Enhancement of Network Throughput in SDN Using Shortest Path Routing Algorithms

MSc Research Project
MSc in Cloud Computing

Sayali Kapse
Student ID: x01731279

School of Computing
National College of Ireland

Supervisor: Muhammad Iqbal

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Sayali Kapse
Student ID:	x01731279
Programme:	MSc in Cloud Computing
Year:	2018-2019
Module:	MSc Research Project
Supervisor:	Muhammad Iqbal
Submission Due Date:	03/02/2020
Project Title:	Enhancement of Network Throughput in SDN Using Shortest Path Routing Algorithms
Word Count:	6718
Page Count:	26

Signature:	
Date:	3rd February 2020

Enhancement of Network Throughput in SDN Using Shortest Path Routing Algorithms

Sayali Kapse
x01731279

Abstract

Software-Defined Network has bought a massive change in traditional networks, which has enabled dynamic adaption and configuration. In data centers, SDN helps to create a virtual network where the controller forms centralized control for the entire network, but the network topology used in data centers needs to be more scalable and flexible to avoid network congestion. Thus, in this project, the fat-tree network topology is created to find the shortest paths between the source node and the destination node. Openflow protocol is used for the transmission of messages. The experiments are performed for a specific time interval to evaluate the TCP and UDP network bandwidth. Also, a comparison is made between the two parameters i.e., hop count and delay based on the shortest path. To simulate the entire network Mininet emulator tool is used to design the network topology, and the RYU controller is used for packet routing. This research shows that the network throughput can be improved when the packets transmitted from source to destination have low hop count and delay(weight).

1 Introduction

Cloud computing today has become an essential and efficient service that provides a platform to the user for accessing and accumulating required data. It also includes cloud storage where users can store their required data, which can be used in the future for retrieval as well as for analysis. With a close connection to the Internet of Things, the data routing and storage are controlled by the software-defined system and the cloud platform (Govindarajan et al.; 2013). On the other hand, for user feasibility, the cloud platform is also utilized in the form of remote access systems like Software as a Service, Platform as a Service or Infrastructure as a Service where the user can avail of their required software and even hardware support. All these services all supported by the Software-Defined Network, which creates a Virtual Network. This helps for the faster exchange of data in a network with a high analytical performance that is desired for the data. So, the Internet of Things, Cloud Service, and Software Defined Network are, in turn, correlated for the utilization of remote data and to access it for the analytical purpose (Fernández et al.; 2018).

The purpose of establishing a Software Defined Network is to take control of the network elements like router and switches, which are responsible for data routing. Data is transferred from one node to another in the network to reach from the source to the destination following a particular path. Sometimes if the path is long or there is a physical presence of a passive device in the route, there are chances of some data loss (Akin and Korkmaz; 2019). The traditional network of cloud computing contains highly dense servers and switches that are not controlled centrally. An overview of such a traditional network and SDN is shown below Figure. 1 where data is coming from cloud computers and transferred to the user.

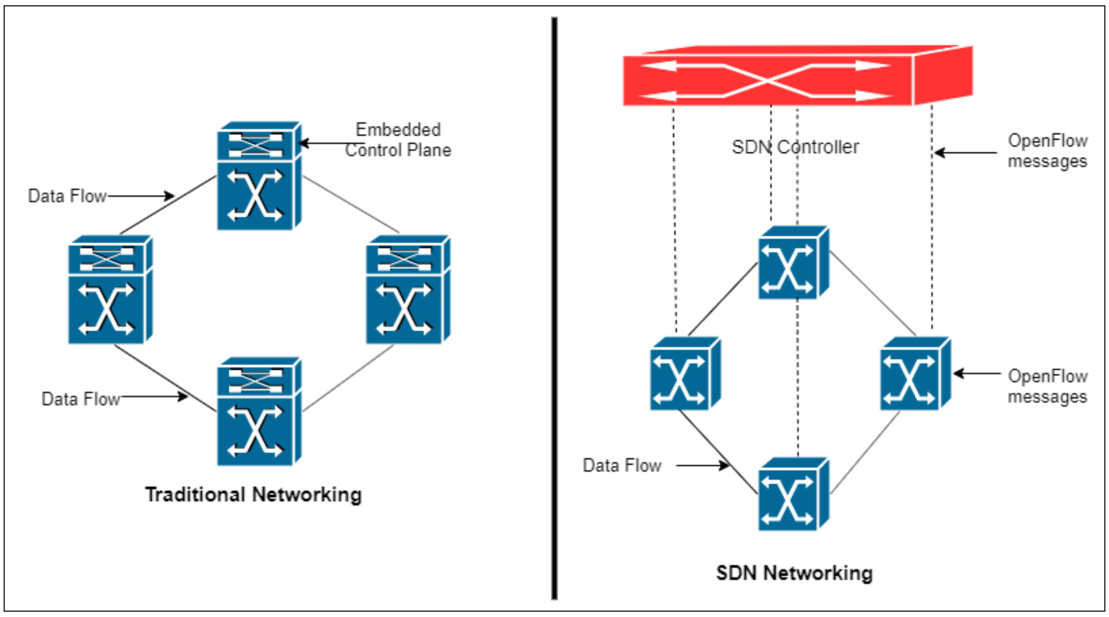


Figure 1: Comparison of Traditional Network and SDN.

In the traditional cloud network, there is no specific control over the network device to generate a decision for the shortest route from the source to destination. Thus, the data is traversed through the network, and with the availability of the free path, it will be navigated to reach the destination. It means the data routing is executed without any centrally controlled device. The main drawback of the traditional network is the low throughput so. There was a need to design a network that can be controlled centrally for the determination of the path through which the data will be navigated easily and without any loss. Thus, this is when Software Defined Network has emerged. Software-Defined Network is the latest technology in the field of cloud computing, which controls the Routers and Switches using the SDN Protocol. Software-Defined Network or SDN communicate with the routers and switches using its protocol, known as OpenFlow (Kreutz et al.; 2015). In OpenFlow control, the routing mechanism contains multiple routing tables which are designed with multiple packet forwarding rule. So, at the time of packet transaction, the decision can be taken by the SDN Controller about the rule to be applied. With the application of such Rules in OpenFlow, different actions like forwarding of packet, modification of the content of the packet as well as the routing path and dropping of the packets are performed (Lee and Sheu; 2016a).

Also, (de Oliveira et al.; 2014) addresses that a good simulation environment is also necessary for simulating large scale networks on a virtual machine. Thus, a Mininet emulator helps to test, customize SDN networks, and also support default network topologies. In the Software-Defined Network, few parameters determine the performance of the operation made by SDN controlled network. They are:

- **Scalability:** In a traditional network, if additional devices are added, the complexity of the network increases. SDN network scalability feature is dependent on controller and switches in the network. Scalability in SDN can be achieved by improving the network throughput and creating a distributed network for data transmission (Govindarajan et al.; 2013).
- **Bandwidth:** Network bandwidth is another feature of the network, which allows the maximum amount of data to be transferred from source to destination or in other words in between two or multiple nodes. The bandwidth can be controlled and varied as per the required data density and devices involved (Akin and Korkmaz; 2019). For example, if the device is the Gigabit Ethernet, the speed can be increased up to 1000 Mbps to 1250 Mbps. With this feature, the desired amount of data can be transferred within the nodes.
- **Security:** Security defines the policies that the network adopts for securing the data within the network. In SDN architecture, the centralized control mechanism over the virtual network prevents the files and contents of the network from any external attack or unauthorized access, modification, and misuse of the data (Son and Buyya; 2018). So, the data become safe during the execution by SDN.
- **Flexibility:** The network can become efficient if it is flexible in adopting and managing the traffic distribution and network latency. In SDN architecture, the network latency and the traffic distribution can be controlled, which are the most sensitive parameters for the data transaction among the nodes or switches and routers (Paluck and Jain; 2018).

1.1 Research Problem

In this research, due to the several limitations of the traditional network, the SDN will be used for the design of the network to manage the resource of it and to take control for routing purposes. So, the ultimate objective is to find the shortest route from the source node to the destination node so that the performance of the SDN can be improved further. The proposed research questions are as follows:-

- *Can the proposed algorithm find the shortest path in SDN networks to minimize the network traffic while sending data packets from source to destination node?*
- *Can the proposed algorithm improve the network performance which depends on the hop counts between the switches to transmit the data between two hosts?*

1.2 Research Objective

The following research objectives are addressed to reduce the network traffic for packet transmission from the source node to the destination node. Further, to evaluate performance parameters and ensure Quality of Service (QoS) in SDN, the algorithms are implemented in the router control plane.

- **Objective 1:** To determine the TCP and UDP traffic between to nodes.
- **Objective 2:** To compare the hop count and delay for the shortest path found in the network.

2 Related Work

2.1 SDN Architecture Overview

In the traditional network structure, there are three planes, namely, Data Plane, Control Plane, and Management Plane. These are the layers of the network device to perform different tasks. As those are connected in a network device, the routing of data is less efficient because it lacks the decision to detect the path of the data. So, there should be the alternation of the network paradigm, which is capable of making a decision over the path of the data traversed in the underlying network by separating the control plane and the data forwarding plane (Cox et al.; 2017). This makes Software Defined Network applicable in the field of the virtual or cloud network structure as it is able to make a decision over the network routing centrally with the help of the programming at the control plane and the data forwarding plane is simplified with its own routing table to transact off the data suitably (Li et al.; 2018). In SDN architecture, there are three layers involved to perform different objectives for the data routing with the application of the OpenFlow Protocol. The brief description of the planes of the SDN architecture are discussed below:

Data Plane: This layer includes the hardware and software components of the router and switches. The objective of this plane is related to the packet forwarding in between interfaces or nodes according to the instructions from the control plane. This plane also involves the MAC address, which shows the address of the packet from where it is transferred and the location where the packet will be transferred. The plane is also known as the forwarding plane as it forwards the packet from one node to another node. The packet is transferred from one hop to another through the router using the routing logic and tables (Challa et al.; 2017).

Control Plane: The control plane is the integrated part of the network device which carries the packet and traffic, and this plane is liable for routing of data and packet. This plane originates from the address of the interface where the packet will be routed. It uses different routing protocols like BGP, EIGRP, IS-IS, etc. for data routing. This plane is responsible for deciding the traffic to be sent to the next-hop (Jarraya et al.; 2014). So, the network algorithm should be executed in this plane so that the intelligent packet routing can be executed. While executing the algorithm for finding the shortest route for packet transaction, the control plane can change the routing table as per the requirement. In the controller layer of the SDN structure, the employed controller is operated in three separate modes, which prepare a new entry for packet flow and the packet forwarding among or between the switches. The modes are briefed below:

- The first mode is referred to as the Reactive mode, where the packet flow seeks the rule from the flow table when it is directed to the switch. If there is no flow observed, that means there is no rule specified for that flow. The packet is redirected to the controller using C-DPI rule. The new flow entry is created by the controller for the recognition of such new routing.

- The second mode is referred to as the Proactive mode. In this mode, all the possible combination of the packet routing is pre-set in the flow table, and thus, the switches know the direction of the packet routing. In this case, for the creation of the new entries of the flow table, the controller is not involved.

- The third mode contains the Hybrid model. This mode is the combination of the previous two modes and works alternatively as per the flow entries specification (Karakus and Durresi; 2017).

Management Plane: This plane is responsible for the device configuration by using different protocols like SNMP or SSH. The packet is routed through the configured device that is controlled by the centralized OpenFlow protocol in SDN.

2.2 OpenFlow Protocol in SDN

OpenFlow is the specific protocol that is exclusively used and standardize in Software-Defined Network (Jarraya et al.; 2014). OpenFlow is specifically defined for communication in Software Defined Network that makes the SDN Controller interact directly with the data forwarding plane and to work with the network devices like routers and switches in both the Virtual and Physical network configuration. SDN Controller works as the central brain of the network, which drives the network with its defined protocol. In a Software Defined Network, if any device wants to connect with the network component, the communication should be made through the OpenFlow network (Paluck and Jain; 2018). In Software Defined Network, OpenFlow is used for controlling the hardware by providing instruction of routing. In this case, the Switches and the Routers are instructed to select the path and the flow of the packet. So, the entire network will be entirely managed by the administrator (Jesús Antonio Puente et al.; 2018). Apart from the OpenFlow Protocol, the Software-Defined Network contains two more entities, namely, OpenFlow Switch and the external controller. These latter two are controlled by the OpenFlow Protocol.

The routing of the packet is controlled and decided by configuring the routing table, and that routing table is basically the result of the soft definition of the OpenFlow Protocol (Challa et al.; 2017). The switches of the OpenFlow protocol and is categorized on the basis of the requirement made in two layers of the network, namely, Layer2 Layer3, where the traffic is redirected by the Legacy Protocol. The legacy protocol is used for the packet redirection in Ethernet, Router, and Switches and using LLP or Lower Layer Protocol and RPST or Rapid Spanning Tree Protocol. When one or more packets are coming towards a Switch and need to be redirected to the destination address, these two protocols help to redirect this, and the decision is made by the OpenFlow Protocol. The address is checked from the packet header, and the data can be retrieved from the packet, which is encapsulated. Depending upon the address, the packet is redirected (Karakus and Durresi; 2017).

2.3 Scalability Issues In SDN Control Plane

(Karakus and Durresi; 2017) and (Govindarajan et al.; 2013) did a survey about the network scalability issues for the SDN architecture. The issue of the scalability occurs due to several reasons like separation of Data Plane from Control Plane, a number of requests and events managed by the controller, and the associated communication delay between controller and switch. The reasons are briefed below:

Separation of control plane and data plane: The control plane in the SDN architecture makes the decision for the packet routing around the network, and finally, the packets are redirected to the specific destination by observing the flow table through the data plane. Thus, the data plane is not eligible to make the decision for the flow control, and all the flows are controlled in the SDN architecture remotely. So, the separation of the data plane and the control plane create the network scalability issues in SDN.

Amount of Request and Event management: The desired network should be dynamic in terms of the number of devices being operated. If more devices are added, the number of requests and event handling will be higher. So, with the increase in the number of the hosts and switches, the load of the controller will also be increased. Thus, this will affect the computational resource of the controller that can lead to undesired load balancing, or there are chances that all the requests or events cannot be processed by the controller.

Controller-switch communication delay: This issue is created by the distance factor between the devices in the network and Controller. The high will be the distance, the latency time will be higher, and this will affect the Round-Trip-Time of the packet forwarding in the network. This leads to the delay in communication between the two entities.

2.4 SDN Packet Routing Frameworks

In a network, the routing algorithm determines all possible paths and finally select the best one. These routing algorithms can be characterized into different parameters like time is taken to reach the destination. A routing algorithm is said to be more efficient when the traffic across multiple links is reduced and, the links used to divert the traffic improve the network throughput (Lee and Sheu; 2016b).

Frameworks in the Software-Defined Network are designed to eliminate the drawbacks of the scalability issue in the network and to minimize the communication delay between the Controller and Switch and the network device. Those drawbacks eventually create drawbacks like Path Congestion and packet loss. MPLS or Multi-Protocol Level Switching is generally used as the remedy for the minimization of the communication time (Lee et al.; 2015). Now to avoid traffic congestion, the best way is to find the shortest path between two nodes in the network tree. In this technique, primarily, the network will find and create the network topology through which the packet transaction will occur, and after that, in the second step, it finds all possible paths between these two nodes and compute the optimal path through which the packet transaction will happen. Lastly, after the creation of the topology and path detection, the set of flow control rules are prepared so that the switch will follow the shortest path and packet will travel through the less congested real-time path (Akin and Korkmaz; 2019).

(Jiang et al.; 2014) has introduced the technique to find the shortest path in the Software-Defined Network with the modification of the existing Dijkstra Algorithm. In this paper, they have considered the Edge weight as the general Dijkstra Algorithm does, and additionally, they have considered the Node weight also in the graph underlying the Software-Defined Network. As the Dijkstra works for the unweighted graph in the network, with the consideration of the node weight and with the implication of the Abilene Network topology for the end-to-end latency, the analysis was done on the directed graph. They have shown their simulation using Mininet Tool, and it was found that this extended Dijkstra Algorithm is outperforming other algorithms.

(Sun et al.; 2016) have shown the path planning with the application of the extended Dijkstra Algorithm for the weighted and directed graph where the strategies are followed that were implicated in (Jiang et al.; 2014). Using the algorithm and with the application of the heuristic search, they have established the path planning for the weighted and directed graph by recovering all the possible drawbacks that can be found in the existing algorithms like Dijkstra's. Thus, the shortest path is the critical issue for the Software-Defined Network, which acts as a crucial parameter to define the efficiency of the network topology.

The currently used shortest path algorithm performance is hampered as the network size is increased. As these algorithms run shortest path queries for each flow, the performance is decreased. The shortest path is over-utilized, choking the bandwidth. The rest of the links are idle. In some cases, the selection of the shortest path does not optimize network performance. The SDN controllers check for the shortest path even in large-scale networks. If the network size is large, the central query on each flow is time-consuming. Hence, Graph compression is introduced, which improves the path calculation. The large-scaled network is scaled down, maintaining the critical parameters. The best path is calculated on this scaled-down graph, and later it is implemented in a large scale network. It calculates the bandwidth of all links and selects the most optimum path (Li et al.; 2018) and (Xu et al.; 2016). The scaled-down version of the network reduces redundant nodes and edges, which helps in the faster calculation.

3 Methodology

In this research, the primary objective is to increase the network throughput by designing the shortest path algorithm, where there is the successful routing of data packets from the source node to the destination node without any network congestion. If the number of nodes is increased, the load of the network will be increased, and thus, the static routers and the tables will not be able to control all the newly added nodes or devices. This creates link failures and network congestion in SDN (Lin et al.; 2016). Thus with the implication of SDN and the OpenFlow protocol, the network can be managed more efficiently. The controller in SDN is the central brain of the network where all the network information and path computation is done. Figure. 2 below explains how the incoming packets at the switch are forwarded from source to destination via the SDN controller. Thus, the section further explains the methods taken into consideration for implementing the shortest path algorithm.

3.1 Building of Platform

In this project, the algorithm is implemented in the virtual environment that is created by the Oracle Virtual Box. In the Virtual Box, Ubuntu 16.04 is installed for creating the operating environment for the simulation. The simulation of the network is done

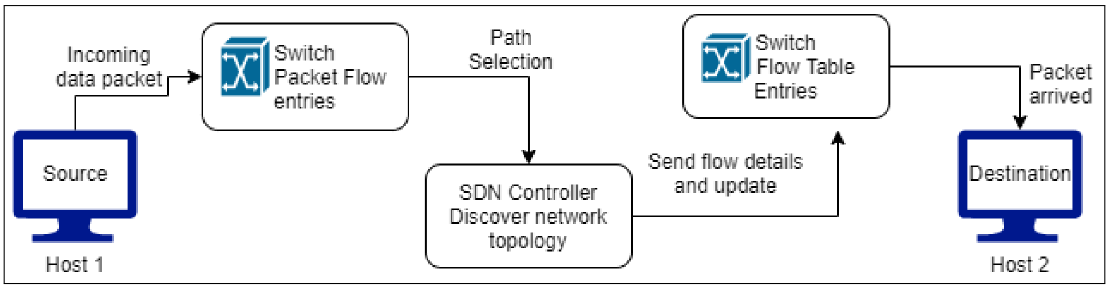

Incoming data packet
Switch Packet Flow entries
Path Selection
Switch Flow Table Entries
Packet arrived
Source
Host 1
SDN Controller Discover network topology
Send flow details and update
Destination
Host 2

Figure 2: Packet Forwarding from Source to Destination

using a Mininet simulator, which is capable of creating the network environment and the relevant simulation within the scope of the virtual environment. The framework is designed where network topology is created to find the shortest path between the source node and the destination node. The framework is segregated into different layers of the network, and in each layer, there are some network components present which define the data routing partially. The bottom layer contains the data plane layer in which the mininet simulation environment is installed. So, all the network creation will be done here. The upper section or layer contains the Network Control Layer, where the Ryu Controller is installed where the routing control is done in this layer. So, below are the description of the required simulation components and the environments (Zhang et al.; 2017).

3.1.1 Mininet

In a virtual environment to simulate a large network, Mininet is the open-source network simulator for Software Defined Network. The primary reason to use the Mininet is that it supports OpenFlow Protocol, which is essential for the network configuration and computation for Software Defined Network. It also provides an inexpensive platform for developing, testing, and creating custom topologies in the network. The remarkable features of Mininet[1] are:

- Mininet is not sensitive to a specific programming language. So, any programming language can be used for the creation of the network topology within the scope of the environment.
- Mininet is capable of virtualizing the network topology in the virtual system of the host, and when the topology executes, the relevant source code will not be modified. So, the network topology can be created easily within the environment.
- The network that is created within the mininet simulator is done in real-time, and so the real-time network topology can be created and so the simulation can be done in real-time.
- Mininet is feasible to add huge numbers of hosts and nodes into the network, and thus, the dense network can be created in real-time.
- The network that is designed using Mininet can act as the real network. It means the behavior of the designed network supports the features of the real network devices.

[1] `http://mininet.org/overview/`

- Mininet is Open Source, and the use of the Mininet in the virtual Box does not draw any credential and, thus, easy to use for this design.

3.1.2 RYU Controller

Ryu controller is basically the SDN Controller, which works as the main controlling device in the SDN. It makes control over the switches and the devices. Additionally, it controls the flow table so that the packet can be routed throughout the network using the Switch Control. The approach of using Ryu Controller in this project is that Ryu Controller can be easily programmed with Python language and is supported by the OpenFlow protocol, which is essential for the design and packet routing operations (Asadollahi et al.; 2018). Figure. 3 below represents how the RYU controller acts as a mediator between the application plane and data plane in SDN.

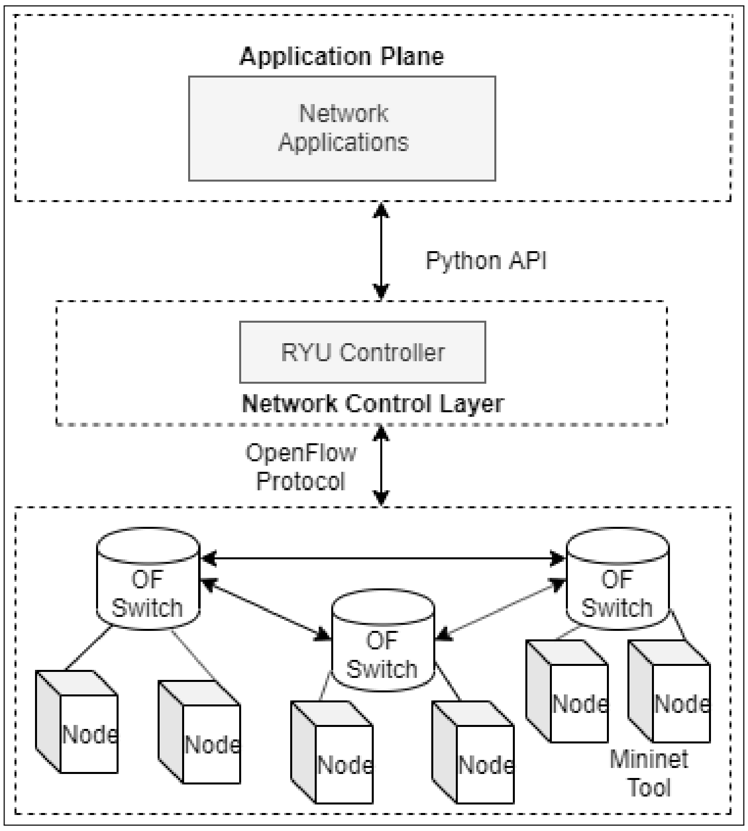


Figure 3: RYU SDN Framework.

3.1.3 Iperf and Gnuplot Tool

Iperf is a network performance tool that is used to measure the bandwidth and datagram loss in a network. This project measures the Transport Control Protocol (TCP) and User Datagram Protocol (UDP) network throughput and data streams. The iperf tool helps to measure the network performance by creating a client and server functionality for both source and destination node. The *gnuplot* tool is used to plot the graph for TCP and UDP traffic.

4 Design

This section explains about the algorithmic flow and the network topology design used in the proposed project.

4.1 Algorithm Description

The aim of the routing algorithm is to identify all available network paths in the topology and discover the best and the shortest path with the least cost in order to route the large flow. All the flow will be pushed to the switches and forwarded accordingly. The algorithm is calculating the transmitted and received bytes on the switch ports and further calculating the computational cost of all the paths to get the minimum cost to reach the destination.

The *ArpHandler* class in the controller is used to initialize the information of network topology. The *create_interior_links* function gets all the links of the source and the destination ports. The *packet_in_handler* function checks if the packet contains all the packet header information. If it is present, the algorithm is executed, and if not, the packet is dropped. When the MAC address of source node and destination node match all the packet header, information is passed to *install_path* function. Finally, the packet is sent to the destination node, and the datapath is printed. The Process flow of the algorithm can be seen in the Figure. 4 below:

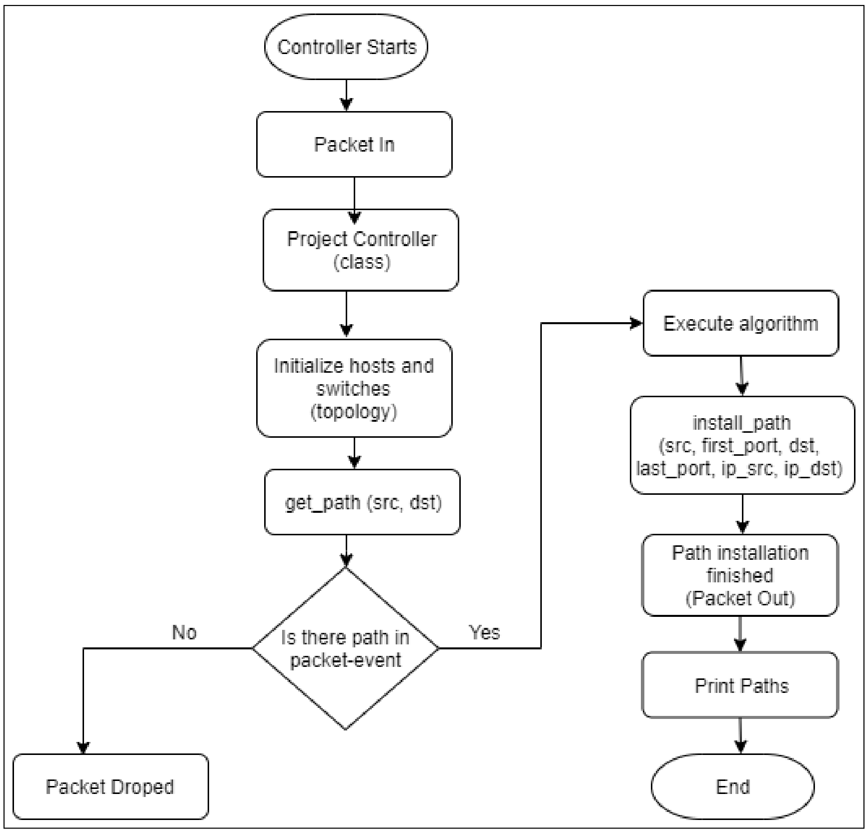

Controller Starts
Packet In
Project Controller (class)
Initialize hosts and switches (topology)
get_path (src, dst)
Is there path in packet-event
No
Yes
Packet Droped
Execute algorithm
install_path (src, first_port, dst, last_port, ip_src, ip_dst)
Path installation finished (Packet Out)
Print Paths
End

Figure 4: Algorithm Flowchart.

4.2 Network Topology Specification

Mininet can be used to create the network topology using the node, edges, and the host that are discussed in the previous section. While the creation of the network topology, the Ryu controller will be acting as the interface controller, and thus, the packet routing can be taken place in that network design. In this project, Fat-tree network topology is designed with the following specifications:

- 8 Hosts
- 20 Switches
- RYU Controller

As network throughput and fault tolerance are the main objectives of data networks; thus, Fat-tree topology is used in SDN Open flow protocol (Raghavendra et al.; 2012). In this topology, all the switches are interconnected to each other, forming a three-layer architecture of switches: core switches, aggregation switches, and edge switches (Wu et al.; 2016). The graphical representation of Fat-tree topology used in this project is shown in Figure. 5 below:

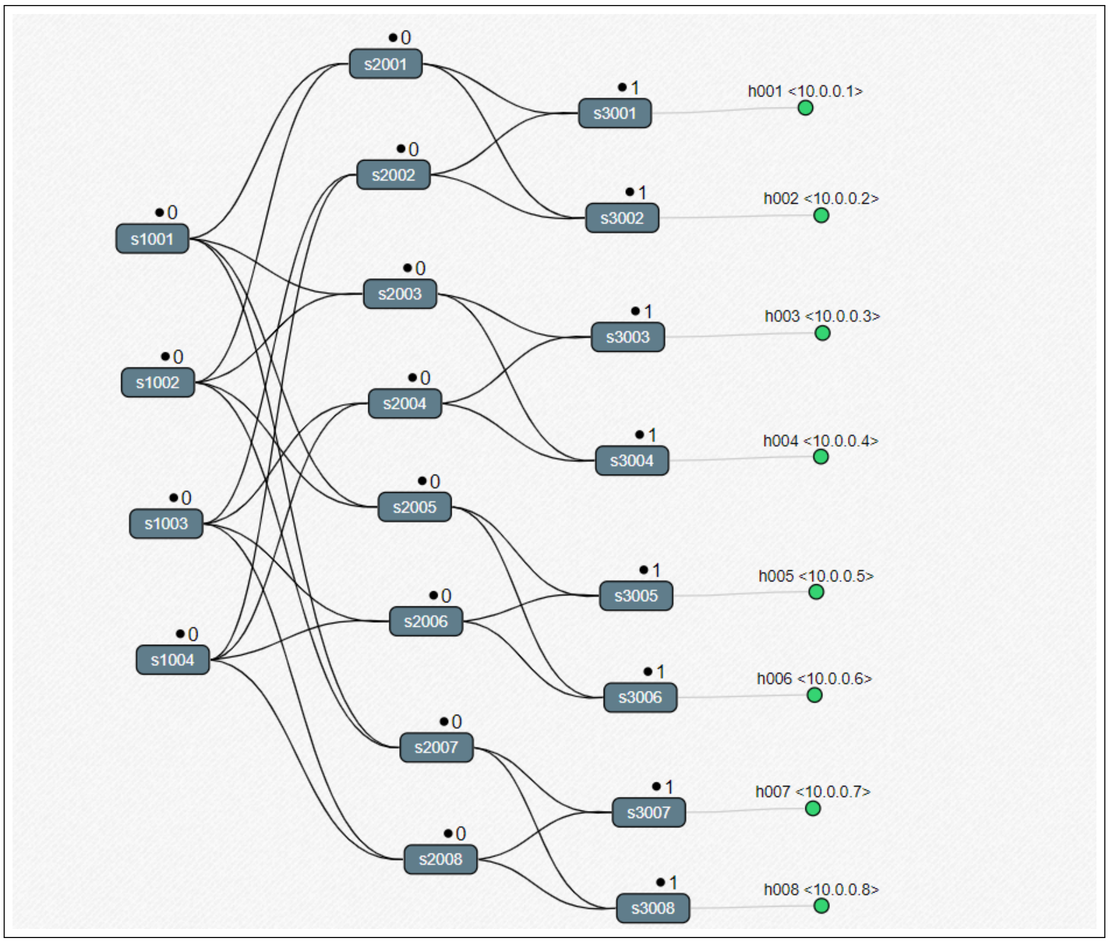


Figure 5: Fat-tree Network Topology.

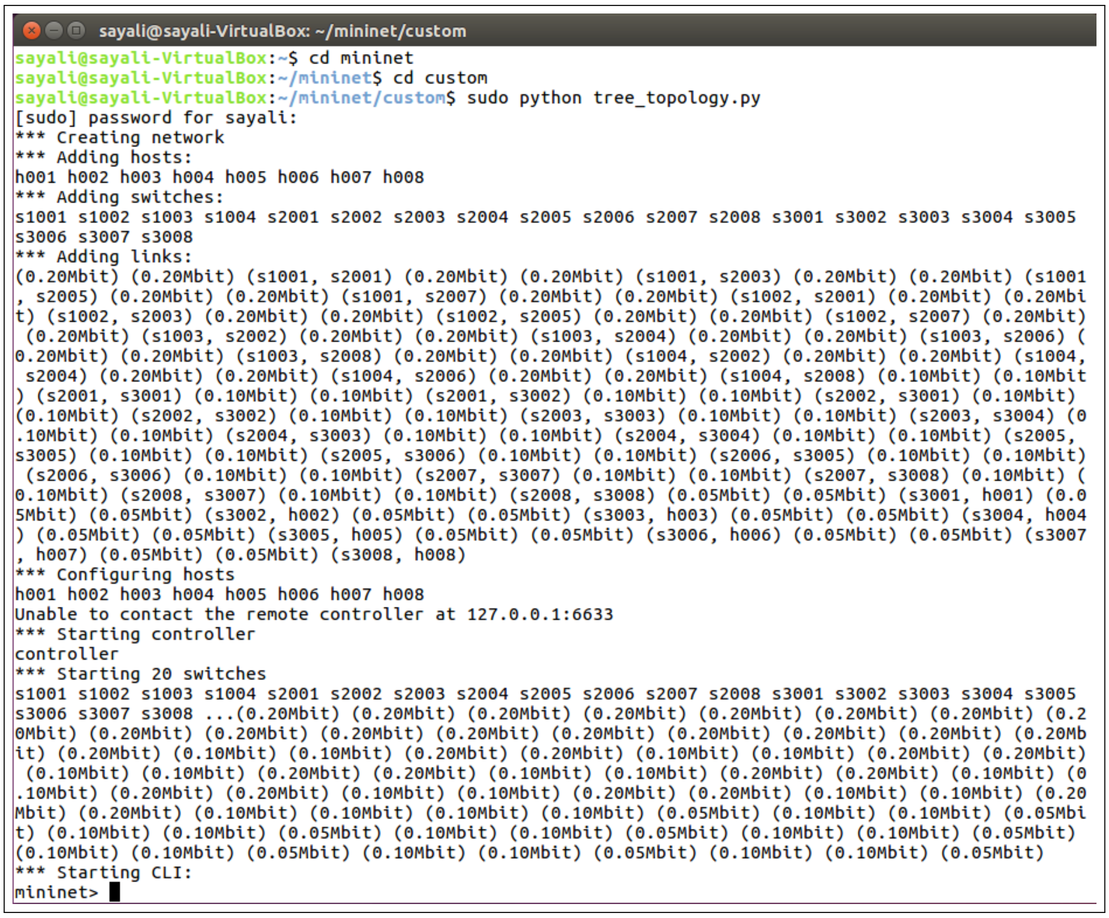
The ryu_manager helps to load the shortest path algorithm files in the RYU controller. It observes links that are formed between hosts and switches and get the shortest path for hop count and delay.

5 Implementation

This section explains the steps taken to implement the proposed shortest path algorithm. To achieve efficient utilization of network resources, the Mininet tool is used for network topology generation that interacts with the RYU controller for sending messages to the destination node. The hosts and switches are controlled by the Controller with the help of flow tables (Ortiz et al.; 2016). In order to choose a suitable path, the algorithm suggested, and network traffic is monitored by the Controller.

5.1 Topology Generation

The controller in SDN stores all the information of network topology. And the proposed algorithm is used to find the shortest path for the available network topology. Figure. 6 below represents that the network is established between the host and switches. To check the connectivity, the *pingall* command in the mininet displays the connectivity between every host in the system and tests whether every host is active and reachable to each other. If the hosts are active, it shows 0% dropped, and all the packets are received. Sometimes *pingall* command may take about 30 minutes or more to complete the connection between the hosts. The links between the hosts and switches are set to 0.20 Mbit, 0.10 Mbit, and 0.05 Mbit bandwidths for the transmission of data packets.

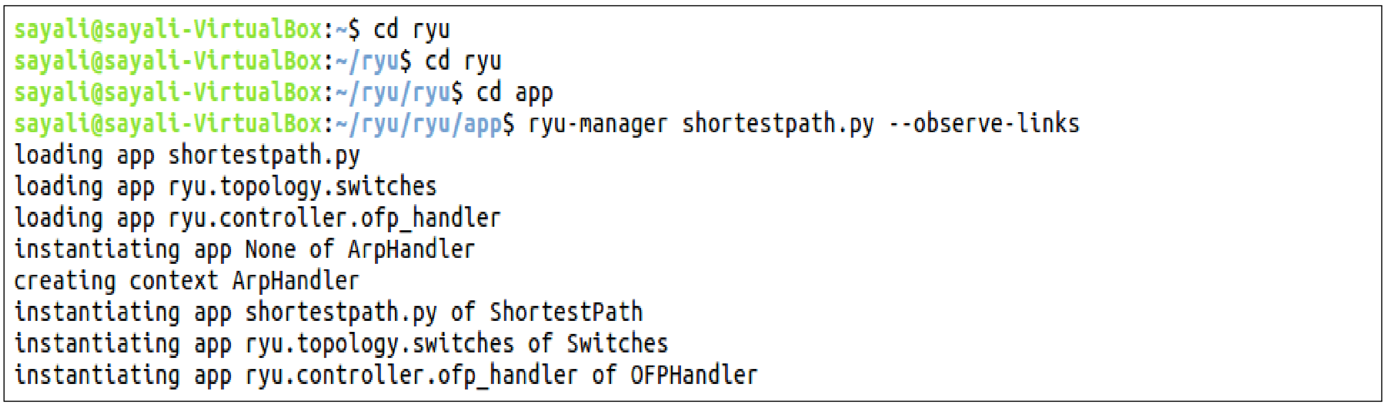


```
sayali@sayali-VirtualBox:~$ cd mininet
sayali@sayali-VirtualBox:~/mininet$ cd custom
sayali@sayali-VirtualBox:~/mininet/custom$ sudo python tree_topology.py
[sudo] password for sayali:
*** Creating network
*** Adding hosts:
h001 h002 h003 h004 h005 h006 h007 h008
*** Adding switches:
s1001 s1002 s1003 s1004 s2001 s2002 s2003 s2004 s2005 s2006 s2007 s2008 s3001 s3002 s3003 s3004 s3005
s3006 s3007 s3008
*** Adding links:
(0.20Mbit) (0.20Mbit) (s1001, s2001) (0.20Mbit) (0.20Mbit) (s1001, s2003) (0.20Mbit) (0.20Mbit) (s1001
, s2005) (0.20Mbit) (0.20Mbit) (s1001, s2007) (0.20Mbit) (0.20Mbit) (s1002, s2001) (0.20Mbit) (0.20Mbi
t) (s1002, s2003) (0.20Mbit) (0.20Mbit) (s1002, s2005) (0.20Mbit) (0.20Mbit) (s1002, s2007) (0.20Mbit)
 (0.20Mbit) (s1003, s2002) (0.20Mbit) (0.20Mbit) (s1003, s2004) (0.20Mbit) (0.20Mbit) (s1003, s2006) (
0.20Mbit) (0.20Mbit) (s1003, s2008) (0.20Mbit) (0.20Mbit) (s1004, s2002) (0.20Mbit) (0.20Mbit) (s1004,
 s2004) (0.20Mbit) (0.20Mbit) (s1004, s2006) (0.20Mbit) (0.20Mbit) (s1004, s2008) (0.10Mbit) (0.10Mbit
) (s2001, s3001) (0.10Mbit) (0.10Mbit) (s2001, s3002) (0.10Mbit) (0.10Mbit) (s2002, s3001) (0.10Mbit)
(0.10Mbit) (s2002, s3002) (0.10Mbit) (0.10Mbit) (s2003, s3003) (0.10Mbit) (0.10Mbit) (s2003, s3004) (0
.10Mbit) (0.10Mbit) (s2004, s3003) (0.10Mbit) (0.10Mbit) (s2004, s3004) (0.10Mbit) (0.10Mbit) (s2005,
s3005) (0.10Mbit) (0.10Mbit) (s2005, s3006) (0.10Mbit) (0.10Mbit) (s2006, s3005) (0.10Mbit) (0.10Mbit)
 (s2006, s3006) (0.10Mbit) (0.10Mbit) (s2007, s3007) (0.10Mbit) (0.10Mbit) (s2007, s3008) (0.10Mbit) (
0.10Mbit) (s2008, s3007) (0.10Mbit) (0.10Mbit) (s2008, s3008) (0.05Mbit) (0.05Mbit) (s3001, h001) (0.0
5Mbit) (0.05Mbit) (s3002, h002) (0.05Mbit) (0.05Mbit) (s3003, h003) (0.05Mbit) (0.05Mbit) (s3004, h004
) (0.05Mbit) (0.05Mbit) (s3005, h005) (0.05Mbit) (0.05Mbit) (s3006, h006) (0.05Mbit) (0.05Mbit) (s3007
, h007) (0.05Mbit) (0.05Mbit) (s3008, h008)
*** Configuring hosts
h001 h002 h003 h004 h005 h006 h007 h008
Unable to contact the remote controller at 127.0.0.1:6633
*** Starting controller
controller
*** Starting 20 switches
s1001 s1002 s1003 s1004 s2001 s2002 s2003 s2004 s2005 s2006 s2007 s2008 s3001 s3002 s3003 s3004 s3005
s3006 s3007 s3008 ...(0.20Mbit) (0.20Mbit) (0.20Mbit) (0.20Mbit) (0.20Mbit) (0.20Mbit) (0.20Mbit) (0.2
0Mbit) (0.20Mbit) (0.20Mbit) (0.20Mbit) (0.20Mbit) (0.20Mbit) (0.20Mbit) (0.20Mbit) (0.20Mbit) (0.20Mb
it) (0.20Mbit) (0.10Mbit) (0.10Mbit) (0.20Mbit) (0.20Mbit) (0.10Mbit) (0.10Mbit) (0.20Mbit) (0.20Mbit)
 (0.10Mbit) (0.10Mbit) (0.20Mbit) (0.20Mbit) (0.10Mbit) (0.10Mbit) (0.20Mbit) (0.20Mbit) (0.10Mbit) (0
.10Mbit) (0.20Mbit) (0.20Mbit) (0.10Mbit) (0.10Mbit) (0.20Mbit) (0.20Mbit) (0.10Mbit) (0.10Mbit) (0.20
Mbit) (0.20Mbit) (0.10Mbit) (0.10Mbit) (0.10Mbit) (0.10Mbit) (0.05Mbit) (0.10Mbit) (0.10Mbit) (0.05Mbi
t) (0.10Mbit) (0.10Mbit) (0.05Mbit) (0.10Mbit) (0.10Mbit) (0.05Mbit) (0.10Mbit) (0.10Mbit) (0.05Mbit)
(0.10Mbit) (0.10Mbit) (0.05Mbit) (0.10Mbit) (0.10Mbit) (0.05Mbit) (0.10Mbit) (0.10Mbit) (0.05Mbit)
*** Starting CLI:
mininet> █
```

Figure 6: Network Topology in Mininet.

5.2 Communication between Mininet and RYU Controller

To create a connection between network topology and controller, the switches communicate with the controller using the default port 6633. Each switch in the topology is assigned a unique port for keeping track of packages sent. In this project, 20 Openflow switches are connected to 8 hosts. Out of these 20 switches, 4 switches from S1001 to S1004 are the core switches then, 8 switches from S2001 to S2008 are the aggregation switches, and the last 8 switches are edges switches which are connected to 8 hosts in the network. To calculate the shortest path for the fat-tree topology, the controller script is executed. Figure. 7 below states that all the topology switches and ofp_handler events are loaded for the sending and receiving packets between the switches.



```
sayali@sayali-VirtualBox:~$ cd ryu
sayali@sayali-VirtualBox:~/ryu$ cd ryu
sayali@sayali-VirtualBox:~/ryu/ryu$ cd app
sayali@sayali-VirtualBox:~/ryu/ryu/app$ ryu-manager shortestpath.py --observe-links
loading app shortestpath.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app None of ArpHandler
creating context ArpHandler
instantiating app shortestpath.py of ShortestPath
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
```

Figure 7: Controller Files loading.

6 Evaluation

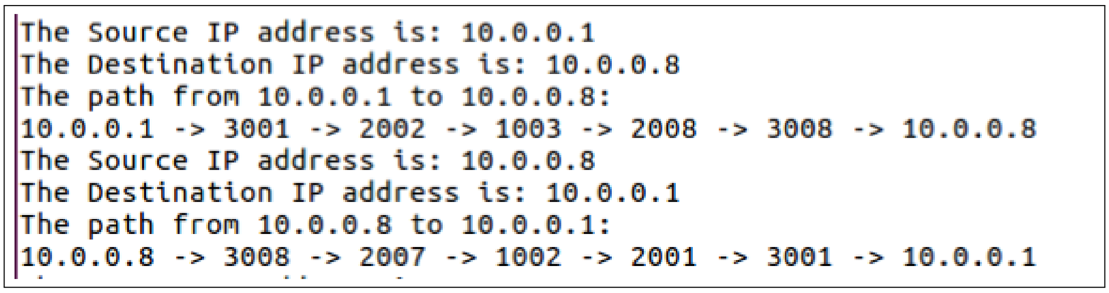
This section presents the outcome for the packet transmission between the source node to the destination node. The performance parameters considered for the packet simulations are Hop count, Delay (weight), Path bandwidth (throughput), Packets transferred, and Time.

6.1 Shortest Path Computation

In SDN, the shortest path is determined based on Open shortest Path First (OSPF), which communicates with the Openflow controller and routers in the network. The packet information is stored in ARPHandler in the controller, which is based on the Address Resolution Protocol (ARP). As the controller operates in two modes, i.e., proactive mode and reactive mode. In the proactive mode, the controller gets the information from switches, and in the reactive mode, the switches forward the ARP request to the controller and compute the shortest path.

6.1.1 Shortest Path with Hop Count

Figure. 8 evaluates the shortest path for H001 and H008. The hop count for path ”H001 to H008” and ”H008 to H001” is ”5,” which is the same. But, the path of packet flow through the switches is different.

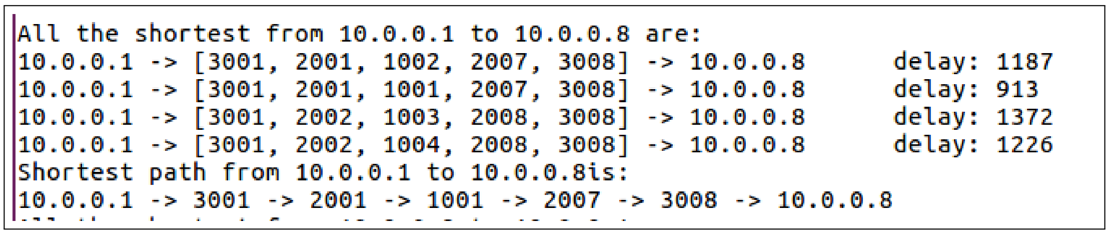


```
The Source IP address is: 10.0.0.1
The Destination IP address is: 10.0.0.8
The path from 10.0.0.1 to 10.0.0.8:
10.0.0.1 -> 3001 -> 2002 -> 1003 -> 2008 -> 3008 -> 10.0.0.8
The Source IP address is: 10.0.0.8
The Destination IP address is: 10.0.0.1
The path from 10.0.0.8 to 10.0.0.1:
10.0.0.8 -> 3008 -> 2007 -> 1002 -> 2001 -> 3001 -> 10.0.0.1
```

Figure 8: Shortest Paths between (Host1 and Host8).

6.1.2 Shortest Path with Delay

Figure. 9 evaluates the shortest path for H001 and H008. The delay (weight) for every path from hosts H001 to H008 is calculated. But,the delay with least is the shortest path for packet transmission. In this Figure. 9 the path whose is delay : 913 is the shortest path.



```
All the shortest from 10.0.0.1 to 10.0.0.8 are:
10.0.0.1 -> [3001, 2001, 1002, 2007, 3008] -> 10.0.0.8      delay: 1187
10.0.0.1 -> [3001, 2001, 1001, 2007, 3008] -> 10.0.0.8      delay: 913
10.0.0.1 -> [3001, 2002, 1003, 2008, 3008] -> 10.0.0.8      delay: 1372
10.0.0.1 -> [3001, 2002, 1004, 2008, 3008] -> 10.0.0.8      delay: 1226
Shortest path from 10.0.0.1 to 10.0.0.8is:
10.0.0.1 -> 3001 -> 2001 -> 1001 -> 2007 -> 3008 -> 10.0.0.8
```

Figure 9: Shortest Paths between (Host1 and Host8).

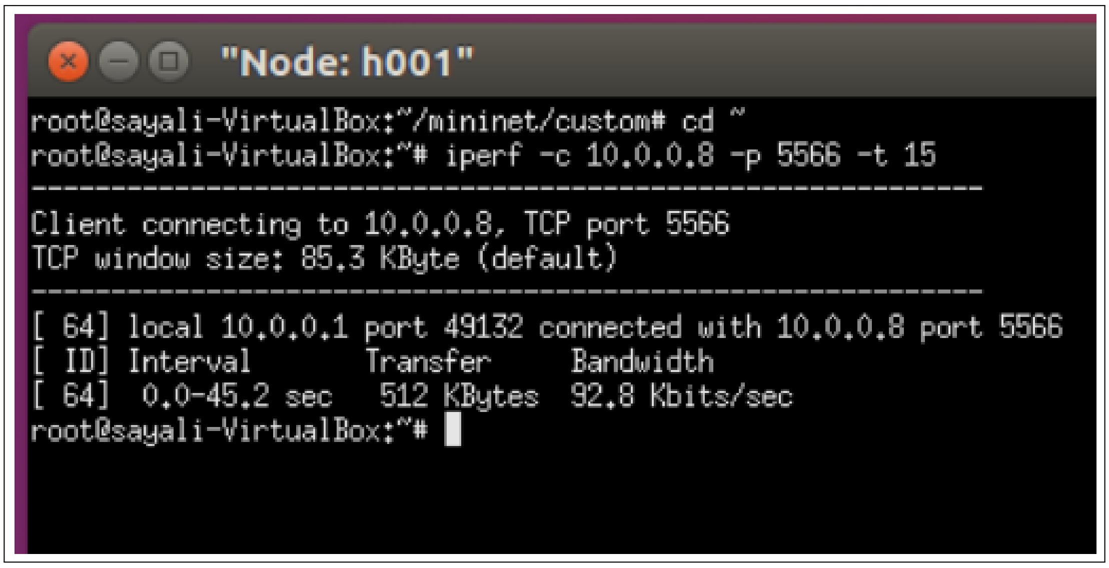
6.2 Experiment Results

6.2.1 Network throughput and Response Time

The bandwidth in the network is measured using the iperf tool. The controller in SDN uses RYU handlers and decorators for sending OpenFlow messages between the nodes. With the iperf tool, the TCP throughput along with UDP throughput and data loss is measured by sending and receiving TCP and UDP packets between pair of hosts. Also, the time taken by both TCP and UDP is calculated for data packets sent and received.

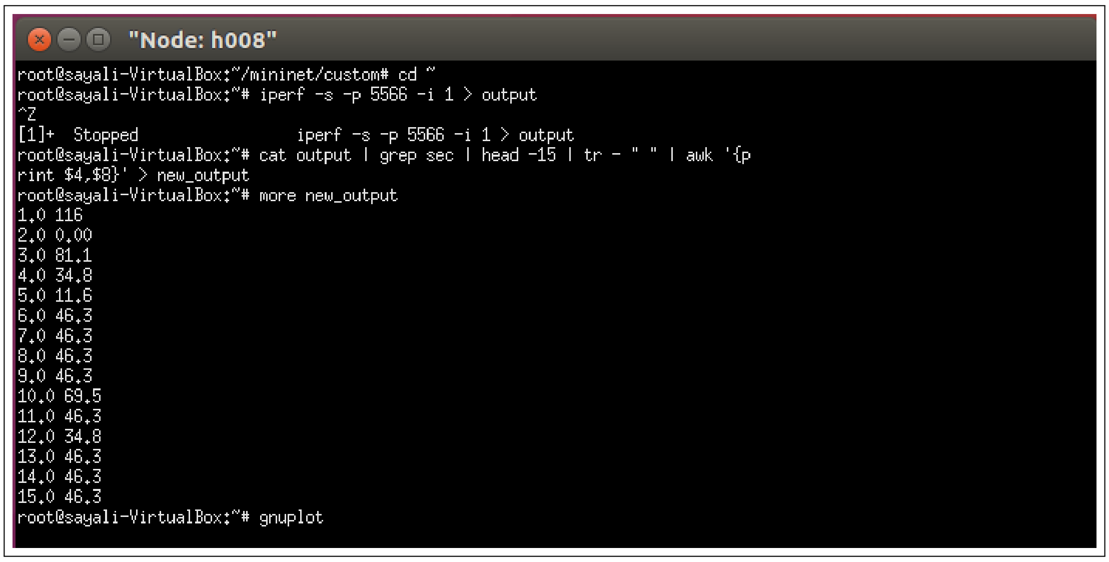
- **Case Study 1 : TCP Throughput Measurement**

The TCP packet transmission is carried out between two hosts i.e., H001 and H008. The H001 is set to client mode where the TCP packets are sent to H008 for a default 85.3 KByte window size, and it calculates the time and bandwidth for the amount of data sent. On the other side, H008 is set to server mode, which receives the TCP packets and calculates the same bandwidth and time for the data received. The Figure. 10 below represents as Client and Figure. 11 represents as Server. Thus, the average throughput between the time interval 0 - 45.2 sec sent from H001 (10.0.0.1) to H008 (10.0.0.8) is 92.8 Kbits/sec. And the total packet transfer is 512 KBytes.



```
root@sayali-VirtualBox:~/mininet/custom# cd ~
root@sayali-VirtualBox:~# iperf -c 10.0.0.8 -p 5566 -t 15
------------------------------------------------------------
Client connecting to 10.0.0.8, TCP port 5566
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 64] local 10.0.0.1 port 49132 connected with 10.0.0.8 port 5566
[ ID] Interval       Transfer     Bandwidth
[ 64]  0.0-45.2 sec   512 KBytes  92.8 Kbits/sec
root@sayali-VirtualBox:~# 
```

Figure 10: TCP traffic for Host1

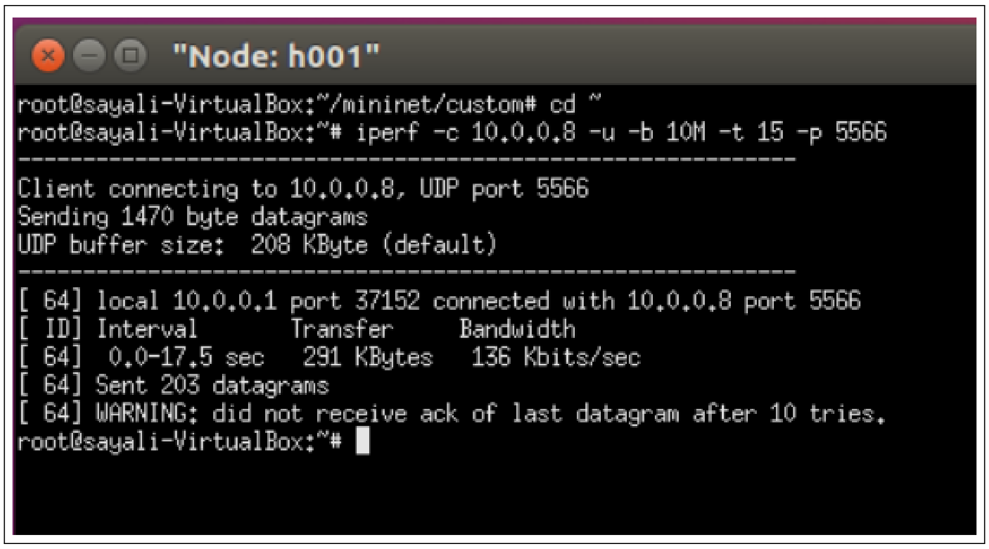


```
root@sayali-VirtualBox:~/mininet/custom# cd ~
root@sayali-VirtualBox:~# iperf -s -p 5566 -i 1 > output
^Z
[1]+  Stopped                 iperf -s -p 5566 -i 1 > output
root@sayali-VirtualBox:~# cat output | grep sec | head -15 | tr - " " | awk '{p
rint $4,$8}' > new_output
root@sayali-VirtualBox:~# more new_output
1.0 116
2.0 0.00
3.0 81.1
4.0 34.8
5.0 11.6
6.0 46.3
7.0 46.3
8.0 46.3
9.0 46.3
10.0 69.5
11.0 46.3
12.0 34.8
13.0 46.3
14.0 46.3
15.0 46.3
root@sayali-VirtualBox:~# gnuplot
```

Figure 11: TCP traffic for Host8

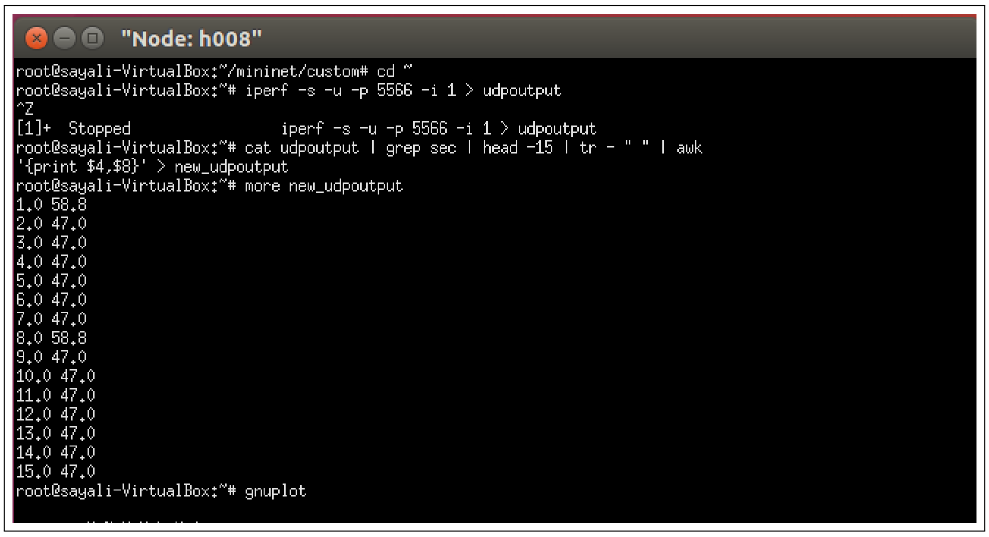
- **Case Study 2 : UDP Throughput and Data Loss Measurement**

Similarly, the TCP packet transmission is carried out between two hosts, i.e., H001 and H008. The H001 is set to client mode, where the UDP packets are sent to H008 with a 10M sending rate. It calculates the time, bandwidth for the amount of data sent. It also calculates the messages sent. On the other side, H008 is set to server mode, which receives the UDP packets and calculates the same bandwidth and time for the data received. Figure. 12 below represents as Client and Figure. 13 represents as Server. Thus, it can be stated that the average throughput between time interval 0 to 17.5 sec is 136 Kbits/sec with 291 KBytes of data sent. Also, 203 datagrams of messages were with few data loss after 10 tries.



```
root@sayali-VirtualBox:~/mininet/custom# cd ~
root@sayali-VirtualBox:~# iperf -c 10.0.0.8 -u -b 10M -t 15 -p 5566
------------------------------------------------------------
Client connecting to 10.0.0.8, UDP port 5566
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 64] local 10.0.0.1 port 37152 connected with 10.0.0.8 port 5566
[ ID] Interval       Transfer     Bandwidth
[ 64]  0.0-17.5 sec   291 KBytes   136 Kbits/sec
[ 64] Sent 203 datagrams
[ 64] WARNING: did not receive ack of last datagram after 10 tries.
root@sayali-VirtualBox:~# 
```

Figure 12: UDP traffic for Host1



```
root@sayali-VirtualBox:~/mininet/custom# cd ~
root@sayali-VirtualBox:~# iperf -s -u -p 5566 -i 1 > udpoutput
^Z
[1]+  Stopped                 iperf -s -u -p 5566 -i 1 > udpoutput
root@sayali-VirtualBox:~# cat udpoutput | grep sec | head -15 | tr - " " | awk
'{print $4,$8}' > new_udpoutput
root@sayali-VirtualBox:~# more new_udpoutput
1.0 58.8
2.0 47.0
3.0 47.0
4.0 47.0
5.0 47.0
6.0 47.0
7.0 47.0
8.0 58.8
9.0 47.0
10.0 47.0
11.0 47.0
12.0 47.0
13.0 47.0
14.0 47.0
15.0 47.0
root@sayali-VirtualBox:~# gnuplot
```

Figure 13: UDP traffic for Host8

- **Case Study 3 : Comparison of TCP and UDP Measurement**

In a network, the performance of packet transmission from source to destination is dependent on the bandwidth of data and the average data rate of packets transferred. In general, the difference between TCP and UDP bandwidth is that the amount of data generated in a network is calculated by TCP, while in UDP, the rate of data to be transmitted needs to be defined. Figure. 14 and Figure. 15 represents the throughput for a specified time interval. The graph is plotted using *gnuplot* tool where X-axis is set to "Time (sec)", and Y-axis is set to "Throughput (Gbps)". Thus, It can be stated that for TCP, the bandwidth keeps on fluctuating, and the highest bandwidth measured was around 119.8 Gbps at 0 to 2 sec time interval. While for the UDP, the highest bandwidth was 58.8 Gbps at 0 to 2 sec time interval.

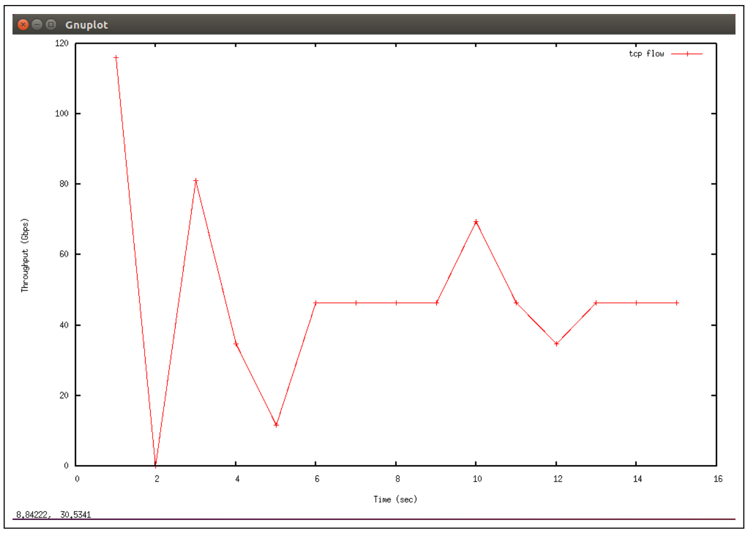


Figure 14: TCP Throughput

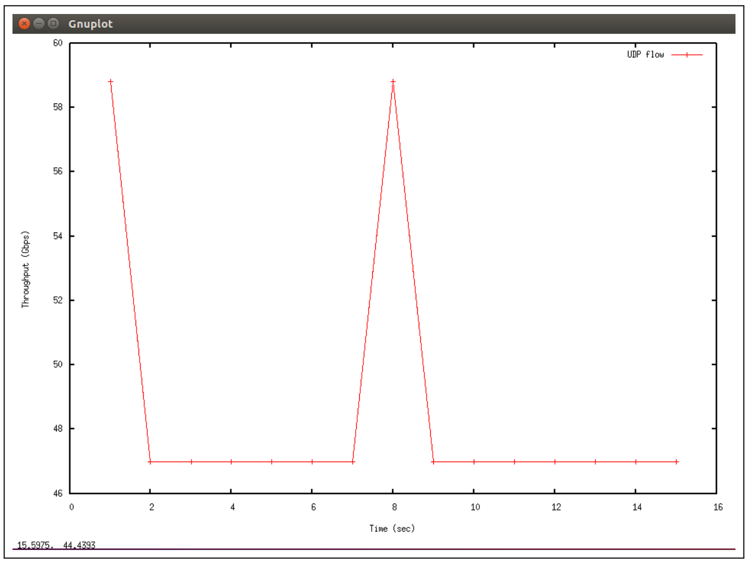


Figure 15: UDP Throughput

- **Case Study 4 : Comparison of Hop Count and Delay**

In a network, the hop count is dependent on the number of switches that a packet transmits between the source and destination. It the distance measured in the network based on the number of networking devices (switches). In general, if the hop count is less, it cannot be stated that the packet transmission between the source and destination will be faster. Also, if the hop count is high, it might transfer packet faster via different paths. The delay is the weight of the links in the network. In this project, the delay for all the paths in the network is calculated, and finally, the delay with the least weight is the shortest path. Figure. 16 represents the comparison of hop count and delay of the network for the transmission of the packets from source to destination. The hop count with "3" are all grouped, and similarly, hop count with "5" are grouped together. The graph also represents the delay for every shortest path calculated.

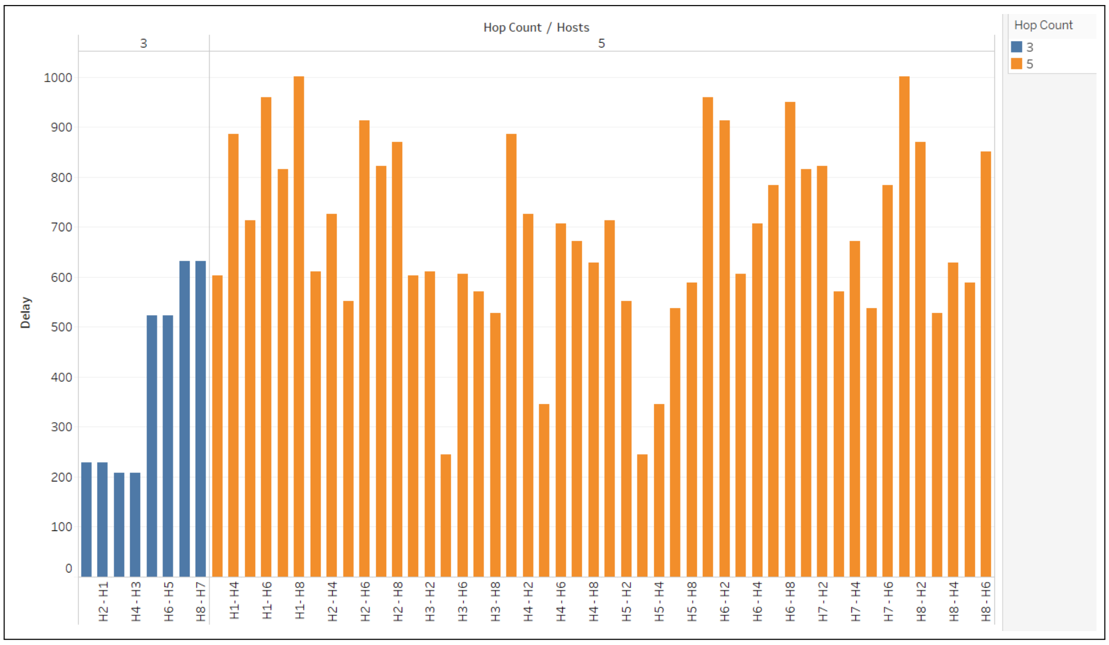


Figure 16: Hop Count and Delay Comparison

6.3 Discussion

In this project, experiments were performed based on four case studies. The simulation results show that the proposed algorithm finds the shortest path for fat-tree topology. The shortest path found is based on two parameters of network hop count and delay, which further determines the TCP and UDP bandwidth of the network. While performing these experiments, it was found that not all the time connectivity was established between the hosts. Also, in case study 3, not all the data packets were sent for UDP traffic, there was some data loss after 10 tries. In case study 4, it can be found that for every hop count for the hosts in the network was 3 and 5; also, the delay for every path kept on fluctuating.

7 Conclusion and Future Work

Today SDN has changed the traditional network into a more flexible and programmable platform that creates and supports virtualization of large networks. In SDN, though separation of data plane and control plan has improved the network performance, but due to network traffic in the control plane, the bandwidth(throughput) is reduced. Thus, in this project, a virtual simulation environment was created where path selection is made for packet transfer from the source node to the destination node. The proposed algorithm implemented for fat-tree network topology brings to the insights that the shortest path with least hop count and delay improves the network throughput.

In the future, this project can be enhanced using Layered Shortest Path Algorithm for different custom network topologies and different controllers. Also, this project is implemented on local machines considering a small network of 8 hosts and 20 switches so, in future Wide Area Network can be considered.

7.1 Acknowledgements

I would like to be grateful for the assistance given by mentor **Muhammad Iqbal** for his valuable and continuous suggestions during the research and development of this project.

Besides my mentor, I am also thankful to my program director and the head of Cloud Competency Center **Dr.Horacio Gonzalez-Velez** for his initial guidance and support in the early stages of research.

I wish to thank my parents and friends for their support and encouragement throughout my studies.

References

Akin, E. and Korkmaz, T. (2019). Comparison of routing algorithms with static and dynamic link cost in sdn, *2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC)*, Las Vegas, NV, USA, pp. 1–8.

Asadollahi, S., Goswami, B. and Sameer, M. (2018). Ryu controller's scalability experiment on software defined networks, *2018 IEEE International Conference on Current Trends in Advanced Computing (ICCTAC)*, pp. 1–5.

Challa, R., Jeon, S., Kim, D. S. and Choo, H. (2017). Centflow: Centrality-based flow balancing and traffic distribution for higher network utilization., *IEEE Access, Access, IEEE* p. 17045.

Cox, J. H., Chung, J., Donovan, S., Ivey, J., Clark, R. J., Riley, G. and Owen, H. L. (2017). Advancing software-defined networks: A survey, *IEEE Access* **5**: 25487–25526.

de Oliveira, R. L. S., Schweitzer, C. M., Shinoda, A. A. and Ligia Rodrigues Prete (2014). Using mininet for emulation and prototyping software-defined networks, *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pp. 1–6.

Fernández, P., J.A, Villalba, G., L.J, Kim and T.-H (2018). Software defined networks in wireless sensor architectures, *Entropy* **20**: 225.
URL: *http://www.sciencedirect.com/science/article/pii/S0167739X17306453*

Govindarajan, K., Kong Chee Meng and Hong Ong (2013). A literature review on software-defined networking (sdn) research topics, challenges and solutions, *2013 Fifth International Conference on Advanced Computing (ICoAC)*, pp. 293–299.

Jarraya, Y., Madi, T. and Debbabi, M. (2014). A survey and a layered taxonomy of software-defined networking, *IEEE Communications Surveys Tutorials* **16**(4): 1955–1980. Impact Factor: 20.230 (2018).

Jesús Antonio Puente, F., Luis Javier García, V. and Tai-Hoon, K. (2018). Software defined networks in wireless sensor architectures., *Entropy, Vol 20, Iss 4, p 225 (2018)* (4): 225. Impact Factor: 2.305 (2018).

Jiang, J., Huang, H., Liao, J. and Chen, S. (2014). Extending dijkstra's shortest path algorithm for software defined networking, *The 16th Asia-Pacific Network Operations and Management Symposium*, Hsinchu, Taiwan, pp. 1–4.

Karakus, M. and Durresi, A. (2017). A survey: Control plane scalability issues and approaches in software-defined networking (sdn), *Computer Networks* **112**: 279–293.

Kreutz, D., Ramos, F. M. V., Veríssimo, P. E., Rothenberg, C. E., Azodolmolky, S. and Uhlig, S. (2015). Software-defined networking: A comprehensive survey, *Proceedings of the IEEE* **103**(1): 14–76.

Lee, D., Hong, P. and Li, J. (2015). Rpa-ra: A resource preference aware routing algorithm in software defined network, *2015 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6.

Lee, M.-C. and Sheu, J.-P. (2016a). An efficient routing algorithm based on segment routing in software-defined networking, *Computer Networks* **103**: 44 – 55.
URL: *http://www.sciencedirect.com/science/article/pii/S1389128616300871*

Lee, M.-C. and Sheu, J.-P. (2016b). An efficient routing algorithm based on segment routing in software-defined networking., *Computer Networks* **103**: 44 – 55.

Li, Z., Ji, L., Huang, R. and Liu, S. (2018). Improving centralized path calculation based on graph compression, *China Communications* **15**(6): 120–124.

Lin, Y., Teng, H., Hsu, C., Liao, C. and Lai, Y. (2016). Fast failover and switchover for link failures and congestion in software defined networks, *2016 IEEE International Conference on Communications (ICC)*, Kuala Lumpur, Malaysia, pp. 1–6.

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

Paluck and Jain, S. (2018). Performance evaluation of multi hop routing using software defined network, *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, pp. 1–5.

Raghavendra, R., Lobo, J. and Lee, K.-W. (2012). Dynamic graph query primitives for sdn-based cloudnetwork management, *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, ACM, New York, NY, USA, pp. 97–102.
URL: *http://doi.acm.org/10.1145/2342441.2342461*

Son, J. and Buyya, R. (2018). A taxonomy of software-defined networking (sdn)-enabled cloud computing, *ACM Computing Surveys* **51**: 1–36.

Sun, Q., Wan, W., Chen, G. and Feng, X. (2016). Path planning algorithm under specific constraints in weighted directed graph, *2016 International Conference on Audio, Language and Image Processing (ICALIP)*, pp. 635–640.

Wu, Z., Lu, K., Wang, X. and Chi, W. (2016). Alleviating network congestion for hpc clusters with fat-tree interconnection leveraging software-defined networking, *2016 3rd International Conference on Systems and Informatics (ICSAI)*, pp. 808–813.

Xu, Q., Zhang, X., Zhao, J., Wang, X. and Wolf, T. (2016). Fast shortest-path queries on large-scale graphs, *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, Singapore, pp. 1–10.

Zhang, L., Deng, Q., Su, Y. and Hu, Y. (2017). A box-covering-based routing algorithm for large-scale sdns, *IEEE Access* **5**: 4048–4056.

8 Appendix- Configuration Manual

8.1 Introduction

In the proposed project, an algorithm is developed for routing a packet from the source node to the destination node in the SDN network. This configuration manual explains the steps taken to implement the shortest path algorithm.

8.2 Installation Prerequisite

To achieve efficient network throughput in SDN, the section below explains the simulation environment set up for the evaluation of results.

8.2.1 Operating System Environment

In this project, Intel Core i5 8th Gen machine with 8GB RAM is used as a host machine. On this to create a cross-platform virtualization Oracle Virtual Box is installed. Later Ubuntu 16.04 operating system is installed on Oracle VB.

8.2.2 Mininet

It is a network simulator tool that creates a virtual network on a single machine and connects multiple hosts and switches. Run the following command in the terminal to download source code for Mininet [2] from GitHub.

```
$ git clone git://github.com/mininet/mininet
```

To create a network topology Mininet tool creates a link between hosts and switches. In this project, fat-tree topology is created using "8 hosts" and "20 switches". This topology is controlled by a remote-controlled to form communication between every host and switch. To run the topology file go to the folder mininet, i.e. "cd mininet/custom" and run the following command:

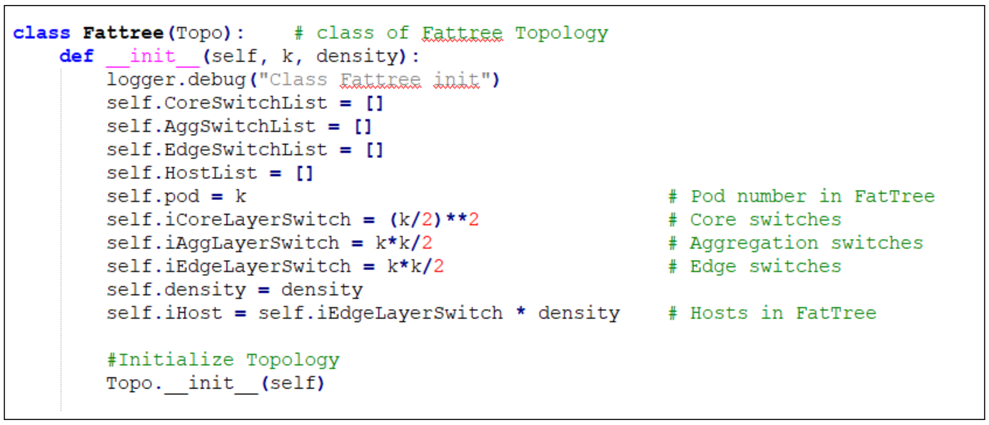
```
sudo python tree_topology.py
```

- **Hosts and Switches Initialized and Topology Creation:**

  The class Fattree is created as shown in Figure. 17 and all the Core switches, Aggregation switches, and Edge Switches are initialized. The hosts are defined on the basis of edge switches density.

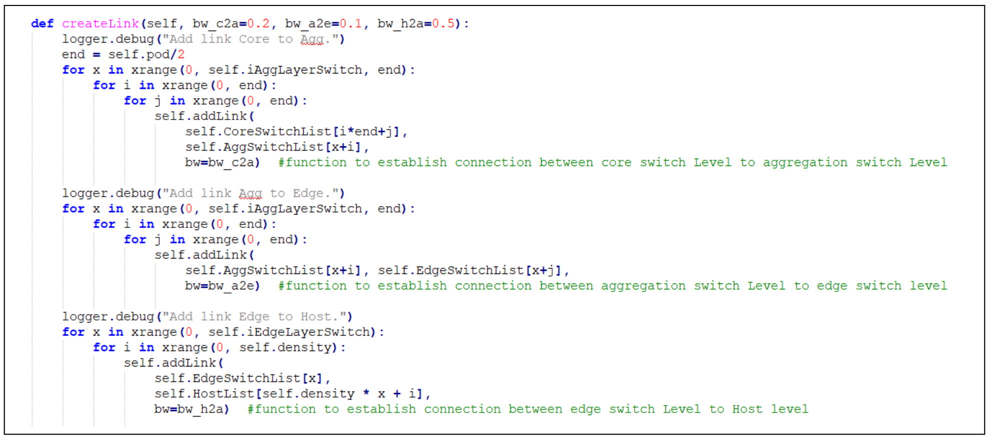
The code below Figure. 18 shows that every layered switch is connected to each other, and the last layer edge switches are connected to hosts.

[2] `http://mininet.org/download/`



```python
class Fattree(Topo):    # class of Fattree Topology
    def __init__(self, k, density):
        logger.debug("Class Fattree init")
        self.CoreSwitchList = []
        self.AggSwitchList = []
        self.EdgeSwitchList = []
        self.HostList = []
        self.pod = k                                        # Pod number in FatTree
        self.iCoreLayerSwitch = (k/2)**2                    # Core switches
        self.iAggLayerSwitch = k*k/2                        # Aggregation switches
        self.iEdgeLayerSwitch = k*k/2                       # Edge switches
        self.density = density
        self.iHost = self.iEdgeLayerSwitch * density        # Hosts in FatTree

        #Initialize Topology
        Topo.__init__(self)
```

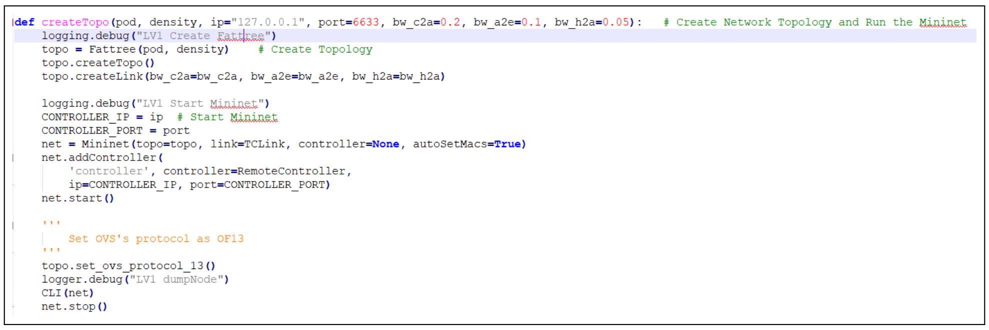
Figure 17: Topology Initialized



```python
def createLink(self, bw_c2a=0.2, bw_a2e=0.1, bw_h2a=0.5):
    logger.debug("Add link Core to Agg.")
    end = self.pod/2
    for x in xrange(0, self.iAggLayerSwitch, end):
        for i in xrange(0, end):
            for j in xrange(0, end):
                self.addLink(
                    self.CoreSwitchList[i*end+j],
                    self.AggSwitchList[x+i],
                    bw=bw_c2a)  #function to establish connection between core switch Level to aggregation switch Level

    logger.debug("Add link Agg to Edge.")
    for x in xrange(0, self.iAggLayerSwitch, end):
        for i in xrange(0, end):
            for j in xrange(0, end):
                self.addLink(
                    self.AggSwitchList[x+i], self.EdgeSwitchList[x+j],
                    bw=bw_a2e)  #function to establish connection between aggregation switch Level to edge switch level

    logger.debug("Add link Edge to Host.")
    for x in xrange(0, self.iEdgeLayerSwitch):
        for i in xrange(0, self.density):
            self.addLink(
                self.EdgeSwitchList[x],
                self.HostList[self.density * x + i],
                bw=bw_h2a)  #function to establish connection between edge switch Level to Host level
```

Figure 18: Links created Code

The code below Figure. 19 shows the creation of fat-tree topology.



```python
def createTopo(pod, density, ip="127.0.0.1", port=6633, bw_c2a=0.2, bw_a2e=0.1, bw_h2a=0.05):   # Create Network Topology and Run the Mininet
    logging.debug("LV1 Create Fattree")
    topo = Fattree(pod, density)     # Create Topology
    topo.createTopo()
    topo.createLink(bw_c2a=bw_c2a, bw_a2e=bw_a2e, bw_h2a=bw_h2a)

    logging.debug("LV1 Start Mininet")
    CONTROLLER_IP = ip  # Start Mininet
    CONTROLLER_PORT = port
    net = Mininet(topo=topo, link=TCLink, controller=None, autoSetMacs=True)
    net.addController(
        'controller', controller=RemoteController,
        ip=CONTROLLER_IP, port=CONTROLLER_PORT)
    net.start()

    '''
        Set OVS's protocol as OF13
    '''
    topo.set_ovs_protocol_13()
    logger.debug("LV1 dumpNode")
    CLI(net)
    net.stop()
```

Figure 19: Topology Code

8.2.3 RYU Controller

To create links and communication between hosts and switches, the RYU controller is used in this project. It is used to write the shortest path algorithm code in python. Run the following command in the terminal the command to download source code for RYU [3] from GitHub.

[3]https://osrg.github.io/ryu/

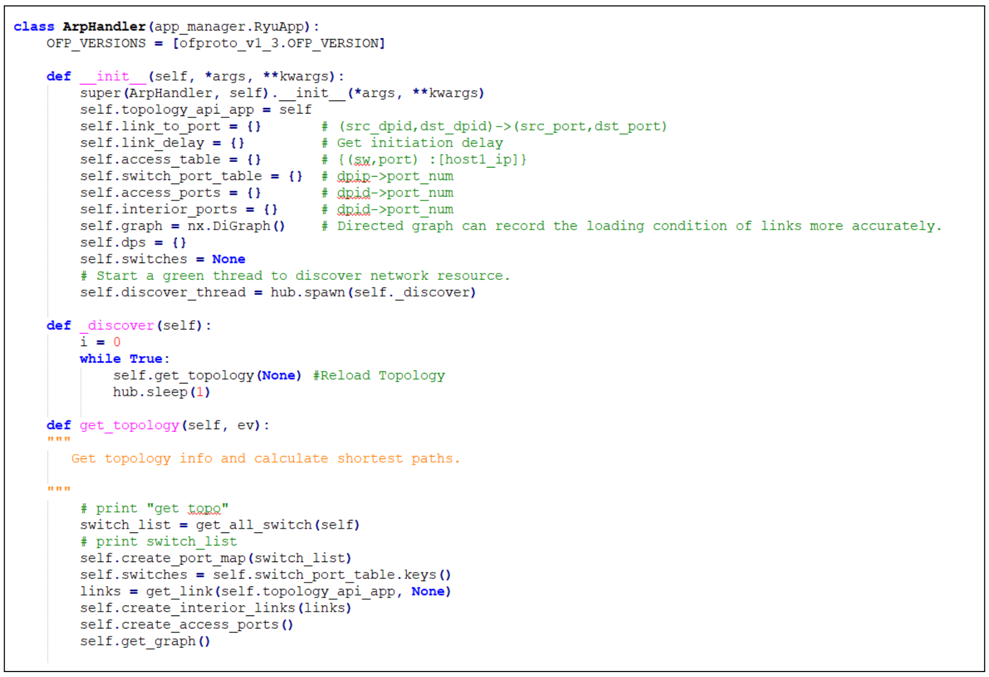
$ git clone git://github.com/osrg/ryu.git

In the RYU controller, the shortest path code is written for hop count and delay. It contains four different files in two different folders. These files are located in ”cd /ryu/ryu/app/shortest-path-hop” and ”cd /ryu/ryu/app/shortest-path-delay. After executing the topology creation command in one terminal. Open another terminal and run the following command for first executing the shortest path algorithm for hop count.

ryu-manager shortestpath-hop.py –observe-links

- **ARPHandler and Shortest Path Initialization and Creation**

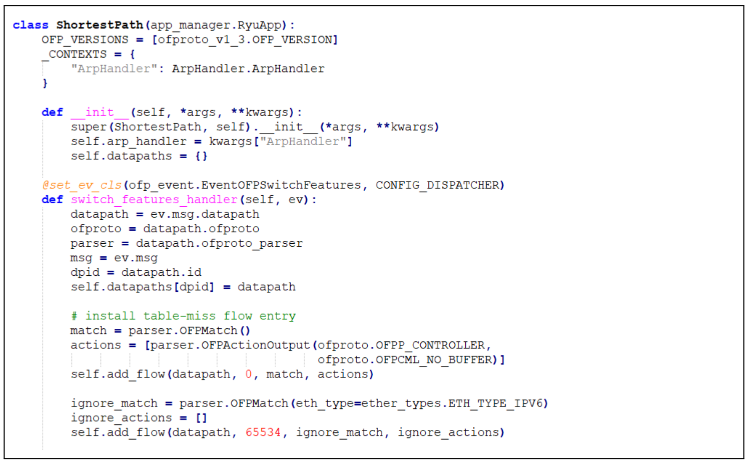
  When the file is executed, the ARPHandler app class is created and initialized (Figure 20) to all the information on network topology. The class initialization is the same for both the shortest path files.



```python
class ArpHandler(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(ArpHandler, self).__init__(*args, **kwargs)
        self.topology_api_app = self
        self.link_to_port = {}       # (src_dpid,dst_dpid)->(src_port,dst_port)
        self.link_delay = {}         # Get initiation delay
        self.access_table = {}       # {(sw,port) :[host1_ip]}
        self.switch_port_table = {}  # dpip->port_num
        self.access_ports = {}       # dpid->port_num
        self.interior_ports = {}     # dpid->port_num
        self.graph = nx.DiGraph()    # Directed graph can record the loading condition of links more accurately.
        self.dps = {}
        self.switches = None
        # Start a green thread to discover network resource.
        self.discover_thread = hub.spawn(self._discover)

    def _discover(self):
        i = 0
        while True:
            self.get_topology(None) #Reload Topology
            hub.sleep(1)

    def get_topology(self, ev):
    """
       Get topology info and calculate shortest paths.

    """
        # print "get topo"
        switch_list = get_all_switch(self)
        # print switch_list
        self.create_port_map(switch_list)
        self.switches = self.switch_port_table.keys()
        links = get_link(self.topology_api_app, None)
        self.create_interior_links(links)
        self.create_access_ports()
        self.get_graph()
```

Figure 20: ARPHandler Class created

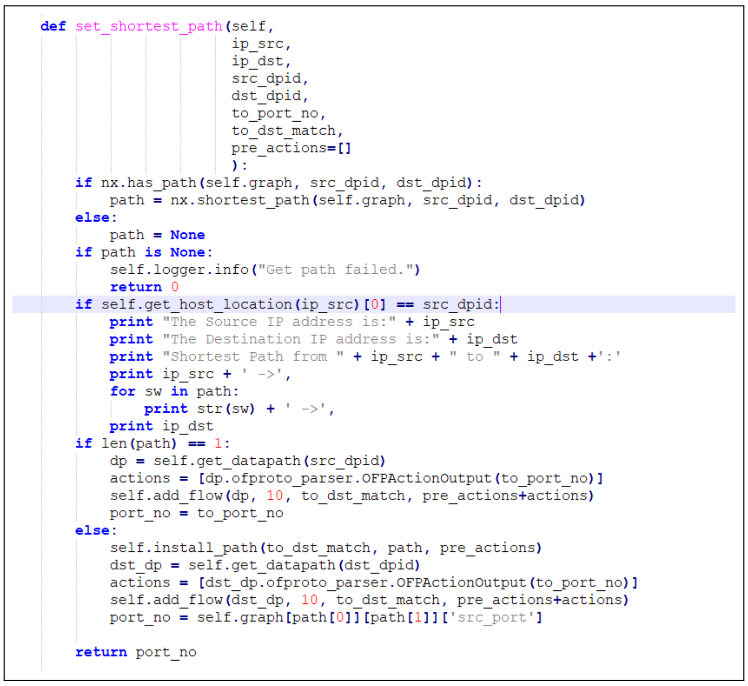
After the ARPHandler app is initialized (Figure 21), the Class *ShortestPath* from the shortest path files is initialized where all the information of paths is stored in *datapaths.*



```python
class ShortestPath(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {
        "ArpHandler": ArpHandler.ArpHandler
    }

    def __init__(self, *args, **kwargs):
        super(ShortestPath, self).__init__(*args, **kwargs)
        self.arp_handler = kwargs["ArpHandler"]
        self.datapaths = {}

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        msg = ev.msg
        dpid = datapath.id
        self.datapaths[dpid] = datapath

        # install table-miss flow entry
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

        ignore_match = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IPV6)
        ignore_actions = []
        self.add_flow(datapath, 65534, ignore_match, ignore_actions)
```

Figure 21: Shortest Path Class created

The shortest path for hop count is calculated based on the following code Figure. 22



```python
    def set_shortest_path(self,
                          ip_src,
                          ip_dst,
                          src_dpid,
                          dst_dpid,
                          to_port_no,
                          to_dst_match,
                          pre_actions=[]
                          ):
        if nx.has_path(self.graph, src_dpid, dst_dpid):
            path = nx.shortest_path(self.graph, src_dpid, dst_dpid)
        else:
            path = None
        if path is None:
            self.logger.info("Get path failed.")
            return 0
        if self.get_host_location(ip_src)[0] == src_dpid:
            print "The Source IP address is:" + ip_src
            print "The Destination IP address is:" + ip_dst
            print "Shortest Path from " + ip_src + " to " + ip_dst +':'
            print ip_src + ' ->',
            for sw in path:
                print str(sw) + ' ->',
            print ip_dst
        if len(path) == 1:
            dp = self.get_datapath(src_dpid)
            actions = [dp.ofproto_parser.OFPActionOutput(to_port_no)]
            self.add_flow(dp, 10, to_dst_match, pre_actions+actions)
            port_no = to_port_no
        else:
            self.install_path(to_dst_match, path, pre_actions)
            dst_dp = self.get_datapath(dst_dpid)
            actions = [dst_dp.ofproto_parser.OFPActionOutput(to_port_no)]
            self.add_flow(dst_dp, 10, to_dst_match, pre_actions+actions)
            port_no = self.graph[path[0]][path[1]]['src_port']

        return port_no
```

Figure 22: Shortest Path Hop Count Code

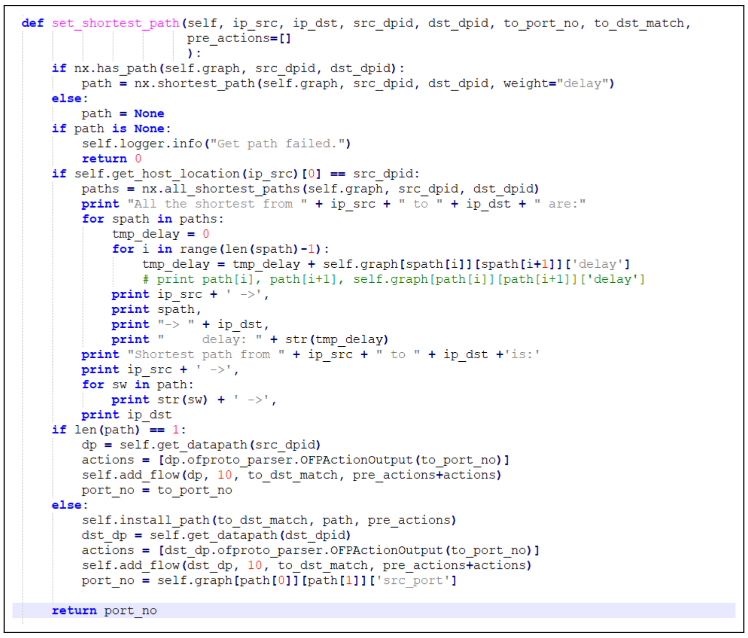
Similarly, after executing the algorithm for hop count again follow the same commands from the creation of network topology and then run the following command for executing the shortest path algorithm for the delay.

ryu-manager shortestpath-delay.py –observe-links

The code below Figure. 23 explains the logic to set the shortest path for the delay.

8.2.4 Iperf Tool

To check the performance of the network after executing the algorithm, *Iperf* tool is used for evaluating TCP and UDP traffic. From the terminal where mininet shell is running,



```python
def set_shortest_path(self, ip_src, ip_dst, src_dpid, dst_dpid, to_port_no, to_dst_match,
                      pre_actions=[]
                      ):
    if nx.has_path(self.graph, src_dpid, dst_dpid):
        path = nx.shortest_path(self.graph, src_dpid, dst_dpid, weight="delay")
    else:
        path = None
    if path is None:
        self.logger.info("Get path failed.")
        return 0
    if self.get_host_location(ip_src)[0] == src_dpid:
        paths = nx.all_shortest_paths(self.graph, src_dpid, dst_dpid)
        print "All the shortest from " + ip_src + " to " + ip_dst + " are:"
        for spath in paths:
            tmp_delay = 0
            for i in range(len(spath)-1):
                tmp_delay = tmp_delay + self.graph[spath[i]][spath[i+1]]['delay']
                # print path[i], path[i+1], self.graph[path[i]][path[i+1]]['delay']
            print ip_src + ' ->',
            print spath,
            print "-> " + ip_dst,
            print "     delay: " + str(tmp_delay)
        print "Shortest path from " + ip_src + " to " + ip_dst +'is:'
        print ip_src + ' ->',
        for sw in path:
            print str(sw) + ' ->',
        print ip_dst
    if len(path) == 1:
        dp = self.get_datapath(src_dpid)
        actions = [dp.ofproto_parser.OFPActionOutput(to_port_no)]
        self.add_flow(dp, 10, to_dst_match, pre_actions+actions)
        port_no = to_port_no
    else:
        self.install_path(to_dst_match, path, pre_actions)
        dst_dp = self.get_datapath(dst_dpid)
        actions = [dst_dp.ofproto_parser.OFPActionOutput(to_port_no)]
        self.add_flow(dst_dp, 10, to_dst_match, pre_actions+actions)
        port_no = self.graph[path[0]][path[1]]['src_port']

    return port_no
```

Figure 23: Shortest Path Delay Code

use *xterm h008* and *xterm h001* command to open the node windows. Then create node h008 as server and h001 as a client. Execute the following commands for TCP traffic from root:

- For h008 : iperf -s -p 5566 -i 1 > output
- For h001: iperf -c 10.0.0.8 -p 5566 -t 15

In this command *s* denote server, *p* denote port number, *i* and *t* denote time, and *c* denote client. The results are redirected to the "output" file.

Execute the following commands for UDP traffic from root:

- For h008 : iperf -s -u -p 5566 -i 1 > udpoutput
- For h001: iperf -c 10.0.0.8 -u -b 10M -t 15 -p 5566

In this command *s* denotes server, *u* denote udp server, *p* denote port number, *i* and *t* denote time, *c* denote client and *b* denote packet sending rate.

8.2.5 Gnuplot Tool

To plot a graph for the evaluation results based on network throughput vs. time in TCP and UDP *gnuplot* tool is used. To install *gnuplot* execute the following command in a new terminal.

sudo apt-get install gnuplot-nox

To plot the results initially execute the following command:

- For TCP from node h008:

```
cat output | grep sec | head -15 | tr – “ “ | awk ‘{print $4, $8}’ > new_output
```

- For UDP from node h008:

```
cat udpoutput | grep sec | head -15 | tr – “ “ | awk ‘{print $4, $8}’ > new_udpoutput
```

Then from the h008 terminal node run *gnuplot*, which redirects to gnuplot shell. The shell starts and executes the following command for TCP and UDP:

plot ”new_output” title ”TCP Flow” with linespoints”

plot ”new_result” title ”UDP Flow” with linespoints”

Run *set xlabel ”Time (sec)”* to set X-axis and *set ylabel ”Throughput (Gbps)”* to set Y-axis.

8.3 Proposed Shortest path Analysis

After executing the commands from Section. 8.2.3, the hop count for every shortest path between all hosts is taken down in excel. Similarly, the least delay (weight) for every shortest path is taken down in excel. This data is further imported to the Visualization tool *Tableau* for analysis. A bar graph is plotted, as shown in Figure. 16. Thus, from the graph, it can be stated that most of the shortest paths have hop count 5, and the delay is different for every shortest path.