TCP Libra: Exploring RTT-Fairness for TCP A report on CSE322 Term Project

Md. Olid Hasan Bhuiyan (1705013)
Supervised By
Syed Md. Mukit Rashid
Lecturer

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

February 2022

Contents

1 Introduction

TCP was initially designed to provide a connection-oriented reliable service in the ARPANET environment, which later became the Internet. TCP addresses three major issues: reliability, flow control and congestion control. To achieve the third goal, the sending rate is dynamically adjusted to avoid both service starvation and network overflow. The most widely deployed version, TCP New Reno, implements a congestion control algorithm, known as AIMD (Additive Increase, Multiplicative Decrease). But this algorithm creates RTT-bias. Because of RTT-bias, competing users with larger RTTs will experience higher file download latency. This problem affects big users (e.g., supercomputer researchers), as well as small users.

In this project, we demonstrate a new congestion control algorithm to solve the above issue and thus ensure RTT fairness.

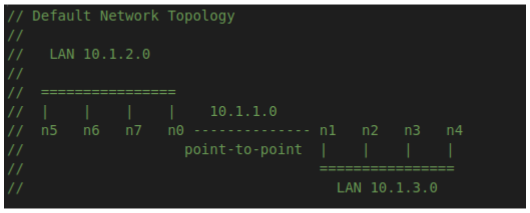
- In Task A, we measure throughput, end to end delay. packet delivery and drop ratio for an experimental topology using the existing congestion control algorithm in ns3.35.
- In Task B, we implement our new congestion control algorithm and compare the results with an existing one.

2 Experimental Topologies

We used a total of three kinds of topology. One of them is wired topology, one is mesh and the other is hybrid.

2.1 Topology Diagrams

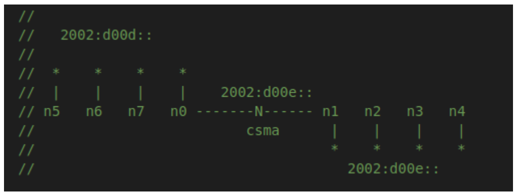
- **Wired**



```
// Default Network Topology
//
//   LAN 10.1.2.0
//
//  ================
//  |    |    |    |    10.1.1.0
//  n5   n6   n7   n0 -------------- n1   n2   n3   n4
//                   point-to-point  |    |    |    |
//                                   ================
//                                     LAN 10.1.3.0
```

Figure 1: Fully Wired Topology

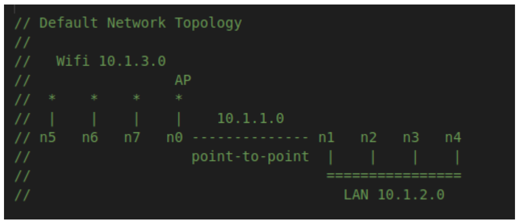
- **Mesh**



```
//
//   2002:d00d::
//
//  *    *    *    *
//  |    |    |    |    2002:d00e::
// n5   n6   n7   n0 -------N------ n1   n2   n3   n4
//                        csma      |    |    |    |
//                                  *    *    *    *
//                                     2002:d00e::
```

Figure 2: Mesh Topology

- **Hybrid**



```
// Default Network Topology
//
//   Wifi 10.1.3.0
//                  AP
//  *    *    *    *
//  |    |    |    |    10.1.1.0
// n5   n6   n7   n0 -------------- n1   n2   n3   n4
//                   point-to-point  |    |    |    |
//                                   ================
//                                     LAN 10.1.2.0
```

Figure 3: Hybrid Topology

3 Parameters Under Variation

In this task, we determined six metrics-

- Network Throughput
- End-to-end delay
- Packet delivery ratio
- Packet drop ratio
- Queue Size(In hybrid topology)
- 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)
- Coverage area(TxRange, 2 x TxRange, 3 x TxRrange, 4 x TxRrange, and 5 x TxRrange)
- Number of packets per second(100,200,300,400,500)

All these metrics are determined from flowmonitor except goodput and Queue size. That means the results are determined in transport layer. Goodput is determined in application layer.

4 Overview of the proposed algorithm

TCP Libra behaves exactly like TCP New Reno except for congestion window management. In fact, Libra differs from TCP New Reno in the following details: In case of a successful transmission-

$$WINDOW_n = WINDOW_n + \frac{1}{WINDOW_n}\frac{\alpha T_n^2}{T_0 + T_n} \tag{1}$$

And in case of 3 DUPACK-.

$$WINDOW_n = WINDOW_n - \frac{T_1 WINDOW_n}{2(T_0 + T_n)} \tag{2}$$

where $window_n$ is the congestion window at step n, α_n a unique control function described in the next section, T_0 and T_1 are fixed parameters and T_n is the RTT at step n. T_1 is the parameter that sets the multiplicative decrease term. T_0 is the parameter that sets the sensitivity of the protocol to the RTT.

4.1 The Alpha Control Function

The design of α was accomplished with the objective of pursuing the following main objectives:

- Increase convergence speed and achieve scalability for the algorithm.
- Keep the algorithm behavior stable.

For the above reasons, the α factor is expressed as the product of two components, namely: $\alpha = SP$, where

- S = Scalability factor
- P = Penalty factor

we set:
$S = k_1 C_r$ where k_1 is a constant and C_r is the capacity of the narrow link.

And $P = e^{-} k_2 (\frac{T_r - T_{min}}{T_{max} - T_{min}})$

In a later section we will show the simulation results for TCP Libra. The simulations have been obtained by substituting $T_0 = 1$, $T_1 = 1$, $k_1 = 2$ and $k_2 = 2$, Packet size = 1000bytes, bottleneck capacity = 100Mbps and delay = 10ms

5 Modifications made in the simulator

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

- tcp-libra.cc
- tcp-libra.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) We have to work on the congestion window and threshold. For this, we need to add some functions and modify some existing functions in TCP NewReno. We also needed some extra attributes for tcp-libra. The modified functions are-

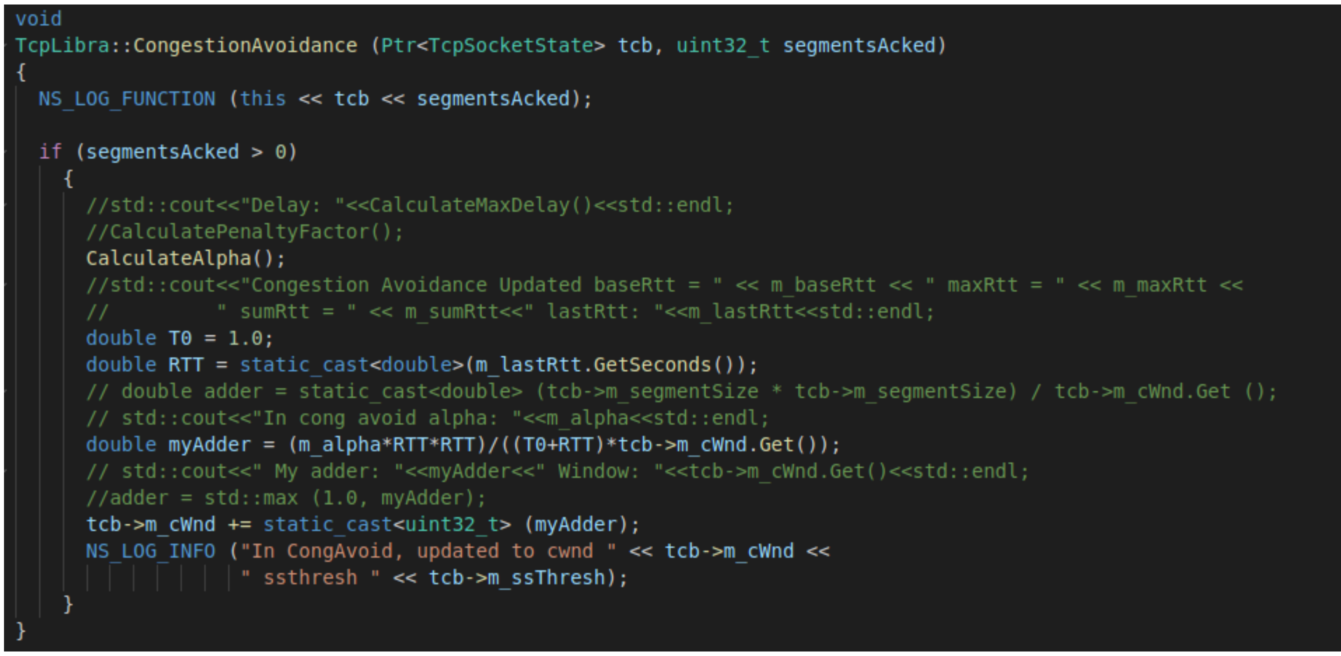
- **CongestionAvoidance**
- **GetSsThresh**
- **dupAck**(in tcp-socket-base.cc)

The newly added functions are-

- **HandleWindowForDupAck**
- **PktsAcked**
- **CalculateAvgDelay**
- **CalculateMaxDelay**
- **CalculateScalabilityFactor**
- **CalculatePenaltyFactor**
- **CalculateAlpha**

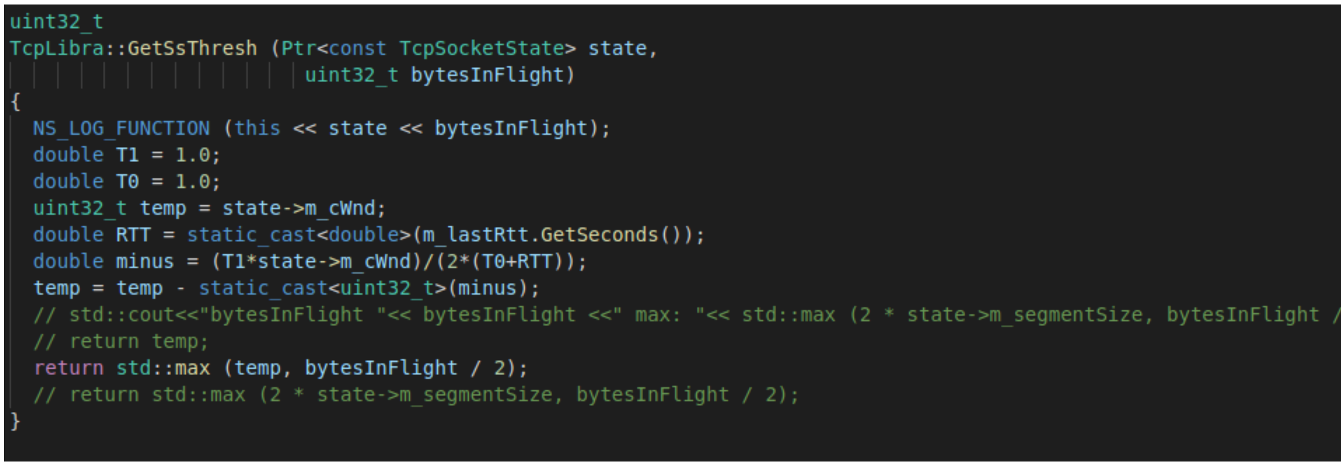
Code snippets of these functions are given below:

5.1 Code Snippets



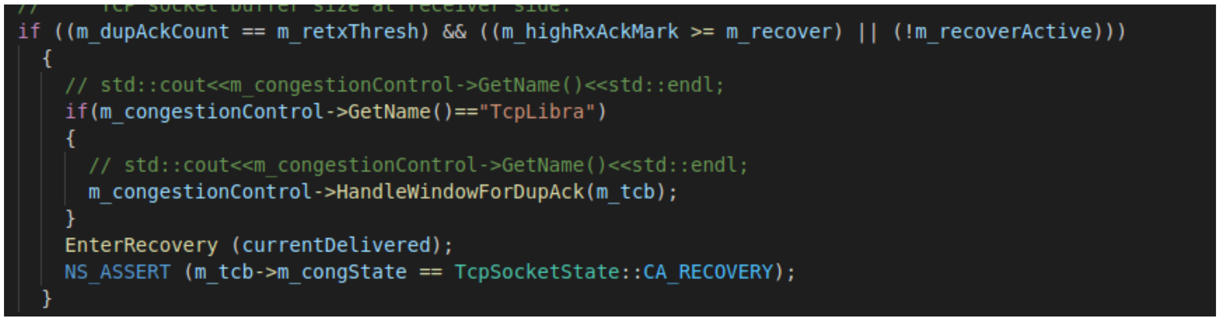
```
void
TcpLibra::CongestionAvoidance (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked)
{
  NS_LOG_FUNCTION (this << tcb << segmentsAcked);

  if (segmentsAcked > 0)
    {
      //std::cout<<"Delay: "<<CalculateMaxDelay()<<std::endl;
      //CalculatePenaltyFactor();
      CalculateAlpha();
      //std::cout<<"Congestion Avoidance Updated baseRtt = " << m_baseRtt << " maxRtt = " << m_maxRtt <<
      //         " sumRtt = " << m_sumRtt<<" lastRtt: "<<m_lastRtt<<std::endl;
      double T0 = 1.0;
      double RTT = static_cast<double>(m_lastRtt.GetSeconds());
      // double adder = static_cast<double> (tcb->m_segmentSize * tcb->m_segmentSize) / tcb->m_cWnd.Get ();
      // std::cout<<"In cong avoid alpha: "<<m_alpha<<std::endl;
      double myAdder = (m_alpha*RTT*RTT)/((T0+RTT)*tcb->m_cWnd.Get());
      // std::cout<<" My adder: "<<myAdder<<" Window: "<<tcb->m_cWnd.Get()<<std::endl;
      //adder = std::max (1.0, myAdder);
      tcb->m_cWnd += static_cast<uint32_t> (myAdder);
      NS_LOG_INFO ("In CongAvoid, updated to cwnd " << tcb->m_cWnd <<
                   " ssthresh " << tcb->m_ssThresh);
    }
}
```

Figure 4: CongestionAvoidance



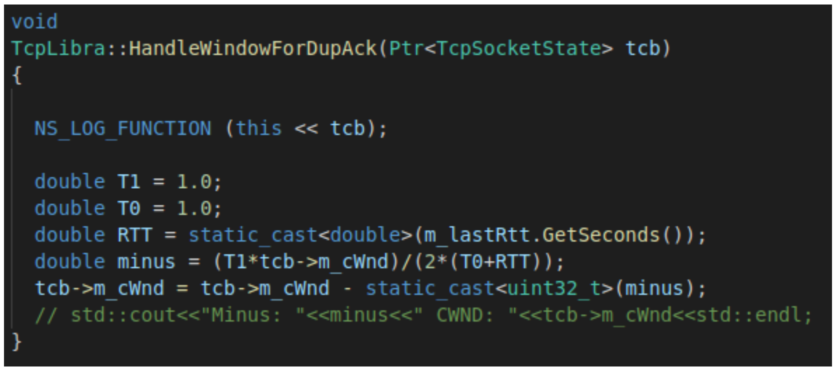
```
uint32_t
TcpLibra::GetSsThresh (Ptr<const TcpSocketState> state,
                         uint32_t bytesInFlight)
{
  NS_LOG_FUNCTION (this << state << bytesInFlight);
  double T1 = 1.0;
  double T0 = 1.0;
  uint32_t temp = state->m_cWnd;
  double RTT = static_cast<double>(m_lastRtt.GetSeconds());
  double minus = (T1*state->m_cWnd)/(2*(T0+RTT));
  temp = temp - static_cast<uint32_t>(minus);
  // std::cout<<"bytesInFlight "<< bytesInFlight <<" max: "<< std::max (2 * state->m_segmentSize, bytesInFlight /
  // return temp;
  return std::max (temp, bytesInFlight / 2);
  // return std::max (2 * state->m_segmentSize, bytesInFlight / 2);
}
```

Figure 5: GetSsThresh



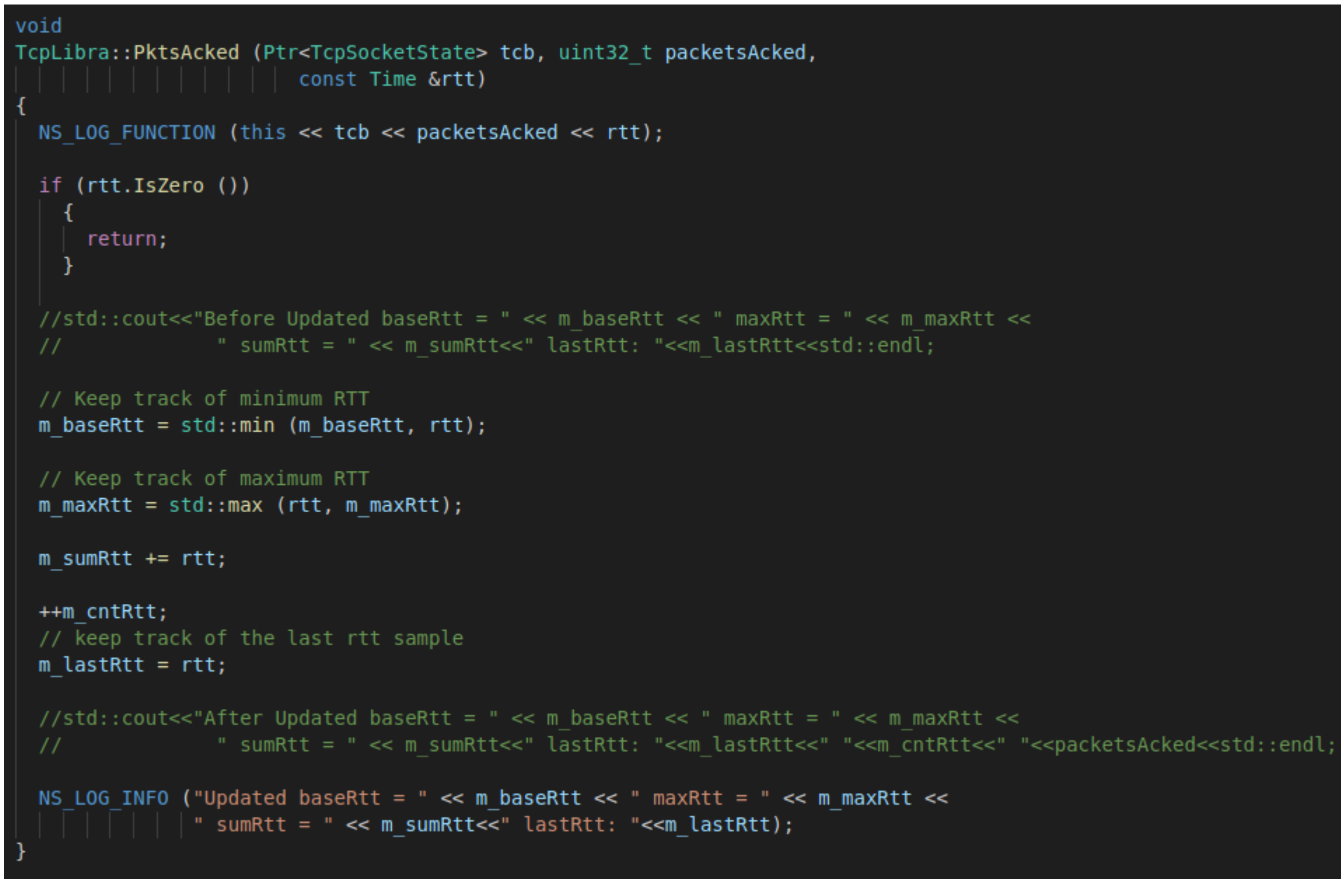
```
//      TCP socket buffer size at receiver side.
if ((m_dupAckCount == m_retxThresh) && ((m_highRxAckMark >= m_recover) || (!m_recoverActive)))
  {
    // std::cout<<m_congestionControl->GetName()<<std::endl;
    if(m_congestionControl->GetName()=="TcpLibra")
    {
      // std::cout<<m_congestionControl->GetName()<<std::endl;
      m_congestionControl->HandleWindowForDupAck(m_tcb);
    }
    EnterRecovery (currentDelivered);
    NS_ASSERT (m_tcb->m_congState == TcpSocketState::CA_RECOVERY);
  }
```

Figure 6: dupAck



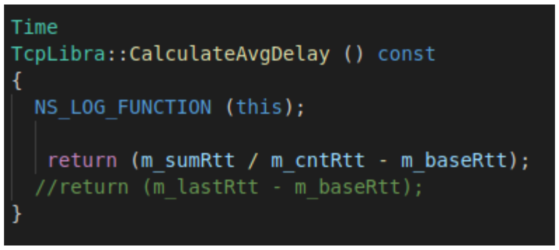
```
void
TcpLibra::HandleWindowForDupAck(Ptr<TcpSocketState> tcb)
{

  NS_LOG_FUNCTION (this << tcb);

  double T1 = 1.0;
  double T0 = 1.0;
  double RTT = static_cast<double>(m_lastRtt.GetSeconds());
  double minus = (T1*tcb->m_cWnd)/(2*(T0+RTT));
  tcb->m_cWnd = tcb->m_cWnd - static_cast<uint32_t>(minus);
  // std::cout<<"Minus: "<<minus<<" CWND: "<<tcb->m_cWnd<<std::endl;
}
```

Figure 7: HandleWindowForDupAck



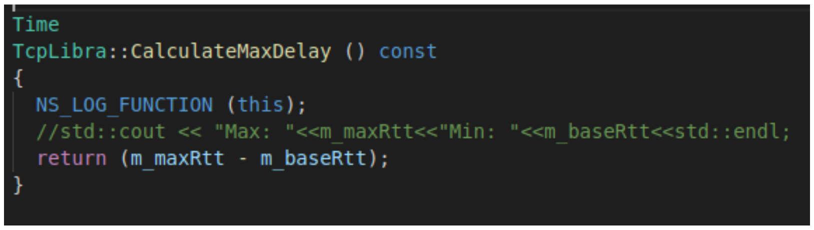
```cpp
void
TcpLibra::PktsAcked (Ptr<TcpSocketState> tcb, uint32_t packetsAcked,
                     const Time &rtt)
{
  NS_LOG_FUNCTION (this << tcb << packetsAcked << rtt);

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

  //std::cout<<"Before Updated baseRtt = " << m_baseRtt << " maxRtt = " << m_maxRtt <<
  //             " sumRtt = " << m_sumRtt<<" lastRtt: "<<m_lastRtt<<std::endl;

  // Keep track of minimum RTT
  m_baseRtt = std::min (m_baseRtt, rtt);

  // Keep track of maximum RTT
  m_maxRtt = std::max (rtt, m_maxRtt);

  m_sumRtt += rtt;

  ++m_cntRtt;
  // keep track of the last rtt sample
  m_lastRtt = rtt;

  //std::cout<<"After Updated baseRtt = " << m_baseRtt << " maxRtt = " << m_maxRtt <<
  //             " sumRtt = " << m_sumRtt<<" lastRtt: "<<m_lastRtt<<" "<<m_cntRtt<<" "<<packetsAcked<<std::endl;

  NS_LOG_INFO ("Updated baseRtt = " << m_baseRtt << " maxRtt = " << m_maxRtt <<
               " sumRtt = " << m_sumRtt<<" lastRtt: "<<m_lastRtt);
}
```

Figure 8: PktsAcked



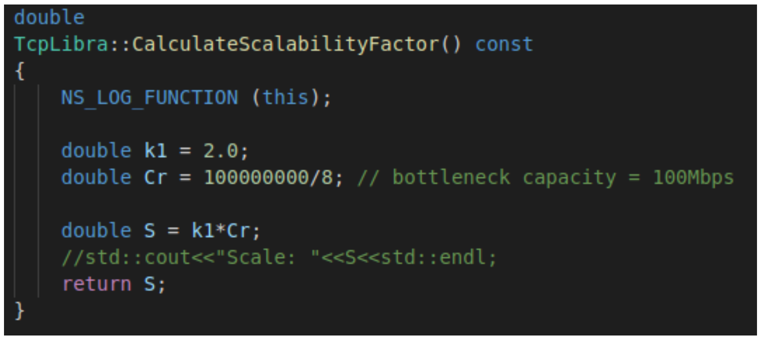
```cpp
Time
TcpLibra::CalculateAvgDelay () const
{
  NS_LOG_FUNCTION (this);

   return (m_sumRtt / m_cntRtt - m_baseRtt);
  //return (m_lastRtt - m_baseRtt);
}
```

Figure 9: CalculateAvgDelay



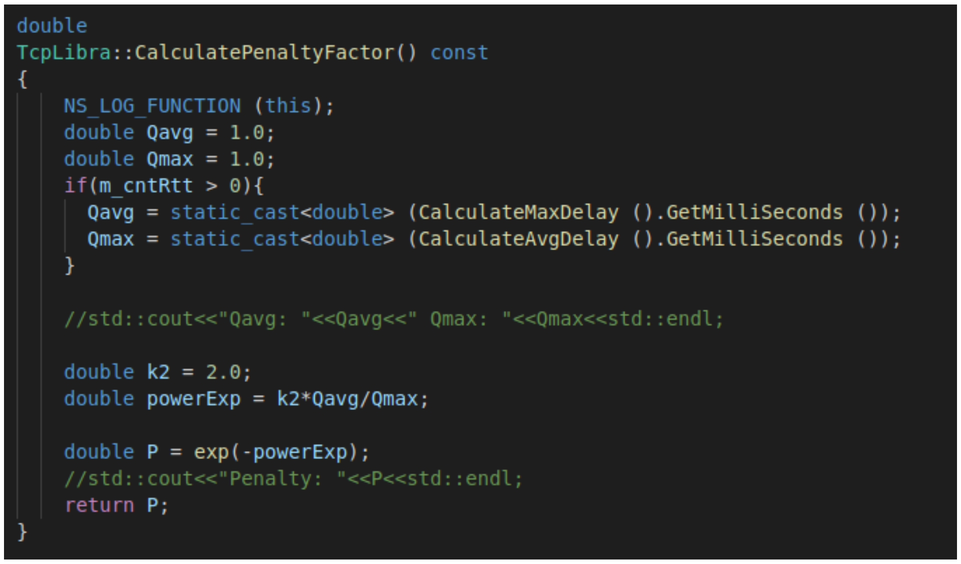
```
Time
TcpLibra::CalculateMaxDelay () const
{
  NS_LOG_FUNCTION (this);
  //std::cout << "Max: "<<m_maxRtt<<"Min: "<<m_baseRtt<<std::endl;
  return (m_maxRtt - m_baseRtt);
}
```

Figure 10: CalculateMaxDelay



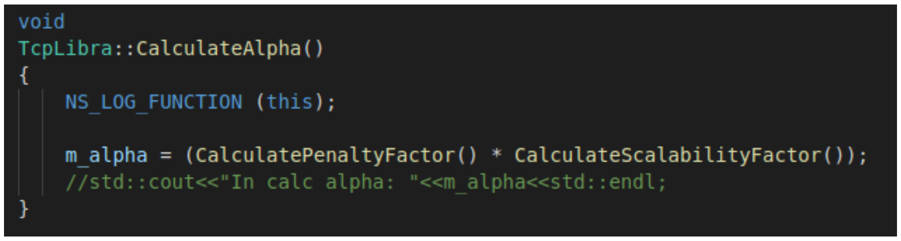
```
double
TcpLibra::CalculateScalabilityFactor() const
{
    NS_LOG_FUNCTION (this);

    double k1 = 2.0;
    double Cr = 100000000/8; // bottleneck capacity = 100Mbps

    double S = k1*Cr;
    //std::cout<<"Scale: "<<S<<std::endl;
    return S;
}
```

Figure 11: CalculateScalabilityFactor



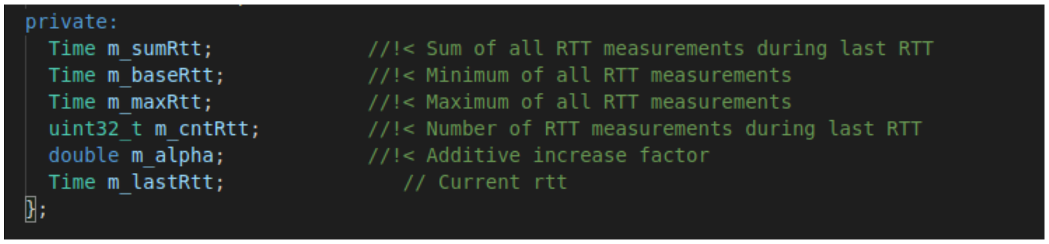
```
double
TcpLibra::CalculatePenaltyFactor() const
{
    NS_LOG_FUNCTION (this);
    double Qavg = 1.0;
    double Qmax = 1.0;
    if(m_cntRtt > 0){
      Qavg = static_cast<double> (CalculateMaxDelay ().GetMilliSeconds ());
      Qmax = static_cast<double> (CalculateAvgDelay ().GetMilliSeconds ());
    }

    //std::cout<<"Qavg: "<<Qavg<<" Qmax: "<<Qmax<<std::endl;

    double k2 = 2.0;
    double powerExp = k2*Qavg/Qmax;

    double P = exp(-powerExp);
    //std::cout<<"Penalty: "<<P<<std::endl;
    return P;
}
```

Figure 12: CalculatePenaltyFactor



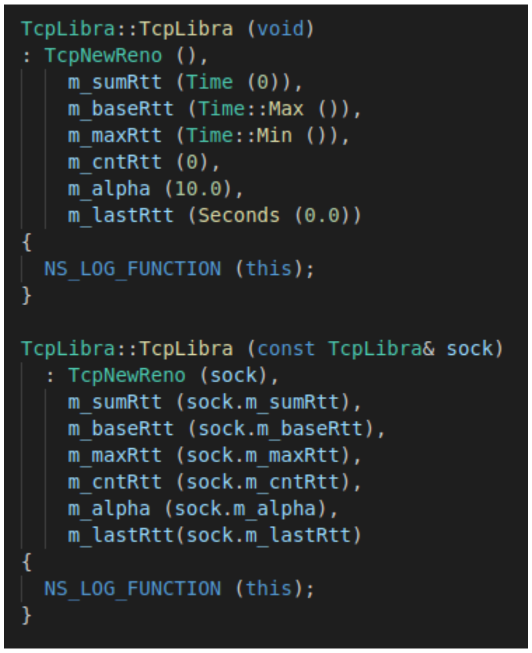
```
void
TcpLibra::CalculateAlpha()
{
    NS_LOG_FUNCTION (this);

    m_alpha = (CalculatePenaltyFactor() * CalculateScalabilityFactor());
    //std::cout<<"In calc alpha: "<<m_alpha<<std::endl;
}
```

Figure 13: CalculateAlpha



```
private:
  Time m_sumRtt;             //!< Sum of all RTT measurements during last RTT
  Time m_baseRtt;            //!< Minimum of all RTT measurements
  Time m_maxRtt;             //!< Maximum of all RTT measurements
  uint32_t m_cntRtt;         //!< Number of RTT measurements during last RTT
  double m_alpha;            //!< Additive increase factor
  Time m_lastRtt;               // Current rtt
};
```

Figure 14: Extra attributes



```
TcpLibra::TcpLibra (void)
: TcpNewReno (),
    m_sumRtt (Time (0)),
    m_baseRtt (Time::Max ()),
    m_maxRtt (Time::Min ()),
    m_cntRtt (0),
    m_alpha (10.0),
    m_lastRtt (Seconds (0.0))
{
  NS_LOG_FUNCTION (this);
}

TcpLibra::TcpLibra (const TcpLibra& sock)
  : TcpNewReno (sock),
    m_sumRtt (sock.m_sumRtt),
    m_baseRtt (sock.m_baseRtt),
    m_maxRtt (sock.m_maxRtt),
    m_cntRtt (sock.m_cntRtt),
    m_alpha (sock.m_alpha),
    m_lastRtt(sock.m_lastRtt)
{
  NS_LOG_FUNCTION (this);
}
```

Figure 15: Attribute Initialization

6 Results with graphs

6.1 Task A(Wired)

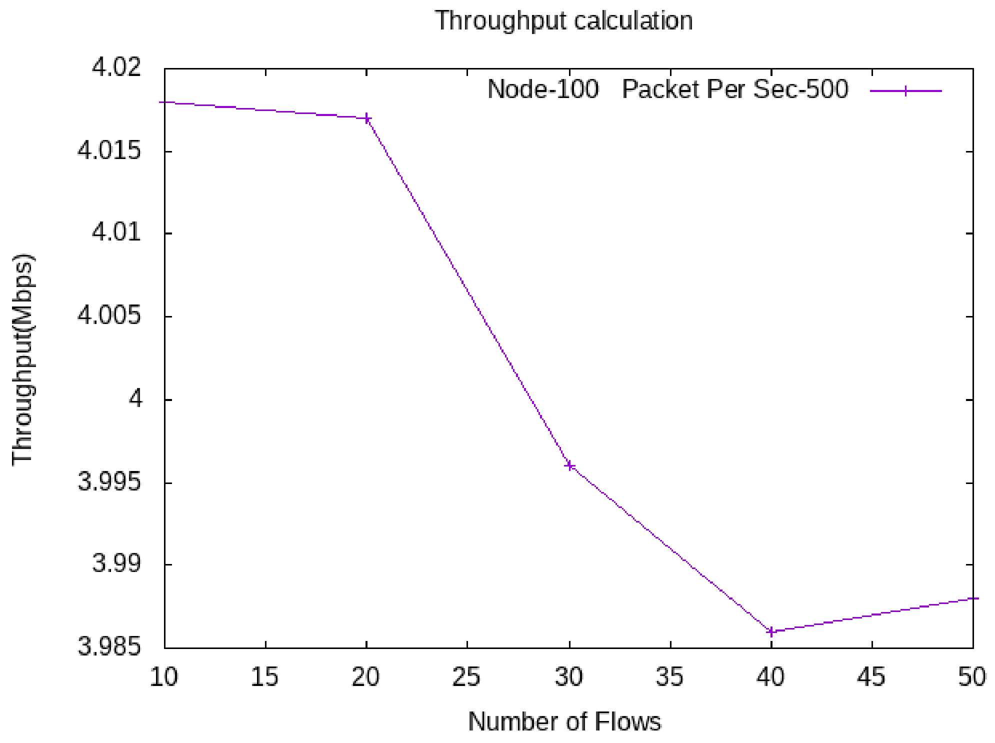


Figure 16: Throughput(varying flow)

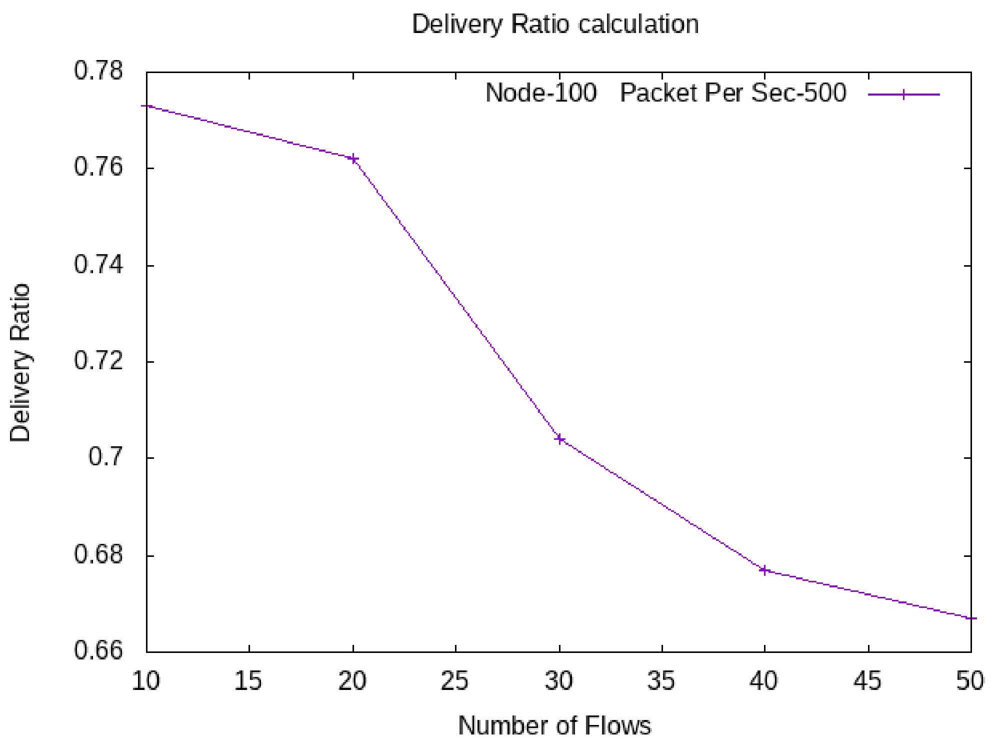


Figure 17: Delivery Ratio(varying flow)

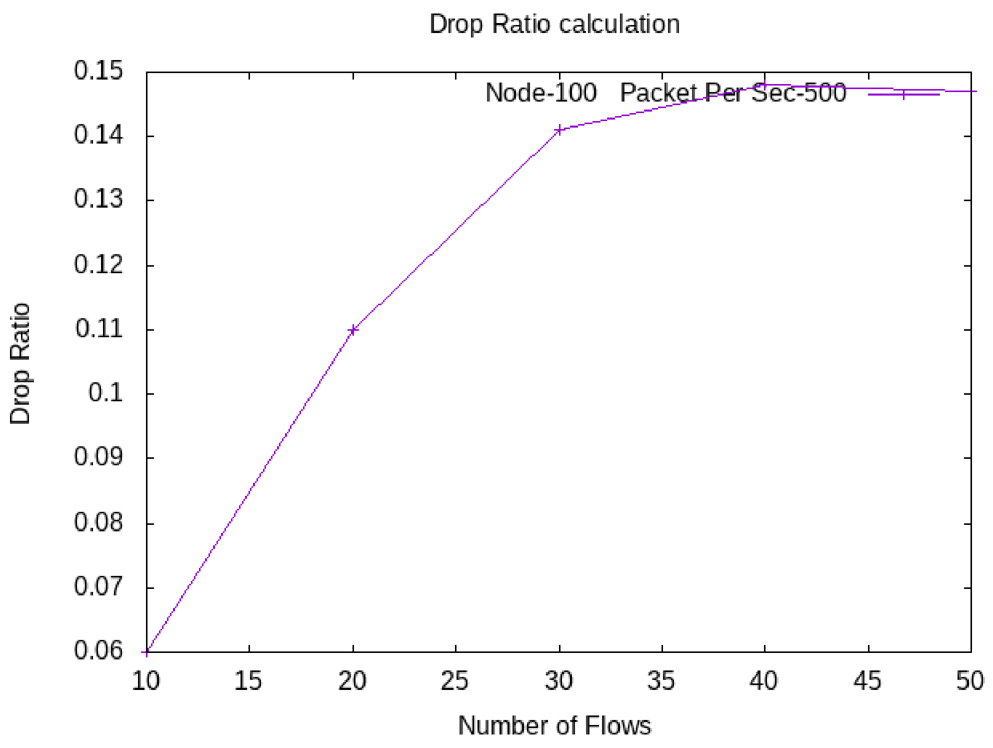


Figure 18: Drop Ratio(varying flow)

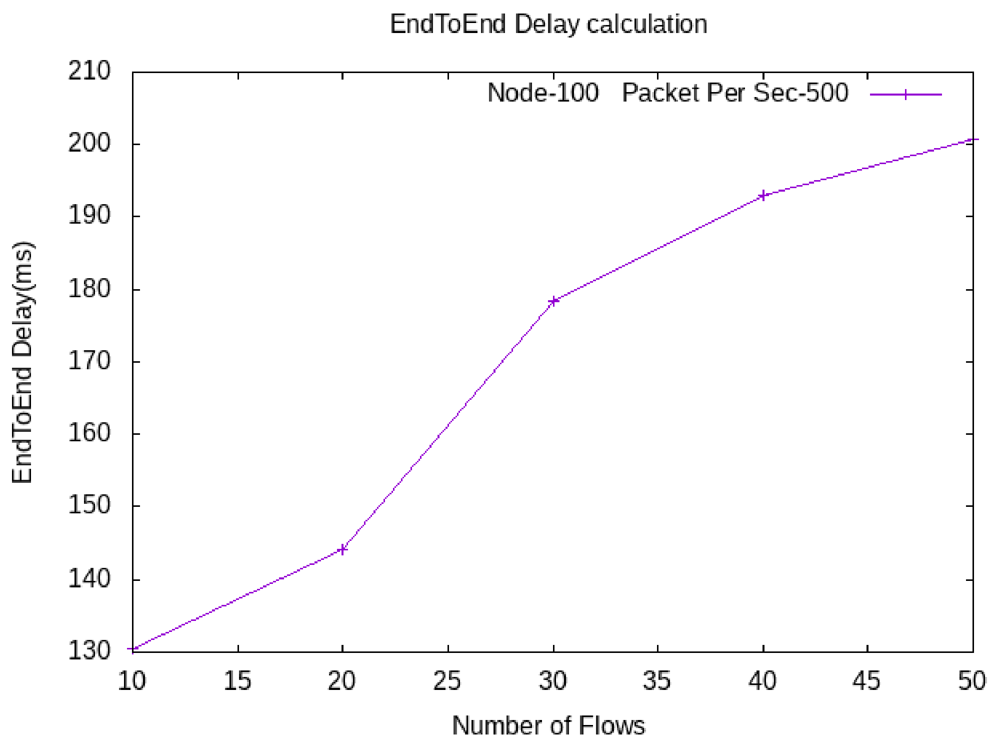


Figure 19: End to end delay(varying flow)

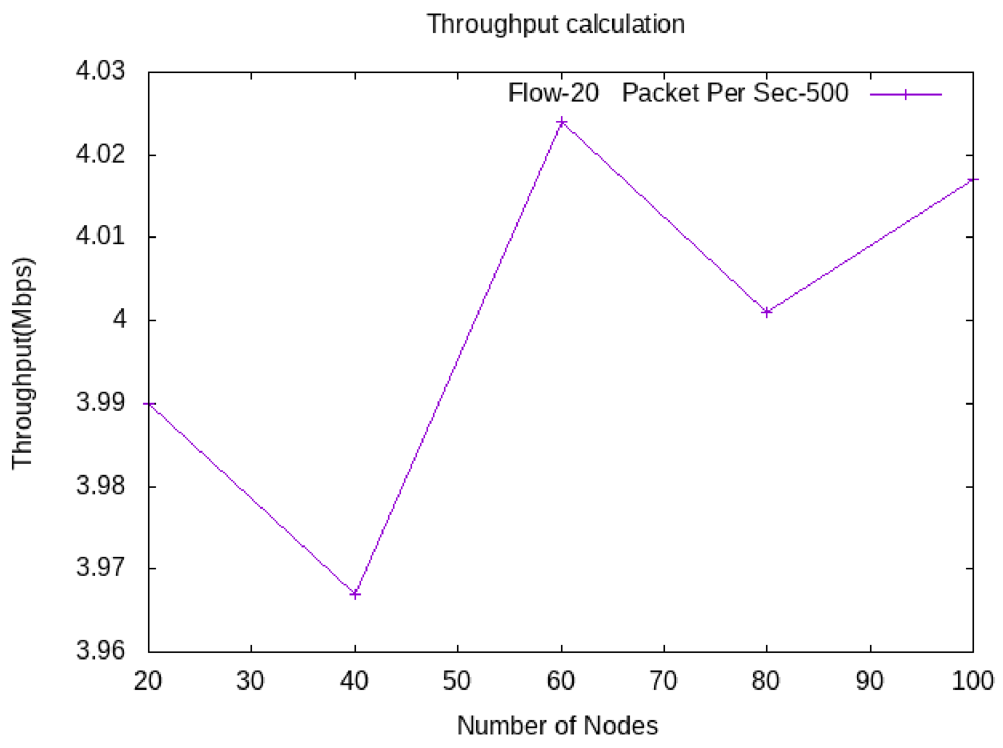


Figure 20: Throughput(Varying Node)

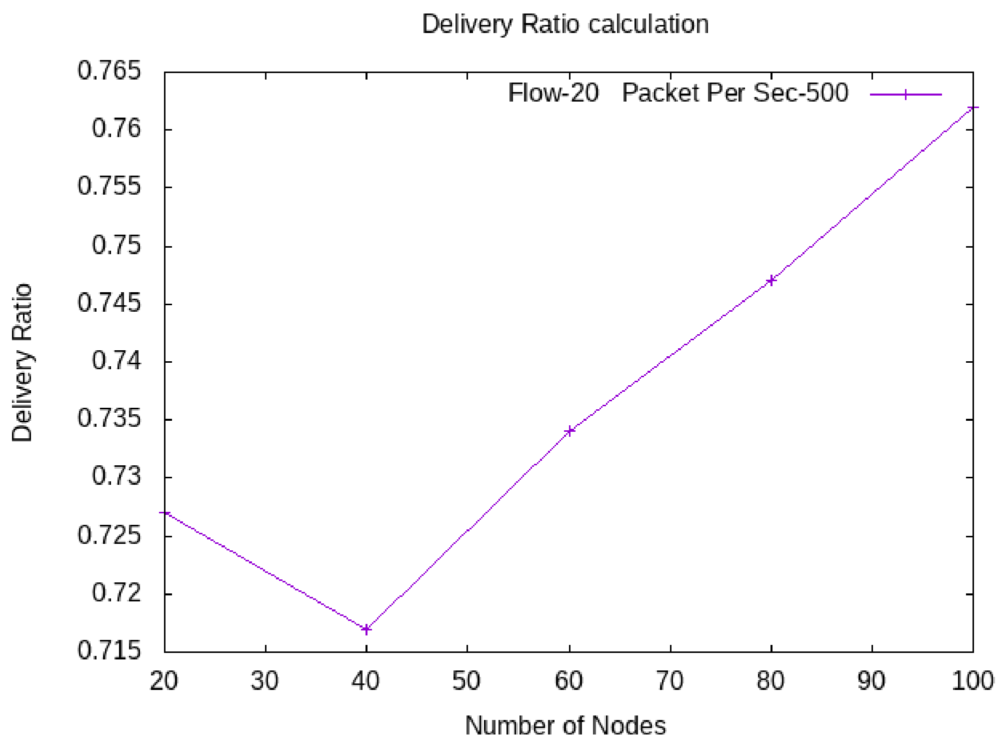


Figure 21: Delivery Ratio(Varying Node)

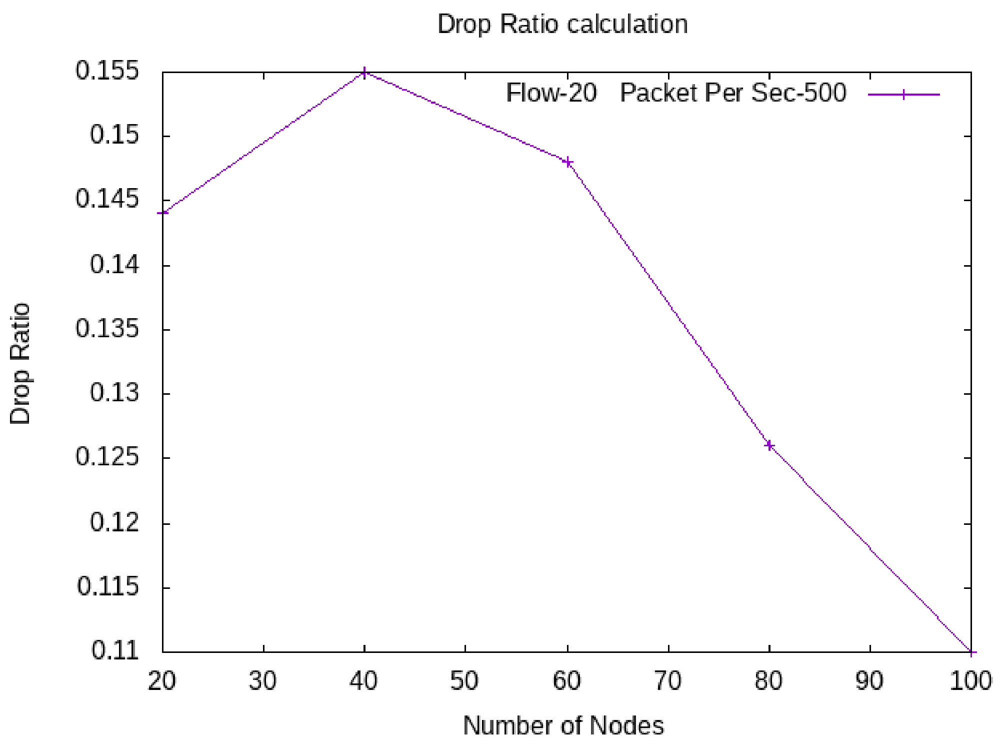


Figure 22: Drop Ratio(Varying Node)

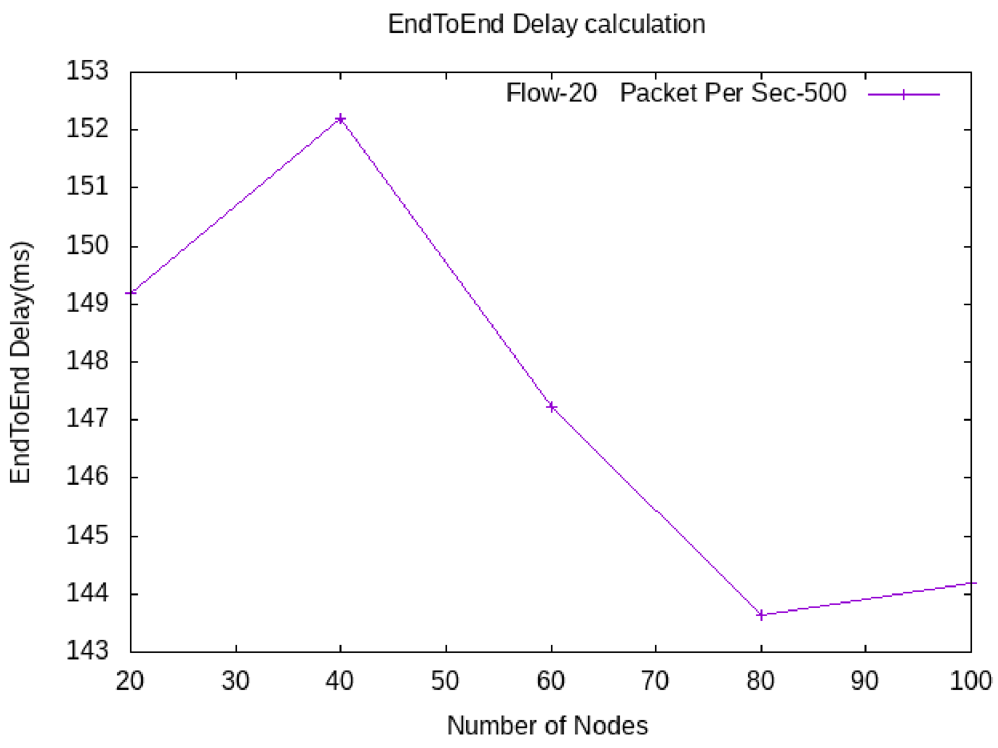


Figure 23: End to end delay(Varying Node)

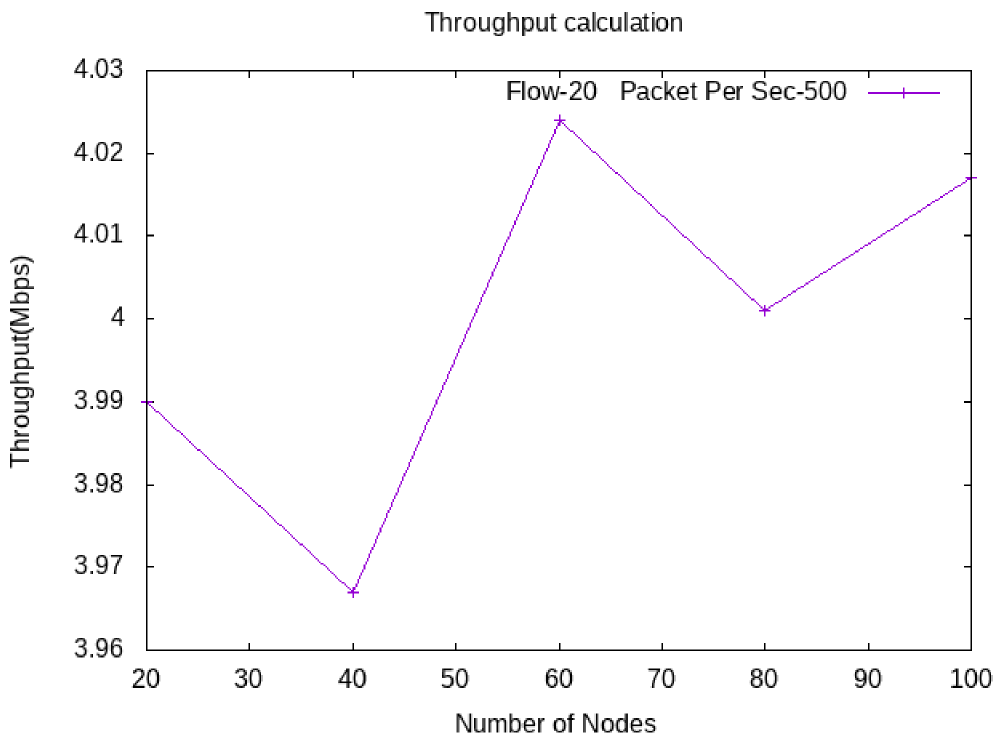


Figure 24: Throughput(Varying Packet per sec)

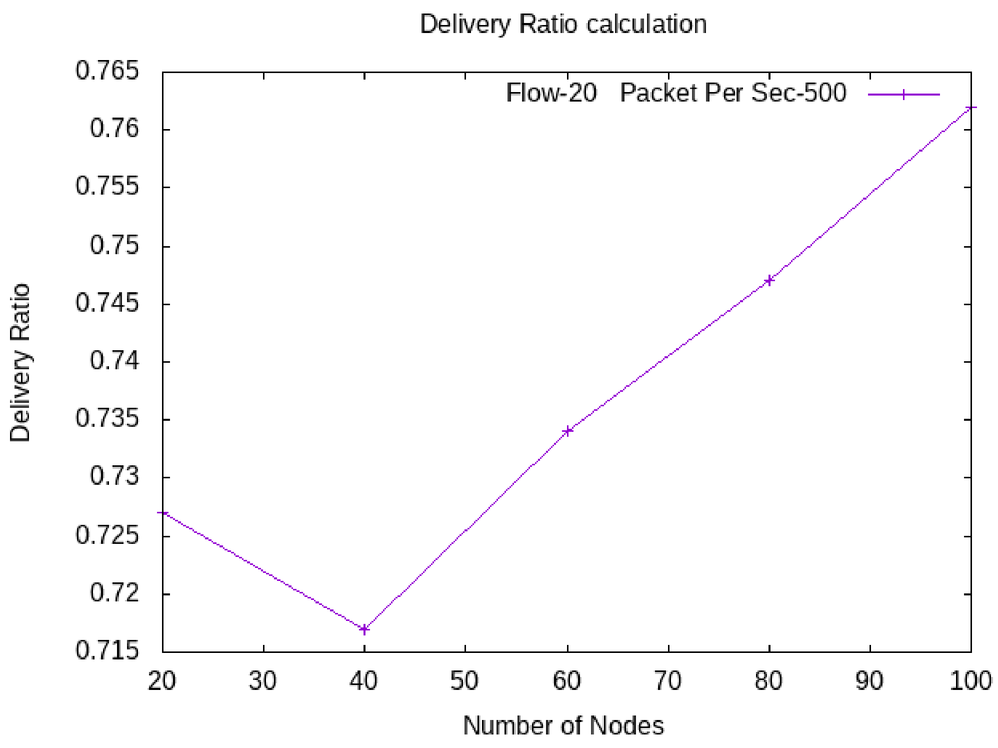


Figure 25: Delivery Ratio(Varying Packet per sec)

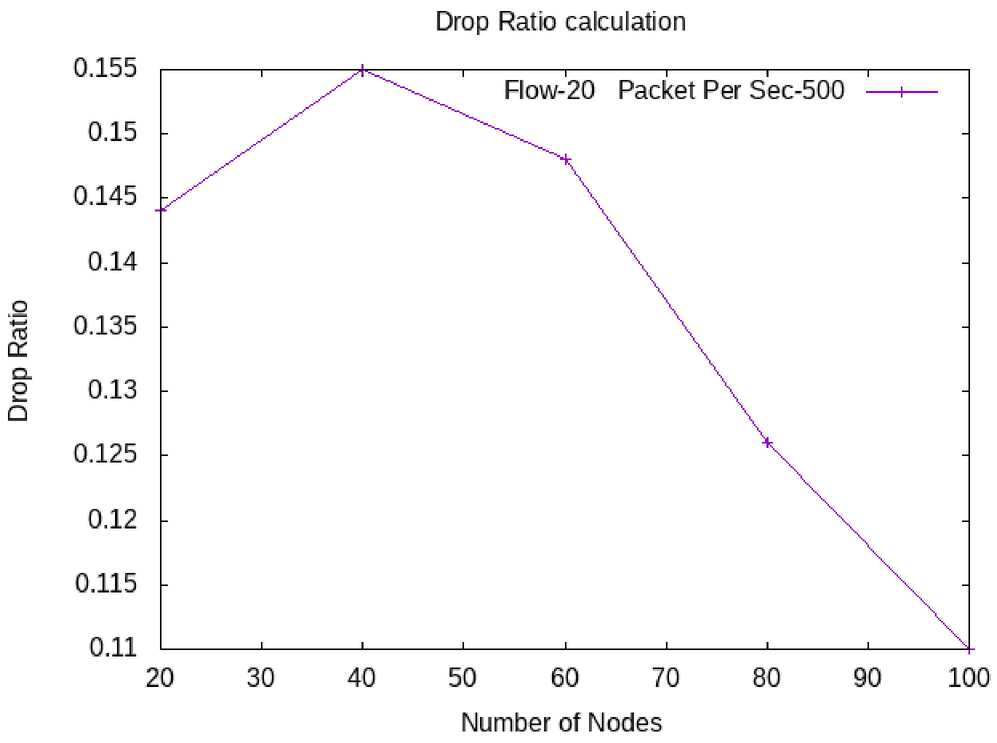


Figure 26: Drop Ratio(Varying Packet per sec)

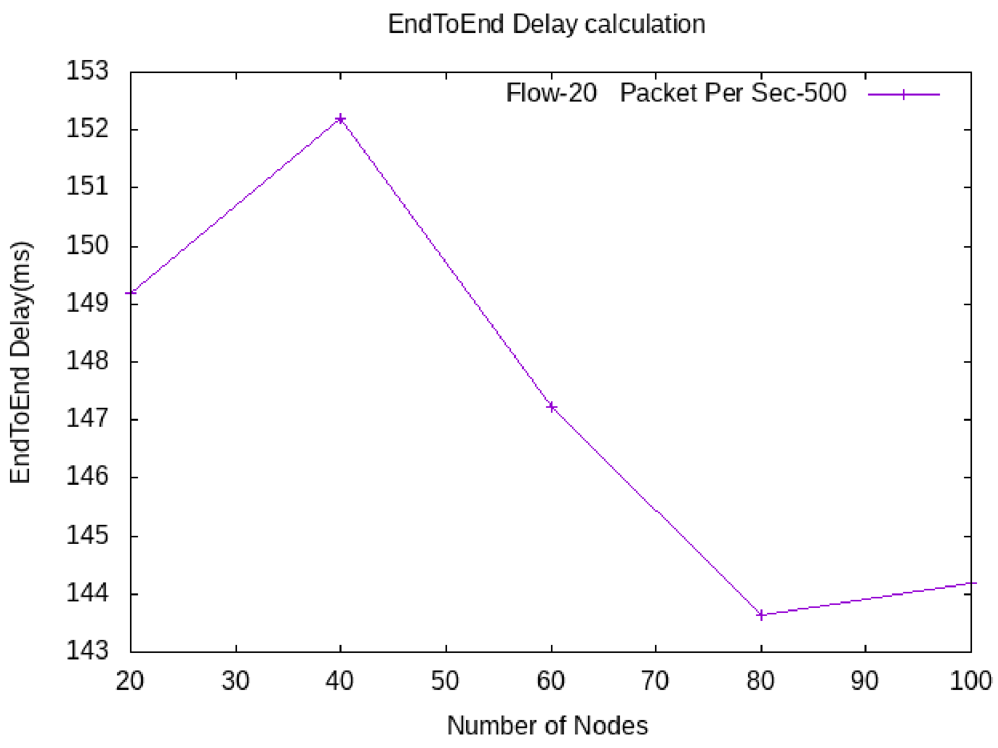


Figure 27: End to end delay(Varying Packet per sec)

6.2 Task A(Wireless)

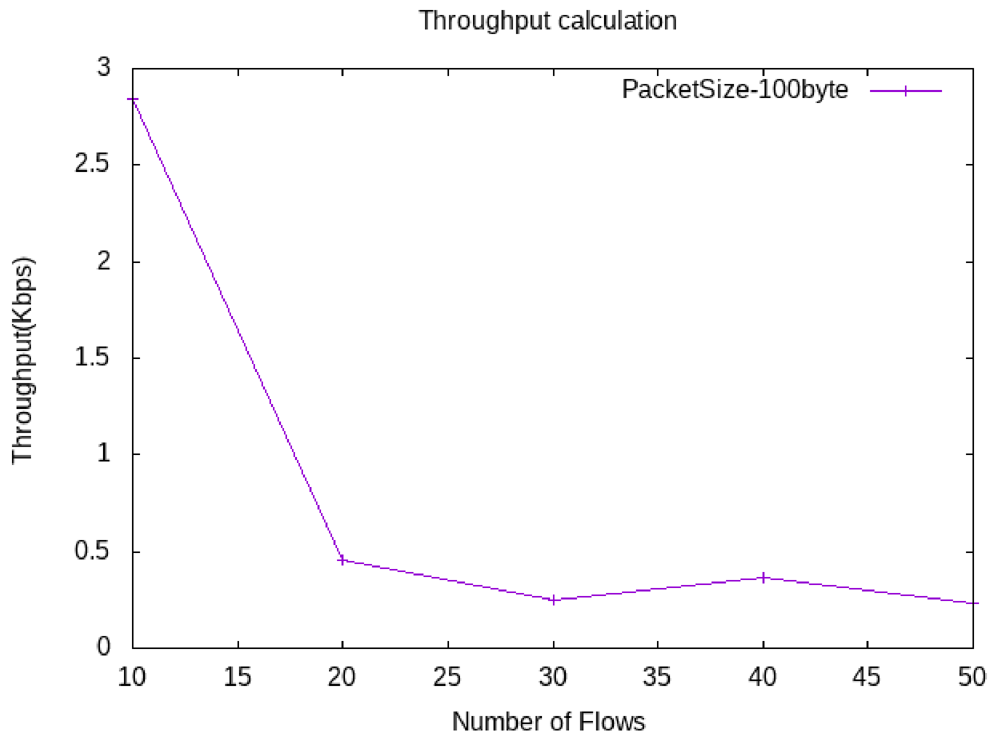


Figure 28: Throughput(varying flow)

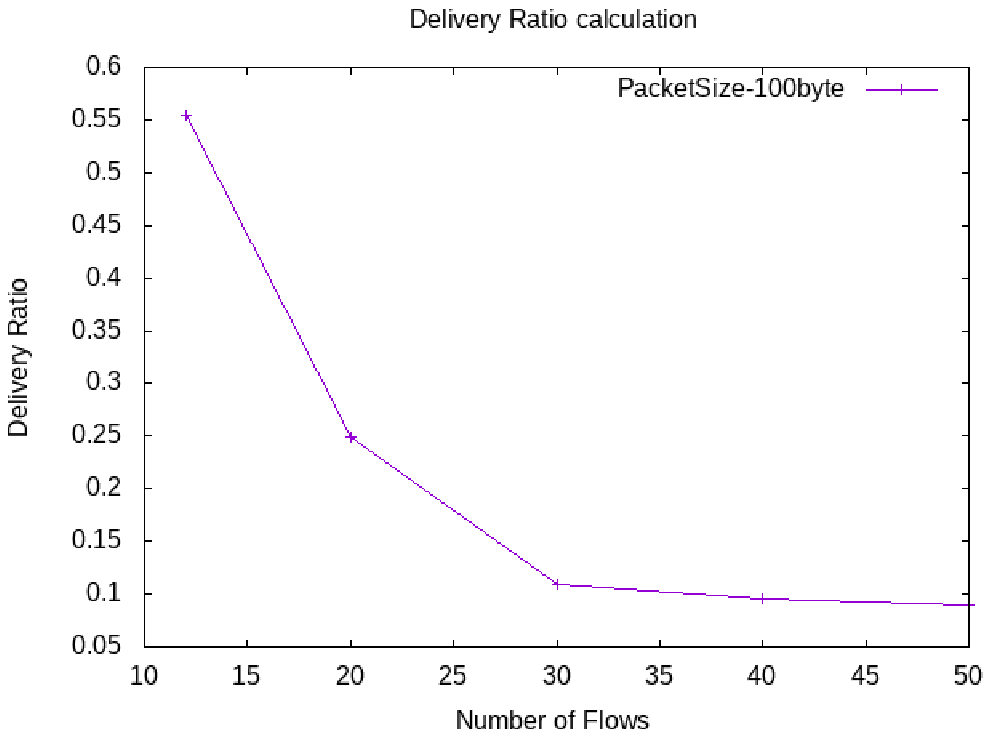


Figure 29: Delivery Ratio(varying flow)

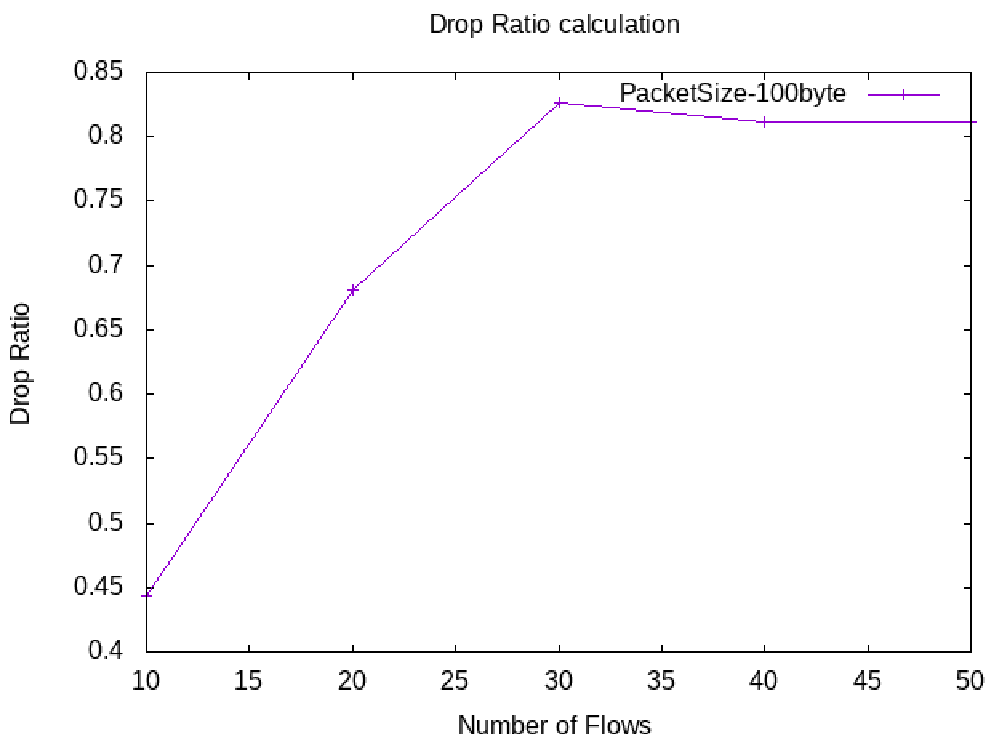


Figure 30: Drop Ratio(varying flow)

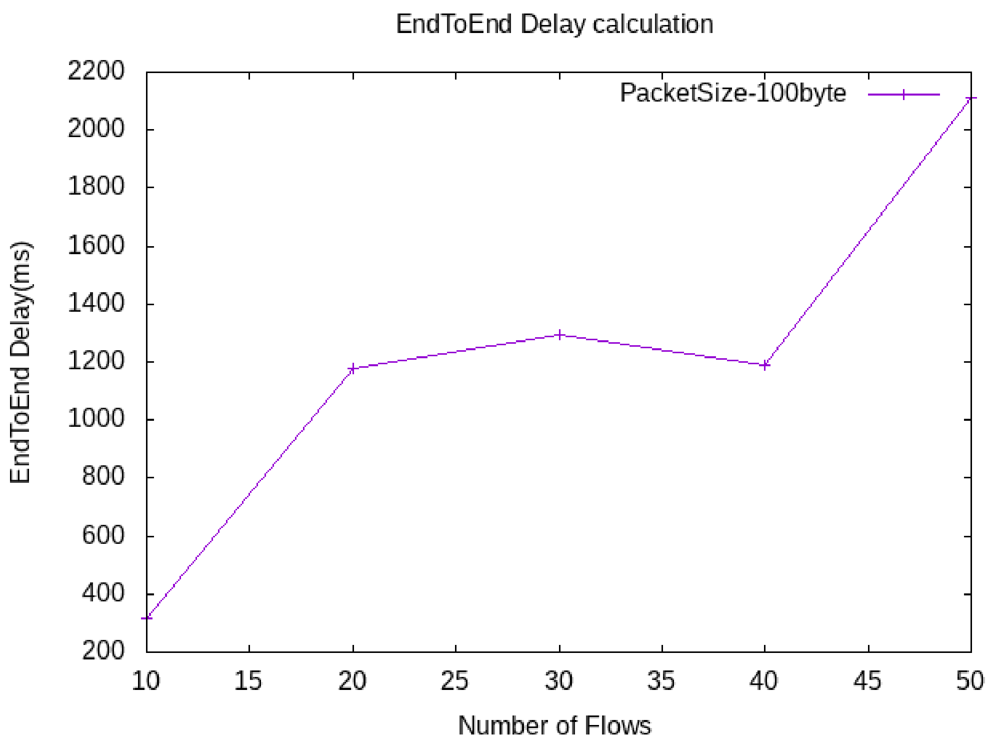


Figure 31: End to end delay(varying flow)

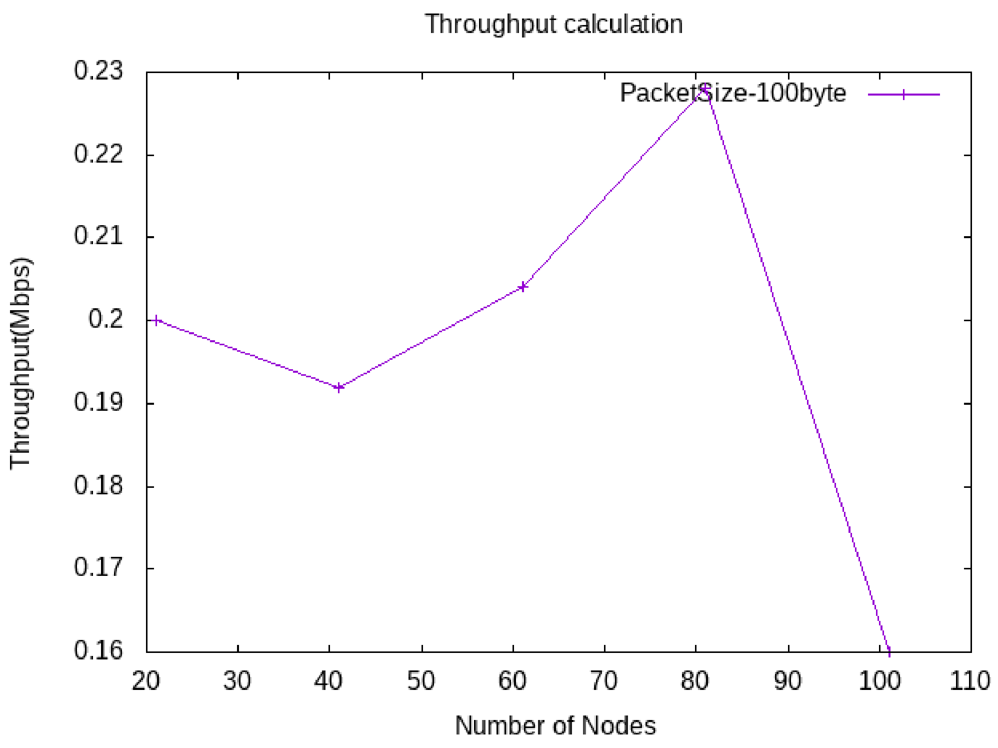


Figure 32: Throughput(Varying Node)

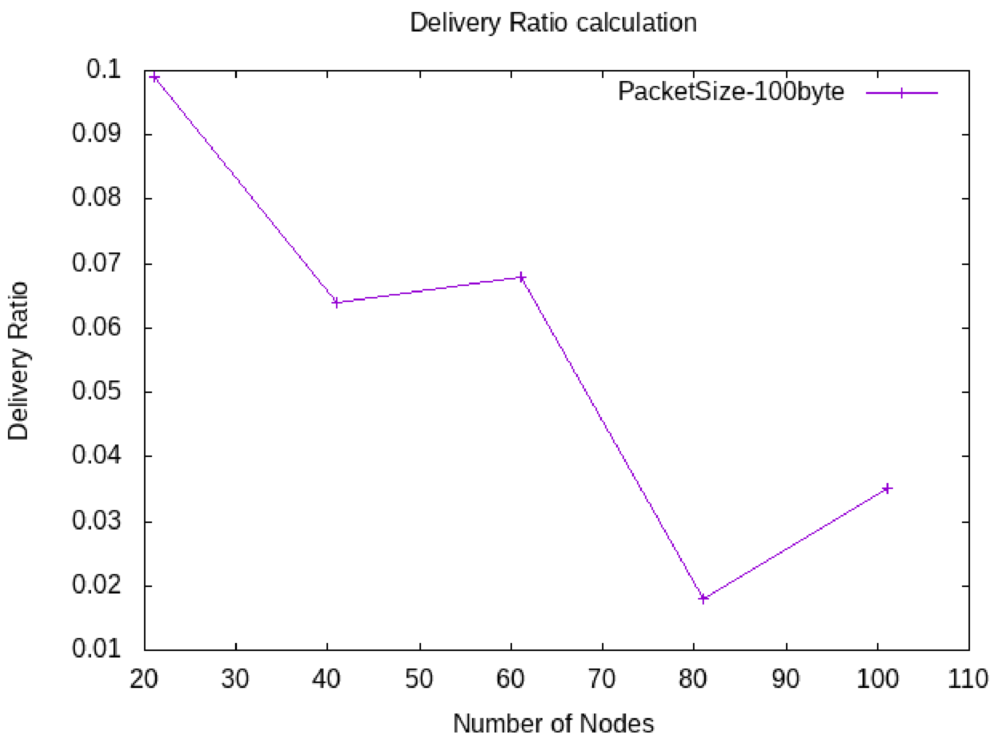


Figure 33: Delivery Ratio(Varying Node)

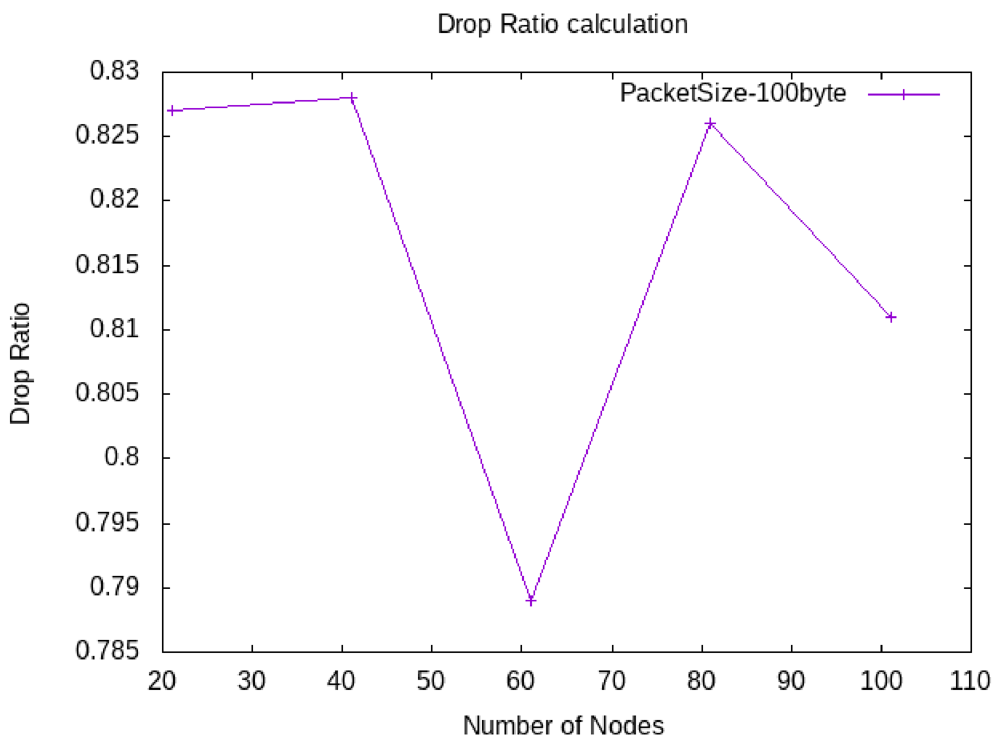


Figure 34: Drop Ratio(Varying Node)

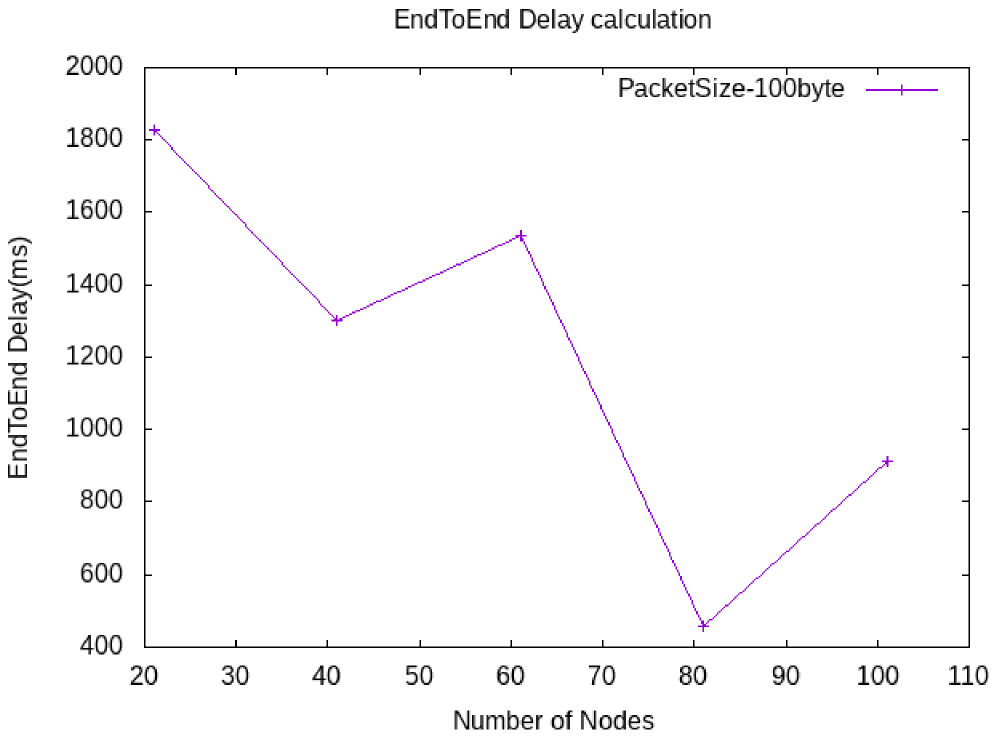


Figure 35: End to end delay(Varying Node)

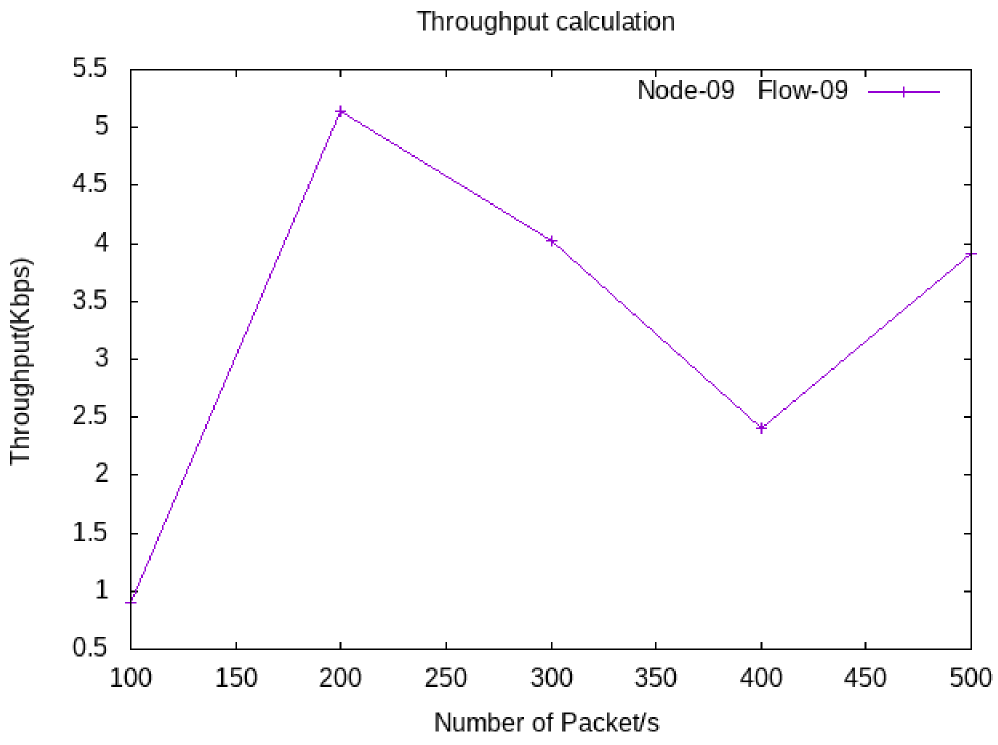


Figure 36: Throughput(Varying Packet per sec)

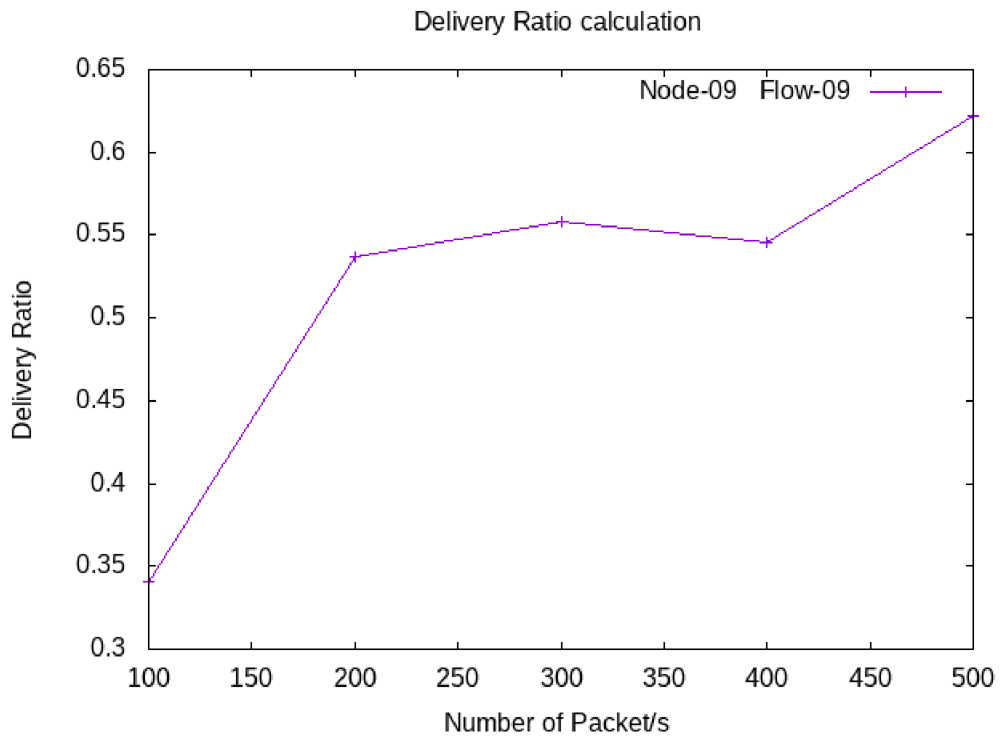


Figure 37: Delivery Ratio(Varying Packet per sec)

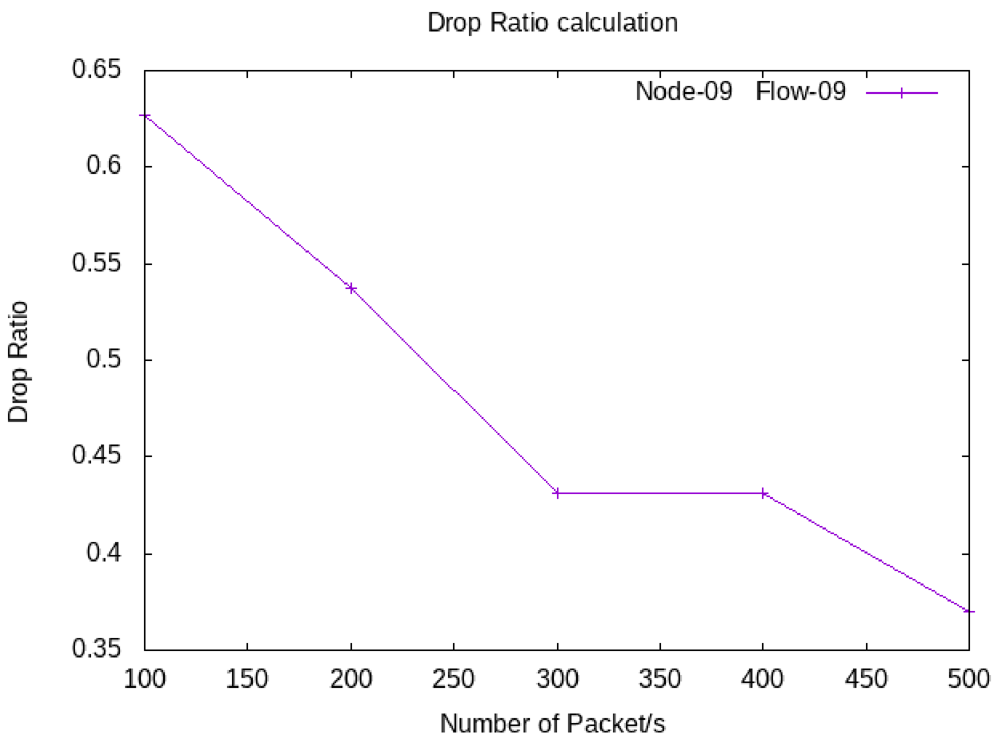


Figure 38: Drop Ratio(Varying Packet per sec)

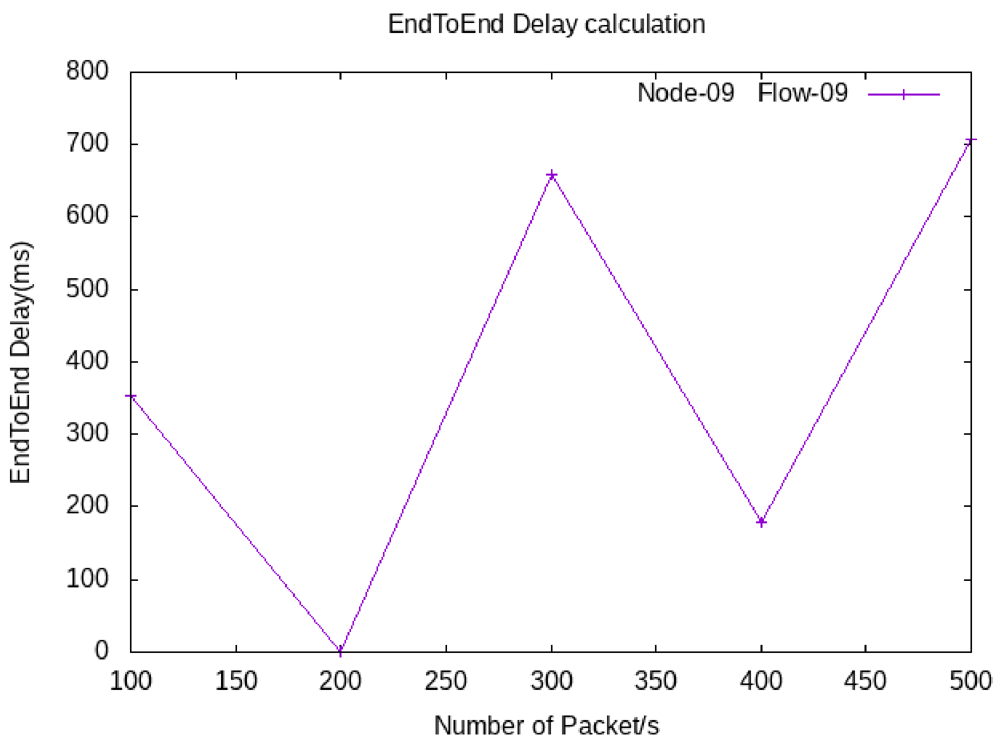


Figure 39: End to end delay(Varying Packet per sec)

6.3 Task B

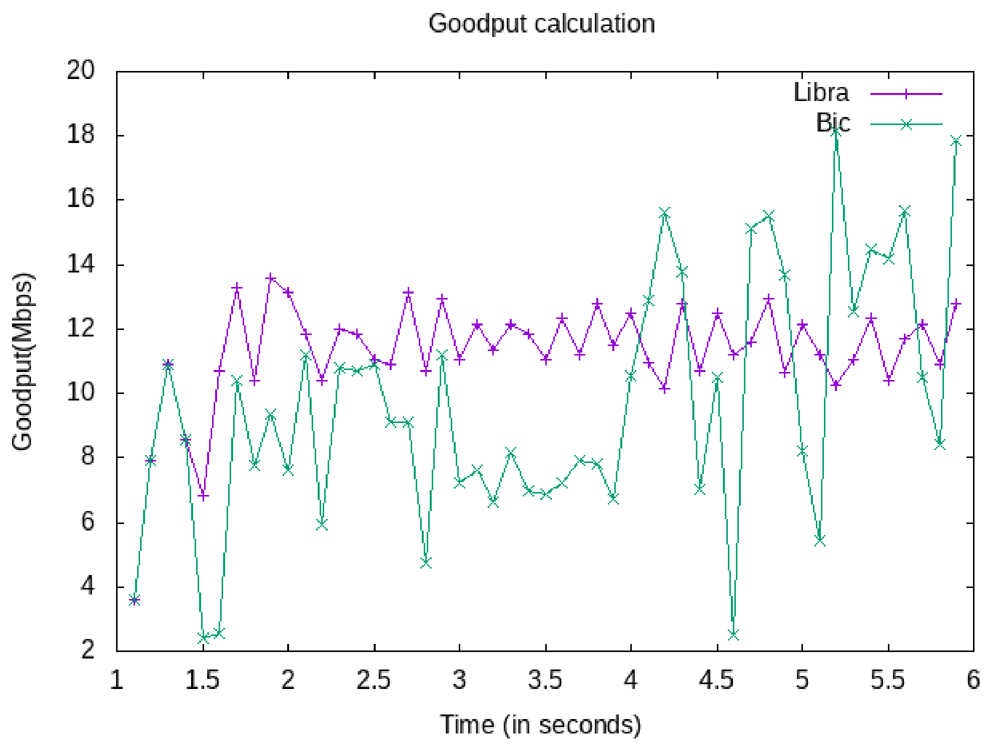


Figure 40: Goodput in every 0.1 sec

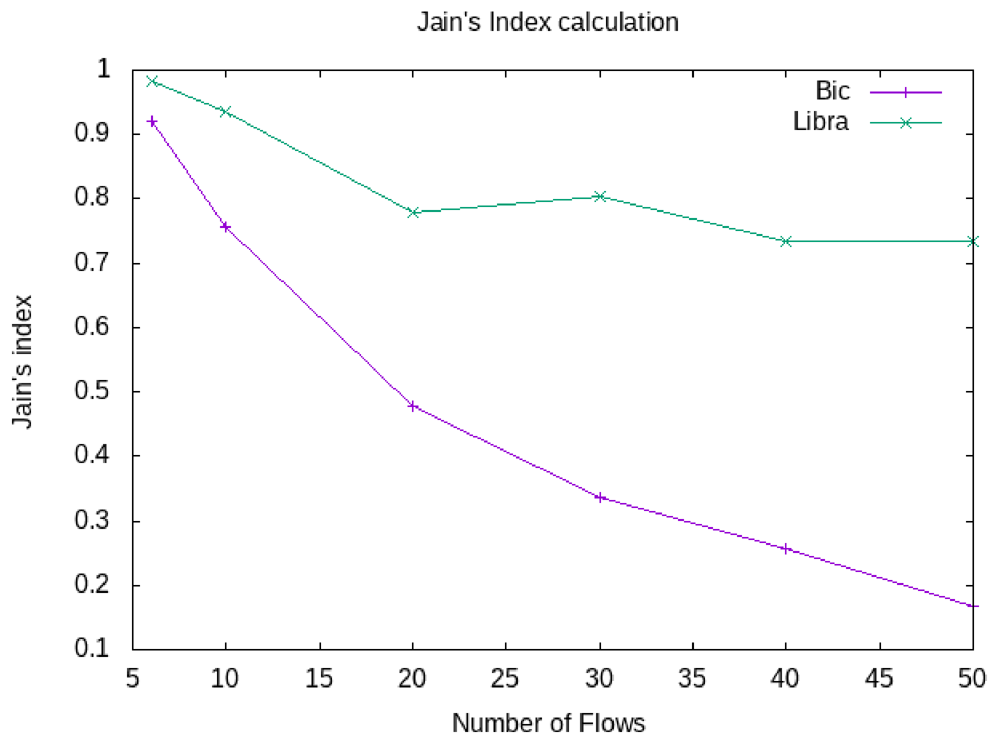


Figure 41: Fairness comparison between existing and new algorithm

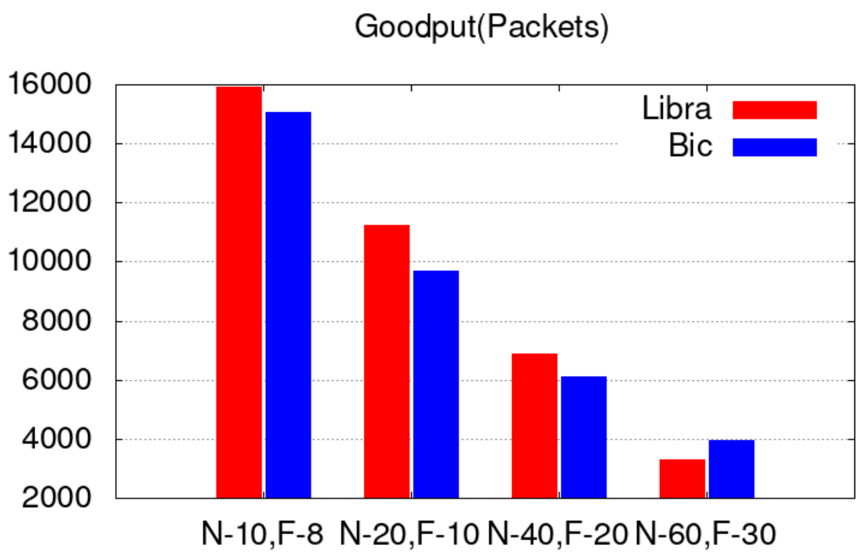


Figure 42: Average Goodput(packet)

6.4 Extra Metrics

We have generated an extra metric - Queue size. This is done in a hybrid topology. That means, cross transmission of packets, i.e., transmission of packets from one type of node to that of another type has been implemeneted in this topology. In this implementation, as our queuing policy we used "DropTailQueue". Data rate and delay are set to 1.5Mbps and 20ms respectively for our simulation purpose. RedQueueDisc is used as our queue disc. From the graph, we can notice that initially the number of packets in queue is 0 and it increases as time passes(Enqueue). It starts to decrease after some period(Dequeue) and reach 0 again.

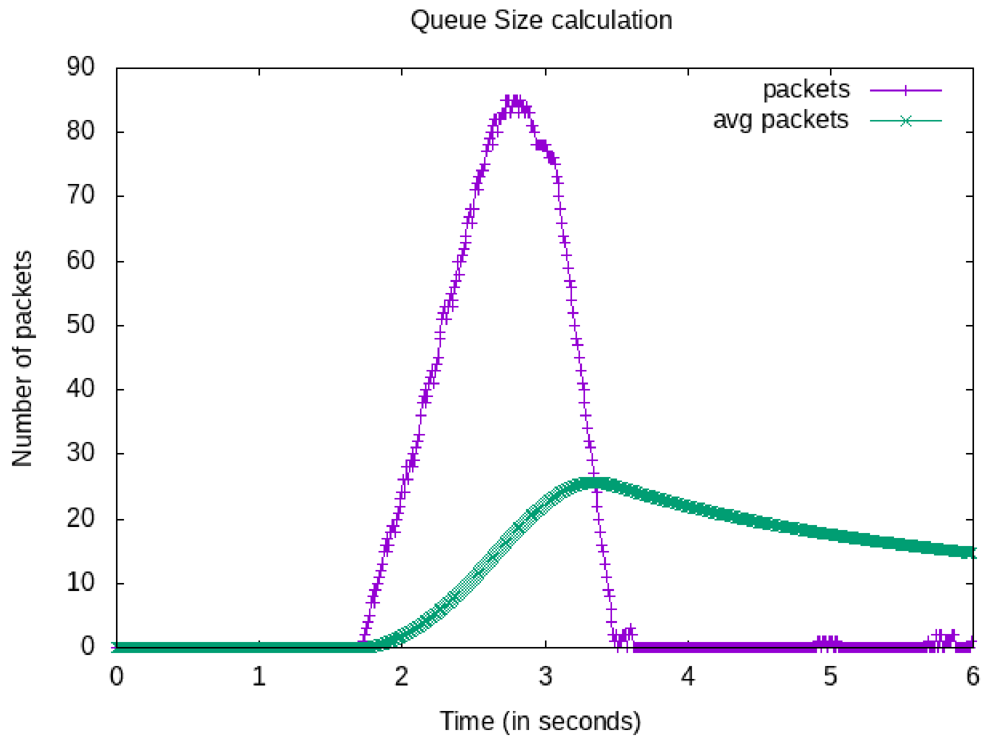


Figure 43: Number of packets and average packets in queue

7 Summary

7.1 Findings of Task A

In task A, we have to simulate two different kinds of networks - Wired and Wireless low rate(static). In case of **wired** network, from our simulation result, we can see that if we increase number of nodes and flows of a network or decrease the packet sent rate, metrics like - **throughput**, **end to end delay**, **drop ratio** increase. On the other hand, **delivery ratio** and **goodput** of that network decrease. This is because traffic increases as the number of nodes and flows increases. As a result, incident like packet drop or lost occurs.
This scenario reverses if we reverse the order of the parameters.
These metrics don't behave regularly in case of static wireless low rate network(802.15.4). Throughput, end to end delay, drop ratio - these metrics increase if we increase number of nodes and flows but there are exceptions. There may be several reasons behind this-

- We used tcp NewReno as our congestion control algorithm. This algorithm may not be the best suitable algorithm to handle low rate data transfer.
- 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 have not come. This may happen 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 packet drop in case of low rate data transfer. They may be dropped due to excessive transmission retries or channel access failure.

While varying the coverage area of each node, we couldn't notice in change in any of the metrics. We varied the range in **RangePropagationLossModel**, **DeltaX**, **DeltaY** and **GridWidth** but in vain. The problem should be associated with **LrWpanHelper** because the same thing(RangePropagationLossModel) works perfectly in YansWifiChannelHelper.

7.2 Findings of Task B

In task B, we implement a new algorithm called **TCP Libra** to ensure RTT fairness. We used TCP Libra for the odd flows and TCP BIC for the even ones. This is done so that they can use the same bottleneck link at the same time.
The result of the simulation is almost same as the paper we followed. TCP Libra outperforms TCP BIC by a mile when the network has lots of flows. In simulation, we can see that the **Jain's index** for TCP Libra seems to drop to 0.77 for 100 flows. This may be happen due to short simulation time compared to the paper we followed.
If we look at the goodput for both the algorithm, we can say that TCP Libra is not very good at maintaining high goodput for a larger number of flows. Our paper also agrees to this fact. This may be the result of high reliance on queuing delay estimation.