National College of Ireland

Performance Optimization of Software Defined Networks By Dynamic Placement of Controllers

MSc Research Project

Cloud Computing

Imran Abdul Rahman Syed
StudentID: x16145119

School of Computing
National College of Ireland

Supervisor: Mr. Vikas Sahni

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Imran Abdul Rahman Syed
Student ID:	x16145119
Programme:	Cloud Computing
Year:	2020
Module:	MSc Research Project
Supervisor:	Mr. Vikas Sahni
Submission Due Date:	17/08/2020
Project Title:	Performance Optimization of Software Defined Networks By Dynamic Placement of Controllers
Word Count:	5789
Page Count:	19

Signature:	
Date:	17th August 2020

Contents

Performance Optimization of Software Defined Networks By Dynamic Placement of Controllers

Imran Abdul Rahman Syed
x16145119
MSc in Cloud Computing

17th August 2020

Abstract

Software Defined Networking is a new model in the era of cloud computing. It provides programmability which in turn helps to configure and administer a network with continuously changing network state. Software Defined Networking has logical plane separated from the data plane. The logical plane is centralized where all the decision making happens. The data plane receives logical decisions from the control plane and works on forwarding of data packets. The control plane consists of the controllers and the data plane consists of the switches. A typical SDN has a controller which handles the routing decisions and switches that work on the decisions made by the controller. However, this has constraints as a single controller can be overloaded and become a single point of failure. This reduces scalability and performance of the whole network. Recent works have been put forward to mark these issues. Most of them suggested the placement of multiple controllers, instead of a single one, which can work concurrently to serve the switches better. However, this approach raises another problem. Since the network's state keeps changing, placing of multiple controllers becomes more challenging. Hence, the problem of Dynamic Controller Placement comes in to picture. The number of controllers in a network should be increased dynamically depending on the number of switches and hosts. This will lead to load balancing among the controllers. When the controllers in a network are balanced, the performance increase can been seen in terms of increased throughput and reduced latency.

• **Keywords**: Software Defined Networking, Dynamic controller placement, Controller Load Balance, Throughput

1 Introduction

Software Defined Networking has been widely adopted by organizations as it suits well to the needs of cloud computing. Surpassing the traditional internet, due to its lack of adaptability to huge networks such as cloud Datacentres, Software Defined Networking meets the key requirements of Cloud Computing. It provides programmability to configure and administer a network. In SDN, the logic and data planes are separated, where the logic plane is centralized for decision making (Bari et al.; 2013). With a typical set-up of the SDN, as the network grows, the controller gets overloaded and also becomes a single point of failure. Moreover, finding the best possible placement locations gets difficult. Similarly, the flow set-up time, the time taken to assign a forwarding rule to an Open Flow switch, increases which in turn affects the performance of networks and applications (Bari et al.; 2013).

Hence, the idea of placing multiple controllers, which are physically distributed but logically centralized, was proposed by (Hassas Yeganeh and Ganjali; 2012) and (Tootoonchian and Ganjali; 2010). This gives rise to another problem which comes up because of the dynamic nature of the networks. Since the networks' state keeps changing continuously, placing of controllers cannot be static. The best possible solution should consider the location of controllers, flow set-up time and controller communication overhead. In a multiple-controller environment, the controllers need to communicate to each other to have an updated network state. This communication can cause slowness. Moreover, the controller placement strategy should dynamically alter the number of controllers as well as the location, as traffic may change at different locations. Hence, the controller communication overhead problem cannot be ignored (Bari et al.; 2013). (Bari et al.; 2013) and (Lange et al.; 2015) are the works done on the aforementioned problems. (Bari et al.; 2013) formulated the problem of Dynamic Controller Placement and approached with 2 algorithms which worked together to solve the controller placement problem. They came up with a Greedy Knapsack algorithm, where they treated controllers as the knapsacks and switches as the objects to be added to those knapsacks. The algorithm assigned the switches to the controllers in each iteration. If any given switch was not assigned to a controller, that particular controller would be turned inactive. When all the switches were assigned to the controllers, the iteration would stop. The output of this algorithm would then go as an input to their second algorithm called Dynamic Controller Provisioning with Simulated Annealing (DCP-SA). This algorithm enhances the controller-switch assignments further. However, in the Greedy Knapsack algorithm has a probability of having unassigned switches. The algorithm assigns these switches to any random controller. This can affect performance significantly. (Lange et al.; 2015) also works on inter-controller latency and controller to node latency using Pareto based Simulated Annealing algorithm. They used POCO framework, implemented in MATLAB, which comprehensively analyses all the probable placements of the desired network topology. It considers various scenarios with respect to NFV also. This work proposes an algorithm to add controllers in a network, based on the number of switches. Furthermore, to map and remap the switches based whenever new controllers are added. The Algorithm is called as Round Robin Switch scheduler, which assigns switches one by one to each controller in turns making sure the load of switches on the controllers is balanced.

1.1 Research Question

Can the performance of Software Defined Networks be enhanced by dynamic placement of controllers using Round Robin Scheduler Algorithm?

1.2 Research Objective

To address the problem raised in the research question, the below mentioned objectives will be followed. Section 2 describes the SDN Architecture with detailed explanation of its components. Section 3 critically analyses the existing literature on multiple controllers problem and proposed solutions. It also describes the problem of Dynamic Controller Placement. Section 4 describes the methodology by explaining the algorithm of addition and removal of controllers in a multi controller SDN. Section 5 details the practical implementation of the proposed logic with brief description of the tools and technologies used. The evaluation of the work done has been described in section 6, with appropriate measurements and visual interpretations. Section 7 concludes with brief re-iteration of the work done and potential areas to focus on as the future work.

2 SDN Architecture

In this section, Software Defined Networking architecture will be discussed. Different SDN layers and components will be discussed. Software Defined Networking divides the control plane and data plane which makes the switches mere forwarding devices. The decision making takes place on logically centralized controllers (Zhang, Cui, Wang and Zhang; 2017). The controllers are in charge of policy enforcement, network configuration, topology management, link discovery etc. A Software Defined Network consists of 3 layers i.e. Application layer, Control layer and Infrastructure layer. It also consists of two interfaces i.e. Southbound interface and Northbound interface.

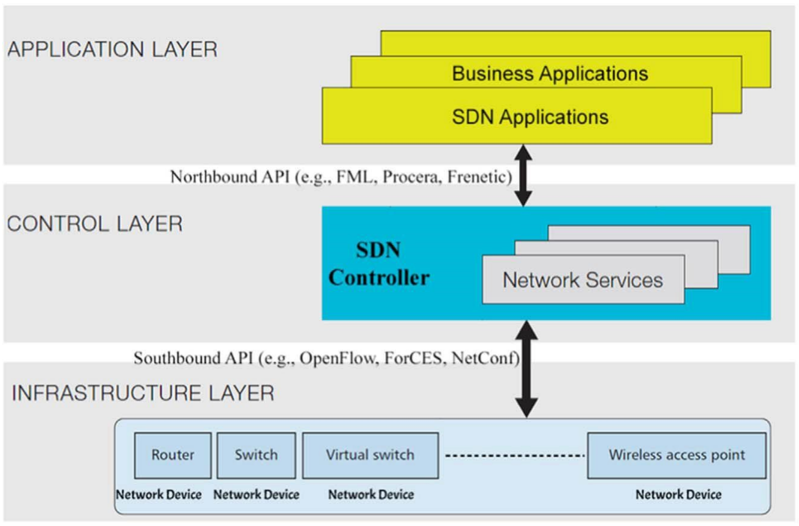


Fig.1 SDN Architecture (Zhang, Cui, Wang and Zhang; 2017).

2.1 Infrastructure Layer

Switches and various network devices constitute the infrastructure layer. The flow forwarding information will be passed on to the controllers by the infrastructure layer via the data-control plane. This plane is also referred as Southbound APIs. The infrastructure layer is responsible to send traffic in accordance with the logic of control layer. There are three main components of the switch, Secure Channel, Flow Table and OpenFlow protocol. Every OpenFlow switch is linked to the controller using Secure Channel interface. Flow table stores all forwarding policies. The southbound API which links the control plane and data plane is the OpenFlow protocol (Zhang, Cui, Wang and Zhang; 2017).

2.2 Southbound APIs

The infrastructure layer and the control layer are linked by the Southbound APIs. Providing communication between the controllers and other network devices, for finding the network topology, update flow table and put in force the control policies, is the key objective of the Southbound APIs. The most widely used Southbound API for SDN is the OpenFlow protocol (Zhang, Cui, Wang and Zhang; 2017).

2.3 Control Layer

It is the central layer that is in charge of managing the network. It can have a single controller or many controllers. It forms required network policies and configures switches by having overall network information. It also communicates with different controllers through east-west interface (Zhang, Cui, Wang and Zhang; 2017).

2.4 Northbound APIs

It is a link in between the application layer and the control layer. Making applications to utilize network resources and capabilities is the main objective of the Northbound APIs. In contrast to the

Southbound APIs, the Northbound APIs does not have standard protocols yet. This Is mainly because of the fact that there are different characteristics of controllers and applications (Zhang, Cui, Wang and Zhang; 2017).

2.5 Application Layer

SDN applications and services developed as per user specifications constitutes the application layer. Both the data plane switches and the control layer can be accessed by the application layer through control layer and Northbound APIs respectively (Zhang, Cui, Wang and Zhang; 2017).

3 Literature Review and Related Work

There are a number of multiple controller platforms that have been worked upon previously. These works were tailored for different applications and requirements. The works done previously were of two approaches. One of them involving shared datastores in order to share the network state and other ones working based on the traffic conditions.

(Bari et al.; 2013) formulated the problem of Dynamic Controller Placement and approached with two algorithms which worked together to solve the controller placement problem. They came up with a Greedy Knapsack algorithm, in which they treated controllers as the knapsacks and switches as the objects to be added to those knapsacks. The algorithm assigned the switches to the controllers in each iteration. If any given switch was not assigned to a controller, that particular controller would be turned inactive. When all the switches were assigned to the controllers, the iteration would stop. The output of this algorithm would then go as an input to their second algorithm called Dynamic Controller Provisioning with Simulated Annealing (DCPSA). This algorithm enhances the controller-switch assignments further. However, in the Greedy Knapsack algorithm has a probability of having unassigned switches. The algorithm assigns these switches to any random controller. This can affect performance significantly.

(Lange et al.; 2015) works on inter-controller latency and controller to node latency using Pareto based Simulated Annealing algorithm. They used POCO framework, implemented in MATLAB, which comprehensively analyses all the probable placements of the desired network topology. It considers various scenarios with respect to NFV also.

(Hassas Yeganeh and Ganjali; 2012) devised a framework which divides the controllers in two layers. The bottom layer consists of local controllers which are unaware of the network state. Local control applications alone are run in this layer. The upper layer consists of a root controller which is aware of the overall network state. The network operators will be capable of replicating the local controllers on the go and reduce overhead on the top layer using Kandoo's design.

(Botelho et al.; 2014) is the work done based on a shared replicated datastore that saves the overall network state. The architecture of Smartlight incorporates a primary centralized controller and backup controllers. The idea is to fail over the backup controllers in when the primary centralized controller goes down. The cache of the datastore is managed by the primary controller.

A clustering based distributed datastore architecture has been proposed by (Kim et al.; 2017). It comprises of Shards, part of the datastore. All cluster members contain a shard. In order to have consistency, the shard leader alone has the authority to receive updates. Dispersing the shard leaders throughout the controllers is the major objective of the OpenDaylight platform.

(Lee et al.; 2014) worked on developing IRIS, a recursive SDN controller platform which involves horizontally scalable architecture. It increases performance on the go by adding servers to the controller cluster. It reduces management cost, transport interoperation by adopting domainbased network abstraction.

Elasticon (Dixit et al.; 2014) was proposed as framework for elastic distributed controllers. They worked on the idea that the controller pool would expand and contract based on the traffic. The load is balanced across controllers. For balancing load among many controllers and incorporating elasticity, novel switch migration protocol and algorithms were used. Load and imbalance among controllers is identified by Elasticon, which monitors controllers in a timely manner and the lightly loaded controllers are assigned with switches from the overloaded ones. Since OpenFlow is not suited for disruption free switch assignment, assignment of a switch from one controller to another might lead to disturbance majorly affecting Datacentre applications. Hence, they came up with a four-phase migration protocol which is compliant to the Open Flow protocol. Generating a single trigger which is communicated over to the controllers, to perform the hand off without any disturbance, is the main functionality of the protocol.

Another novel approach in (Krishnamurthy et al.; 2014) is used for allocation of switches and partitions of application state to the distributed controllers. The assignment problem has been developed with strict time constraints. They have claimed to achieve reduction of flow set-up time and controller operating costs by 44% and 42% respectively.

(Moazzeni et al.; 2018) works on improving the reliability of controllers by devising a new method called "Reliable Distributed SDN". Their idea is to divide the network into multiple domains, each having a master controller along with one or many slave controllers. The reliability rate of the controllers is found out by the master controller of each domain by using the proposed formula. Each controller receives the reliability rate in regular time intervals via edge switches through the East/West bound interface. To check the status of all the controllers on a regular time interval, the controller with the best reliability rate would be assigned as a coordinator. In case of a controller discovered to be inactive, the coordinator makes a decision on selecting the best possible active controllers to take over the sub network. The coordinates do this by getting the reliability rates from the cache and will run the fast recovery till the failed controller is fixed.

3.1 OpenFlow protocol

OpenFlow protocol is an open standard southbound API for Software Defined Networking. It is one of the most widely used protocols (Zhang, Cui, Wang and Zhang; 2017). Open Networking Foundation, an organization committed to SDN's advancement, governs this protocol's standards. To enhance reliability of controllers, OpenFlow describes a multi-controller approach from version 1.2.0 (Zhang, Cui, Wang and Zhang; 2017). Creating communication of switches with multiple controllers was put forward in version 1.2.0. The protocol is getting enhanced since version 1.0.0 to version 1.5.0 (Zhang, Cui, Wang and Zhang; 2017). Table 1 describes different the development of the OpenFlow protocol.

Version	Date	Features
OpenFlow 1.0.0	Dec, 2009	Fundamental architecture, single flow table, IPv4
OpenFlow 1.1.0	Feb, 2011	Multiple flow table, group table, MPLS and VLAN
OpenFlow 1.2.0	Dec, 2011	IPv6, multiple controllers
OpenFlow 1.3.0	June, 2012	Single flow measure, IPv6 extend header
OpenFlow 1.4.0	October, 2013	Flow table synchronization mechanism, bundling message
OpenFlow 1.5.0	December, 2014	et type identification process, egress table, scheduled bundle ex

Table 1: OpenFlow protocol versions

(Zhang, Cui, Wang and Zhang; 2017)

3.2 Architecture of multiple controllers

SDN was initially designed to handle the network switches from a centralized controller. However, one controller managing the switches gets overloaded and becomes a single point of failure. To combat this issue, the concept of having multiple controllers came into existence. Based on the functionality of controllers, the architecture of SDN multiple controllers comes in two classifications: centralized and distributed architecture (Zhang, Cui, Wang and Zhang; 2017). A centralized controller architecture has only one controller, whereas a distributed architecture has multiple controllers which are logically centralized (Zhang, Cui, Wang and Zhang; 2017). The centralized controller architecture was worked upon by (Botelho et al.; 2014). It was done based on a shared replicated datastore that saves the overall network state. The architecture of Smartlight incorporates a primary centralized controller. There are other backup controllers which come into play only in case failure of the primary one. (Lee et al.; 2014) involved scalable architecture. It increases performance on the go by adding servers to the controller cluster. It reduces management cost, transport interoperation by adopting domain-based network abstraction. The centralized architecture has its limitations in terms of performance and scalability. The distributed architecture addresses the concerns related to the former. Smartlight (Botelho et al.; 2014) devised backup controllers for fault tolerance. However, it does not fully eliminate the issues of performance and scalability as single controller still handles requests from all the switches. In case of IRIS (Lee et al.; 2014), the scalability is relatively high.

This section discussed two major categories of controller architecture in SDN: centralized and distributed. The centralized architecture has only a single controller responsible for managing the underlying switches. In contrary, the distributed architecture has many controllers working together which in turn performs better in terms of performance and scalability.

3.3 Dynamic controller placement

Even though physically dispersed, the distributed controllers should work in a logically centralized way. This requires having a consistent overall view of the network which demands the sharing of information between controllers. Issues arise while communicating information between controllers, such as deciding the number and location of controllers (Zhang, Cui, Wang and Zhang; 2017). The placement of controllers dynamically comes with two major problems: Controller consensus overhead and controller to switch latency. In order to appropriately place the controllers, a number of metrics have been put forward (Zhang, Cui, Wang and Zhang; 2017). (Bari et al.; 2013) has worked on different costs as parameters to decide upon the correct placement of controllers.

- Statistics collection cost – For gathering information from their assigned switches, the number of messages required for the controllers every second, is the statistics collection cost (Bari et al.; 2013).

- Flow setup cost – Initiating flow related rules between switch and controller generates flow setup cost (Bari et al.; 2013). Initial path setup request, intermediate path setup request and the rule installation cost are further derived from the flow setup cost (Bari et al.; 2013).

- Synchronization cost – To maintain the overall view of the network, the controllers need to communicate with each other. The number of messages sent and received among controllers is the synchronization cost (Bari et al.; 2013).

- Switch reassignment cost – The cost incurred while allocating a switch to a new controller is the switch reassignment cost (Bari et al.; 2013).

The POCO framework used by (Lange et al.; 2015) considered the problem of controller-to-controller latency to make sure the latency is as low as possible during the synchronization of controllers. Their work represented the calculation in Fig 2. They did so by representing the controllers in colours. i.e. every controller is shaded according to the distance to the most far away controller.

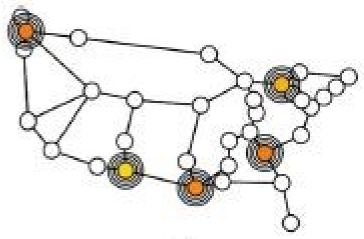


Fig.2 Controller-to-Controller latency (Lange et al.; 2015).

Therefore, the POCO framework in (Lange et al.; 2015) has considered both controller to controller latency and node to controller latency. (Lange et al.; 2015) also points out that the latencies among controllers and controller to switch latency is low in a tightly coupled clusters while it is vice versa in case of a widely dispersed layout of controllers. Hence, POCO framework gives the options to the decision makers about their choices while finding out possible placements. While (Lange et al.; 2015) builds a Pareto based optimal placement framework POCO for evaluating all the possible placements for given topologies, it does not consider the traffic factor. POCO is a Heuristic based approach which can be used for large-scale networks as well. However, as the networks grow, one the most important factors to consider is the network traffic, which is ever-changing, Hence, we can conclude that POCO is not dynamic in nature, whereas (Bari et al.; 2013) considers various metrics, described in cost measurements and also considers traffic. The algorithms used by (Bari et al.; 2013), Greedy Knapsack and Simulated Annealing, consider the traffic matrix which gives (Bari et al.; 2013) an edge over (Lange et al.; 2015) which is not dynamic. In this section, we discussed the problems related to placement of controllers, the metrics to be considered and the placement considering traffic as a key factor. Placing controllers by speculating the number and location of controllers, dynamically is the best possible way to deal with the controller placement problem.

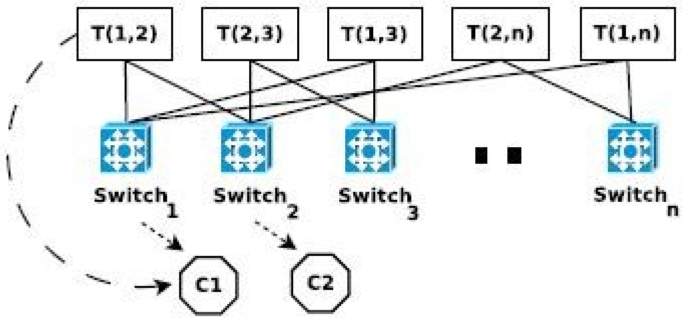


Fig.3 Application state for multi-tenant virtualized datacentre application (Krishnamurthy et al.; 2014).

In this section, we have discussed distributed datastore approach and application state partitioning approach. Both of these approaches are quite different than that of typical approaches to the controller placement problem. The former approach works by dividing the datastores into several parts called ‘shards’, whereas, the latter approach works on dividing the state of an application and assigning it, along with the SDN switch, to the controllers.

4 Methodology

The placement of controllers in a dynamic environment is the necessity of the day. In this work, we propose an algorithm known as Round Robin Switch Scheduler, which would allocate the OpenFlow switches to multiple controllers on a round robin basis. The idea is to provide load balancing among the controllers, so that the controllers are not overwhelmed with more number of switches. On a particular Network Topology, based on the number of controllers running, the algorithm assigns each controller, the switches in every turn. This gives the network better stability as the load of switches on a set of controllers is continuously balanced. Furthermore, it maintains the upper threshold of 70%. If this threshold is reached on any of the controllers, another controller is added dynamically and the switches are automatically redistributed based on the number of switches and controllers. The controllers would be removed or made inactive if the percentage on any of the controllers falls back to below 30%. This would increase the network performance by reducing the controller communication overhead and optimizing the network throughput.

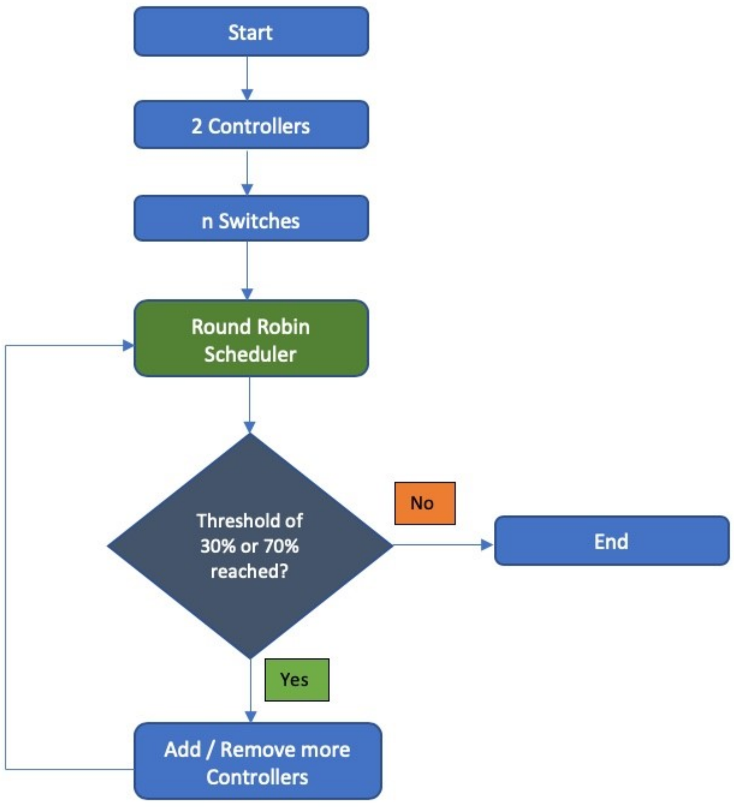

Start
2 Controllers
n Switches
Round Robin Scheduler
Threshold of 30% or 70% reached?
No
End
Yes
Add / Remove more Controllers

Fig.4 Flow Chart of Round Robin Switch Scheduler.

5 Implementation and Design

5.1 Tools and Technologies

5.1.1 Mininet and Mininet Python API

Algorithms can be tested and worked upon by the Mininet simulator. It has been worked upon to enhance the performance (Mallesh; 2017). As (Mallesh; 2017) has pointed out,(Barrett et al.; 2017) has worked on contrasting the performance of the Mininet and other test beds. They have inferred that Mininet simulator works better for in order to test. Moreover, it suits OpenFlow protocol. The real-time networks can be duplicated involving various topologies.(Mallesh; 2017) also mentioned that (Ortiz et al.; 2016) worked on evaluating the performance and compatibility of the Mininet framework. Datacentre type topology was duplicated and it was concluded that the performance of Mininet was better.

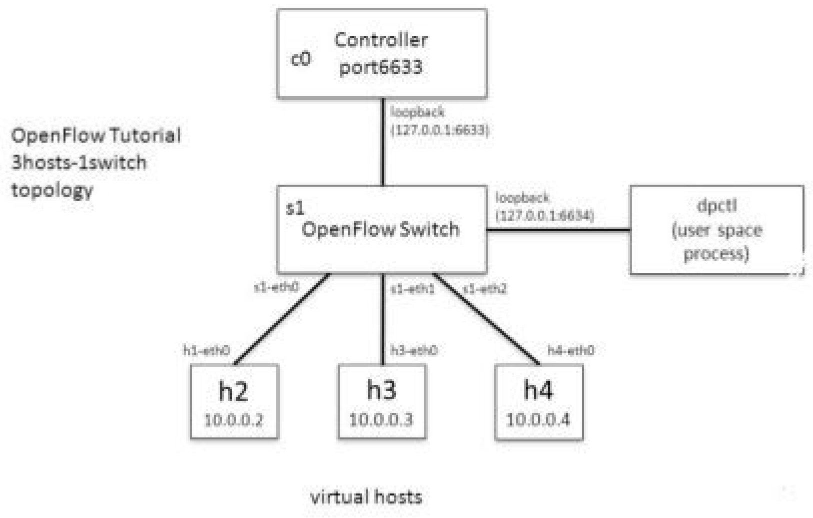


Fig.5 Active, Mininet Design (Mallesh; 2017).

Mininet allows you to create and run networks using the mn command. However, to implement customized topologies and better controlling the network deployment, the Mininet Python API is used. It uses Python programming for Software Defined Networking. In this research project, we have used the Python API for generating custom topologies with single and multi-controller scenarios. The network is deployed using python scripts.

5.1.2 POX Controller

POX is one of the famous types of Controllers used. The OVS or OpenFlow Switches talk to this controller seamlessly to receive the flow rules. In this project, we have deployed the POX controller on a separate LINUX based Ubuntu VM. The network topologies connect to this Controller using its IP address and the default port for Controllers.

5.1.3 Operating System and Virtualization Platform

Mininet simulator works with LINUX operating systems such as Ubuntu. In order to install, configure and work on Mininet, a 64-bit virtual machine with Ubuntu 20.04 Desktop Operating System, has been created in Oracle Virtual Box. It also provides a CLI for running Linux commands to manage the framework. The virtual machine will need to have at least 2 GB of RAM. As part of the network configurations, a Host-Only Network Adapter has been added with DHCP enabled. The IP address assigned by the DHCP will be used to SSH into the VM from the Host machine, in this case, a MacBook Pro.

5.1.4 XTerminal

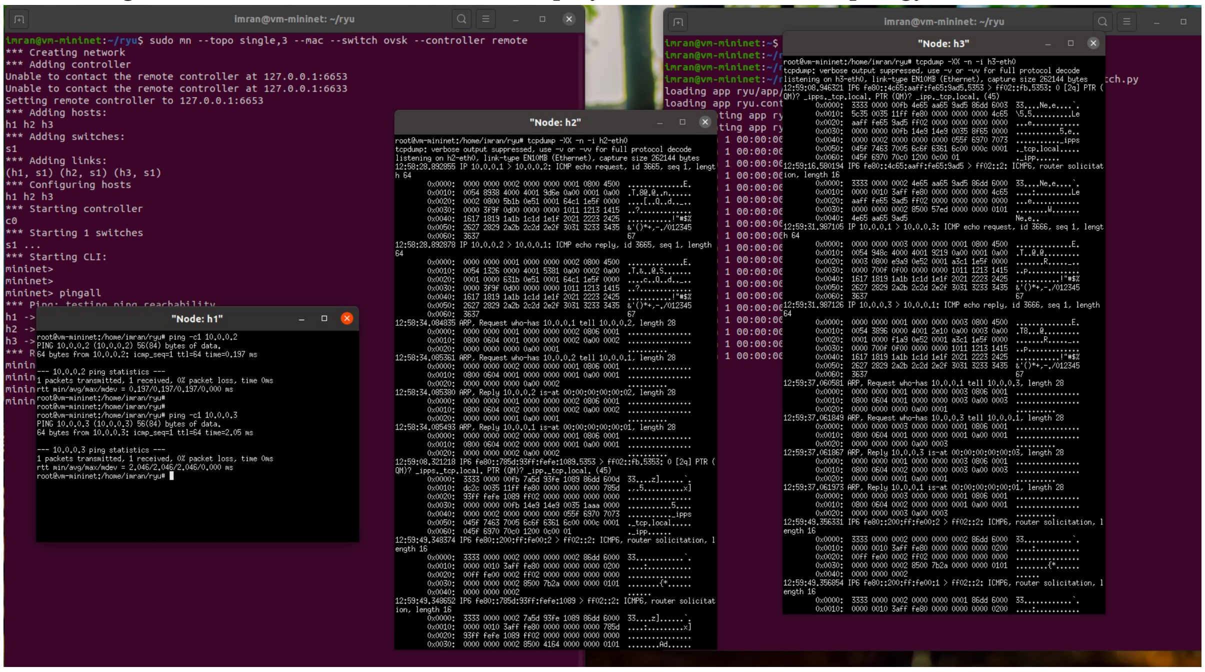
Along with the tools used, another important application used is XTerminal. It is used to leverage the graphical experience of applications in LINUX based Operating Systems.

In this project, XTerm is used extensively for the following needs:

- SSH into the VM from host - This is done by using the -X or -Y option in the SSH command

- Running Shell of each host in the deployed Mininet topology - This is used in the Mininet CLI when the network is running

- Running Wireshark on individual hosts deployed in the Mininet topology.



```
imran@vm-mininet:~/ryu$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
mininet>
mininet> pingall
*** Ping: testing ping reachability
h1 ->
h2 ->
h3 ->
*** R
minin
minin
minin
minin
```

```
"Node: h1"
root@vm-mininet:/home/imran/ryu# ping -c1 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.197 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.197/0.197/0.197/0.000 ms
root@vm-mininet:/home/imran/ryu#
root@vm-mininet:/home/imran/ryu#
root@vm-mininet:/home/imran/ryu# ping -c1 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=2.05 ms

--- 10.0.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.046/2.046/2.046/0.000 ms
root@vm-mininet:/home/imran/ryu#
```

```
"Node: h2"
root@vm-mininet:/home/imran/ryu# tcpdump -XX -n -i h2-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h2-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
12:58:28.892855 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 3665, seq 1, lengt
h 64
        0x0000:  0000 0000 0002 0000 0000 0001 0800 4500  ..............E.
        0x0010:  0054 8938 4000 4001 9d6e 0a00 0001 0a00  .T.8@.@..n......
        0x0020:  0002 0800 5b1b 0e51 0001 64c1 1e5f 0000  ....[..Q..d.._..
        0x0030:  0000 3f9f 0d00 0000 0000 1011 1213 1415  ..?.............
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
12:58:28.892878 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 3665, seq 1, length
64
        0x0000:  0000 0000 0001 0000 0000 0002 0800 4500  ..............E.
        0x0010:  0054 1326 0000 4001 5381 0a00 0002 0a00  .T.&..@.S.......
        0x0020:  0001 0000 631b 0e51 0001 64c1 1e5f 0000  ....c..Q..d.._..
        0x0030:  0000 3f9f 0d00 0000 0000 1011 1213 1415  ..?.............
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
12:58:34.084835 ARP, Request who-has 10.0.0.1 tell 10.0.0.2, length 28
        0x0000:  0000 0000 0001 0000 0000 0002 0806 0001  ................
        0x0010:  0800 0604 0001 0000 0000 0002 0a00 0002  ................
        0x0020:  0000 0000 0000 0a00 0001                 ..........
12:58:34.085361 ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28
        0x0000:  0000 0000 0002 0000 0000 0001 0806 0001  ................
        0x0010:  0800 0604 0001 0000 0000 0001 0a00 0001  ................
        0x0020:  0000 0000 0000 0a00 0002                 ..........
12:58:34.085380 ARP, Reply 10.0.0.2 is-at 00:00:00:00:00:02, length 28
        0x0000:  0000 0000 0001 0000 0000 0002 0806 0001  ................
        0x0010:  0800 0604 0002 0000 0000 0002 0a00 0002  ................
        0x0020:  0000 0000 0001 0a00 0001                 ..........
12:58:34.085493 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01, length 28
        0x0000:  0000 0000 0002 0000 0000 0001 0806 0001  ................
        0x0010:  0800 0604 0002 0000 0000 0001 0a00 0001  ................
        0x0020:  0000 0000 0002 0a00 0002                 ..........
12:59:08.321218 IP6 fe80::785d:93ff:fefe:1089.5353 > ff02::fb.5353: 0 [2q] PTR (
QM)? _ipps._tcp.local. PTR (QM)? _ipp._tcp.local. (45)
        0x0000:  3333 0000 00fb 7a5d 93fe 1089 86dd 600d  33....z]......`.
        0x0010:  dc2c 0035 11ff fe80 0000 0000 0000 785d  .,.5..........x]
        0x0020:  93ff fefe 1089 ff02 0000 0000 0000 0000  ................
        0x0030:  0000 0000 00fb 14e9 14e9 0035 1aaa 0000  ...........5....
        0x0040:  0000 0002 0000 0000 0000 055f 6970 7073  ..........._ipps
        0x0050:  045f 7463 7005 6c6f 6361 6c00 000c 0001  ._tcp.local.....
        0x0060:  045f 6970 70c0 1200 0c00 01              ._ipp......
12:59:49.348374 IP6 fe80::200:ff:fe00:2 > ff02::2: ICMP6, router solicitation, l
ength 16
        0x0000:  3333 0000 0002 0000 0000 0002 86dd 6000  33............`.
        0x0010:  0000 0010 3aff fe80 0000 0000 0000 0200  ....:...........
        0x0020:  00ff fe00 0002 ff02 0000 0000 0000 0000  ................
        0x0030:  0000 0000 0002 8500 7b2a 0000 0000 0101  ........{*......
        0x0040:  0000 0000 0002                           ......
12:59:49.348652 IP6 fe80::785d:93ff:fefe:1089 > ff02::2: ICMP6, router solicitat
ion, length 16
        0x0000:  3333 0000 0002 7a5d 93fe 1089 86dd 6000  33....z]......`.
        0x0010:  0000 0010 3aff fe80 0000 0000 0000 785d  ....:.........x]
        0x0020:  93ff fefe 1089 ff02 0000 0000 0000 0000  ................
        0x0030:  0000 0000 0002 8500 4164 0000 0000 0101  ........Ad......
```

```
"Node: h3"
root@vm-mininet:/home/imran/ryu# tcpdump -XX -n -i h3-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h3-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
12:59:08.946321 IP6 fe80::4c65:aaff:fe65:9ad5.5353 > ff02::fb.5353: 0 [2q] PTR (
QM)? _ipps._tcp.local. PTR (QM)? _ipp._tcp.local. (45)
        0x0000:  3333 0000 00fb 4e65 aa65 9ad5 86dd 6003  33....Ne.e....`.
        0x0010:  5c35 0035 11ff fe80 0000 0000 0000 4c65  \5.5..........Le
        0x0020:  aaff fe65 9ad5 ff02 0000 0000 0000 0000  ...e............
        0x0030:  0000 0000 00fb 14e9 14e9 0035 8f65 0000  ...........5.e..
        0x0040:  0000 0002 0000 0000 0000 055f 6970 7073  ..........._ipps
        0x0050:  045f 7463 7005 6c6f 6361 6c00 000c 0001  ._tcp.local.....
        0x0060:  045f 6970 70c0 1200 0c00 01              ._ipp......
12:59:16.580194 IP6 fe80::4c65:aaff:fe65:9ad5 > ff02::2: ICMP6, router solicitat
ion, length 16
        0x0000:  3333 0000 0002 4e65 aa65 9ad5 86dd 6000  33....Ne.e....`.
        0x0010:  0000 0010 3aff fe80 0000 0000 0000 4c65  ....:.........Le
        0x0020:  aaff fe65 9ad5 ff02 0000 0000 0000 0000  ...e............
        0x0030:  0000 0000 0002 8500 57ed 0000 0000 0101  ........W.......
        0x0040:  4e65 aa65 9ad5                           Ne.e..
12:59:31.987105 IP 10.0.0.1 > 10.0.0.3: ICMP echo request, id 3666, seq 1, lengt
h 64
        0x0000:  0000 0000 0003 0000 0000 0001 0800 4500  ..............E.
        0x0010:  0054 948c 4000 4001 9219 0a00 0001 0a00  .T..@.@.........
        0x0020:  0003 0800 e9a9 0e52 0001 a3c1 1e5f 0000  .......R....._..
        0x0030:  0000 700f 0f00 0000 0000 1011 1213 1415  ..p.............
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
12:59:31.987126 IP 10.0.0.3 > 10.0.0.1: ICMP echo reply, id 3666, seq 1, length
64
        0x0000:  0000 0000 0001 0000 0000 0003 0800 4500  ..............E.
        0x0010:  0054 3896 0000 4001 2e10 0a00 0003 0a00  .T8...@.........
        0x0020:  0001 0000 f1a9 0e52 0001 a3c1 1e5f 0000  .......R....._..
        0x0030:  0000 700f 0f00 0000 0000 1011 1213 1415  ..p.............
        0x0040:  1617 1819 1a1b 1c1d 1e1f 2021 2223 2425  ...........!"#$%
        0x0050:  2627 2829 2a2b 2c2d 2e2f 3031 3233 3435  &'()*+,-./012345
        0x0060:  3637                                     67
12:59:37.060581 ARP, Request who-has 10.0.0.1 tell 10.0.0.3, length 28
        0x0000:  0000 0000 0001 0000 0000 0003 0806 0001  ................
        0x0010:  0800 0604 0001 0000 0000 0003 0a00 0003  ................
        0x0020:  0000 0000 0000 0a00 0001                 ..........
12:59:37.061849 ARP, Request who-has 10.0.0.3 tell 10.0.0.1, length 28
        0x0000:  0000 0000 0003 0000 0000 0001 0806 0001  ................
        0x0010:  0800 0604 0001 0000 0000 0001 0a00 0001  ................
        0x0020:  0000 0000 0000 0a00 0003                 ..........
12:59:37.061867 ARP, Reply 10.0.0.3 is-at 00:00:00:00:00:03, length 28
        0x0000:  0000 0000 0001 0000 0000 0003 0806 0001  ................
        0x0010:  0800 0604 0002 0000 0000 0003 0a00 0003  ................
        0x0020:  0000 0000 0001 0a00 0001                 ..........
12:59:37.061973 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01, length 28
        0x0000:  0000 0000 0003 0000 0000 0001 0806 0001  ................
        0x0010:  0800 0604 0002 0000 0000 0001 0a00 0001  ................
        0x0020:  0000 0000 0003 0a00 0003                 ..........
12:59:49.356331 IP6 fe80::200:ff:fe00:2 > ff02::2: ICMP6, router solicitation, l
ength 16
        0x0000:  3333 0000 0002 0000 0000 0002 86dd 6000  33............`.
        0x0010:  0000 0010 3aff fe80 0000 0000 0000 0200  ....:...........
        0x0020:  00ff fe00 0002 ff02 0000 0000 0000 0000  ................
        0x0030:  0000 0000 0002 8500 7b2a 0000 0000 0101  ........{*......
        0x0040:  0000 0000 0002                           ......
12:59:49.356854 IP6 fe80::200:ff:fe00:1 > ff02::2: ICMP6, router solicitation, l
ength 16
        0x0000:  3333 0000 0002 0000 0000 0001 86dd 6000  33............`.
        0x0010:  0000 0010 3aff fe80 0000 0000 0000 0200  ....:...........
```

Fig.6 Using XTerm to display instances of multiple hosts.

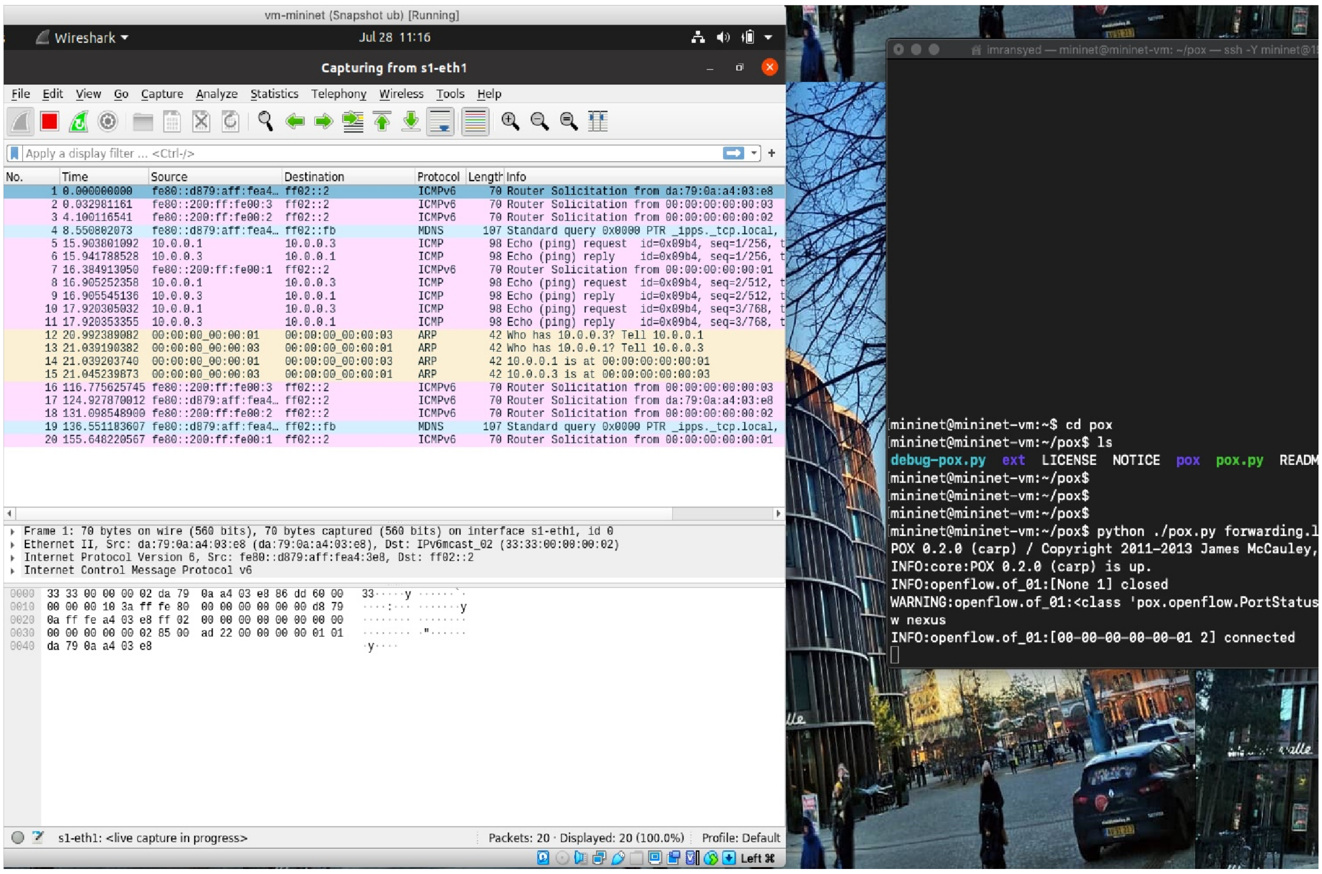
5.1.5 Gnuplot and Iperf

Iperf is utilized to measure speed and throughput of a network. In Mininet, it can be used to dump measurements readings to a file. We can use this file as an input and pass it on to the Gnuplot tool for generating some meaningful insights.
Using Gnuplot, we can chart graphs of throughput and latency from the input file carrying the measurement readings.

5.1.6 Wireshark

Wireshark is one of the best packet analyser tools and it is widely used by researchers, developers and networking engineers. It records detailed information of the packets generated on a particular network interface. In this project, Wireshark has been used on the Ubuntu VM and on individual hosts from the Mininet Network. It lists all the packets with details such as Time, Source (MAC address/IP address), Destination (MAC address/IP address), Protocol (ICMP, MDNS, OF etc).

Fig.7 Wireshark packet analysis of the Mininet Network connected a remote POX Controller.


vm-mininet (Snapshot ub) [Running]
Wireshark
Jul 28 11:16
Capturing from s1-eth1
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
imransyed — mininet@mininet-vm: ~/pox — ssh -Y mininet@1
s1-eth1: <live capture in progress>
Packets: 20 · Displayed: 20 (100.0%)
Profile: Default

```
No.  Time          Source                 Destination          Protocol Length Info
   1 0.000000000   fe80::d879:aff:fea4…   ff02::2              ICMPv6    70 Router Solicitation from da:79:0a:a4:03:e8
   2 0.032981161   fe80::200:ff:fe00:3    ff02::2              ICMPv6    70 Router Solicitation from 00:00:00:00:00:03
   3 4.100116541   fe80::200:ff:fe00:2    ff02::2              ICMPv6    70 Router Solicitation from 00:00:00:00:00:02
   4 8.550802073   fe80::d879:aff:fea4…   ff02::fb             MDNS     107 Standard query 0x0000 PTR _ipps._tcp.local,
   5 15.903801092  10.0.0.1               10.0.0.3             ICMP      98 Echo (ping) request  id=0x09b4, seq=1/256, t
   6 15.941788528  10.0.0.3               10.0.0.1             ICMP      98 Echo (ping) reply    id=0x09b4, seq=1/256, t
   7 16.384913050  fe80::200:ff:fe00:1    ff02::2              ICMPv6    70 Router Solicitation from 00:00:00:00:00:01
   8 16.905252358  10.0.0.1               10.0.0.3             ICMP      98 Echo (ping) request  id=0x09b4, seq=2/512, t
   9 16.905545136  10.0.0.3               10.0.0.1             ICMP      98 Echo (ping) reply    id=0x09b4, seq=2/512, t
  10 17.920305032  10.0.0.1               10.0.0.3             ICMP      98 Echo (ping) request  id=0x09b4, seq=3/768, t
  11 17.920353355  10.0.0.3               10.0.0.1             ICMP      98 Echo (ping) reply    id=0x09b4, seq=3/768, t
  12 20.992389082  00:00:00_00:00:01      00:00:00_00:00:03    ARP       42 Who has 10.0.0.3? Tell 10.0.0.1
  13 21.039190382  00:00:00_00:00:03      00:00:00_00:00:01    ARP       42 Who has 10.0.0.1? Tell 10.0.0.3
  14 21.039203740  00:00:00_00:00:01      00:00:00_00:00:03    ARP       42 10.0.0.1 is at 00:00:00:00:00:01
  15 21.045239873  00:00:00_00:00:03      00:00:00_00:00:01    ARP       42 10.0.0.3 is at 00:00:00:00:00:03
  16 116.775625745 fe80::200:ff:fe00:3    ff02::2              ICMPv6    70 Router Solicitation from 00:00:00:00:00:03
  17 124.927870012 fe80::d879:aff:fea4…   ff02::2              ICMPv6    70 Router Solicitation from da:79:0a:a4:03:e8
  18 131.098548900 fe80::200:ff:fe00:2    ff02::2              ICMPv6    70 Router Solicitation from 00:00:00:00:00:02
  19 136.551183607 fe80::d879:aff:fea4…   ff02::fb             MDNS     107 Standard query 0x0000 PTR _ipps._tcp.local,
  20 155.648220567 fe80::200:ff:fe00:1    ff02::2              ICMPv6    70 Router Solicitation from 00:00:00:00:00:01

▸ Frame 1: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface s1-eth1, id 0
▸ Ethernet II, Src: da:79:0a:a4:03:e8 (da:79:0a:a4:03:e8), Dst: IPv6mcast_02 (33:33:00:00:00:02)
▸ Internet Protocol Version 6, Src: fe80::d879:aff:fea4:3e8, Dst: ff02::2
▸ Internet Control Message Protocol v6

0000  33 33 00 00 00 02 da 79  0a a4 03 e8 86 dd 60 00   33····y ······`·
0010  00 00 00 10 3a ff fe 80  00 00 00 00 00 00 d8 79   ····:··· ·······y
0020  0a ff fe a4 03 e8 ff 02  00 00 00 00 00 00 00 00   ········ ········
0030  00 00 00 00 00 02 85 00  ad 22 00 00 00 00 01 01   ········ ·"······
0040  da 79 0a a4 03 e8                                  ·y····
```

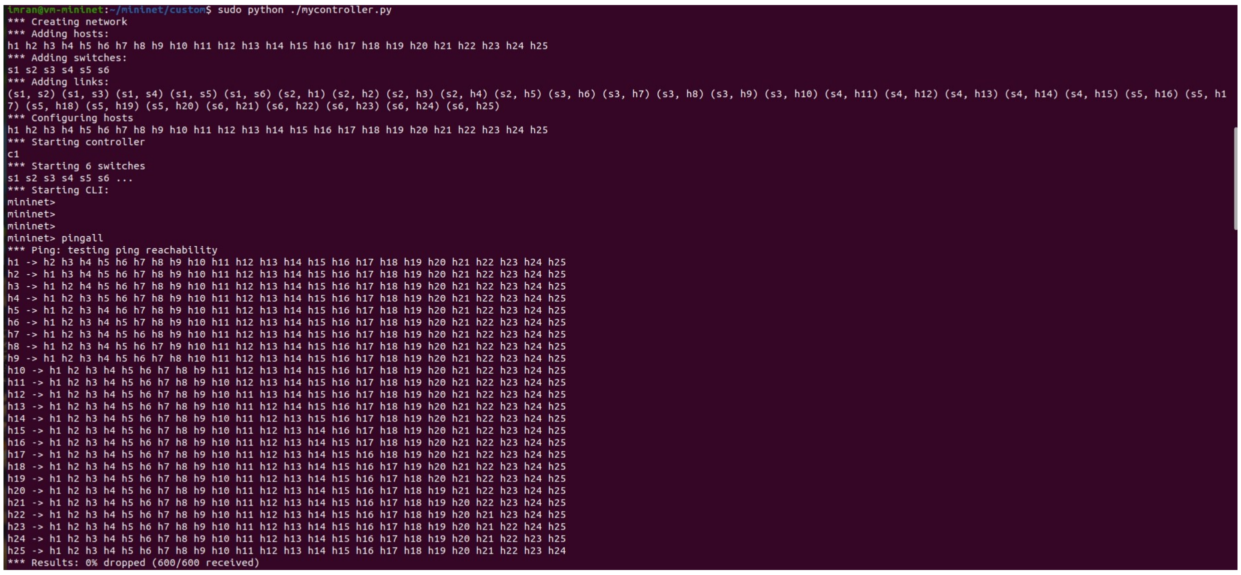
```
mininet@mininet-vm:~$ cd pox
mininet@mininet-vm:~/pox$ ls
debug-pox.py  ext  LICENSE  NOTICE  pox  pox.py  READM
mininet@mininet-vm:~/pox$
mininet@mininet-vm:~/pox$
mininet@mininet-vm:~/pox$
mininet@mininet-vm:~/pox$ python ./pox.py forwarding.l
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley,
INFO:core:POX 0.2.0 (carp) is up.
INFO:openflow.of_01:[None 1] closed
WARNING:openflow.of_01:<class 'pox.openflow.PortStatus
w nexus
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
```

5.2 Topology Generation

In order to address the issue of network congestion and reduced throughput because of having a single controller, we have demonstrated the following scenarios with single and multiple controllers environment.

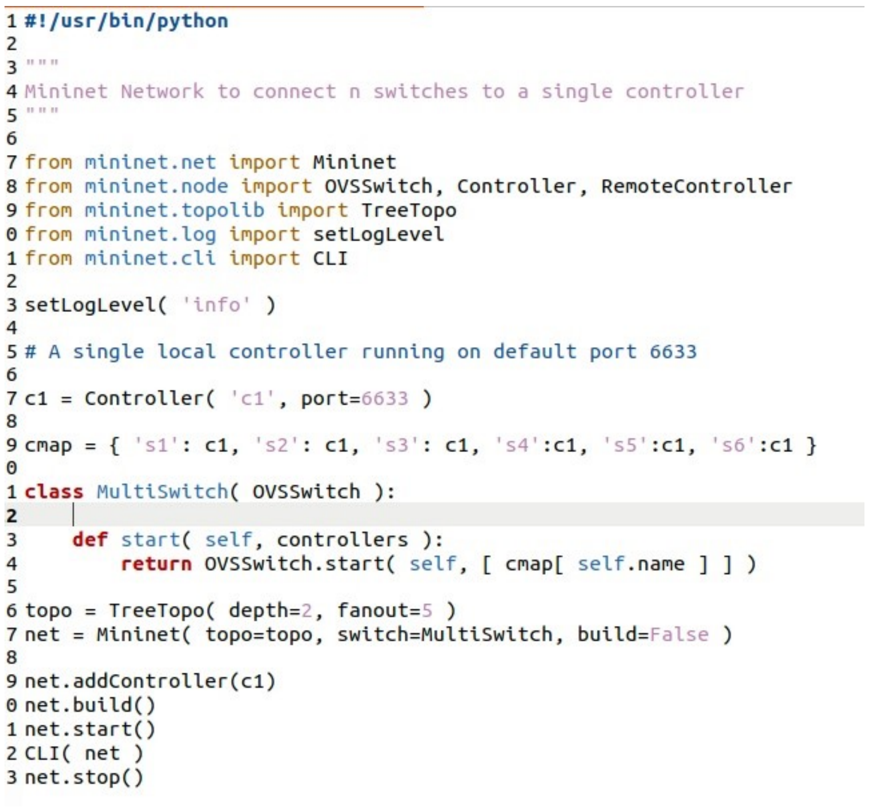
- **Scenario 1: Having a single controller**

  In this scenario, we have deployed a Mininet network with a single local Controller running on default port 6633. We have used tree topology with depth level 2 and Fanout 5. This will deploy 1 Controller, 6 OVSSwitches and 25 hosts.



```
imran@vm-mininet:~/mininet/custom$ sudo python ./mycontroller.py
*** Creating network
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
*** Adding switches:
s1 s2 s3 s4 s5 s6
*** Adding links:
(s1, s2) (s1, s3) (s1, s4) (s1, s5) (s1, s6) (s2, h1) (s2, h2) (s2, h3) (s2, h4) (s2, h5) (s3, h6) (s3, h7) (s3, h8) (s3, h9) (s3, h10) (s4, h11) (s4, h12) (s4, h13) (s4, h14) (s4, h15) (s5, h16) (s5, h1
7) (s5, h18) (s5, h19) (s5, h20) (s6, h21) (s6, h22) (s6, h23) (s6, h24) (s6, h25)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
*** Starting controller
c1
*** Starting 6 switches
s1 s2 s3 s4 s5 s6 ...
*** Starting CLI:
mininet>
mininet>
mininet>
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h11 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h12 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h13 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h14 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h15 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
h16 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h17 h18 h19 h20 h21 h22 h23 h24 h25
h17 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h18 h19 h20 h21 h22 h23 h24 h25
h18 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h19 h20 h21 h22 h23 h24 h25
h19 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h20 h21 h22 h23 h24 h25
h20 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h21 h22 h23 h24 h25
h21 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h22 h23 h24 h25
h22 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h23 h24 h25
h23 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h24 h25
h24 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h25
h25 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24
*** Results: 0% dropped (600/600 received)
```

Fig.8 Topology generated in Mininet using Python Script

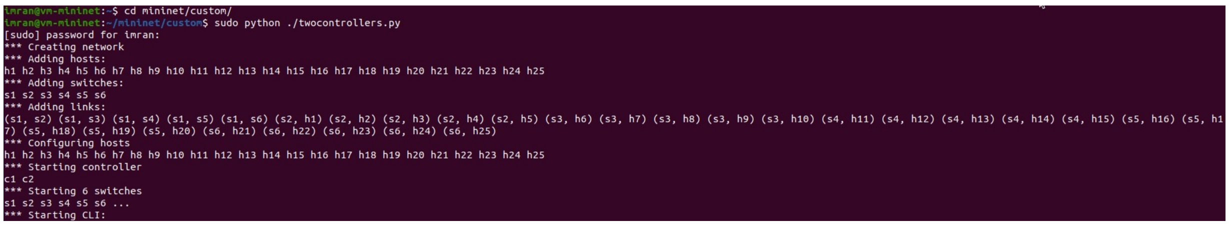


```python
#!/usr/bin/python

"""
Mininet Network to connect n switches to a single controller
"""

from mininet.net import Mininet
from mininet.node import OVSSwitch, Controller, RemoteController
from mininet.topolib import TreeTopo
from mininet.log import setLogLevel
from mininet.cli import CLI

setLogLevel( 'info' )

# A single local controller running on default port 6633

c1 = Controller( 'c1', port=6633 )

cmap = { 's1': c1, 's2': c1, 's3': c1, 's4':c1, 's5':c1, 's6':c1 }

class MultiSwitch( OVSSwitch ):

    def start( self, controllers ):
        return OVSSwitch.start( self, [ cmap[ self.name ] ] )

topo = TreeTopo( depth=2, fanout=5 )
net = Mininet( topo=topo, switch=MultiSwitch, build=False )

net.addController(c1)
net.build()
net.start()
CLI( net )
net.stop()
```

Fig.9 Python Script for Single Controller Network.

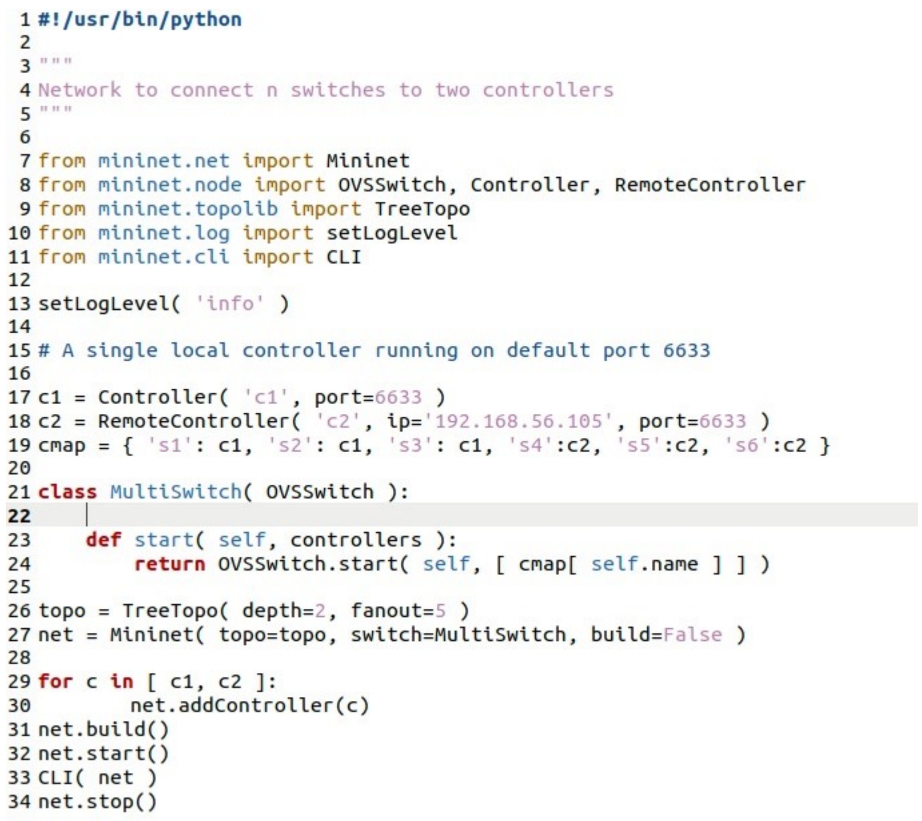
• **Scenario 2: Having two Controllers in a Network**

In this scenario, we have deployed 2 Controllers, one running locally on 127.0.0.1 and another one is the POX Controller running remotely on 192.168.56.105. We have used tree topology with 2 levels of depth and Fanout being 5. This will deploy 2 Controllers, 6 OVSSwitches and 25 hosts. The Switches are mapped to their corresponding Controllers using the cmap function.



```
imran@vm-mininet:~$ cd mininet/custom/
imran@vm-mininet:~/mininet/custom$ sudo python ./twocontrollers.py
[sudo] password for imran:
*** Creating network
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
*** Adding switches:
s1 s2 s3 s4 s5 s6
*** Adding links:
(s1, s2) (s1, s3) (s1, s4) (s1, s5) (s1, s6) (s2, h1) (s2, h2) (s2, h3) (s2, h4) (s2, h5) (s3, h6) (s3, h7) (s3, h8) (s3, h9) (s3, h10) (s4, h11) (s4, h12) (s4, h13) (s4, h14) (s4, h15) (s5, h16) (s5, h1
7) (s5, h18) (s5, h19) (s5, h20) (s6, h21) (s6, h22) (s6, h23) (s6, h24) (s6, h25)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25
*** Starting controller
c1 c2
*** Starting 6 switches
s1 s2 s3 s4 s5 s6 ...
*** Starting CLI:
```

Fig.10 Topology with two controllers generated in Mininet using Python Script.

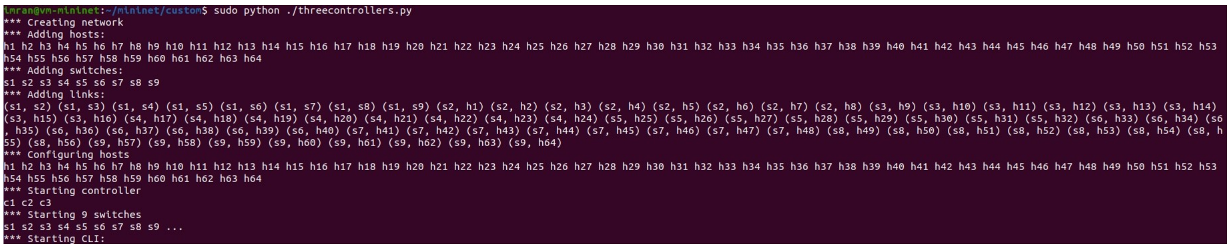


```python
#!/usr/bin/python

"""
Network to connect n switches to two controllers
"""

from mininet.net import Mininet
from mininet.node import OVSSwitch, Controller, RemoteController
from mininet.topolib import TreeTopo
from mininet.log import setLogLevel
from mininet.cli import CLI

setLogLevel( 'info' )

# A single local controller running on default port 6633

c1 = Controller( 'c1', port=6633 )
c2 = RemoteController( 'c2', ip='192.168.56.105', port=6633 )
cmap = { 's1': c1, 's2': c1, 's3': c1, 's4':c2, 's5':c2, 's6':c2 }

class MultiSwitch( OVSSwitch ):

    def start( self, controllers ):
        return OVSSwitch.start( self, [ cmap[ self.name ] ] )

topo = TreeTopo( depth=2, fanout=5 )
net = Mininet( topo=topo, switch=MultiSwitch, build=False )

for c in [ c1, c2 ]:
        net.addController(c)
net.build()
net.start()
CLI( net )
net.stop()
```

Fig.11 Python Script for Two Controllers Network.

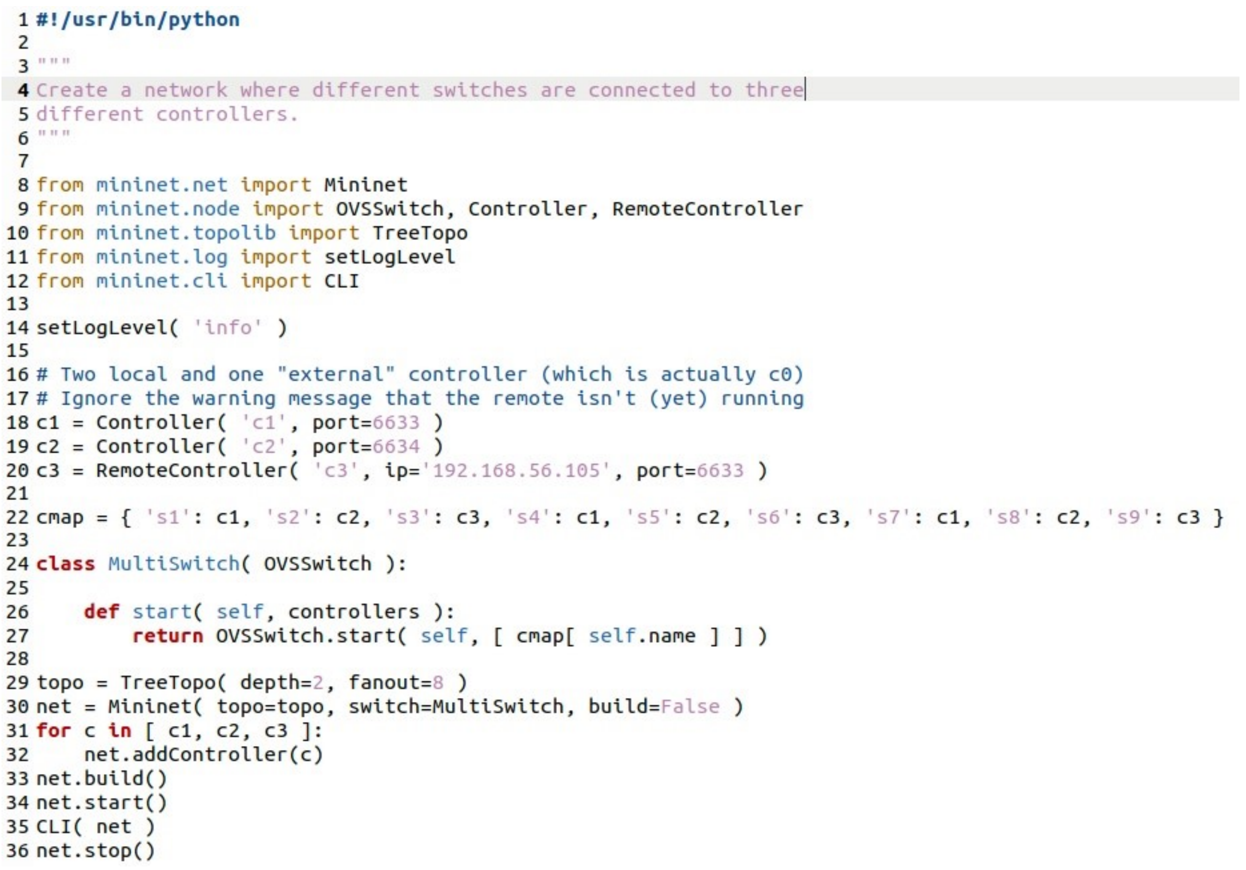
- **Scenario 3: Having three Controllers in a Network**

In this scenario, a network has been deployed with 2 local controllers and a remote POX Controller. We have used tree topology with 2 levels of depth and Fanout of 8. This deploys 3 Controllers, 9 OVSSwitches and 64 hosts.



```
imran@vm-mininet:~/mininet/custom$ sudo python ./threecontrollers.py
*** Creating network
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25 h26 h27 h28 h29 h30 h31 h32 h33 h34 h35 h36 h37 h38 h39 h40 h41 h42 h43 h44 h45 h46 h47 h48 h49 h50 h51 h52 h53
h54 h55 h56 h57 h58 h59 h60 h61 h62 h63 h64
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8 s9
*** Adding links:
(s1, s2) (s1, s3) (s1, s4) (s1, s5) (s1, s6) (s1, s7) (s1, s8) (s1, s9) (s2, h1) (s2, h2) (s2, h3) (s2, h4) (s2, h5) (s2, h6) (s2, h7) (s2, h8) (s3, h9) (s3, h10) (s3, h11) (s3, h12) (s3, h13) (s3, h14)
(s3, h15) (s3, h16) (s4, h17) (s4, h18) (s4, h19) (s4, h20) (s4, h21) (s4, h22) (s4, h23) (s4, h24) (s5, h25) (s5, h26) (s5, h27) (s5, h28) (s5, h29) (s5, h30) (s5, h31) (s5, h32) (s6, h33) (s6, h34) (s6
, h35) (s6, h36) (s6, h37) (s6, h38) (s6, h39) (s6, h40) (s7, h41) (s7, h42) (s7, h43) (s7, h44) (s7, h45) (s7, h46) (s7, h47) (s7, h48) (s8, h49) (s8, h50) (s8, h51) (s8, h52) (s8, h53) (s8, h54) (s8, h
55) (s8, h56) (s9, h57) (s9, h58) (s9, h59) (s9, h60) (s9, h61) (s9, h62) (s9, h63) (s9, h64)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25 h26 h27 h28 h29 h30 h31 h32 h33 h34 h35 h36 h37 h38 h39 h40 h41 h42 h43 h44 h45 h46 h47 h48 h49 h50 h51 h52 h53
h54 h55 h56 h57 h58 h59 h60 h61 h62 h63 h64
*** Starting controller
c1 c2 c3
*** Starting 9 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 ...
*** Starting CLI:
```

Fig.12 Topology with three controllers generated in Mininet using Python Script.



```
#!/usr/bin/python

"""
Create a network where different switches are connected to three
different controllers.
"""

from mininet.net import Mininet
from mininet.node import OVSSwitch, Controller, RemoteController
from mininet.topolib import TreeTopo
from mininet.log import setLogLevel
from mininet.cli import CLI

setLogLevel( 'info' )

# Two local and one "external" controller (which is actually c0)
# Ignore the warning message that the remote isn't (yet) running
c1 = Controller( 'c1', port=6633 )
c2 = Controller( 'c2', port=6634 )
c3 = RemoteController( 'c3', ip='192.168.56.105', port=6633 )

cmap = { 's1': c1, 's2': c2, 's3': c3, 's4': c1, 's5': c2, 's6': c3, 's7': c1, 's8': c2, 's9': c3 }

class MultiSwitch( OVSSwitch ):

    def start( self, controllers ):
        return OVSSwitch.start( self, [ cmap[ self.name ] ] )

topo = TreeTopo( depth=2, fanout=8 )
net = Mininet( topo=topo, switch=MultiSwitch, build=False )
for c in [ c1, c2, c3 ]:
    net.addController(c)
net.build()
net.start()
CLI( net )
net.stop()
```

Fig.13 Python Script for Three Controllers Network.

6 Evaluation

When a large scale SDN is deployed, it is practically next to impossible to leave it on a single Controller. The throughput and latency of the network takes a hit and the whole purpose of the SDN is not served. We have demonstrated this by deploying the network in 3 scenarios using the Mininet Emulator. There has been thorough analysis and of the network performance by measuring the throughput.

As discussed in the topology section, in the first scenario, we have seen that the throughput of the network with a single controller is quite low when compared to that of multi controller networks. As we progress with the multi controller environment, we see that the overall throughput of the network increases significantly. As the proposed algorithm is executed, the number of controllers increase as there is a rise in the fanout number. Hence, when the switch count increases, the controllers are increased and the load is balanced across all the network controller. When this model is deployed on the real time large scale networks, there would be better throughput and stability in the network.

6.1 Performance of a Single Controller Network

The Single Controller Network recorded a maximum throughput of around 22 Gbps. However, when the packets were flooded for a brief period of time, the throughput decreased and kept fluctuating around 21 Gbps mark. The graph clearly shows that the throughput decreased and remained on the lower side.

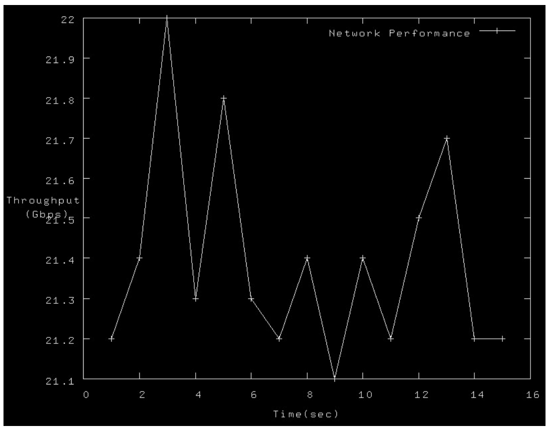


Fig.14 Throughput of a Single Controller Network

6.2 Performance of a Two-Controller Network

With two controllers, we can see that the throughput improved and kept at around 23 Gbps throughout.

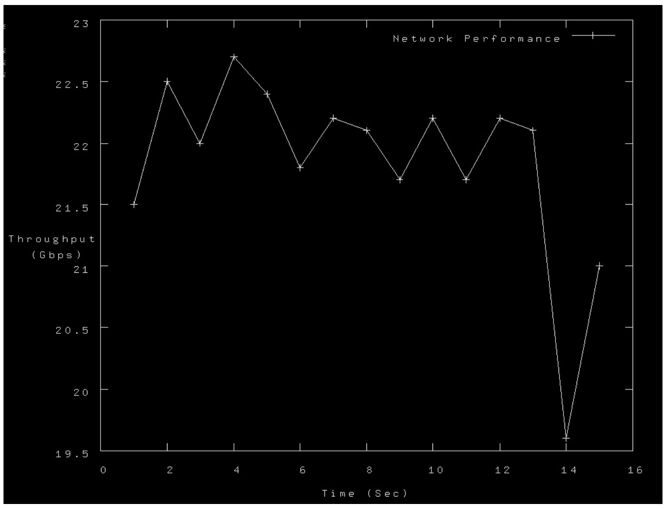


Fig.15 Throughput of a Two Controller Network

6.3 Performance of a Three-Controller Network

With three controllers, the network performance surged significantly. The network throughout kept around 25 Gbps and with minimal fluctuations among the values.

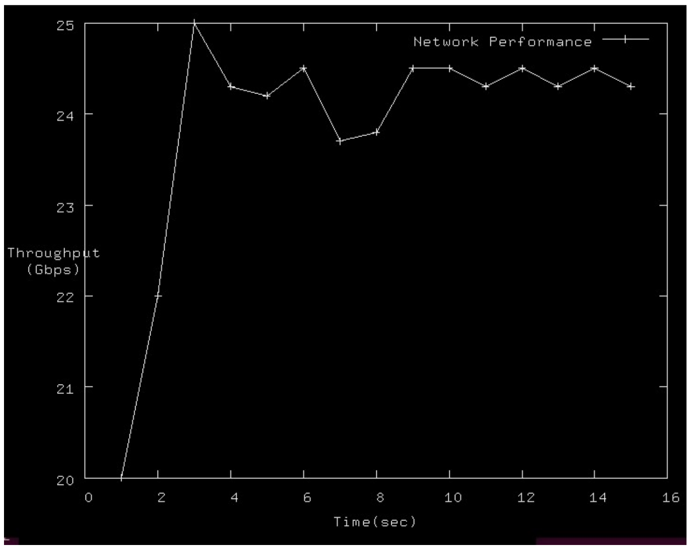


Fig.16 Throughput of a Three Controller Network

7 Conclusion and Future Work

In this paper, we have discussed about Software Defined Networks with single controller and multi controllers. Based on the multi controller scenarios, an algorithm is proposed to add multiple controllers in a network and assign switches to them in a round robin fashion. Based on the upper threshold of 70% of switches and the lower threshold of 30% of switches the number of controllers will increase and decrease respectively. Mininet is used to simulate multiple topology networks and demonstrate the working of algorithm. Based on the experimental results, it is evident that the throughput increases drastically from one controller to two and more controllers.

Another area to focus in the future works could be applying this working model on a real time large Datacentre SDN network wherein the performance can be measured on the real time network traffic. Furthermore, this model can be enhanced further to include inter-controller latency and overall network latency.

Acknowledgement

Finally, I must express my heartfelt gratitude to my supervisor **Mr.Vikas Sahni**, Professor in Computing at National College of Ireland who has been the driving force throughout the course of this research. Without his encouragement, guidelines and continuous support, this project would not have been possible. His valuable inputs and suggestions have always been extremely helpful to me in improvising the idea in completion of the project.

I would also like to take this opportunity to thank **Dr.Horacio Gonzalez-Velez** who helped me giving continuous feedbacks in the initial stages of research . Lastly, I would also like to thank my parents and friends for their continuous support and encouragement.

References

Bari, M. F., Roy, A. R., Chowdhury, S. R., Zhang, Q., Zhani, M. F., Ahmed, R. and Boutaba, R. (2013). Dynamic controller provisioning in software defined networks, *Network and Service Management (CNSM), 2013 9th International Conference on*, IEEE, Zurich, Switzerland, pp. 18–25. CORE Ranking: B.

Barrett, R., Facey, A., Nxumalo, W., Rogers, J., Vatcher, P. and St-Hilaire, M. (2017). Dynamic traffic diversion in sdn: testbed vs mininet, *Computing, Networking and Communications (ICNC), 2017 International Conference on*, IEEE, Silicon Valley, USA, pp. 167–171.

Botelho, F., Bessani, A., Ramos, F. and Ferreira, P. (2014). Smartlight: A practical fault-tolerant sdn controller, *arXiv preprint arXiv:1407.6062* .

Dixit, A., Hao, F., Mukherjee, S., Lakshman, T. and Kompella, R. R. (2014). Elasticon; an elastic distributed sdn controller, *Architectures for Networking and Communications Systems (ANCS), 2014 ACM/IEEE Symposium on*, IEEE, Los Angeles, CA, USA, pp. 17–27.

Hassas Yeganeh, S. and Ganjali, Y. (2012). Kandoo: a framework for efficient and scalable offloading of control applications, *Proceedings of the first workshop on Hot topics in software defined networks*, ACM, pp. 19–24.

Kim, T., Choi, S.-G., Myung, J. and Lim, C.-G. (2017). Load balancing on distributed datastore in opendaylight sdn controller cluster, *Network Softwarization (NetSoft), 2017 IEEE Conference on*, IEEE, Bologna, Italy, pp. 1–3.

Krishnamurthy, A., Chandrabose, S. P. and Gember-Jacobson, A. (2014). Pratyaastha: an efficient elastic distributed sdn control plane, *Proceedings of the third workshop on Hot topics in software defined networking*, ACM, Chicago, USA, pp. 133–138.

Lange, S., Gebert, S., Zinner, T., Tran-Gia, P., Hock, D., Jarschel, M. and Hoffmann, M. (2015). Heuristic approaches to the controller placement problem in large scale sdn networks, *IEEE Transactions on Network and Service Management* **12**(1): 4–17.

Lantz, B., Heller, B. and McKeown, N. (2010). A network in a laptop: rapid prototyping for software-defined networks, *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ACM, Monterey, CA, USA, p. 19.

Lee, B., Park, S. H., Shin, J. and Yang, S. (2014). Iris: the openflow-based recursive sdn controller, *Advanced Communication Technology (ICACT), 2014 16th International Conference on*, IEEE, Pyeongchang, South Korea, pp. 1227–1231.

Mallesh, S. (2017)). Automatic detection of elephant flows through openflow-based openvswitch.

Moazzeni, S., Khayyambashi, M. R., Movahhedinia, N. and Callegati, F. (2018). On reliability improvement of software-defined networks, *Computer Networks* **133**: 195–211. CORE Ranking: A.

Ortiz, J., Londoño, J. and Novillo, F. (2016). Evaluation of performance and scalability of mininet in scenarios with large data centers, *Ecuador Technical Chapters Meeting (ETCM), IEEE*, IEEE, Guayaquil, Ecuador, pp. 1–6.

Ros, F. J. and Ruiz, P. M. (2016). On reliable controller placements in software-defined networks, *Computer Communications* **77**: 41–51. CORE Ranking: C.

Shamugam, V., Murray, I., Leong, J. and Sidhu, A. S. (2016). Software defined networking challenges and future direction: A case study of implementing sdn features on openstack private cloud, *IOP Conference Series: Materials Science and Engineering*, Vol. 121, IOP Publishing, p. 012003.

Simulator, M. (n.d.).
URL: *https://mininet.org*

Tootoonchian, A. and Ganjali, Y. (2010). Hyperflow: A distributed control plane for openflow, *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pp. 3–3.

ul Huque, M. T. I., Si, W., Jourjon, G. and Gramoli, V. (2017). Large-scale dynamic controller placement, *IEEE Transactions on Network and Service Management* **14**(1): 63–76.

Zhang, T., Giaccone, P., Bianco, A. and De Domenico, S. (2017). The role of the inter-controller consensus in the placement of distributed sdn controllers, *Computer Communications* **113**: 1–13. CORE Ranking: B.

Zhang, Y., Cui, L., Wang, W. and Zhang, Y. (2017). A survey on software defined networking with multiple controllers, *Journal of Network and Computer Applications* . CORE Ranking: A.