7COM1076-0901-2024 - Wireless, Mobile and Multimedia Networking

22076082

December 6, 2024

Software Defined Networking(SDN) emulation with Mininet and OpenFlow

University of Hertfordshire
Hatfield

Contents

List of Tables

List of Figures

Abstract

The world has been considered a digital society with the intervention of the Internet. These computer networks form a strong foundation in the manner we access information, communicate, and handle data, providing convenience in various aspects of life. This technology relies strongly on the operations of traditional IP networks and despite being widely adopted, they do get complex and hard to manage in terms of configuration according to predefined policies and reconfiguration in response to load, faults, and changes. On top of that, conventional networks are vertically integrated(the control plane and data plane are bundled together). Emerging technologies such as Software Defined Networking (SDN) provides a convenient approach in handling computer networks. SDN brings a proposed paradigm which changes state of affairs by separating the network's control logic from the data plane. This approach enables centralisation of network control and ability to program the network improving network availability, scalability, and management. This report presents a practical survey on SDN. It starts by utilising concepts of SDN in both a wireless and wired network scenario, an Ad-Hoc Network for emergency scenario, and multi-cast video streaming. We measure and test out performance metrics such as throughput and jitter for communications under Session Initiation Protocol (SIP), connectionless communication in server and client scenario. All activities are carried out on a single machine running Linux with underlying applications such as Mininet, Open Network Operating System(ONOS) and Wireshark.

1 Introduction

Since the conception and realisation of the Advanced Research Projects Agency Networks(ARPANET) [5], developments and innovations in computer networks has seen a significant growth in size and requirements and navigating traditional network switches has become a challenge. In traditional networks, the control plane operation has a distributed infrastructure that requires protocols such as OSPF, STP, EIGRP, to operate independently on network devices. The forwarding decision which is the process of determining how to send a packet from one device (such as a router or switch) to another device within a network, based on certain criteria is performed by these network devices. This decision is made based on the information available in the routing table (for routers) or the switching table (MAC table) (for switches). It is accepted that the network devices connect but there is no centralised machine to manage or summarise the whole network [2]. Also, the distributed control and transport protocols running inside routers and switches makes it complex to express desired high-level network policies since network operators must configure each individual network device separately [3]. This main issue is rectified by SDN introducing the needed network control, flexibility, and programmability.

1.1 Software-Defined Networking (SDN) and OpenFlow

Software-Defined Networking (SDN) is a computer technology that brings a change to the limitations of current computer networks infrastructure. SDN [4] decouples the control and data plane of a network leaving the switch with the simple function of forwarding packets based on a set of rules. By doing so, it breaks the vertical integration of conventional networks as the control logic is separated from the data plane i.e. the underlying switches and routers that forward the traffic [3]. To make possible the communication between the controller and the network devices, OpenFlow [4] is used in conjunction with SDN. OpenFlow standardises the communication between the switches and the software-based controller in an SDN architecture. Researchers found it difficult to test out new ideas in current hardware because the source code of the software running on the network device cannot be modified making the infrastructure 'rigid'. As a result, a standardised protocol(eg. OpenFlow) to control flow table of switches through software was provided by identifying common features in the flow tables of the ethernet switches [4].
The figures below show a comparison of the traditional network and SDN network and an overview of the SDN architecture.

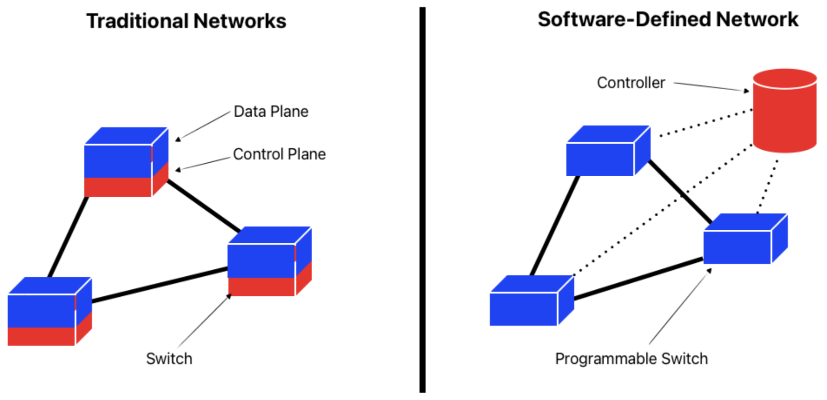


Figure 1: Traditional Networks vs SDN Networks

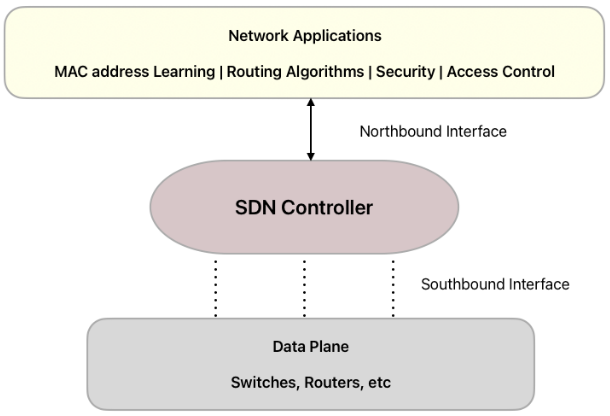


Figure 2: SDN Architecture

The underlying network applications or packages which are the softwares for the logical operations within the network are installed and activated on the SDN controller. These applications are known as the Northbound APIs or interface as shown in fig.2. The Southbound interface consist the protocols that facilitates communication between the remote SDN controller and the network devices, for example the OpenFlow [6] protocol. Examples of such network applications can be STP, routing algorithms, and access control and examples of a SDN controller can be the ONOS controller. The prospect of such architecture is noticed in terms of ease of network management, programmability and openness to innovations and virtualisation.

1.2 Mininet

For the scope of this paper, we aim to determine and investigate the convenience of SDN networks in a variety of network scenarios and requirements. To test out the efficiency, reliability, cost, and flexibility of the SDN approach in our network designs, the experiment will be carried out on a network emulators known as Mininet. It is one of the many network simulation tools(another example is OMNETT++) that have been developed to virtualise and test network performance [2, 6].

Mininet [1] provides a system that allows rapid prototyping of large networks and creates scalable software-defined networks using lightweight virtualisation mechanism. This alleviates the downside of conventional networking by providing a centralised view of the network and paves the way for more controllability and managing how networks should operate regardless of their size or their complexity.

Since research on this topic is still in progress, there are not many devices such as routers and switches that implement SDN functionalities, moreover, the existing ones are very expensive. Thus, in order to enable researchers to perform experiments and test novel features of this new paradigm in practice at a low financial cost, one solution is to use virtual network emulators such as Mininet. It creates SDN elements, customise them, and share them among other networks and perform interactions [1].

1.3 Emulation environment specifications

For this experiment, we utilise a microcomputer iMac with the following specifications: Processor 3 GHz 6-Core Intel Core i5, 16GB of ram, running the macOS 15.1, and VirtualBox Oracle VM version 7.1.2. In this microcomputer, under the management of VirtualBox, we installed the following guest operating systems: Mininet emulator version 2.0 on Linux operating system Ubuntu 64bits with 4GB of RAM; ONOS controller version ??.

2 Task 1 - WiFi Network Simulation

In this task, we emulate a wireless network designed for a floor in the new building. For this purpose, we emulate 3 stations and 5 access points. The stations may represent a smart hand-held device which can vary from smartphone to a laptop, UE or to any WiFi compatible device. The stations carry a Class C private IP address of the same network. Access points are connected using a physical link facilitating a linear topology. The new building would create a minimalistic noise threshold of -91dBm. The access points are positioned strategically within the floor to make space for a signal dead-zone(red-spot) and 3 stations will be in mobility state to emulate real-life network scenario.
A python script will contain the code for this configuration and upon completion, we start the network using Mininet on a Linux terminal and view the network emulation on the Mininet-GUI feature while having to observe access point association, full connectivity between all nodes in the network on the Mininet CLI, and briefly understand how communication channels are used in a wireless network.

The figures below show the outline of the floor-plan in context and position of APs and stations in the design.

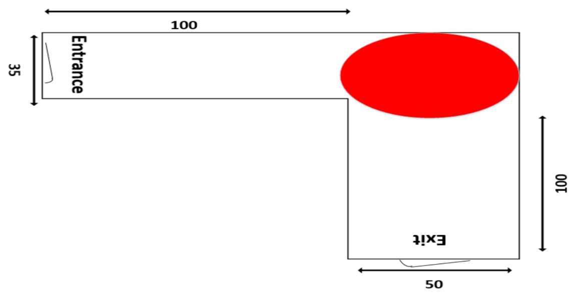


Figure 3: Floor-plan in context

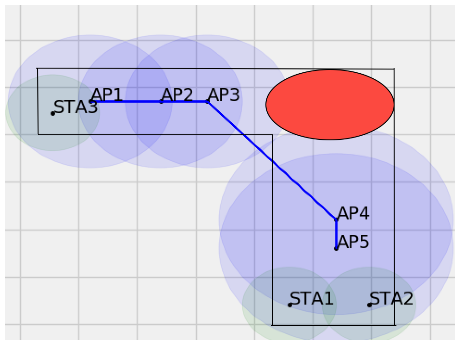


Figure 4: Floor-plan with infrastructure

2.1 Design and Configuration

Both figures show the outline of the floor-plan in context with the dimensions included on fig. 3 while fig. 4 is a snapshot of the Mininet GUI with the positions of the stations and access points and the range of coverage of all devices. The 'red-spotted' zone as seen remains free of WiFi signals as required in the design. Table 1 contains the details of stations and access points which are used for the network configuration in a python script. All three stations share the same network address and have full authenticated access to each access points.
The access points are positioned strategically and linked serially to ensure the required zones within the

floor have strong WiFi frequency coverage and support full connectivity of the stations. To emulate real networking scenario, mobility is included for the stations, details are shown in Table 2. Mobility attributes in terms of velocity or speed of motion for each station are specified. The mobility attributes are added to the python code for the network configuration which is captured in the appendix section of this document.

DEVICE	MAC	IPv4	(x,y)	SSID	PASSWORD	RANGE	CHANNEL
STA1	00:00:00:00:00:10	192.168.50.11/24	15,115	n/a	n/a	20	n/a
STA2	00:00:00:00:00:11	192.168.50.12/24	20,130	n/a	n/a	20	n/a
STA3	00:00:00:00:00:12	192.168.50.13/24	140,10	n/a	n/a	20	n/a
AP1	00:00:00:00:00:00	n/a	30,117.5	AP1	n/a	35	1
AP2	00:00:00:00:00:01	n/a	60,117.5	AP2	n/a	35	1
AP3	00:00:00:00:00:02	n/a	80,117.5	AP3	n/a	35	1
AP4	00:00:00:00:00:03	n/a	135,55	AP4	n/a	50	1
AP5	00:00:00:00:00:04	n/a	135,40	AP5	n/a	50	1

Table 1: Details of Stations and Access Points

DEVICE	START LOCATION	END LOCATION	START-STOP TIME	MOVING SPEED(min-max)
STA1	15,115	115,10	10s-20s	`min_v=1, max_v=5`
STA2	20,130	150,10	30s-60s	`min_v=5, max_v=10`
STA3	140,10	15,120	25s-60s	`min_v=2, max_v=7`

Table 2: Mobility of the stations

2.2 Results and Analysis

Commencing the test, we prepare the python script for the network topology with all necessary libraries then run it with Mininet on the Linux terminal. The command `sudo ./<pycode filename>` is used to start the network using the configuration from the python script on Mininet. Mininet creates the SDN elements such as the controller, stations, and access points and sets up the topology as described in the python script.

The controller used in this task is a default one used by Mininet since it has not been specified. The access points are started and operated based on instructions from the controller. Fig. 5 & 6 show the Mininet GUI view of the topology prior mobility and after mobility respectively. All stations are within strong WiFi signal coverage and successfully communicate with each other as shown in Fig. 8.

The `ping -c 3` command was utilised to send out 3 Internet Control Message Protocol(ICMP) [7] packets between specified stations to determine full connectivity within the network. This uses a series of request & reply messages to establish communication between stations. The results show all 3 packets were transmitted and received successfully, no packet loss, and took an average time of 2000 milliseconds for all communications which verifies the optimal performance of the network.

Prior mobility, we see that stations 1 & 2 would likely be associated with AP1 and station 3 with AP5 due to their close proximity to these APs but when in mobility state, the stations will lose association with access points, go through a handoff operation, and get associated with the access point in close proximity at the end of mobility state. To confirm this, we use the `<station name> iwconfig` command on the Mininet CLI to pull the wireless interface information for each station as illustrated in Fig. 7. This same activity could be carried out on each station directly when accessed with the `xterm <station name>` command on Mininet CLI.

Briefly, wireless communication is based on digital signals that are transmitted from one end and received at another end over the radio frequency spectrum. Medium Access Control(MAC) [7] is a mechanism devised to management how these radio channels are accessed by wireless capable devices for a wireless communication and have two main techniques; Contention-based & Contention-free MAC.

At the end of this experiment, we experience the sort of flexibility SDN brings into wireless network configuration and management, cost of deployment, quick response time, and overall summary of the network compared to the conventional methods of networking.

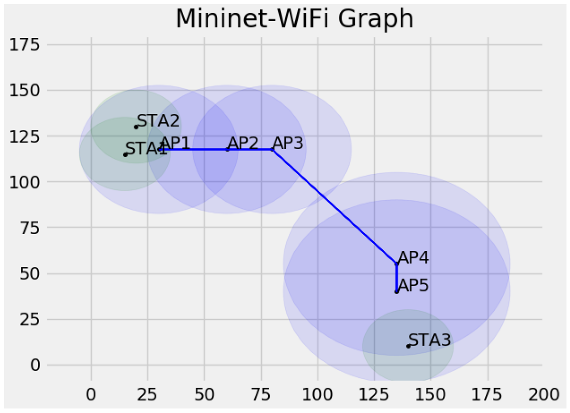


Figure 5: Prior Mobility

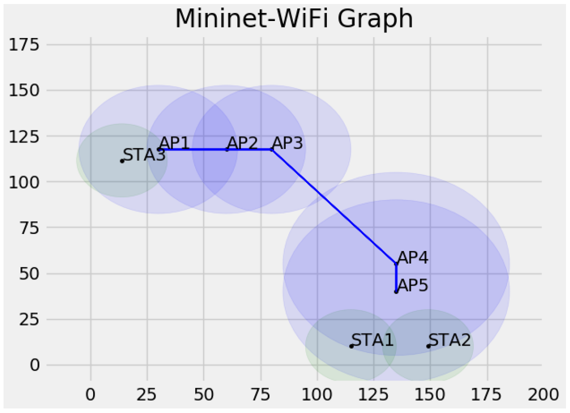
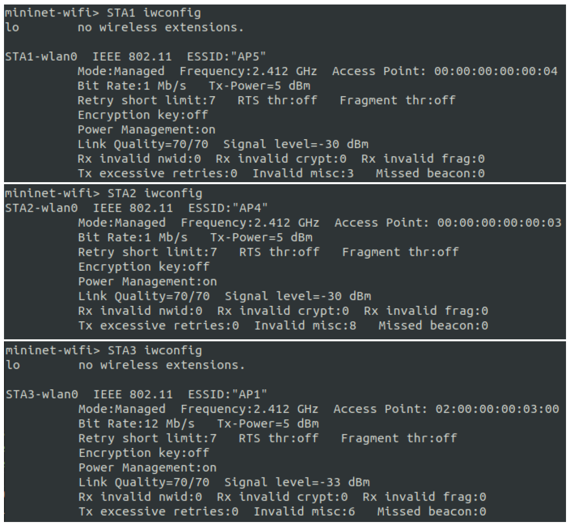


Figure 6: After Mobility

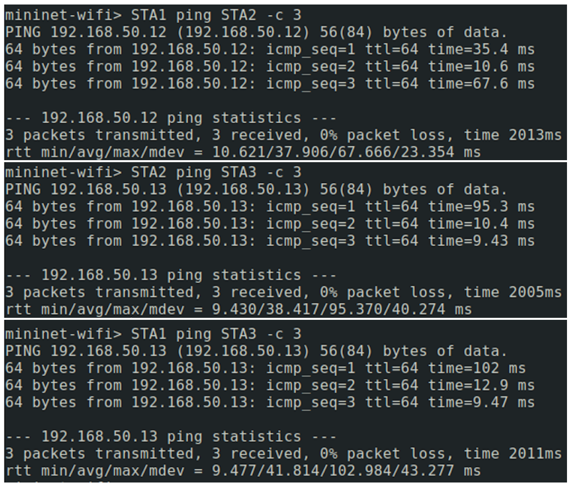


```
mininet-wifi> STA1 iwconfig
lo        no wireless extensions.

STA1-wlan0  IEEE 802.11  ESSID:"AP5"
          Mode:Managed  Frequency:2.412 GHz  Access Point: 00:00:00:00:00:04
          Bit Rate:1 Mb/s   Tx-Power=5 dBm
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Encryption key:off
          Power Management:on
          Link Quality=70/70  Signal level=-30 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:3   Missed beacon:0
```

```
mininet-wifi> STA2 iwconfig
STA2-wlan0  IEEE 802.11  ESSID:"AP4"
          Mode:Managed  Frequency:2.412 GHz  Access Point: 00:00:00:00:00:03
          Bit Rate:1 Mb/s   Tx-Power=5 dBm
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Encryption key:off
          Power Management:on
          Link Quality=70/70  Signal level=-30 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:8   Missed beacon:0
```

```
mininet-wifi> STA3 iwconfig
lo        no wireless extensions.

STA3-wlan0  IEEE 802.11  ESSID:"AP1"
          Mode:Managed  Frequency:2.412 GHz  Access Point: 02:00:00:00:03:00
          Bit Rate:12 Mb/s   Tx-Power=5 dBm
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Encryption key:off
          Power Management:on
          Link Quality=70/70  Signal level=-33 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:6   Missed beacon:0
```

Figure 7: APs connected after mobility



```
mininet-wifi> STA1 ping STA2 -c 3
PING 192.168.50.12 (192.168.50.12) 56(84) bytes of data.
64 bytes from 192.168.50.12: icmp_seq=1 ttl=64 time=35.4 ms
64 bytes from 192.168.50.12: icmp_seq=2 ttl=64 time=10.6 ms
64 bytes from 192.168.50.12: icmp_seq=3 ttl=64 time=67.6 ms

--- 192.168.50.12 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2013ms
rtt min/avg/max/mdev = 10.621/37.906/67.666/23.354 ms
```

```
mininet-wifi> STA2 ping STA3 -c 3
PING 192.168.50.13 (192.168.50.13) 56(84) bytes of data.
64 bytes from 192.168.50.13: icmp_seq=1 ttl=64 time=95.3 ms
64 bytes from 192.168.50.13: icmp_seq=2 ttl=64 time=10.4 ms
64 bytes from 192.168.50.13: icmp_seq=3 ttl=64 time=9.43 ms

--- 192.168.50.13 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 9.430/38.417/95.370/40.274 ms
```

```
mininet-wifi> STA1 ping STA3 -c 3
PING 192.168.50.13 (192.168.50.13) 56(84) bytes of data.
64 bytes from 192.168.50.13: icmp_seq=1 ttl=64 time=102 ms
64 bytes from 192.168.50.13: icmp_seq=2 ttl=64 time=12.9 ms
64 bytes from 192.168.50.13: icmp_seq=3 ttl=64 time=9.47 ms

--- 192.168.50.13 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2011ms
rtt min/avg/max/mdev = 9.477/41.814/102.984/43.277 ms
```

Figure 8: Successful Connectivity

3 Task 2 - Adhoc Network Simulation

In this task, we emulate an AdHoc network scenario at an emergency gathering car park at the new building when response units will communicate via AdHoc services. We evaluate some adhoc protocols to find which is best for the given situation. For this purpose, 3 stations are emulated and the adhoc protocols which will be tested are the 'batmand', 'batman_adv', and 'olsrd' protocols. ICMP streams are initiated between the stations closest to each other and 3 VoIP for each adhoc protocol tested. For analysis of connection and network performance, we use a network protocol analytic tool known as Wireshark to inspect and capture traffic of a network in real-time. It shows details about protocols and can be used to generate a graphical or visualised view of network traffic.

3.1 AdHoc Networks

AdHoc [7] networks are quick to deploy and dynamic form of networks that can be designed for a particular use-case. It is a wireless network however individual nodes can act as hops for forwarding packets within the local network. AdHoc networks are less reliant on infrastructure and efficient in setting up wireless communications in cases of emergencies.

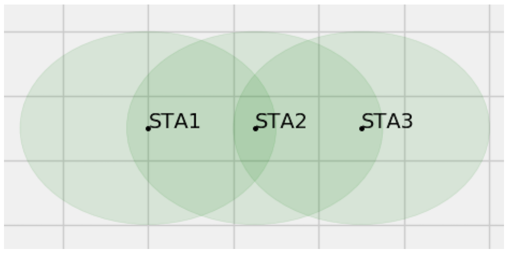


Figure 9: Adhoc Network with 3 stations

3.2 Design and Configuration

A python script is used for the configuration of the network where all 3 stations are created and configured with a protocol to support communication over the AdHoc network. Fig. 9 depicts the floor plan of the new building and emergency gathering car park with 3 stations in close proximity. Table 3 contains the station names, IPv6 and MAC addresses, their positions, range of radio frequency coverage, and wireless configuration with details about those fields below;

- A_HEIGHT: the height of the antenna typically in physical distance(metres) between the antenna and the ground or reference point
- A_GAIN: the measure of how good an antenna converts input power into radio waves in a specific direction. Usually expressed in decibels (dB)
- SSID: Service Set Identifier, unique name used to identify a wireless local area network(WLAN)
- HT_CAP.: High Throughput Capability, relates to 802.11n/ac standards to support high data rate and better performance

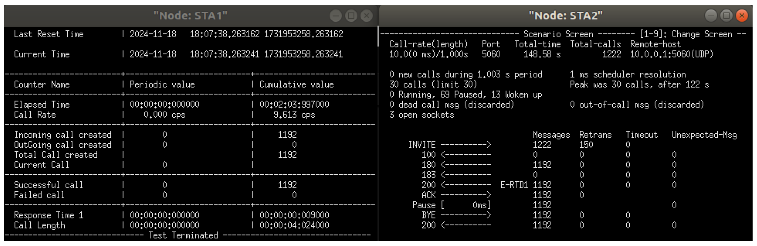
NAME	IPv6	MAC	POSITION	RANGE	A_HEIGHT	A_GAIN	SSID	HT_CAP
STA1	2024::11	00:00:00:00:01:11	20,10,0	30	1	5	adhocUH	HT40+
STA2	2024::12	00:00:00:00:01:12	45,10,0	30	2	6	adhocUH	HT40+
STA3	2024::13	00:00:00:00:01:13	70,10,0	30	3	7	adhocUH	HT40+

Table 3: Adhoc stations configuration details

3.3 Results and Analysis

After completion of the script, the network is started on the Linux terminal with the `sudo ./<python filename>` command. For this experiment, we aim to establish a successful connection between the closest stations in our setup i.e. between STA1 and STA2 and test out a VoIP connection between the stations under three different adhoc protocols.
The Session Initiation Protocol(SIP) which is a protocol used to establish, modify, and terminate multimedia sessions over IP networks will be initiated on one station as a server and on the other as client. The SIP commands are run on each stations directly accessing them with `xterm <station name>` on Mininet CLI.

- on server(STA1): `sipp -sn -uas`
- on client(STA2): `sipp -sn uac <STA1 IPv4> -timeout 120`



```
Last Reset Time          | 2024-11-18   18:07:38.263162 1731953258.263162

Current Time             | 2024-11-18   18:07:38.263241 1731953258.263241

-------------------------+---------------------------+--------------------------
Counter Name             | Periodic value            | Cumulative value
-------------------------+---------------------------+--------------------------
Elapsed Time             | 00:00:00:000000           | 00:02:03:997000
Call Rate                |    0.000 cps              |    9.613 cps
-------------------------+---------------------------+--------------------------
Incoming call created    |        0                  |     1192
OutGoing call created    |        0                  |        0
Total Call created       |                           |     1192
Current Call             |        0                  |
-------------------------+---------------------------+--------------------------
Successful call          |        0                  |     1192
Failed call              |        0                  |        0
-------------------------+---------------------------+--------------------------
Response Time 1          | 00:00:00:000000           | 00:00:00:009000
Call Length              | 00:00:00:000000           | 00:00:04:024000
------------------------------ Test Terminated --------------------------------
```

```
------------------------------ Scenario Screen -------- [1-9]: Change Screen --
  Call-rate(length)   Port   Total-time  Total-calls  Remote-host
  10.0(0 ms)/1.000s   5060     148.58 s         1222  10.0.0.1:5060(UDP)

  0 new calls during 1.003 s period      1 ms scheduler resolution
  30 calls (limit 30)                    Peak was 30 calls, after 122 s
  0 Running, 69 Paused, 13 Woken up
  0 dead call msg (discarded)            0 out-of-call msg (discarded)
  3 open sockets

                                 Messages  Retrans   Timeout   Unexpected-Msg
      INVITE ---------->         1222      150       0
         100 <----------         0         0         0         0
         180 <----------         1192      0         0         0
         183 <----------         0         0         0         0
         200 <----------  E-RTD1 1192      0         0         0
         ACK ---------->         1192      0
       Pause [      0ms]         1192                          0
         BYE ---------->         1192      0         0
         200 <----------         1192      0         0         0
```

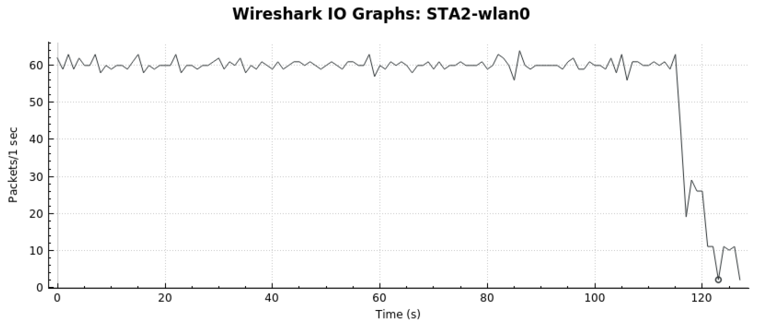
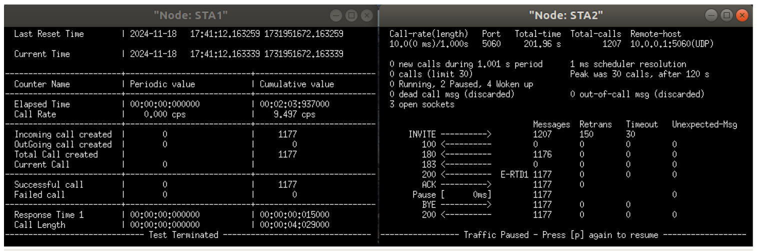


Figure 10: SIP connection & throughput graph under the BATMAN_ADV protocol

The `sipp -sn -uas` command sets up a User Agent Server on STA1 to respond to incoming SIP requests such as an INVITE messages with appropriate SIP responses such as 180 RINGING, 200 OK.
The `sipp -sn uac <STA1 IPv4> -timeout 120` command sets up STA2 as a User Agent Client to send INVITE messages to initiate SIP calls and wait for response such as 180 RINGING and 200 OK.

We capture and inspect the communication on Wireshark by running another instance of xterm on the client station on the Mininet CLI, then entered the command `sudo wireshark` to launch Wireshark. After it launches, the interface for the client station is selected to view the detailed packet information and throughput graph is generated by clicking on the 'Statistics' tab in Wireshark then selecting 'I/O Graph' from the dropdown list.



```
"Node: STA1"
  Last Reset Time          | 2024-11-18   17:41:12.163259 1731951672.163259

  Current Time             | 2024-11-18   17:41:12.163339 1731951672.163339

-------------------------+---------------------------+--------------------------
  Counter Name             | Periodic value            | Cumulative value
-------------------------+---------------------------+--------------------------
  Elapsed Time             | 00:00:00:000000           | 00:02:03:937000
  Call Rate                |    0.000 cps              |    9.497 cps
-------------------------+---------------------------+--------------------------
  Incoming call created    |        0                  |     1177
  OutGoing call created    |        0                  |        0
  Total Call created       |                           |     1177
  Current Call             |        0                  |
-------------------------+---------------------------+--------------------------
  Successful call          |        0                  |     1177
  Failed call              |        0                  |        0
-------------------------+---------------------------+--------------------------
  Response Time 1          | 00:00:00:000000           | 00:00:00:015000
  Call Length              | 00:00:00:000000           | 00:00:04:029000
------------------------------ Test Terminated --------------------------------
```

```
"Node: STA2"
  Call-rate(length)   Port   Total-time  Total-calls  Remote-host
  10.0(0 ms)/1.000s   5060     201.96 s         1207  10.0.0.1:5060(UDP)

  0 new calls during 1.001 s period      1 ms scheduler resolution
  0 calls (limit 30)                     Peak was 30 calls, after 120 s
  0 Running, 2 Paused, 4 Woken up
  0 dead call msg (discarded)            0 out-of-call msg (discarded)
  3 open sockets

                                 Messages  Retrans   Timeout   Unexpected-Msg
      INVITE ---------->         1207      150       30
         100 <----------         0         0         0         0
         180 <----------         1176      0         0         0
         183 <----------         0         0         0         0
         200 <----------  E-RTD1 1177      0         0         0
         ACK ---------->         1177      0
       Pause [      0ms]         1177                          0
         BYE ---------->         1177      0         0
         200 <----------         1177      0         0         0

----------------- Traffic Paused - Press [p] again to resume -----------------
```

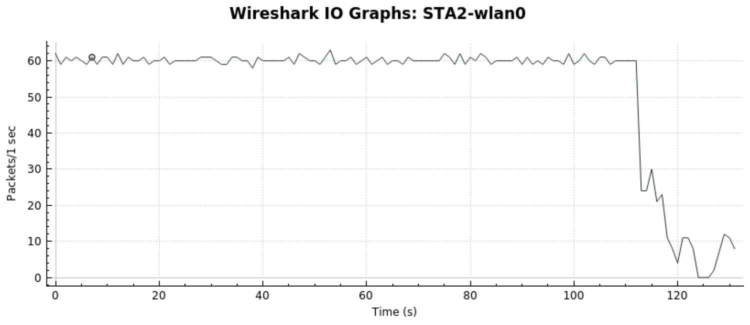


Figure 11: SIP connection & throughput under BATMAND protocol

The figures 10, 11 & 12 show the statistics from the SIP connection which lasts for 120 seconds on both server and client terminal CLI and the throughput graph from Wireshark.

Under the 'batman_adv' protocol, the connection statistics on the server(STA1) shows;

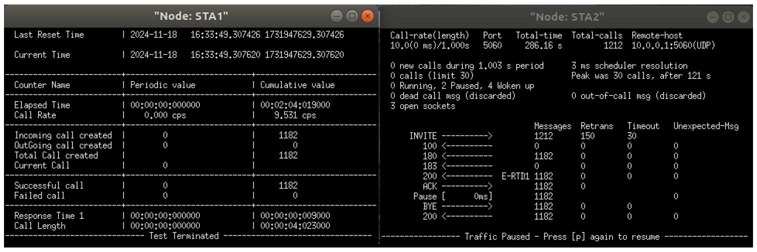
- Cumulative Call Rate of 9.613 calls per second
- 1192 Successful Calls created
- 0 Failed Calls

Under the 'batmand' protocol, the connection statistics on the server(STA1) shows;

- Cumulative Call Rate of 9.497 calls per second
- 1177 Successful Calls created
- 0 Failed Calls

Under the 'olsrd' protocol, the connection statistics on the server(STA1) shows;

- Cumulative Call Rate of 9.531 calls per second
- 1182 Successful Calls created
- 0 Failed Calls



```
"Node: STA1"
 Last Reset Time          | 2024-11-18   16:33:49.307426 1731947629.307426

 Current Time             | 2024-11-18   16:33:49.307620 1731947629.307620

-------------------------+---------------------------+--------------------------
 Counter Name            | Periodic value            | Cumulative value
-------------------------+---------------------------+--------------------------
 Elapsed Time            | 00:00:00:000000           | 00:02:04:019000
 Call Rate               |    0.000 cps              |    9.531 cps
-------------------------+---------------------------+--------------------------
 Incoming call created   |        0                  |     1182
 OutGoing call created   |        0                  |        0
 Total Call created      |                           |     1182
 Current Call            |        0                  |
-------------------------+---------------------------+--------------------------
 Successful call         |        0                  |     1182
 Failed call             |        0                  |        0
-------------------------+---------------------------+--------------------------
 Response Time 1         | 00:00:00:000000           | 00:00:00:009000
 Call Length             | 00:00:00:000000           | 00:00:04:023000
------------------------------ Test Terminated --------------------------------
```

```
"Node: STA2"
 Call-rate(length)   Port   Total-time  Total-calls  Remote-host
 10.0(0 ms)/1.000s   5060     286.16 s         1212  10.0.0.1:5060(UDP)

 0 new calls during 1.003 s period      3 ms scheduler resolution
 0 calls (limit 30)                     Peak was 30 calls, after 121 s
 0 Running, 2 Paused, 4 Woken up
 0 dead call msg (discarded)            0 out-of-call msg (discarded)
 3 open sockets

                                 Messages  Retrans   Timeout   Unexpected-Msg
     INVITE ---------->          1212      150       30
        100 <----------          0         0         0         0
        180 <----------          1182      0         0         0
        183 <----------          0         0         0         0
        200 <----------  E-RTD1 1182      0         0         0
        ACK ---------->          1182      0
      Pause [      0ms]          1182                          0
        BYE ---------->          1182      0         0
        200 <----------          1182      0         0         0

----------------- Traffic Paused - Press [p] again to resume ------------------
```

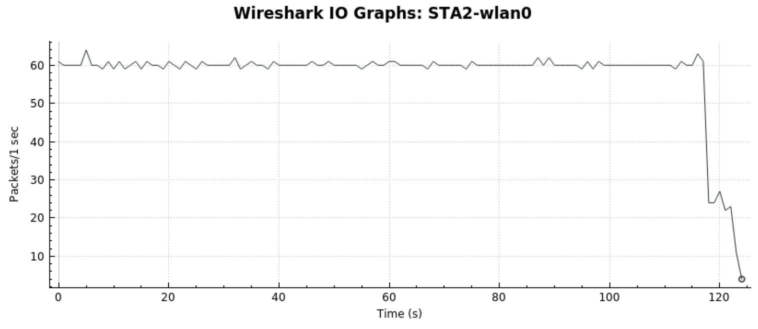


Figure 12: SIP connection & throughput under OLSRD protocol

At the end of this experiment, we have achieved a functioning adhoc network with successful VoIP connectivity under different adhoc protocols. Analysing the results from the statistics shows the best protocol to be the 'batman_adv' protocol since it achieved the highest call rate however, we notice slight difference among the 3 protocols tested in terms of network performance, connections maintained, and average bandwidth maintained during transmission period.

4 Task 3 - SDN Simulation

This task activity utilises the concept of Software Defined Networking in conjunction with OpenFlow on the local machine to manage our network. The experiment entails creating a network that connects an old building to a new building at the University of Hertfordshire. The task emulates 3 servers housed in the old building and 2 hosts in the new building. We utilise 5 switches for the underlying data forwarding and an SDN controller for the purpose of this activity. We also undertake a UDP transmission between two nodes for a specified duration, port number, and bandwidth.

4.1 ONOS Controller

ONOS stands for Open Network Operating System. ONOS provides the control plane for a software-defined network (SDN), managing network components, such as switches and links, and running software programs or modules to provide communication services to end hosts and neighbouring networks. It was developed by the ONOS Project, which is led by a collaboration of various universities and industry partners, including Intel and Ericsson. ONOS controller helps to attain the advantages of SDN by achieving network programmability, high-performance and scalability, support for OpenFlow protocols, support virtualisation of networks, fault tolerant and high-availability, and support for heterogeneity of network devices.

4.2 Design and Configuration

Table 4 contains the configuration information for hosts and servers in the network which is used in the python script for the network configuration. After completing the python script, the ONOS controller is built and started on a Linux terminal with the commands `bazel build onos` and `runsdn`. When the ONOS server is ready, we log in with `onos localhost` command from a new terminal where we have access to the controller and can activate some networking applications for the network to operate properly. The commands;

- `app activate org.onosproject.openflow`
- `app activate org.onosproject.fwd`

are used to activate flows and reactive routing on the controller which is necessary for transfer of packets across the network. Now, on a new terminal we run the network configuration file with the command;
`sudo mn --custom <python filename> --controller remote,ip=<ONOS controller IP> --topo <topo name>` which specifies the controller to be used for the network.
We access the ONOS GUI at `http://localhost:8181/onos/ui` to view the topology of the network which is shown in fig. 13. This is available only after all nodes can have full connectivity to each other, thus servers and hosts can communicate successfully. The hosts are made visible on the ONOS GUI by pressing the "H" key on the keyboard.

NAME	IPv4	MAC ADDRESS
H1	192.170.50.11	00:00:00:00:15:98
H2	192.170.50.12	00:00:00:00:15:99
SERVER1	20.0.0.2	00:00:00:00:16:00
SERVER2	40.0.0.2	00:00:00:00:16:01
SERVER3	60.0.0.2	00:00:00:00:16:02

Table 4: Network Configuration

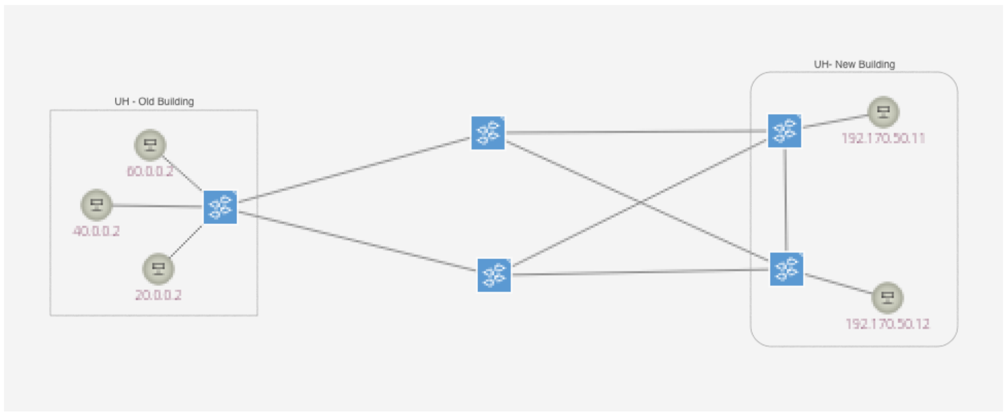


Figure 13: Building Plan

As seen in the configuration details in Table 4, the hosts and servers are in different networks. Without any routing configured on the nodes, nodes will not be able to communicate with nodes in a different network. To solve this issue, we configured static routes on each node with information of the other network. The command below is a generic one of the routing configured on each node;

- `ip route add <network address>/<netmask> dev <interface>`

This can be carried out on the Mininet CLI, the ip route command preceded with the node name or directly on the node terminal CLI accessed with 'xterm' from Mininet. Once the route configurations are done, we can expect to have full connectivity in the network and proceed to the UDP transmission. To carry out this activity, a terminal CLI for both nodes(Server & Host1) is launched using the 'xterm' command on Mininet. The 'iperf' is a command which can be used to stress-test a network to determine communication performance of the network.
On the server, we enter the command;

- `iperf -s -u -p <port> -i 1` : initiate the server to listen for UDP connections on port specfied for every second

And on client, we enter command;

- `iperf -c <server IP> -u -p <port> -t <time> -b <bandwidth>` : start the UDP connection by the client to the server, at port, duration, and bandwidth specified

4.3 Results and Analysis

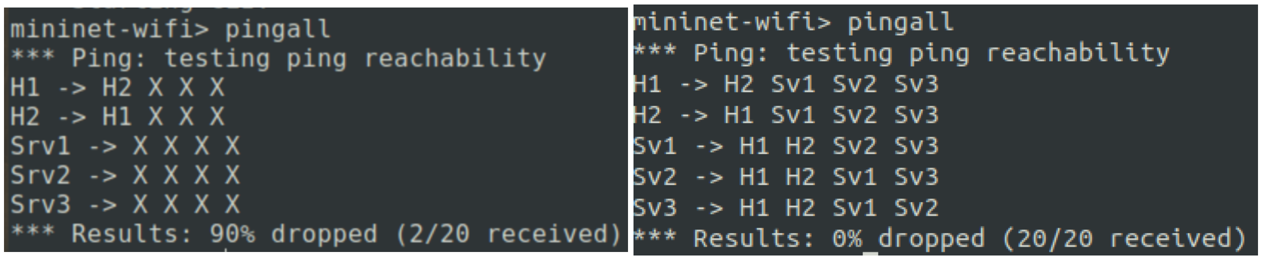
After activating the networking apps on the controller and starting the network on Mininet, we check the connectivity of nodes in our network prior to configuring static routes on each node. The fig. 14 has images of the results from a 'pingall' command in Mininet.

As shown, 90% of packets dropped from testing connectivity prior configuring the ip routes on nodes due to the nodes having no idea how to reach other networks outside their original network. Once the nodes have information on how to reach other networks, full connectivity is achieved as we see 0% packets were dropped.

After connectivity is confirmed, we commence the UDP(connectionless) communication between Server1 & Host1 for 600 seconds, with bandwidth of 100 megabytes per second, and on port 5656.

The results from the UDP test in fig. 15 showed successful connection between Server1 & Host1 with the following statistics;

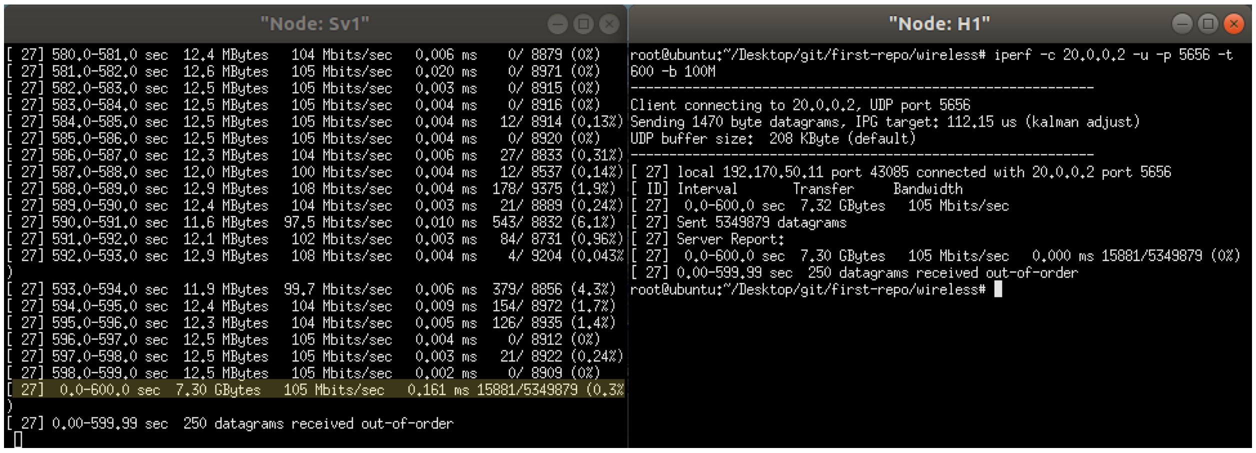
- connection time of 600 seconds
- total jitter(time of arrival of each packet) of 0.161 milliseconds
- transferred total of 7.32 gigabytes of data
- maintained bandwidth of 105 megabytes per second
- 5349875 total datagrams sent and 250 datagrams received out-of-order



```
mininet-wifi> pingall
*** Ping: testing ping reachability
H1 -> H2 X X X
H2 -> H1 X X X
Srv1 -> X X X X
Srv2 -> X X X X
Srv3 -> X X X X
*** Results: 90% dropped (2/20 received)
```

```
mininet-wifi> pingall
*** Ping: testing ping reachability
H1 -> H2 Sv1 Sv2 Sv3
H2 -> H1 Sv1 Sv2 Sv3
Sv1 -> H1 H2 Sv2 Sv3
Sv2 -> H1 H2 Sv1 Sv3
Sv3 -> H1 H2 Sv1 Sv2
*** Results: 0% dropped (20/20 received)
```

Figure 14: Connectivity tests before & after reactive routing activation



```
[ 27] 580.0-581.0 sec  12.4 MBytes   104 Mbits/sec   0.006 ms    0/ 8879 (0%)
[ 27] 581.0-582.0 sec  12.6 MBytes   105 Mbits/sec   0.020 ms    0/ 8971 (0%)
[ 27] 582.0-583.0 sec  12.5 MBytes   105 Mbits/sec   0.003 ms    0/ 8915 (0%)
[ 27] 583.0-584.0 sec  12.5 MBytes   105 Mbits/sec   0.004 ms    0/ 8916 (0%)
[ 27] 584.0-585.0 sec  12.5 MBytes   105 Mbits/sec   0.004 ms   12/ 8914 (0.13%)
[ 27] 585.0-586.0 sec  12.5 MBytes   105 Mbits/sec   0.004 ms    0/ 8920 (0%)
[ 27] 586.0-587.0 sec  12.3 MBytes   104 Mbits/sec   0.006 ms   27/ 8833 (0.31%)
[ 27] 587.0-588.0 sec  12.0 MBytes   100 Mbits/sec   0.004 ms   12/ 8537 (0.14%)
[ 27] 588.0-589.0 sec  12.9 MBytes   108 Mbits/sec   0.004 ms  178/ 9375 (1.9%)
[ 27] 589.0-590.0 sec  12.4 MBytes   104 Mbits/sec   0.003 ms   21/ 8889 (0.24%)
[ 27] 590.0-591.0 sec  11.6 MBytes  97.5 Mbits/sec   0.010 ms  543/ 8832 (6.1%)
[ 27] 591.0-592.0 sec  12.1 MBytes   102 Mbits/sec   0.003 ms   84/ 8731 (0.96%)
[ 27] 592.0-593.0 sec  12.9 MBytes   108 Mbits/sec   0.004 ms    4/ 9204 (0.043%
)
[ 27] 593.0-594.0 sec  11.9 MBytes  99.7 Mbits/sec   0.006 ms  379/ 8856 (4.3%)
[ 27] 594.0-595.0 sec  12.4 MBytes   104 Mbits/sec   0.009 ms  154/ 8972 (1.7%)
[ 27] 595.0-596.0 sec  12.3 MBytes   104 Mbits/sec   0.005 ms  126/ 8935 (1.4%)
[ 27] 596.0-597.0 sec  12.5 MBytes   105 Mbits/sec   0.004 ms    0/ 8912 (0%)
[ 27] 597.0-598.0 sec  12.5 MBytes   105 Mbits/sec   0.003 ms   21/ 8922 (0.24%)
[ 27] 598.0-599.0 sec  12.5 MBytes   105 Mbits/sec   0.002 ms    0/ 8909 (0%)
[ 27]  0.0-600.0 sec  7.30 GBytes   105 Mbits/sec   0.161 ms 15881/5349879 (0.3%
)
[ 27] 0.00-599.99 sec  250 datagrams received out-of-order
```

```
root@ubuntu:~/Desktop/git/first-repo/wireless# iperf -c 20.0.0.2 -u -p 5656 -t
600 -b 100M
------------------------------------------------------------
Client connecting to 20.0.0.2, UDP port 5656
Sending 1470 byte datagrams, IPG target: 112.15 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 27] local 192.170.50.11 port 43085 connected with 20.0.0.2 port 5656
[ ID] Interval       Transfer     Bandwidth
[ 27]  0.0-600.0 sec  7.32 GBytes   105 Mbits/sec
[ 27] Sent 5349879 datagrams
[ 27] Server Report:
[ 27]  0.0-600.0 sec  7.30 GBytes   105 Mbits/sec   0.000 ms 15881/5349879 (0%)
[ 27] 0.00-599.99 sec  250 datagrams received out-of-order
root@ubuntu:~/Desktop/git/first-repo/wireless# 
```

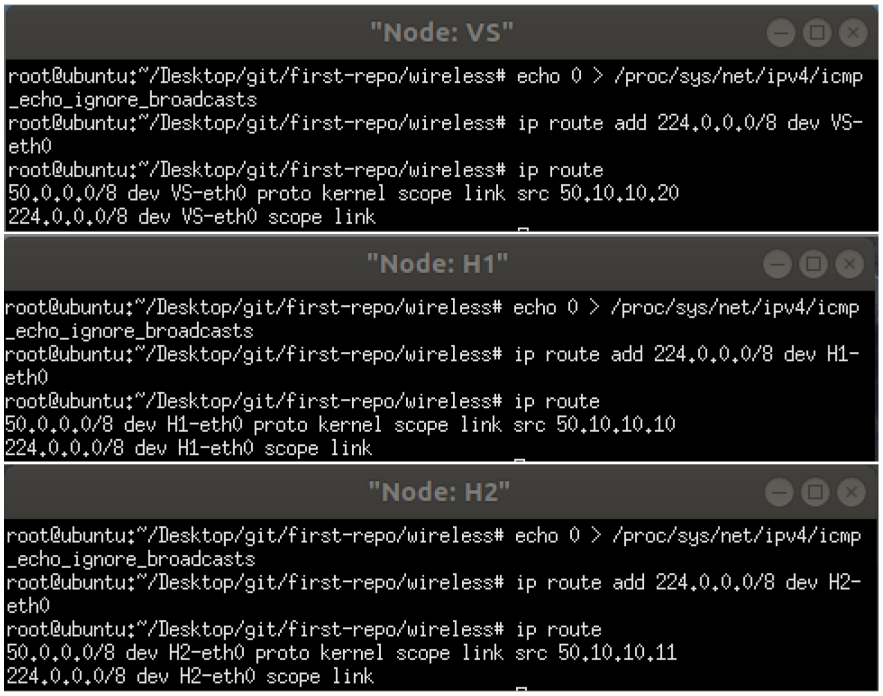
Figure 15: UDP connection between Server & Client

UDP based communications are faster compared to TCP based connections. This is due to the lack of error checks and the acknowledgement technique used by TCP to establish secure and less error in communications. This snippet from Wireshark gives details on how the UDP protocol works. The transfer is one-way(from Host1 to Server1) only with no acknowledgment from Server1.

5 Task 4 - Multicast Video Stream Service

DEVICE NAME	IP ADDRESS
H1	50.10.10.10
H2	50.10.10.11
H3	50.10.10.12
VS	50.10.10.20

Table 5: Network Configuration

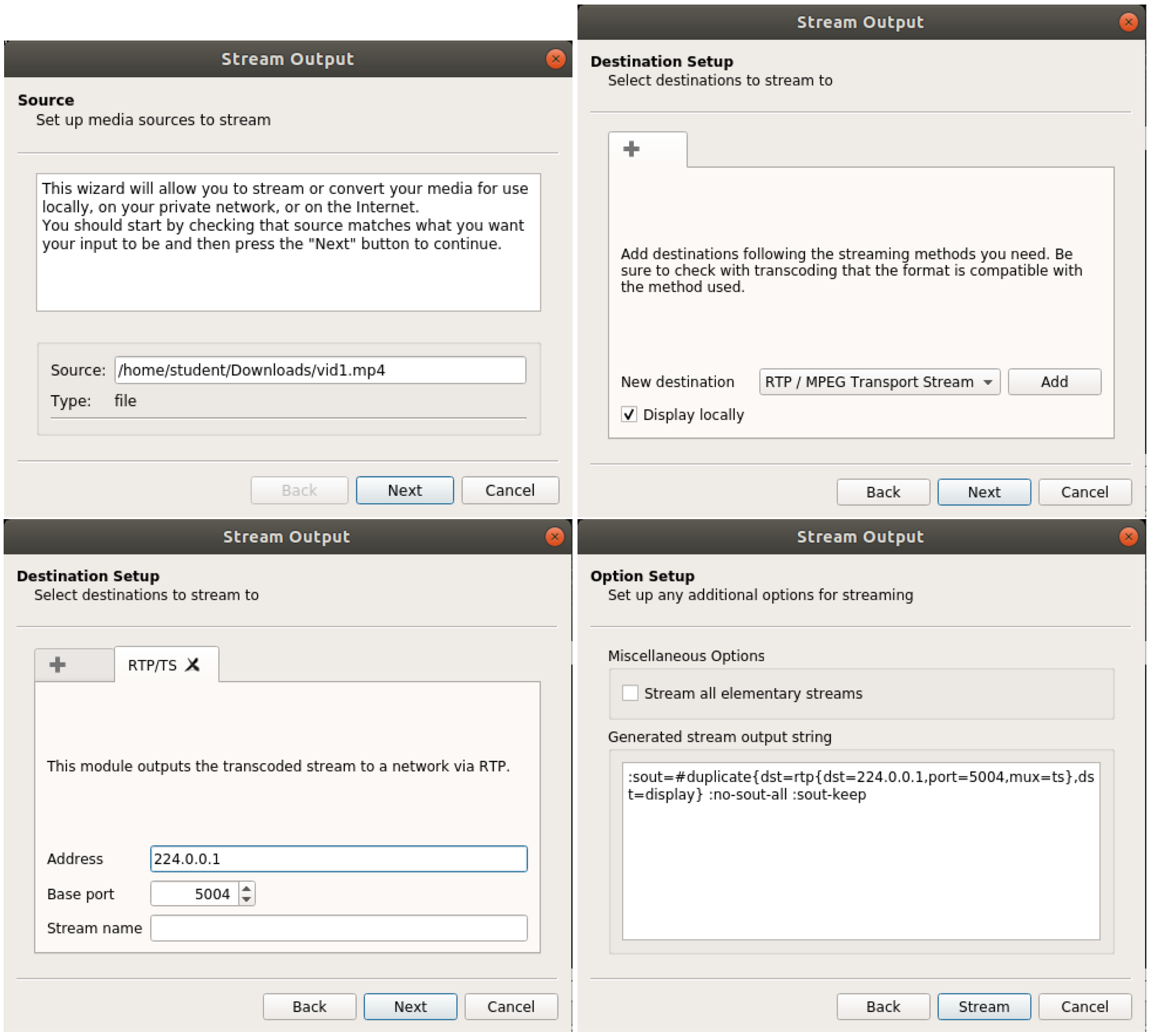


```
root@ubuntu:~/Desktop/git/first-repo/wireless# echo 0 > /proc/sys/net/ipv4/icmp
_echo_ignore_broadcasts
root@ubuntu:~/Desktop/git/first-repo/wireless# ip route add 224.0.0.0/8 dev VS-
eth0
root@ubuntu:~/Desktop/git/first-repo/wireless# ip route
50.0.0.0/8 dev VS-eth0 proto kernel scope link src 50.10.10.20
224.0.0.0/8 dev VS-eth0 scope link
```

```
root@ubuntu:~/Desktop/git/first-repo/wireless# echo 0 > /proc/sys/net/ipv4/icmp
_echo_ignore_broadcasts
root@ubuntu:~/Desktop/git/first-repo/wireless# ip route add 224.0.0.0/8 dev H1-
eth0
root@ubuntu:~/Desktop/git/first-repo/wireless# ip route
50.0.0.0/8 dev H1-eth0 proto kernel scope link src 50.10.10.10
224.0.0.0/8 dev H1-eth0 scope link
```

```
root@ubuntu:~/Desktop/git/first-repo/wireless# echo 0 > /proc/sys/net/ipv4/icmp
_echo_ignore_broadcasts
root@ubuntu:~/Desktop/git/first-repo/wireless# ip route add 224.0.0.0/8 dev H2-
eth0
root@ubuntu:~/Desktop/git/first-repo/wireless# ip route
50.0.0.0/8 dev H2-eth0 proto kernel scope link src 50.10.10.11
224.0.0.0/8 dev H2-eth0 scope link
```

Figure 16: Activating multicast for required nodes



Source
Set up media sources to stream

This wizard will allow you to stream or convert your media for use locally, on your private network, or on the Internet.
You should start by checking that source matches what you want your input to be and then press the "Next" button to continue.

Source: /home/student/Downloads/vid1.mp4
Type: file

Destination Setup
Select destinations to stream to

Add destinations following the streaming methods you need. Be sure to check with transcoding that the format is compatible with the method used.

New destination: RTP / MPEG Transport Stream
[✓] Display locally

Destination Setup
Select destinations to stream to

RTP/TS

This module outputs the transcoded stream to a network via RTP.

Address: 224.0.0.1
Base port: 5004
Stream name:

Option Setup
Set up any additional options for streaming

Miscellaneous Options
[ ] Stream all elementary streams

Generated stream output string

```
:sout=#duplicate{dst=rtp{dst=224.0.0.1,port=5004,mux=ts},ds
t=display} :no-sout-all :sout-keep
```

Figure 17: Steps for multicast video streaming

5.1 Results and Analysis

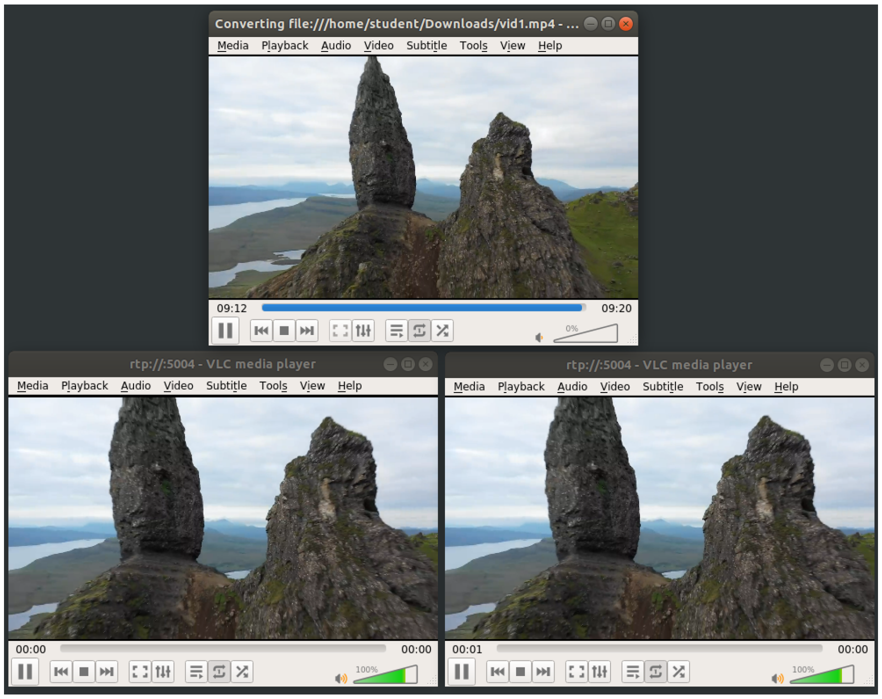

Converting file:///home/student/Downloads/vid1.mp4 - ...
Media Playback Audio Video Subtitle Tools View Help
09:12
09:20
rtp://:5004 - VLC media player
00:00
00:00
100%
rtp://:5004 - VLC media player
00:01
00:00
100%

Figure 18: Multicast video stream from source to hosts

6 References

References

[1] Rogério Leão Santos de Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. pages 1–6, 2014.

[2] Saad H. Haji, Subhi R. M. Zeebaree, Rezgar Hasan Saeed, Siddeeq Y. Ameen, Hanan M. Shukur, Naaman Omar, Mohammed A. M. Sadeeq, Zainab Salih Ageed, Ibrahim Mahmood Ibrahim, and Hajar Maseeh Yasin. Comparison of software defined networking with traditional networking. *Asian Journal of Research in Computer Science*, 9(2):1–18, May 2021.

[3] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[4] Adrian Lara, Anisha Kolasani, and Byrav Ramamurthy. Network innovation using openflow: A survey. *IEEE Communications Surveys Tutorials*, 16(1):493–512, 2014.

[5] Stephen Lukasik. Why the arpanet was built. *IEEE Annals of the History of Computing*, 33(3):4–21, 2011.

[6] Koketso Molemane Rodney Mokoena, Ramahlapane Lerato Moila, and Prof Mthulisi Velempini. Improving network management with software defined networking using openflow protocol. In *2023 International Conference on Artificial Intelligence, Big Data, Computing and Data Communication Systems (icABCD)*, pages 1–5, 2023.

[7] C. S. R. Murthy and B. S. Manoj. *Ad hoc wireless networks: architectures and protocols.* Prentice Hall PTR, Upper Saddle River, N.J, 2004.

7 Appendix

a Task 1 code

```
import sys
from mininet.node import Controller
from mininet.log import setLogLevel, info
from mn_wifi.cli import CLI
from mn_wifi.net import Mininet_wifi

def topology():
    info("**creating network**\n")
    net = Mininet_wifi()

    info("**creating nodes**\n")
    ap1 = net.addAccessPoint('ap1', mac='00:00:00:00:00:00', ssid='AP1', mode
        ='g', channel='1', position='30,117.5', band='5', range=35)
    ap2 = net.addAccessPoint('ap2', mac='00:00:00:00:00:01', ssid='AP2', mode
        ='g', channel='1', position='60,117.5', band='5', range=35)
    ap3 = net.addAccessPoint('ap3', mac='00:00:00:00:00:02', ssid='AP3', mode
        ='g', channel='1', position='80,117.5', band='5', range=35)
    ap4 = net.addAccessPoint('ap4', mac='00:00:00:00:00:03', ssid='AP4', mode
        ='g', channel='1', position='135,55', band='5', range=50)
    ap5 = net.addAccessPoint('ap5', mac='00:00:00:00:00:04', ssid='AP5', mode
        ='g', channel='1', position='135,40', band='5', range=50)
    sta1 = net.addStation('iphone5', mac='00:00:00:00:00:10', ip
        ='192.168.50.11/24', position='15,115', range=20, min_v=1, max_v=5)
    sta2 = net.addStation('ipad', mac='00:00:00:00:00:11', ip
        ='192.168.50.12/24', position='20,130', range=20, min_v=5, max_v=10)
    sta3 = net.addStation('macbook', mac='00:00:00:00:00:12', ip
        ='192.168.50.13/24', position='140,10', range=20, min_v=2, max_v=7)
    c0 = net.addController('c0')
    net.setPropagationModel(model="logDistance", exp=5)

    info("**configuring wifi nodes and mobility**\n")
    net.configureWifiNodes()
    net.addLink(ap1, ap2)
    net.addLink(ap2, ap3)
    net.addLink(ap3, ap4)
    net.addLink(ap4, ap5)
    net.plotGraph(min_x=-20, min_y=-10, max_x=200, max_y=180)

    net.startMobility(time=0)
    net.mobility(sta1, 'start', time=10, position='15,115')
    net.mobility(sta1, 'stop', time=20, position='115,10')
    net.mobility(sta2, 'start', time=30, position='20,130')
    net.mobility(sta2, 'stop', time=60, position='150,10')
    net.mobility(sta3, 'start', time=25, position='140,10')
    net.mobility(sta3, 'stop', time=60, position='15,110')
    net.stopMobility(time=120)

    info("**starting network**\n")
    net.build()
    c0.start()
    ap1.start([c0])
    ap2.start([c0])
    ap3.start([c0])
    ap4.start([c0])
    ap5.start([c0])

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

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

if __name__ == '__main__':
    setLogLevel('info')
    plot = False if '-p' in sys.argv else True
    topology()
```

b Task 2 code

```
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")
    sta1 = net.addStation('STA1', ip6='2024::11', mac='00:00:00:00:01:11',
        position='20,10,0', range=30, antennaGain=5, antennaHeight=1)
    sta2 = net.addStation('STA2', ip6='2024::12', mac='00:00:00:00:01:12',
        position='45,10,0', range=30, antennaGain=6, antennaHeight=2)
    sta3 = net.addStation('STA3', ip6='2024::13', mac='00:00:00:00:01:13',
        position='70,10,0', range=30, antennaGain=7, antennaHeight=3)
    net.setPropagationModel(model="logDistance", exp=4)
    info("*** Configuring nodes\n")
    net.configureNodes()

    info("*** Creating links\n")
    #testing 'batman_adv', 'batmand', 'olsrd' manet protocol
    net.plotGraph(min_x=-20, min_y=-40, max_x=120, max_y=90)
    net.addLink(sta1, cls=adhoc, intf='STA1-wlan0', ssid='adhocUH', mode='g',
        channel=5, ht_cap='HT40+', proto='batman_adv')
    net.addLink(sta2, cls=adhoc, intf='STA2-wlan0', ssid='adhocUH', mode='g',
        channel=5, ht_cap='HT40+', proto='batman_adv')
    net.addLink(sta3, cls=adhoc, intf='STA3-wlan0', ssid='adhocUH', mode='g',
        channel=5, ht_cap='HT40+', proto='batman_adv')

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

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

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

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

c Task 3 code

```
from mininet.topo import Topo
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSSwitch
from mininet.topo import Topo

class VLANHost(Host):
        def config(self, vlan=100, **params):
                """Configure VLANHost with a VLAN."""
                r = super(Host, self).config(**params)
                intf = self.defaultIntf()
                self.cmd('ifconfig %s inet 0' % intf)
                self.cmd('vconfig add %s %d' % (intf, vlan))
                self.cmd('ifconfig %s.%d inet %s' % (intf, vlan, params['ip'])
                    )
                newName = '%s.%d' % (intf, vlan)
                intf.name = newName
                self.nameToIntf[newName] = intf
                return r

class MyTopo(Topo):
        "Simple topology with VLAN support."
        def __init__(self):
                "Create custom topology."

                # Initialize topology
                Topo.__init__(self)

                h1 = self.addHost( 'H1', mac='00:00:00:00:15:98', ip
                    ='192.170.50.11/24' )
                h2 = self.addHost( 'H2', mac='00:00:00:00:15:99', ip
                    ='192.170.50.12/24' )
                SERVER1 = self.addHost( 'Sv1', mac='00:00:00:00:16:00', ip
                    ='20.0.0.2/8' )
                SERVER2 = self.addHost( 'Sv2', mac='00:00:00:00:16:01', ip
                    ='40.0.0.2/8' )
                SERVER3 = self.addHost( 'Sv3', mac='00:00:00:00:16:02', ip
                    ='60.0.0.2/8' )

                Switch1 = self.addSwitch( 'Switch1', cls=OVSSwitch )
                Switch2 = self.addSwitch( 'Switch2', cls=OVSSwitch )
                Switch3 = self.addSwitch( 'Switch3', cls=OVSSwitch )
                Switch4 = self.addSwitch( 'Switch4', cls=OVSSwitch )
                Switch5 = self.addSwitch( 'Switch5', cls=OVSSwitch )

                self.addLink( h1, Switch4 )
                self.addLink( h2, Switch5 )
                self.addLink( SERVER1, Switch1 )
                self.addLink( SERVER2, Switch1 )
                self.addLink( SERVER3, Switch1 )
                self.addLink( Switch1, Switch2 )
                self.addLink( Switch1, Switch3 )
                self.addLink( Switch2, Switch4 )
                self.addLink( Switch2, Switch5 )
                self.addLink( Switch3, Switch4 )
                self.addLink( Switch3, Switch5 )
                self.addLink( Switch4, Switch5 )

topos = { 'mytopo': ( lambda: MyTopo() ) }
```

d Task 4 code

```
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import Node
from mininet.log import setLogLevel, info
from mininet.cli import CLI

class MyTopo( Topo ):
    "Simple topology example."
    def __init__( self ):
        "Create custom topo."

        # Initialize topology
        Topo.__init__( self )

        # hosts and switches
        h1 = self.addHost('H1', ip='50.10.10.10/8')
        h2 = self.addHost('H2', ip='50.10.10.11/8')
        h3 = self.addHost('H3', ip='50.10.10.12/8')
        h4 = self.addHost('vidSrc', ip='50.10.10.20/8')
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')

        self.addLink(h1, s2)
        self.addLink(h2, s2)
        self.addLink(h3, s2)
        self.addLink(s1, s2)
        self.addLink(h4, s1)

topos = { 'mytopo': ( lambda: MyTopo() ) }
```