

Report On Implementation of TCPW BR in NS3

TCPW BR: A Wireless Congestion Control Scheme
Base on RTT

Submitted By:

Saem Hasan

1705027

Overview of Proposed Algorithm

TCP Westwood adopts the idea of bandwidth estimation. TCP Westwood is more prominent in wireless congestion control. TCPW is an end-to-end congestion control mechanism protocol. TCPW cannot distinguish between congestion loss and wireless loss. Aiming at the shortcomings of TCPW that cannot distinguish between congestion loss and wireless self-loss, the TCPW BR algorithm is proposed.

TCPW BR scheme

Based on RTT, TCPW BR scheme is proposed. The overview of this algorithm is described below.

Congestion level division

The first step is to estimate an accurate RTT. Then We need to use RTT min and RTT max where RTT max and RTT min respectively represent the maximum and minimum values of the measured RTT during the transmission of the TCP data segment. Substituting the measured RTT values into the following weighted average mathematical expressions (1) and (2) indirectly reflects network congestion. The specific practices areas follows:

$$F = \text{RTT max} - \text{RTT min} \quad (1)$$

$$R = (\text{RTT} - \text{RTT min})/F \quad (2)$$

$R \in [0,1]$ indicates the extent to which the currently confirmed data segment is used in the network transmission process. The smaller the R , the less time the data segment spends, and the network is idle; otherwise, the network is more congested. Divide R into 4 levels L , as shown in Tab. 1, where a higher level indicates a greater likelihood of congestion.

R	[0,0.25]	(0.25, 0.5]	(0.5, 0.75]	(0.75, 1.00]
L	1	2	3	4

Table 1 : Congestion level classification

TCPW BR Algorithm

When the congestion level is equal to 1, it proves that the network condition is better and the congestion probability is small. If packet loss occurs at this time, it is considered that it is a large wireless packet loss, so there is no need to reduce the values of $Cwnd$ and $Ssthresh$ excessively. When the congestion level is 2, it is proved that there is slight congestion, and the transmission rate can be appropriately changed to reduce the value of the growth factor P . When the congestion level is greater than 3, the congestion is proved to be serious. The packet loss is considered to be congestion and packet loss.

L	1	2	3	4
P	1	0.867	0.5	0.4

Table 2 : Growth factors corresponding to congestion levels

The algorithm is described as follows:

```
(1) Each time an ACK of a new data segment is received,  
    If (congestion level=1||congestion level=2)//Think it is wireless packet loss, mild  
    congestion  
        Cwnd=Cwnd+1;  
    If (Cwnd>Ssthresh)  
        Cwnd=Cwnd+(1/Cwnd)*p;  
(2) After receiving a duplicate ACK before timing out  
    If (duplicate ACK=3&& congestion level=1)  
        Fast retransmission;  
        Quick recovery  
    If (duplicate ACK=2&& (congestion level=3||congestion level=4))//Think it is a  
    congestion packet  
        Slow start or congestion avoidance;  
        Cwnd=Cwnd*p;  
        Ssthresh =(BWE*RTTmin)/seg_size;  
        If (Cwnd>Ssthresh) then Cwnd=Ssthresh;  
    If (duplicate ACK=3&& congestion level>2)  
        Slow start or congestion avoidance;  
        Cwnd=Cwnd*p;  
        Ssthresh=(BWE*RTTmin)/seg_size;  
        If (Cwnd>Ssthresh) then Cwnd=Ssthresh;
```

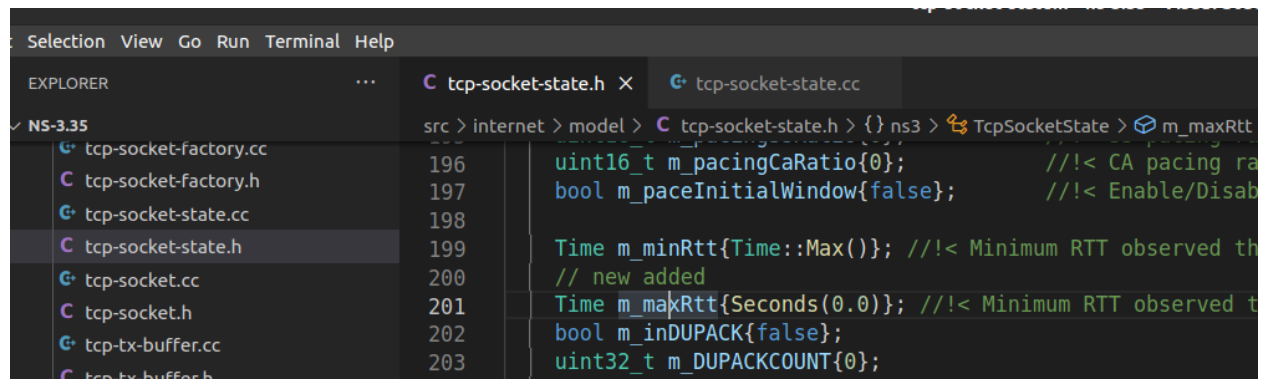
Figure 1: TCPW Algorithm

TCPW BR algorithm mainly works when any duplicate acknowledgement is received. When any duplicate acknowledgement is received, this algorithm checks the dupack count and the congestion level. Based on dupack count and congestion level, this algorithm multiplies the congestion window with the growth factor, P.

This algorithm also does some modification in the congestion avoidance phase.

Modifications in Simulator to implement Algorithm

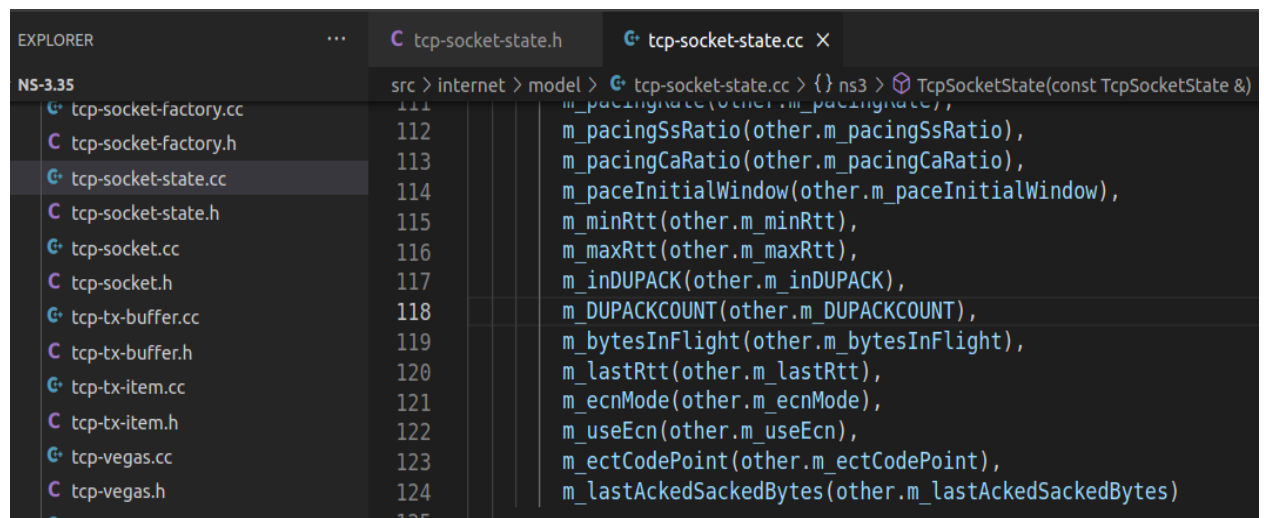
1. To access the maximum RTT and duplicate acknowledge count in our algorithm to determine the congestion level, we add three variables in the tcp-socket-state.h file.



```
Selection View Go Run Terminal Help
EXPLORER
NS-3.35
tcp-socket-factory.cc
tcp-socket-factory.h
tcp-socket-state.cc
tcp-socket-state.h
tcp-socket.cc
tcp-socket.h
tcp-tx-buffer.cc
tcp-tx-buffer.h
src > internet > model > tcp-socket-state.h > {} ns3 > TcpSocketState > m_maxRtt
196 uint16_t m_pacingCaRatio{0}; //!< CA pacing ra
197 bool m_paceInitialWindow{false}; //!< Enable/Disab
198
199 Time m_minRtt{Time::Max()}; //!< Minimum RTT observed th
200 // new added
201 Time m_maxRtt{Seconds(0.0)}; //!< Minimum RTT observed t
202 bool m_inDUPACK{false};
203 uint32_t m_DUPACKCOUNT{0};
```

Figure 2 : New variable in tcp-socket-state.h

2. Then we have to update the copy constructor of **TcpSocketState**.



```
EXPLORER
NS-3.35
tcp-socket-factory.cc
tcp-socket-factory.h
tcp-socket-state.cc
tcp-socket-state.h
tcp-socket.cc
tcp-socket.h
tcp-tx-buffer.cc
tcp-tx-buffer.h
tcp-tx-item.cc
tcp-tx-item.h
tcp-vegas.cc
tcp-vegas.h
src > internet > model > tcp-socket-state.cc > {} ns3 > TcpSocketState(const TcpSocketState &)
111 m_pacingRate(other.m_pacingRate),
112 m_pacingSsRatio(other.m_pacingSsRatio),
113 m_pacingCaRatio(other.m_pacingCaRatio),
114 m_paceInitialWindow(other.m_paceInitialWindow),
115 m_minRtt(other.m_minRtt),
116 m_maxRtt(other.m_maxRtt),
117 m_inDUPACK(other.m_inDUPACK),
118 m_DUPACKCOUNT(other.m_DUPACKCOUNT),
119 m_bytesInFlight(other.m_bytesInFlight),
120 m_lastRtt(other.m_lastRtt),
121 m_ecnMode(other.m_ecnMode),
122 m_useEcn(other.m_useEcn),
123 m_ectCodePoint(other.m_ectCodePoint),
124 m_lastAckedSackedBytes(other.m_lastAckedSackedBytes)
```

Figure 3 : Copy constructor update in tcp-socket-state.cc

3. To get the maximum RTT throughout the transmission, we update the maxRtt variable in **TcpSocketBase::EstimateRtt** method.

```

tcp-socket-base.cc X
src > internet > model > tcp-socket-base.cc > {} ns3 > EstimateRtt(const TcpHeader &)
3483     }
3484     m_history.pop_front(); // Remove
3485 }
3486
3487 if (!m.IsZero())
3488 {
3489     m_rtt->Measurement(m); // Log the measurement
3490     // RFC 6298, clause 2.4
3491     m_rto = Max(m_rtt->GetEstimate() + Max(m_clockGranularity, m_rtt->GetVariation() * 4), m_minRto);
3492     m_tcb->m_lastRtt = m_rtt->GetEstimate();
3493     m_tcb->m_minRtt = std::min(m_tcb->m_lastRtt.Get(), m_tcb->m_minRtt);
3494     m_tcb->m_maxRtt = std::max(m_tcb->m_lastRtt.Get(), m_tcb->m_maxRtt); // new added
3495     NS_LOG_INFO(this << m_tcb->m_lastRtt << m_tcb->m_minRtt);
3496 }
3497 }

```

Figure 4: Max. Rtt update in tcp-socket-base.cc

4. When any duplicate acknowledgement is received, TCPW BR needs to update the congestion window. That's why we need a method of TCPW BR which we can access from tcp-socket-base. So we add a new method in the tcp-congestion-ops.h file.

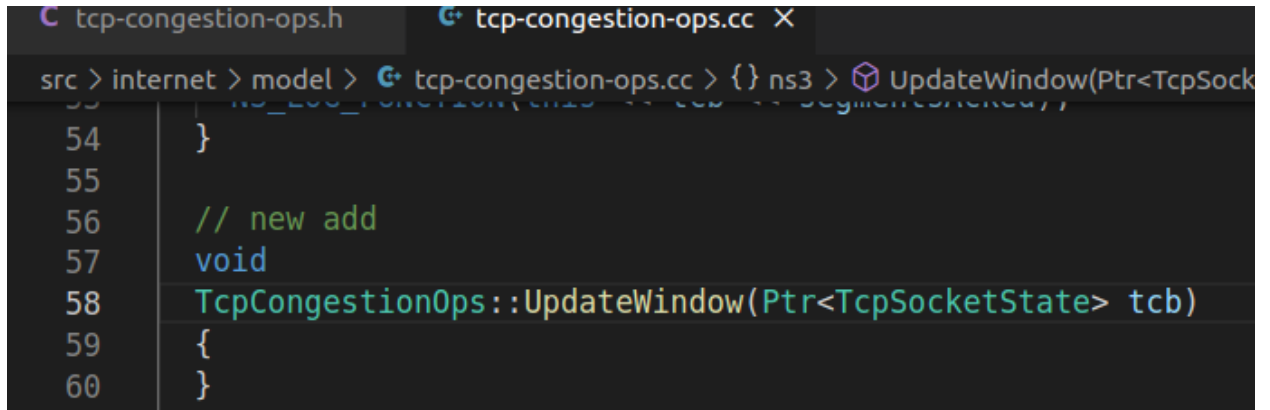
```

tcp-congestion-ops.h X tcp-congestion-ops.cc
src > internet > model > tcp-congestion-ops.h > {} ns3 > TcpCongestionOps
115     */
116     virtual void IncreaseWindow(Ptr<TcpSocketState> tcb, uint32_t segmentsAacked);
117
118     // new add
119     virtual void UpdateWindow(Ptr<TcpSocketState> tcb);
120

```

Figure 5: UpdateWindow method add in tcp-congestion-ops.h

- 4.1. Then we define the **UpdateWindow** method in the tcp-congestion-ops.cc file.



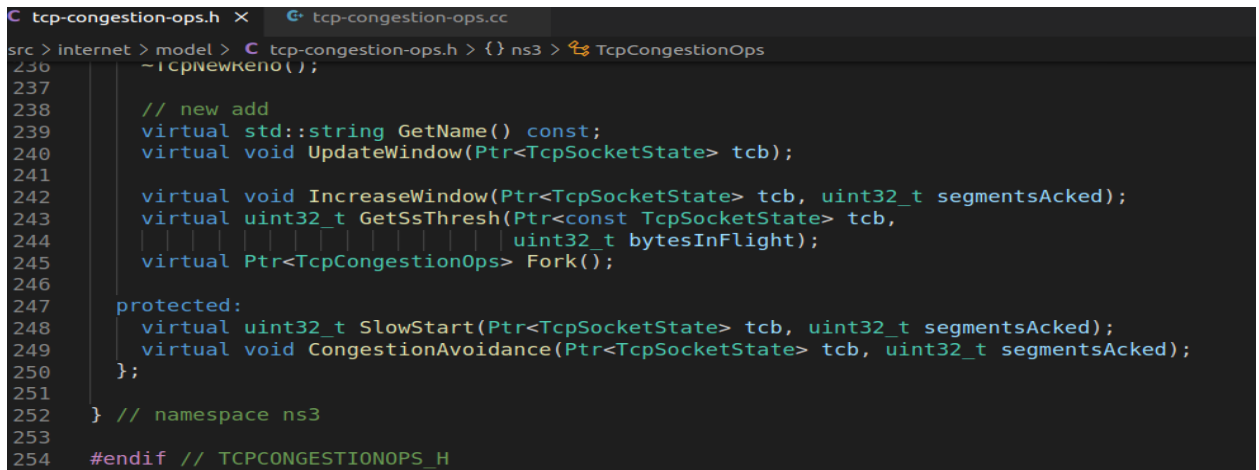
```

src > internet > model > tcp-congestion-ops.cc > {} ns3 > UpdateWindow(Ptr<TcpSock
53 ns_200_1.UpdateWindow(tcb, tcb->GetSsThresh(), tcb->GetBytesInFlight());
54 }
55
56 // new add
57 void
58 TcpCongestionOps::UpdateWindow(Ptr<TcpSocketState> tcb)
59 {
60 }

```

Figure 6 : UpdateWindow definition

4.2. We do the same in the **TcpNewReno** class.



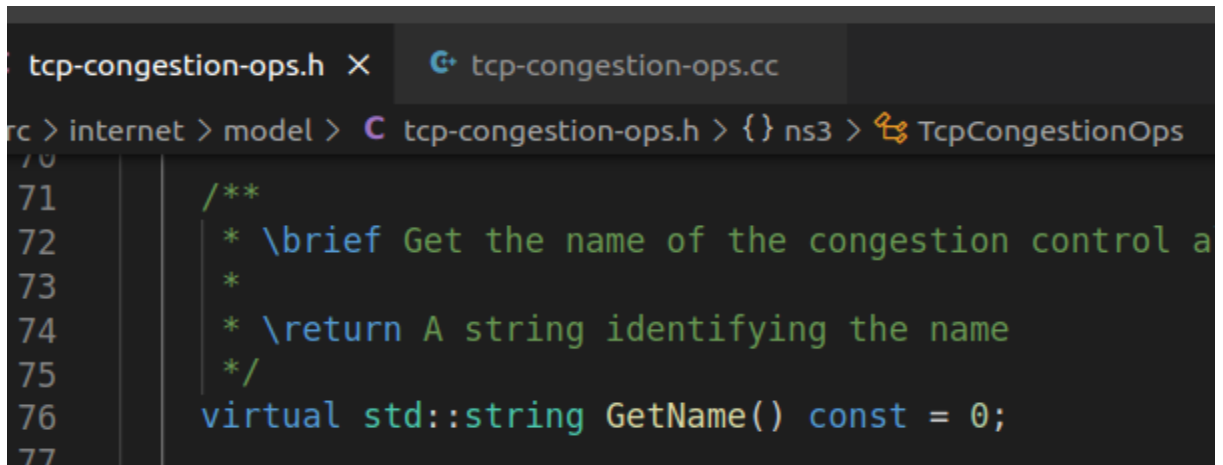
```

src > internet > model > tcp-congestion-ops.h > {} ns3 > TcpCongestionOps
230 ~TcpNewReno();
231
232 // new add
233 virtual std::string GetName() const;
234 virtual void UpdateWindow(Ptr<TcpSocketState> tcb);
235
236 virtual void IncreaseWindow(Ptr<TcpSocketState> tcb, uint32_t segmentsAacked);
237 virtual uint32_t GetSsThresh(Ptr<const TcpSocketState> tcb,
238                               uint32_t bytesInFlight);
239 virtual Ptr<TcpCongestionOps> Fork();
240
241 protected:
242 virtual uint32_t SlowStart(Ptr<TcpSocketState> tcb, uint32_t segmentsAacked);
243 virtual void CongestionAvoidance(Ptr<TcpSocketState> tcb, uint32_t segmentsAacked);
244 };
245
246 } // namespace ns3
247
248 #endif // TCPCONGESTIONOPS_H

```

Figure 7 : UpdateWindow method add in TcpNewReno

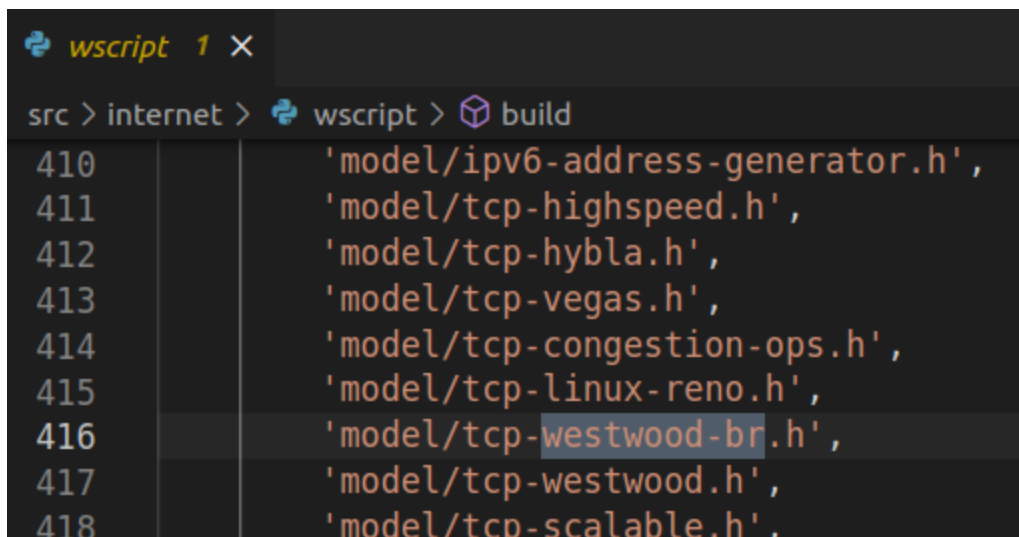
5. We call the **UpdateWindow** method when the congestion control algorithm is **TcpWestwoodBR**. That's why we make the **GetName** method of **TcpCongestionOp** virtual.



```
tcp-congestion-ops.h X tcp-congestion-ops.cc
src > internet > model > C tcp-congestion-ops.h > {} ns3 > TcpCongestionOps
70
71 /**
72  * \brief Get the name of the congestion control a
73  *
74  * \return A string identifying the name
75  */
76 virtual std::string GetName() const = 0;
77
```

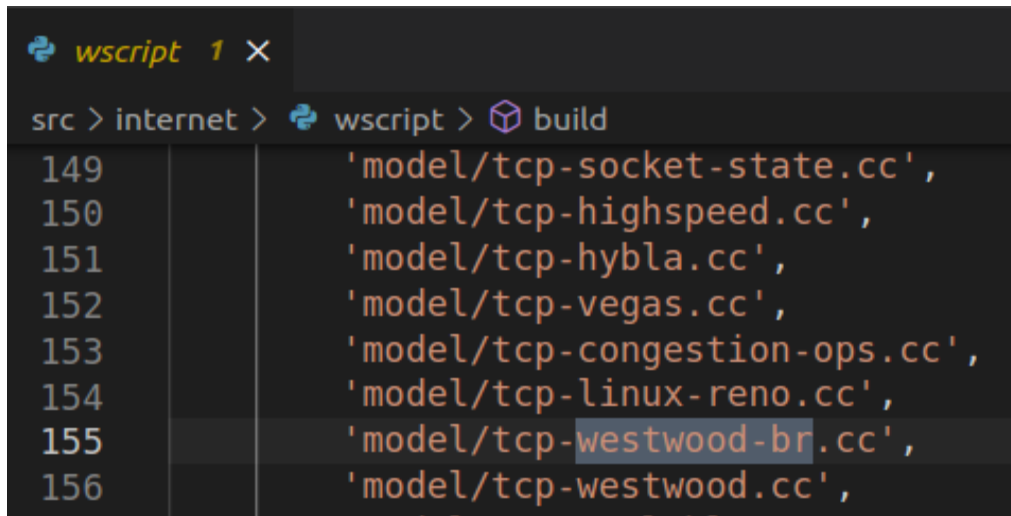
Figure 8 : Change GetName to virtual

6. To implement the TCPW-BR algorithm, we add a new class and header file in the src/internet/model module. We add the paths of the new class and header file in the 'ns-allinone-3.35/ns-3.35/src/internet/wscript' file.



```
wscript 1 X
src > internet > wscript > build
410 'model/ipv6-address-generator.h',
411 'model/tcp-highspeed.h',
412 'model/tcp-hybla.h',
413 'model/tcp-vegas.h',
414 'model/tcp-congestion-ops.h',
415 'model/tcp-linux-reno.h',
416 'model/tcp-westwood-br.h',
417 'model/tcp-westwood.h',
418 'model/tcp-scalable.h',
```

Figure 9 : Update in wscript



```
wscript 1 x
src > internet > wscript > build
149      'model/tcp-socket-state.cc',
150      'model/tcp-highspeed.cc',
151      'model/tcp-hybla.cc',
152      'model/tcp-vegas.cc',
153      'model/tcp-congestion-ops.cc',
154      'model/tcp-linux-reno.cc',
155      'model/tcp-westwood-br.cc',
156      'model/tcp-westwood.cc',
```

Figure 10 : Update in wscript

-
7. Then we create a header file in the src/internet/model folder named 'tcp-westwood-br.h'.

```
class TcpWestwoodBR : public TcpNewReno
{
public:
    /**
     * \brief Get the type ID.
     * \return the object TypeId
     */
    static TypeId GetTypeId(void);

    TcpWestwoodBR(void);
    /**
     * \brief Copy constructor
     * \param sock the object to copy
     */
    TcpWestwoodBR(const TcpWestwoodBR &sock);
    virtual ~TcpWestwoodBR(void);

    /**
     * \brief Protocol variant (Westwood or Westwood+)
     */
    // enum ProtocolType
    // {
    //     WESTWOOD,
    //     WESTWOODPLUS
    // };

    /**
     * \brief Filter type (None or Tustin)
     */
    enum FilterType
    {
        NONE,
        TUSTIN
    };

    // new added
    virtual std::string GetName() const;
    virtual void UpdateWindow(Ptr<TcpSocketState> tcb);
    virtual void IncreaseWindow(Ptr<TcpSocketState> tcb, uint32_t segmentsAked);
};
```

Figure 11 : Add TcpWestwoodBR header file

The TcpWestwoodBR class inherits the TcpNewReno class. Here we declare the GetName method and UpdateWindow Method.

```

private:
    /**
     * Update the total number of acknowledged packets during the current RTT
     *
     * \param [in] acked the number of packets the currently received ACK acknowledges
     */
    void UpdateAkedSegments(int acked);

    /**
     * Estimate the network's bandwidth
     *
     * \param [in] rtt the RTT estimation.
     * \param [in] tcb the socket state.
     */
    void EstimateBW(const Time &rtt, Ptr<TcpSocketState> tcb);
    // new added
    int CongestionLevel(const Time &rtt, Ptr<TcpSocketState> tcb);

protected:
    // new added
    virtual uint32_t SlowStart(Ptr<TcpSocketState> tcb, uint32_t segmentsAked);
    virtual void CongestionAvoidance(Ptr<TcpSocketState> tcb, uint32_t segmentsAked);

    TracedValue<double> m_currentBW; //!< Current value of the estimated BW
    double m_lastSampleBW;          //!< Last bandwidth sample
    double m_lastBW;                //!< Last bandwidth sample after being filtered
    // enum ProtocolType m_pType;    //!< 0 for Westwood, 1 for Westwood+
    enum FilterType m_fType;        //!< 0 for none, 1 for Tustin

    uint32_t m_ackedSegments;        //!< The number of segments ACKed between RTTs
    bool m_IsCount;                  //!< Start keeping track of m_ackedSegments for Westwood+ if TRUE
    EventId m_bwEstimateEvent;        //!< The BW estimation event for Westwood+
    Time m_lastAck;                  //!< The last ACK time

    double control_P;                //!< growth factor
    int Level;                       //!< congestion Level
};

} // namespace ns3

#endif /* TCP_WESTWOODBR_H */

```

Figure 12 : TcpWestwoodBR methods and variables

Here we add two new variables for congestion level and growth factor.

8. Then we create a file in the src/internet/model folder named 'tcp-westwood-br.cc' to define the methods and constructors.

```
src > internet > model > tcp-westwood-br.cc > ...
35 #include "ns3/log.h"
36 #include "ns3/simulator.h"
37 #include "rtt-estimator.h"
38 #include "tcp-socket-base.h"
39
40 NS_LOG_COMPONENT_DEFINE("TcpWestwoodBR");
41
42 namespace ns3
43 {
44     NS_OBJECT_ENSURE_REGISTERED(TcpWestwoodBR);
45
46    TypeId
47     TcpWestwoodBR::GetTypeId(void)
48     {
49         static TypeId tid = TypeId("ns3::TcpWestwoodBR")
50             .SetParent<TcpNewReno>()
51             .SetGroupName("Internet")
52             .AddConstructor<TcpWestwoodBR>()
53             .AddAttribute("FilterType", "Use this to choose no filter or Tustin's approximation filter",
54                 EnumValue(TcpWestwoodBR::TUSTIN), MakeEnumAccessor(&TcpWestwoodBR::m_fType),
55                 MakeEnumChecker(TcpWestwoodBR::NONE, "None", TcpWestwoodBR::TUSTIN, "Tustin"))
56             .AddTraceSource("EstimatedBW", "The estimated bandwidth",
57                 MakeTraceSourceAccessor(&TcpWestwoodBR::m_currentBW),
58                 "ns3::TracedValueCallback::Double");
59         return tid;
60     }
61
62     TcpWestwoodBR::TcpWestwoodBR(void) : TcpNewReno(),
63         m_currentBW(0),
64         m_lastSampleBW(0),
65         m_lastBW(0),
66         m_ackedSegments(0),
67         m_IsCount(false),
68         m_lastAck(0),
69         control_P(1.0),
70         Level(1)
71     {
72         NS_LOG_FUNCTION(this);
73     }
74 }
```

Figure 13 : TcpWestwoodBR's constructors definition

9. To Get the name of the algorithm first we define **GetName** method of **TcpWestwoodBR** class.

```
src > internet > model > tcp-westwood-br.cc > ...
253
254     std::string
255     TcpWestwoodBR::GetName() const
256     {
257         return "TcpWestwoodBR";
258     }
259 }
```

Figure 14: GetName() method definition

-
10. After Each acknowledgement, This algorithm needs to determine the congestion level based on RTT. So We define a method to determine the level and growth factor.

```
260 int
261 TcpWestwoodBR::CongestionLevel(const Time &rtt, Ptr<TcpSocketState> tcb)
262 {
263     // std::cout << "rtt : " << rtt << "max : " << tcb->m_maxRtt << " min : " << tcb->m_minRtt << std::endl;
264     double F = (tcb->m_maxRtt - tcb->m_minRtt).GetMilliSeconds();
265
266     double d = (rtt - tcb->m_minRtt).GetMilliSeconds();
267
268     double R = (d * 1.0) / F;
269
270     // std::cout << "F :" << F << " d : " << d << std::endl;
271     // std::cout << "\ntcpw-br R : " << R << std::endl;
272
273     if (R >= 0 && R <= 0.25)
274     {
275         Level = 1;
276         control_P = 1;
277     }
278     else if (R <= 0.5)
279     {
280         Level = 2;
281         control_P = 0.867;
282     }
283     else if (R <= 0.75)
284     {
285         Level = 3;
286         control_P = 0.5;
287     }
288     else
289     {
290         Level = 4;
291         control_P = 0.4;
292     }
293
294     // std::cout << "\nP : " << control_P << " Level : " << Level << std::endl;
295
296     return Level;
297 }
```

Figure 15 : CongestionLevel() method defination

Then we call the CongestionLevel method after each acknowledgement is received.

```

void
TcpWestwoodBR::PktsAacked(Ptr<TcpSocketState> tcb, uint32_t packetsAacked,
                           const Time &rtt)
{
    NS_LOG_FUNCTION(this << tcb << packetsAacked << rtt);

    if (rtt.IsZero())
    {
        NS_LOG_WARN("RTT measured is zero!");
        return;
    }

    m_ackedSegments += packetsAacked;

    CongestionLevel(rtt, tcb);

    EstimateBW(rtt, tcb);
}

```

Figure 16: PktsAacked() method of TcpWestwoodBR class

(1) Each time an ACK of a new data segment is received,
 If (congestion level=1||congestion level=2)//Think it is wireless packet loss, mild congestion

Cwnd=Cwnd+1;

If (Cwnd>Ssthresh)

Cwnd=Cwnd+(1/Cwnd)*p;

11. [congestion level=1||congestion level=2](#)

Figure 17: Slow start and Congestion Avoidance part of TCPW BR algorithm

To implement the first part of the TCPW BR algorithm, we modify the CongestionAvoidance method. If the congestion level ==1 or the congestion level == 2, then we multiply the adder with the growth factor.

```

233
234 void
235 TcpWestwoodBR::CongestionAvoidance(Ptr<TcpSocketState> tcb, uint32_t segmentsAked)
236 {
237     NS_LOG_FUNCTION(this << tcb << segmentsAked);
238
239     if (segmentsAked > 0)
240     {
241         // std::cout << "tcpw-br here in congestion avoidance\n";
242         double adder = static_cast<double>(tcb->m_segmentSize * tcb->m_segmentSize) / tcb->m_cWnd.Get();
243
244         // new add
245         if (Level == 1 || Level == 2)
246             adder = adder * control_P;
247
248         adder = std::max(1.0, adder);
249         tcb->m_cWnd += static_cast<uint32_t>(adder);
250         NS_LOG_INFO("In CongAvoid, updated to cwnd " << tcb->m_cWnd << " ssthresh " << tcb->m_ssThresh);
251     }
252 }
253
254 std::string
255 TcpWestwoodBR::GetName() const
256 {
257     return "TcpWestwoodBR";
258 }

```

Figure 17 : CongestionAvoidance method of TcpWestwoodBR

12. When duplicate acknowledgement is received, dupack method of tcp-socket-base checks the congestion control algorithm name and if the name is "TcpWestwoodBR", then it calls the UpdateWindow method of the TcpWestwoodBR.

```

tcp-socket-base.h  tcp-socket-base.cc X
rc > internet > model > tcp-socket-base.cc > {} ns3 > DupAck(uint32_t)
1639     ++m_dupAckCount;
1640 }
1641
1642 // new added
1643 m_tcb->m_inDUPACK = true;
1644 m_tcb->m_DUPACKCOUNT = m_dupAckCount;
1645 if (m_congestionControl->GetName() == "TcpWestwoodBR")
1646 {
1647     // std::cout << "dup ack count : " << m_dupAckCount << std::endl;
1648     // std::cout << "cwnd : " << m_tcb->m_cWnd << std::endl;
1649     // m_tcb->m_cWnd = m_tcb->m_cWnd *
1650     // std::cout << "algo : " << m_congestionControl->GetName() << "\t L: " << m_tcb->m_inDUPACK << std::endl;
1651     m_congestionControl->UpdateWindow(m_tcb);
1652 }
1653 m_tcb->m_inDUPACK = false;
1654

```

Figure 18 : UpdateWindow method call from tcp-socket-base

13. In the UpdateWindow method, it checks the dupack count and the congestion level and updates the congestion window size accordingly.

```

168
169 // new added
170 void
171 TcpWestwoodBR::UpdateWindow(Ptr<TcpSocketState> tcb)
172 {
173     // std::cout << "here in TcpWestwoodBR::Update window\n\n";
174     if (tcb->m_inDUPACK)
175     {
176         if (tcb->m_DUPACKCOUNT == 2 && (Level == 3 || Level == 4))
177         {
178             tcb->m_cWnd = tcb->m_cWnd * control_P;
179             if (tcb->m_cWnd > tcb->m_ssThresh)
180                 tcb->m_cWnd = tcb->m_ssThresh;
181         }
182
183         if ((tcb->m_DUPACKCOUNT == 3) && Level > 2)
184         {
185             tcb->m_cWnd = tcb->m_cWnd * control_P;
186             if (tcb->m_cWnd > tcb->m_ssThresh)
187                 tcb->m_cWnd = tcb->m_ssThresh;
188         }
189     }
190 }
191

```

Figure 19: UpdateWindow method of TcpWestwoodBR

Network topology under Simulation

For Proposed Modification:

```
27
28 // Network Topology
29 //
30 // Wifi 10.1.3.0
31 //          AP
32 // *      *      *
33 // |      |      |      10.1.1.0
34 // n6    n7    n0 ----- n1    n2    n3
35 //                               point-to-point |    |    |
36 //                                               =====
37 //                                               LAN 10.1.2.0
38
```

Figure 20 : Network Topology for Task B

TCPW BR paper used a mixed wired-wireless topology. That's why we use a mixed wired-wireless topology to test our modified algorithm. Here, n0, n7 and n6 are the wifi nodes. n0 is an access point node for the wireless part. The link between n0 and n1 is the bottleneck link. Here, n6 and n7 are the stationary nodes of the wireless part. Here, n1, n2 and n3 are the csma nodes. Data rate was set at 100Mbps for csma nodes. The bottleneck bandwidth was set at 5Mbps. Mainly We follow examples/third.cc file for the implementation of this mixed wired-wireless topology. To install the sink application in the csma nodes, we use PacketSinkHelper. To install the source application, we use BulkSendHelper. The maxbytes attribute of BulkSendHelper was set to unlimited bytes.

Results & Explanation of Task B

Throughput:

The TCP versions New Reno, TCPW and TCPW BR were simulated on the wireless link using the NS3 simulation platform and their changes were recorded. Fig. 20 shows the network topology.

The network related parameters are: the bottleneck link bandwidth is 5Mbps, the unidirectional transmission time is 2ms. During the simulation, the wireless link error rate was set as 3%.

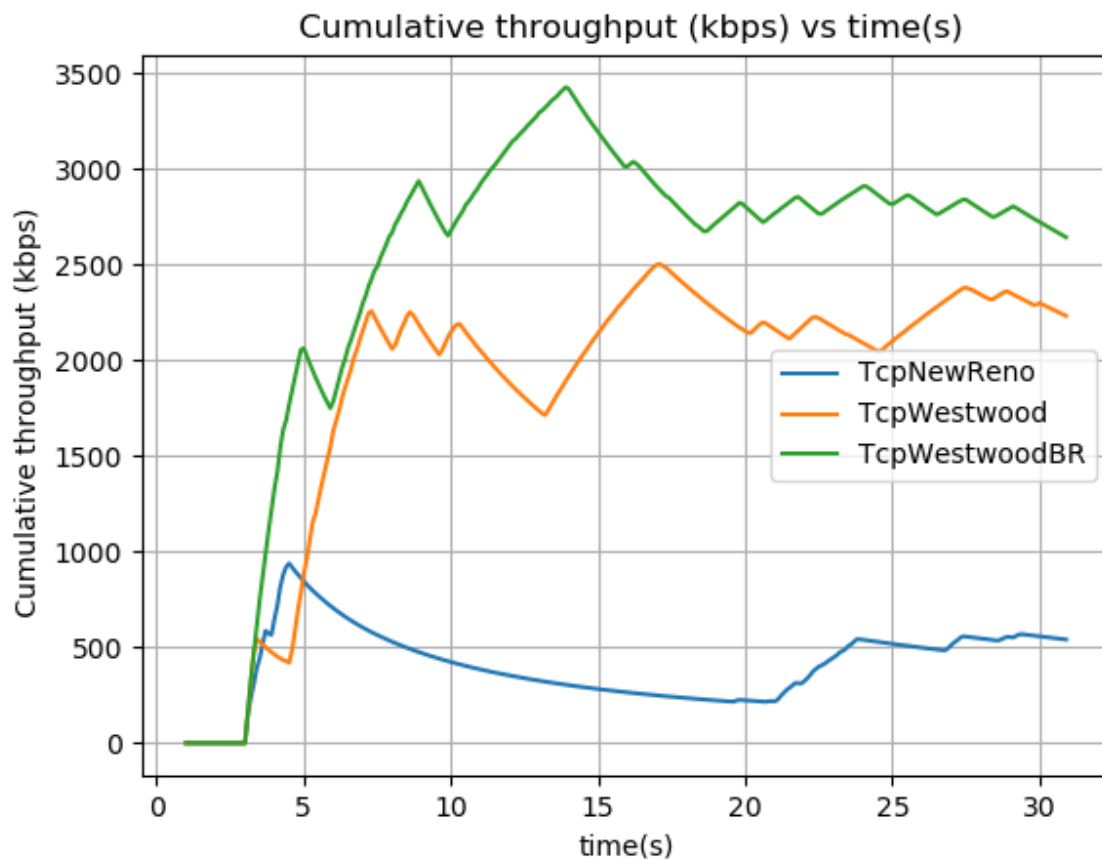


Figure 21 : Comparison Cumulative Throughput

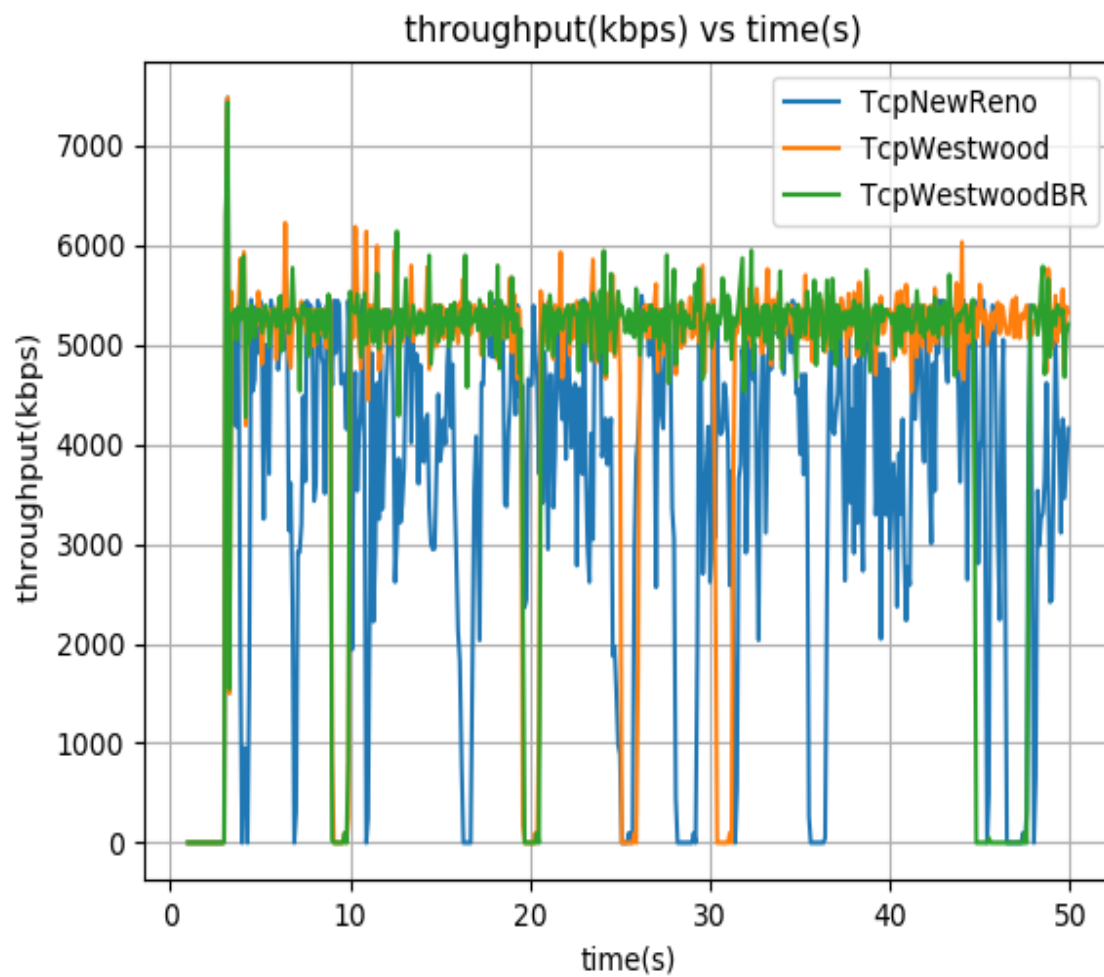


Figure 22: Comparison interval throughput

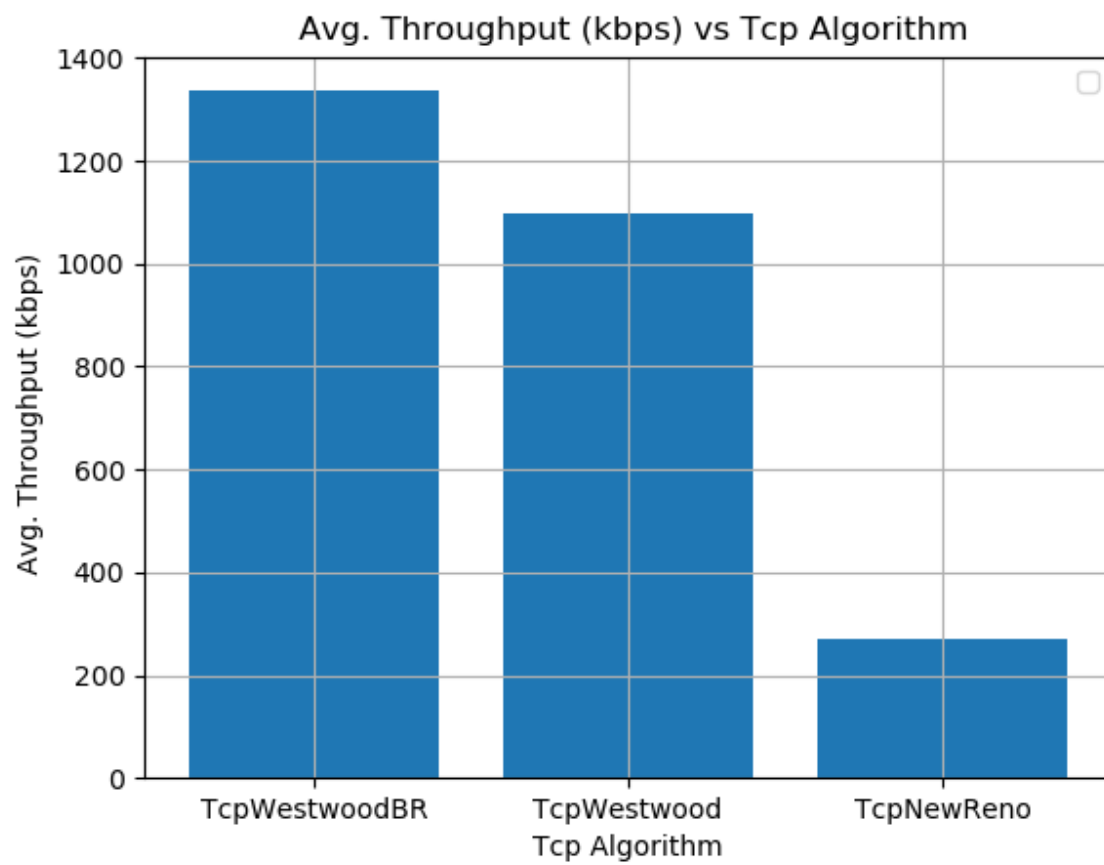


Figure 23 : Comparison Average Throughput

Packet Loss Ratio :

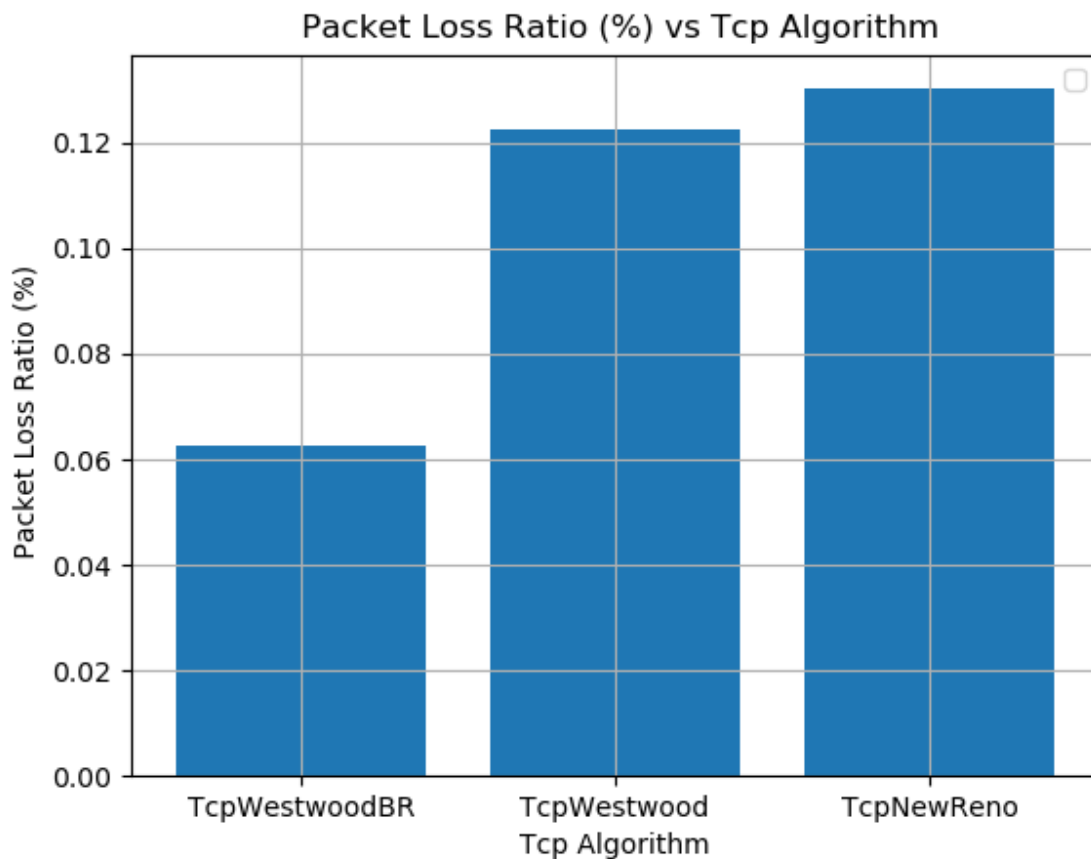


Figure 24: Comparison of packet loss ratios

Explanation

It is easy to see from Fig.21 that the average cumulative throughput of TCPW BR is nearly 20% higher than TCP Westwood, almost six times that of New Reno. We can also see that from the bar chart of Fig. 23. This is because TCPW BR effectively distinguishes between wireless packet loss and congestion packet loss, avoids frequently calling congestion mechanism, and improves throughput. Fig. 22 shows the interval throughput of the TCPW BR, TCPW and New Reno algorithms. From this graph, we can also see that TCPW BR is better than both TCPW and New Reno.

As we stated earlier in this report, TCP Westwood and TCP New Reno can't distinguish between congestion and wireless packet loss and they overestimate the available bandwidth. That's why from the cumulative throughput graph and average throughput graph, we can see the throughput of both of these algorithms are less than the throughput of TCPW BR. So we can say that TCPW BR can effectively distinguish between wireless packet loss and congestion packet loss. That's why, TCPW BR is giving better throughput.

The simulation experiment also made a statistic on the packet loss rate corresponding to the three different algorithms. Since TCPW BR is more flexible for the event handling mechanism, the packet loss rate is improved. From Fig. 24, we can say TCPW BR is 0.06 percentage points higher than TCPW and TCPW BR is 0.07 percentage points higher than Tcp New Reno.

Network topology under Simulation

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

For Wireless high rate simulation, We use Fig.1 topology. There are three networks in this topology. In the middle, we can see n0 and n1 nodes which are the access point nodes for two networks. The point to point link between n0 and n1 is two communicate between two wireless networks. The stationary nodes of the 10.1.3.0 network are n5, n6 and n7. The stationary nodes of the 10.1.2.0 network are n2, n3, n4. To install the wireless part here, we mainly follow the examples/third.cc file. To install mobility in the wireless nodes, we use

ns3::MobilityHelper. We use the ns3::RandomWalk2dMobilityMode mobility model. We set the speed of stationary nodes to a constant speed. We use PacketSinkHelper to install applications in sink nodes and OnOffHelper to install applications in source nodes.

For Wireless Low Rate (Mobile):

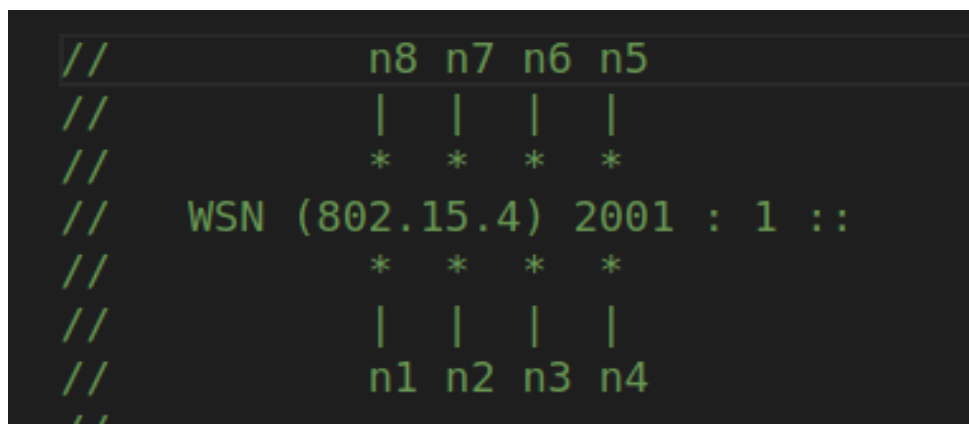


Figure 2 : Wireless Low Rate Topology

For wireless low rate, we use Fig.2 topology. There is only one network in this topology. If we use multiple networks here, we don't receive any significant amount of packets. That's why we use a single network. To create a channel, we use ns3::LrWpanHelper here. We add and install LrWpanNetDevice on each node. We installed the internet ipv6 stack using InternetStackHelper. Then we install SixLowpan layer on the top of LrWpanNetDevice. To install mobility in the nodes, we use the same mobility model as Fig.1 topology. To install applications, we use PacketSinkHelper and OnOffHelper for sink and source apps.

Variation of Parameters

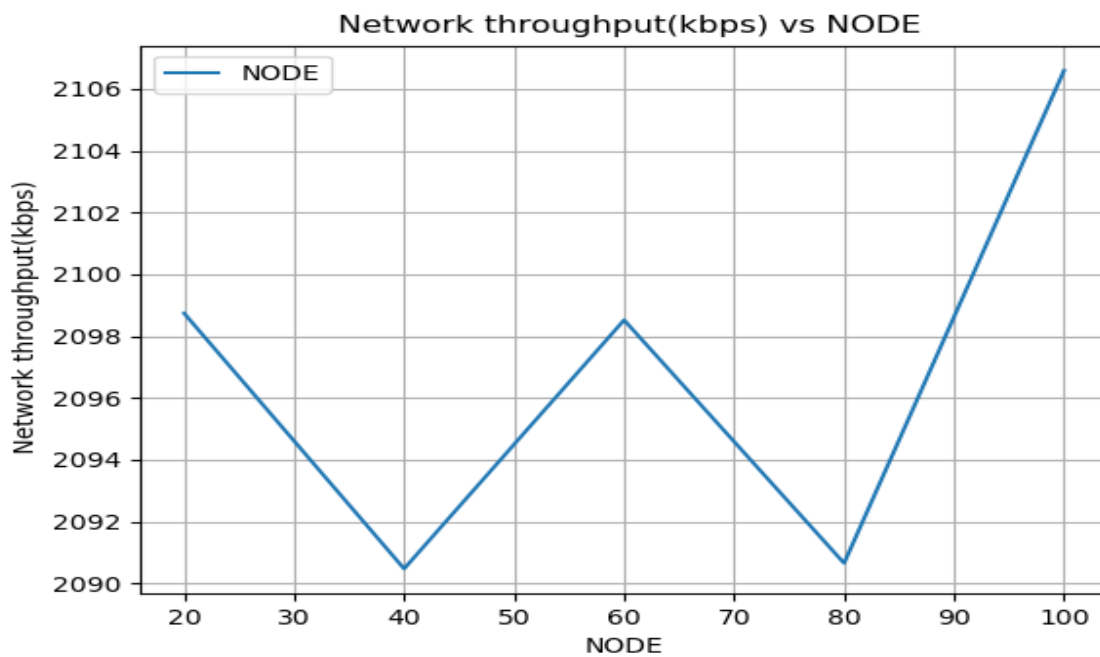
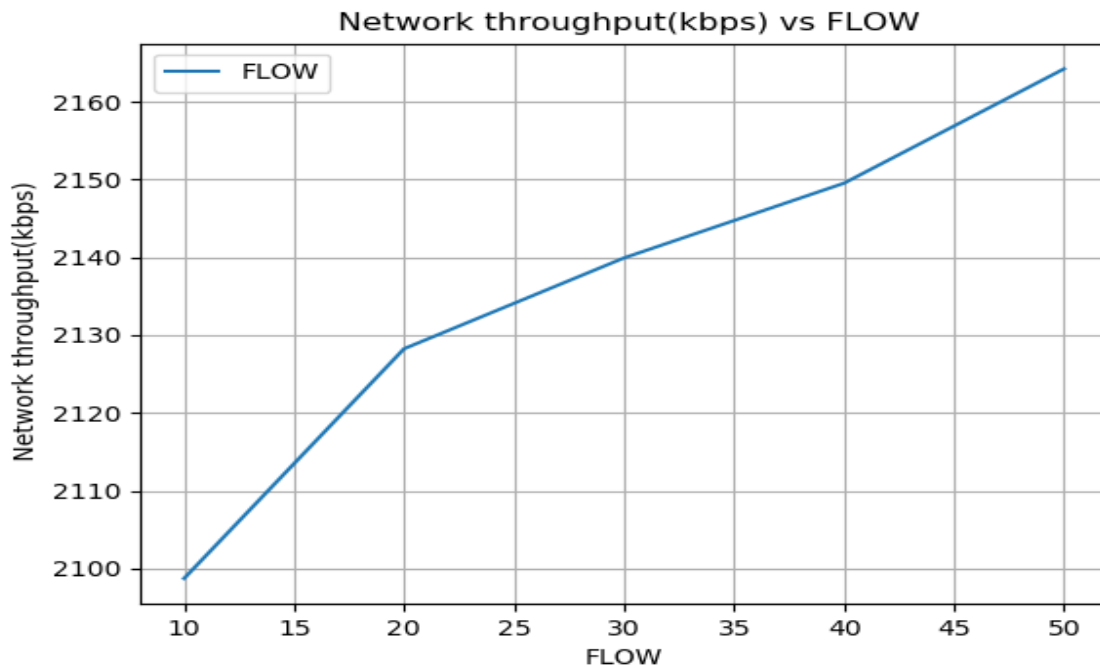
In our simulation, we varied the total number of nodes in the graph. We varied the number of nodes as 20,40,60,80,100.

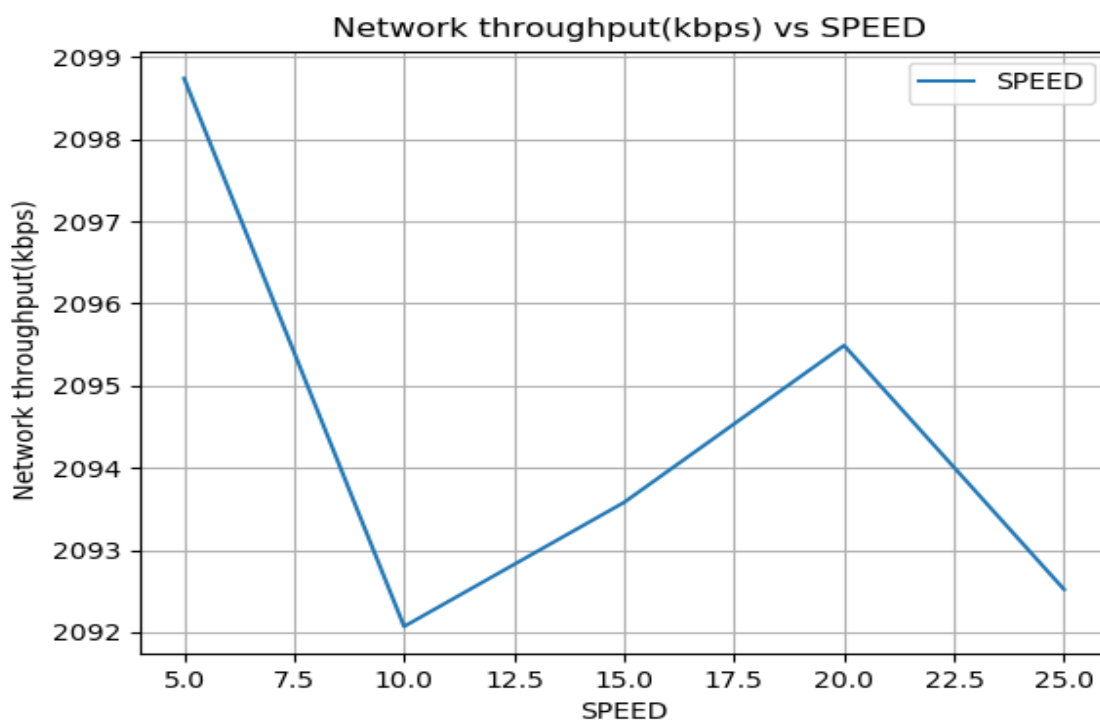
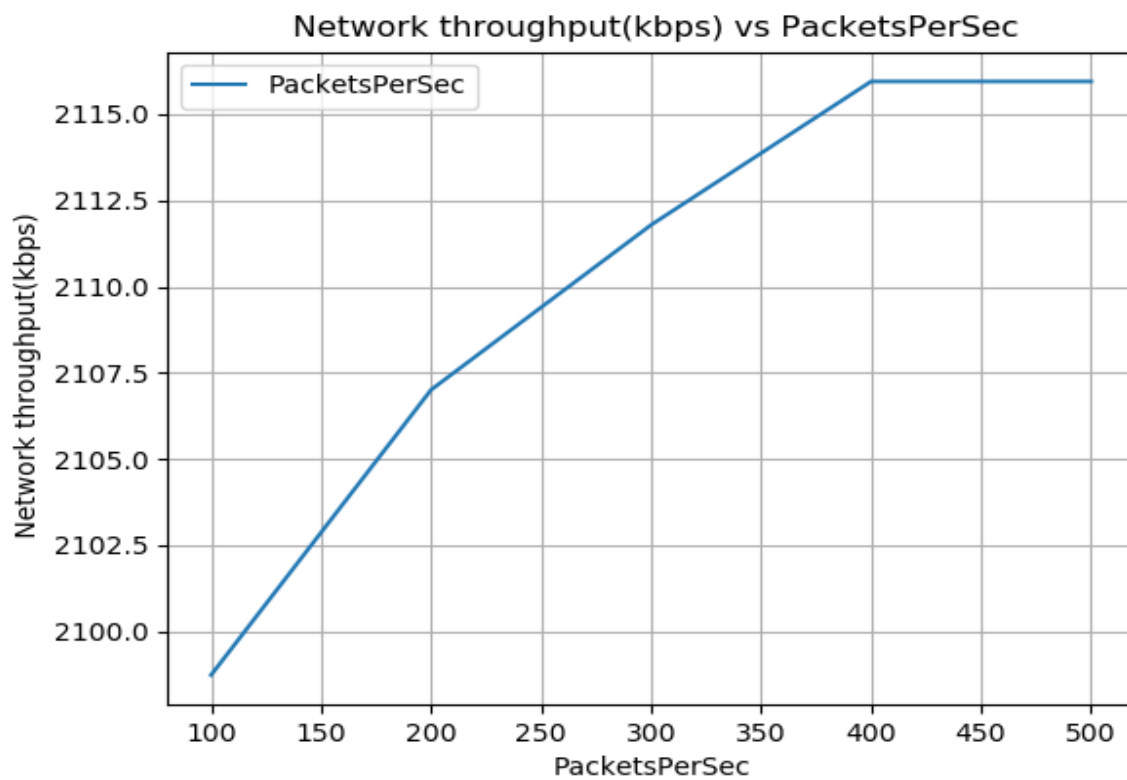
We also varied the number of flows, the number of packets per second, and the speed of the nodes in our simulation. We save network throughput, end to end delay, packet drop ratio and packet delivery ratio for different values of each parameter.

1. The number of nodes are varied as 20, 40, 60, 80, and 100.
2. The number of flows are varied as 10, 20, 30, 40 and 50.
3. The number of packets per second are varied as 100, 200, 300, 400 and 500.
4. Speed of nodes are varied as 5 m/s, 10 m/s, 15 m/s, 20 m/s, and 25 m/s.

Results of Task A (Wifi High Rate - Mobile):

Network Throughput:

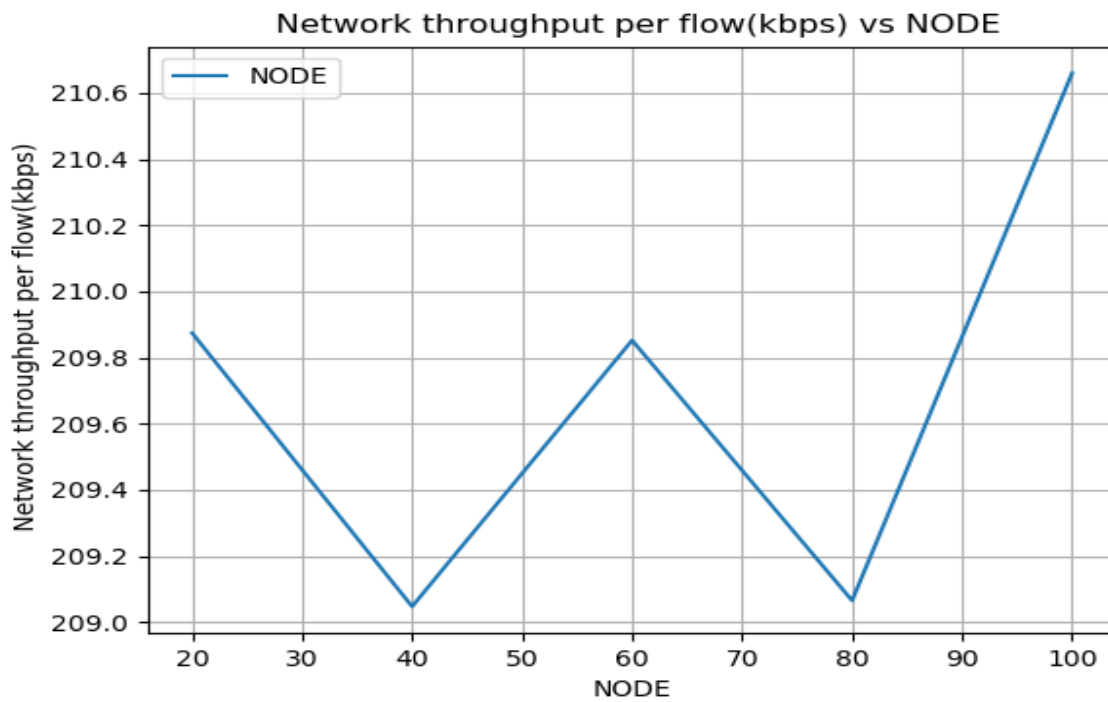
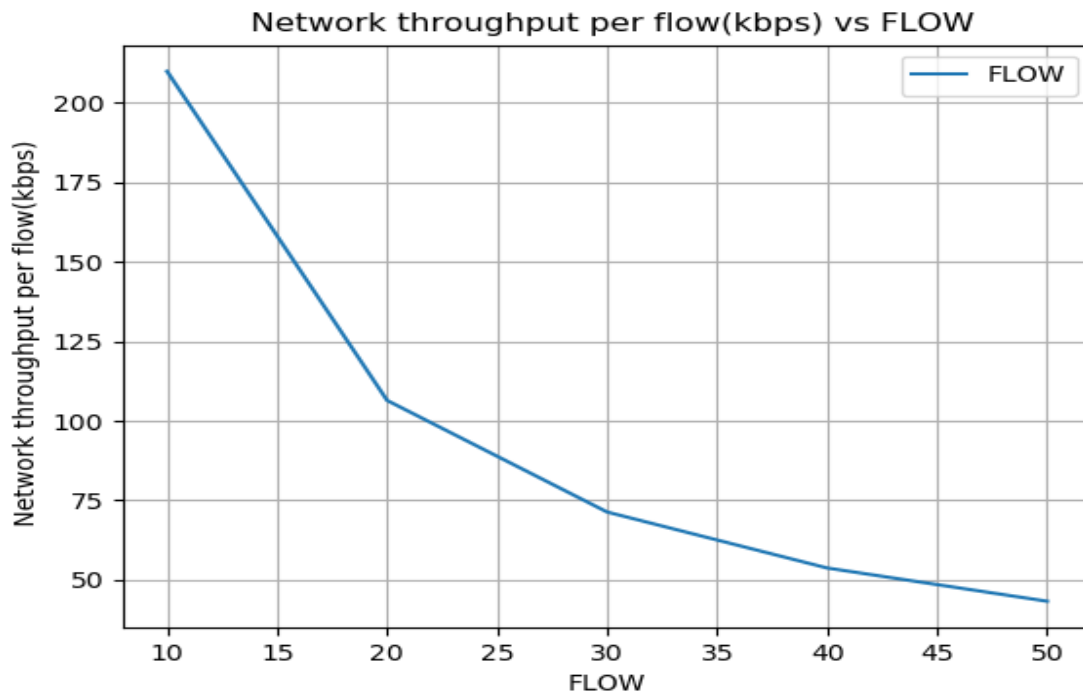


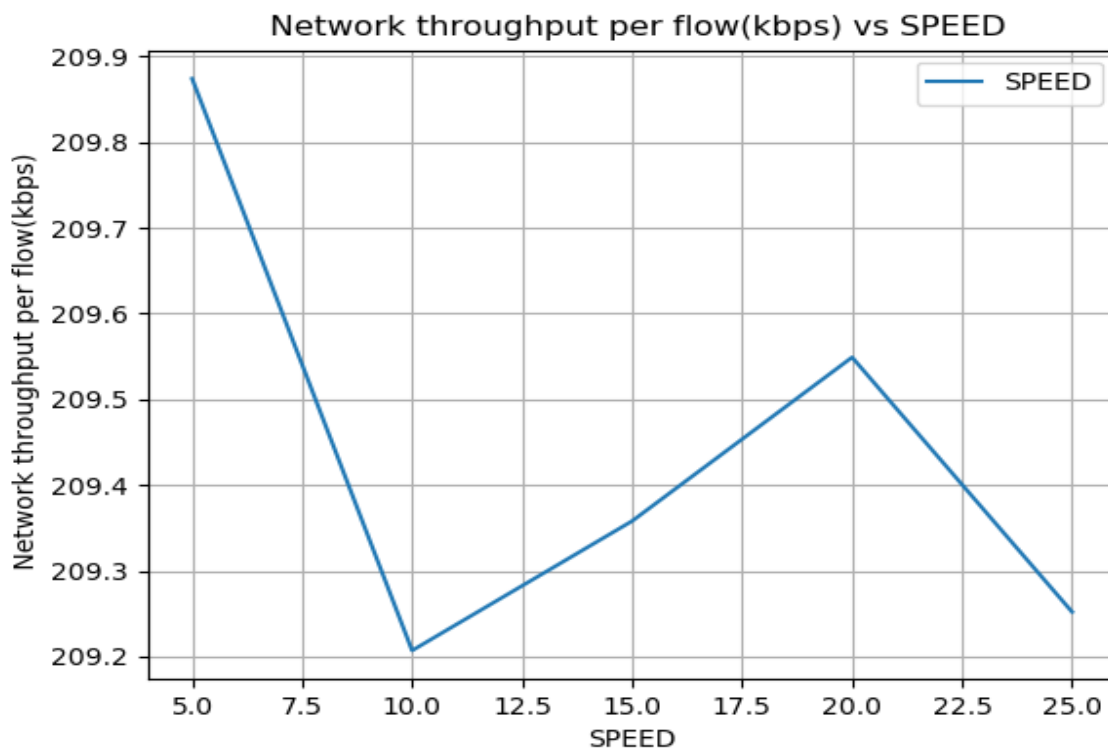
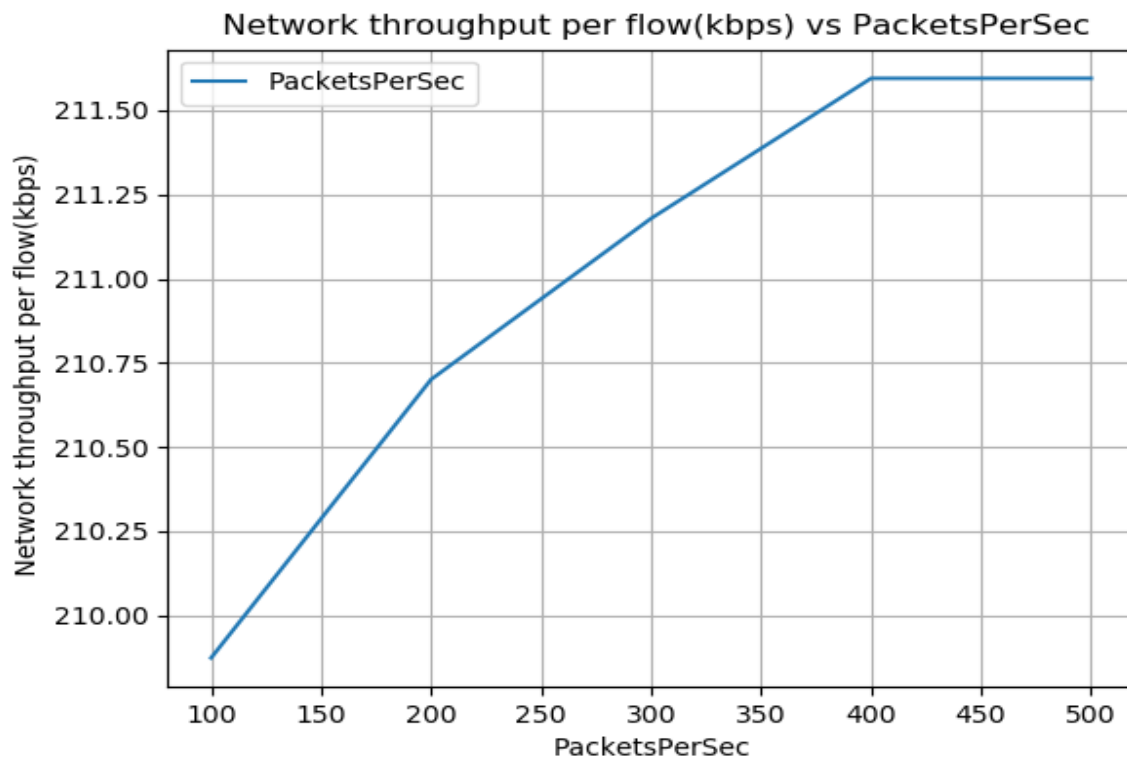


Explanation:

From the above 4 graphs, we can see, as we increase the number of flows and the number of packets per second in the network, the network throughput increases which is logical. But if we increase the number of nodes in the network, then we can see some ups and downs in network throughput because as nodes increase, packet drops in the network will increase. Same reason goes for Speed. As we increase the speed of stationary nodes, the randomness will increase. So The packet loss will increase. As a result, throughput will vary with speed.

Throughput Per Flow:

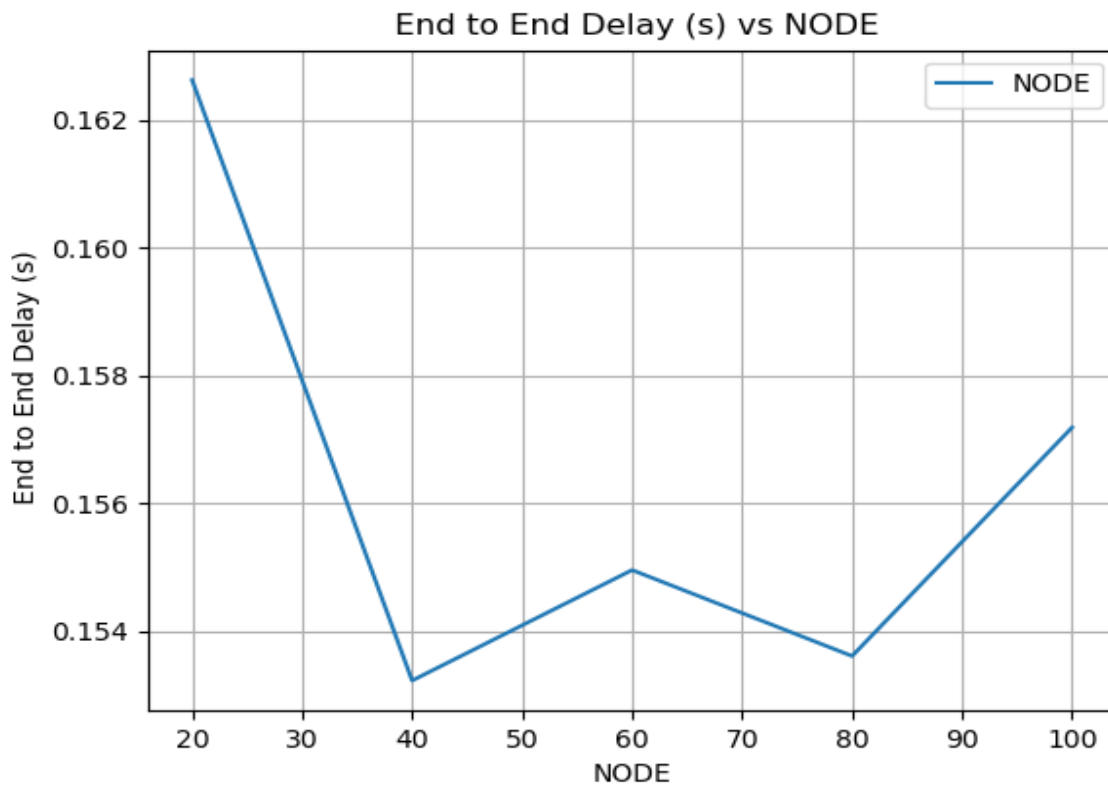
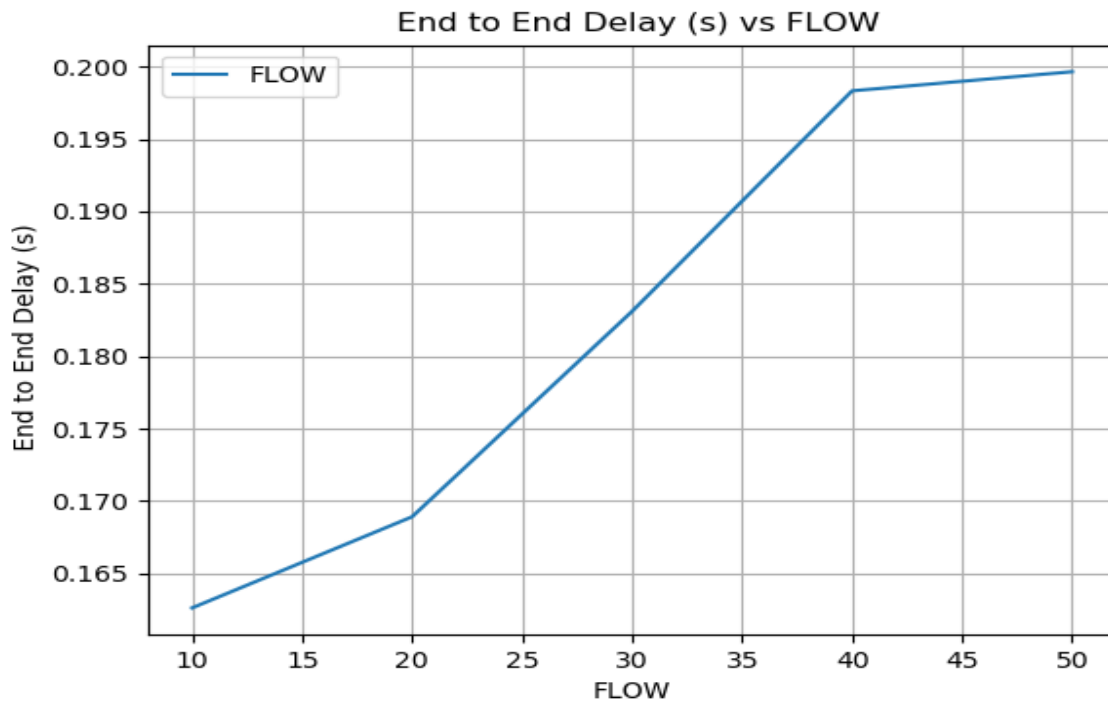


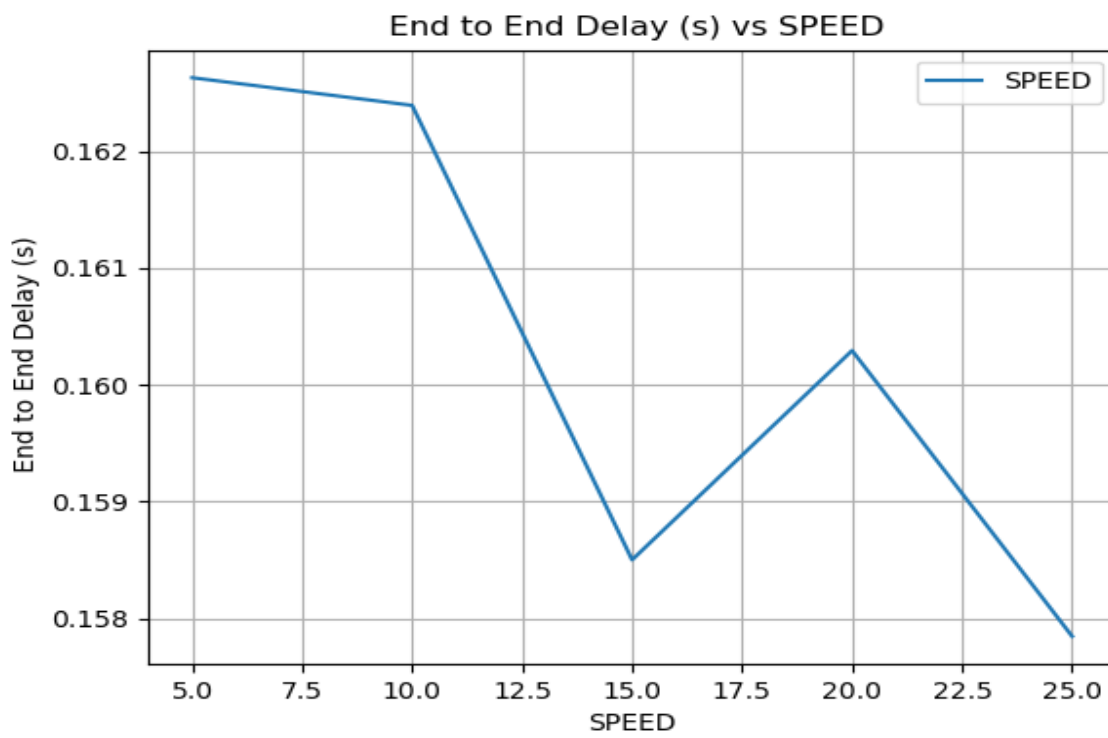
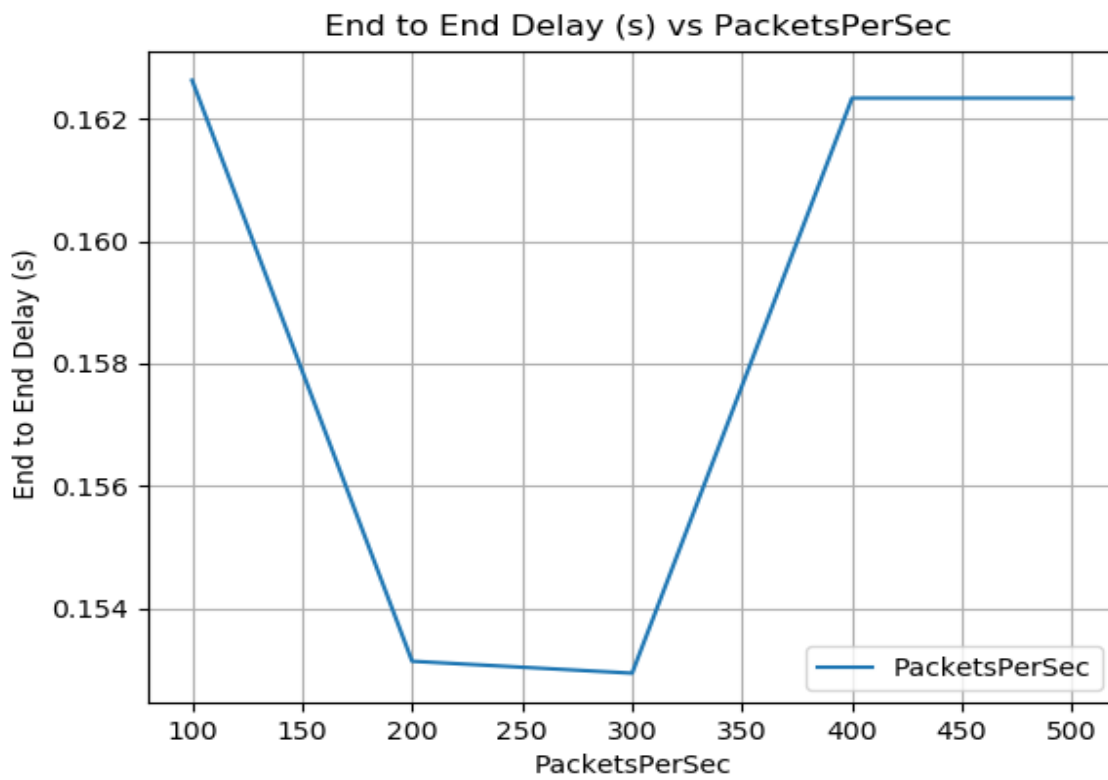


Explanation:

We can see the same graph for throughput per flow for varying the number of nodes, speed of the nodes and packets per second. Because we are not increasing the number of flows here. But when we increase the number of flows, we can see throughput per flow decreases. As we increase the number of flows, the bandwidth is distributed among the flows. That's why we can see throughput decreasing as we increase the number of flows.

End to End Delay:

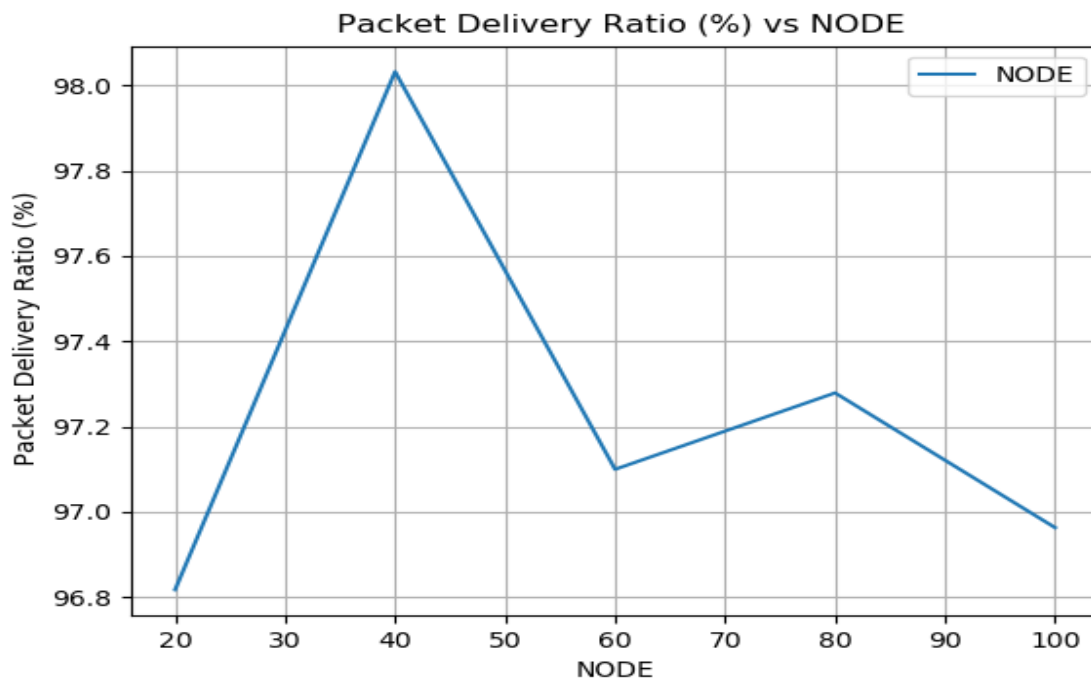
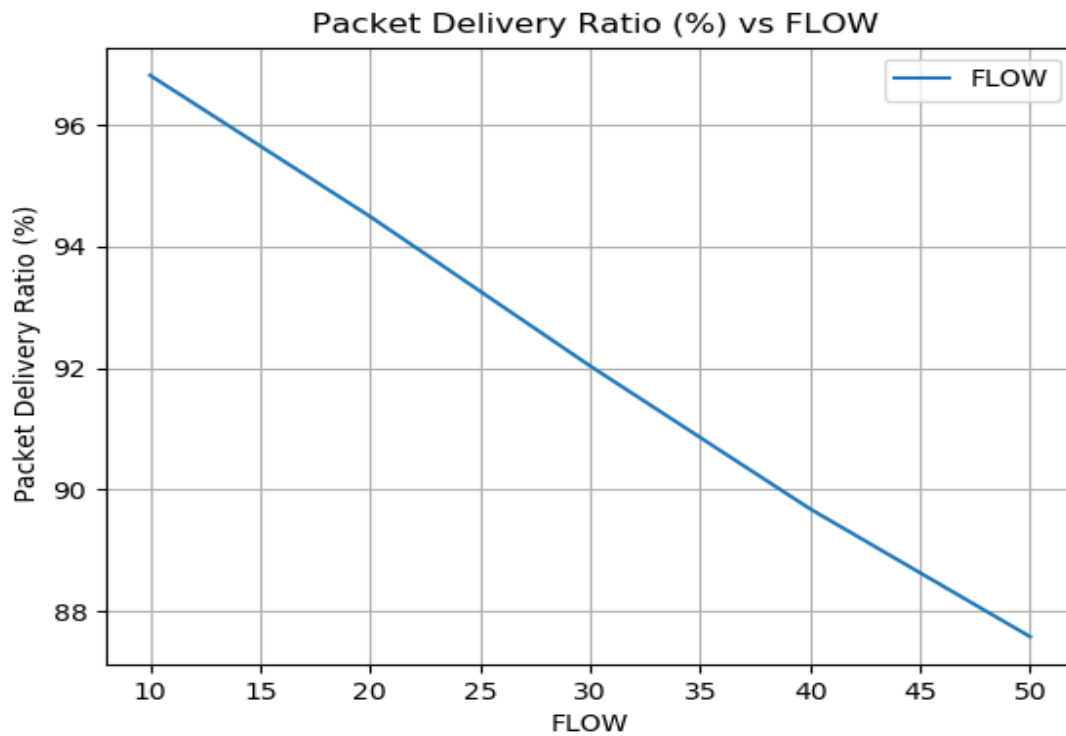


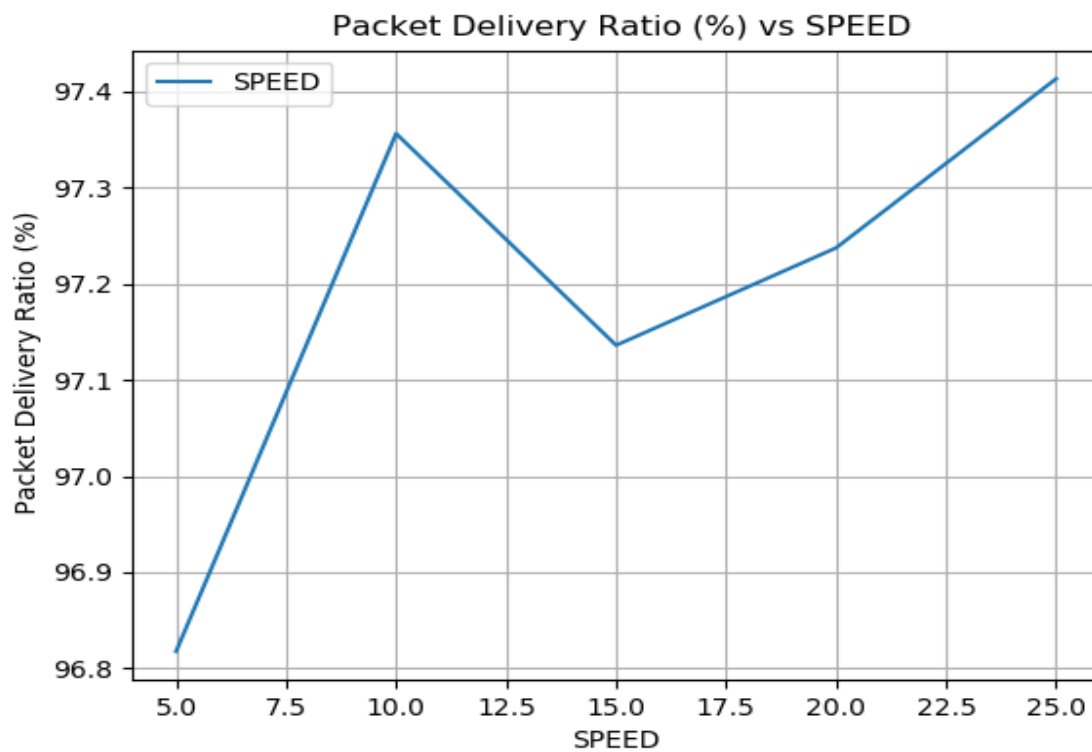
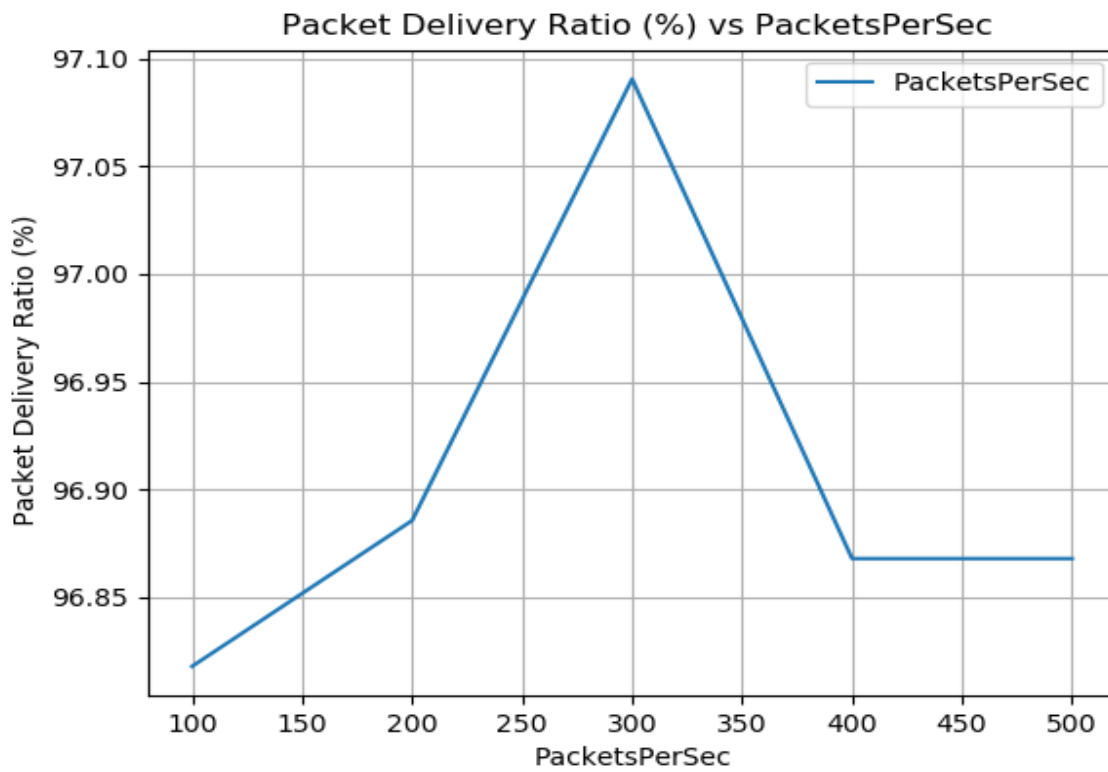


Explanation:

As we increase the number of flows in the network, there will be more congestion in the network. That's why end to end delay is increasing when we increase the number of flows. But as we increase nodes or packets per second or speed of the nodes, we can see delay is decreasing. One reason can be, here we are not increasing the number of flows when we are varying the other's parameters. So As we increase packets per sec, delay is decreasing. For speed and number of node parameters, The nodes which are acting as source and sink applications are at different distances at each flow. But as the number of flows here is fixed, that's why varying others parameters doesn't affect much.

Packet Delivery Ratio:

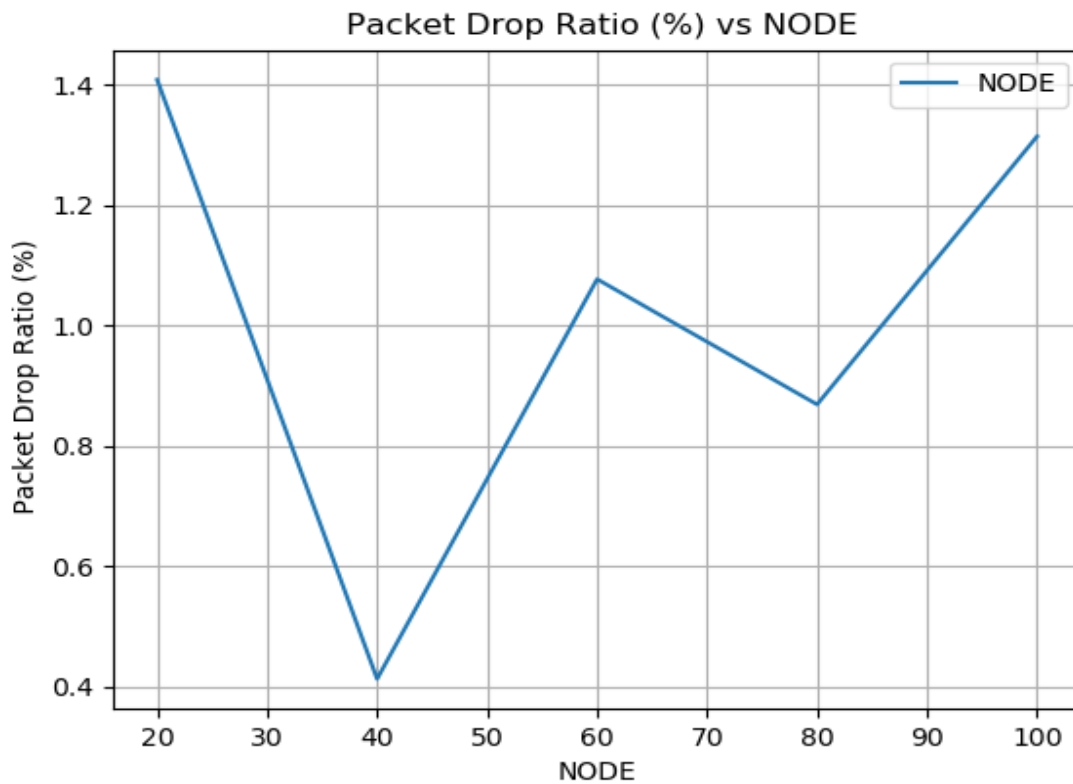
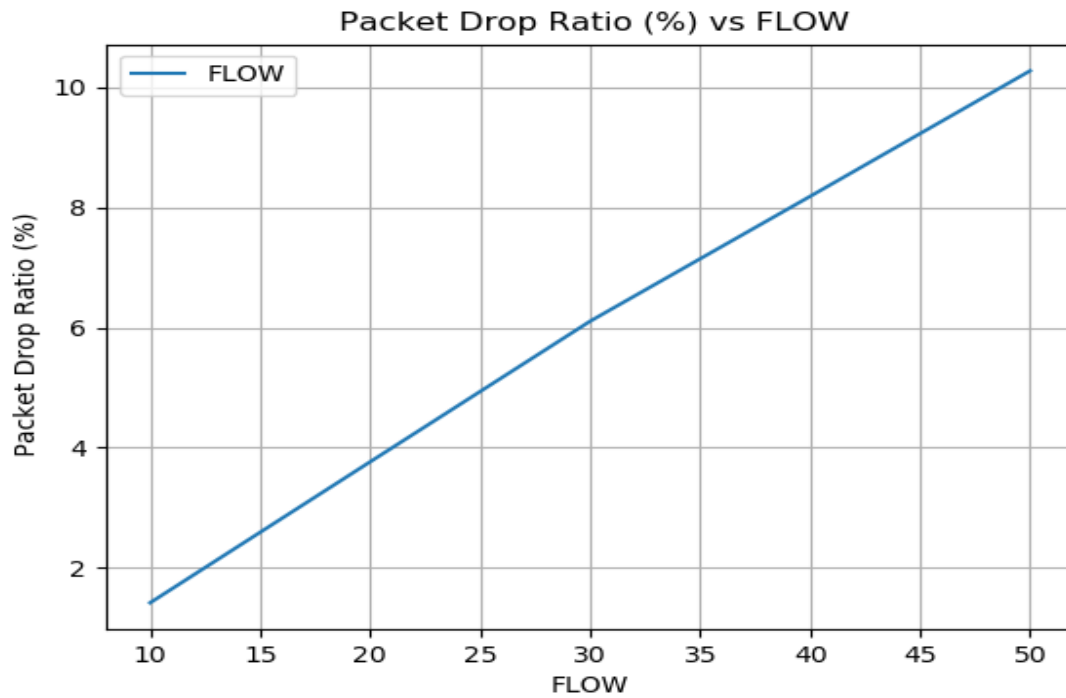


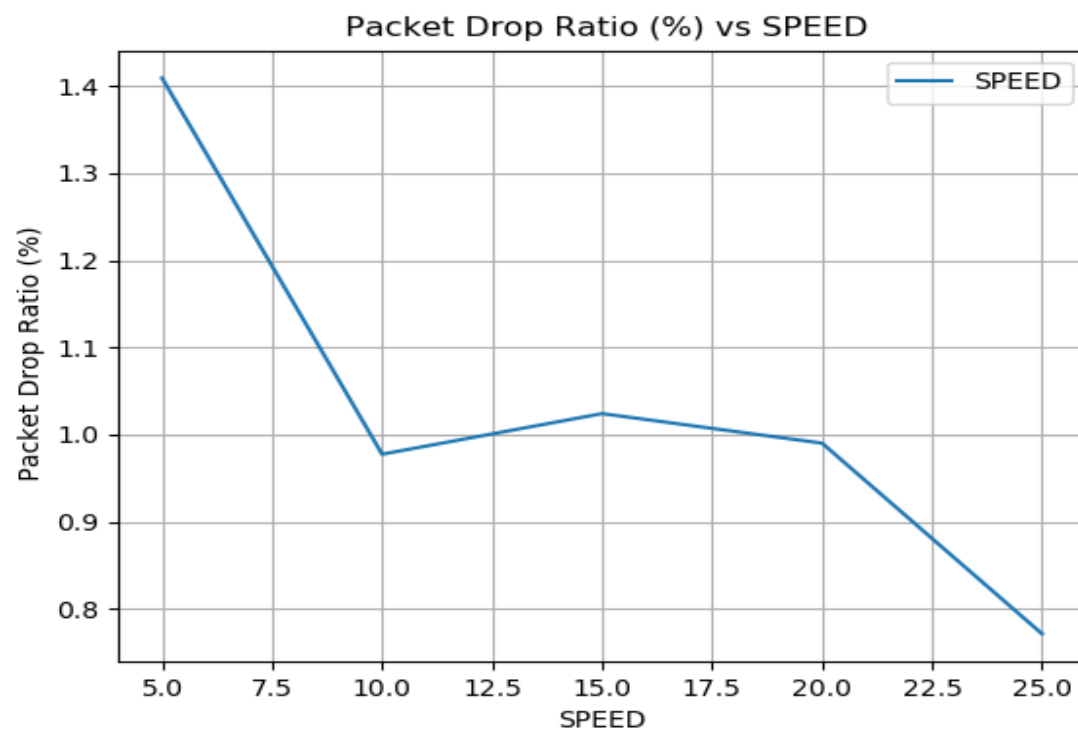
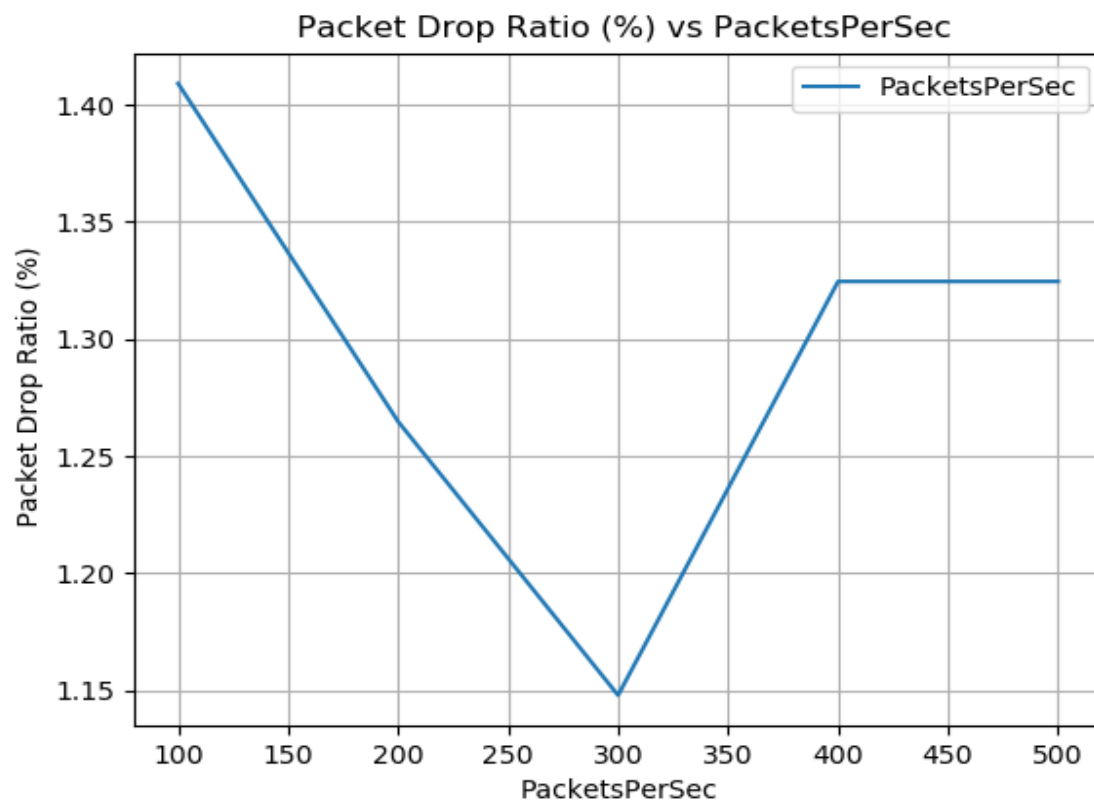


Explanation:

As we increase the number of flows in the network, there will be more congestion in the network. That's why the packet delivery ratio is decreasing when we increase the number of flows. But for other parameters which we are varying, they don't affect much because the number of flows is fixed here. As we can see here, packet delivery ratios are varying between 96 and 97%. So speed of nodes, packets per second or number of nodes does not matter much.

Packet Drop Ratio:



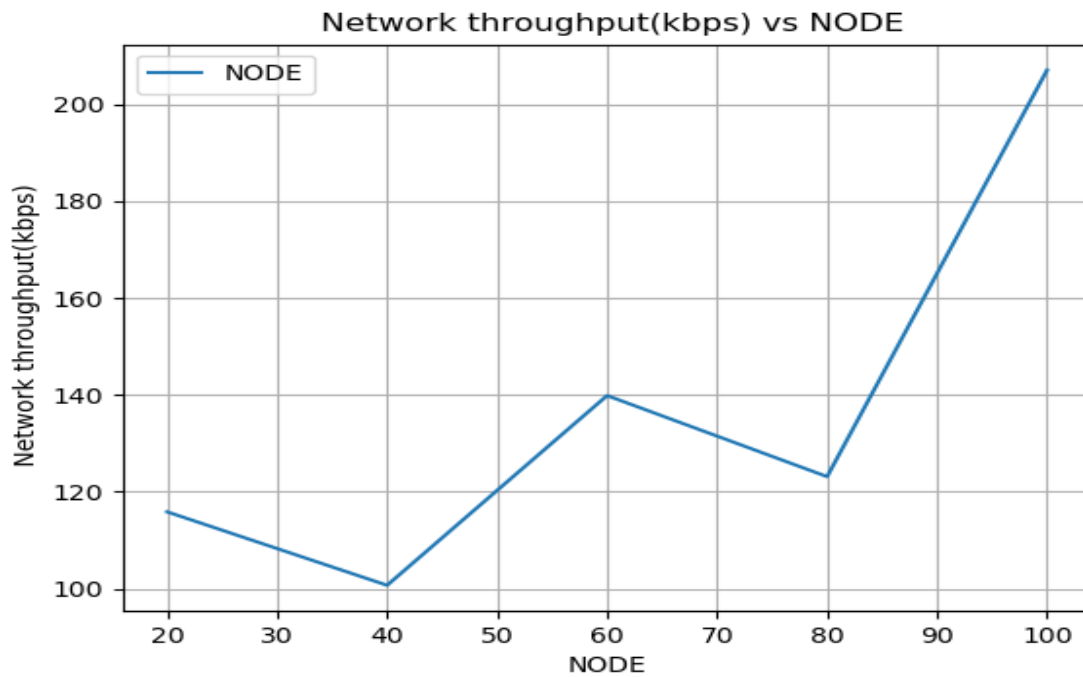
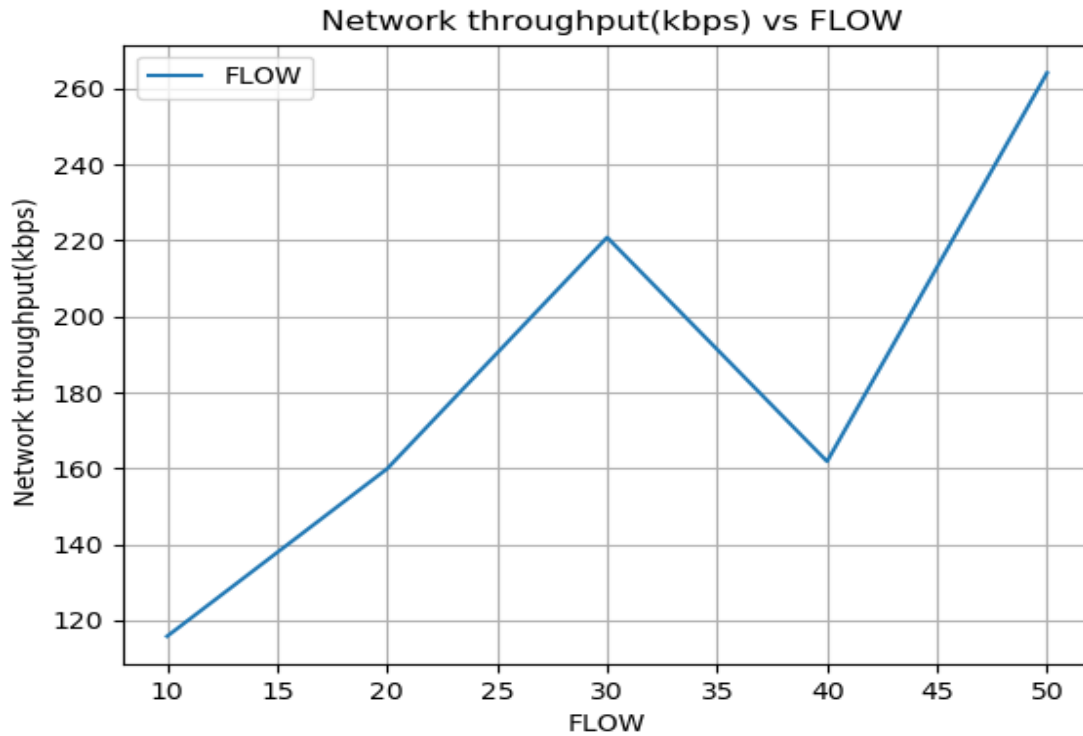


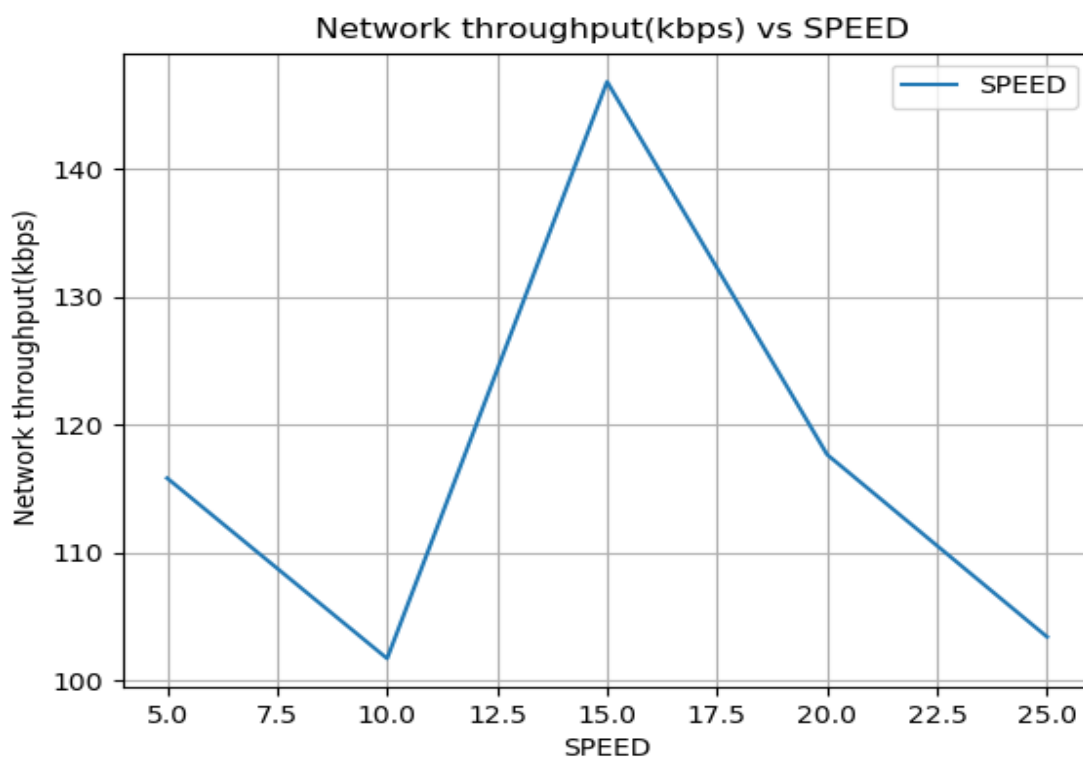
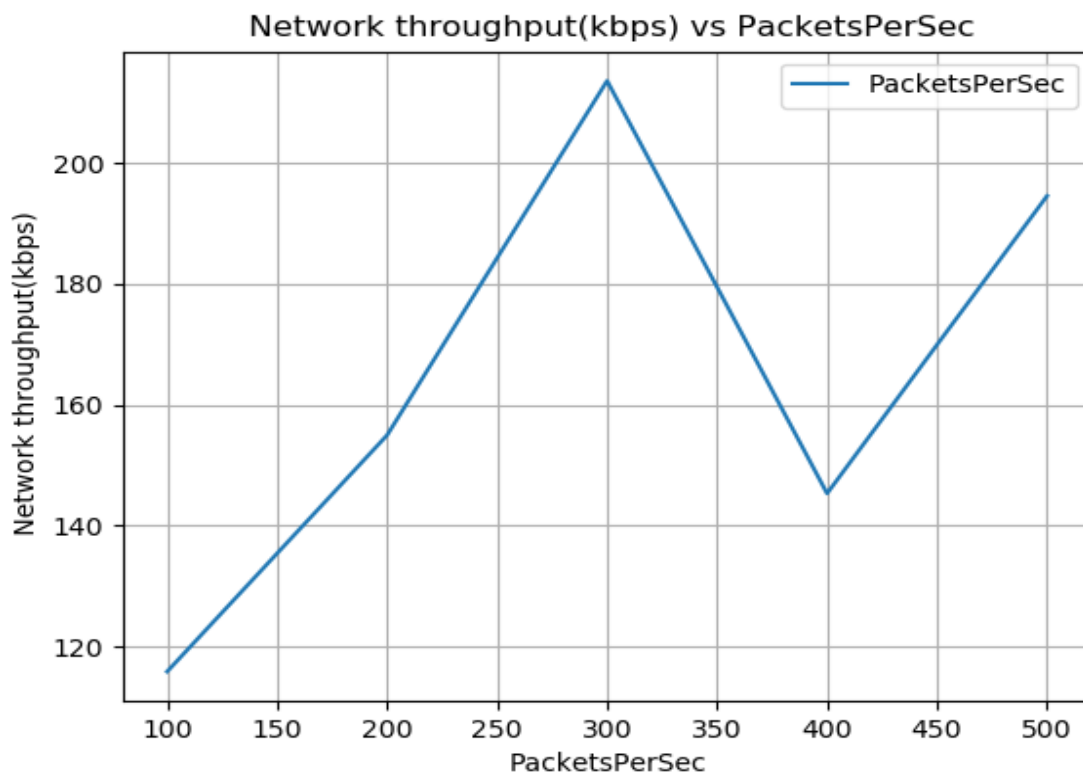
Explanation:

As we increase the number of flows in the network, there will be more congestion in the network. That's why the packet drop ratio is increasing when we increase the number of flows. But for other parameters which we are varying, they don't affect much because the number of flows is fixed here. As we can see here, packet drop ratios are varying between 0.5 and 1.5%. So speed of nodes, packets per second or number of nodes does not matter much.

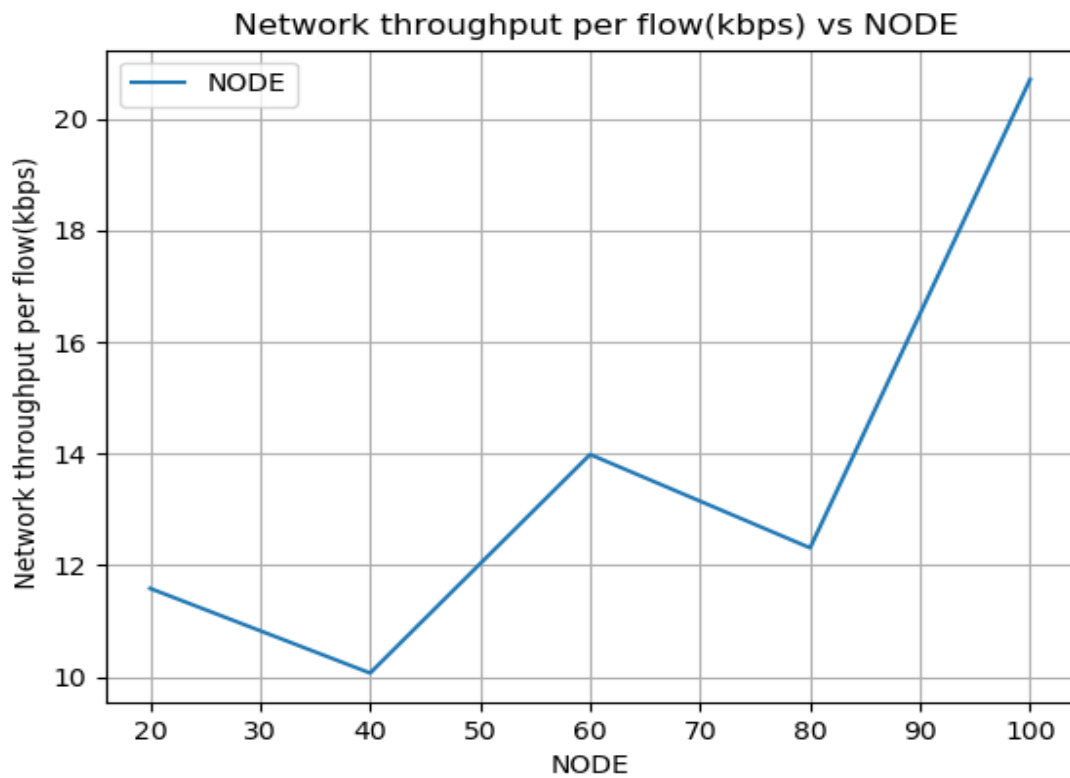
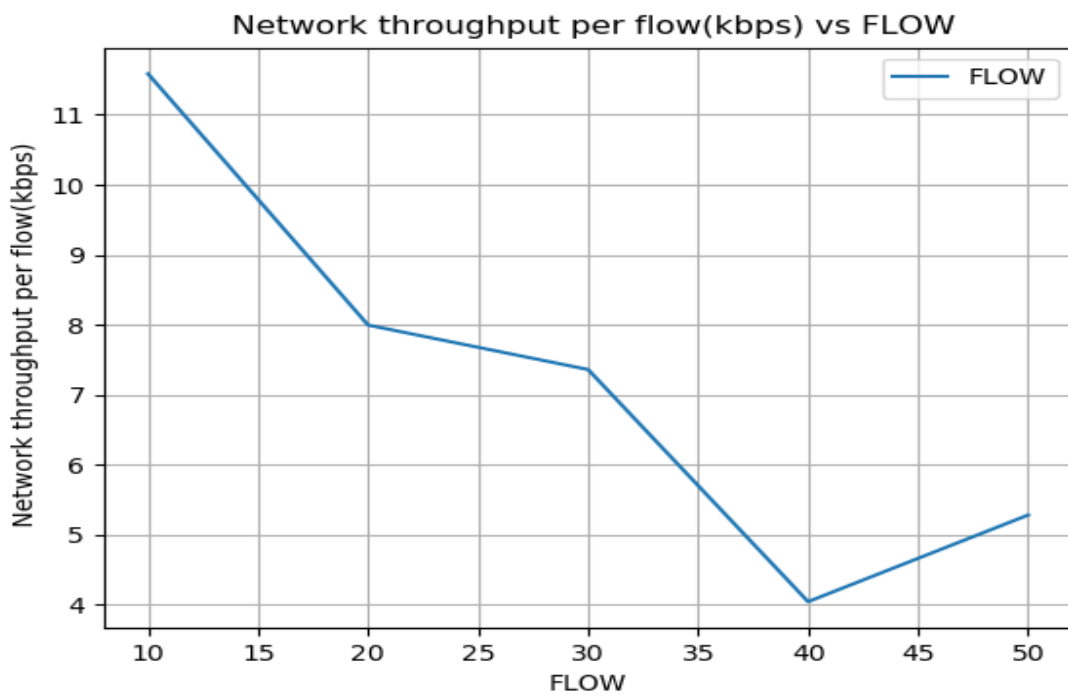
Results of Task A (Wifi Low Rate - Mobile):

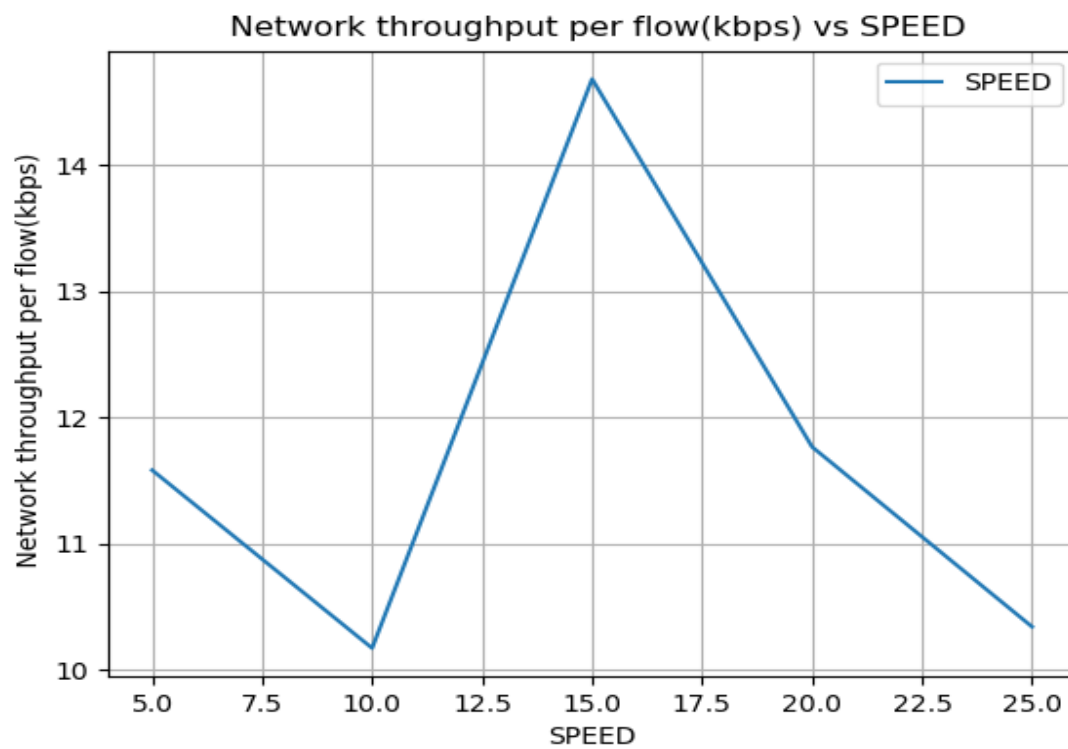
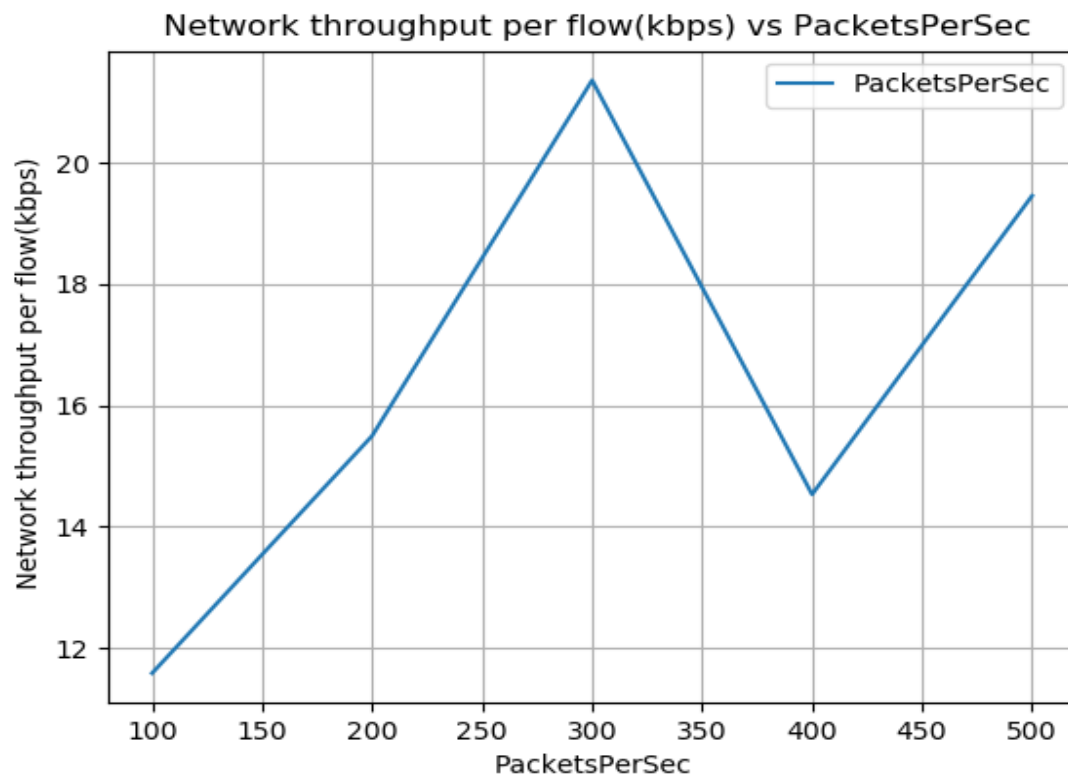
Network Throughput:



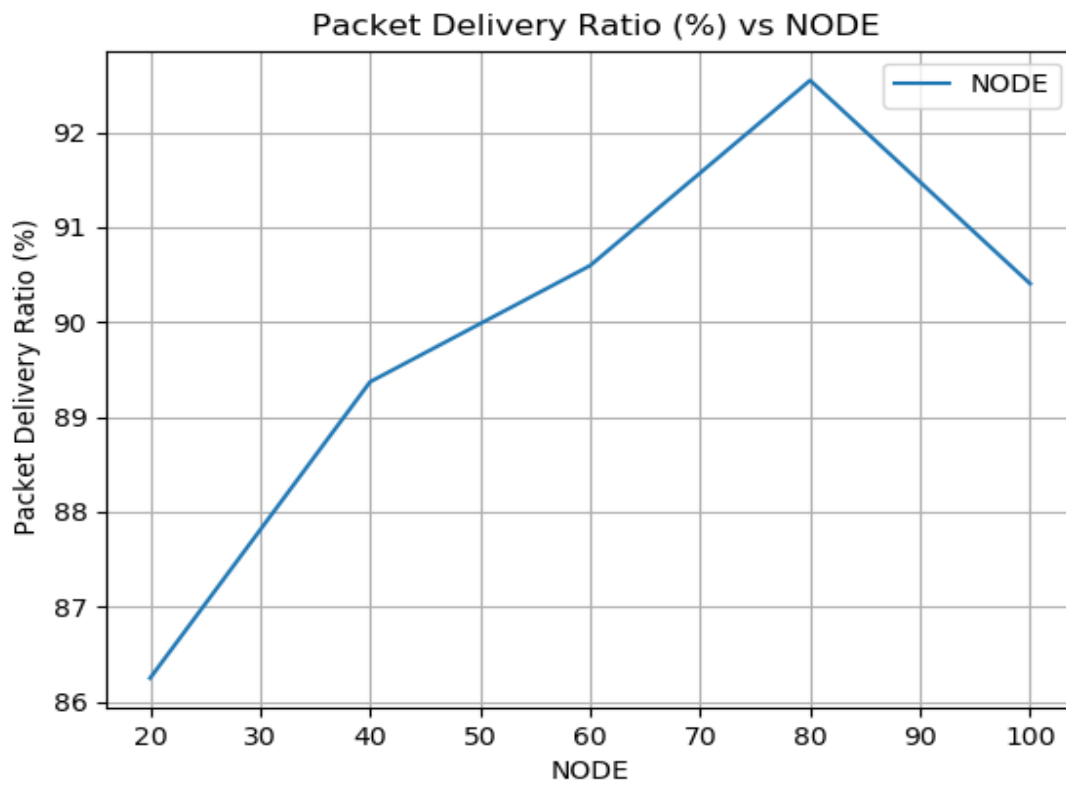
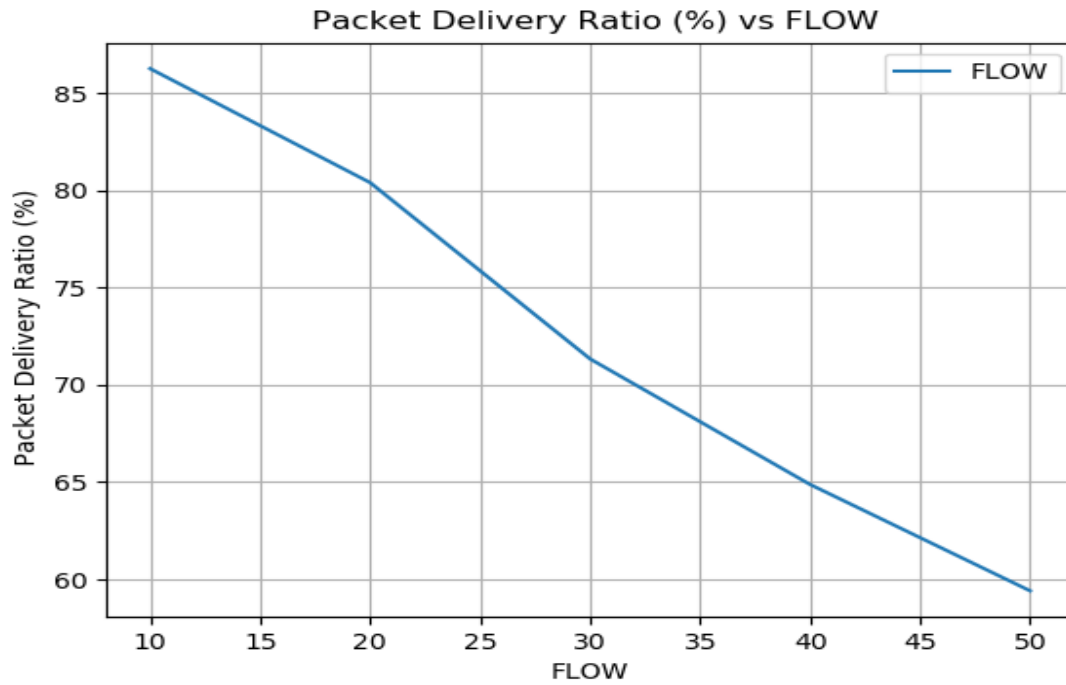


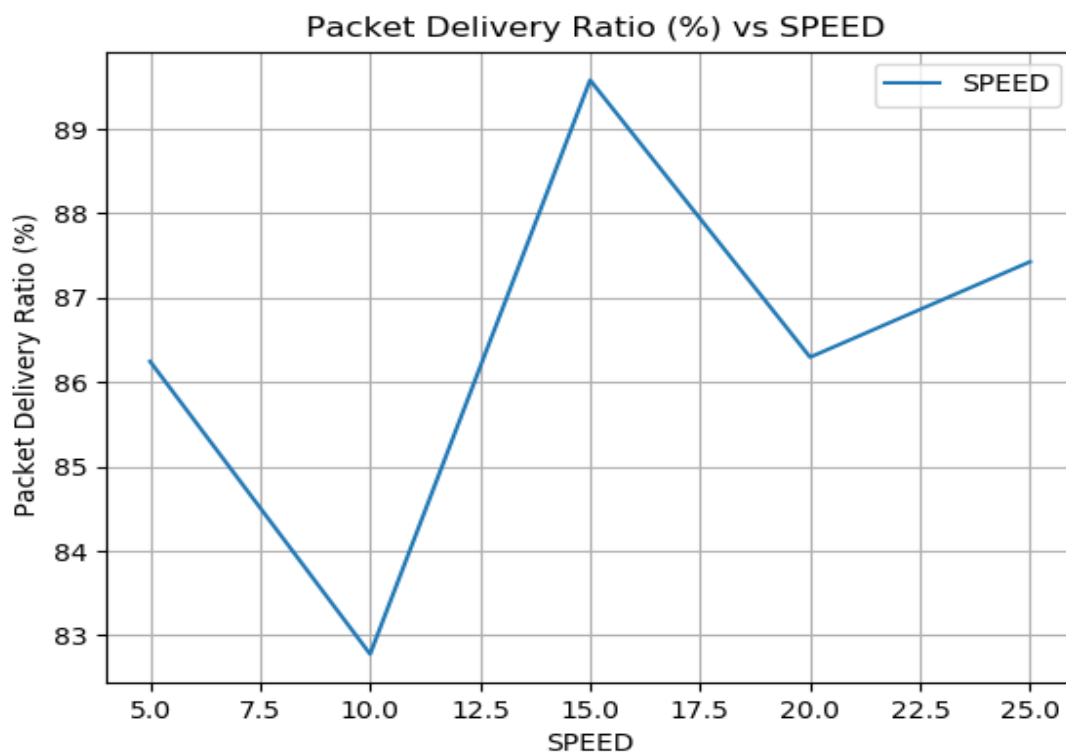
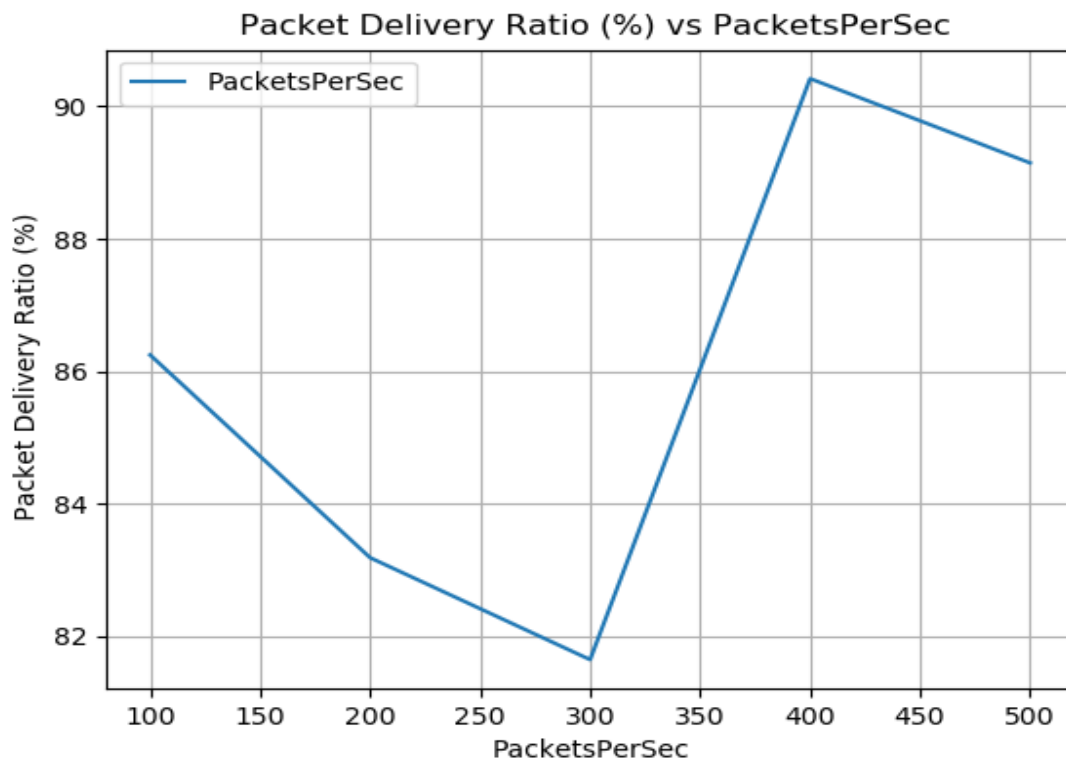
Throughput Per Flow:



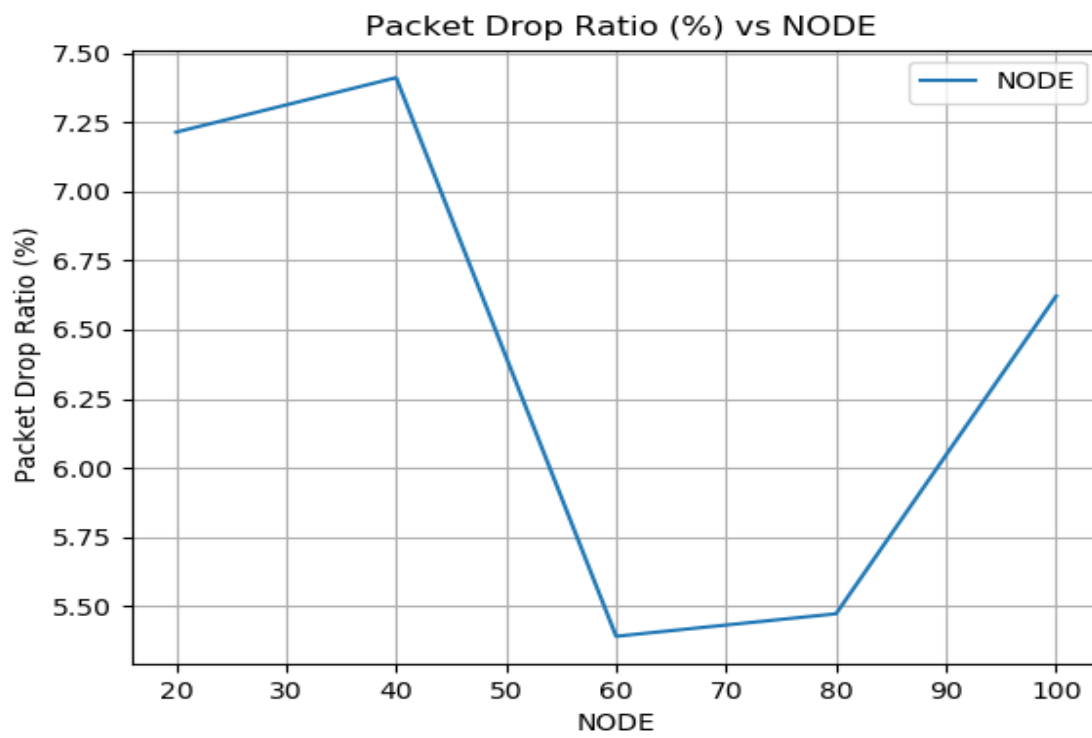
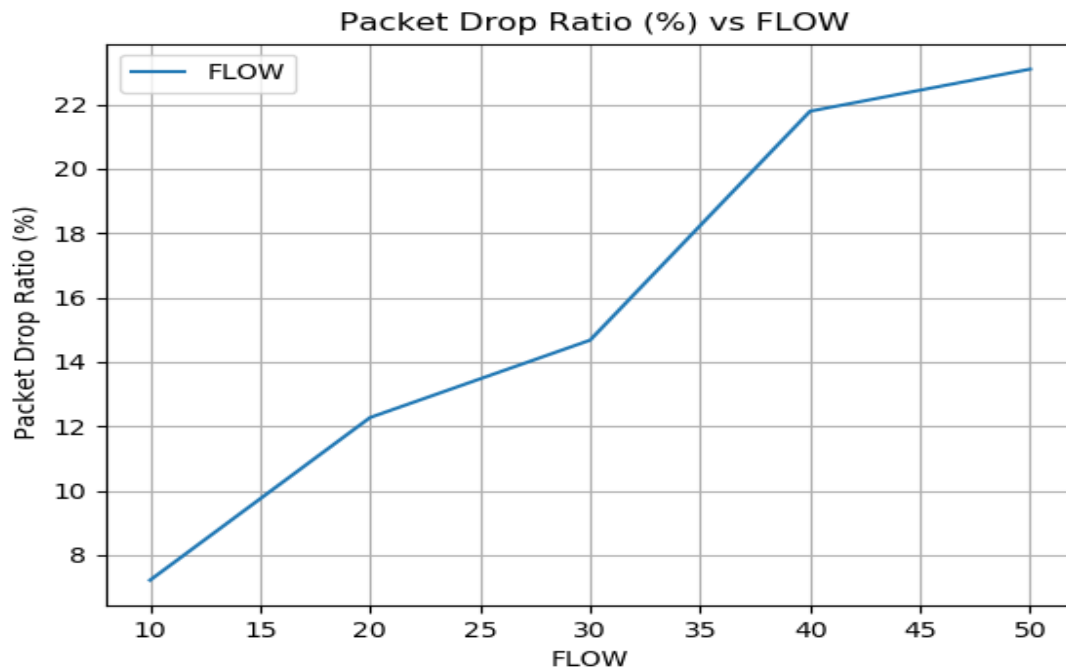


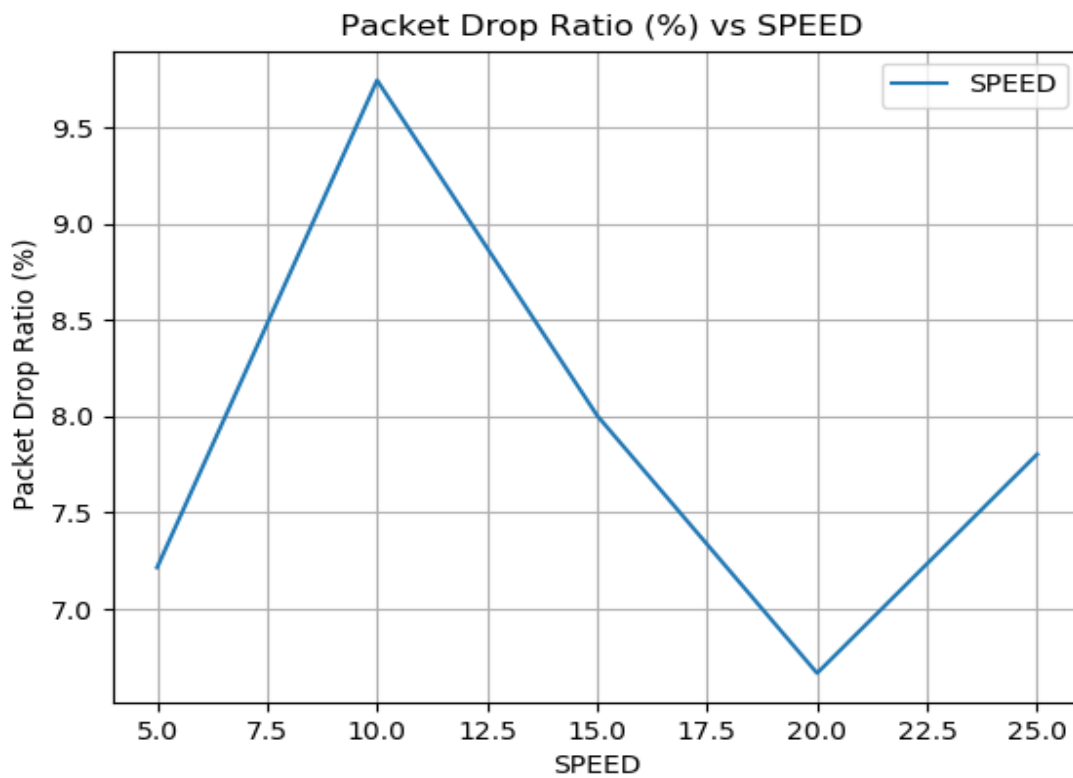
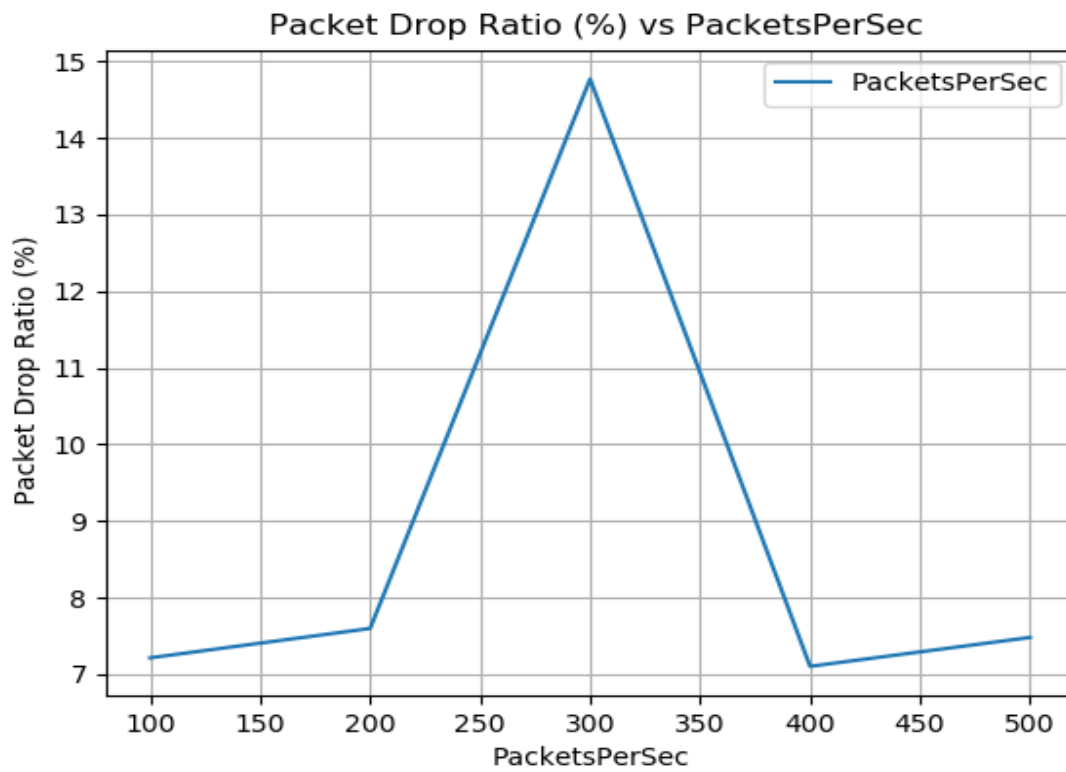
Packet Delivery Ratio:



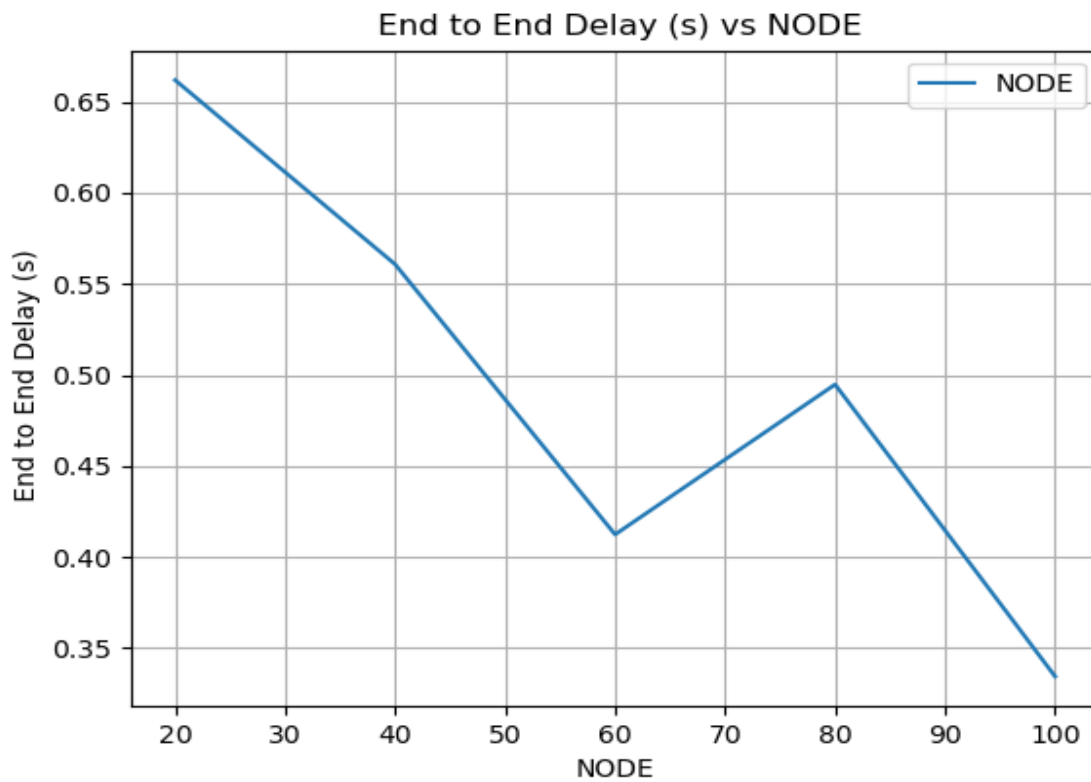
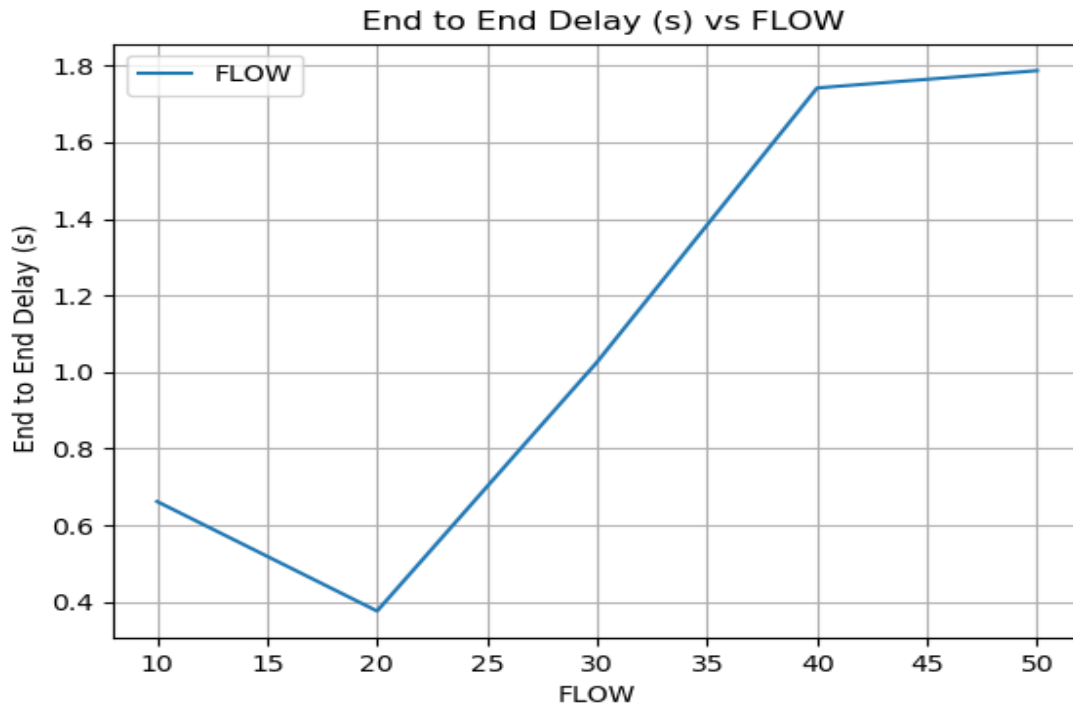


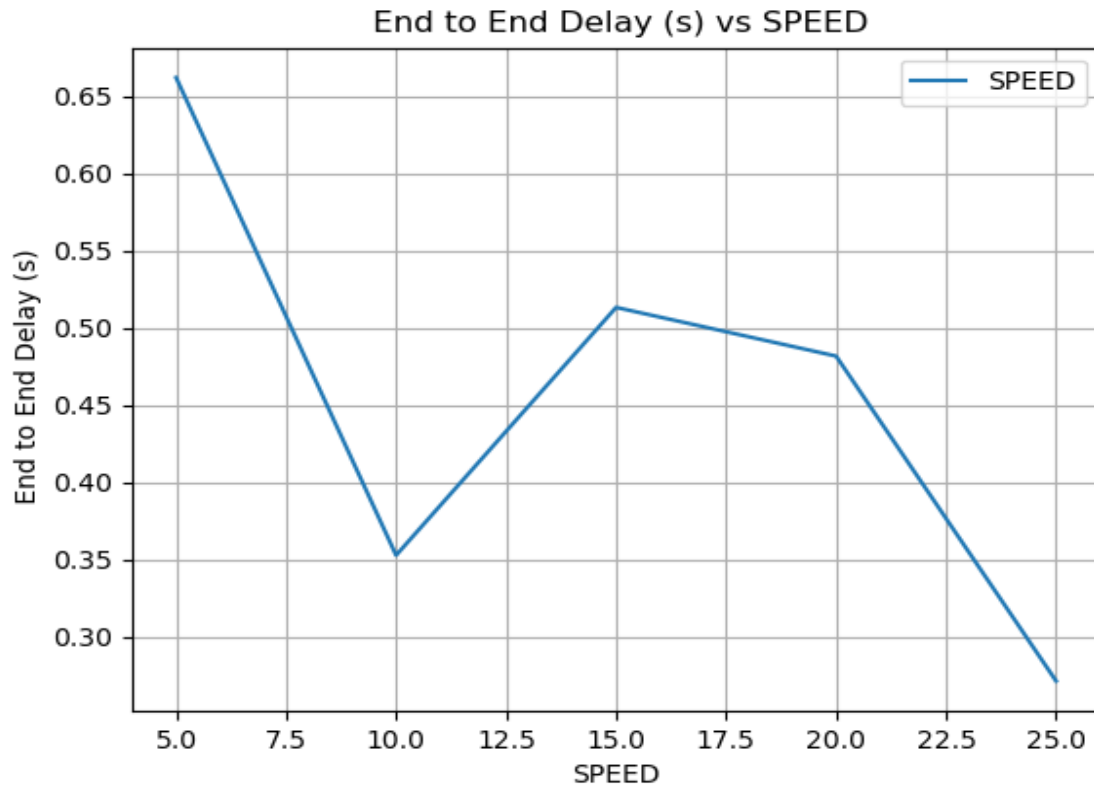
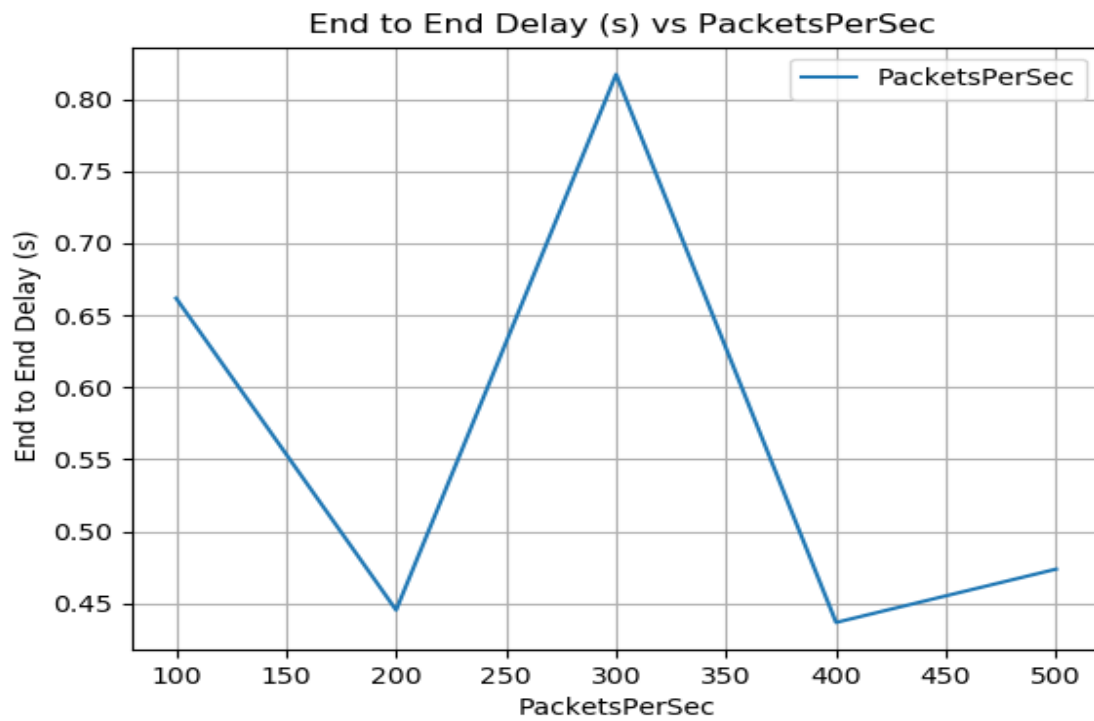
Packet Drop Ratio:





End to End Delay:





Explanation:

From all the graphs we can see, when we are varying the number of flows, then only we are seeing some logical changes in the graphs of throughput, end to end delay, packet drop ratio and packet delivery ratio. But when we are keeping the number of flows fixed and varying other parameters, the variation of the parameters doesn't affect the outputs much.

As we increase the number of flows and the number of packets per second in the network, the network throughput increases.

As we increase the number of flows, the bandwidth is distributed among the flows. That's why we can see throughput decreasing as we increase the number of flows.

As we increase the number of flows in the network, there will be more congestion in the network. That's why the packet delivery ratio is decreasing and the packet drop ratio and end to end delay are increasing when we increase the number of flows.