Network Simulation Project Report

Md. Asif Haider

ID: 1805112

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

February 28, 2023

Contents

1 Introduction

In this network simulation and modification project, we were asked to perform a simulation on each of the two assigned network categories. The network simulator used for the simulations is ns-2.35. As specified by the network assignment for student ID 1805112, we worked on both wired and wireless (mobile) topologies. And then we were asked to modify the existing C++ codebase inside and obtain the effects of changes due to it.

The following report is presented as follows: first we show the required network simulation and calculate different metrics based on the parameter changes. In the later segment, we show the modification details and the effects of those changes visually in a comparative manner.

2 Wired Network Simulation

The network topology, parameters under variation, and metrics determined with necessary graphs are shown below.

2.1 Topology

We used a basic dumbbell topology with various counts of nodes. Two of the nodes marked 0 and n-1 (where n is the total number of nodes) are placed as the core nodes at two ends. Nodes marked from 1 to $\frac{n}{2} - 1$ are connected with the 0th node, while the rest of the nodes are connected with the last (the other core node). The link between the two core nodes is known commonly as the bottleneck link of such topology. Nodes of this topology are connected with wires, and hence static or fixed in their location.

We used TCP agents as the transport layer protocol and FTP application on top of it.

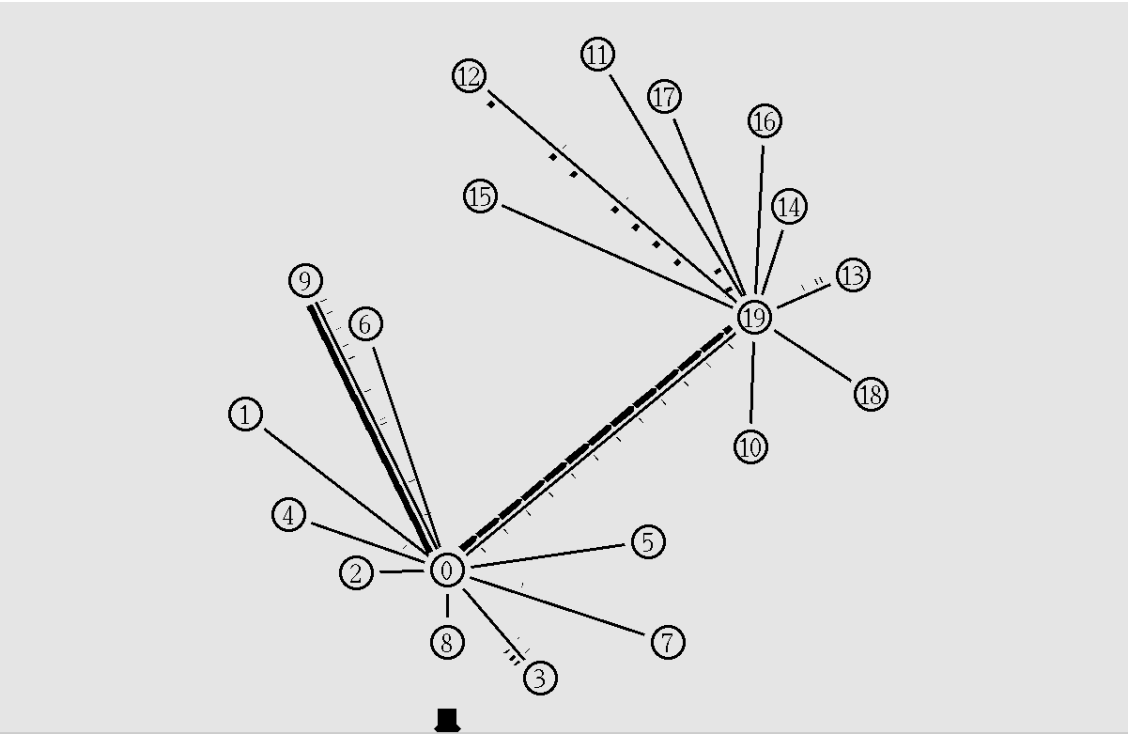


Figure 1: Dumbbell wired topology

2.2 Parameters

3 parameters were considered under variation for this simulation:

- **Number of Nodes:** 20, 40, 60, 80, and 100
- **Number of Flows:** 10, 20, 30, 40, and 50
- **Number of Packets per second:** 100, 200, 300, 400, and 500

As the baseline values of the parameters, we chose 40, 20, and 200 respectively.

2.3 Metrics

For each of the varying parameters, we were asked to plot 4 graphs showing the change of 4 different metrics listed below:

- **Network throughput**
- **End-to-end delay**
- **Packet delivery ratio** (total no. of packets delivered to end destination / total no. of packets sent)
- **Packet drop ratio** (total no. of packets dropped / total no. of packets sent)

The throughput is the measure of how fast we can actually send data through the network. It is the measurement of the number of packets that are transmitted through the network in a unit of time. It is desirable to have a network with high throughput.

End-to-end delay is the average time delay consumed by data packets to propagate from source to destination. This delay includes the total time of transmission i.e. propagation time, queuing time, route establishment time, etc. A network with minimum average end-to-end delay offers better speed of communication.

The packet delivery ratio is the ratio of the number of packets received at the destination to the number of packets generated at the source. A network should work to attain a high delivery ratio in order to have a better performance, which shows the amount of reliability offered by the network.

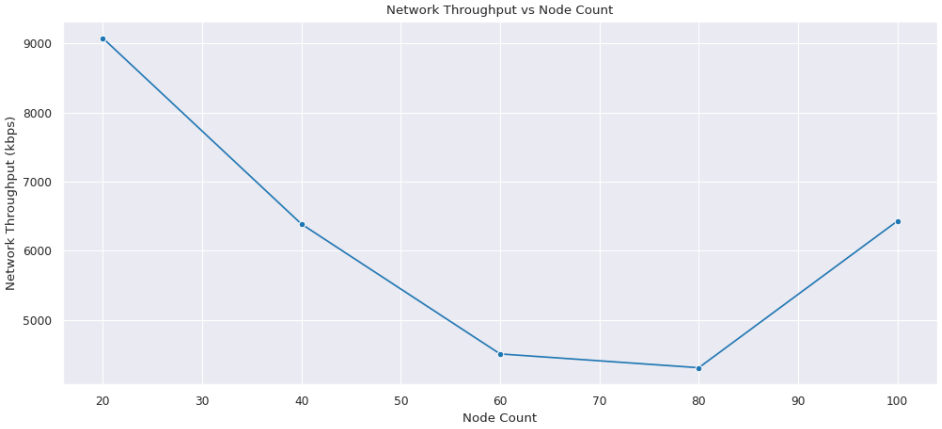
The packet drop ratio is just the complement of the delivery ratio. The lower the drop rate, the better the network stack.

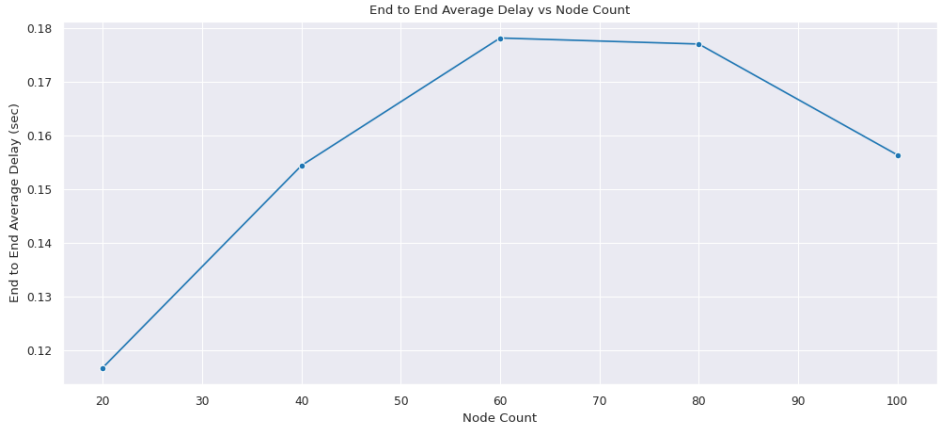
2.4 Result

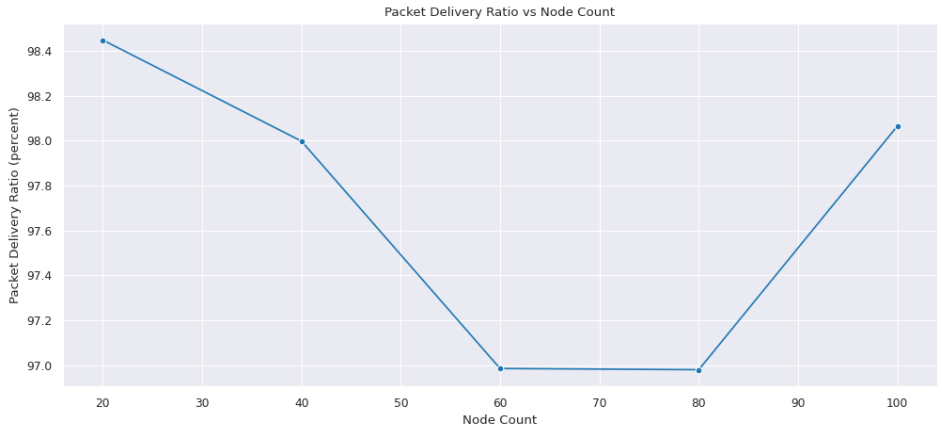
As a combination of 3 parameters and 4 metrics, we produced 12 graphs in total.

2.4.1 Node Count

We experimented with various node counts among 20, 40, 60, 80, and 100.


Network Throughput vs Node Count
Network Throughput (kbps)
9000
8000
7000
6000
5000
20
30
40
50
60
70
80
90
100
Node Count


End to End Average Delay vs Node Count
End to End Average Delay (sec)
0.18
0.17
0.16
0.15
0.14
0.13
0.12
20
30
40
50
60
70
80
90
100
Node Count


Packet Delivery Ratio vs Node Count
Packet Delivery Ratio (percent)
98.4
98.2
98.0
97.8
97.6
97.4
97.2
97.0
20
30
40
50
60
70
80
90
100
Node Count

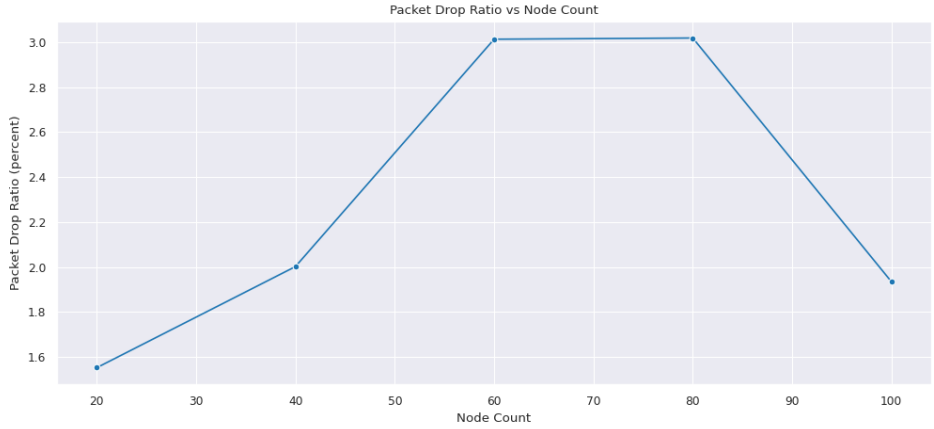
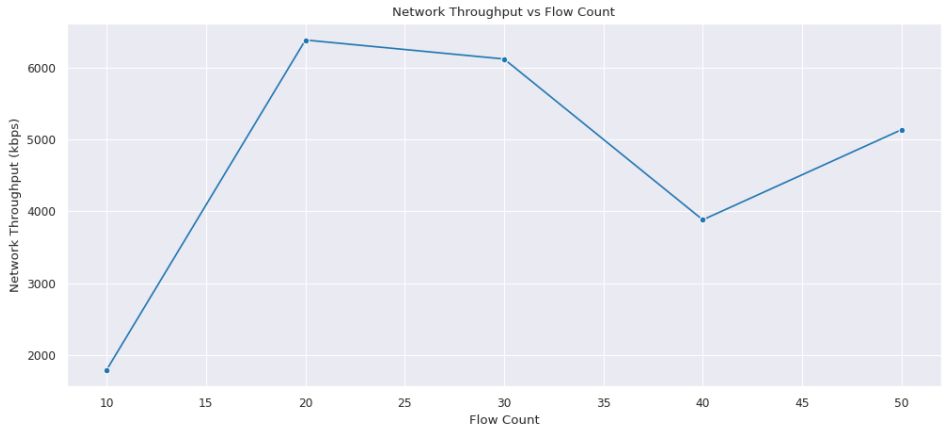
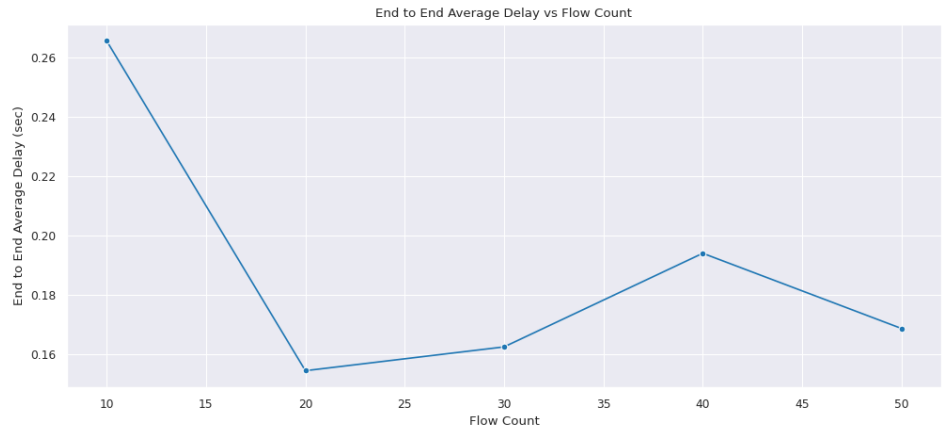


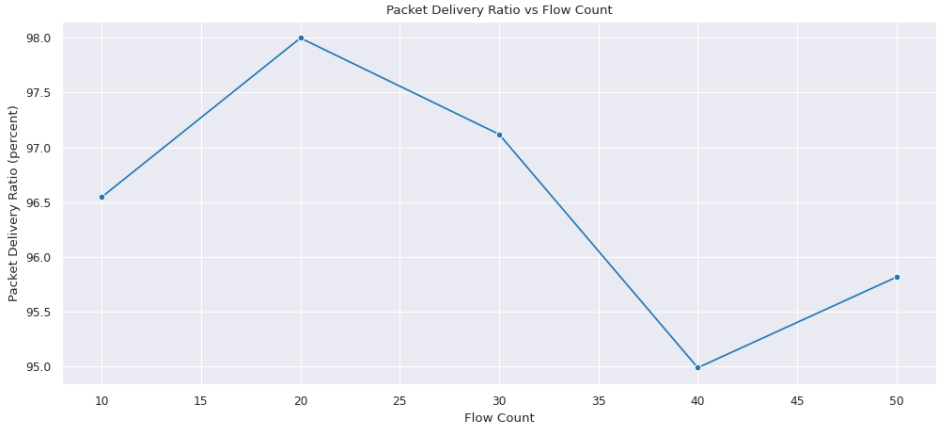
Figure 2: Varying number of nodes

2.4.2 Flow Count

We plotted various numbers of flows counting from 10, 20, 30, 40, and 50.







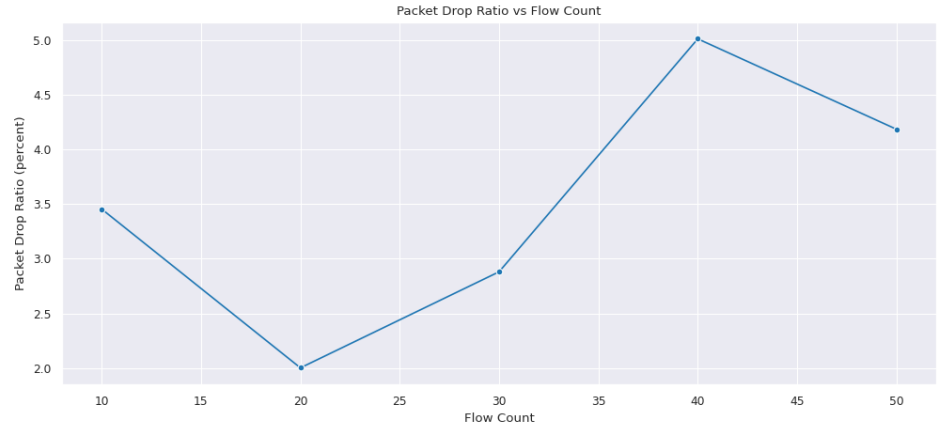
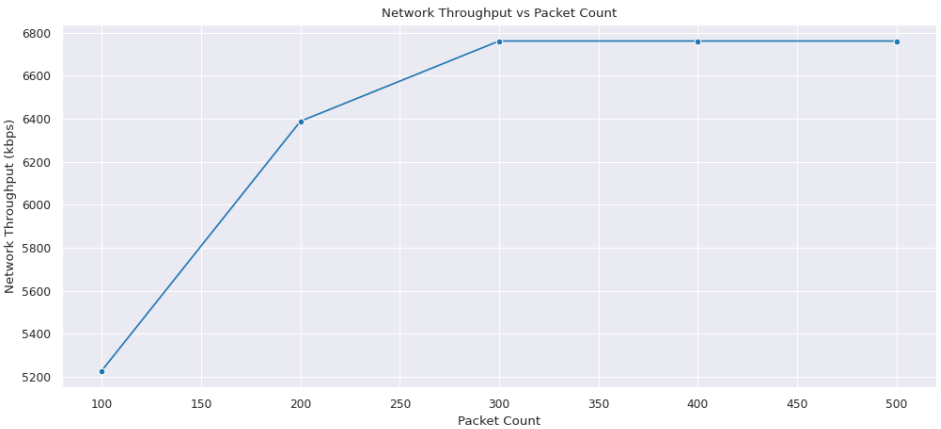
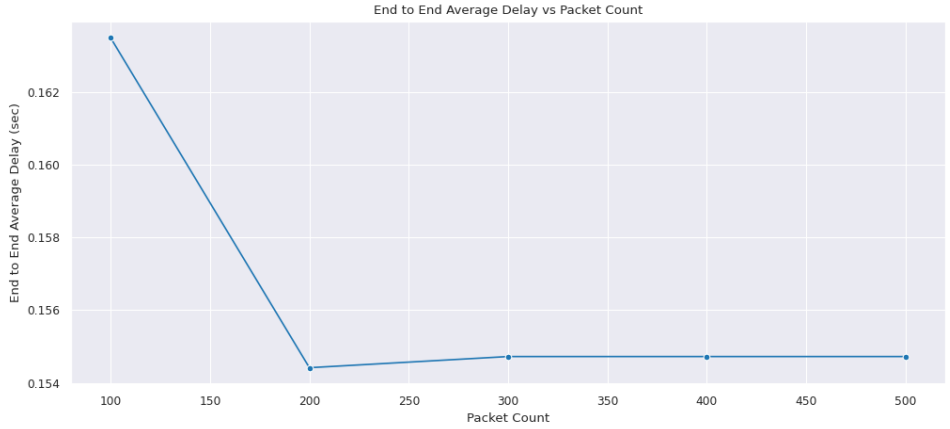


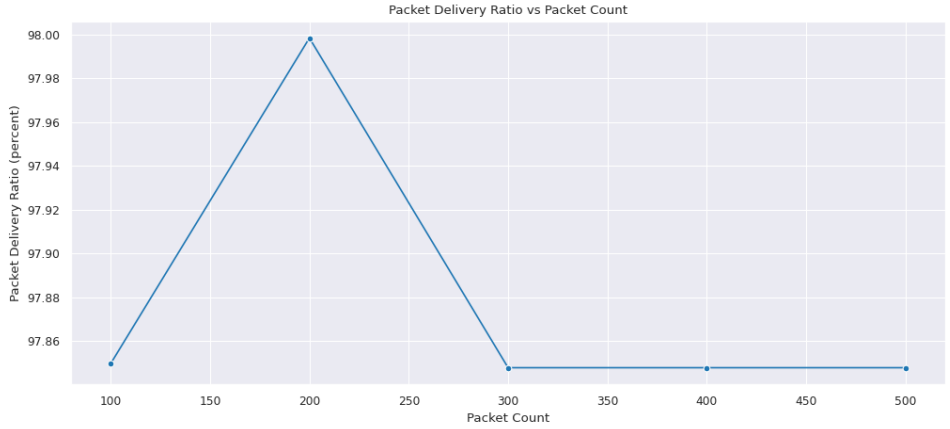
Figure 3: Varying number of flows

2.4.3 Packet Rate

We varied the packet count per second in the range of 100, 200, 300, 400, and 500.







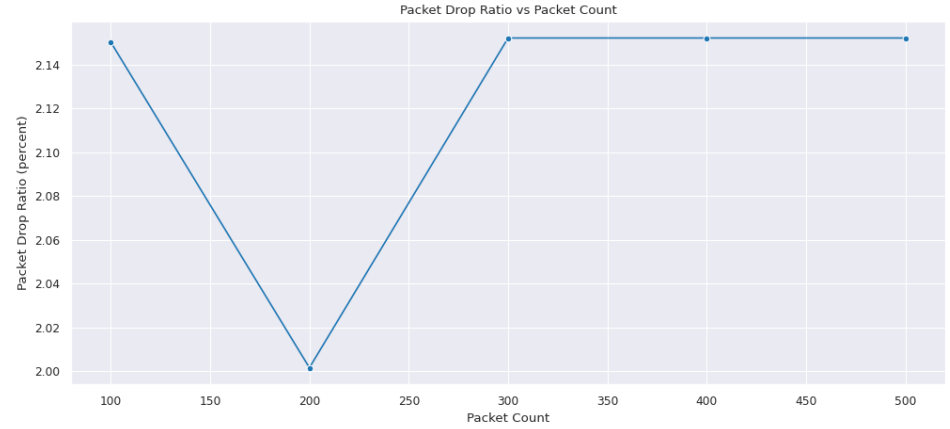


Figure 4: Varying number of packets per second

2.5 Findings

As can be seen from the graphs, the metrics we used here usually follow a particular pattern with respect to varying node counts and flow counts. But interestingly, once it reaches a peak (maxima or minima), it changes direction. Taking multiple plots we found that the data points in the graph keep fluctuating.

As the packet rate increase, the network throughput increases, and the delay decreases as expected. But the packet delivery and drop ratios show quite peculiar graphs.

3 Wireless Network Simulation

The network topology, parameters under variation, and metrics determined with necessary graphs are shown below.

3.1 Topology

The network topology attributes for the wireless mobile node simulation are listed below:

- **Wireless MAC Type:** Wireless 802.11 protocol
- **Routing Protocol:** AODV
- **Agent:** TCP
- **Application:** FTP traffic
- **Node Positioning:** Random
- **Flow Type:** 1 random source, multiple random sink (except source itself)
- **Interface Queue:** Droptail, Priority Queue, Maximum size 50 packets
- **Antenna:** Omni-directional, unity gain
- **Radio Propagation Model:** Two ray ground
- **Energy Model:** Default energy model

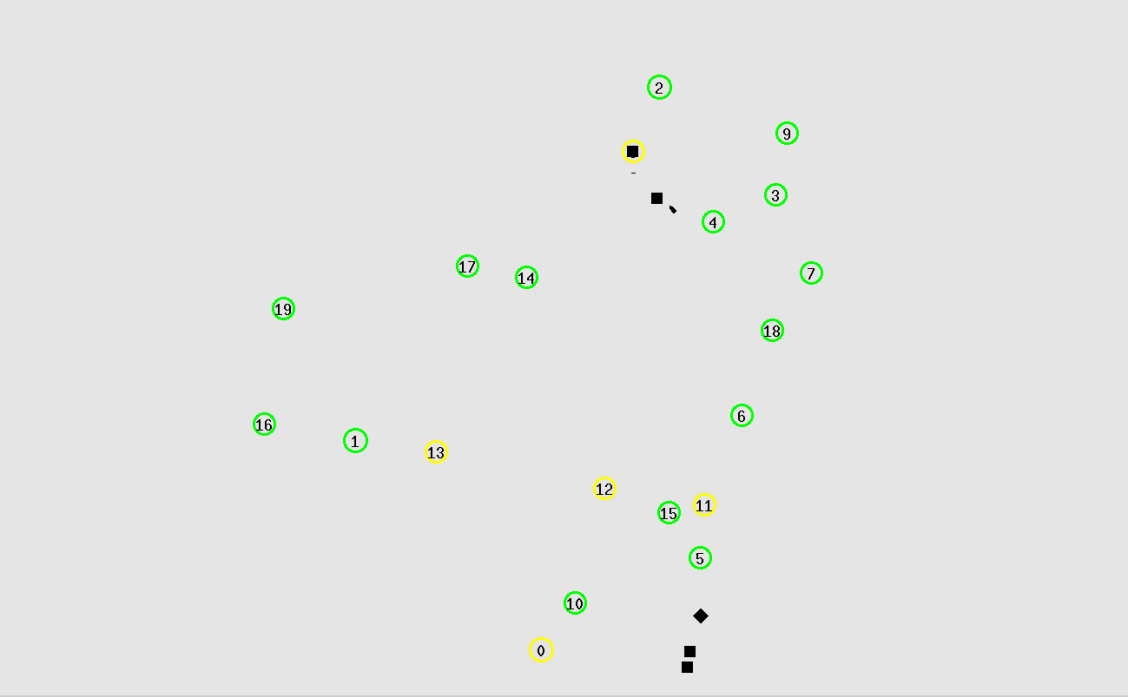


Figure 5: Wireless MANET topology

IEEE 802.11 is a set of **Physical and Data Link layer** standards for **W**ireless **L**ocal **A**rea **N**etwork **(WLAN)** communication. It is commonly used to provide wireless connectivity in homes, offices, and public spaces. There are several different versions of 802.11 providing different levels of performance and security.

These networks can be used in two modes. The **Infrastructure** mode consists of client devices associated with an **Access point (AP)**, which in turn, is connected with other networks. The clients send and receive data packets via the AP.

The other mode is known as the **Ad Hoc** network, which is a collection of computers associated with each other to directly send and receive frames. There is no separate access point here. We are concerned with this ad hoc type of network in this simulation.

Network of nodes staying near each other are known as **M**obile **A**d hoc **Net**works **(MANETs)**. As opposed to wired networks, the topology in ad hoc networks might be changing all the time, making routing in these networks more challenging. We used MANET for this simulation.

3.2 Parameters

4 parameters were considered under variation for this simulation:

- **Number of Nodes:** 20, 40, 60, 80, and 100
- **Number of Flows:** 10, 20, 30, 40, and 50
- **Number of Packets per second:** 100, 200, 300, 400, and 500
- **Speed of Nodes:** 5, 10, 15, 20 and 25 m/s

As the baseline values of the parameters, we chose 40, 20, 200, and 10 respectively.

3.3 Metrics

For each of the varying parameters, we were asked to plot 5 graphs showing the change of 5 different metrics listed below:

- **Network throughput**
- **End-to-end delay**
- **Packet delivery ratio** (total no. of packets delivered to end destination / total no. of packets sent)
- **Packet drop ratio** (total no. of packets dropped / total no. of packets sent)
- **Energy Consumption**

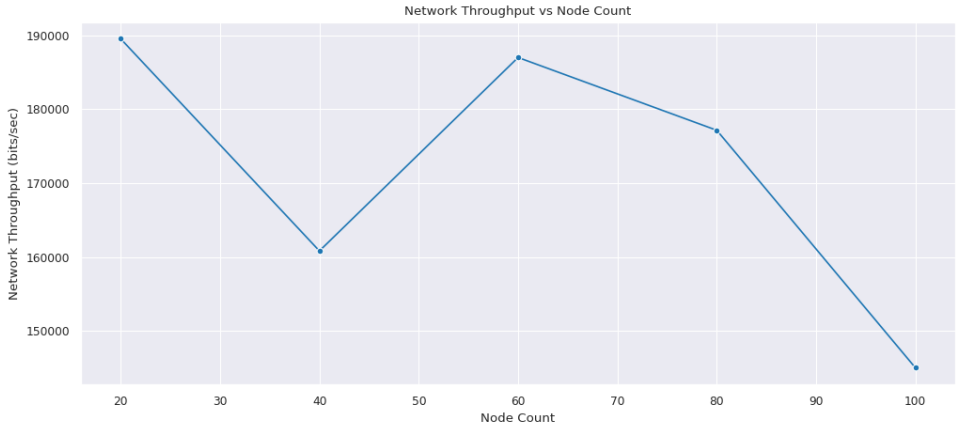
We were asked to calculate energy consumption also specifically for wireless nodes. Energy is consumed in the wireless network during transmit, receive, and idle modes. It is the amount of energy consumed during the packet transmission by each node and calculates the overall energy of the whole network. The energy model in ns-2 represents the level of energy in a mobile host. The energy model in a node has an initial value which is the level of energy the node has at the beginning of the simulation.

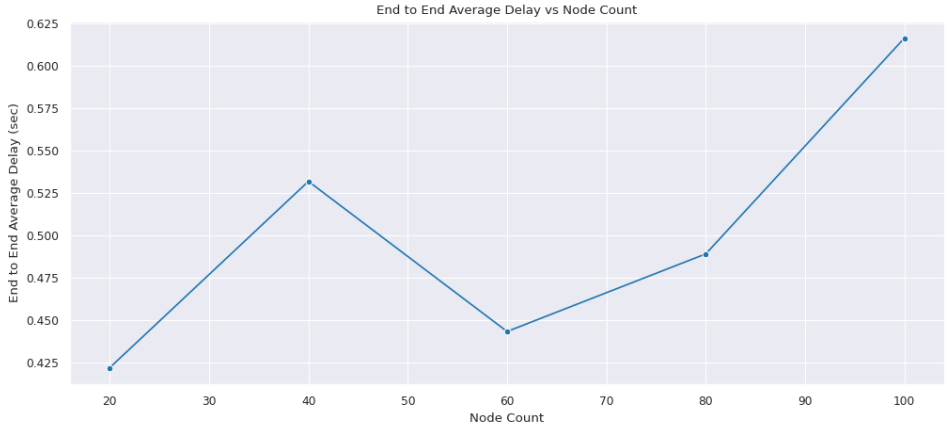
3.4 Result

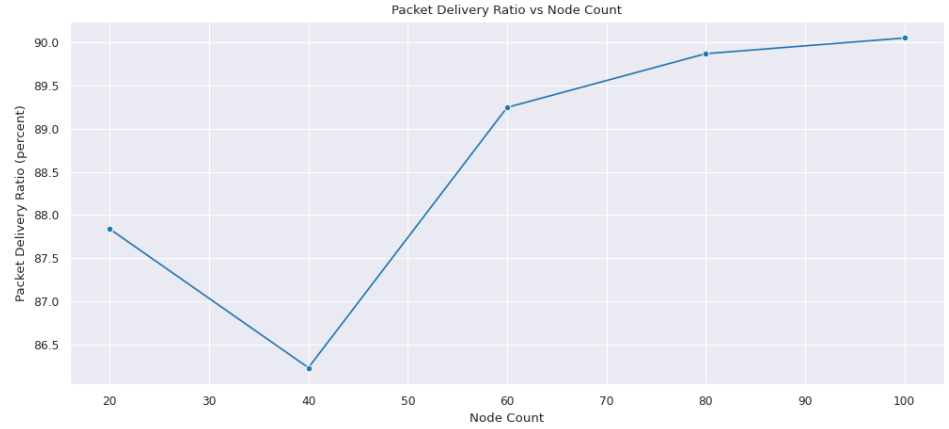
As a combination of 4 parameters and 5 metrics now, we produced 20 graphs in total.

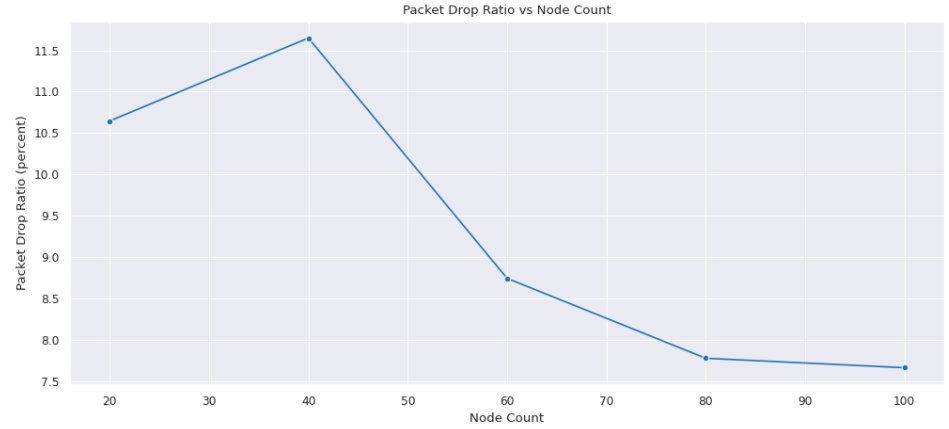
3.4.1 Node Count

We experimented with various node counts among 20, 40, 60, 80, and 100.




End to End Average Delay vs Node Count
End to End Average Delay (sec)
0.625
0.600
0.575
0.550
0.525
0.500
0.475
0.450
0.425
20
30
40
50
60
70
80
90
100
Node Count


Packet Delivery Ratio vs Node Count
Packet Delivery Ratio (percent)
90.0
89.5
89.0
88.5
88.0
87.5
87.0
86.5
20
30
40
50
60
70
80
90
100
Node Count


Packet Drop Ratio vs Node Count
Packet Drop Ratio (percent)
11.5
11.0
10.5
10.0
9.5
9.0
8.5
8.0
7.5
20
30
40
50
60
70
80
90
100
Node Count

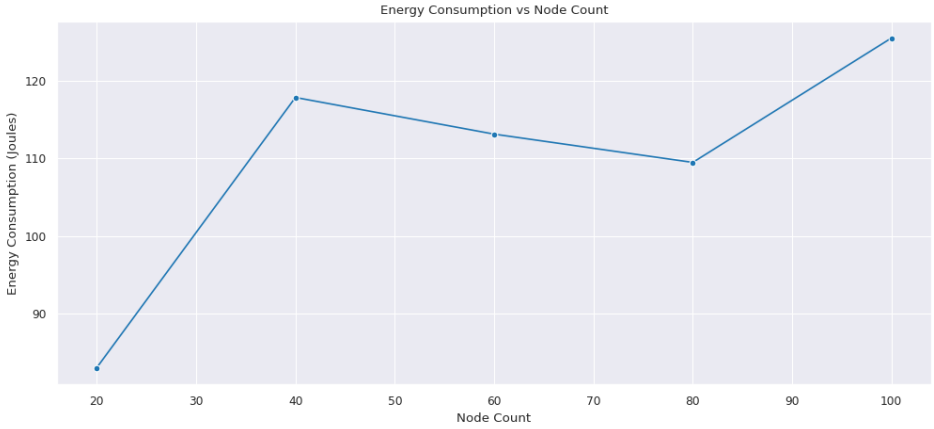
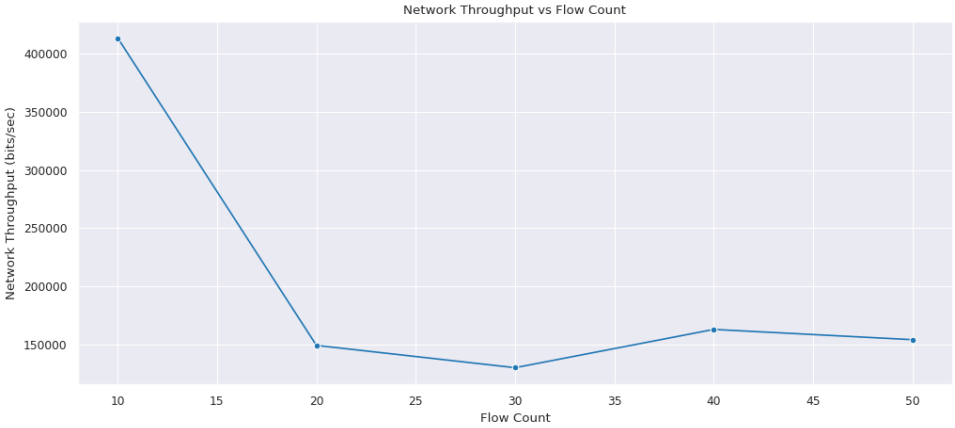
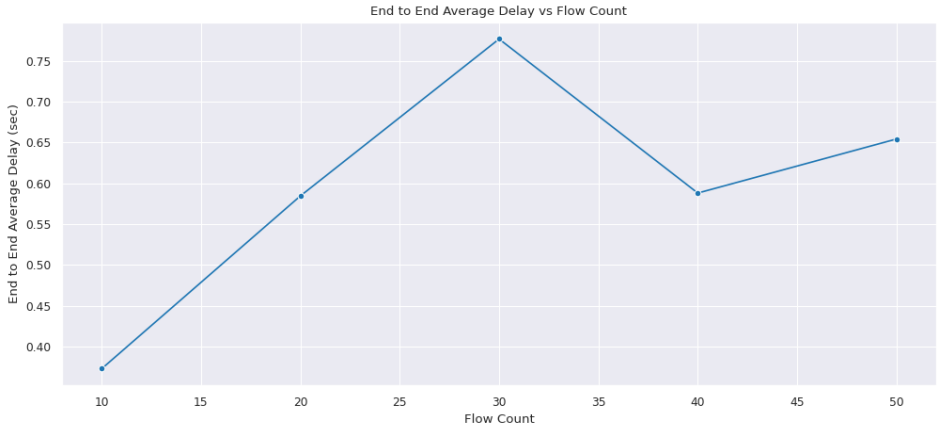


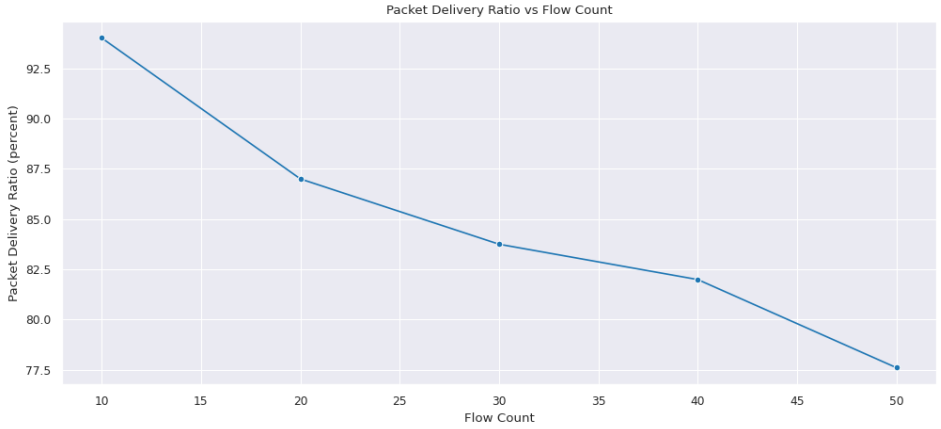
Figure 6: Varying number of nodes

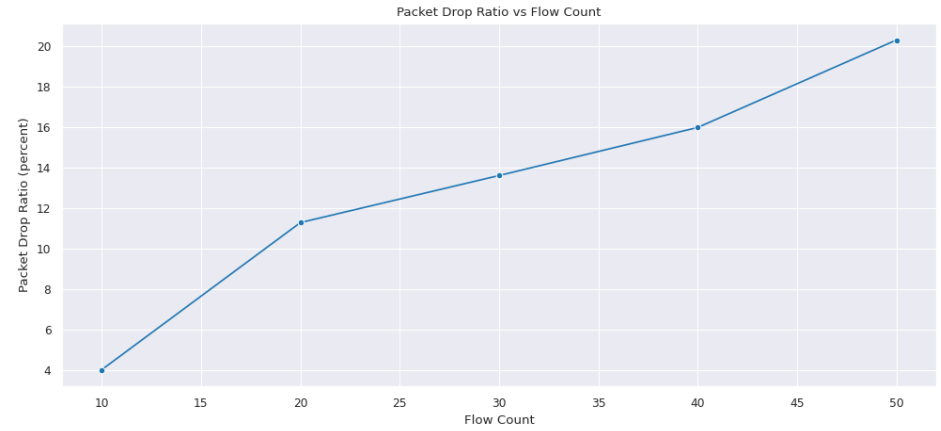
3.4.2 Flow Count

We plotted various numbers of flows counting from 10, 20, 30, 40, and 50.









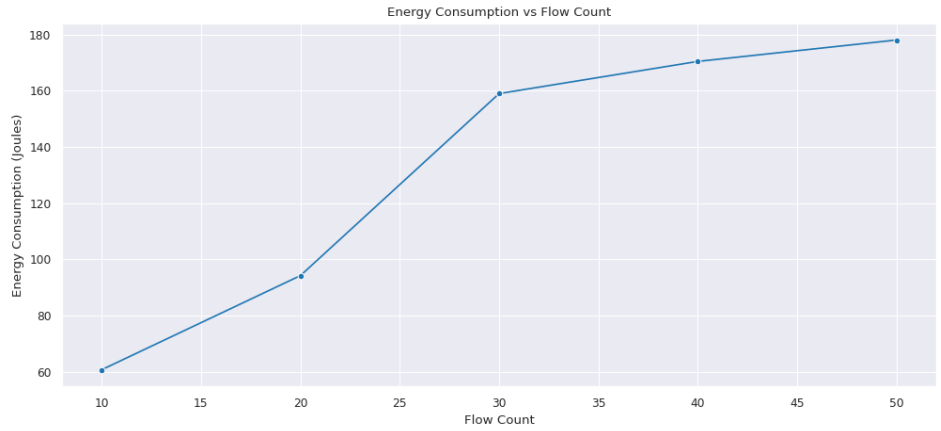
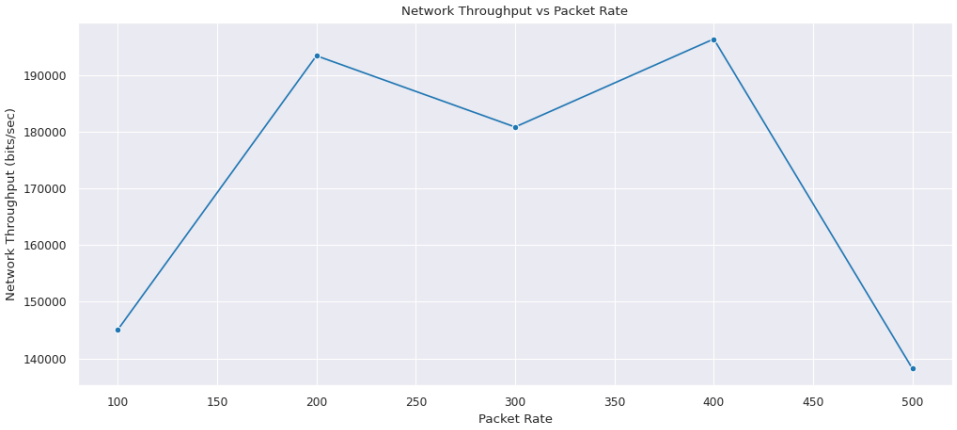
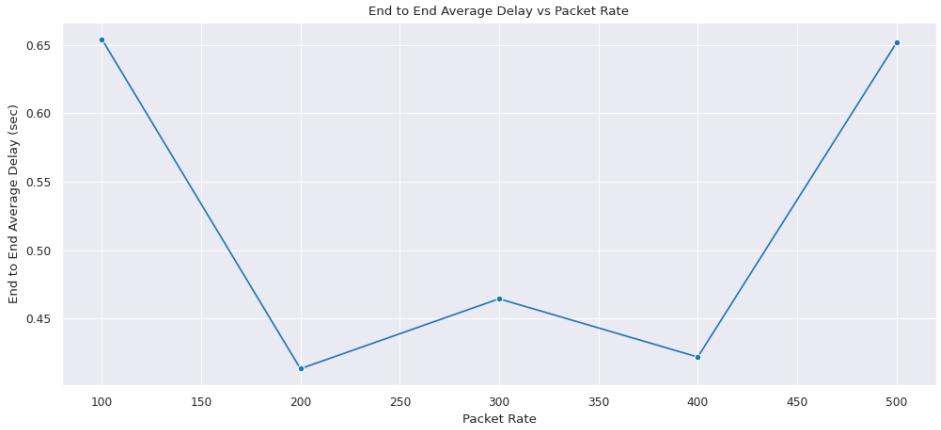


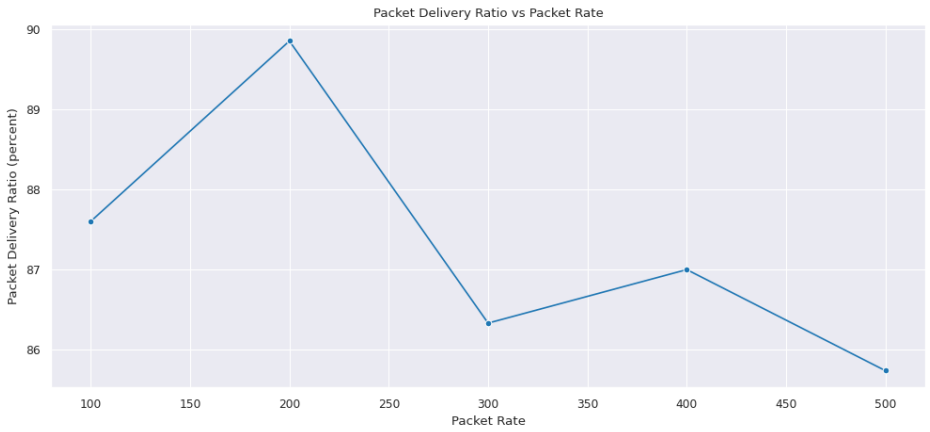
Figure 7: Varying number of flows

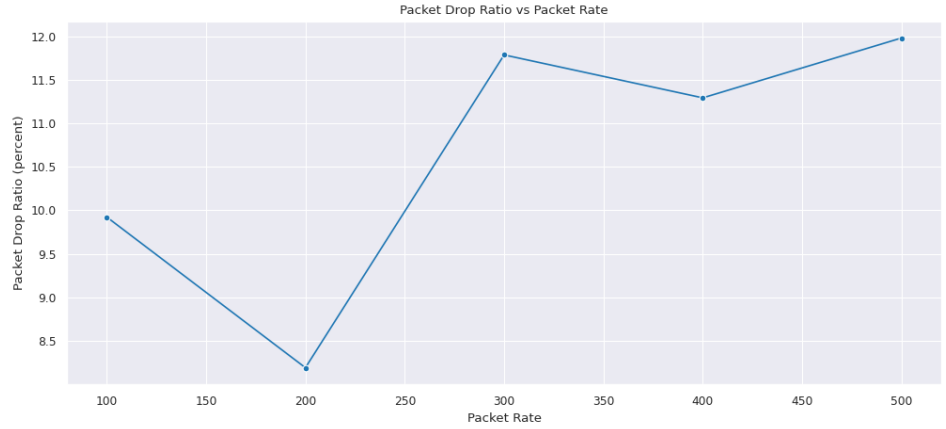
3.4.3 Packet Rate

We varied the packet count per second in the range of 100, 200, 300, 400, and 500.









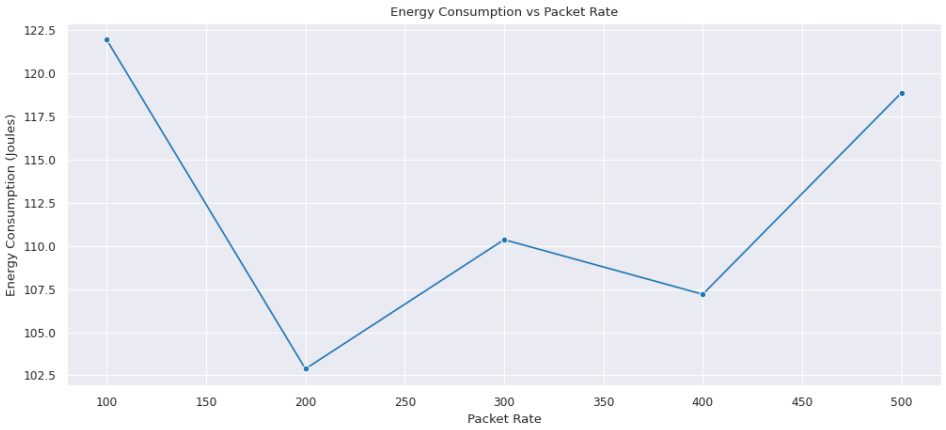
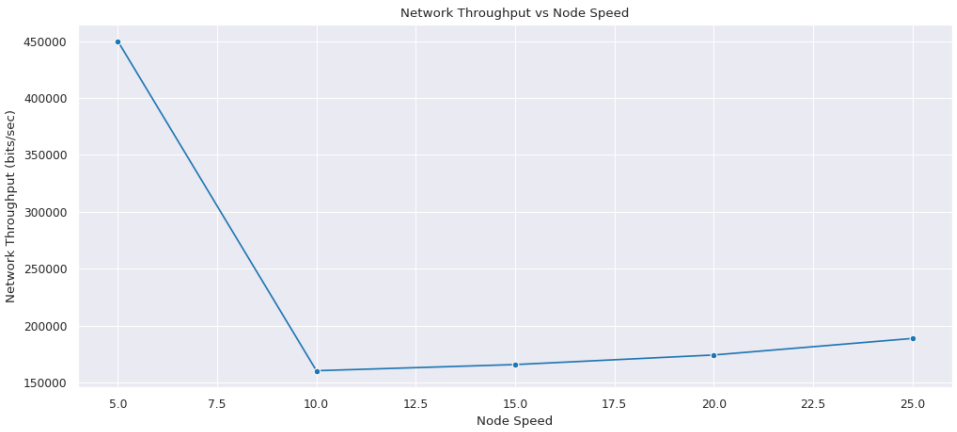
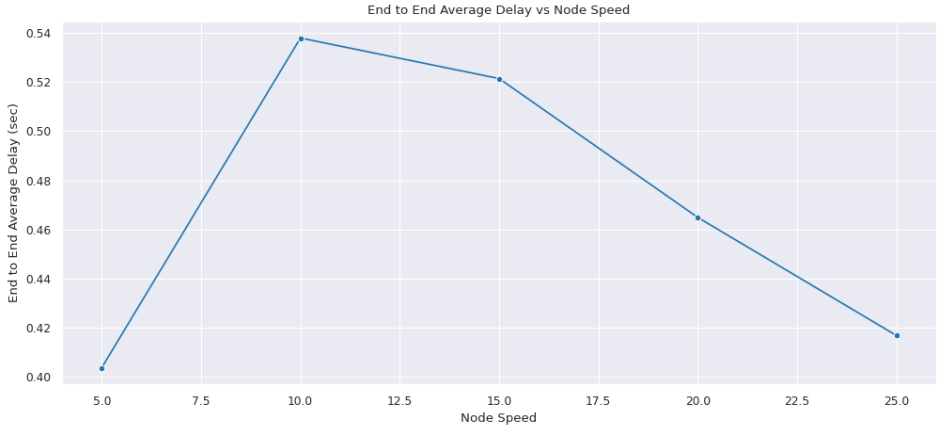


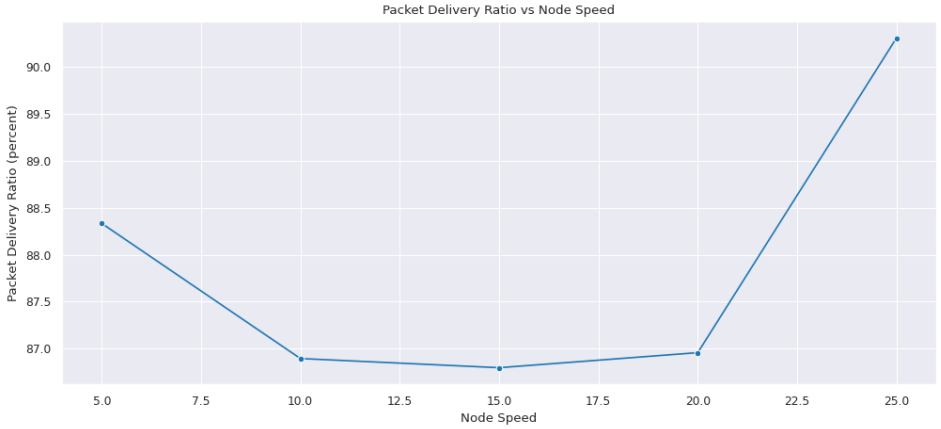
Figure 8: Varying number of packets per second

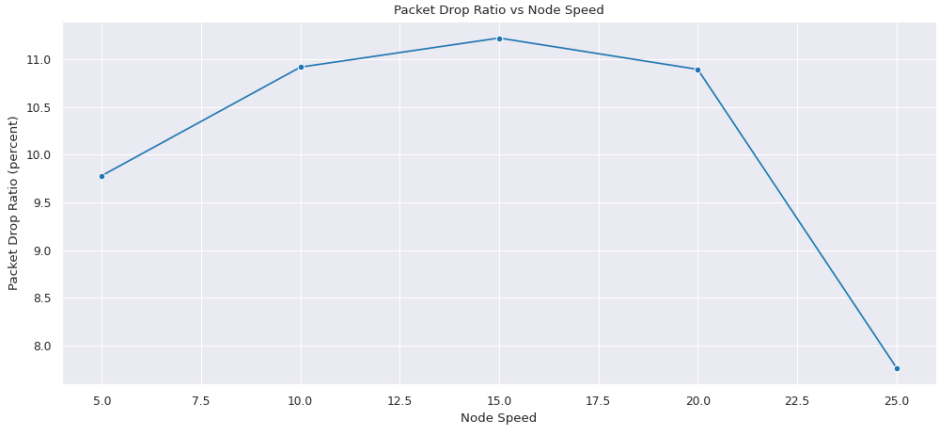
3.4.4 Node Speed

We varied the speed of mobile nodes in the range of 5, 10, 15, 20, and 25 m/s.




End to End Average Delay vs Node Speed
End to End Average Delay (sec)
0.54
0.52
0.50
0.48
0.46
0.44
0.42
0.40
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
Node Speed


Packet Delivery Ratio vs Node Speed
Packet Delivery Ratio (percent)
90.0
89.5
89.0
88.5
88.0
87.5
87.0
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
Node Speed


Packet Drop Ratio vs Node Speed
Packet Drop Ratio (percent)
11.0
10.5
10.0
9.5
9.0
8.5
8.0
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
Node Speed

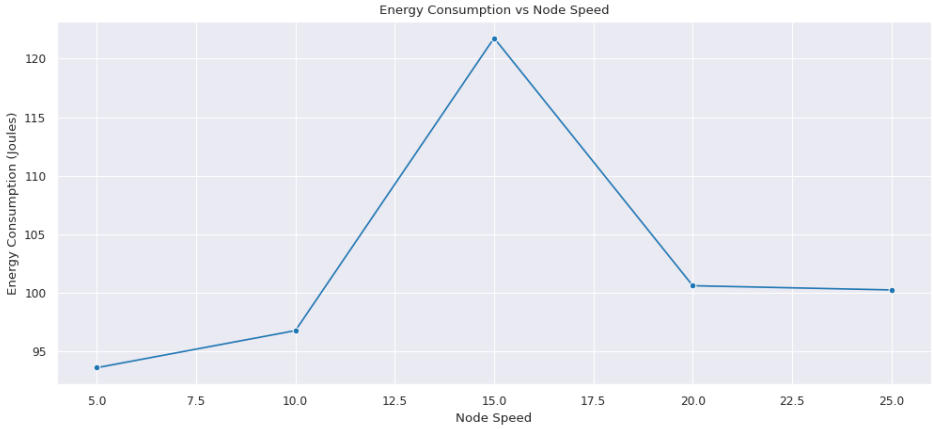


Figure 9: Varying speed of nodes

3.5 Findings

The changes in node count for the wireless topology case don't follow any specific pattern. But the packet delivery and drop ratios maintain a smooth almost linear curves as the flow count is changed. In this case, the throughput decreases and delay is introduced more than before.

The packet transmission rate also does not follow any specific pattern. It seems to be almost randomized. One potential reason behind the unpredictability of packet transmission in ad hoc networks might be the random positioning and random speed of nodes, effectively changing the network structure as the various parameters grow larger.

The network throughput decreases drastically once the node speeds are on an increase.

4 Proposed Modification

In this section, we discuss in detail the modification done in the ns-2.35 C++ codebase to demonstrate the effects of change. We chose to study RTT (Round Trip Time), RTO (Retransmission Time Out) and change the existing mechanism slightly.

4.1 RTT and RTO

RTT stands for **Round Trip Time**, and it refers to the time it takes for a packet to travel from a sender to a receiver and back again. In other words, it's the time it takes for a signal to travel from a source to a destination and back again. RTT is usually measured in milliseconds (ms), and it's an important metric in network performance analysis.

RTO stands for **Retransmission Time Out**, and it refers to the amount of time a sender will wait before retransmitting a packet that has not been acknowledged by the receiver. The RTO value is calculated based on the RTT and other factors, and it's used to determine when a packet has been lost or delayed. When a packet is lost or delayed, the sender will retransmit the packet after waiting for the RTO time period to elapse.

Together, RTT and RTO play an important role in TCP (Transmission Control Protocol) and other network protocols, helping to ensure reliable communication between devices across a network.

4.2 Changes

In this modification attempt, we tried to change the existing logic behind **RTT calculation**, and introduce a simple weighted average approach to update the RTT in every iteration taking into account the weighted RTO values before considering it with smoothed RTT values.

- **Step 1:** Included the necessary header file inside **tcp.cc** to incorporate standard C++ libraries inside the codebase.

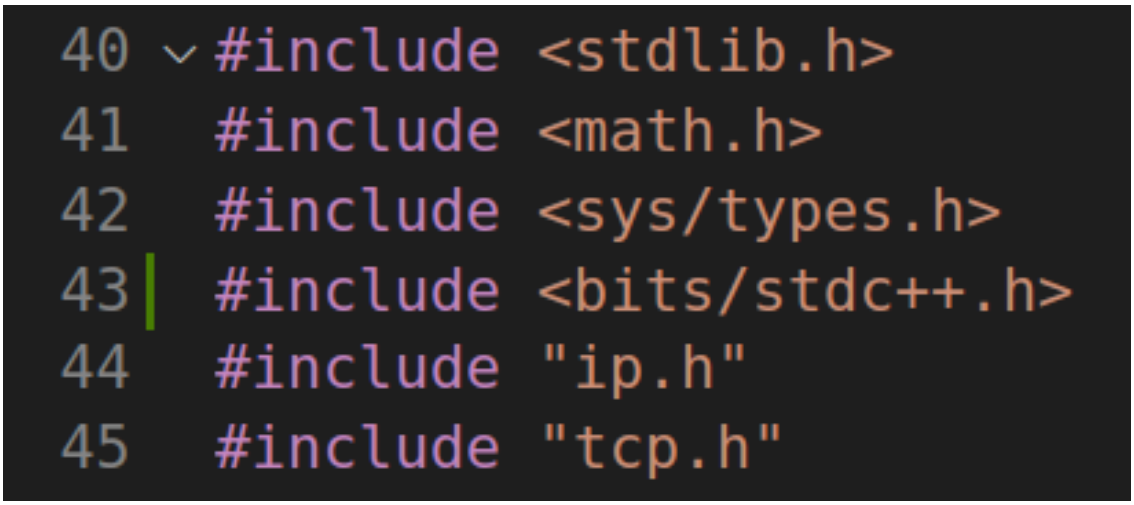


Figure 10: Header file inclusion

- **Step 2:** Written a new function titled **void new_rtt_update()** inside **tcp.cc** to update the RTT apart from the existing technique. We calculate the new average taking into account the previous 10 (hard-coded) values. Hence it checks if we already have observed more than 9 values, and upon getting true, new_rtt_update() function gets called.

```
589  void TcpAgent::rtt_update(double tao)
590  {
591      double now = Scheduler::instance().clock();
592      if (ts_option_)
593          t_rtt_ = int(tao /tcp_tick_ + 0.5);
594      else {
595          double sendtime = now - tao;
596          sendtime += boot_time_;
597          double tickoff = fmod(sendtime, tcp_tick_);
598          t_rtt_ = int((tao + tickoff) / tcp_tick_);
599      }
600      if (t_rtt_ < 1)
601          t_rtt_ = 1;
602
603      register short delta;
604      if(last_observed_data > 9){// call the newly added user function to proceed,
605          // if at least 10 packets have been observed
606          new_rtt_update();
607      }
```

Figure 11: Function call to update RTT

This newly introduced function is added as a prototype in the **tcp.h** header file.

```
285     virtual void rtt_init();
286     virtual double rtt_timeout();   /* provide RTO based on RTT estimates */
287     virtual void rtt_update(double tao);    /* update RTT estimate */
288     void new_rtt_update();  /* update RTT estimate in a modified fashion */
289     virtual void rtt_backoff();     /* double multiplier */
290     /* End of state for the round-trip-time estimate. */
```

Figure 12: Function prototype to update RTT

- **Step 3:** The newly introduced function is shown in detail here. What is does is simply call a newly function written **int new_rtt_estimator()** that returns the actual estimated value.

```
580
581 // task is to return averaged srtt value
582 void TcpAgent::new_rtt_update(){
583     printf("Inside modified RTT update mechanism\n");
584     t_srtt_.val_ = new_rtt_estimator(t_srtt_.val_);
585
586 }
587
```

Figure 13: RTT estimator function call

This returned value is directly stored into the **t_srtt_** via the attribute **val_**. This part is a bit tricky, as the **val_** inside **t_srtt_** is by default set as a protected variable. So before accessing it properly to set its new value, we had to change its access modifier inside the file **tracedvar.h** and make it publicly accessible for our modification purpose.

```
127
128     virtual char* value(char* buf, int buflen);
129     int val_;       // value stored by this trace variable
130 protected:
131     virtual void assign(const int newval);
132 };
133
```

Figure 14: Changing the **val_** to be public

- **Step 4:** Next comes the most important function for our change, **new_rtt_estimator()**.

  At first, we initialize our total observed data count to be zero. Then we declare a queue for storing the previously observed RTO values. We ensure that the size of the queue will always be the fixed value 10, as we noted earlier.

```
int last_observed_data = 0;
std::deque <int> observed_RTOs; // always 10 sized, ensured later
int weights[10] = {1, 1, 1, 3, 4, 4, 4, 3, 4, 5};   // weights

// task is to return averaged srtt value, average taken in terms of previous values
int new_rtt_estimator(int old_srtt_){
    //  std::vector <int> rtt_values_observed;
    //  rtt_values_observed.push_back(old_srtt_);
     double total_values = 0;
     for (int i = 0; i < observed_RTOs.size(); i++){
        total_values += observed_RTOs[i] * weights[i];
        total_values += old_srtt_ * 20;
     }
     return (int)(total_values / 50);
}

// task is to return averaged srtt value
```

Figure 15: RTT estimation algorithm using weighted average of RTTs and RTOs

The next task was to declare the weights we used for taking average, making the metric a weighted average of last RTOs. The weights are taken in such way that the most recent RTO entries are weighted higher gradually than the oldest ones, and the middle values get proper mid-valued weights.

Then finally we add the latest **s_rtt** value with a greater weight than the combined RTOs, and divide by the total weights to get an estimation of new RTT.

- **Step 5:** Finally we get back to the original update function, and take the conditional logic branch that makes use of the original mechanism.

```

    else {
        if (t_srtt_ != 0) {
        delta = t_rtt_ - (t_srtt_ >> T_SRTT_BITS);  // d = (m - a0)
        if ((t_srtt_ += delta) <= 0)    // a1 = 7/8 a0 + 1/8 m
            t_srtt_ = 1;
        if (delta < 0)
            delta = -delta;
        } else
            t_rtt_ = t_rtt_ << T_SRTT_BITS;     // srtt = rtt
    }
    delta -= (t_rttvar_ >> T_RTTVAR_BITS);
    if ((t_rttvar_ += delta) <= 0)  // var1 = 3/4 var0 + 1/4 |d|
            t_rttvar_ = 1;
    else {
        //  t_srtt_ = t_rtt_ << T_SRTT_BITS;        // srtt = rtt
        t_rttvar_ = t_rtt_ << (T_RTTVAR_BITS-1);    // rttvar = rtt / 2
    }
    //
```

Figure 16: Original RTT, SRTT calculation logic

- **Step 6:** Here we make sure that the queue size doesn't exceed 10, and we did it by popping the front entry of the queue. Finally we push the latest RTO after it gets calculated back by the original logic kept intact for this part.

```
634     //
635     // Current retransmit value is
636     //    (unscaled) smoothed round trip estimate
637     //    plus 2^rttvar_exp_ times (unscaled) rttvar.
638     //
639     t_rtxcur_ = (((t_rttvar_ << (rttvar_exp_ + (T_SRTT_BITS - T_RTTVAR_BITS))) +
640         t_srtt_)  >> T_SRTT_BITS ) * tcp_tick_;
641
642     if(observed_RTOs.size() > 9){
643         observed_RTOs.pop_front();
644     }
645
646     // adding the latest RTO for further processing
647     observed_RTOs.push_back(t_rtxcur_);
648     last_observed_data++;
649
650     return;
651 }
```

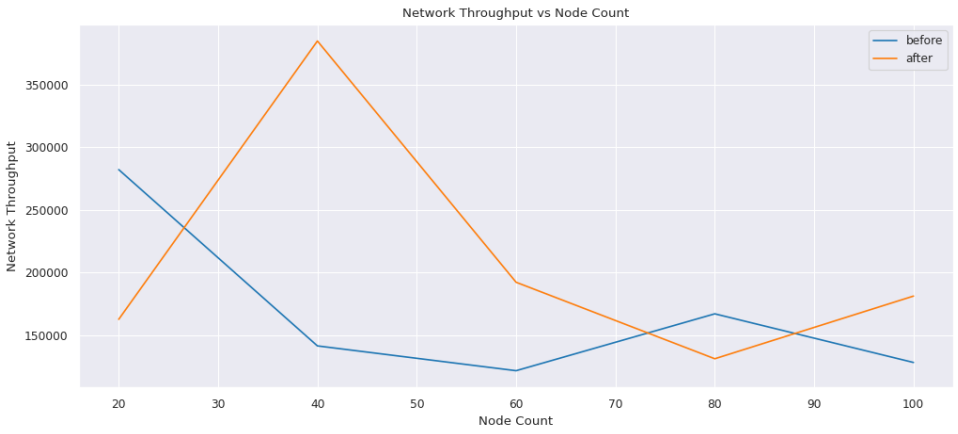
Figure 17: RTO vector push and pop

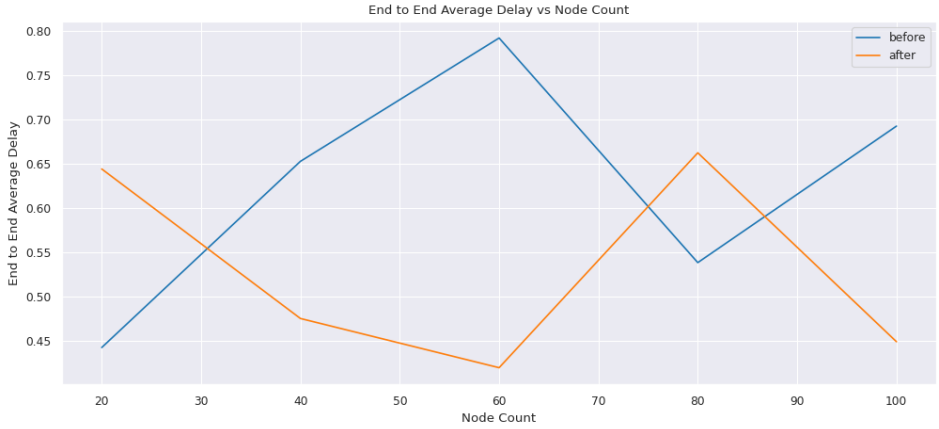
Thus our modification is complete.

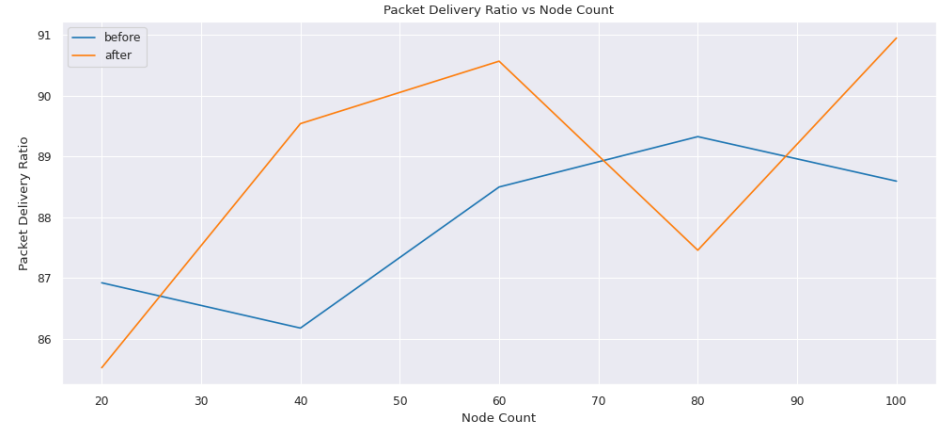
4.3 Updated Result

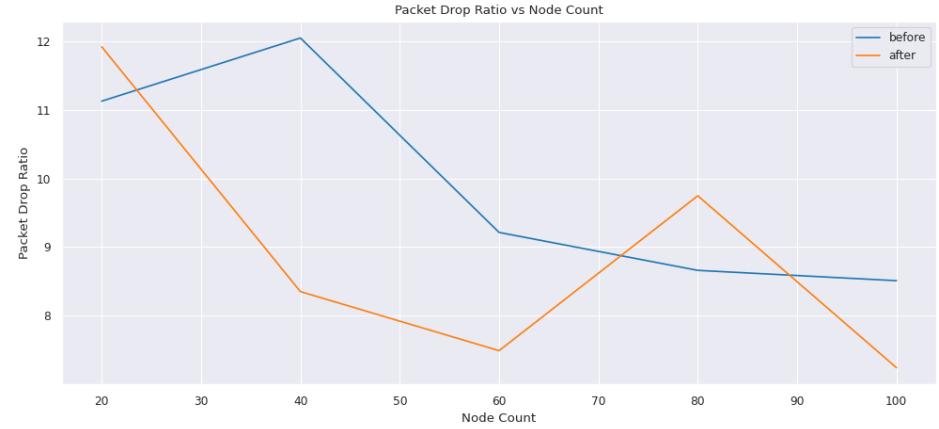
Here we present the result in a merged graph, that includes both the scenario before and after modification.

4.3.1 Node Count




End to End Average Delay vs Node Count
End to End Average Delay
Node Count
before
after
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
20
30
40
50
60
70
80
90
100


Packet Delivery Ratio vs Node Count
Packet Delivery Ratio
Node Count
before
after
91
90
89
88
87
86
20
30
40
50
60
70
80
90
100


Packet Drop Ratio vs Node Count
Packet Drop Ratio
Node Count
before
after
12
11
10
9
8
20
30
40
50
60
70
80
90
100

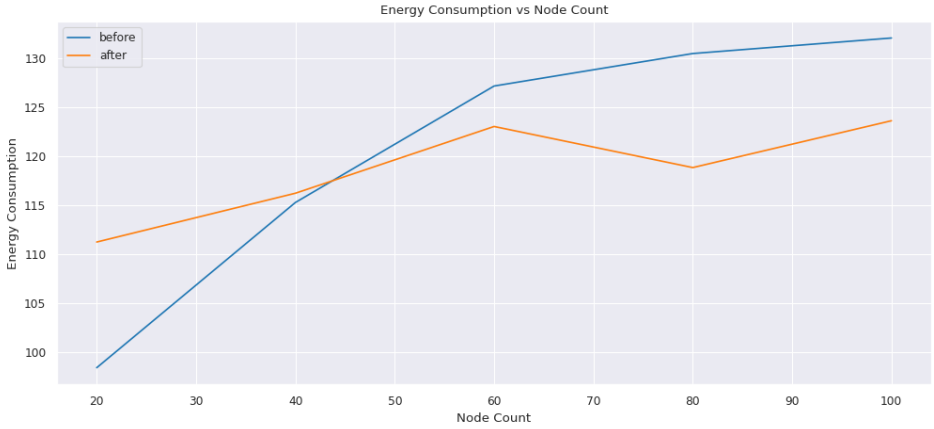
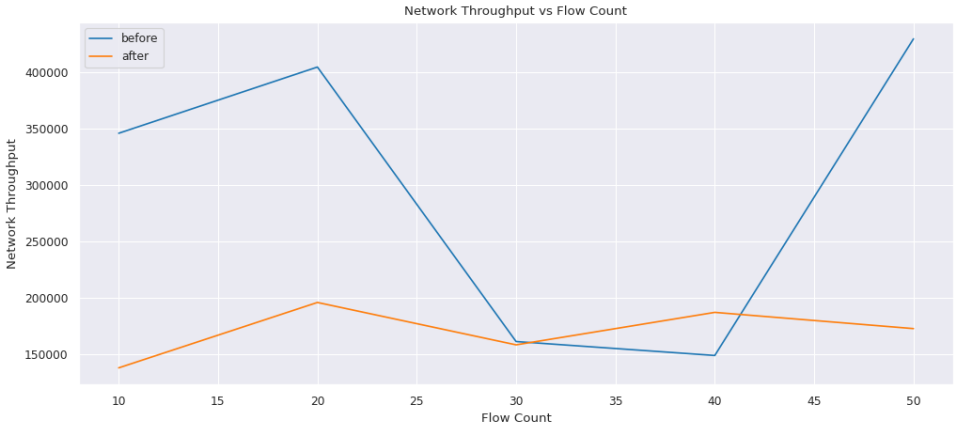
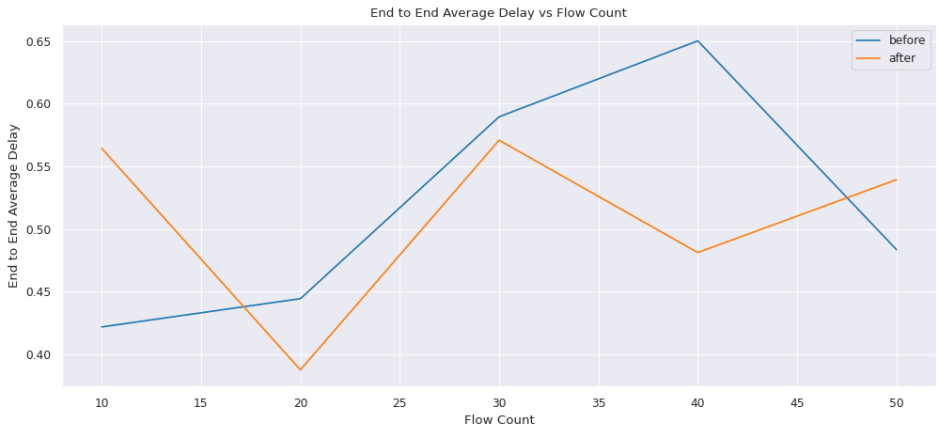
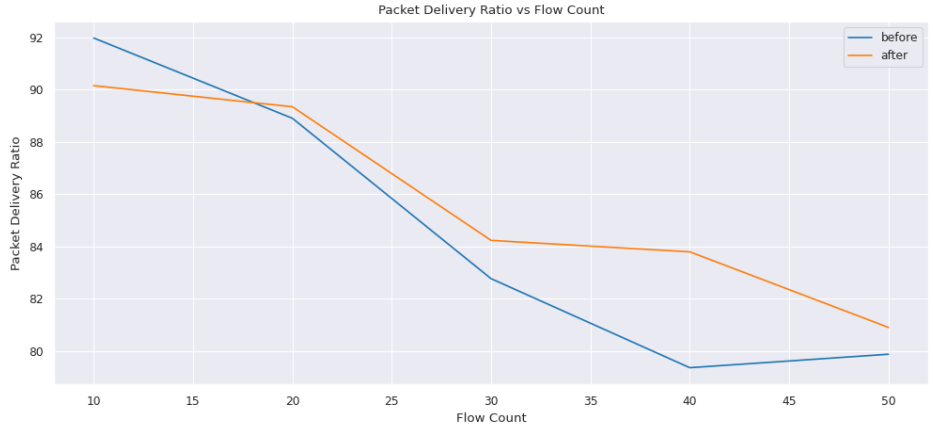


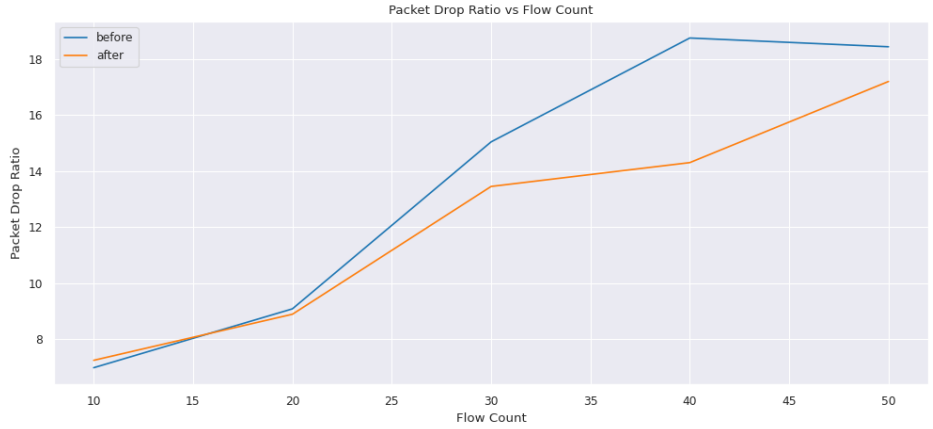
Figure 18: Varying number of nodes

4.3.2 Flow Count









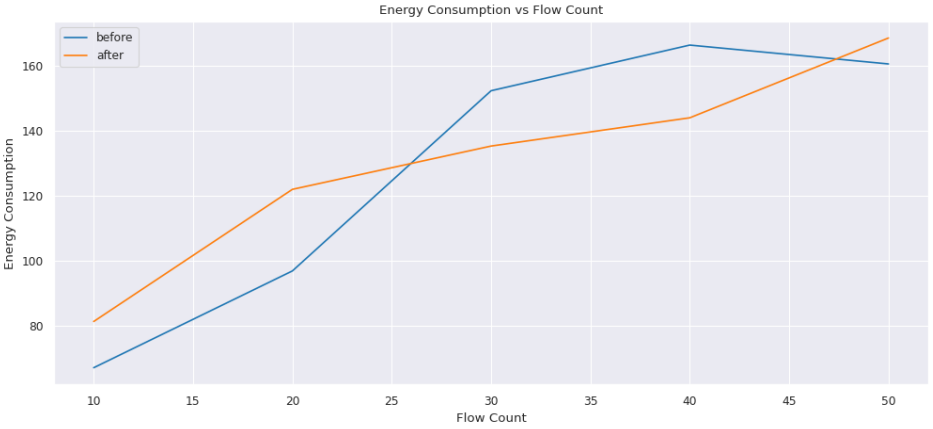
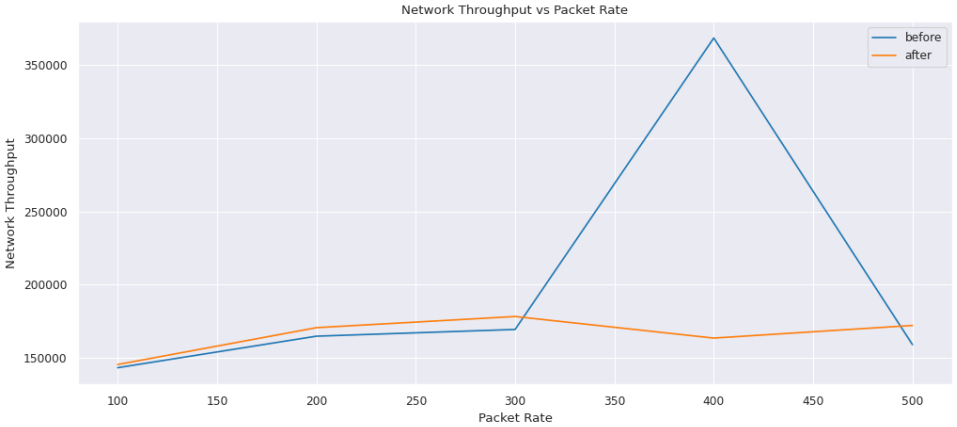
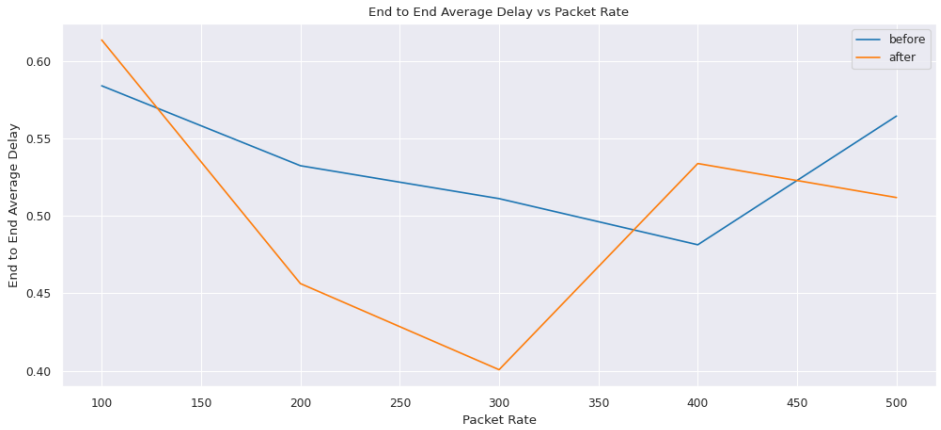
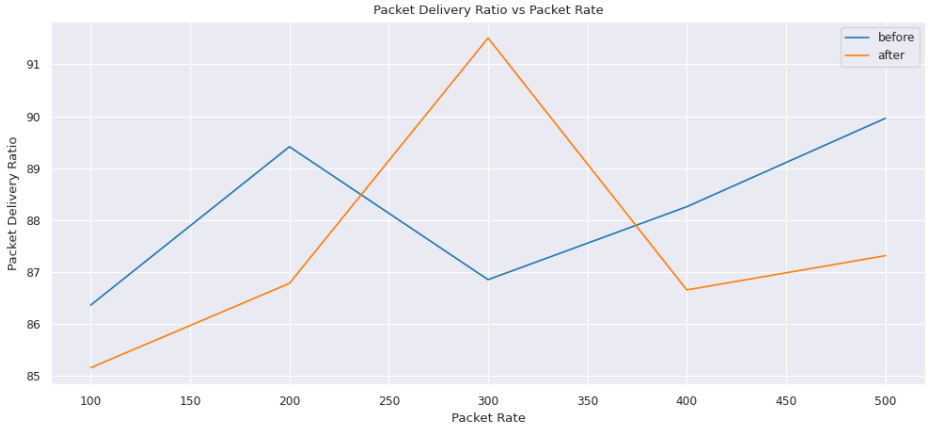


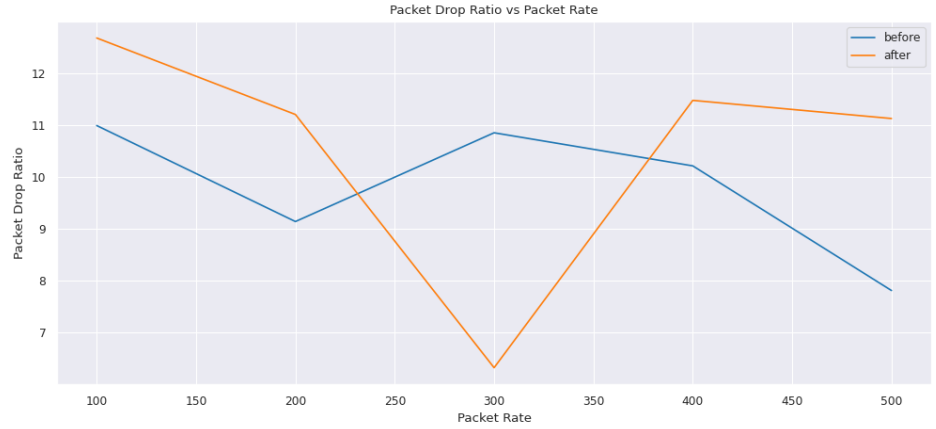
Figure 19: Varying number of flows

4.3.3 Packet Rate









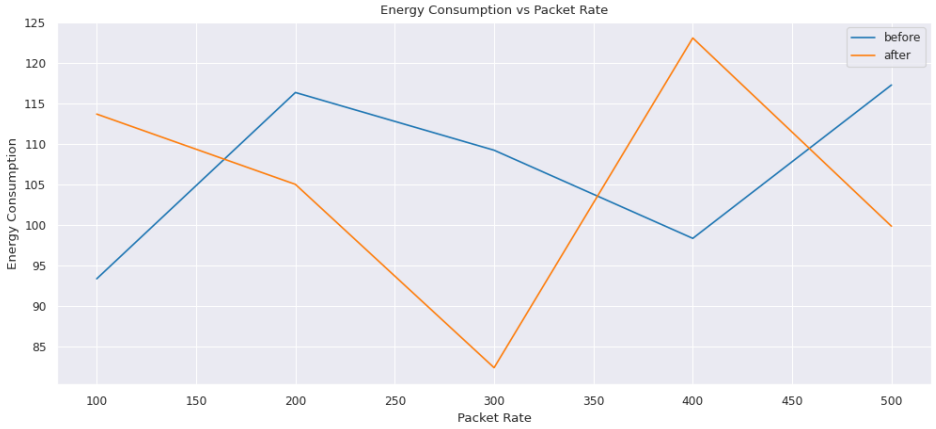
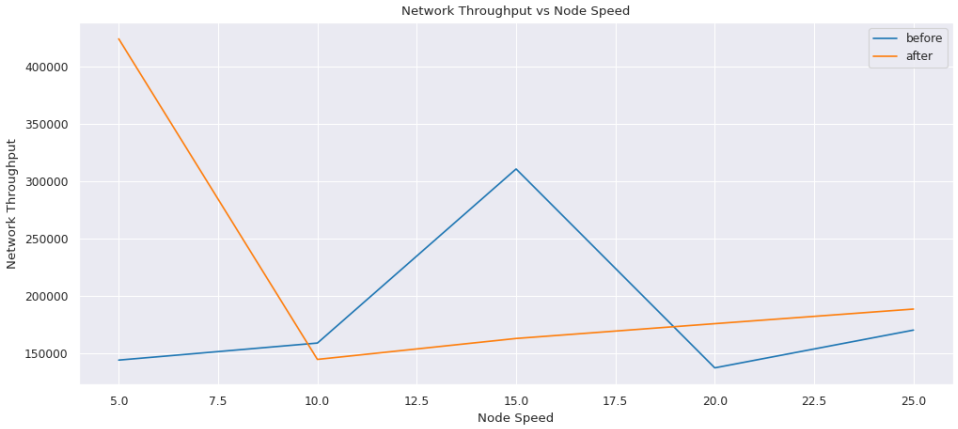
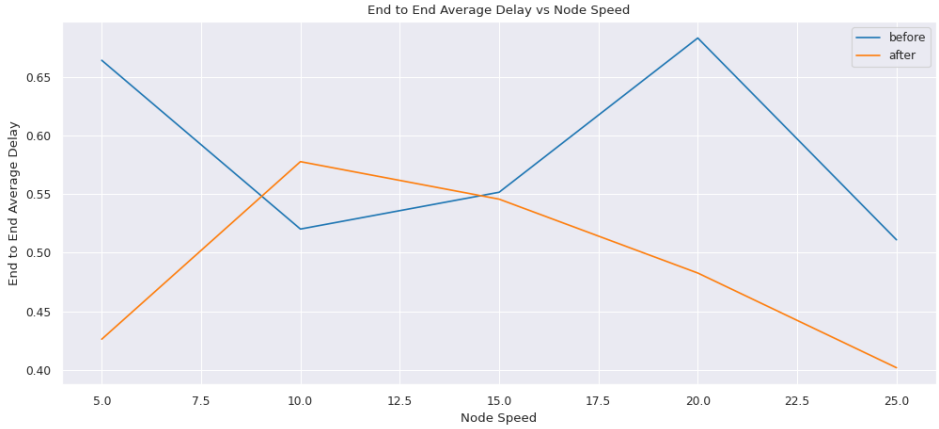
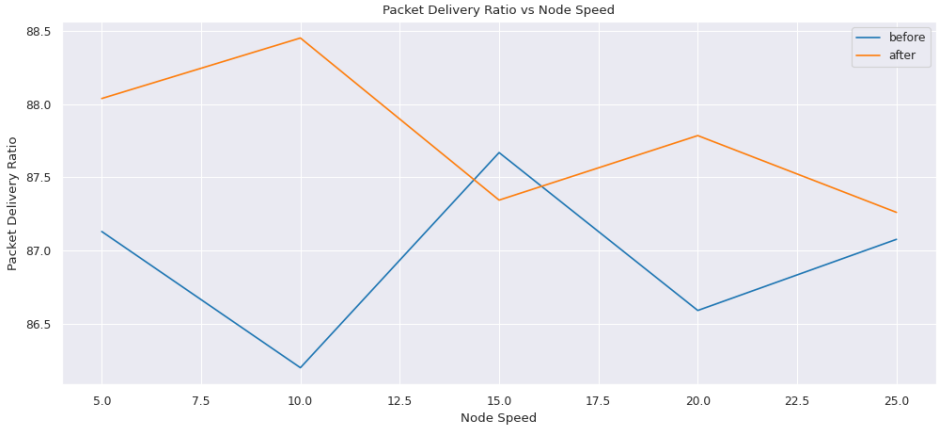


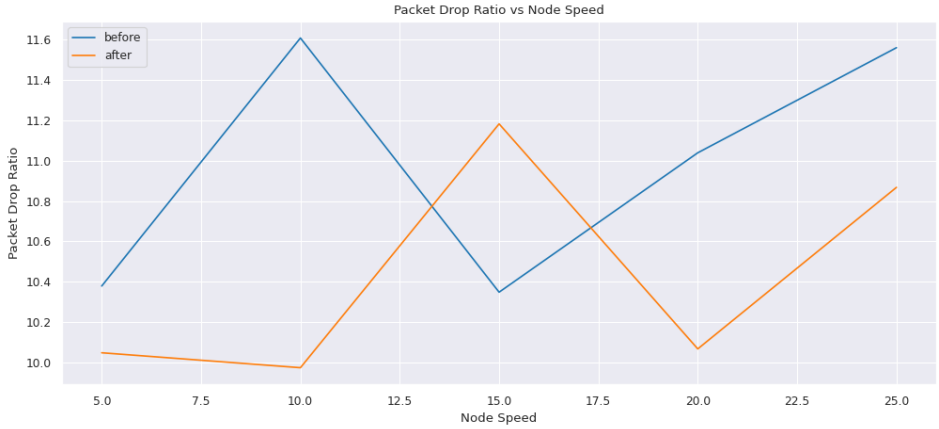
Figure 20: Varying number of packets per second

4.3.4 Node Speed




End to End Average Delay vs Node Speed
before
after
End to End Average Delay
0.65
0.60
0.55
0.50
0.45
0.40
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
Node Speed


Packet Delivery Ratio vs Node Speed
before
after
Packet Delivery Ratio
88.5
88.0
87.5
87.0
86.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
Node Speed


Packet Drop Ratio vs Node Speed
before
after
Packet Drop Ratio
11.6
11.4
11.2
11.0
10.8
10.6
10.4
10.2
10.0
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
Node Speed

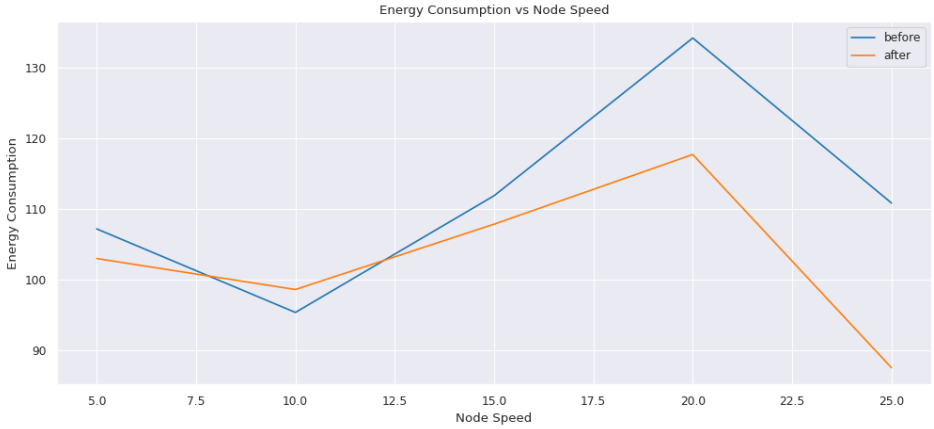


Figure 21: Varying speed of nodes

4.4 Findings

The throughput sees a random peak, but continues to decrease as before when the node count increases. The average delay increases in general. The packet delivery ratio increases. The energy consumption is also lower.

In case of varying flow count, only the drop ratio gets improvement, the rest of other metrics are not noteworthy.

The packet rate seems to put random effect after the modification, as the changes are sometimes similar, sometimes the opposite.

In terms of node speed, almost all of the metrics show an improve than before. To summarize, after the modification, throughput, packet delivery ratio and energy consumption seems to get slightly improved in a variety of combination of the parameters mentioned.

5 Bonus Modification

In this segment, we present the progress of an attempted modification as a bonus task of the project.

5.1 AODV Routing Protocol

Ad hoc **O**n-demand **D**istance **V**ector **(AODV)** is one of the most popular routing protocols used for MANETs in **Network layer**. At each node, AODV maintains a routing table. The routing table entry for a destination contains three essential fields: a next hop node, a sequence number, and a hop count. All packets destined for the destination are sent to the next hop node. The sequence number acts as a form of time-stamping. The hop count represents the current distance to the destination node.

In AODV, routes to a destination are discovered only when a source wants to send a packet to that particular destination, eventually saving much of the wasted work needed when the topology changes before that route is used. This property of discovering routes exactly when they are needed is what makes this hop-by-hop routing algorithm **on demand**. This is achieved by a request-response cycle based on RREQ and RREP messages between the nodes.

5.2 Black Hole Attack

A blackhole attack is a type of security threat that can occur in wireless networks, particularly those that use the AODV routing protocol. In a black hole attack, a malicious node or attacker

pretends to have a valid route to a destination node and sends fake route replies (RREP) to other nodes in the network. The fake RREPs advertise a shorter route to the destination, which causes other nodes to update their routing tables and send their traffic through the malicious node.

Once the traffic is redirected through the malicious node, it drops or discards the packets instead of forwarding them to the intended destination. This causes the traffic to disappear or become "lost" in the network, hence the name "blackhole attack".

5.3 Changes

In this part of the project, we attempt to achieve two separate goals. The first one is to simulate a single black hole node by modifying the existing codebase. And the second part would be to prevent such attacks from a single or multiple mobile nodes.

5.3.1 Simulation

```
276 +        bool            malicious; // Added for Blackhole Attack -
```

Figure 22: Adding a malicious boolean field inside AODV protocol

```
86 +     if(strncasecmp(argv[1], "hacker", 6) == 0) {
87 +      malicious = true;
88 +       return TCL_OK;
89 +     }
```

Figure 23: Setting a particular node as black hole attacker

```
154 +   malicious = false; // Added for Blackhole Attack
155 +
```

Figure 24: Initializing malicious as false

```
453 +         if (malicious == true ) {
454 +            drop(p, DROP_RTR_ROUTE_LOOP); // DROP_RTR_ROUTE_LOOP is added for no reason.
455 +            return;
456 +         }
```

Figure 25: Drop the packet if it is malicious

```
790 +  else if (malicious == true) {
791 +     seqno = max(seqno, rq->rq_dst_seqno)+1;
792 +     if (seqno%2) seqno++;
793 +
794 +     sendReply(rq->rq_src,          // IP Destination
795 +               1,                   // Hop Count is set to 1
796 +               rq->rq_dst,          // Dest IP Address
797 +               seqno,               // Dest Sequence Num
798 +               MY_ROUTE_TIMEOUT,    // Lifetime
799 +               rq->rq_timestamp);   // timestamp
800 +    Packet::free(p);
801 +  }
```

Figure 26: Freeing the packet inside RREQ message

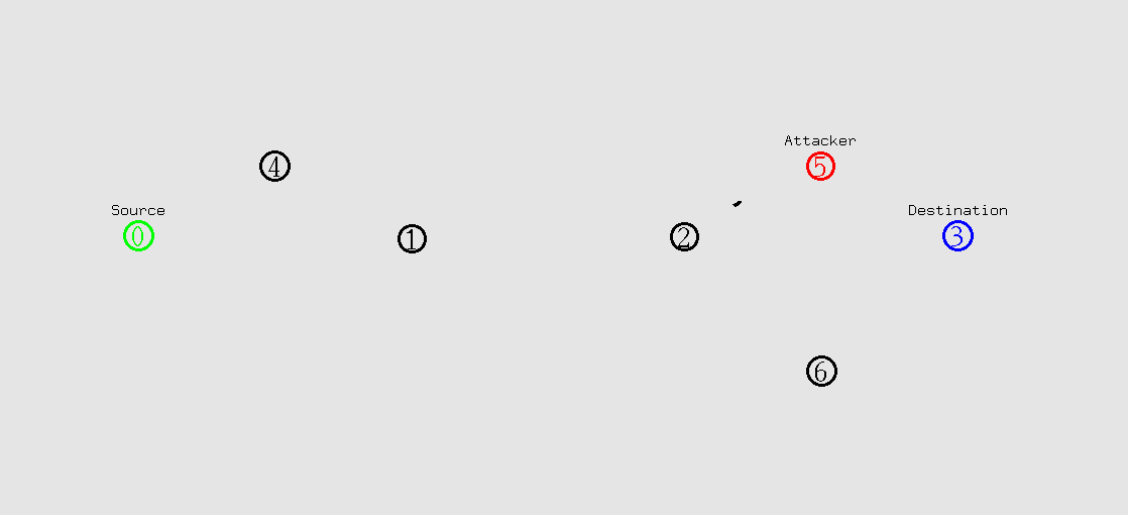


Figure 27: Destinaion does not receive any packet

5.3.2 Prevention

Unfortunately, this segment has not been completed yet. We hope to implement a full working prevention mechanism that can effectively tackle black hole attacks in the future.