

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

Student ID: 22076082

December 12, 2024

Software Defined Networking(SDN) emulation with
Mininet and OpenFlow

University of Hertfordshire
Hatfield

Contents

1	Introduction	5
1.1	Software-Defined Networking (SDN) and OpenFlow	5
1.2	Mininet	6
1.3	Emulation environment specifications	6
2	Task 1 - WiFi Network Emulation	7
2.1	Wireless Networks	7
2.2	Design and Configuration	7
2.3	Results and Analysis	8
3	Task 2 - Adhoc Network Emulation	10
3.1	AdHoc Networks	10
3.2	Design and Configuration	10
3.3	Results and Analysis	11
4	Task 3 - Software-Defined Networking(SDN)	14
4.1	ONOS Controller	14
4.2	Design and Configuration	14
4.3	Results and Analysis	16
5	Task 4 - Multicast Video Stream Service	18
5.1	Design and Configuration	18
5.2	Results and Analysis	20
6	Conclusion	21
7	References	22
8	Appendix	23
a	Task 1 code	23
b	Task 2 code	24
c	Task 3 code	25
d	Task 4 code	27

List of Tables

1	Details of stations and access points	8
2	Mobility information for stations	8
3	Ad-hoc stations configuration details	11
4	Network configuration details	15
5	Network configuration details	18

List of Figures

1	Architecture for Traditional and SDN networks	5
2	SDN architecture	6
3	Floor-plan	7
4	Floor-plan with network topology	7
5	Prior mobility	8
6	After mobility	8
7	Access points connected after mobility	9
8	Full node connectivity	9
9	Adhoc network with 3 stations	10
10	SIP connection & throughput under BATMAN_ADV protocol	11
11	SIP connection & throughput under BATMAND protocol	12
12	SIP connection & throughput under OLSRD protocol	13
13	Network topology from ONOS GUI	15
14	Connectivity tests before & after reactive routing activation	16
15	UDP connection between Server & Client	16
16	Wireshark capture on H1-eth0 interface	17
17	Activating multicast for required nodes	18
18	Multicast streaming setup at video source	19
19	Multicast video connection at multicast host	19
20	Multicast video stream from source to hosts	20
21	Failed video stream at host 3	20
22	Packet details captured on Wireshark	21

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) provide 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 use-cases of 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) and UDP(connectionless) communication in client-server 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

Developments and innovations in computer networks have seen 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 agreed that the network devices connect but there is no centralised machine to manage or summarise the whole network (Haji et al. 2021).

Also, the distributed control and transport protocols running inside routers and switches make it complex to express desired high-level network policies since network operators must configure each network device separately (Kreutz et al. 2015). This main issue is rectified by SDN, which introduces the needed network control, flexibility, and programmability that network operators want.

1.1 Software-Defined Networking (SDN) and OpenFlow

Software-Defined Networking (SDN) is a computer technology that changes the limitations of current computer network infrastructure. SDN (Lara et al. 2014) 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 (Kreutz et al. 2015). To make possible the communication between the controller and the network devices, OpenFlow (Lara et al. 2014) 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 such as OpenFlow, to control the flow table of switches through software was provided by identifying common features in the flow tables of the ethernet switches (Lara et al. 2014).

The figures below show a comparison of the overview for traditional and SDN network architecture.

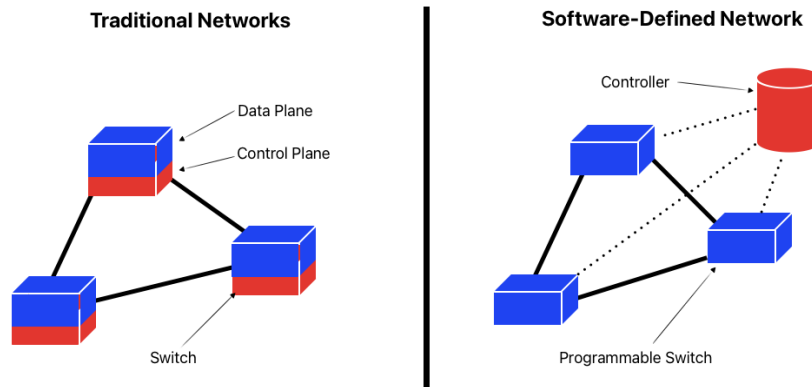


Figure 1: Architecture for Traditional and SDN networks

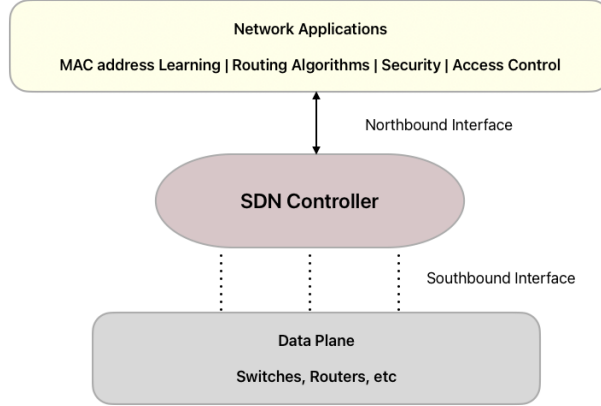


Figure 2: SDN architecture

The underlying network applications or packages which are the software for the logical operations within the network are installed and activated on the SDN controller. These applications are known as the Northbound APIs or interfaces as shown in Fig. 2. The Southbound interface consists of the protocols that facilitate communication between the remote SDN controller and the network devices, for example, the OpenFlow (Mokoena et al. 2023) protocol. Examples of such network applications can be STP, routing algorithms, and access control and examples of an 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 it presents.

1.2 Mininet

To test out the efficiency, reliability, cost, and flexibility of the SDN approach in our network designs, the experiments are carried out on a network emulator 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 (Haji et al. 2021, Mokoena et al. 2023).

Mininet (de Oliveira et al. 2014) provides a system that allows rapid prototyping of large networks and creates scalable software-defined networks using a lightweight virtualisation mechanism. It provides efficient ways for network creation 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, customises them, shares them among other networks, and performs interactions (de Oliveira et al. 2014).

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 4.2.14.

2 Task 1 - WiFi Network Emulation

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 a smartphone to a laptop, UE or 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 a mobility state to emulate a real-life network scenario.

2.1 Wireless Networks

Wireless networks are forms of computer networks which use radio-based communication technologies. The nodes in the network share communication channels for transmission which can lead to issues such as "Hidden Node Problem" and "Exposed Node Problem" dealt with at the MAC layer. Medium Access Control or MAC (Murthy & Manoj 2004) is a mechanism devised to manage how radio channels are accessed by wireless capable devices for wireless communication and there are two main techniques;

- Contention-based MAC: nodes compete to gain access to channel e.g. CSMA, ALOHA, MACA
- Contention-free MAC: nodes do not compete to gain access to channel e.g. TDMA, FDMA, CDMA

2.2 Design and Configuration

The figures below show the outline of the floor-plan in context and position of access points and stations in the network.

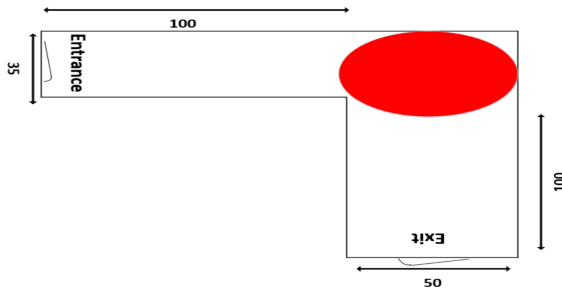


Figure 3: Floor-plan

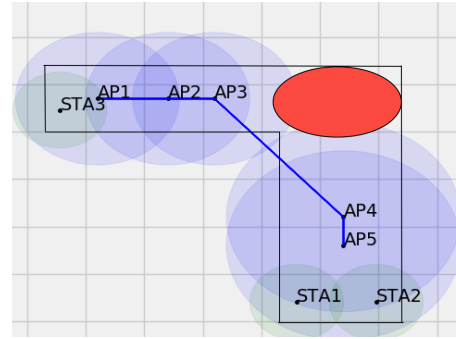


Figure 4: Floor-plan with network topology

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 positioned to have full access to access points.

The access points are linked serially to ensure the required zones within the floor have strong WiFi frequency coverage and support full connectivity of the stations. To emulate a 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 information for stations

2.3 Results and Analysis

After the Python script for the network topology with all necessary libraries are set, we start it with Mininet on a Linux terminal with the command, `sudo ./<pycode filename>`. 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 to mobility and after mobility respectively. All stations are within strong WiFi signal coverage and successfully communicate with each other shown in Fig. 8.

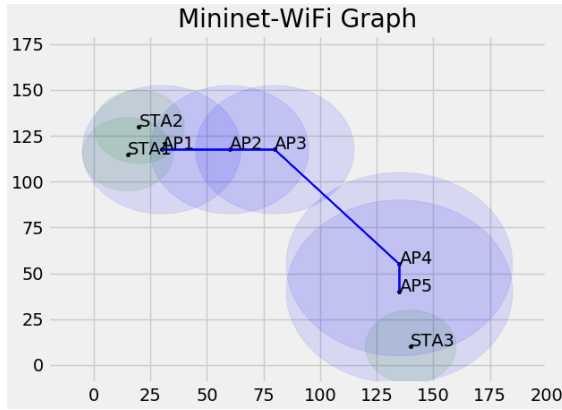


Figure 5: Prior mobility

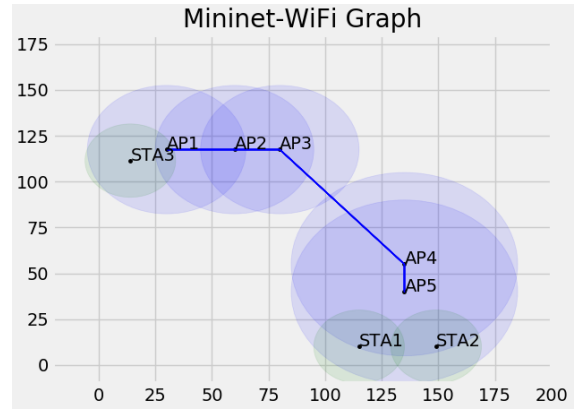


Figure 6: After mobility

The `ping -c 3` command is used to send out 3 Internet Control Message Protocol(ICMP) (Murthy & Manoj 2004) packets between specified stations to determine full connectivity within the network. ICMP uses a series of echo request & reply messages to establish communication between stations. The results show all 3 packets were transmitted and received successfully, with no packet loss, and took an average time of 2000 milliseconds. This shows the optimal performance of the network is good.

Prior mobility, stations 1 & 2 would likely be associated with AP1 and station 3 with AP5 due to their close proximity. After mobility, the stations will lose association with access points or go through a handoff operation then get associated with the access point in close proximity. The `<station name> iwconfig` command is used on the Mininet CLI to get 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.

```
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:0  Missed beacon:0
```

Figure 7: Access points 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: Full node connectivity

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.

3 Task 2 - Adhoc Network Emulation

In this task, we emulate an Ad-Hoc network scenario at an emergency gathering car park at the new building when response units communicate via AdHoc services and evaluate some ad-hoc protocols to find which was best. For this purpose, 3 stations are emulated and the ad-hoc protocols which will be tested are the 'batmand', 'batman_adv', and 'olsrd' protocols. To aid in analysing network performance, we use a network protocol analytic tool known as Wireshark to inspect and capture the 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 (Murthy & Manoj 2004) networks are quick-to-deploy and dynamic forms 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 and can move randomly within the local network. AdHoc networks are less reliant on infrastructure and are efficient in setting up wireless communications in cases of emergencies or quick deployment. It is applied in Wireless Mesh Networks(WMN), Wireless Sensor Networks(WSN), and Mobile Ad-Hoc Networks(MANET).

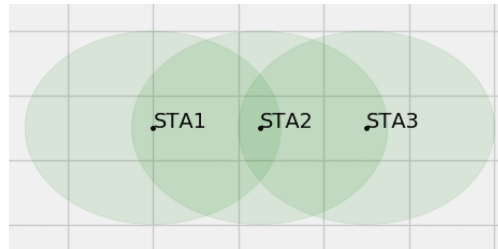


Figure 9: Adhoc network with 3 stations

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: Ad-hoc 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 the client. The SIP commands are run on each station 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`

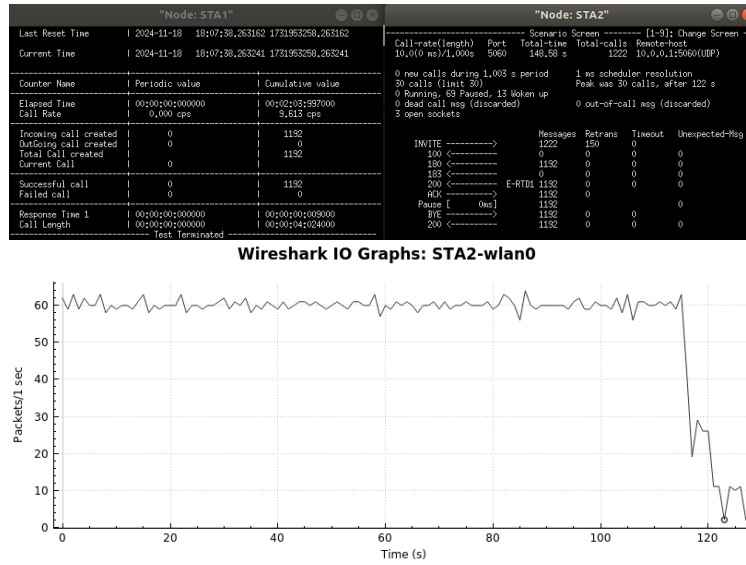


Figure 10: SIP connection & throughput under BATMAN-ADV protocol

The `sipp -sn -uas` command sets up a User Agent Server on STA1 to respond to incoming SIP requests such as an INVITE message 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 enter 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 the throughput graph is generated by clicking on the 'Statistics' tab in Wireshark and then selecting 'I/O Graph' from the dropdown list.

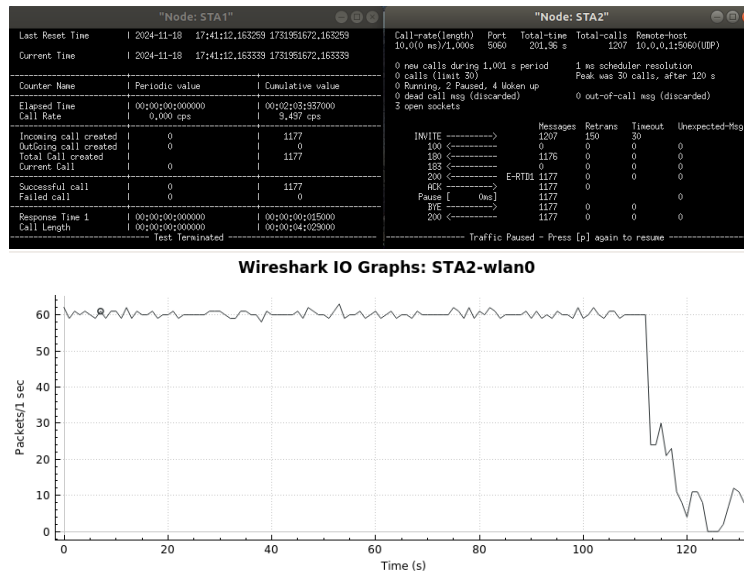


Figure 11: SIP connection & throughput under BATMAND protocol

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

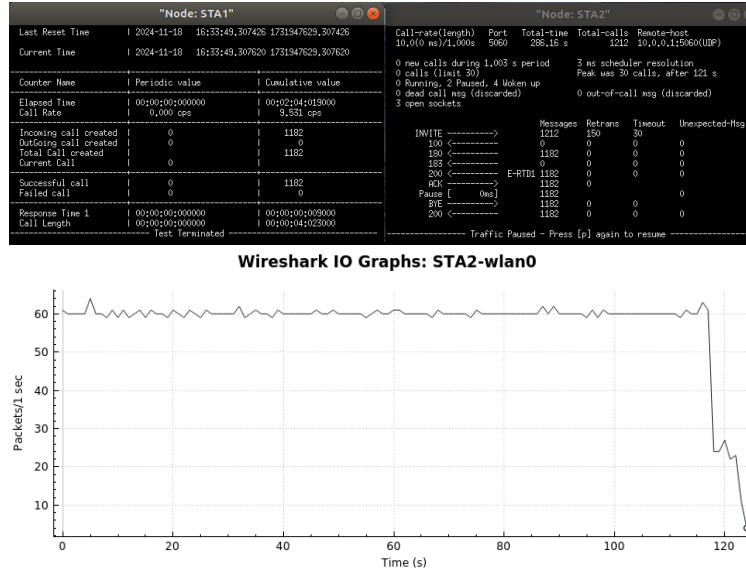


Figure 12: SIP connection & throughput under OLSRD protocol

At the end of this experiment, we have achieved a functioning ad-hoc network with successful VoIP connectivity under different ad-hoc 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 differences among the 3 protocols tested in terms of network performance, connections maintained, and average bandwidth maintained during transmission period.

4 Task 3 - Software-Defined Networking(SDN)

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 User Datagram Protocol(UDP) transmission between two nodes for a specified duration, port number, and bandwidth.

A UDP transmission is a type of communication protocol which unlike Transmission Control Protocol(TCP)(Conrad et al. 2012) the communication does not require the three-way handshake(SYN - SYN_ACK - ACK) to establish a secure connection.

- SYN - sender sends synchronisation message to receiver
- SYN_ACK - receiver sends synchronisation-acknowledgement message to sender
- ACK - sender sends acknowledgment message to receiver to establish communication

The sender sends out packets without waiting for an acknowledgement from the recipient. This form of communication does check errors and may encounter loss of packets during transmission but in turn, achieves higher transmission speeds and supports transfer of large size data.

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 how we desire. These 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 the 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 details

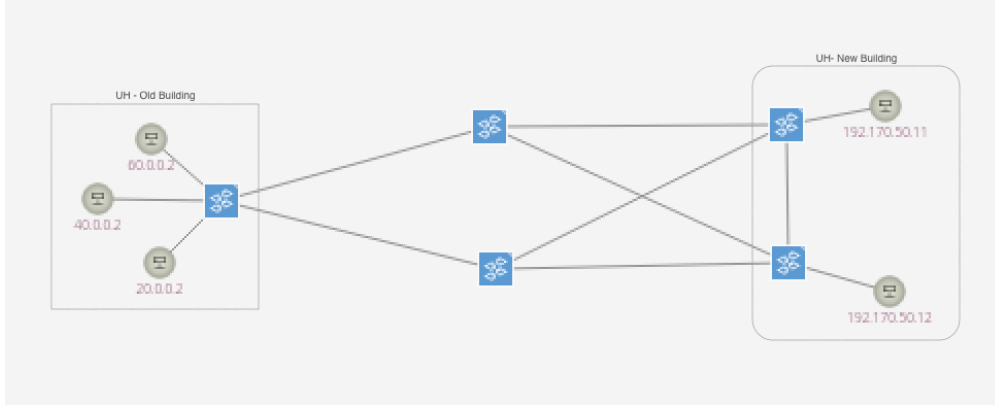


Figure 13: Network topology from ONOS GUI

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 on 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 by the node name or directly on the node terminal CLI accessed with 'xterm' on 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 the 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 specified 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 we activate the network apps on the ONOS controller and start the network on Mininet, we check the connectivity of nodes in our network prior to configuring static routes on each node. Fig. 14 has images of the results from a 'pingall' command in Mininet.

```

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

As shown, 90% of packets dropped from testing connectivity prior to 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 routing information to reach other networks, full connectivity is achieved as shown from 0% packets dropped.

Then commenced the UDP communication between Server1 & Host1 for 600 seconds, with a bandwidth of 100 megabytes per second, and on port 5656 after full connectivity of the network is established.

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 interval of packet arrival) 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

```

"Node: Sv1"
[ 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 0,005 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/ 8375 (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 12,3 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

"Node: H1"
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

In general, 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 security and less error in communications.

No.	Time	Source	Destination	Protocol	Length	Info
3	0.350317428	192.170.50.11	20.0.0.2	UDP	1512	57081 → 5656 Len=1470
4	0.350937563	192.170.50.11	20.0.0.2	UDP	1512	57081 → 5656 Len=1470
5	0.350958881	192.170.50.11	20.0.0.2	UDP	1512	57081 → 5656 Len=1470
6	0.350969293	192.170.50.11	20.0.0.2	UDP	1512	57081 → 5656 Len=1470
7	0.350980321	192.170.50.11	20.0.0.2	UDP	1512	57081 → 5656 Len=1470

Figure 16: Wireshark capture on H1-eth0 interface

This snippet from Wireshark gives details on how the UDP protocol works. The transfer is one-way(from Host1 to Server1) with no acknowledgement from Server1. This allows for high-rate data transfer and is beneficial in applications where data transfer is needed in real-time.

SDN supports the building and testing of such network scenarios with no reliance on infrastructure or hardware equipment, reduces expenses, and provides faster and more convenient control of the entire network.

5 Task 4 - Multicast Video Stream Service

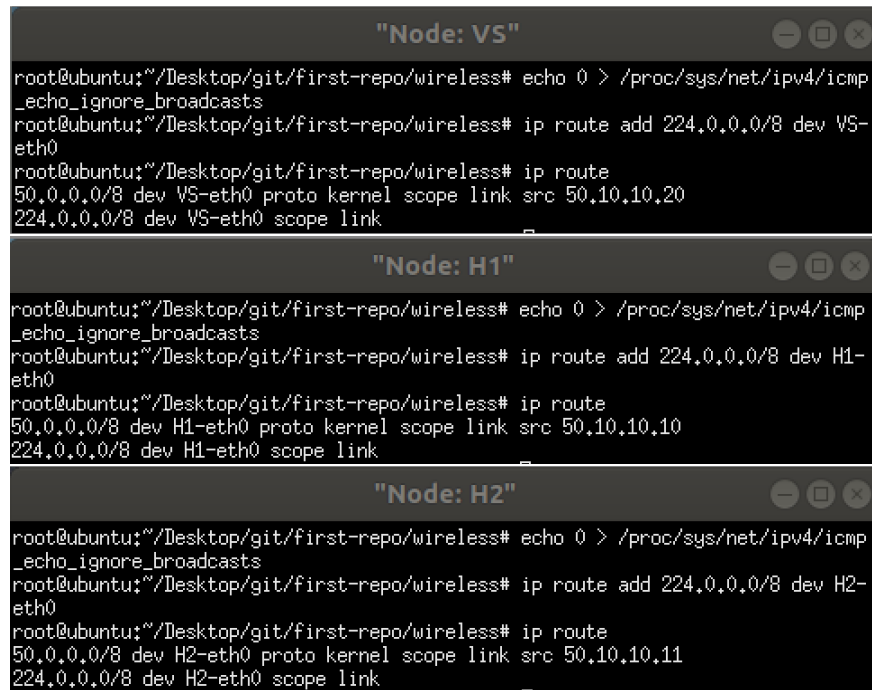
The final experiment emulates a local area network(LAN) where we test the multicast video streaming over the network for the new building to facilitate learning. Multicasting (Toh 1997) in networking is a type of communication which is also known as one-to-many communication. In a connectionless network, multicast basically involves replicating packets and distributing them to multiple receivers. A multicast group refers to a set of senders and receivers who are involved in a multi-party communication session. It is always advantageous to use multicast rather than multiple unicast communication since it provides an efficient means of communication among selected nodes within a large network (Murthy & Manoj 2004).

5.1 Design and Configuration

The network to be tested consists of 2 switches, a video source acting as the server, and three host devices which could be any ethernet-enabled device. Table 5 contains the name and IP address for configuration on host devices. The host devices will have the same network address and the configuration to allow for multicast communication is activated on video source(VS), host1(H1), and host2(H2) which creates a multicast group for these devices.

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 details



```
"Node: VS"
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

"Node: H1"
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

"Node: H2"
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 17: Activating multicast for required nodes

Fig. 16 shows the steps in activating the multicast on the required nodes. This can be done directly on the Mininet or at the node terminal when accessed with 'xterm' command on Mininet. The commands;

- `echo 0 > /proc/sys/net/ipv4/icmp_ignore_broadcasts`: allow node response to Internet Control Message Protocol(ICMP) echo request(ping request) broadcasted to every host on the network
- `ip route add 224.0.0.0/8 dev <interface>`: configures route that ensures multicast traffic for 224.0.0.0/8 address range is sent through the interface specified

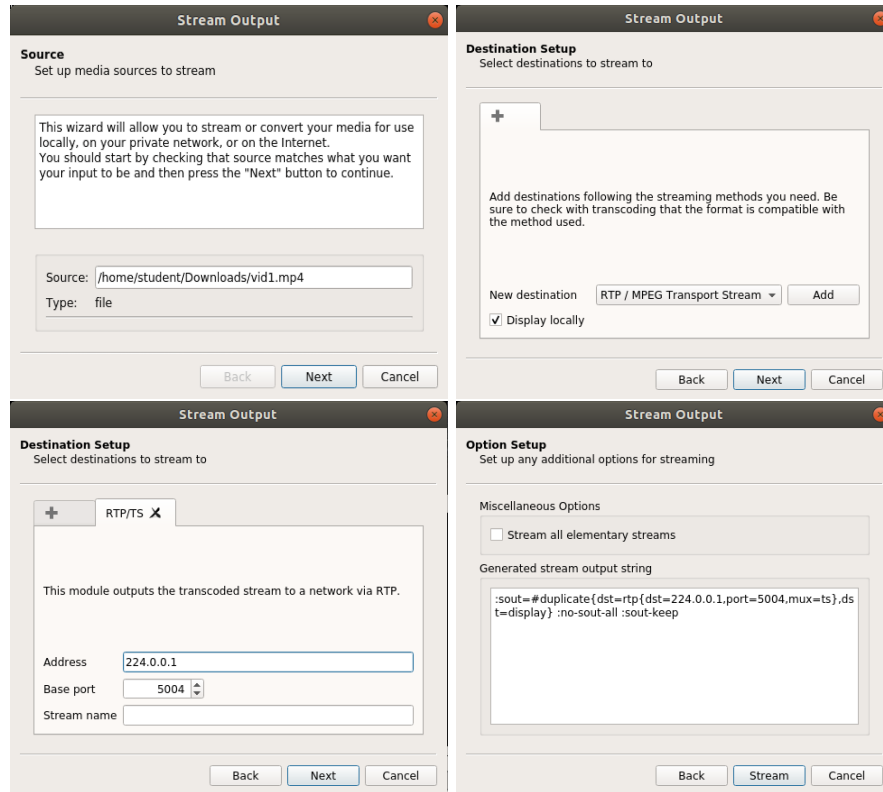


Figure 18: Multicast streaming setup at video source

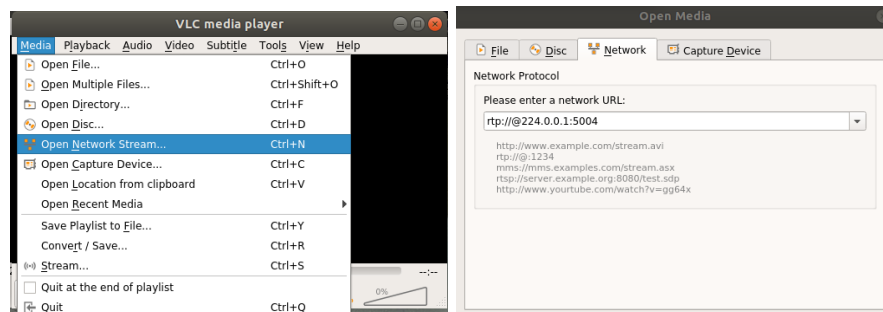


Figure 19: Multicast video connection at multicast host

After routing configuration, the video stream output is initiated from the video source on VLC through the terminal CLI of the video source, the command `vlc-wrapper` launches VLC where we select the video file,

transfer protocol, and provide the network to which the streaming will be sent across. On the host terminal CLI, VLC is launched with the same command and the multicast video stream is connected using the URL `rtp://@224.0.0.1:5004`. Fig. 17 and 18 show these steps taken out respectively.

5.2 Results and Analysis

Once all configurations are completed, the video as seen in Fig. 20 has been streamed from the source across the multicast network where the two hosts(1 & 2) can have the video output on the VLC player.

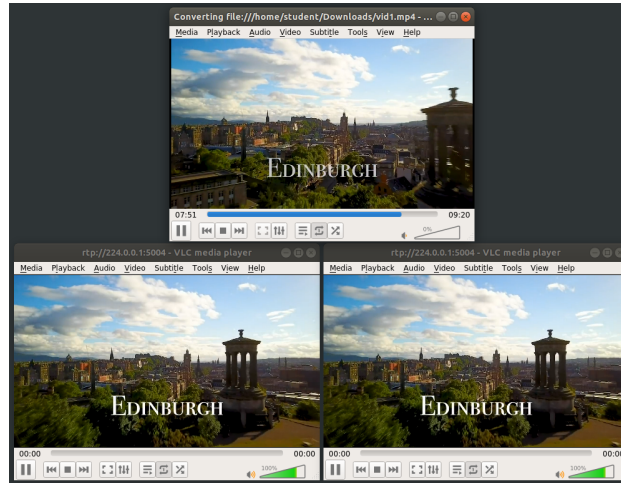


Figure 20: Multicast video stream from source to hosts

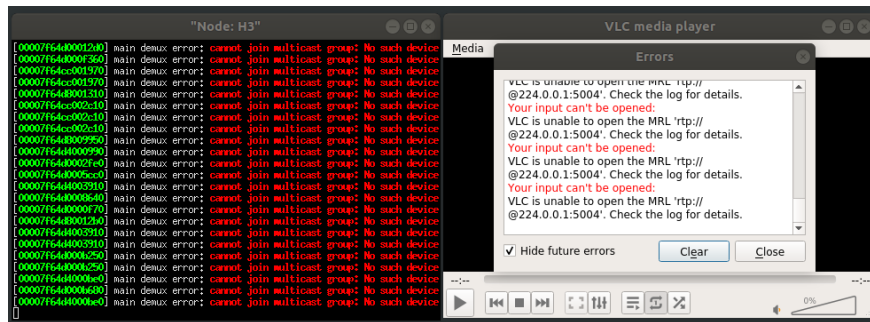


Figure 21: Failed video stream at host 3

The third host who is not part of the multicast group cannot stream the video as shown in Fig. 21. This confirms that the streaming is a multicast stream and only host devices within the multicast group can get access to the video stream. Fig. 22 shows a snippet of a Wireshark capture where we can see from the Ethernet section of the packet that the destination MAC address is of a multicast type(IPv4mcast_01).

During this process, we encounter a protocol known as the Real-Time Transfer Protocol(RTP) which is used for transmission of media files(audio or graphics) over an IP network. Since it is also a form of UDP connection it does not require acknowledgement in setting up the communication which makes it able to support voice and video communications over IP and online gaming.

No.	Time	Source	Destination	Protocol	Length	Info
7484	53.404170051	50.10.10.20	224.0.0.1	UDP	1370	55011 → 5004 Len=1328
7485	53.430292161	50.10.10.20	224.0.0.1	UDP	1370	55011 → 5004 Len=1328
7486	53.441390481	50.10.10.20	224.0.0.1	UDP	1370	55011 → 5004 Len=1328

▶	Frame 3110: 1370 bytes on wire (10960 bits), 1370 bytes captured (10960 bits) on interface 0
▼	Ethernet II, Src: b6:fb:57:77:ea:3c (b6:fb:57:77:ea:3c), Dst: IPv4mcast_01 (01:00:5e:00:00:01)
▶	Destination: IPv4mcast_01 (01:00:5e:00:00:01)
▶	Source: b6:fb:57:77:ea:3c (b6:fb:57:77:ea:3c)
▶	Type: IPv4 (0x0800)
▶	Internet Protocol Version 4, Src: 50.10.10.20, Dst: 224.0.0.1
▶	User Datagram Protocol, Src Port: 55011, Dst Port: 5004
▶	Data (1328 bytes)

Figure 22: Packet details captured on Wireshark

6 Conclusion

At the end of the experiments, it is confirmed that SDN provides greater advantages to network developers or operators in the manner of network control and also a summary of the network compared to the traditional way of computer networking. There are also vast improvements in the deployment of networks, configuration, fault tolerance, and scalability as demonstrated in the tasks.

The table below shows a comparison of some features between traditional and software-defined networking.

Feature	Traditional Networking	Software-Defined Networking
Control plane	Distributed across devices	Centralised & managed by SDN controller
Network Configuration	Manual configuration	Configuration via software and APIs
Agility & Flexibility	Difficult to adapt quickly	Quick reconfiguration
Cost Efficiency	High cost of infrastructure	Low cost due to virtualisation
Programmability	Difficult to program	Easy to program via SDN controller

To conclude, we point out the achievements of software-defined networking in the emulation of the wireless and ad-hoc network, in testing the Voice over IP(VoIP) communication, and the multicast streaming service over a network. The configuration and deployment of the networks require less time and effort, are at a very low cost, and can be easily managed due to the agility and flexibility offered. Researchers and network designers can test out new ideas and innovations while network concepts can be taught in an education setting in a very efficient and convenient way.

7 References

References

- Conrad, E., Misenar, S. & Feldman, J. (2012), Chapter 3 - domain 2: Telecommunications and network security, in E. Conrad, S. Misenar & J. Feldman, eds, ‘CISSP Study Guide (Second Edition)’, second edition edn, Syngress, Boston, pp. 63–141.
URL: <https://www.sciencedirect.com/science/article/pii/B9781597499613000030>
- de Oliveira, R. L. S., Schweitzer, C. M., Shinoda, A. A. & Prete, L. R. (2014), ‘Using mininet for emulation and prototyping software-defined networks’, pp. 1–6.
- Haji, S. H., Zeebaree, S. R. M., Saeed, R. H., Ameen, S. Y., Shukur, H. M., Omar, N., Sadeeq, M. A. M., Ageed, Z. S., Ibrahim, I. M. & Yasin, H. M. (2021), ‘Comparison of software defined networking with traditional networking’, *Asian Journal of Research in Computer Science* **9**(2), 1–18.
URL: <https://journalajrcos.com/index.php/AJRCOS/article/view/169>
- Kreutz, D., Ramos, F. M. V., Veríssimo, P. E., Rothenberg, C. E., Azodolmolky, S. & Uhlig, S. (2015), ‘Software-defined networking: A comprehensive survey’, *Proceedings of the IEEE* **103**(1), 14–76.
- Lara, A., Kolasani, A. & Ramamurthy, B. (2014), ‘Network innovation using openflow: A survey’, *IEEE Communications Surveys Tutorials* **16**(1), 493–512.
- Mokoena, K. M. R., Moila, R. L. & Velepini, P. M. (2023), 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)’, pp. 1–5.
- Murthy, C. S. R. & Manoj, B. S. (2004), *Ad hoc wireless networks: architectures and protocols*, Prentice Hall PTR, Upper Saddle River, N.J.
- Toh, C. . (1997), *Wireless ATM and AD-HOC networks: Protocols and architectures*, Kluwer Academic, Boston, [Mass.].

8 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

#function for wireless network topology
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( 'STA1', 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( 'STA2', 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( 'STA3', 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' ) #control for the OpenFlow switches
    net.setPropagationModel( model="logDistance", exp=5 ) #define the rate of
        signal loss in the network model

    info( "**configuring wifi nodes**\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] ) #start APs based on configurations from the controller
ap2.start( [c0] )
ap3.start( [c0] )
ap4.start( [c0] )
ap5.start( [c0] )

info( "**running CLI**\n" )
CLI( net ) #allows interaction with network on a terminal

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 )
    # Mininet_wifi func with wireless links specified
    # simulate signal interference of wireless devices in range transmitting
    # simultaneoously

    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" )
# test 'batman_adv', 'batmand', 'olsrd' 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 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() ) }
```