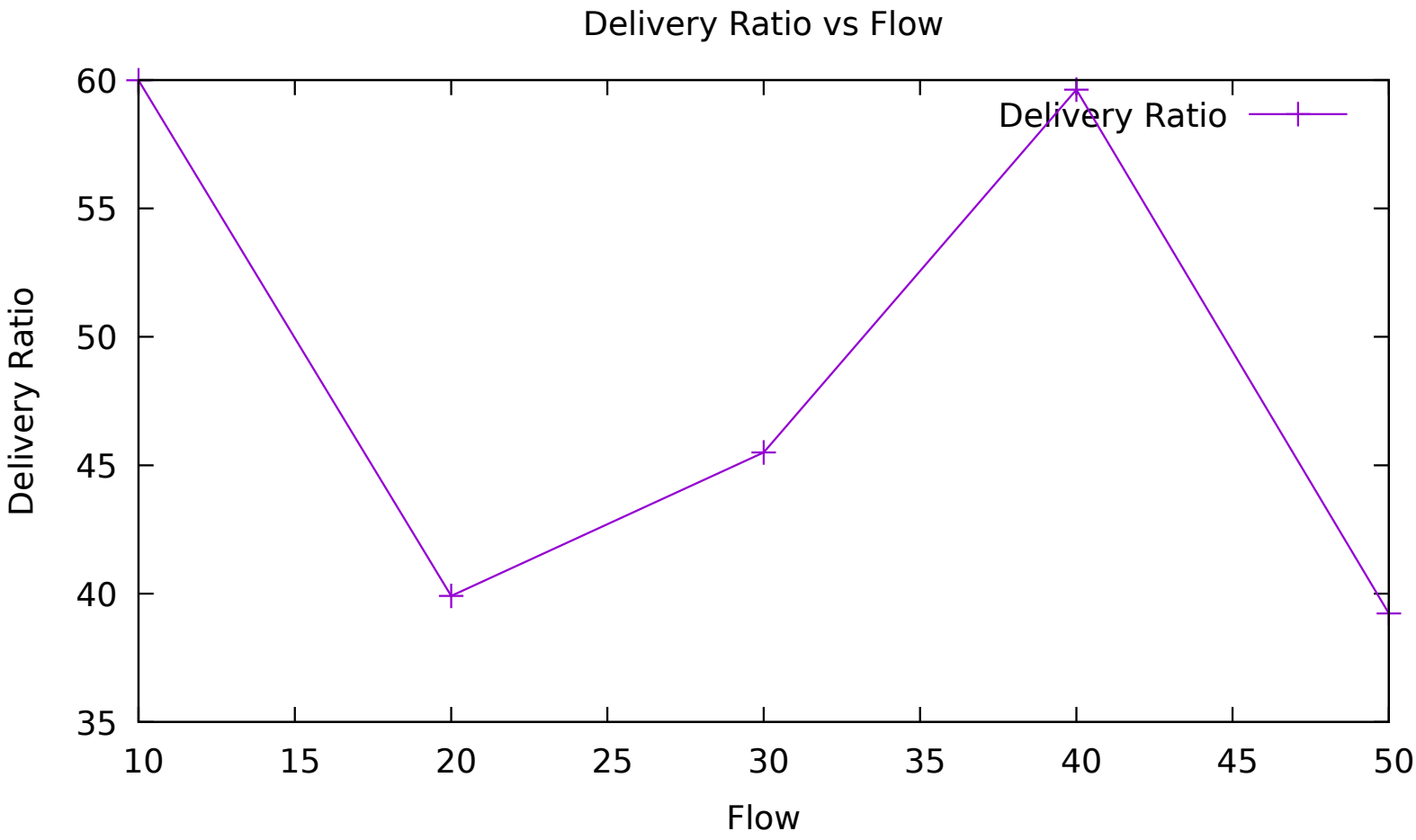

DeliveryRatio vs coverageArea
DeliveryRatio
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
100
150
200
250
300
coverageArea

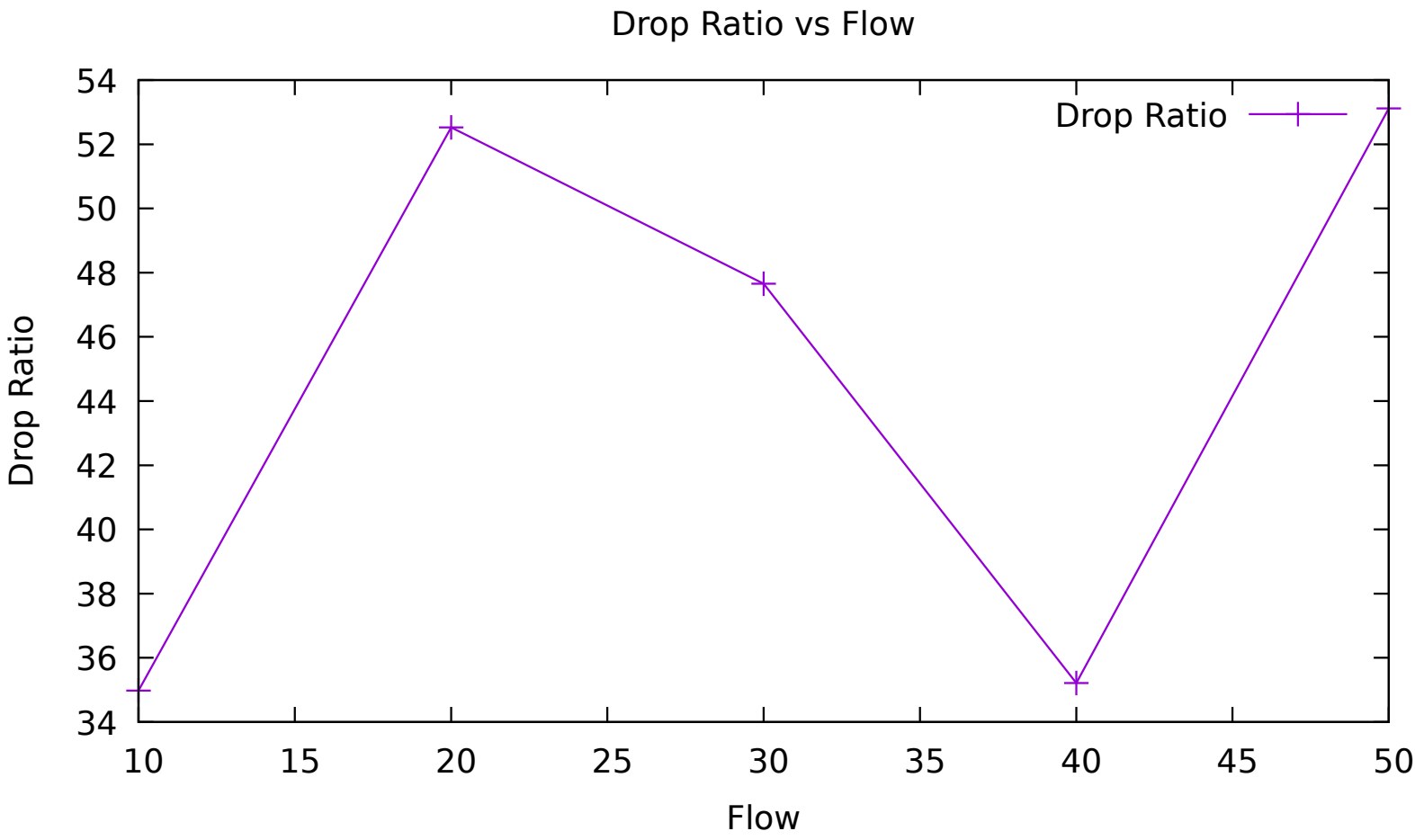

DropRatio vs coverageArea
DropRatio
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100
150
200
250
300
coverageArea
DropRatio

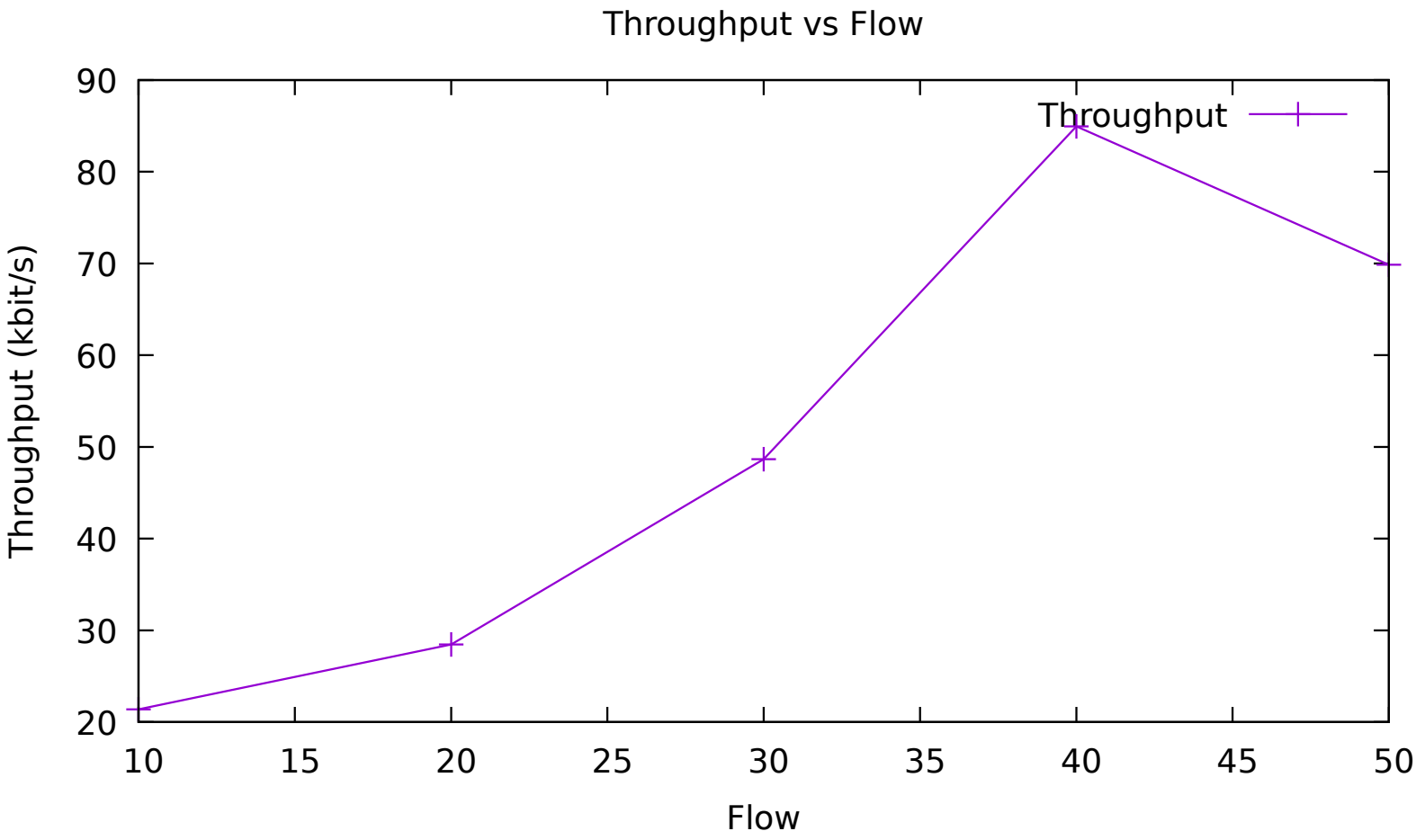

Throughput vs coverageArea
Throughput
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
100
150
200
250
300
coverageArea

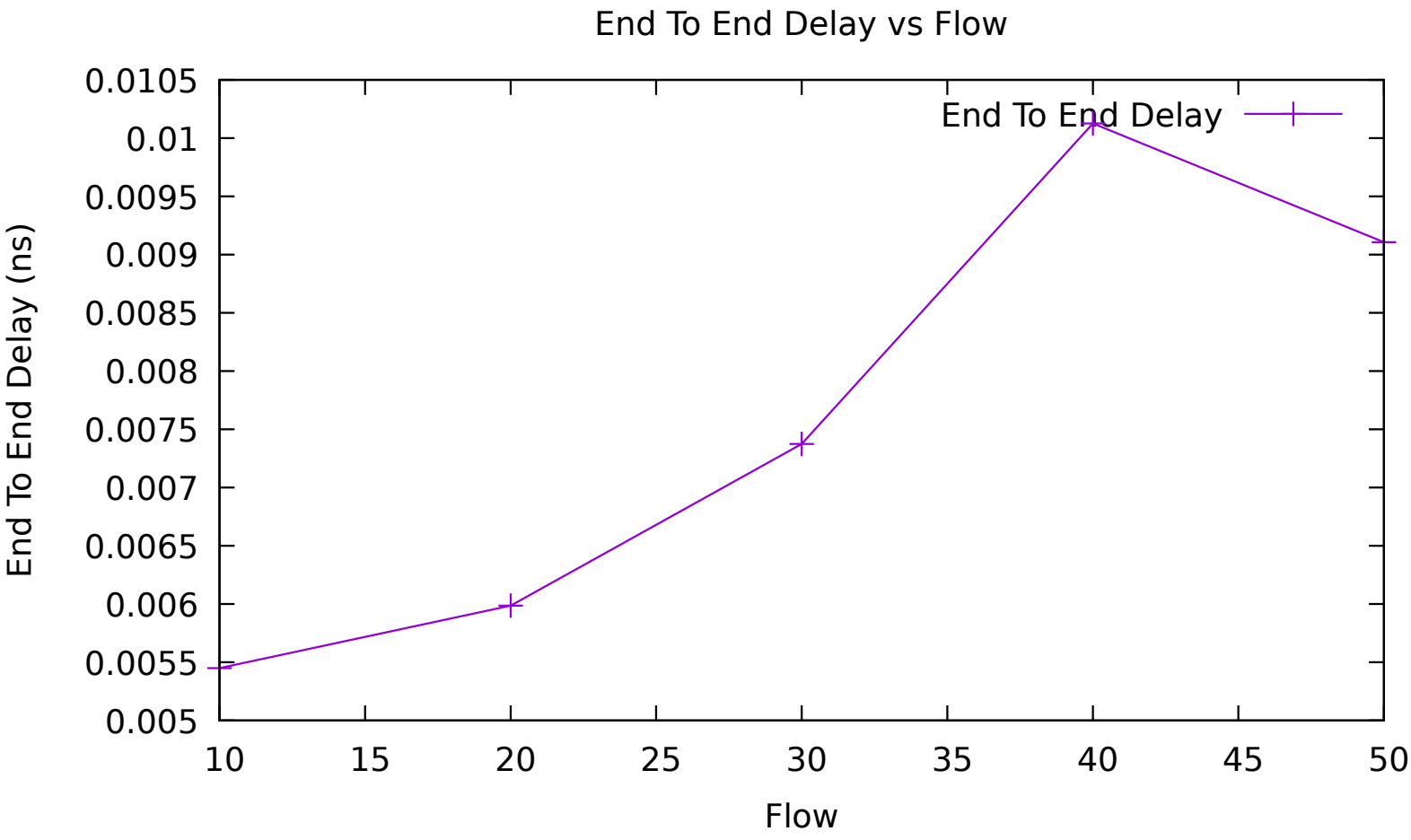

EndToEndDelay vs coverageArea
EndToEndDelay
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
EndToEndDelay
100
150
200
250
300
coverageArea

Varying Number of Flows

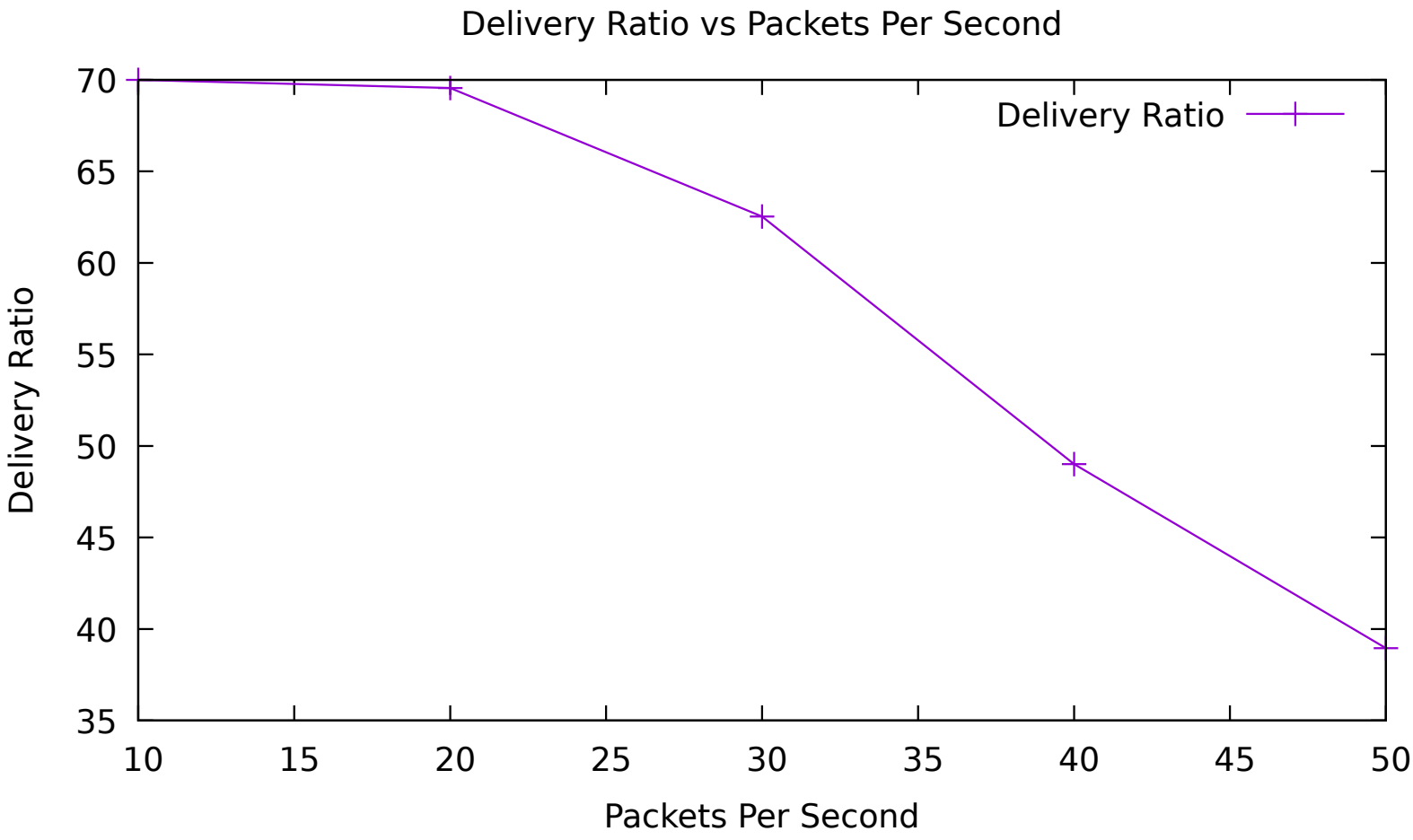

Delivery Ratio vs Flow
Delivery Ratio
Delivery Ratio
Flow
60
55
50
45
40
35
10
15
20
25
30
35
40
45
50

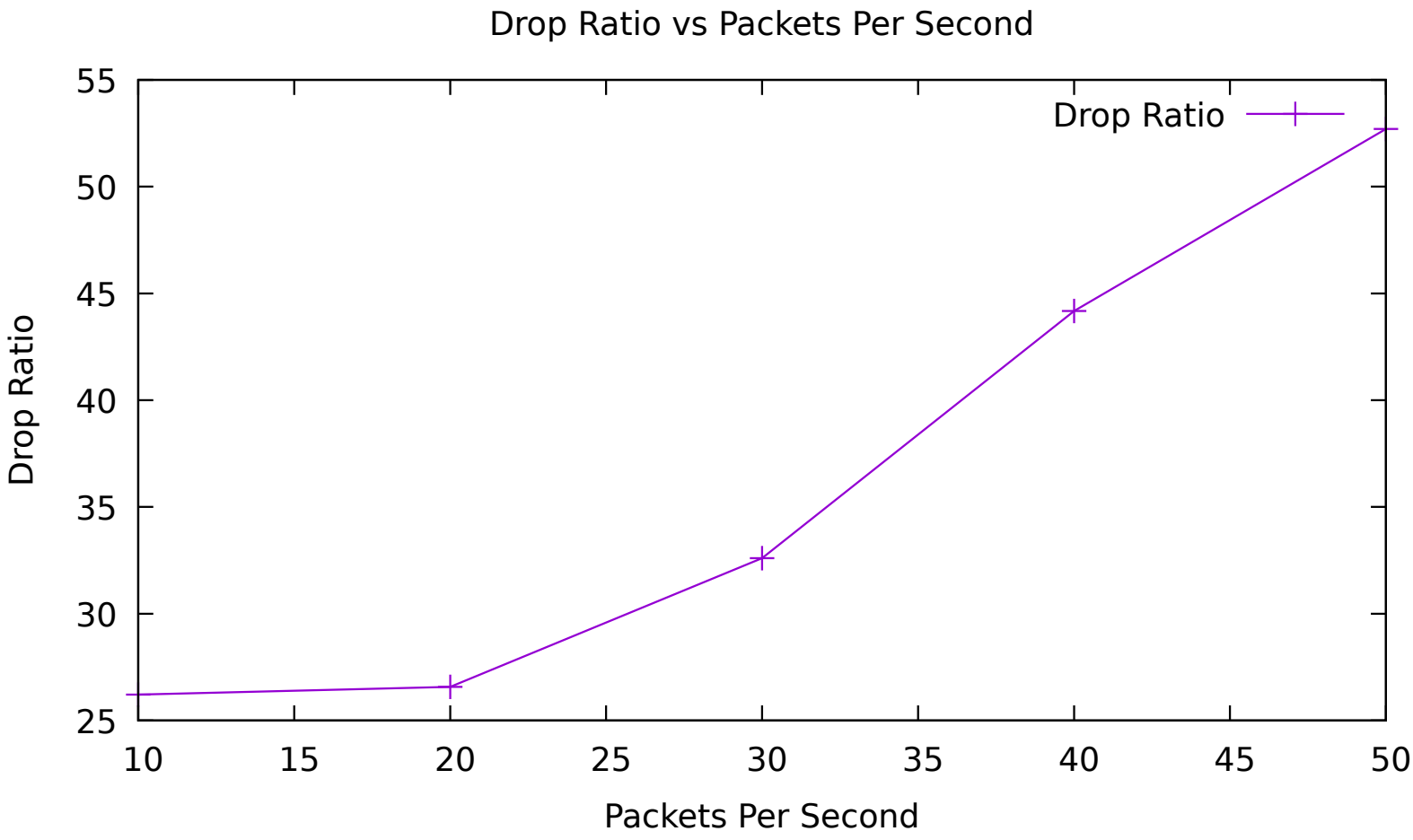

Drop Ratio vs Flow
Drop Ratio
Flow
Drop Ratio
54
52
50
48
46
44
42
40
38
36
34
10
15
20
25
30
35
40
45
50

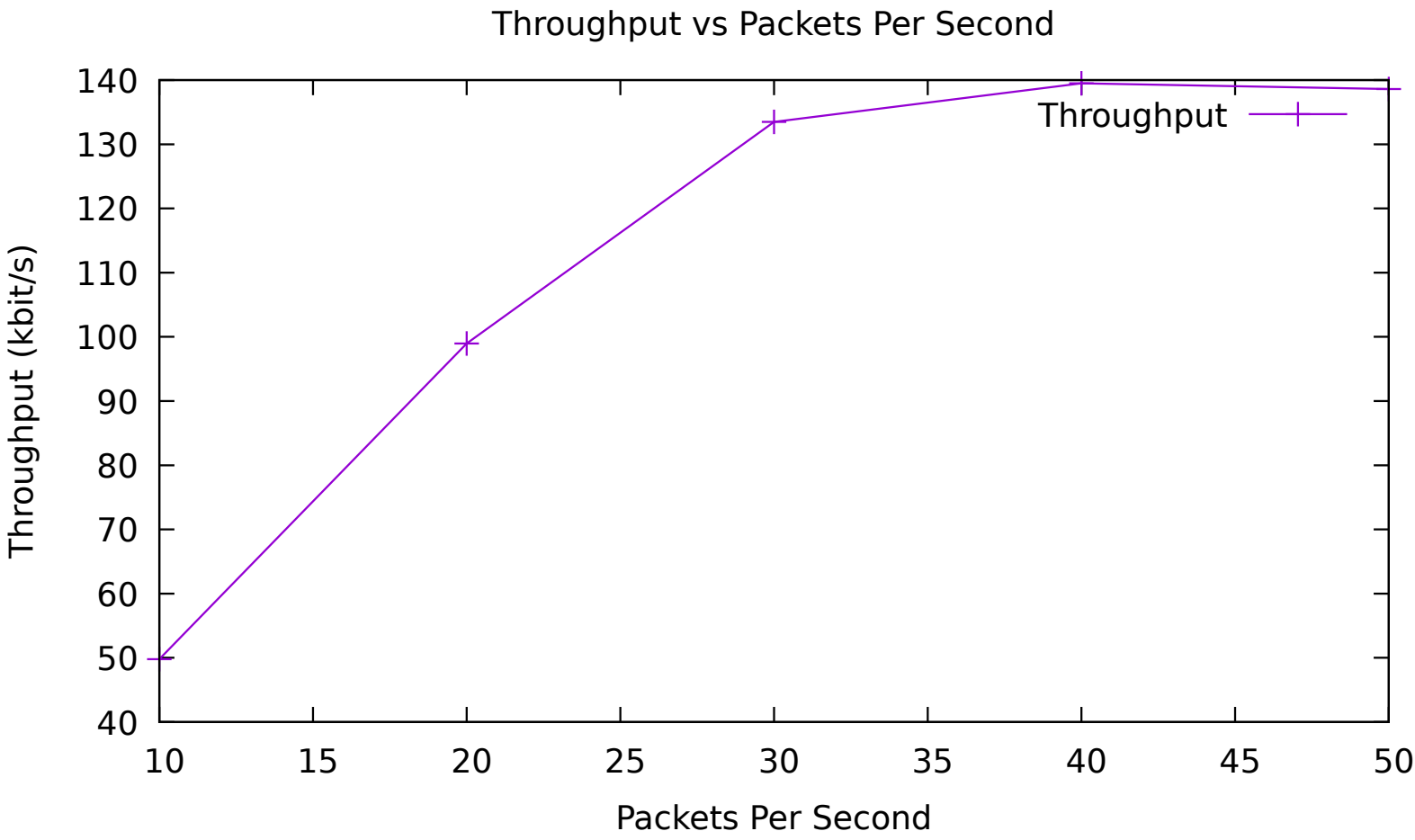

Throughput vs Flow
Throughput
Throughput (kbit/s)
Flow
90
80
70
60
50
40
30
20
10
15
20
25
30
35
40
45
50

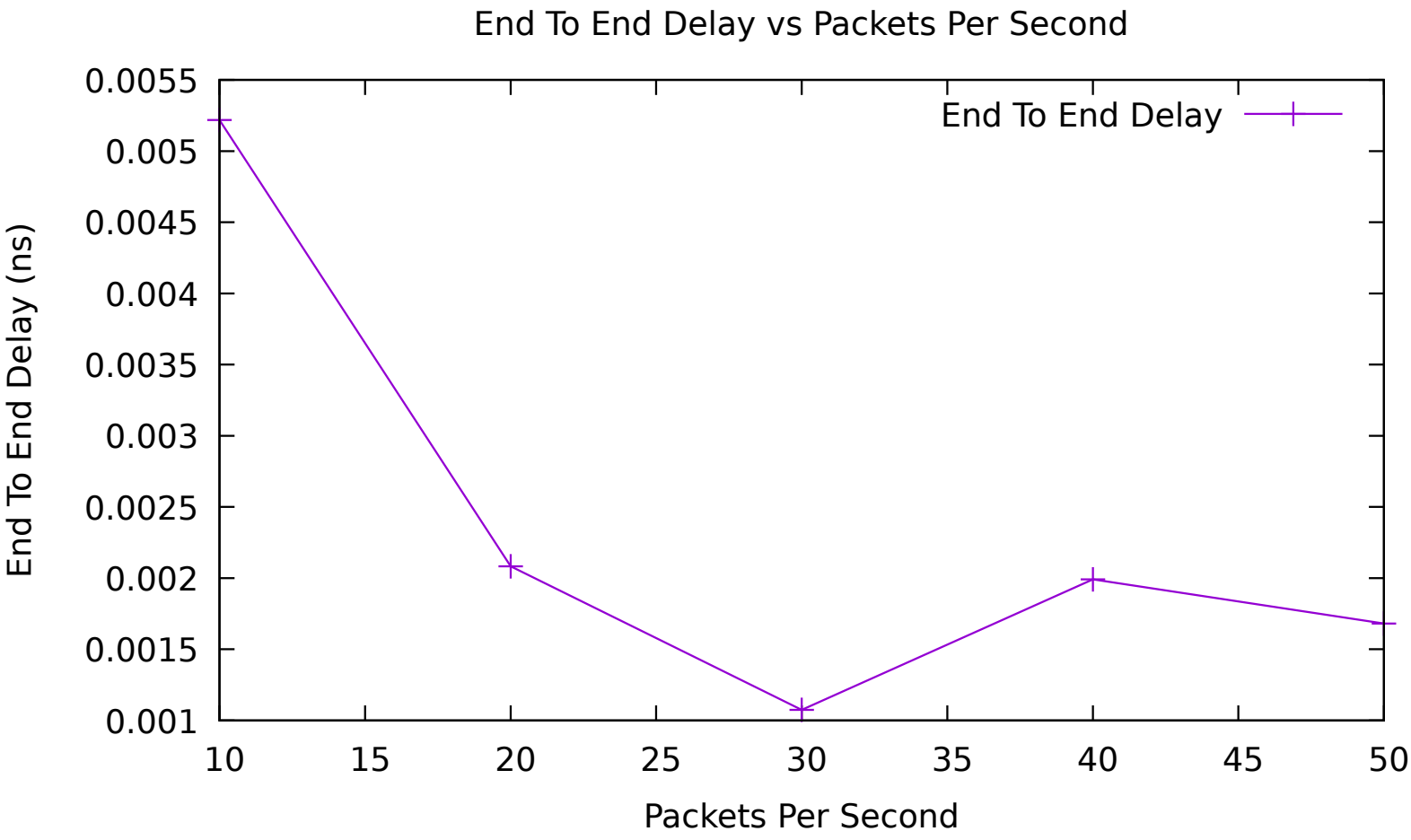

End To End Delay vs Flow
End To End Delay
End To End Delay (ns)
0.0105
0.01
0.0095
0.009
0.0085
0.008
0.0075
0.007
0.0065
0.006
0.0055
0.005
10
15
20
25
30
35
40
45
50
Flow

Varying Number of Packets/Second


Delivery Ratio vs Packets Per Second
Delivery Ratio
Delivery Ratio
70
65
60
55
50
45
40
35
10
15
20
25
30
35
40
45
50
Packets Per Second


Drop Ratio vs Packets Per Second
Drop Ratio
Packets Per Second
Drop Ratio
55
50
45
40
35
30
25
10
15
20
25
30
35
40
45
50


Throughput vs Packets Per Second
Throughput (kbit/s)
Packets Per Second
Throughput
140
130
120
110
100
90
80
70
60
50
40
10
15
20
25
30
35
40
45
50


End To End Delay vs Packets Per Second
End To End Delay
End To End Delay (ns)
0.0055
0.005
0.0045
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
10
15
20
25
30
35
40
45
50
Packets Per Second

7 Summary

7.1 Explanation of Results found in 802.11 static network

Varying Number of Nodes

- **Throughput** : Decreases as increase of node congesting the network flow more
- **End-to-End Delay** : Increases because of more congested network
- **Delivery Ratio** : Decreases with increase of more nodes congested in same network
- **Drop Ratio** : Starts to increase a little bit because of network congestion.

Varying Number of Flows

- **Throughput** : Increases due to more flow in the network
- **End-to-End Delay** :Decreases due to more flow in the network
- **Delivery Ratio** : Shows a trend of decrease with increasing flows
- **Drop Ratio** : Increases because of network congestion.

Varying Number of Packets/Second

- **Throughput** : Increases due to more packets flowing in the network.
- **End-to-End Delay** :Decreases due to more packets flowing in the network.
- **Delivery Ratio** : Shows a trend of decrease with more packets flowing.
- **Drop Ratio** : Increases because of network congestion due to more packet flow.

Varying Number of Coverage Area

- **Throughput** : Decreases due to large area and randomness in the network.
- **End-to-End Delay** :Increases due to larger network.
- **Delivery Ratio** : Shows a trend of decrease larger area network.
- **Drop Ratio** : Increases because of randomness in a large network.

7.2 Explanation of Results found in 802.15.4 static network

Varying Number of Nodes

- **Throughput** :No common pattern of monotonic increase or decrease shown.
- **End-to-End Delay** : Increases because of more congested network
- **Delivery Ratio** : Decreases with increase of more nodes congested in same network
- **Drop Ratio** : Starts to increase a little bit because of network congestion.

Varying Number of Flows

- **Throughput** : Increases due to more flow in the network
- **End-to-End Delay** :Increases due to more flow and congestion in the network
- **Delivery Ratio** : Shows a trend of decrease with increasing flows
- **Drop Ratio** : Increases because of network congestion.

Varying Number of Packets/Second

- **Throughput** : Increases due to more packets flowing in the network.
- **End-to-End Delay** :Decreases due to more packets flowing in the network.
- **Delivery Ratio** : Shows a trend of decrease with more packets flowing.
- **Drop Ratio** : Increases because of network congestion due to more packet flow.

Varying Number of Coverage Area

- **Throughput** : Decreases due to large area and randomness in the network.
- **End-to-End Delay** :Increases due to larger network.
- **Delivery Ratio** : Shows a trend of decrease larger area network.
- **Drop Ratio** : Increases because of randomness in a large network.

7.3 Explanation of Performance Metrics Result in RTT/RTO modification algorithm

The PeakHopper paper calculated three performance metrics and I analysed two of them in my implementation.From the mean retransmission graph , it's seen that modified algorithm provides a lower mean retransmission than default Jacobson's implementation.It was expected because a lower mean indicates that retransmission timer follows a narrow waiting time and a higher mean indicates to a longer waiting time.

Another metric,known as the mean of RTO_N/RTT_L where RTO_N is the newly calculated RTO and RTT_L is the last RTT sample prior to the RTO update. We then calculated P as the mean value of this fraction over an entire simulation run.This mean is found to be lower in modified algorithm than in default implementation which is expected too.A retransmission timer with lower mean of this ratios is expected to perform better.

From these mean calculations , an explanation of those sudden rise of RTO can be given.Since a mean over the entire simulation run is taken and those spikes are very transient , therefore , they don't contribute much to the performance metrics calculated in the paper.Thus , the implementation appears to perform in line with our expectation.