CSE 322
Computer Networks Sessional

Report on NS3 Project
Timely: An RTT-based Congestion Control Algorithm

Prepared by:
1705064 - Md. Tanvir Raihan
Supervisor: Syed Md. Mukit Rashid

Introduction

Datacenter transports aim to deliver low latency messaging together with high throughput.To meet these requirements, datacenter transports must simultaneously deliver high bandwidth (¿¿Gbps) and utilization at low latency (¡¡msec).Since traditional loss-based transports do not meet these strict requirements,we need somw alternative so that completion latency can be minimized.RTT is a fine-grained measure of congestion that comes with every acknowledgment.Further, it requires no support from network switches.So, **Transport Informed by MEasurement of LatencY (TIMELY)** takes advantage of it and uses a congestion control algorithm purely based on RTT instead of traditional window based approach.This algorithm ensures low latency while maintaining high bandwidth. The algorithm also ensures fairness within the network.

Network Topology description

Task A(Wireless high-rate Mobile)

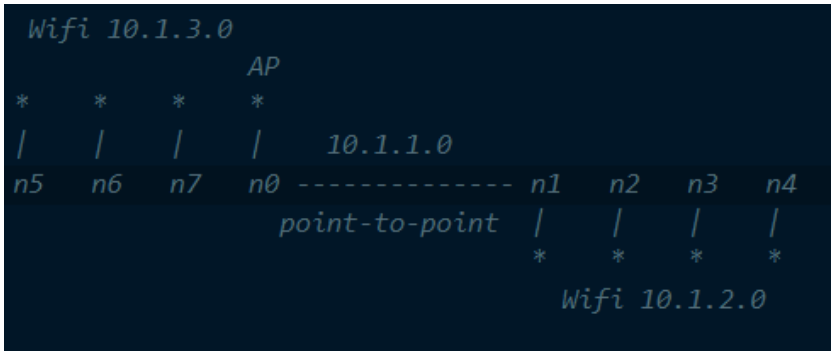

Figure 1: Topology Used for Wireless High Rate

There are 3 networks in total. The first one is between n_0 and n_1 which is a point to point connection. Nodes n_0 and n_1 are called the Access Point(AP) nodes. On the right and left side, there are two Wifi Networks with each having variable number of nodes including one of the access point (AP) nodes.

Task A(Wireless low-rate Static)

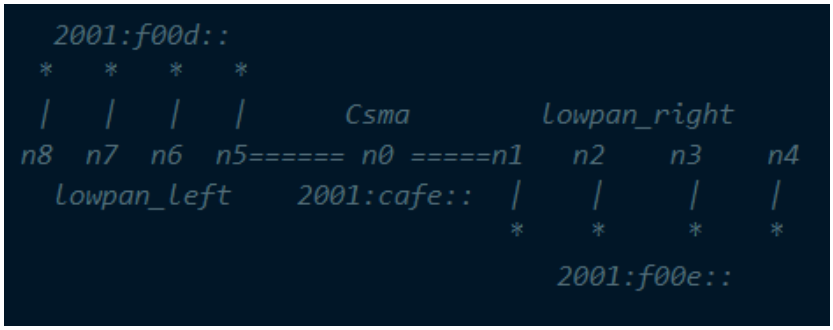

Figure 2: Topology Used for Wireless High Rate

There are 3 networks in total.Two of them are low-rate wpan and there is one node in the middle which is connected to the wpans through wired(Csma) connection.These networks use IPv6 address instead of IPv4 address.

Task B

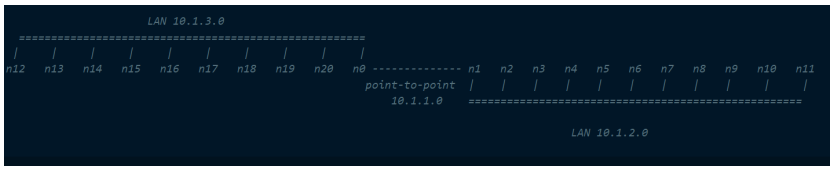

Figure 3: Topology Used for Implementing Proposed Algorithm

There are 3 networks in total. The first one is between n_0 and n_1 which is a point to point connection. Nodes n_0 and n_1 are called the Access Point(AP) nodes. On the right and left side, there are two Wifi Networks with each having variable number of nodes including one of the access point (AP) nodes.

Variation of Parameters

Task A(Wireless high-rate Mobile)

- Number of Nodes is varied as 20,40,60,80,100.
- Number of flows is varied as 10,20,30,40,50.
- Number of Packets per second is varied as 100,200,300,40,50.
- Speed of Nodes is varied as 5,10,15,20,25 (meter/second).

Task A(Wireless low-rate Static)

- Number of Nodes is varied as 20,40,60,80,100.
- Number of flows is varied as 10,20,30,40,50.
- Number of Packets per second is varied as 100,200,300,40,50.
- Coverage area is varied as 100,200,300,400,500 meter

Task B

- Number of flows varied as 10,20,30,40,50
- Number of Nodes varied as 20,40,60,80,100

Overview of Proposed Algorithm

My proposed algorithm is **TcpTimely - An RTT-based Congestion Control for Data Center** which uses variation of RTT or RTT gradient to detect the onset of congestion and adjust data sending rate accordingly. The algorithm uses rate of data transmission rather than traditional window-based congestion control approach.The algorithm maintains a single rate for each connection and updates it on every completion of events. It uses two threshold namely low threshold (T_{low}) and high threshold(T_{high}). The algorithm increments the data sending rate by step delta(generally 10Mbps) and decrements the rate by multiplying with a decrement factor beta. It also uses normalized gradient to update the pacing rate.

Algorithm 1: TIMELY congestion control.

Data: new_rtt
Result: Enforced rate
new_rtt_diff = new_rtt - prev_rtt ;
prev_rtt = new_rtt ;
rtt_diff = (1 - α) · rtt_diff + α · new_rtt_diff ;
▷ α: EWMA weight parameter
normalized_gradient = rtt_diff / minRTT ;
if *new_rtt* < T_{low} **then**
  rate ← rate + δ ;
  ▷ δ: additive increment step
  return;
if *new_rtt* > T_{high} **then**
  rate ← rate · $\left(1 - \beta \cdot \left(1 - \frac{T_{high}}{new\_rtt}\right)\right)$;
  ▷ β: multiplicative decrement factor
  return;
if *normalized_gradient* ≤ *0* **then**
  rate ← rate + N · δ ;
  ▷ N = 5 if gradient<0 for five completion events
  (HAI mode); otherwise N = 1
else
  rate ← rate · (1 - β · normalized_gradient)

Figure 4: Proposed Algorithm

The proposed algorithm expects to give better throughput than existing Data Center Tcp protocols(e.g.Dctcp) while maintaining low latency. It also ensures fairness of sharing.

Modifications Made in Simulator to implement Algorithm

For implementing the proposed algorithm, I had to add two new files in ns3. These files are

- tcp-timely.h
- tcp-timely.cc

The header file includes declaration of variables and methods required to implement TcpTimely algorithm. Here are the variables I have used to implement my proposed algorithm:

```
double m_alpha;   //!<EWMA weight parameter
double m_beta;    //!<multiplicative decrement factor
uint32_t m_delta;   //!<additive increment step in Mbps
Time m_lowThreshold;    //!<RTT low threshold
Time m_highThreshold;   //!<RTT high threshold
Time m_baseRtt;                  //!< Minimum of all Timely RTT measurements seen during connection
Time m_minRtt;                   //!< Minimum of all RTT measurements within last RTT
double m_rttDiffernce;    //!<difference between two consecutive rtt measurements
uint32_t m_cntRtt;               //!< Number of RTT measurements during last RTT
bool m_doingTimelyNow;            //!< If true, do Timely for this RTT
double m_normalizedGradient;    //!<normalized gradient used to calculate data sending rate
double m_sendingRate;   //!<data sending rate,adjusted according to traffic condition in the network
double m_previousRtt;   //previous rtt used to calculate new rtt difference
```

Figure 5: Variables Used for Proposed Algorithm

And Here are the methods I have declared:

```
void IncreaseRate(Ptr<TcpSocketState> tcb,double rtt);

//in case of new_rtt < low threshold, rate will be incremented by a certain value
void AdditiveIncrement(Ptr<TcpSocketState> tcb);

//in case of new_rtt > high threshold, rate will be decremented by a certain value
void MultiplicativeDecrement(Ptr<TcpSocketState> tcb,double rtt);

//when rate depends on gradient
void CalculateFromGradient(Ptr<TcpSocketState> tcb,double normalized_gradient);
```

Figure 6: Methods Used for Proposed Algorithm

The modifications I had to make to implement my algorithm have been listed here:

- Set the pacing to true in PktsAcked Method

```
tcb->m_pacing = true;
double latest_rtt = static_cast<double>(rtt.GetSeconds());
IncreaseRate(tcb,latest_rtt);
```

Figure 7: Changes in the PktsAcked Method

- Adjust Data Sending Rate by calculating rtt difference

```
void
TcpTimely::IncreaseRate(Ptr<TcpSocketState> tcb,
                        double new_rtt)
{
  double new_rtt_difference = new_rtt - m_previousRtt;
  m_previousRtt = new_rtt;
  m_rttDiffernce = (1-m_alpha)*m_rttDiffernce + m_alpha*new_rtt_difference;
  m_normalizedGradient = m_rttDiffernce/static_cast<double>(m_minRtt.GetSeconds());

  if(new_rtt < static_cast<double>(m_lowThreshold.GetSeconds())){
    AdditiveIncrement(tcb);
  }
  else if(new_rtt > static_cast<double>(m_highThreshold.GetSeconds())){
    MultiplicativeDecrement(tcb,new_rtt);
  }
  else{
    CalculateFromGradient(tcb,m_normalizedGradient);
  }
}
```

Figure 8: Method to Adjust Data Sending Rate

- Increasing Rate

```
void
TcpTimely::AdditiveIncrement(Ptr<TcpSocketState> tcb)
{
  m_sendingRate = m_sendingRate + (m_delta/1000);
  tcb->m_pacingRate = DataRate(m_sendingRate);
}
```

Figure 9: Method to Increase Data Sending Rate

- Decreasing Rate

```
void
TcpTimely::MultiplicativeDecrement(Ptr<TcpSocketState> tcb,double new_rtt)
{

  double ratio = 1 - (static_cast<double>(m_highThreshold.GetSeconds())/new_rtt);
   m_sendingRate = m_sendingRate * (1- (m_beta * ratio));
   tcb->m_pacingRate = DataRate(m_sendingRate);
}
```

Figure 10: Method to Decrement Data Sending Rate

- Calculate rate from gradient

```
void
TcpTimely::CalculateFromGradient(Ptr<TcpSocketState> tcb,double normalized_gradient)
{
  uint32_t n = 5;
  if(normalized_gradient <= 0){
    m_sendingRate = m_sendingRate + static_cast<double>(m_delta * n);
    tcb->m_pacingRate = DataRate(m_sendingRate);
  }
  else{
    m_sendingRate = m_sendingRate * (1 - (m_beta * normalized_gradient));
    tcb->m_pacingRate = DataRate(m_sendingRate);
  }
}
```

Figure 11: Method to Calculate Rate From Gradient

Results of Task A

Task A (Wireless High Rate Mobile)

- Throughput Varying Number of Nodes

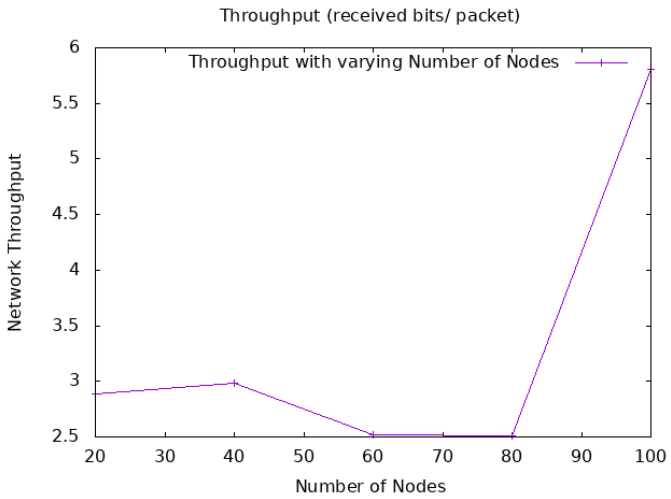

Figure 12: Throughput vs Number of Nodes

- Throughput Varying Number of Flows

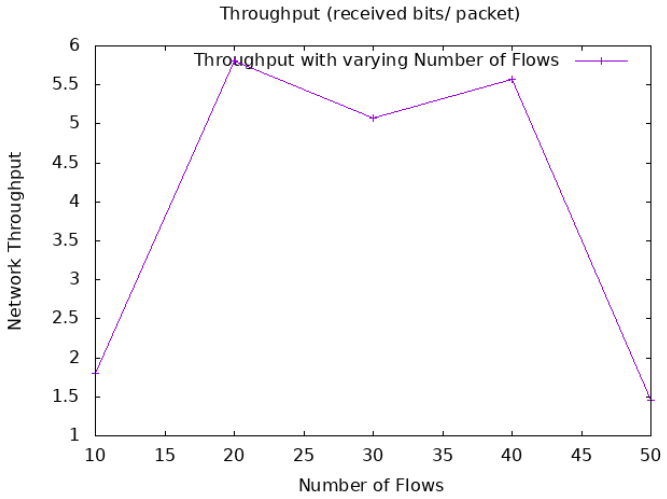

Figure 13: Throughput vs Number of Flows

- Throughput Varying Number of Packets Per Second

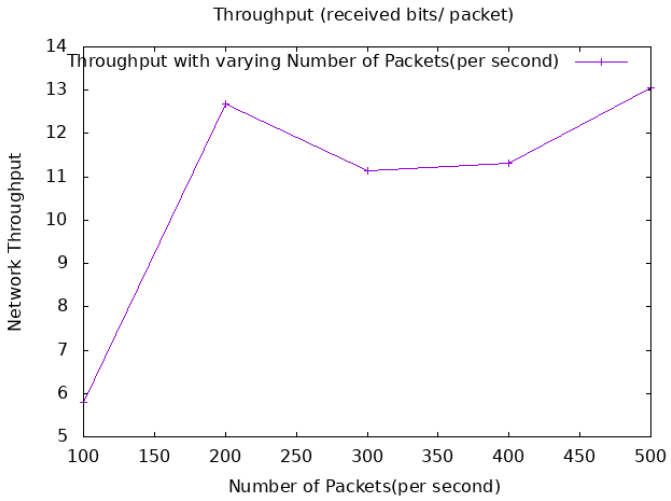

Figure 14: Throughput vs Number of Packets

- Throughput Varying Speed of Nodes

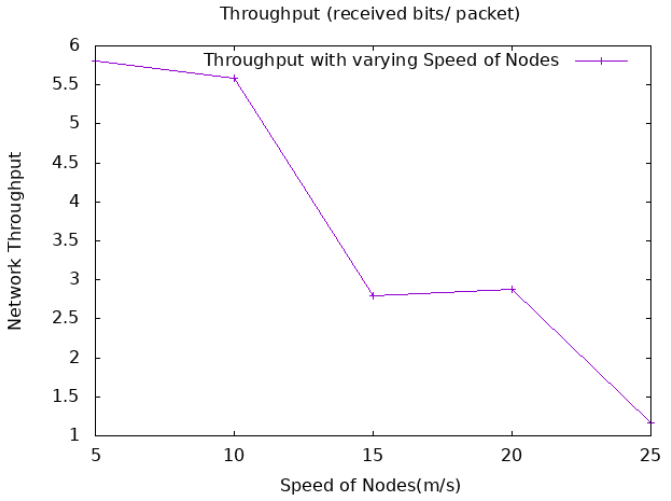

Figure 15: Throughput vs Speed of Nodes

- Packet Delivery Ratio Varying Number of Nodes

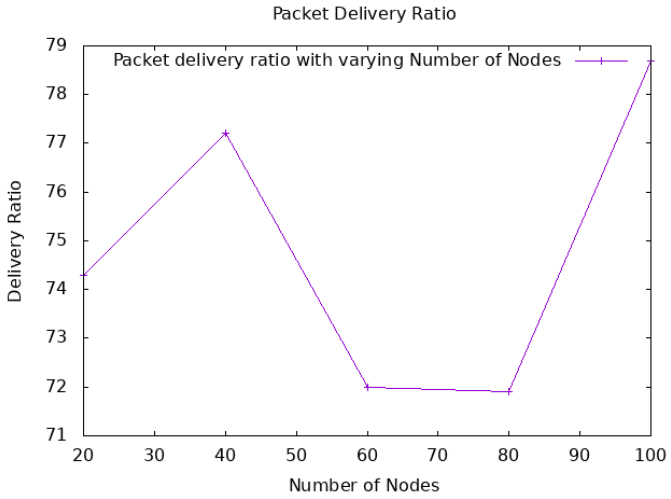

Figure 16: Packet Delivery Ratio vs Number of Nodes

- Packet Delivery Ratio Varying Number of Flows

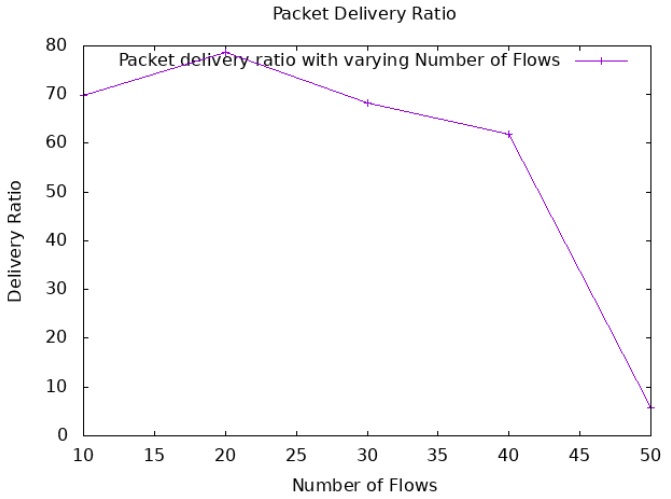

Figure 17: Packet Delivery Ratio vs Number of Flows

- Packet Delivery Ratio Varying Number of Packets Per Second

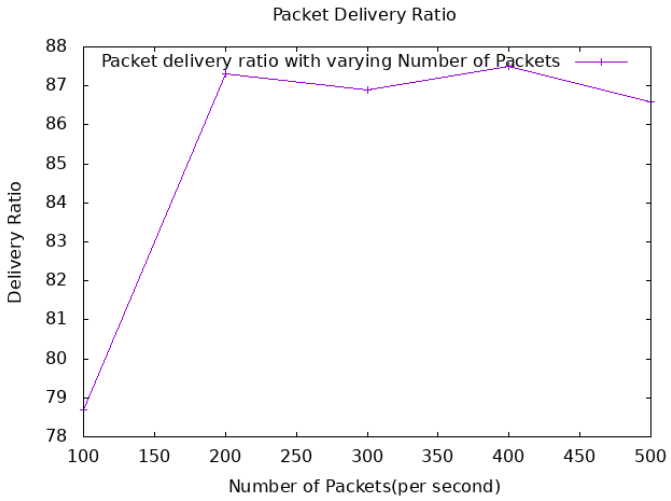

Figure 18: Packet Delivery Ratio vs Number of Packets

- Packet Delivery Ratio Varying Speed of Nodes

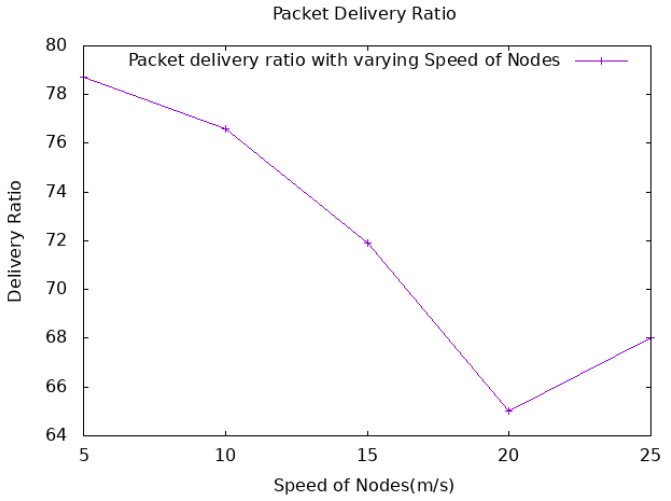

Figure 19: Packet Delivery Ratio vs Speed of Nodes

- Packet Drop Ratio Varying Number of Nodes

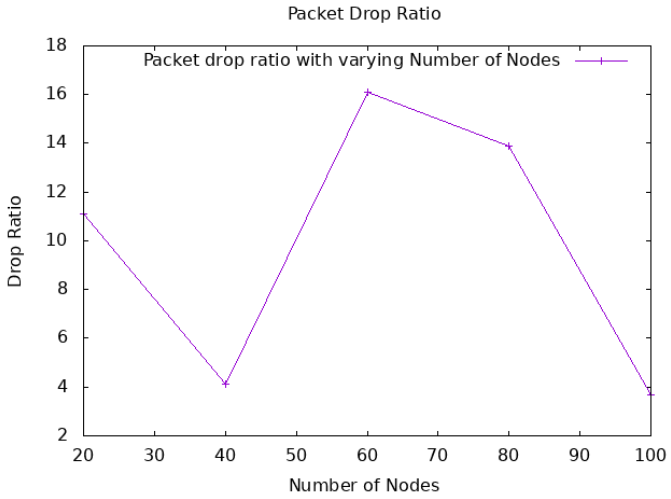

Figure 20: Packet Drop Ratio vs Number of Nodes

- Packet Drop Ratio Varying Number of Flows

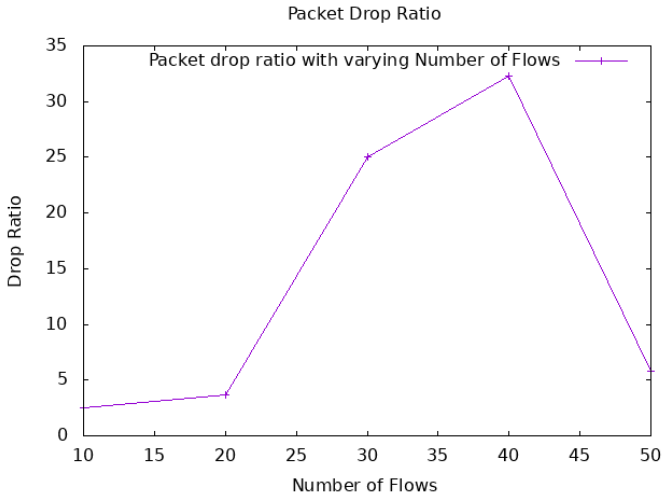

Figure 21: Packet Drop Ratio vs Number of Flows

- Packet Drop Ratio Varying Number of Packets Per Second

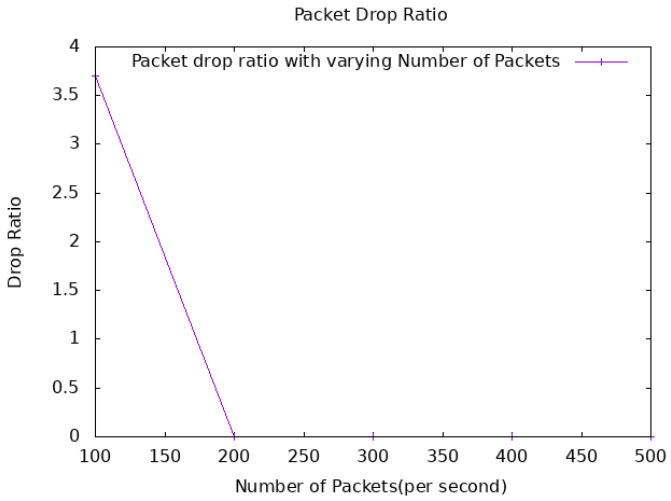

Figure 22: Packet Drop Ratio vs Number of Packets

- Packet Drop Ratio Varying Speed of Nodes

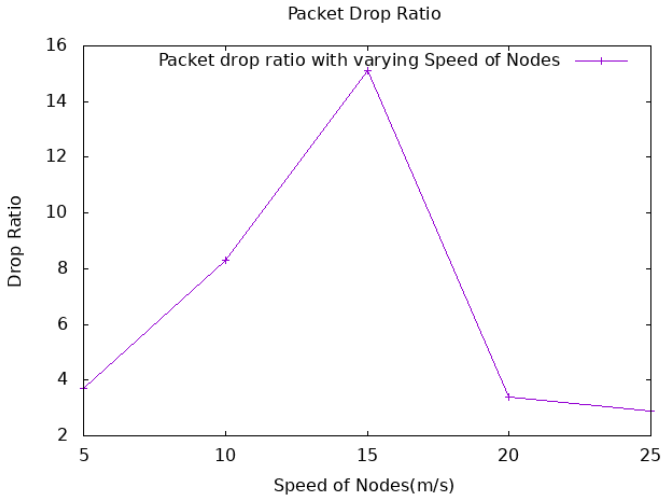

Figure 23: Packet Drop Ratio vs Speed of Nodes

- End to End Delay Varying Number of Nodes

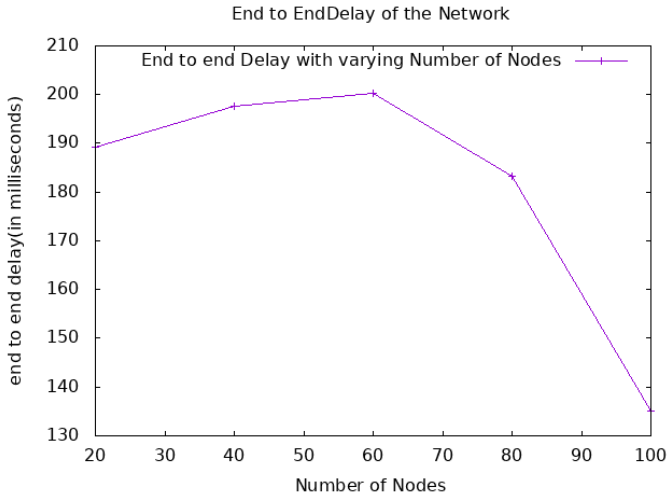

Figure 24: End to End Delay vs Number of Nodes

- End to End Delay Varying Number of Flows

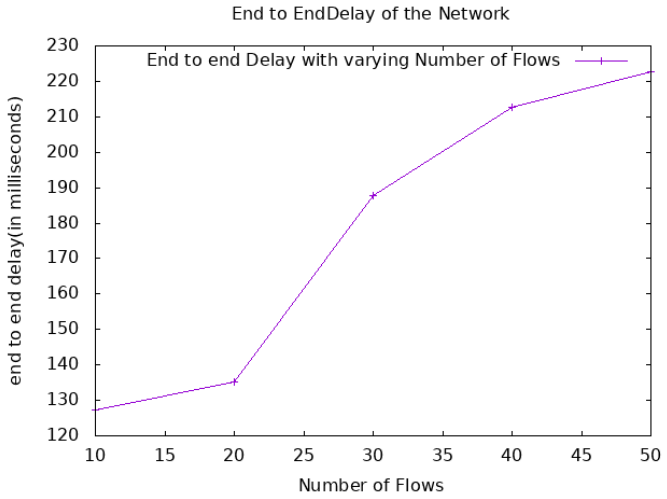

Figure 25: End to End Delay vs Number of Flows

- End to End Delay Varying Number of Packets Per Second

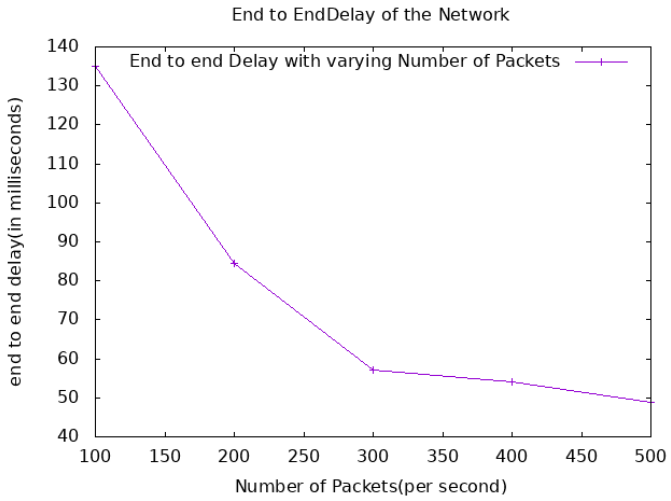

Figure 26: End to End Delay vs Number of Packets

- End to End Delay Varying Speed of Nodes

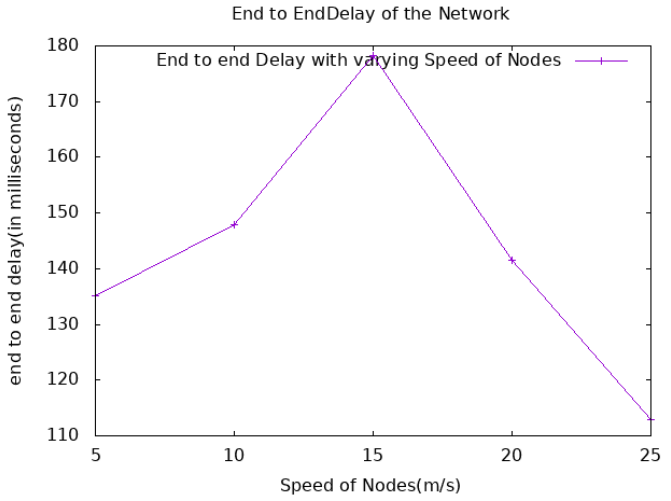

Figure 27: End to End Delay vs Speed of Nodes

Task A (Wireless Low Rate Static)

- Throughput Varying Number of Nodes

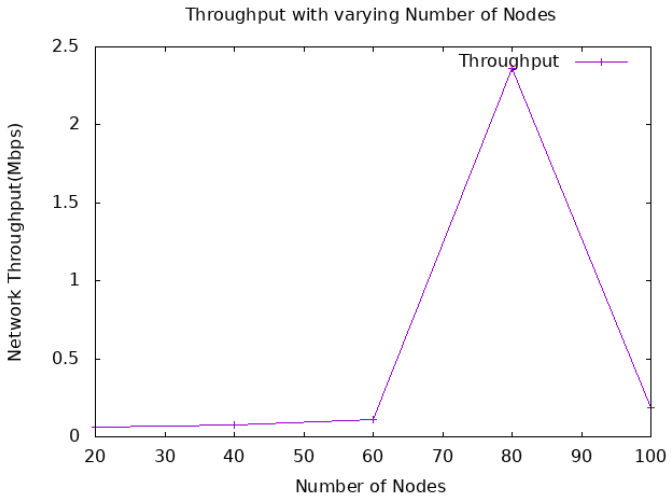

Figure 28: Throughput vs Number of Nodes

- Throughput Varying Number of Flows

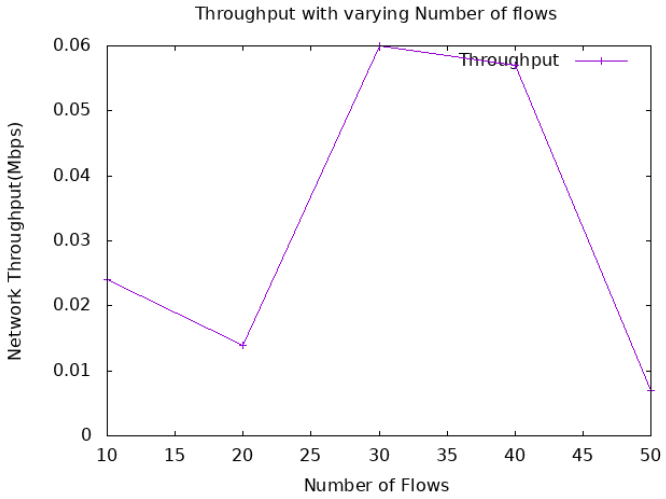

Figure 29: Throughput vs Number of Flows

- Throughput Varying Number of Packets Per Second

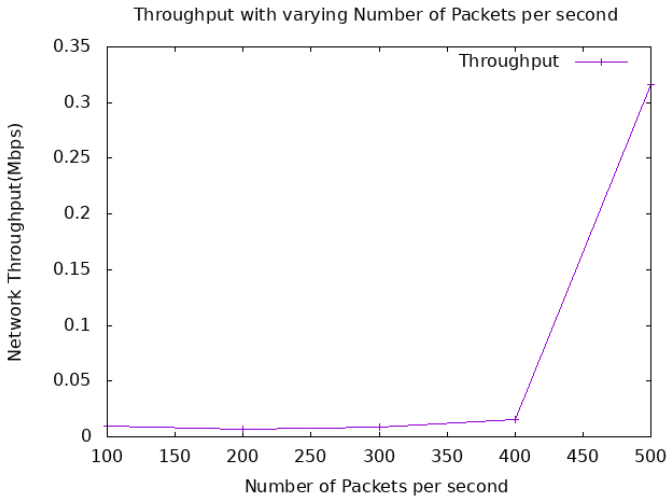

Figure 30: Throughput vs Number of Packets

- Throughput Varying Speed of Nodes

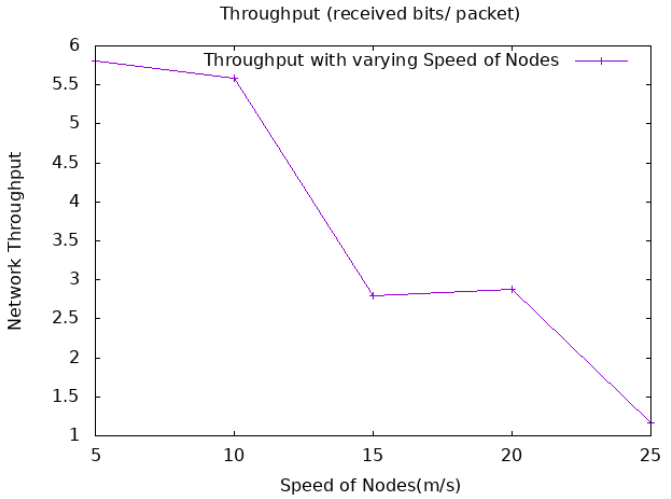

Figure 31: Throughput vs Speed of Nodes

- Packet Delivery Ratio Varying Number of Nodes

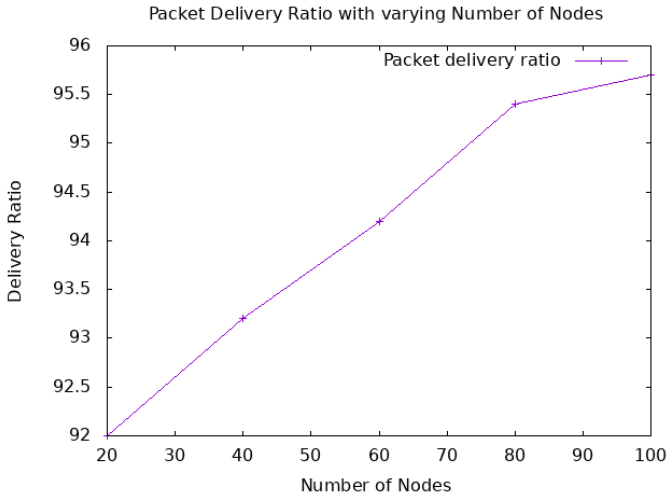

Figure 32: Packet Delivery Ratio vs Number of Nodes

- Packet Delivery Ratio Varying Number of Flows

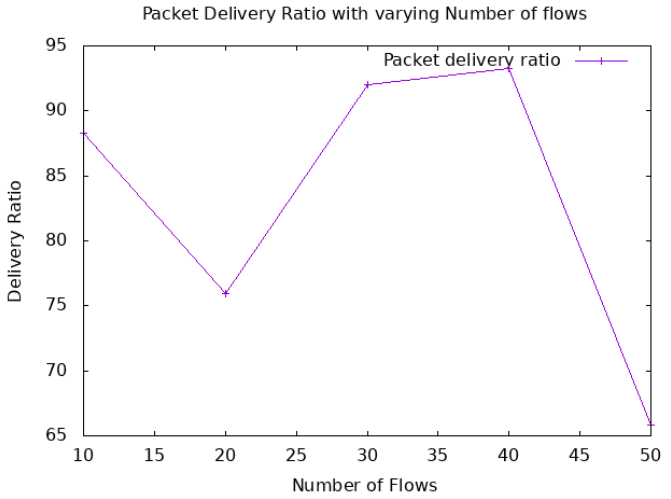

Figure 33: Packet Delivery Ratio vs Number of Flows

- Packet Delivery Ratio Varying Number of Packets Per Second

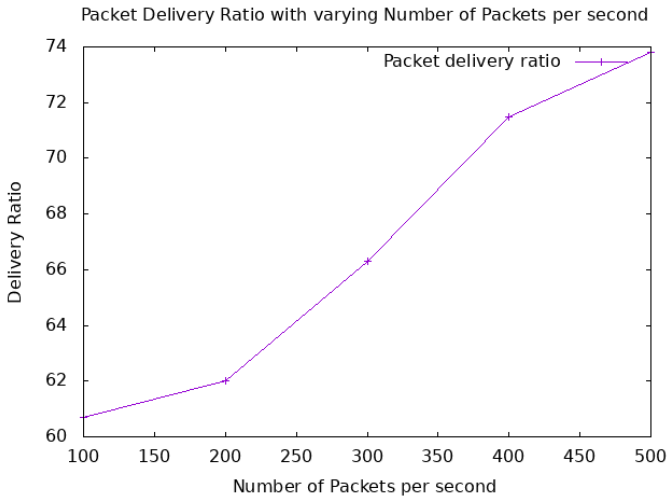

Figure 34: Packet Delivery Ratio vs Number of Packets

- Packet Delivery Ratio Varying Speed of Nodes

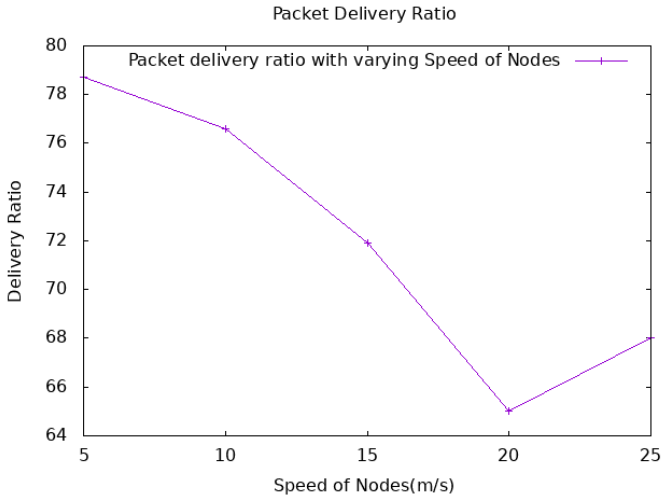

Figure 35: Packet Delivery Ratio vs Speed of Nodes

- Packet Drop Ratio Varying Number of Nodes

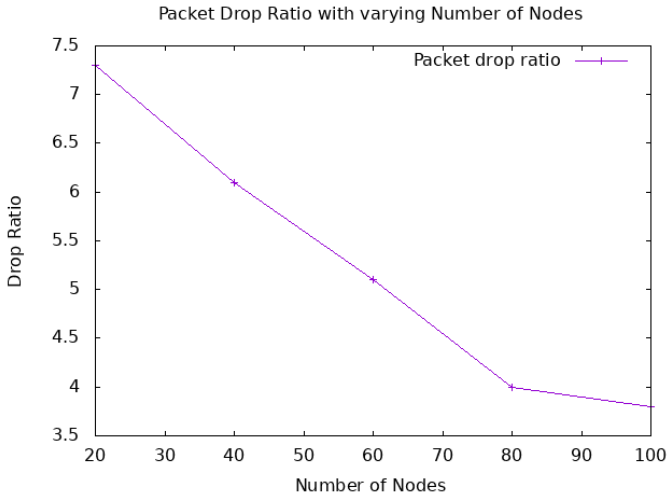

Figure 36: Packet Drop Ratio vs Number of Nodes

- Packet Drop Ratio Varying Number of Flows

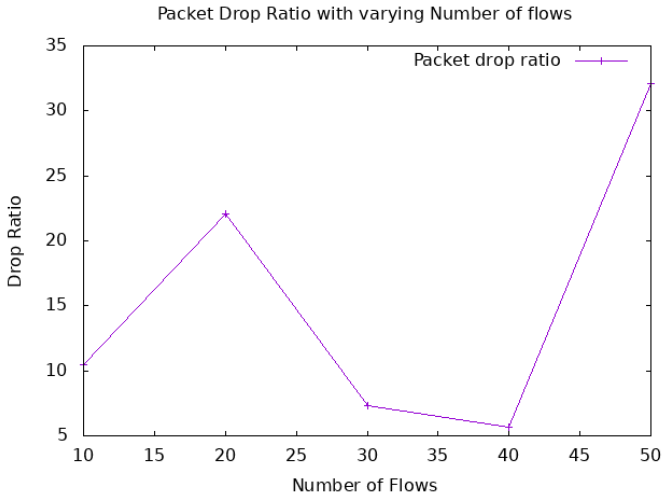

Figure 37: Packet Drop Ratio vs Number of Flows

- Packet Drop Ratio Varying Number of Packets Per Second

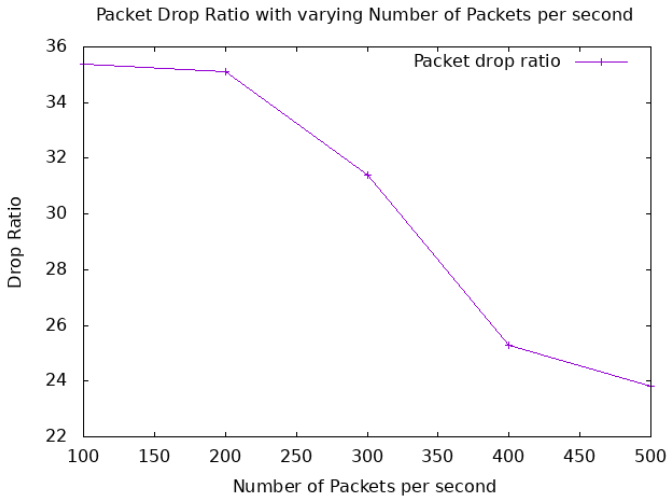

Figure 38: Packet Drop Ratio vs Number of Packets

- Packet Drop Ratio Varying Speed of Nodes

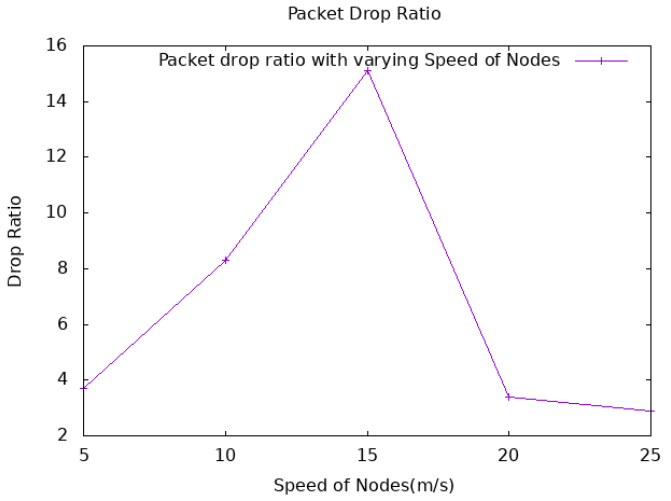

Figure 39: Packet Drop Ratio vs Speed of Nodes

- End to End Delay Varying Number of Nodes

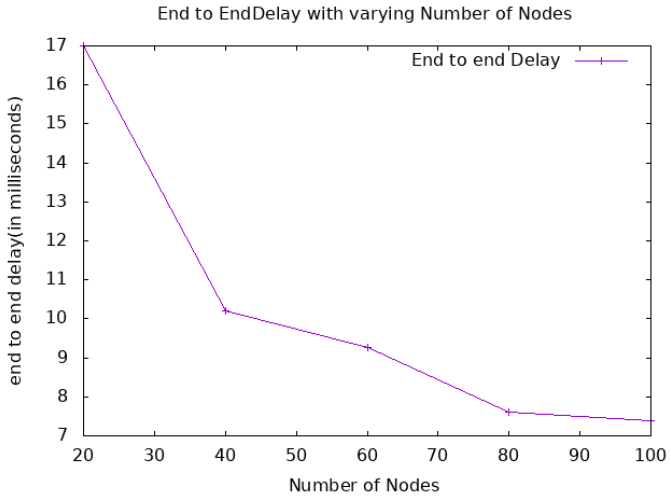

Figure 40: End to End Delay vs Number of Nodes

- End to End Delay Varying Number of Flows

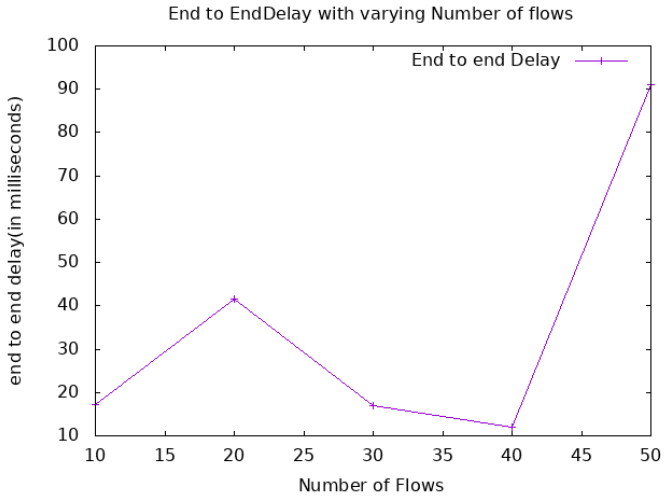

Figure 41: End to End Delay vs Number of Flows

- End to End Delay Varying Number of Packets Per Second

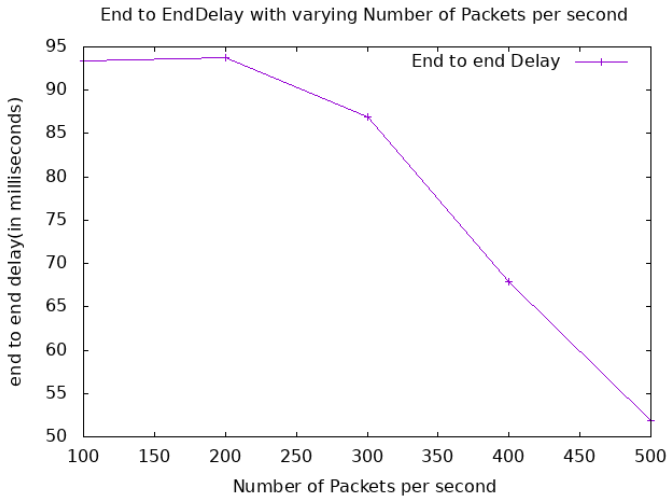

Figure 42: End to End Delay vs Number of Packets

- End to End Delay Varying Speed of Nodes

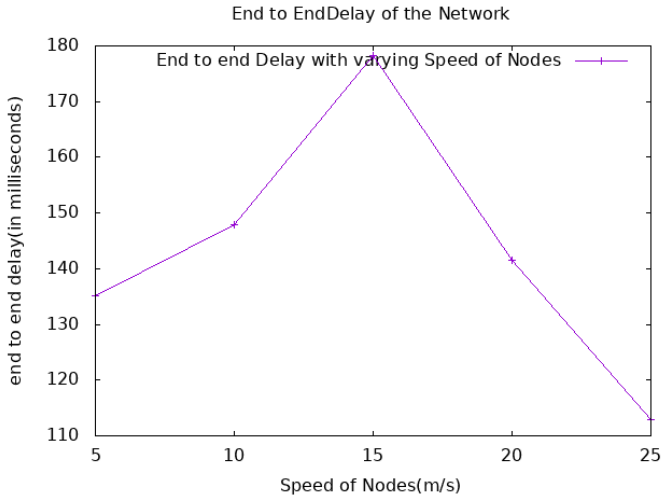

Figure 43: End to End Delay vs Speed of Nodes

Explanation of Task A

In task A, we had to simulate two different kinds of networks - Wireless high rate(mobile) and Wireless low rate(static). In case of **Wireless High Rate(Mobile)** network, from the simulation result, we can observe that **Throughput**,**Packet Drop Ratio** and **End to end delay** of the network increases as we increase number of nodes,flows and packets per second. On the otherhand, **Packet Delivery Ratio** decreases.The reason behind such behavior is because of excessive traffic generated in the network as we increase number of nodes and packets. The opposite behavior can be observed by tweaking the parameters in reverse direction.
In case of **Wireless low rate(static) network**,the metrics do not necessarily follow the trend we observed in case of wireless low rate(mobile).It may produce reasonable output in some cases nut most of the cases the generated results are not in accordance with our expectation.There may be several reasons behind this.Some of them have been listed below:

- There is always an issue of range in case of wireless network and as this is a low rate network the range of each node should have a significant impact on packet transfer. From flowmonitor, we observed that some of the acknowledgement messages did not come. This might be another reason for this poor performance.
- The model-802.15.4 has not been validated against real hardware. So, there can be several loophole that is not known yet
- There seems to be a lot of packet drop in case of low rate data transfer. They may be dropped due to excessive transmission retries or channel access failure.

Results of Task B

- Fairness Comparison with Dctcp with Varying Number of Flows

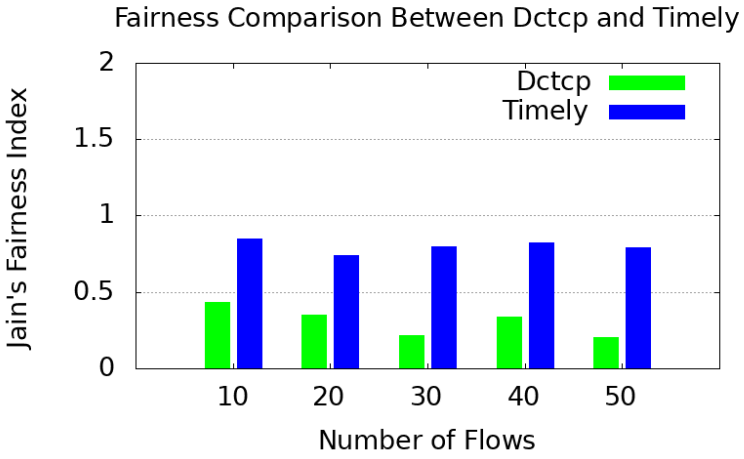

Figure 44: Fairness Comparison

- Throughput with Varying Number of Nodes

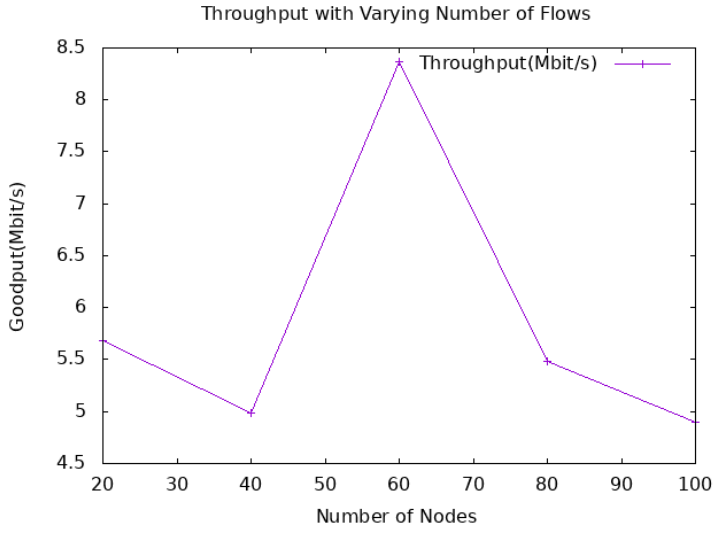

Figure 45: Throughput vs Number of Nodes

- End to end delay with Varying Number of Nodes

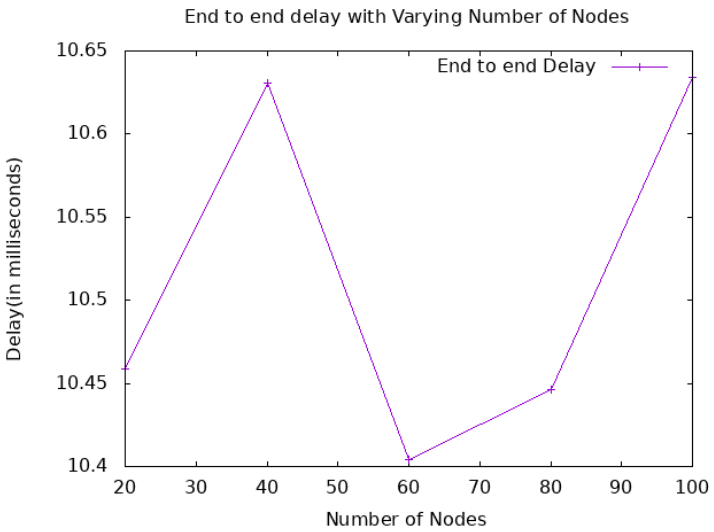

Figure 46: Delay vs Number of Nodes

Explanation of Task B

In task B, I tried to implement a new algorithm called **TCP TIMELY** which focuses on high throughput while maintaining low latency. **TIMELY** also offers better fairness than existing Data Center TCP protocol - **DCTCP**.I have tried to compare the fairness of the two algorithms by dividing the overall flow of the network between them.From the generated result, we can see that **TIMELY** ensures better fairness than **DCTCP**.
I also tried to show the trend of throghput by varying the number of nodes.As we can see from the **Throughput vs Number of nodes** graph, there is some irregularity in the output.This might be becasuse I could not incorporate all the changes to fully implement the algorithm.Finally, I tried to show the **Delay vs Number of Nodes** to show that **TIMELY** maintains low latency.