

# Evaluation of MANET protocols through Mininet WiFi

Τμήμα Πληροφορικής ΟΠΑ

Χαρμαντάς Χρήστος

3160189

## Contents

Introduction .....	2
MANET Protocols .....	2
OLSR.....	2
BATMAN Advanced.....	6
MANET Protocols Evaluations on Mininet WiFi .....	7
Evaluation Approach.....	7
Mininet WiFi .....	7
Topology .....	7
Experimentation Setup .....	9
Mininet WiFi topology code.....	9
Experiments .....	10
Evaluation of Results.....	16
Challenges.....	23
Conclusion .....	23
References .....	24
Appendix.....	24

## Introduction

In Ad hoc topologies there are many MANET routing protocols that are used to connect the nodes of the network. Some protocols operate in a similar way and others in a very different one. The main question we ask is: which one of them operates best. With the help of Mininet WiFi emulator we created a network, where we tested these MANET protocols.

## MANET Protocols

The MANET (Mobile Ad Hoc Network) protocols we examined are OLSR, B.A.T.M.A.N and BABEL. Mininet WiFi claims that it can support all the three, but after testing them, we discovered that BABEL could not be applied only by the emulator and that more actions were needed to make it work. More information about BABEL will be found on the Challenges section. Both OLSR and B.A.T.M.A.N advanced are explained bellow.

### OLSR

OLSR stands for Optimized Link State Routing Protocol [1]. It is a proactive or table-driven protocol. This means that in order to send a packet, a node will have to consult its routing table.

OLSR uses some of the network nodes as multipoint relays (MPR). The idea of multipoint relays is to minimize the flooding of broadcast packets in the network by reducing duplicate retransmissions in the same region. Each node in the network selects a set of nodes in its neighborhood, to which it retransmits its packets. The neighbors of any node N which are not in its MPR set, read and process the packet but do not retransmit the broadcast packet received from node N. For this purpose, each node maintains a set of its neighbors which are called the MPR Selectors of the node. Every broadcast message coming from these MPR Selectors of a node is assumed to be retransmitted by that node. This set can change over time, which is indicated by the selector nodes in their HELLO messages.

The HELLO Protocol is responsible for establishing and maintaining neighbor relationships. It also ensures that communication between neighbors is bidirectional. Hello packets are sent periodically out of all node interfaces. Bidirectional communication is indicated when the node sees itself listed in the neighbor's HELLO Packet.

Each node selects its MPR set among its one hop neighbors in such a manner that the set covers (in terms of radio range) all the nodes that are two hops away. The multipoint relay set of node N, called  $MPR(N)$ , is an arbitrary subset of the neighborhood of N which satisfies the following condition: every node in the two-hop neighborhood of N must have a bidirectional link toward  $MPR(N)$ .

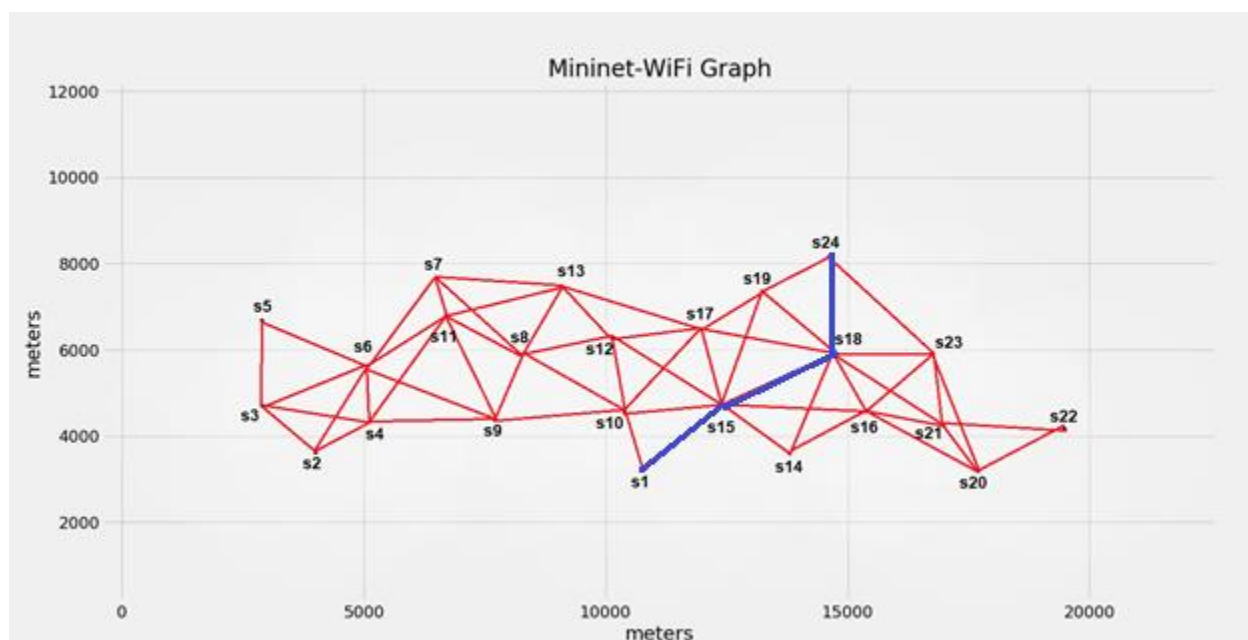


Figure 1 : Path from s1 to s24

For example, on Figure 1 we see the path from node s1 to node s24. The path is s1-s15-s18-s24. First, we will find the MPRs of the nodes. For s1 we have the one hop neighbors s10 and s15. These two can give access to the two hop neighbors of s1. On Figure 2, we see in red colour the nodes that belong in the

minimum set of nodes, which can give s1 access to all his two hop neighbors.

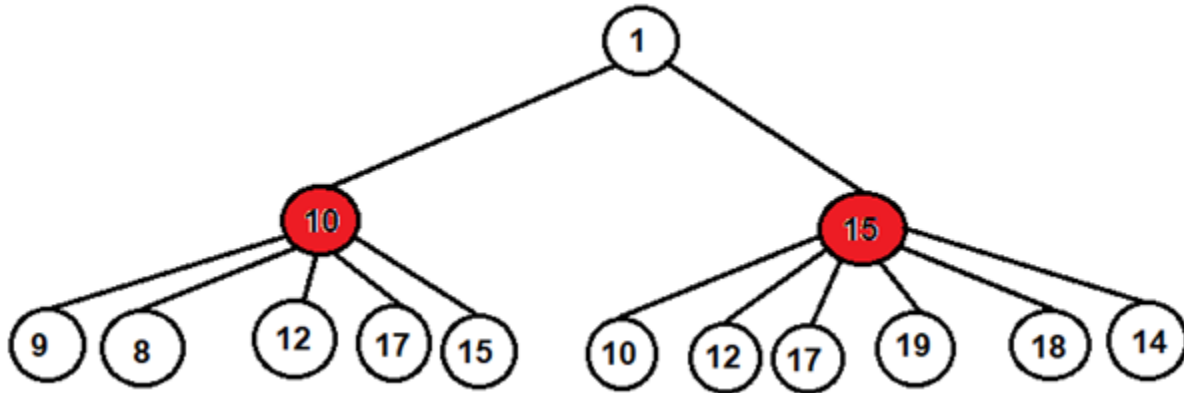
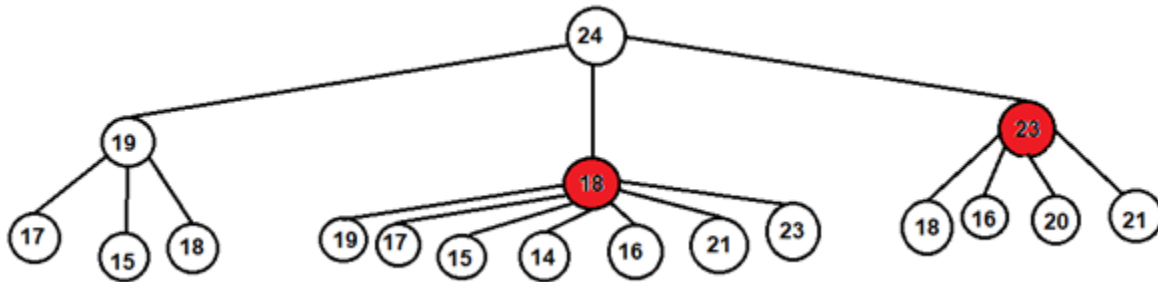


Figure 2 : MPRs of node s1

In figure 2 the MPRs are all s1's on hop neighbors. On figure 3 we see s24's two hop neighborhood.



We see that the nodes to which node 19 gives access to, can be given access from node 18 as well. So, it will not be an MPR of node 24. In the cases of node 18 and node 23, they can both reach some common nodes, but also some different ones. That is why we will make them both MPRs of node 24 and they will be part of the node 1 - node 24 path.

The smaller the multipoint relay set is, the better the routing protocol is, because we will have paths with fewer intermediate nodes.

The OLSR protocol relies on the selection of multipoint relays, and calculates its routes to all known destinations through these nodes, i.e. MPR nodes are selected as intermediate nodes in the path. So, each node in the network periodically broadcasts the information about its one-hop neighbors which have selected it as a multipoint relay. Upon receipt of this MPR Selectors' information, each node calculates and updates its routes to each known destination. Therefore, the route is a sequence of hops through the multipoint relays from-source to destination.

Each node must detect the neighbor nodes with which it has a direct and bi-directional link. Consequently, all links must be checked in both directions. To accomplish this, each node periodically broadcasts its HELLO messages, containing the information about its neighbor. These control messages are transmitted in the broadcast mode. These are received by all one-hop neighbors, but they are not relayed to further nodes. Also, a link between two nodes is considered valid based on the ETX metric [8]. The ETX metric value of a link, is established by measuring the rate of successful exchange of OLSR control packets over that link. More specifically, node A computes the value of the ETX metric of its link to node B by

continuously estimating the loss rates over this link, in both directions: from B to A (this rate is called  $R_{\text{etx}}$ ), and from A to B (this rate is called  $D_{\text{etx}}$ ). Node A computes  $R_{\text{etx}}$  as the measured proportion of packets successfully arriving from B and signals this value in HELLO messages by inclusion of a  $R_{\text{etx}}$ . Symmetrically, node B computes the proportion of packets successfully arriving from A and signals its value in HELLO messages by inclusion of a  $R_{\text{etx}}$ , which node A can then take as  $D_{\text{etx}}$  value for this link.

The value of the ETX metric of the link is then  $\text{ETX} = R_{\text{etx}} * D_{\text{etx}}$ , which corresponds to the expected number of attempts to successfully receive and acknowledge a packet over this link. It should be taken into account that this metric is symmetric, i.e. on a link connecting node A and node B, the ETX metric value for this link computed by node B will be the same as the ETX metric value computed by node A [9].

A HELLO message contains:

- the list of addresses of the neighbors to which there exists a valid bi-directional link.
- the list of addresses of the neighbors which are heard by this node (a HELLO has been received) but the link is not yet validated as bi-directional if a node finds its own address in a HELLO message, it considers the link to the sender node as bi-directional.
- the  $R_{\text{etx}}$  value.

These HELLO messages allow each node to learn the information of its neighbors up to two hops. Based on this information, each node performs the selection of its multipoint relays. These selected multipoint relays are indicated in the HELLO messages with the link status MPR. On the reception of HELLO messages, each node can construct its MPR Selector table with the nodes who have selected it as a multipoint relay. In the neighbor table, each node records the information about its one hop neighbors, the status of the link with these neighbors, and a list of two hop neighbors that these one-hop neighbors give access to.

To build the intra-forwarding database needed for routing packets, each node broadcasts specific control messages called Topology Control (TC) messages. TC messages are forwarded like usual broadcast messages in the entire network.

A TC message is sent periodically by each node in the network to declare its MPR Selector set, i.e., the message contains the list of neighbors who have selected the sender node as a multipoint relay. The information diffused in the network by these TC messages will help each node to build its topology table. A node which has an empty MPR selector set, i.e., nobody has selected it as a multipoint relay, may not generate any TC message.

Each node of the network maintains a topology table, in which it records the information about the topology of the network obtained from the TC messages. A node records information about the multipoint relays of other nodes in this table. Based on this information, the routing table is calculated. An entry in the topology table consists of an address of a (potential) destination (an MPR Selector in the received TC message), address of a last-hop node to that destination (originator of the TC message) and the corresponding MPR Selector set sequence number (of the sender node). It implies that the destination node can be reached in the last hop through this last hop node. Each topology entry has an associated holding time, upon expiry of which it is no longer valid and hence removed.

In summary, the steps of the protocol's operation are as follows:

- Each node sends HELLO messages and calculate the ETX metric for its links, in order to learn its neighbors up to two hops.
- Based on that information, it chooses its multipoint relays.
- Each node broadcasts its multipoint relays through them so that every other node in the network can learn them.
- Based on this information every node builds its own routing table.
- When a source node sends packets to a destination node, the path will be based on the routing table and its intermediate nodes will be the multipoint relays from source to destination.

### BATMAN Advanced

BATMAN stands for Better Approach to Mobile Ad-hoc Networking [2]. It is a proactive link-state based routing protocol that aims to offer a Better Approach to Mobile Ad hoc Networking. It is designed for lossy and unstructured multi-hop mesh networks and operates on data link layer using mac addresses.

BATMAN nodes do not have to calculate the entire routes for their outgoing packets, and they do not need to know the full topology of the mesh network. In BATMAN, all nodes periodically broadcast "hello" signals, also known as originator messages (OGM) to their neighbors. Each originator message consists of an originator address, a sending node address and a unique sequence number. When an OGM is received, the receiving node changes the sending address to its own address and re-broadcasts the message. The sequence number is used to identify which of a pair of messages is newer. With this process, not only does each node in the network learn its own direct neighbors, but it also learns about other nodes that are not in range through a direct link but can be reached by hopping through a neighbor.

BATMAN divides a given link quality into 2 distinct parts: receiving link quality and transmit link quality. The receiving link quality expresses the probability of a successful packet transmission to the node. The transmit link quality describes the probability of a successful transmission towards a neighbor node. BATMAN is more interested in the transmit link quality as the receiving link quality cannot be used to influence the routing decision.

The transmit link quality can be derived from the receiving link quality by applying some calculations on the packet count. More specifically, BATMAN knows the receiving link quality (RQ) by counting the packets of its neighbors. It also knows the echo link quality (EQ) by counting rebroadcasts of its own OGMs from its neighbors so it can calculate the transmit link quality (TQ) by dividing the echo link quality by the receiving link quality.

Whenever the OGM is generated, TQ is set to maximum length (255) before it is broadcasted. The receiving neighbor will calculate their link quality into the received TQ value and rebroadcast the packet. Hence, every node receiving a packet knows about the transmit quality towards the originator node.

Each BATMAN node maintains an originator table which lists the addresses of all other reachable nodes in the network. The estimation metric (TQ= Transmit Quality) is used to evaluate the link transmit quality between an originator node and a destination node, based on throughput [10]. Each address entry in the originator table is assigned a TQ metric. The value of TQ metric rank is defined on a scale between 0 and 255 (0 indicating no connection and 255 being excellent link quality).

When a data packet arrives for transmission, the node refers to the originator table to determine the direction in which the packets are to be sent. Specifically, it checks its originator table and forwards the packet towards the destination with the highest rank entry (based on the TQ) - the best next neighbor.

When a node enters or leaves the network, or when a link fails or is degraded, this will be reflected in the TQ metric and the node/link will be avoided if a better path is available.

BATMAN nodes do not maintain the full route to every destination. Each node along the route only maintains the information about the best next neighbor through which it finds the best route. So, the complete topology is not known to any single node, and topology and routing decisions are distributed among all the nodes.

More specifically:

- Each node sends hello messages to its neighbors to establish a connection.
- Each node that receives a hello message, changes the sending address and rebroadcast it to its neighbors.
- Nodes do not know the entire topology, so they create a table with all the other nodes of the network, based on the hello messages. In this table they do not know the full path for each node, but they do know which is the best next hop to send the packet in order to arrive to its destination.

## MANET Protocols Evaluations on Mininet WiFi

### Evaluation Approach

#### Mininet WiFi

Mininet WiFi is an open source tool that is used to emulate wireless networks. It is suitable for SDN (Software Defined Networks), Ad hoc and mesh networks. Mininet WiFi is an extension of the Mininet SDN network emulator [3] and extends the functionality of Mininet by adding virtualized WiFi stations and Access Points based on the standard Linux wireless drivers and the 80211\_hwsim wireless simulation driver. It has additional classes to support the addition of these wireless devices in a Mininet network scenario and to emulate the attributes of a mobile station such as position and movement relative to the access points. Mininet WiFi extended the base Mininet code by adding or modifying classes and scripts. So, Mininet WiFi adds new functionality and still supports all the normal SDN emulation capabilities of the standard Mininet network emulator.

#### Topology

Using Mininet WiFi we created a virtual network that contains 24 wireless stations randomly placed. Each station has 3095 meters as antenna range and TX power of 2 dBm. The area that the nodes have been placed in, is 82.377.750 m<sup>2</sup>. The minimum distance between nodes is 1.262,23 meters and the maximum is 16.765,47 meters.

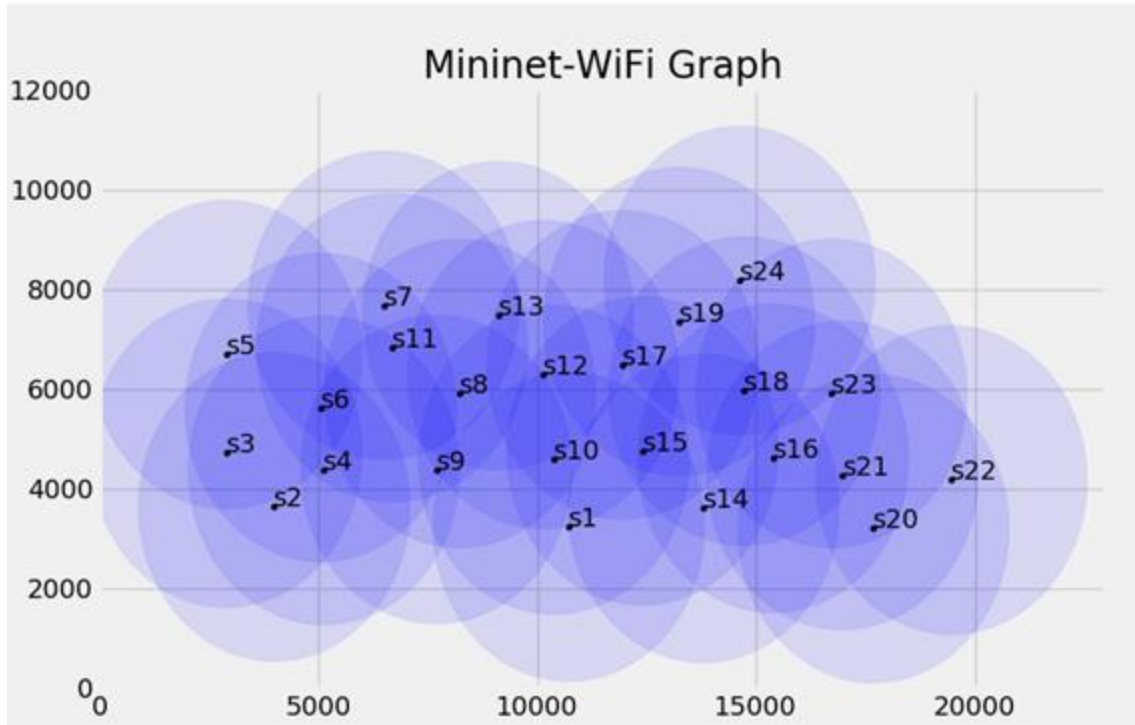


Figure 3: Topology image with the antenna range circles.

The topology is described in figure 4. The antenna's range of each station is defined by a circle centered at the corresponding node.

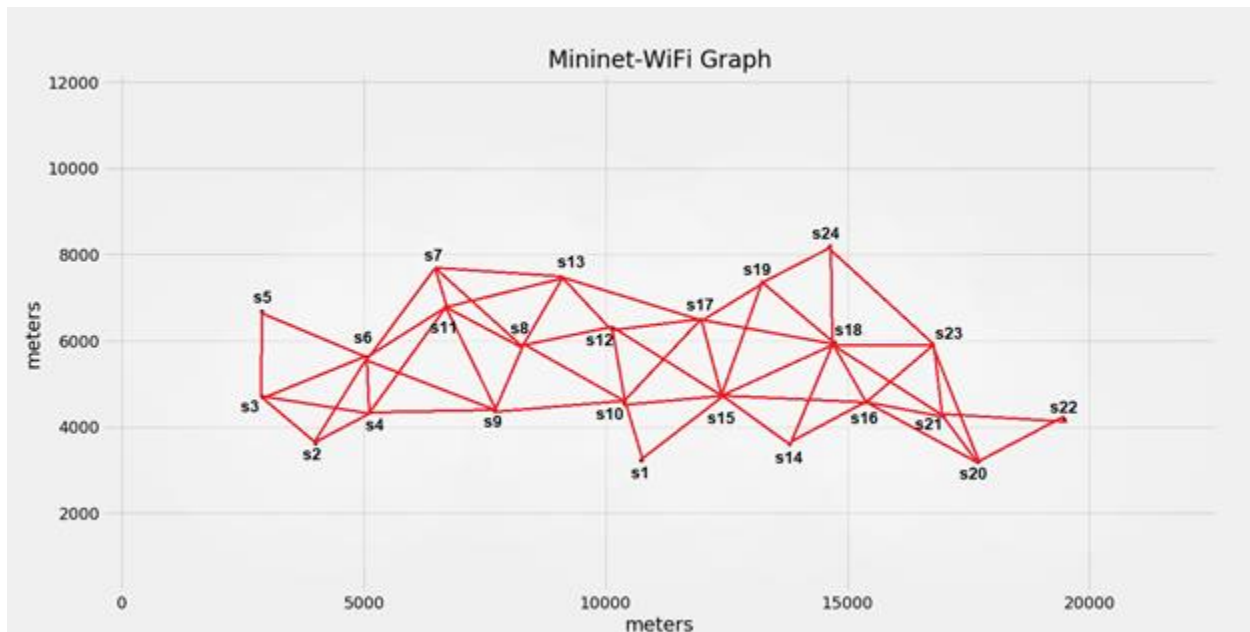


Figure 4 : Graph of the network

In figure 5, we see the network graph. There is a line connecting every pair of nodes that communicate directly. These red lines (links) are used to create the paths between nodes by the routing protocols.



## Experimentation Setup

### Mininet WiFi topology code

For creating the simulation of the network, we used the Mininet WiFi emulator. The virtual network is created by running a python script. The code of the script uses the python API of the Mininet WiFi. This API gives us the ability to configure some parameters of the network for the experiments. The code for the network we used for the experiments is explained below and contained on *Appendix 1*.

All the libraries (eg CLI) that will be needed are imported.

In the topology method the variables of the network are initialized. The implemented radio model is wmedium. We chose this, because the kernel module mac80211\_hwsim [11] uses the same virtual medium for all wireless nodes. This means that all nodes are internally in range of each other, and they can be discovered in a wireless scan on the virtual interfaces. Mininet WiFi simulates their position and wireless ranges by assigning stations to other stations or access points and revoking these wireless associations. If wireless interfaces need to be isolated from each other (e.g. in adhoc or mesh networks) a solution like wmediumd is required. Wmediumd uses a kind of a dispatcher to permit or deny the transfer of packets from one interface to another [12].

All 24 stations of the network are declared, by setting their names, MAC addresses, positions and MANET protocols by extra arguments as shown below.

```
s2 = net.addStation('s2', mac='00:00:00:00:00:02', position='3990,3630,0',  
**kwargs)
```

From all available propagation models that Mininet WiFi offers, Friis and TwoRayGround were tested for exterior space topologies. The free-space path loss (Friis) is the attenuation of radio energy between the feedpoints of two antennas that results from the combination of the receiving antenna's capture area plus the obstacle-free, line of sight path through free space which is usually air. It does not include any power loss in the antennas themselves due to imperfections such as resistance. Free space loss increases with the square of distance between the antennas because the radio waves spread out by the inverse square law and decreases with the square of the wavelength of the radio waves. [4] In the case of Friis, there is no limitation on signal transmission and we could not test the routing protocols, because every node would connect directly with everyone.

The TwoRayGround model is a multipath radio propagation model which predicts the path losses between a transmitting antenna and a receiving antenna when they are in line of sight (LOS). Generally, the two antenna each have different height. The received signal having two components, the LOS component and the reflection component formed predominantly by a single ground reflected wave [5]. When the twoRayGround model were tested we noticed that it was not working properly. More specifically, it did not have any limitation on range. We could ping any node as far away as it was. After many tests and communication with the Mininet WiFi developer (Ramon Fontes [6]), he informed us that the model had not been developed properly and thus it was not functional.

Finally, we used the logDistance propagation model that works properly. The log-distance path loss model is a radio propagation model that predicts the path loss of a signal encounters inside a building or densely populated areas over distance. We chose 1.8 as the path loss exponent for the type of grocery store. In

this way, we approximate as much as possible open space, having max range based on TX power and many objects that can interfere electromagnetic waves, like a thick forest.

For every station, an interface is created with enabled the Ad hoc mode of the WiFi interface, the proper station name that helps us identify it, g mode, on channel 5 and with the protocol that we pass through arguments.

```
net.addLink(s1, cls=adhoc, intf='s1-wlan0', ssid='adhocNet', mode='g',  
channel=5, **kwargs)
```

The g mode refers to IEEE 802.11g. IEEE 802.11 is part of the IEEE 802 set of local area network (LAN) technical standards and specifies the set of media access control (MAC) and physical layer (PHY) protocols for implementing wireless local area network (WLAN) computer communication. The standard and amendments provide the basis for wireless network products using the Wi-Fi brand and are the world's most widely used wireless computer networking standards. In 2002 and 2003, WLAN products supporting a newer standard called 802.11g emerged on the market. 802.11g attempts to combine the best of both 802.11a and 802.11b. 802.11g supports bandwidth up to 54 Mbps, and it uses the 2.4 GHz frequency for greater range. 802.11g is backwards compatible with 802.11b, meaning that 802.11g access points will work with 802.11b wireless network adapters and vice versa [7].

The network is built and we set every station's TX power to 2 dBm.

```
s1.setTxPower(2, intf='s1-wlan0')
```

After running the Mininet WiFi python script to create the network, the CLI will allow us to write commands on the same terminal. This will give us the ability to change some network variables. The commands we will use to do that is based on the Mininet WiFi API. Through it we can access many node variables, such as the TX power, position or open an xterm, which is a terminal for a chosen station.

Some API command examples:

```
mininet-wifi> xterm s1 (It opens a terminal for station s1)
```

```
mininet-wifi> s1 iw dev sta1-wlan0 set txpower fixed 10 (Sets the TX power of s1 to  
10 dBm)
```

```
mininet-wifi> py s1.wintfs[0].range (It prints the range of station s1)
```

```
mininet-wifi> py s1.setPosition('10,10,0') (It sets the position of station s1 on the three  
axis)
```

## Experiments

The experiments we ran on both MANET protocols used all 24 nodes. The results were taken for those nodes that are at the edge of the network because we wanted to see how the network operates when stations communicate from the one edge of the network to the other. In this way all nodes have a role as a sender, receiver, or intermediate. The edge nodes are 1, 2, 3, 5, 7, 13, 14, 20, 22, 23 and 24. The inputs and outputs for our study are:

- Distances between the edge stations of the network.
- Hops between every station and which they are.
- Throughput vs input rates of 1, 5, 10, 50, 100, 500 Kbps and 1, 5, 10, 50 Mbps for every station.
- Average throughput vs input rate for every station.
- Average throughput vs input rate in every number of hops.
- Average throughput vs number of hops and its plot.
- Delay vs input rates of 8, 80, 800 Kbps and 16, 32, 64 Mbps for every station.
- Average delay vs input rate for every station.
- Average delay vs input rate for every number of hops.
- Average delay vs number of hops and its plot.
- Network capacity
- Throughput vs input rate for simultaneous flows in the network
- Delay vs input rate for simultaneous flows in the network
- Packet loss vs input rate for simultaneous flows in the network

To produce all the above data, we ran many experiments using different tools, such as iperf, traceroute or ping. For every experiment, many commands with many different variables were necessary to run on many different machines (stations). Unfortunately, because the commands should run on many different terminals, there was not a way to write a script that could run all the necessary commands on all terminals automatically. So, the best option was to write shell scripts running the commands with all the variables that were needed for the experiments and with the appropriate combinations. A lot of time was needed to run all these scripts manually on every station's terminal, but there was a significant time reduction using these shell scripts, even for every one of the stations separately. Also, an advantage of using this code was that we could store all the outputs of the experiment commands. Every experiment script produced a file with useful data that we used on excel sheets.

Below we explain how we produced the data through the experiments.

For the distance between every node a script was used based on the Mininet WiFi API running the following:

```
for i in [4, 2, 1, 0, 13, 19, 21, 22, 23, 12, 6]:
    for j in [4, 2, 1, 0, 13, 19, 21, 22, 23, 12, 6]:
        d=nodelist[i].get_distance_to(nodelist[j])
```

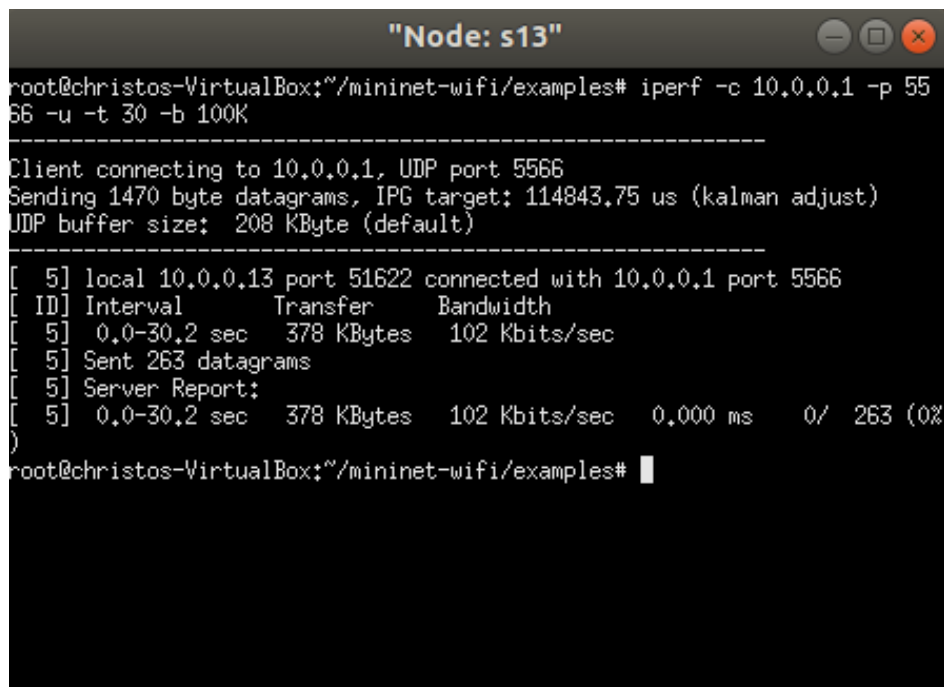
For the graph on both protocols the traceroute command was used. In the case of the BATMAN protocol, the command was used is `batctl traceroute <ip>` cause of the layer 2 that the protocol is running on. The shell script of this experiment is on *Appendix 5*.

Based on the graph, the number of hops between the nodes were extracted.

For every edge node we run experiments between this and all other edge nodes for throughput. On *Appendix 2* there is an example of a shell script that runs iperf and iperf3 commands calculating throughput. For every other throughput experiment this script was changed to produce the results that were needed. The input rates were 1, 5, 10, 50, 100, 500 Kbps and 1, 5, 10, 50 Mbps. We have the average for every input rate of all nodes. For the throughput experiments iperf (2) and iperf3 were used. Iperf was used for the input rates below 1 Mbps and for the rest iperf3 was used.

We noticed that iperf3 could not produce correct results below 700 Kbps and iperf had problems with the results for nodes that were some hops away and were tested for input rate bigger than 1 Mbps. So, at first we had to initialize two servers for every station. One for iperf running `iperf -s -u -p 5566 -i 1` and one for iperf3 `iperf3 -s -p 5566 -i 1`. The -s indicates the server option, the -u that is needed only in iperf indicates the UDP protocol flows, -p is for the port and -i is the interval. After that, we ran the command `iperf -c <ip> -p 5566 -u -t 30 -b 100K` and the same with "iperf3" for iperf3. The different parameters are -t that indicates how many packets the flow will send and -b which is the bandwidth. In the case of 8 Kbps, to work besides the -b 8K, we need to change the packet size with the extra parameter -l 1000.

Some results of the commands:



```

"Node: s13"
root@christos-VirtualBox:~/mininet-wifi/examples# iperf -c 10.0.0.1 -p 5566 -u -t 30 -b 100K
-----
Client connecting to 10.0.0.1, UDP port 5566
Sending 1470 byte datagrams, IPG target: 114843.75 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[  5] local 10.0.0.13 port 51622 connected with 10.0.0.1 port 5566
[ ID] Interval      Transfer    Bandwidth
[  5] 0.0-30.2 sec  378 KBytes  102 Kbits/sec
[  5] Sent 263 datagrams
[  5] Server Report:
[  5] 0.0-30.2 sec  378 KBytes  102 Kbits/sec  0.000 ms  0/ 263 (0%)
root@christos-VirtualBox:~/mininet-wifi/examples#

```

Figure 6 : Example of iperf client for station s13, for input rate 100 Kbps

"Node: s1"							
[ 5]	8.0- 9.0 sec	12.9 KBytes	106 Kbits/sec	5.445 ms	0/	9 (0%)	
[ 5]	9.0-10.0 sec	12.9 KBytes	106 Kbits/sec	5.487 ms	0/	9 (0%)	
[ 5]	10.0-11.0 sec	11.5 KBytes	94.1 Kbits/sec	6.986 ms	0/	8 (0%)	
[ 5]	11.0-12.0 sec	12.9 KBytes	106 Kbits/sec	9.973 ms	0/	9 (0%)	
[ 5]	12.0-13.0 sec	12.9 KBytes	106 Kbits/sec	8.040 ms	0/	9 (0%)	
[ 5]	13.0-14.0 sec	11.5 KBytes	94.1 Kbits/sec	8.190 ms	0/	8 (0%)	
[ 5]	14.0-15.0 sec	12.9 KBytes	106 Kbits/sec	8.253 ms	0/	9 (0%)	
[ 5]	15.0-16.0 sec	12.9 KBytes	106 Kbits/sec	6.455 ms	0/	9 (0%)	
[ 5]	16.0-17.0 sec	12.9 KBytes	106 Kbits/sec	7.171 ms	0/	9 (0%)	
[ 5]	17.0-18.0 sec	11.5 KBytes	94.1 Kbits/sec	7.278 ms	0/	8 (0%)	
[ 5]	18.0-19.0 sec	12.9 KBytes	106 Kbits/sec	6.666 ms	0/	9 (0%)	
[ 5]	19.0-20.0 sec	12.9 KBytes	106 Kbits/sec	6.194 ms	0/	9 (0%)	
[ 5]	20.0-21.0 sec	11.5 KBytes	94.1 Kbits/sec	5.732 ms	0/	8 (0%)	
[ 5]	21.0-22.0 sec	12.9 KBytes	106 Kbits/sec	6.648 ms	0/	9 (0%)	
[ 5]	22.0-23.0 sec	12.9 KBytes	106 Kbits/sec	8.470 ms	0/	9 (0%)	
[ 5]	23.0-24.0 sec	12.9 KBytes	106 Kbits/sec	7.684 ms	0/	9 (0%)	
[ 5]	24.0-25.0 sec	11.5 KBytes	94.1 Kbits/sec	6.092 ms	0/	8 (0%)	
[ 5]	25.0-26.0 sec	12.9 KBytes	106 Kbits/sec	9.007 ms	0/	9 (0%)	
[ 5]	26.0-27.0 sec	12.9 KBytes	106 Kbits/sec	7.469 ms	0/	9 (0%)	
[ 5]	27.0-28.0 sec	11.5 KBytes	94.1 Kbits/sec	7.273 ms	0/	8 (0%)	
[ 5]	28.0-29.0 sec	12.9 KBytes	106 Kbits/sec	6.273 ms	0/	9 (0%)	
[ 5]	29.0-30.0 sec	12.9 KBytes	106 Kbits/sec	8.303 ms	0/	9 (0%)	
[ 5]	0.0-30.2 sec	378 KBytes	102 Kbits/sec	69.447 ms	0/	263 (0%)	

Figure 7 : Results of s1 iperf server.

In figure 7 we see the terminal of the iperf client and in figure 8 the terminal of iperf server. In figure 7 we see that s13 station is running iperf as a client for bandwidth of 100 Kbps and in figure 8, we see s1 station running as an iperf server. After s13 starts iperf running, s1 shows the results for every interval that we have declare. Here the interval is 1 second. On the s1 terminal we see on the first column the interval sequence, on the second the size of the transferred packet, on third the bandwidth, on fourth the jitter and on fifth the lost datagrams and the percentage. After the command is completed, server sends a report on the client as we can see on the client terminal. These are the results we store with the shell scripts.

Throughput results were splitted in number of hops. Average based on every input rate produced for every number of hops.

We measured the delay between every edge node of the network using the ping command. For this measurement we took the avg RTT, which is the average duration measured in milliseconds, from the time sender node sent a request to the time it received a response. The results were extracted from ping commands that we ran for interval of 360. The command was `ping -c <ip> -s <packet size> -i <interval>`. By changing the packet size and interval, we were able to adjust the input rate we needed. For example, with packet size 102 bytes and interval of 0.1 seconds we could transmit with 8 Kbps. The input rates that we used were 8, 80, 800 Kbps and 8, 16, 32, 64 Mbps.

```
"Node: s13"
1024 bytes from 10.0.0.1: icmp_seq=341 ttl=62 time=115 ms
1024 bytes from 10.0.0.1: icmp_seq=342 ttl=62 time=110 ms
1024 bytes from 10.0.0.1: icmp_seq=343 ttl=62 time=106 ms
1024 bytes from 10.0.0.1: icmp_seq=344 ttl=62 time=114 ms
1024 bytes from 10.0.0.1: icmp_seq=345 ttl=62 time=112 ms
1024 bytes from 10.0.0.1: icmp_seq=346 ttl=62 time=109 ms
1024 bytes from 10.0.0.1: icmp_seq=347 ttl=62 time=103 ms
1024 bytes from 10.0.0.1: icmp_seq=348 ttl=62 time=101 ms
1024 bytes from 10.0.0.1: icmp_seq=349 ttl=62 time=107 ms
1024 bytes from 10.0.0.1: icmp_seq=350 ttl=62 time=117 ms
1024 bytes from 10.0.0.1: icmp_seq=351 ttl=62 time=111 ms
1024 bytes from 10.0.0.1: icmp_seq=352 ttl=62 time=120 ms
1024 bytes from 10.0.0.1: icmp_seq=353 ttl=62 time=114 ms
1024 bytes from 10.0.0.1: icmp_seq=354 ttl=62 time=111 ms
1024 bytes from 10.0.0.1: icmp_seq=355 ttl=62 time=111 ms
1024 bytes from 10.0.0.1: icmp_seq=357 ttl=62 time=105 ms
1024 bytes from 10.0.0.1: icmp_seq=358 ttl=62 time=96.4 ms
1024 bytes from 10.0.0.1: icmp_seq=359 ttl=62 time=93.6 ms
1024 bytes from 10.0.0.1: icmp_seq=360 ttl=62 time=88.3 ms

--- 10.0.0.1 ping statistics ---
360 packets transmitted, 352 received, 2% packet loss, time 2782ms
rtt min/avg/max/mdev = 11.898/123.778/171.511/29.236 ms, pipe 20
root@christos-VirtualBox:~/mininet-wifi/examples#
```

Figure 8 : Ping from s13 to s1 example for packet size 1024 bytes and interval 0.001 seconds.

On Appendix 9 we see the shell script were used to produce all the data for this experiment.

We also produced, as we did in throughput, the average delay per input rate for every station and the results of delay were splitted in number of hops as well, with the average again for every input rate.

We chose six different one hop pairs with one being the same. We tested the delay via ping for the two flows from the two stations to the same destination. The test was made for every combination of the input rates. We added both stations' results in a new array. So, by these six different arrays, we produced the network capacity based on the input rate that network started having big changes on delays. The units of the bellow plots are in Kbps.

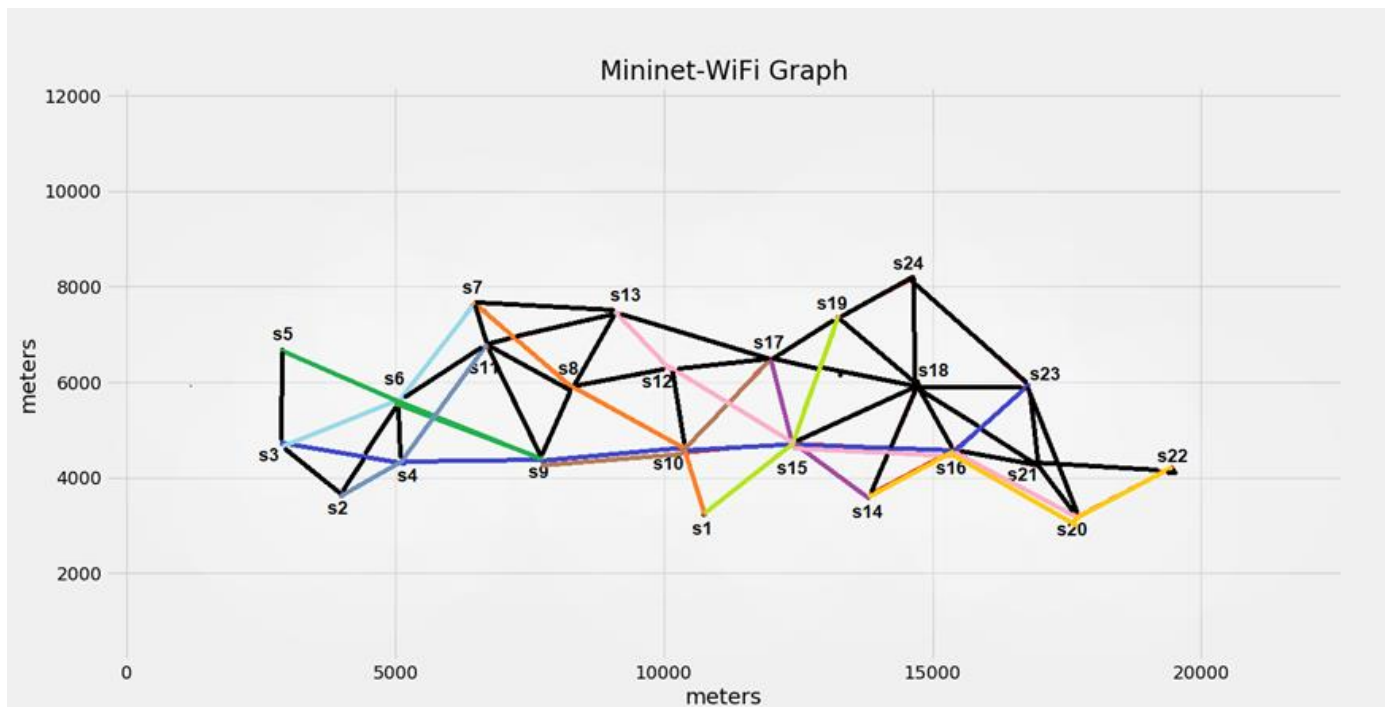


Figure 9 : The path from s3 to s23 has blue colour, the other colors are simultaneous paths for different pairs of nodes in the network, crossing the blue path. These paths have been produced by OLSR protocol.

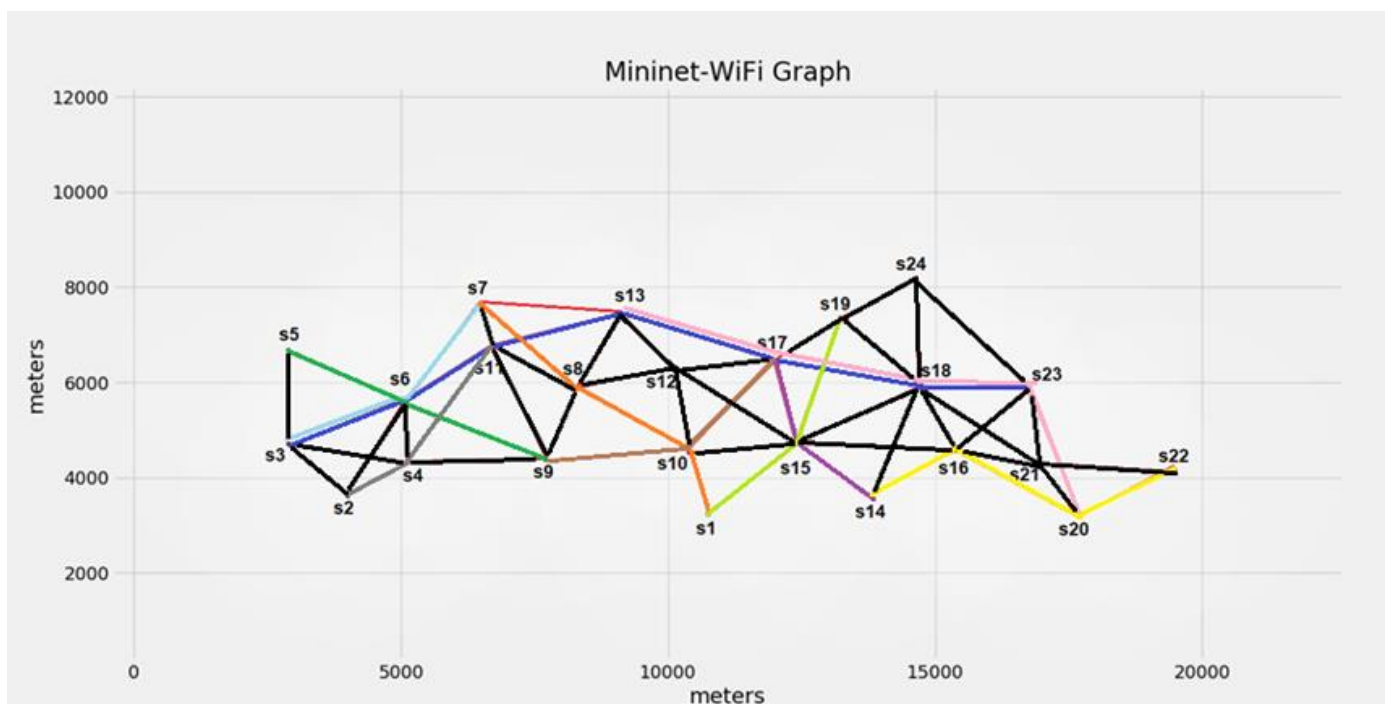


Figure 10 :The from s3 to s23 has blue colour, the other colors are simultaneous paths for different pairs of nodes in the network crossing the blue path. These paths have been produced by BATMAN protocol.

We chose a big flow in the network with many hops that cross through the network, from one edge to the other. That flow was the s3 – s23 (the blue line in figure 9 for OLSR and in figure 10 for BATMAN). The

other colour lines (paths) are the parallel flows. We ran iperf and iperf3 tests on it while we ran the parallel flows that were crossing through it. For every number of simultaneous flows and for every input rate from 8, 80, 800 Kbps and 1, 4 and 8 Mbps we measured the throughput for both protocols.

In the same way we measured the delay for both protocols by using the ping command *Appendix 3*.

## Evaluation of Results

For every edge node, we have found the path to all other edge nodes. Also, we have calculated the respective paths based on the Dijkstra algorithm, with weight on the distance between the nodes. We observed that both protocols are using minimum hop paths. That means that for each path they produce, there is not a path with less nodes. But it is not always the shortest path in distance. Based on the weighted Dijkstra algorithm, there are shortest paths. The reason that the two protocols choose different paths, is because they have different ways and metrics to produce them.

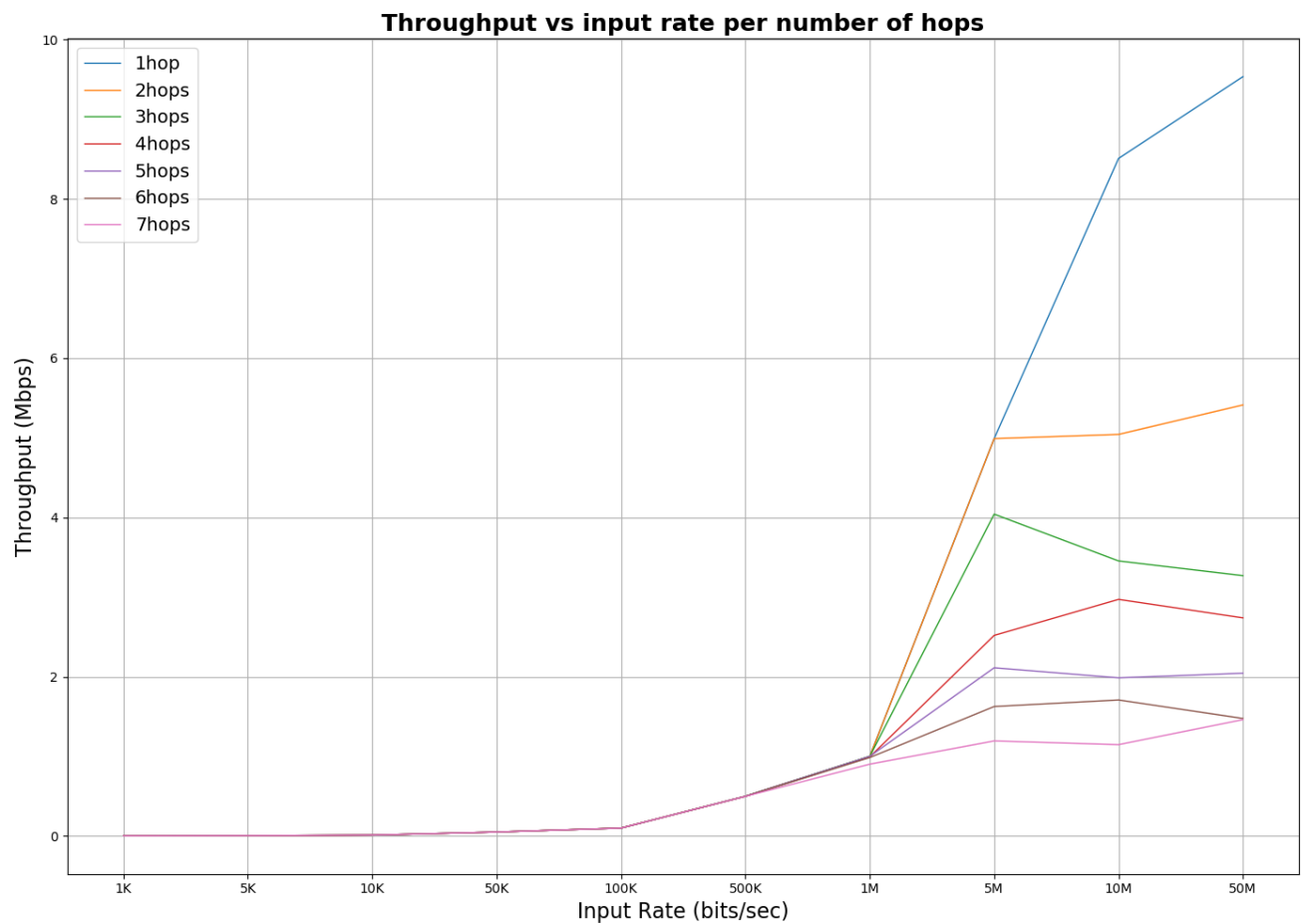


Figure 11: Average throughput of  $h$  hops versus input rate per number of  $h$  hops for OLSR protocol.



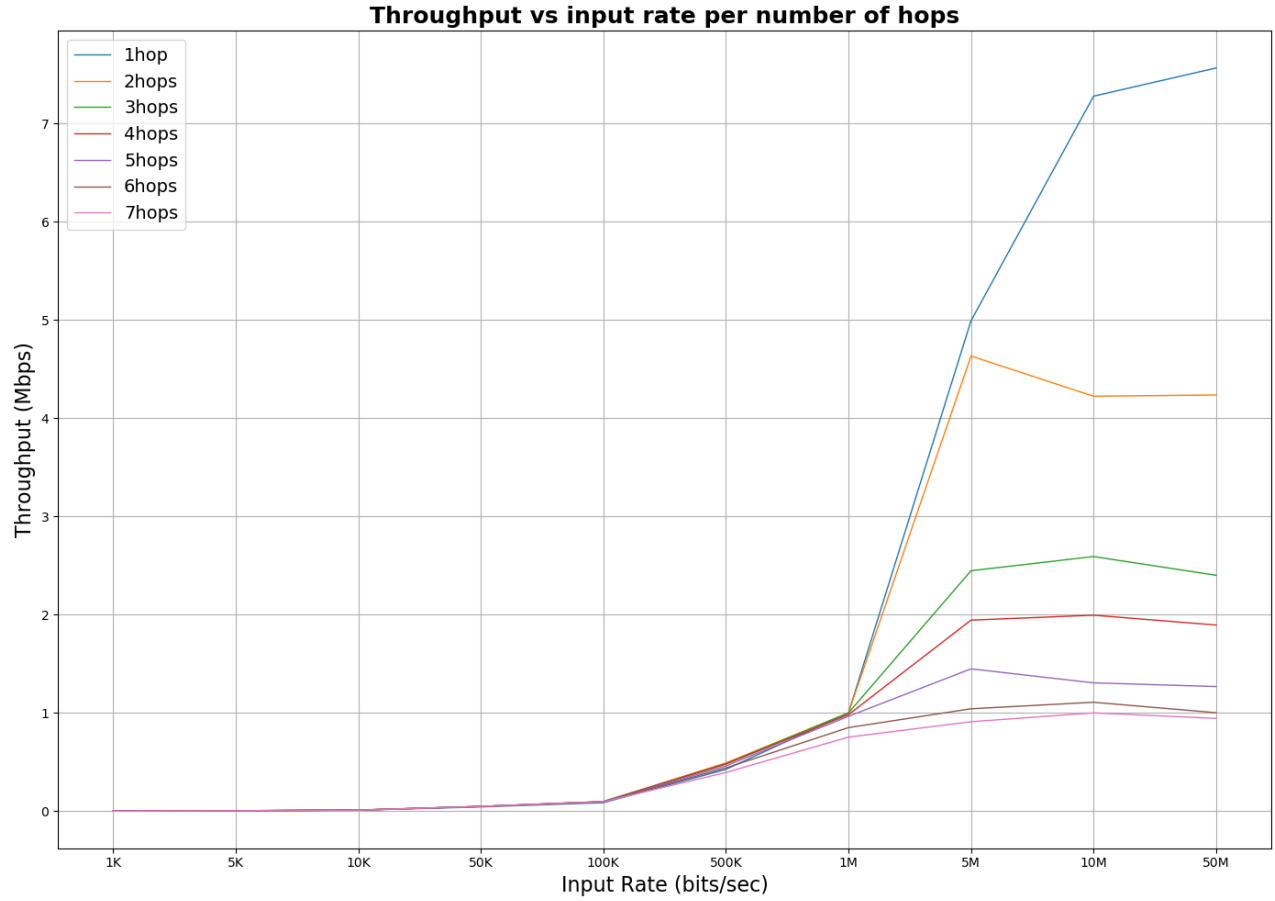


Figure 12: Average throughput of  $h$  hops versus input rate per number of  $h$  hops for BATMAN protocol.

As we can see in figures 12 and 13, for all the numbers of hops, OLSR has better values of throughput, but without significant difference. We see on the plots that the two protocols have the same behavior between hops. The distribution of throughput values among the number of hops are very similar. After 1 Mbps the values per hop are becoming very different, and they are approximately the same for higher values of the input rate as well. Also, throughput values for paths of 3 to 7 hops on both protocols are much closer among themselves than for paths of 1 and 2 hops.

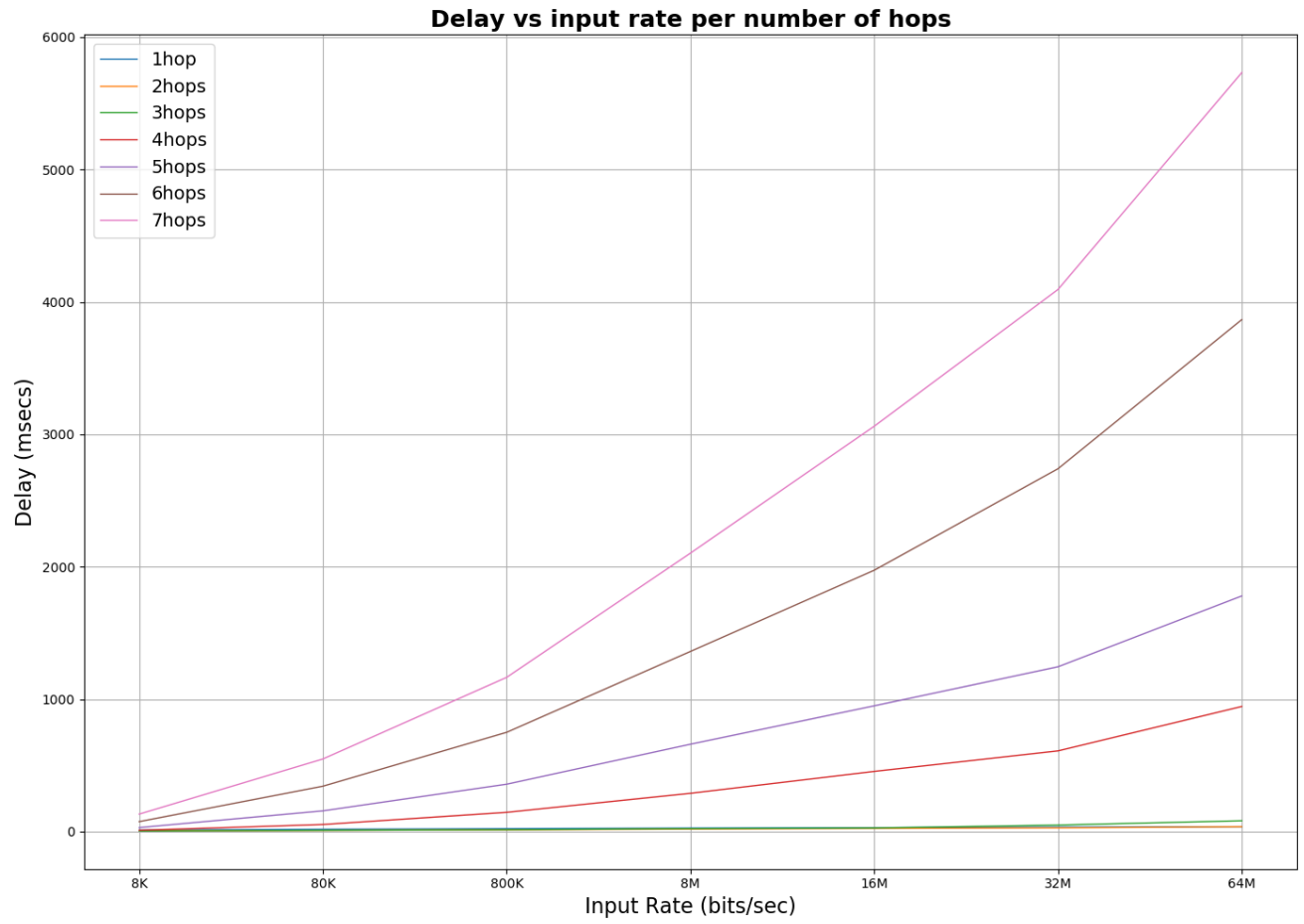


Figure 13 : Average delay of  $h$  hops versus input rate per number of  $h$  hops for OLSR protocol.

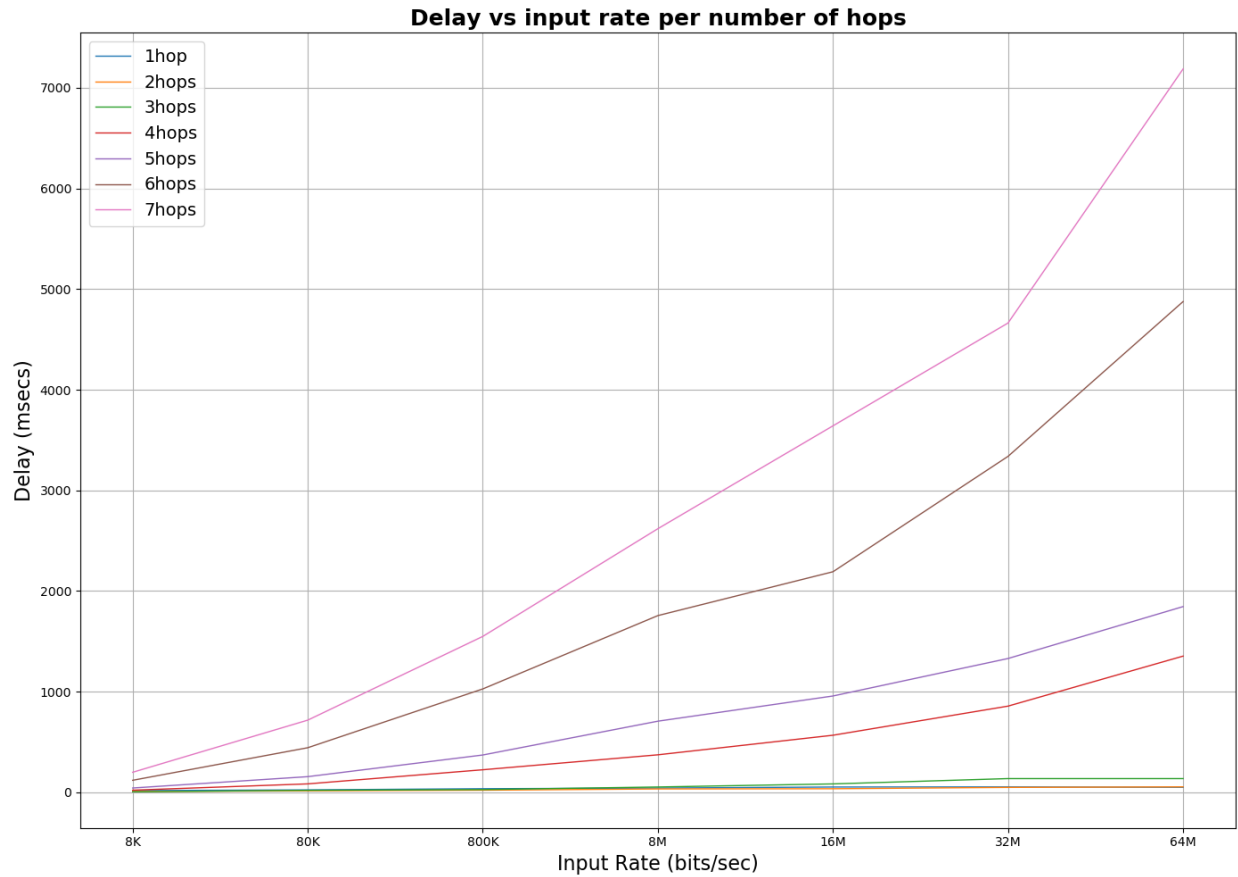


Figure 14 : Average delay of  $h$  hops versus input rate per number of  $h$  hops for BATMAN protocol.

In both plots we see that the delay increases as the number of hops increase, which is expected. The plots show BATMAN results in larger delays than OLSR for paths with any number of hops. We can see that as in the case of throughput, the way values differ in every number of hops for both protocols, are very similar.

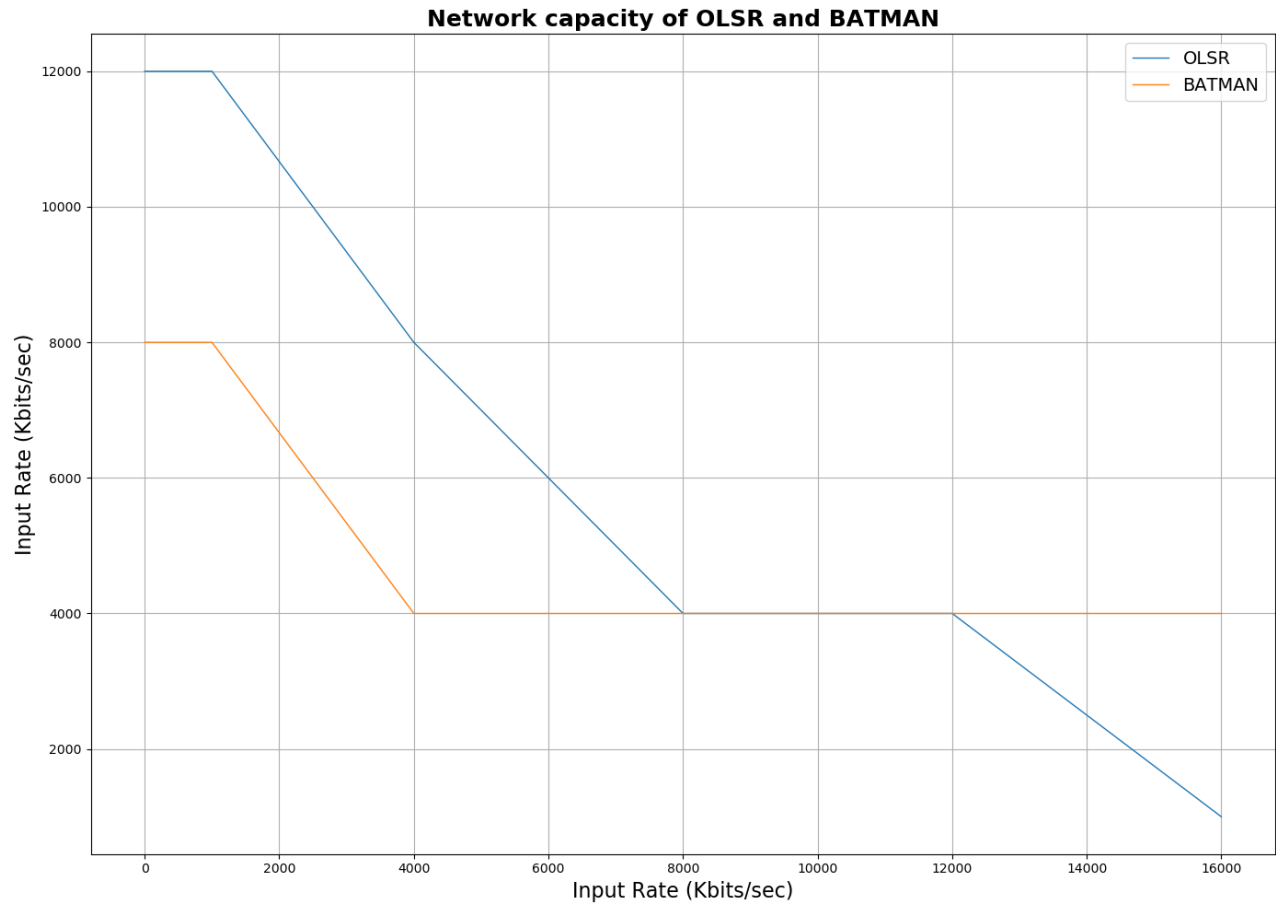


Figure 15 : Network capacity of OLSR and BATMAN

Based on the plot of figure 16, we see that OLSR has a larger network capacity region. BATMAN is having a greater value on the input rate of 16,000 Kbps, but overall OLSR has better results.

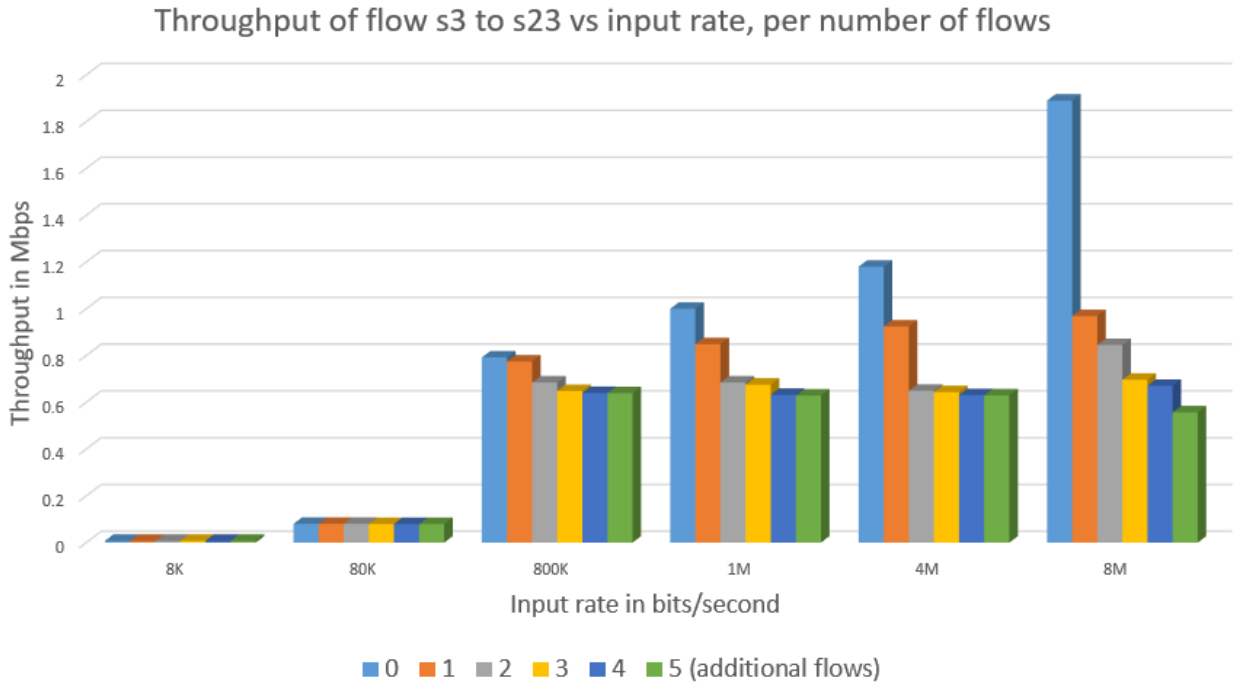


Figure 16: Throughput of flow s3 to s23 versus input rate, per number of additional flows for OLSR protocol

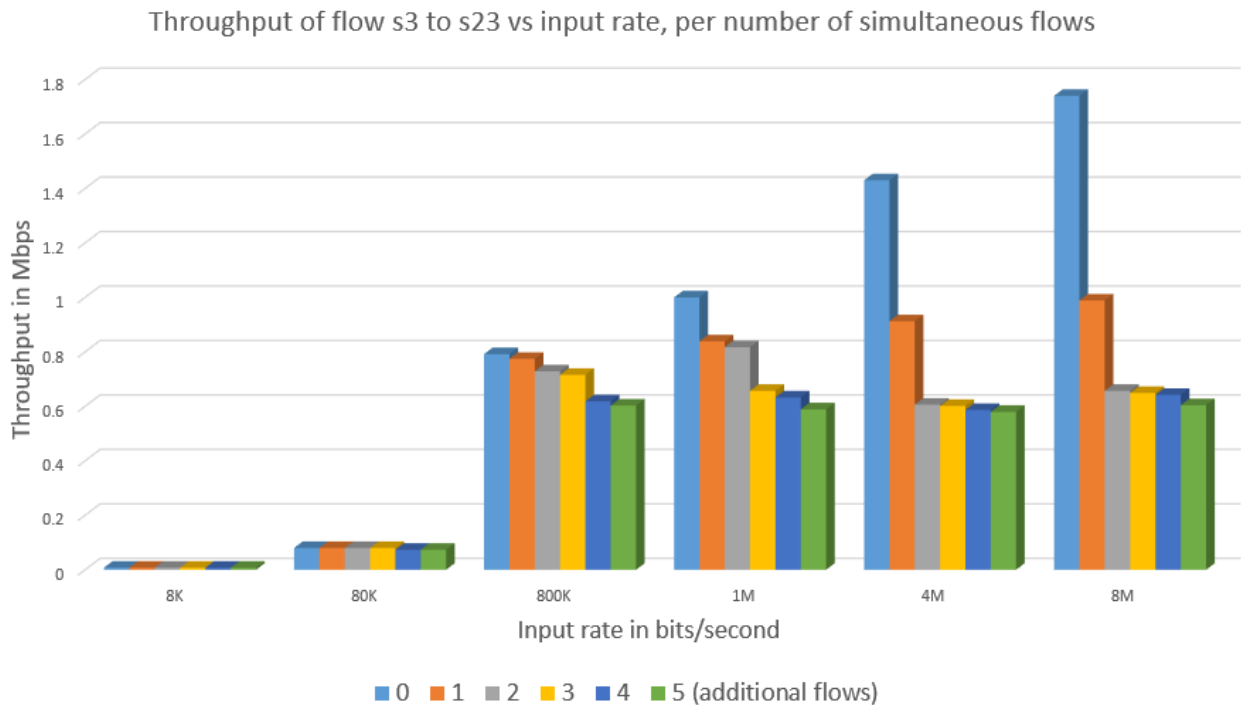


Figure 17 : Throughput of flow s3 to s23 versus input rate, per number of additional flows for BATMAN protocol.

On Figure 17 where we have the results of OLSR, we see that while the input rate increases, we do not see a significant drop to throughput from one flow to two flows in the network. We also observe that after two flows on the network, the throughput begins to stabilize.

On the contrary, on figure 18 where we have the results of BATMAN, we see a big decrease in throughput when transitioning from one concurrent flow to two concurrent flows in the network. Also, we see that the throughput begins to stabilize after three flows.

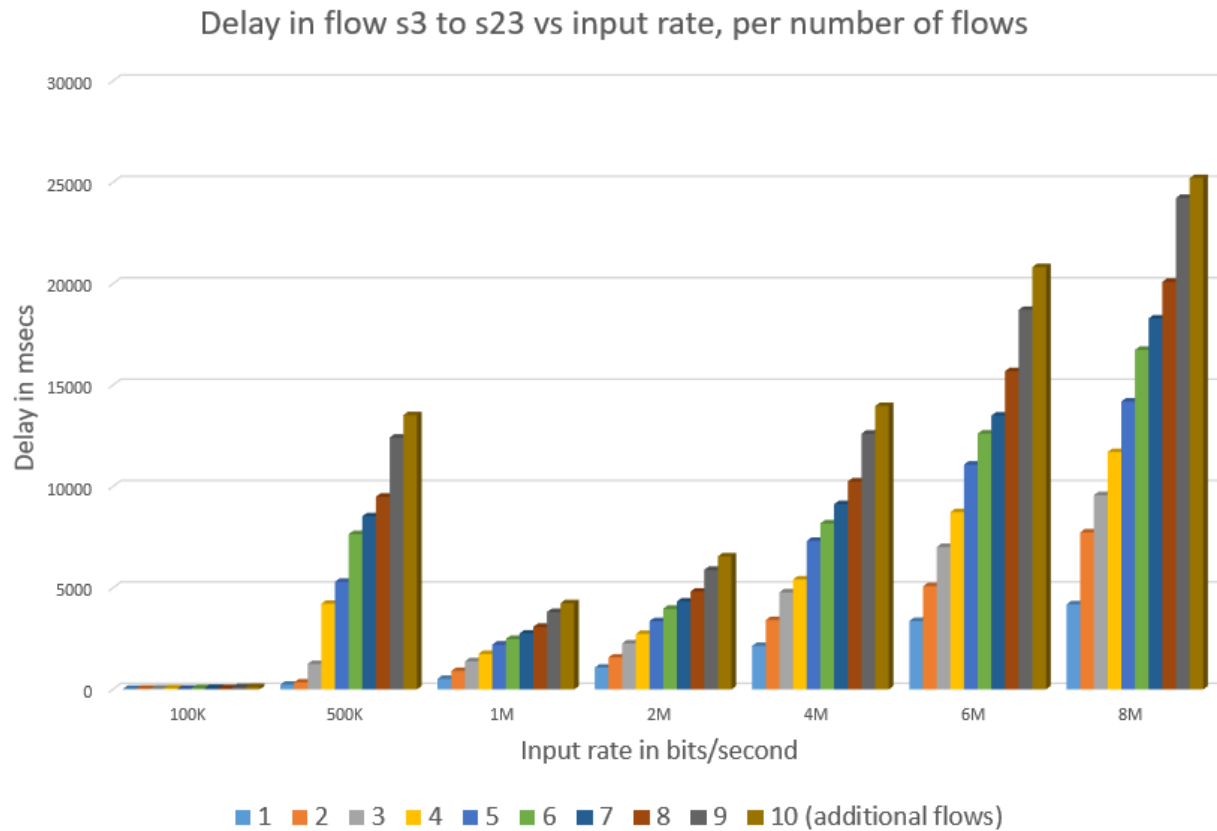


Figure 18 : Delay in flow s3 to s23 versus input rate, per number of additional flows for OLSR protocol.

On Figure 19 we see the delay results for OLSR. We see the delay increased in almost the same way in every number of flows in the network and for every input rate.

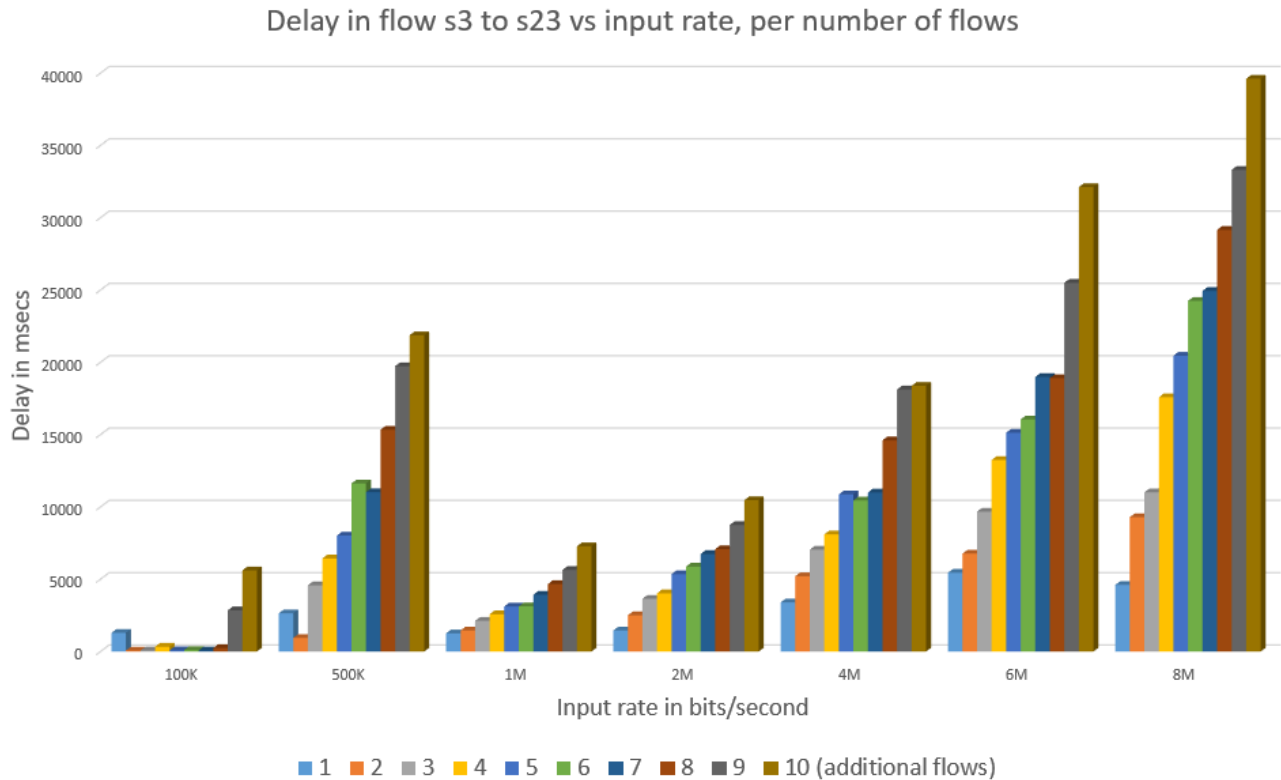


Figure 19 : Delay in flow s3 to s23 versus input rate, per number of additional flows for BATMAN protocol.

We see that there is a difference on the way the delay increases for every number of flows, between BATMAN and OLSR. In OLSR we see that the delay increases gradually, but here we have a different distribution.

## Challenges

We tried to run the network with the BABEL protocol as well without success. Unfortunately, Mininet WiFi does not support it properly. The only way to apply it on the network, was to run it as service in every station separately. While we were testing it, the protocol seemed to be working right at first. While the routing tables were being filled, we noticed that for some reason, some of the routes were being deleted. So, if a station could ping another station, it may could not do this again after a few seconds. For some reason it was so heavy to run that it was unusable and so we could not complete the test of BABEL protocol.

## Conclusion

Having run many tests on throughput and delay measures, we conclude that OLSR performs better than BATMAN advanced. With respect to the graph of the network, delay and throughput for alone or concurrent flows it is observed that

- Both protocols produce paths with the minimum possible number of hops, but not optimal based on the distance.

- OLSR has better values on throughput for all number of hops than BATMAN, without significant deviations.
- There are similar fluctuations in throughput and delay for all number of hops, between the two protocols.
- BATMAN has bigger delays for any number of hops than OLSR.
- OLSR has a better network capacity than BATMAN.
- For concurrent flows both protocols face congestion on the network after two flows.

## References

- [1] Optimized link state routing protocol for ad hoc networks.  
<https://ieeexplore.ieee.org/document/995315>
- [2] <https://www.open-mesh.org/projects/open-mesh/wiki>
- [3] <http://mininet.org/>
- [4] [https://en.wikipedia.org/wiki/Free-space\\_path\\_loss](https://en.wikipedia.org/wiki/Free-space_path_loss)
- [5] [https://en.wikipedia.org/wiki/Two-ray\\_ground-reflection\\_model](https://en.wikipedia.org/wiki/Two-ray_ground-reflection_model)
- [6] <https://github.com/ramonfontes>
- [7] [https://en.wikipedia.org/wiki/IEEE\\_802.11](https://en.wikipedia.org/wiki/IEEE_802.11)
- [8] <https://github.com/OLSR/OONF>
- [9] <https://tools.ietf.org/id/draft-funkfeuer-manet-olsrv2-etx-00.html>
- [10] <https://github.com/open-mesh-mirror/batman-adv>
- [11] [https://www.kernel.org/doc/html/latest/networking/mac80211\\_hwsim/mac80211\\_hwsim.html](https://www.kernel.org/doc/html/latest/networking/mac80211_hwsim/mac80211_hwsim.html)
- [12] <https://usermanual.wiki/Pdf/mininetwifidraftmanual.297704656/html#pdf>

## Appendix

1)

```
import sys
from mininet.log import setLogLevel, info
from mn_wifi.link import wmediumd, adhoc
from mn_wifi.cli import CLI
from mn_wifi.net import Mininet_wifi
from mn_wifi.wmediumdConnector import interference
def topology(args):
    "Create a network."
    net = Mininet_wifi(link=wmediumd, wmediumd_mode=interference)
    info("*** Creating nodes\n")
    kwargs = dict()
```



```

if '-a' in args:
    kwargs['range'] = 100

s1= net.addStation('s1', mac='00:00:00:00:00:01', position='10725,3225,0',
**kwargs)
s2 = net.addStation('s2', mac='00:00:00:00:00:02', position='3990,3630,0',
**kwargs)
s3 = net.addStation('s3', mac='00:00:00:00:00:03', position='2895,4710,0',
**kwargs)
s4 = net.addStation('s4', mac='00:00:00:00:00:04', position='5130,4365,0',
**kwargs)
s5 = net.addStation('s5', mac='00:00:00:00:00:05', position='2895,6690,0',
**kwargs)
s6 = net.addStation('s6', mac='00:00:00:00:00:06', position='5055,5625,0',
**kwargs)
s7 = net.addStation('s7', mac='00:00:00:00:00:07', position='6495,7680,0',
**kwargs)
s8 = net.addStation('s8', mac='00:00:00:00:00:08', position='8235,5910,0',
**kwargs)
s9 = net.addStation('s9', mac='00:00:00:00:00:09', position='7695,4380,0',
**kwargs)
s10 = net.addStation('s10', mac='00:00:00:00:00:10',
position='10380,4575,0', **kwargs)
s11 = net.addStation('s11', mac='00:00:00:00:00:11',
position='6690,6825,0', **kwargs)
s12 = net.addStation('s12', mac='00:00:00:00:00:12',
position='10140,6285,0', **kwargs)
s13 = net.addStation('s13', mac='00:00:00:00:00:13',
position='9105,7470,0', **kwargs)
s14 = net.addStation('s14', mac='00:00:00:00:00:14',
position='13800,3600,0', **kwargs)
s15 = net.addStation('s15', mac='00:00:00:00:00:15',
position='12405,4755,0', **kwargs)
s16 = net.addStation('s16', mac='00:00:00:00:00:16',
position='15390,4605,0', **kwargs)
s17 = net.addStation('s17', mac='00:00:00:00:00:17',
position='11955,6480,0', **kwargs)
s18 = net.addStation('s18', mac='00:00:00:00:00:18',
position='14730,5955,0', **kwargs)
s19 = net.addStation('s19', mac='00:00:00:00:00:19',
position='13245,7350,0', **kwargs)
s20 = net.addStation('s20', mac='00:00:00:00:00:20',
position='17685,3195,0', **kwargs)
s21 = net.addStation('s21', mac='00:00:00:00:00:21',
position='16980,4260,0', **kwargs)
s22 = net.addStation('s22', mac='00:00:00:00:00:22',
position='19470,4170,0', **kwargs)
s23 = net.addStation('s23', mac='00:00:00:00:00:23',
position='16725,5910,0', **kwargs)
s24 = net.addStation('s24', mac='00:00:00:00:00:24',
position='14625,8175,0', **kwargs)
net.setPropagationModel(model="logDistance", exp=1.8)
info("*** Configuring wifi nodes\n")
net.configureWifiNodes()

info("*** Creating links\n")

```

```

protocols = ['batman_adv','olsrd']
kwargs = dict()
for proto in args:
    if proto in protocols:
        kwargs['proto'] = proto

    net.addLink(s1, cls=adhoc, intf='s1-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s2, cls=adhoc, intf='s2-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s3, cls=adhoc, intf='s3-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s4, cls=adhoc, intf='s4-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s5, cls=adhoc, intf='s5-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s6, cls=adhoc, intf='s6-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s7, cls=adhoc, intf='s7-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s8, cls=adhoc, intf='s8-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s9, cls=adhoc, intf='s9-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s10, cls=adhoc, intf='s10-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s11, cls=adhoc, intf='s11-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s12, cls=adhoc, intf='s12-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s13, cls=adhoc, intf='s13-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s14, cls=adhoc, intf='s14-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s15, cls=adhoc, intf='s15-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s16, cls=adhoc, intf='s16-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s17, cls=adhoc, intf='s17-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s18, cls=adhoc, intf='s18-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s19, cls=adhoc, intf='s19-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s20, cls=adhoc, intf='s20-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s21, cls=adhoc, intf='s21-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s22, cls=adhoc, intf='s22-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s23, cls=adhoc, intf='s23-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)
    net.addLink(s24, cls=adhoc, intf='s24-wlan0', ssid='adhocNet', mode='g',
channel=5, **kwargs)

net.plotGraph(max_x=23000, max_y=12000)

```

```

info("*** Starting network\n")
net.build()

s1.setTxPower(2, intf='s1-wlan0')
s2.setTxPower(2, intf='s2-wlan0')
s3.setTxPower(2, intf='s3-wlan0')
s4.setTxPower(2, intf='s4-wlan0')
s5.setTxPower(2, intf='s5-wlan0')
s6.setTxPower(2, intf='s6-wlan0')
s7.setTxPower(2, intf='s7-wlan0')
s8.setTxPower(2, intf='s8-wlan0')
s9.setTxPower(2, intf='s9-wlan0')
s10.setTxPower(2, intf='s10-wlan0')
s11.setTxPower(2, intf='s11-wlan0')
s12.setTxPower(2, intf='s12-wlan0')
s13.setTxPower(2, intf='s13-wlan0')
s14.setTxPower(2, intf='s14-wlan0')
s15.setTxPower(2, intf='s15-wlan0')
s16.setTxPower(2, intf='s16-wlan0')
s17.setTxPower(2, intf='s17-wlan0')
s18.setTxPower(2, intf='s18-wlan0')
s19.setTxPower(2, intf='s19-wlan0')
s20.setTxPower(2, intf='s20-wlan0')
s21.setTxPower(2, intf='s21-wlan0')
s22.setTxPower(2, intf='s22-wlan0')
s23.setTxPower(2, intf='s23-wlan0')
s24.setTxPower(2, intf='s24-wlan0')

info("*** Running CLI\n")
CLI(net)

info("*** Stopping network\n")
net.stop()

if __name__ == '__main__':
    setLogLevel('info')
    topology(sys.argv)

```

2)

```

#!/bin/bash

k="K"
m="M"
oneh=100

Kcommand=$(iperf -c 192.168.123.$1 -u -p 5566 -b 8$k -t 100 -l 1000 | grep -i
"% " | awk '{for (I=1;I<NF;I++) if ($I == "KBytes" || $I == "MBytes") print
$(I+1)}' | tr -d '\n')
echo $Kcommand
if [ $Kcommand -gt $oneh ]; then
    echo "greater 2"

```

```

    calc $Kcommand/1048576 | tr -d '\n' | tr -d '\t'>> results/batman/iperf3-
23/iperf3-23.txt
    sleep 1
    echo -n -e '\t' >> results/batman/iperf3-23/iperf3-23.txt
else
    echo "no 2"
    calc $Kcommand/1024 | tr -d '\n' | tr -d '\t'>> results/batman/iperf3-
23/iperf3-23.txt
    sleep 1
    echo -n -e '\t' >> results/batman/iperf3-23/iperf3-23.txt
fi

for j in 80
do
    if [ $1 -ne $j ]
    then
        Kcommand=$(iperf -c 192.168.123.$1 -u -p 5566 -b 1$k -t 30 | grep -i
"% " | awk '{for (I=1;I<NF;I++) if ($I == "KBytes" || $I == "MBytes") print
$(I+1)}' | tr -d '\n')

        if [ $Kcommand -gt $oneh ]; then
            echo $Kcommand
            echo "greater"
            calc $Kcommand/1048576 | tr -d '\n' | tr -d '\t'>>
results/batman/iperf3-23/iperf3-23.txt
            sleep 0.1
            echo -n -e '\t' >> results/batman/iperf3-23/iperf3-23.txt
        else
            echo "no"
            calc $Kcommand/1024 | tr -d '\n' | tr -d '\t'>>
results/batman/iperf3-23/iperf3-23.txt
            sleep 0.1
            echo -n -e '\t' >> results/batman/iperf3-23/iperf3-23.txt
        fi
    fi
done
echo >> results/batman/iperf/iperftest$1.txt

for i in 800
do
    Kcommand=$(iperf3 -c 192.168.123.$1 -u -p 5565 -b $i$k -t 100 | grep -i
"% " | awk '{for (I=1;I<NF;I++) if ($I == "KBytes" || $I == "MBytes") print
$(I+1)}' | tr -d '\n')
    echo $Kcommand
    calc $Kcommand/1024 | tr -d '\n' | tr -d '\t'>> results/batman/iperf3-
23/iperf3-23.txt
    sleep 1
    echo -n -e '\t' >> results/batman/iperf3-23/iperf3-23.txt

for l in 1 4 8
do
    Mcommand=$(iperf3 -c 10.0.0.$1 -u -p 5565 -b $l$m -t 100 | grep -i "% " |
awk '{for (I=1;I<NF;I++) if ($I == "KBytes" || $I == "MBytes") print $(I+1)}'
| tr -d '\n')
    echo $Mcommand
    if [ $Mcommand -gt $oneh ]; then
        echo "greater 3"
    fi
done

```

```

        calc $Mcommand/1024 | tr -d '\n' | tr -d '\t'>>
results/batman/iperf3-23/iperf3-23.txt
        sleep 1
        echo -n -e '\t' >> results/batman/iperf3-23/iperf3-23.txt
    else
        echo "no 3"
        calc $Mcommand/1 | tr -d '\n' | tr -d '\t'>> results/batman/iperf3-
23/iperf3-23.txt
        sleep 1
        echo -n -e '\t' >> results/batman/iperf3-23/iperf3-23.txt
    fi
done

```

3)

```

#!/bin/bash

result=$(ping -c $3 192.168.123.23 -s $1 -i $2)
sleep 0.1

echo -n "$result" | grep -i "rtt" | cut -d "=" -f2 | cut -d "/" -f2 | tr -d
'\n' >> results/batman/delay3-23.txt
echo -n "\t" >> results/batman/delay3-23.txt

echo -n "$result" | grep -i "loss" | cut -d "," -f3 | cut -d "%" -f1 | tr -d
'\n' >> results/batman/loss3-23.txt
echo -n "\t" >> results/batman/loss3-23.txt

```

4)

```

#!/bin/bash

for j in 5 3 2 1 14 20 22 23 24 13 7
do
    if [ "$1" -ne "$j" ]
    then
        for attr in "102 0.1" "102 0.01" "102 0.001" "1024 0.001" "2048
0.001" "4096 0.001" "8192 0.001"
        do
            s=$(echo $attr | cut -d' ' -f1)
            i=$(echo $attr | cut -d' ' -f2)
            result=$(ping -c 360 192.168.123.$j -s $s -i $i | grep -i "rtt" |
cut -d "=" -f2 | cut -d "/" -f2 | tr -d '\n')
            sleep 0.1
            echo -n $result >> results/batman/rtt/avgrtt$1.txt
            echo -n "\t" >> results/batman/rtt/avgrtt$1.txt
        done
        echo >> results/batman/rtt/avgrtt$1.txt
    fi
done

```

5)

```
#!/bin/bash

echo "Trace for $1 -----" >> results/batman/traceResults.txt

for i in 5 3 2 1 14 20 22 23 24 13 7
do
    batctl traceroute 192.168.123.$i | cut -f3 -d " " >>
results/batman/traceResults.txt
    sleep 0.3
done
```