TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

THESIS NO: 071/MSI/613

Dynamic Load Balancing in Software Defined Networking

by

Sadhu Ram Basnet

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN INFORMATION AND COMMUNICATION ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

LALITPUR, NEPAL

OCTOBER, 2016

Dynamic Load Balancing in Software Defined Networking

by

Sadhu Ram Basnet

071/MSI/613

Supervisor

Prof. Dr. Subarna Shakya

A thesis submitted in partial fulfillment of the requirement for the Degree of Master of Science in Information and Communication Engineering

Department of Electronics and Computer Engineering

Institute of Engineering, Pulchowk Campus

Tribhuvan University

Lalitpur, Nepal

October, 2016

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

The undersigned certify that they have read and recommended to the Institute of Engineering for acceptance, a thesis entitled "Dynamic load balancing in Software Defined Networking", submitted by Sadhu Ram Basnet in partial fulfillment of the requirement for the award of the degree of "Master of Science in Information and Communication Engineering".

Defense Date: 26th October 2016 (10th Kartik 2073)

Examination Committee

Prof. Dr. Subarna Shakya ....
Supervisor

Prof. Dr. Shashidhar Ram Joshi ....
Chairperson, Evaluation Committee

Prof. Dr. Dinesh Kumar Sharma ....
Member, Evaluation Committee

Dr. Surendra Shrestha ....
Program Coordinator
Department of Electronics and Computer Engineering
Institute of Engineering, Pulchowk Campus
Tribhuvan University

....
External Examineer

DEPARTMENTAL ACCEPTANCE

The thesis entitled “Dynamic load balancing in Software Defined Networking”, submitted by Sadhu Ram Basnet in partial fulfillment of the requirement for the award of the degree of “Master of Science in Information and Communication Engineering” has been accepted as a bonafide record of work independently carried out by him in the department.

____

Dr. Dibakar Raj Pant

Head of the Department

Department of Electronics and Computer Engineering

IOE,Pulchowk Campus, Tribhuvan University, Nepal

ABSTRACT

Networks need to handle a huge amount of traffic serving thousands of clients day by day. A single standalone server to cater such a huge load is almost impossible. The solution is to use multiple servers using load balancer at the front end. Traditional Load balancer uses dedicated hardware which forwards the client requests to different servers depending upon load balancing strategy. This sort of hardware is expensive and inflexible. Network administrators cannot create their own algorithms since traditional load balancer are vendor locked, non programmable.This thesis implements a dynamic load balancer which can balance server loads as well as path loads of the network in SDN environment. Moreover, the dynamic load balancer implements customized load balancing strategy. SDN load balancer is programmable and allows to design and implement own customized load balancing strategy. Other advantage of SDN load balancer is that it does not need dedicated hardware. In this thesis, OpenFlow protocol is used for communication in SDN environment. This better manages communication between number of hosts in the network. The result of dynamic load balancer use in this thesis shows that the server load as well as path load are better managed with a better traffic scheduling performance of network. Compared with the traditional load balancing method, this dynamic load balancing effectively improves the performance of the load balancing in the network and reduce the complexity of implementation.

Keywords: Software Defined Networking, Open Flow, Load Balancer, SDN Controller

ACKNOWLEDGEMENT

I am very thankful to the Department of Electronics and Computer Engineering, Pulchowk Campus, IOE for providing me the opportunity to do the thesis entitled "Dynamic Load Balancing in Software Defined Networking".

I express my sincere gratitude to my thesis supervisor Prof. Dr. Subarna Shakya, for his expert guidance, motivation, suggestions and encouragement. Without his guidance and persistent help, this thesis progress would not have been possible.

I would like to express special thanks of gratitude to the Head of Department of Electronics and Computer Engineering, IOE, Pulchowk Campus, Dr. Dibakar Raj Pant and the Program Coordinator of Master of Science in Information and Communication Engineering, Dr. Surendra Shrestha for their support.

I acknowledge my sincere gratitude to Prof. Dr. Sashidhar Ram Joshi, Prof. Dr. Dinesh Kumar Sharma, Assistant Prof. Dr. Nanda Bikram Adhikari, and teachers Mr. Dinesh Baniya, Mr. Dayasagar Baral, Mr. Baburam Dawadi and all the teachers of the Department of Electronics and Computer Engineering, Pulchowk Campus, IOE who have helped me. Finally, I would like to thank my family and friends for precious encouragement and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

BW	Bandwidth
CLI	Command Line Interface
DALB	Dynamic and Adaptive Load Balance
DLB	Dynamic Load Balance
ECMP	Equal Cost Multi Path
FC	Free Capacity
GCM	Gnome Connection Manager
JVM	Java Virtual Machine
LFC	Link Flows Collection
PC	Paths Collection
RR	Round Robin
RTT	Round Trip Time
SDN	Software Defined Networking
TCP	Transmission Control Protocol
UB	Used Bandwidth
UDP	User Datagram Protocol
VM	Virtual Machine
VLB	Valiant Load Balance
WORA	Write Once Run Anywhere
WRRA	Weighted Round Robin Algorithm

CHAPTER 1: INTRODUCTION

1.1 Background

In recent years, with the rapid expansion of Internet business, using the load balancing technology to deal with this challenge has become a back-end server necessary measure. The load balancer is a bridge between the network and server, load balancer usually need to learn the health status of the server, also don't need to be able to set up the network protocol, packet content information to modify or read. Traditional load balancing device are considered in the design of the operation condition of the server computer equipment (such as CPU utilization) to make a decision, but considering the network traffic is less load of network equipment, the lack of fine grained monitoring and scheduling, the main reason is that the current network system is a relatively closed system. Although the traditional router can allocate bandwidth between different paths, but only a thick line of control. In general, professional load balancing server needs to be done from two to seven layers of data processing, so it is easy to become the bottleneck of the whole service system [1].

With the emergence of Software Defined Networking (SDN) technology, network openness was the largest. Thus far, OpenFlow protocol which is the most widely used in a south interface protocol is the concrete embodiment of the concept of SDN. OpenFlow offers another kind of design method of load balancing, the server load balance based on SDN compared with the traditional load balancing method. It has a simple implementation and high performance characteristics.

1.2 Problem Statement

Data center networks are designed for satisfying the data transmission demand of densely interconnected hosts in the data center. The network topology and switching/routing mechanism can affect the performance and latency significantly. Nowadays, the fat-tree network is one of the most widely used typologies for data center networks. Network engineers also adopt load balancing methods in the design of switching and routing algorithms. However, the requirement of load balancing in fat-tree networks cannot be fully satisfied by traditional approaches. The main reason is the lack of efficient ways to obtain network traffic statistics from network devices. This degrades of quality of service of the network.

1.3 Objectives

The main objectives of this thesis are

- To implement a dynamic path load balancer in SDN.

- To implement a dynamic server load balancer in SDN data center.

CHAPTER 2: LITERATURE REVIEW

Many researches have been proposed on load balance in traditional multipath network. Two load balance strategies have been widely used in multipath network at present: Equal-Cost Multipath (ECMP) [2] and Valiant Load Balance (VLB). The core idea of ECMP is to evenly distribute data-flow to next-hop switches, and VLB distributes traffic among all available paths and randomly picks the next-hop switch. These two strategies both use fixed methods and cannot pick transmission path adaptively to the path load condition.

In SDN architecture, limited researches have been proposed on network load balance. A number of load balance system have been proposed in [3]-[5]. In these systems, controller is used to analyze replying information from OpenFlow switches and modify the flow-Tables by specific load balance strategy, so as to efficiently plan data transmission path and achieve load balance in SDN. But these strategies belong to static load balance method which cannot make dynamic routing plan according to real-time network load condition. Besides, these several methods take little advantages of SDN to make a better load balance design. A dynamic load balance algorithm, known as Dynamic Load Balance (DLB), has been proposed in [6]. The DLB algorithm simply applies greedy selection strategy to pick next-hop link which transmits least data load. Although these algorithm implements load balance on multipath SDN, this routing strategy is only decided by link load of every next-hop without combining the superiority of global network view in SDN. Hence, this routing strategy may not find the best transmission path in global view so that may not achieve the best load balance effect.

S. Bhandarkar, et. al. [7], describes how the best calculated path is obtained using the dynamic load balancer to reduce the collision and information loss, when the load on the link will be greater than the bandwidth of the link. The experiment results that it can handle more packets and having greater efficiency than round-robin load balancer in both the modes. Y. Zhou, et. al. [8], describes a load balancing strategy named Dynamic and Adaptive Load Balance (DALB) for SDN controller based on distributed decision. But there is not much focused attention on testing the algorithms in more detail.

I. Keslassy, et. al. [8-9], follows the two ideas of DIFANE. The network administrator has the authority to specify the policies which defines how the switches can forward, drop, modify and measure the traffic. It is an efficient solution which keeps the traffic in the data plane and forwards the packets through intermediate switches having necessary rules and the controller partition rules over the switches.

G Shou, et.al. [10], has introduced R-SDN. It has a vertically distributed control plane. Number of network/forwarding devices on each layer increases according to the Fibonacci series as the idea keep in mind that series increase like branches of a tree (spanning tree) with no loop. They manages the network by using Fibonacci heap ordered tree for load balancing and routing. The algorithm is solvable in polynomial time and gives less response time as compared to the traditional network.

Shi, et.al. [11], introduced the concept of Flow Slice (FS). It was used to divide each traffic flow into several flow slices and balance the load through various paths in a network. The paper claimed that if the setting of a slicing threshold was 1 to 4 milliseconds, the FS strategy could obtain nearly optimal performance. Based on the measurement, the paper presented various slice thresholds with other variables, such as Flow-Slice packet count, Flow Slice size, and Flow-Slice number, to find the impact. Finally, the paper measured delay, packet loss rate, and out-of-order packet value to determine the performance of the FS scheme.

F. Farina, et. al. [12], suggested author configuring a mesh Ethernet network using SDN topology showing L2 essential/necessary features e.g. creation of spanning tree is still missing in the SDN creation. They focus on typical computing centers of cloud where both loop free network topologies and their energy efficiency. GreenMST is the proposed prototype fulfills the basic requirements of loop free L2 network topology which is suitable or fit for various production and experimental network production. This prototype avoids the drawback of traditional non-openflow solutions like STP protocol by providing network applications to specify the metric dynamically, which is used by the controller for preparing the spanning tree. It reduces the energy consumption by switching off inactive ports. The future work focuses on providing the solutions on current prototype i.e. introducing the cache with the list of deactivate interfaces.

In SDN, distributed controllers [13] have been proposed to solve the scalability issues and reliability issues of network control plane. There is a limitation of distributed controllers, the mapping of switch and controller is configured statically due to which the load distributed among various controllers is not even. To solve this problem, this architecture is proposed in which the pool of controllers is shrink and grow dynamically according to the traffic on the link and load on each link is dynamically shifted across all controllers.

In Coronett, et. al.[14], the scalability of SDN is improved as compared to standard approach of SDN. VLAN reduces the number of packets forwarding rules and packet forwarding. It only specifies logical paths rather than physical paths. There are various openflow applications exists which directly control the packets path, these applications can be rewritten using CORONET architecture. In future work, author plan to evaluate the generality of CORONET to support common SDN applications and build a general framework which allows seamless integration with any SDN application. For the control plane, they check feasibility using traditional distributed mechanisms of the network like spanning tree protocol, and compare with a approach in which the controller reconfigures the control plane when faults are detected.

Yao Shen, et. al. [15], introduces a new algorithm focuses on the issue of load balancing and strategies of routing in SDN. Although there are various algorithms present on this issue but they are not suitable for the large flow distributed network because they don't consider load collision on the middle of the transmission of packets. They proposed an efficient algorithm for path switching to balance the uneven load exists on the network. The experimental results show that this algorithm gives better performance than the other one.

Hata, et. al. [16], assumed that openflow switch in SDN deals with multi-protocol packet header in various packets like Ethernet, HTTP, and SIP etc. He discussed architecture and requirements of a openflow switch to work with multi-protocol packet header. For this, more intelligent and programmable switch function is required. Therefore, he developed architecture by combining active network technology and SDN. It’s done with virtual CPU and memory called packet processor and a user program is loaded in it which is invoked packet by packet. All packet processing shouldn't do by this this user program. Several system calls, library functions and utility functions are provided by this platform to the user program. A component named transfer engine are programmed to support lower layer protocols.

Since, the user program handles various protocols. They prepare various transfer engines according to the platform. The controller has the responsibility to send the user program to the openflow switch and notifies what kind of transfer engine is invoked for that program. They proposed various load balancing servers (proxy) for HTTP using this platform.

CHAPTER 3: RELATED THEORY

3.1 Software Defined Networking

Software defined network [17] is a new emerging technology in the field of networking in which programs written in high-level languages like C, java, ruby, Perl etc for control plane by the network administrator is used to control the behavior of whole network. It deals with splitting of infrastructure layer from control layer which enhances the programming capability, flexibility, malleability and manageability of the network. In spite of having lots of benefits over traditional network availability of SDN, it is not friendly with the growing organizational network.

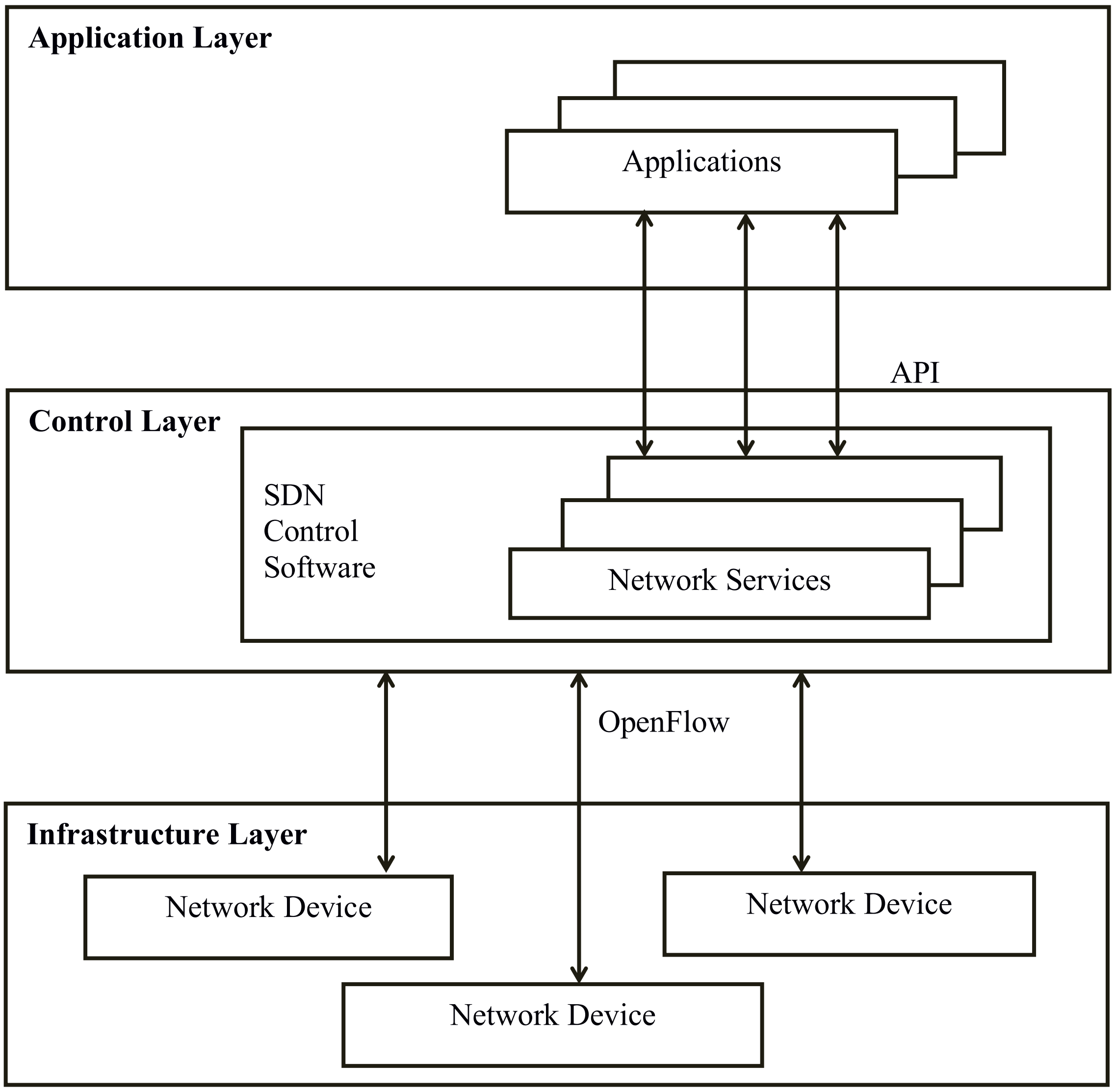


Figure 1: SDN Architecture

The architecture of SDN is clearly shown in the Figure 1. SDN is commonly associated with OpenFlow protocol.

3.2 Load Balancing

Load balancing is a methodology to distribute workload across multiple computers to achieve optimal resource utilization, maximize throughput, minimize response time and avoid overload [18].

Load balancer acts as the “traffic cop” sitting in front of your servers and routing client requests across all servers capable of fulfilling those requests in a manner that maximizes speed and capacity utilization and ensures that no one server is overworked, which could degrade performance. If a single server goes down, the load balancer redirects traffic to the remaining online servers. When a new server is added to the server group, the load balancer automatically starts to send requests to it .

In this manner, a load balancer performs the following functions:

- Distributes client requests or network load efficiently across multiple servers
- Ensures high availability and reliability by sending requests only to servers that are online
- Provides the flexibility to add or subtract servers as demand dictates

For the purpose of choosing real-time least loaded path, load balancer immediately calculate the integrated load condition of multiple path and as well as server when receiving the path information transmitted from SDN controller.

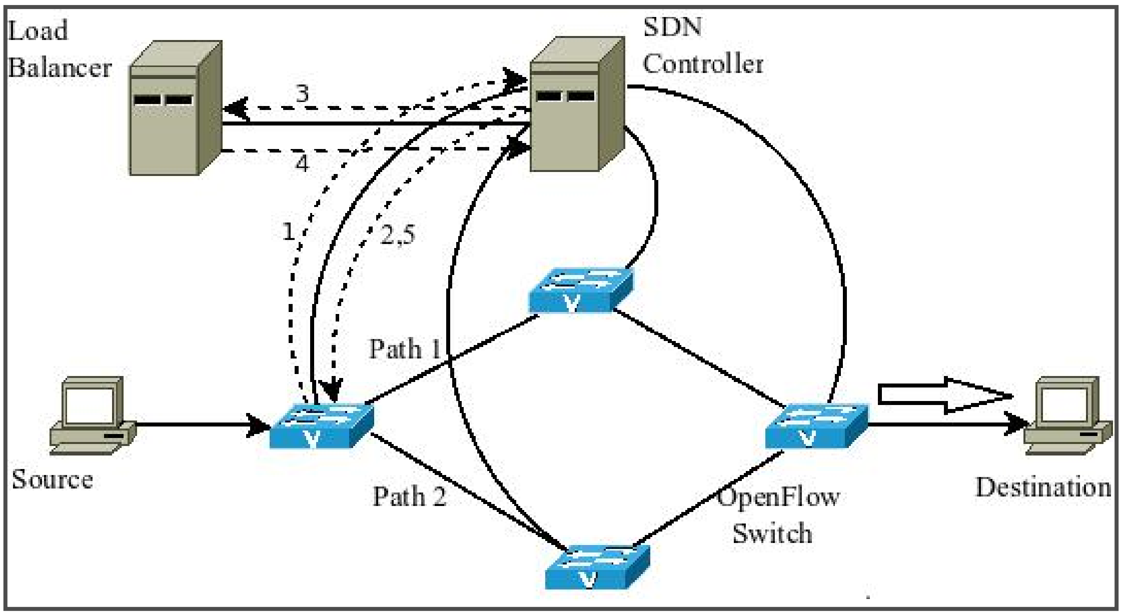


Figure 2: Load Balancer

CHAPTER 4: METHODOLOGY

4.1 System Design

SDN controller obtains the ability to show the global view of network at the beginning of network construction. By updating topology information of global network, SDN controller can discover all paths between each source node to each destination node. Network architecture for SDN load balance is shown in Figure 3. The architecture employs a dedicated load balance in each path. Hence, in the design, controller periodically transmits the load information of each path to load balancer as well as the load information of each server to the load balancer. And when the SDN controller need to process the load balance function, the load balancer return a least loaded path back to controller according to the calculated load condition of each path. After SDN controller receives the chosen path for transmission, it will allocate flow-Tables for OpenFlow switches to achieve the plan of data-flow transmission.

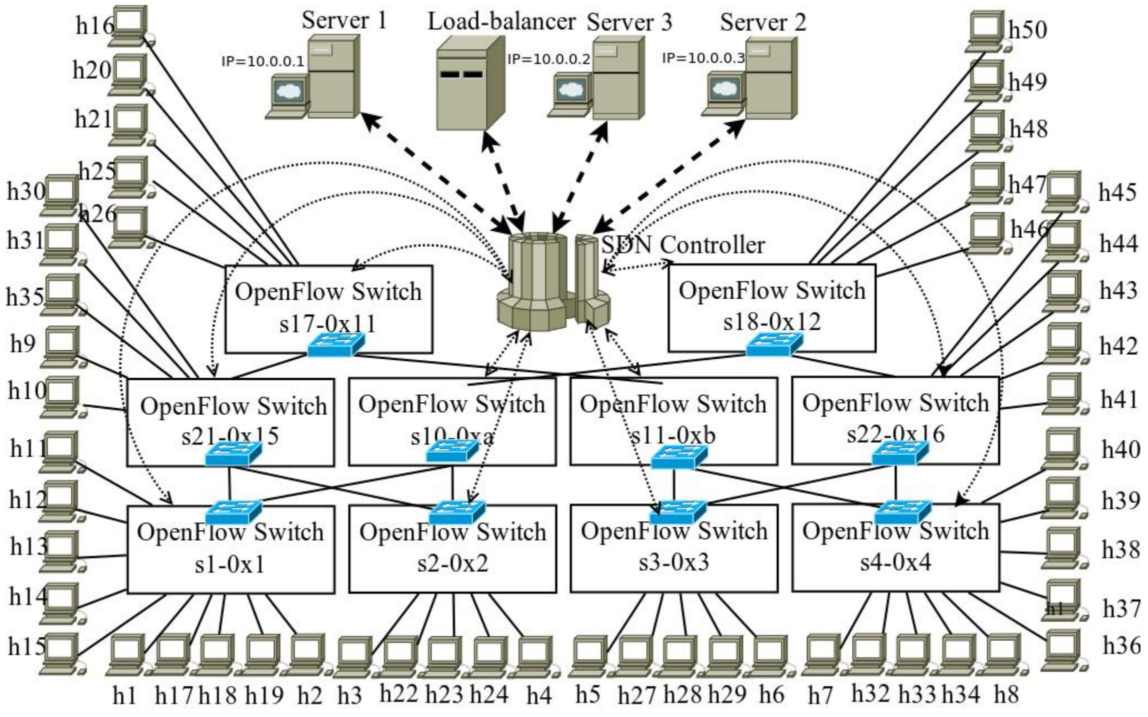


Figure 3: Network Architecture

In dynamic load balancing, the work load is calculated and distributed among the servers at runtime. The controller assigns new requests to the servers based on the load information collected. As shown in Figure 3, a set of clients and servers are connected to a network. The controller connected to the network communicates via the OpenFlow protocol and has a set of defined load balancing algorithms.

4.2 Algorithm

The block diagram of dynamic load balancing in SDN is shown in Figure 4. The aim of the algorithm is to balance dynamically the loads depending on traffic conditions in order to achieve the best resource profit possible. In pursuance of such goal, is essential to keep track of the current state of the network in terms of traffic. In pursuance of such goal, is essential to keep track of the current state of the network in terms of traffic.

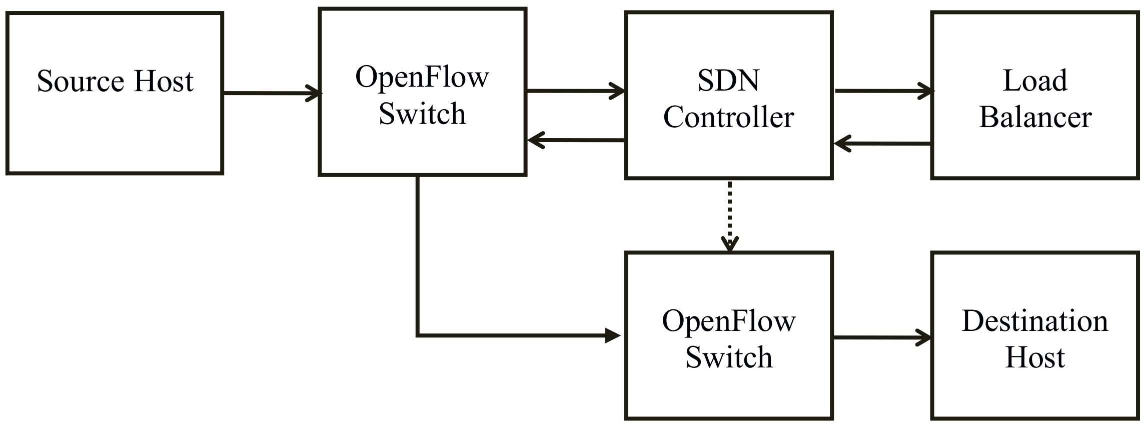


Figure 4: Block diagram of Dynamic Load Balancing in SDN

The procedure of control data in proposed dynamic load balancing in SDN system is as :

1. When a new data-flow transmitted into SDN domain, OpenFlow switches process the matching between packet head information and flow-Tables. If the flow-Table matches the packet head information, this data-flow will be transmitted by the Action field in flow-Table. And if there is no flow-Table to match this packet, OpenFlow switches will transmit this packets head information to SDN controller to decide the transmission path.
2. When finding only one path for data transmission, the SDN controller will create new flow-Tables and allocate them to OpenFlow switches to active data transmission.
3. When finding multiple paths for data transmission, the SDN controller will transmit multiple path load information to the load balancer.
4. The load balancer calculates the integrated load for every path and chooses one least loaded path as the result to return back to SDN controller.
5. SDN Controller receives the chosen path from load balancer and creates

flow-Tables to allocate to OpenFlow switches.

4.2.1 New flow detection mechanism

The first step is detecting a new flow event. This is done when a packet arrives to a switch in the network, and the header of that packet doesn't matches with any of the rules that the switch has, which will trigger the load balancer.

The next step is to find if all the possible paths between the to points have been computed already, thus avoid to recalculate the possible paths in case that have been computed already. This is useful due the possibility that different flows can have the same ingress and egress switches (e.g flows from the same source host to the same destination but with different ports, or hosts attached to the same switch). In the case that the paths between the ingress and egress switches have not been computed yet, a function to calculated it is triggered. The way to find all the possible paths is by using a Dijkstra algorithm. For each new path discovered, a new Path is stored into the PathCollection, having as a unique identifier a string with all the nodes which the path goes through.

Once all the possible paths are found, it is needed to, first, select a route, and second, write the rules in the flow Tables of each switch within the selected route. To select the route, the algorithm looks for the Path with bigger FreeCapacity value. Since when the flow starts there is no information about the traffic that it will bear and this way we guarantee that the flow will use the route where there is more capacity available.

After the route is selected, and using the Hops and Links of the specific Path, it is needed to send the appropriated messages to the switches to forward the packets through that path.

With that, every switch within the selected route will have the necessary flow entries to carry out the communication between the two end points.

The main objective of new flow detection mechanism is to find out whether there exists a flow rule added in the OpenFlow switch or not. If the new flow routing path does not exist, then all parallel paths between source and destination are searched and the best optimal path is selected for routing.

The block diagram of new flow detection mechanism in dynamic load balancing in software defined networking is shown in the Figure 5.

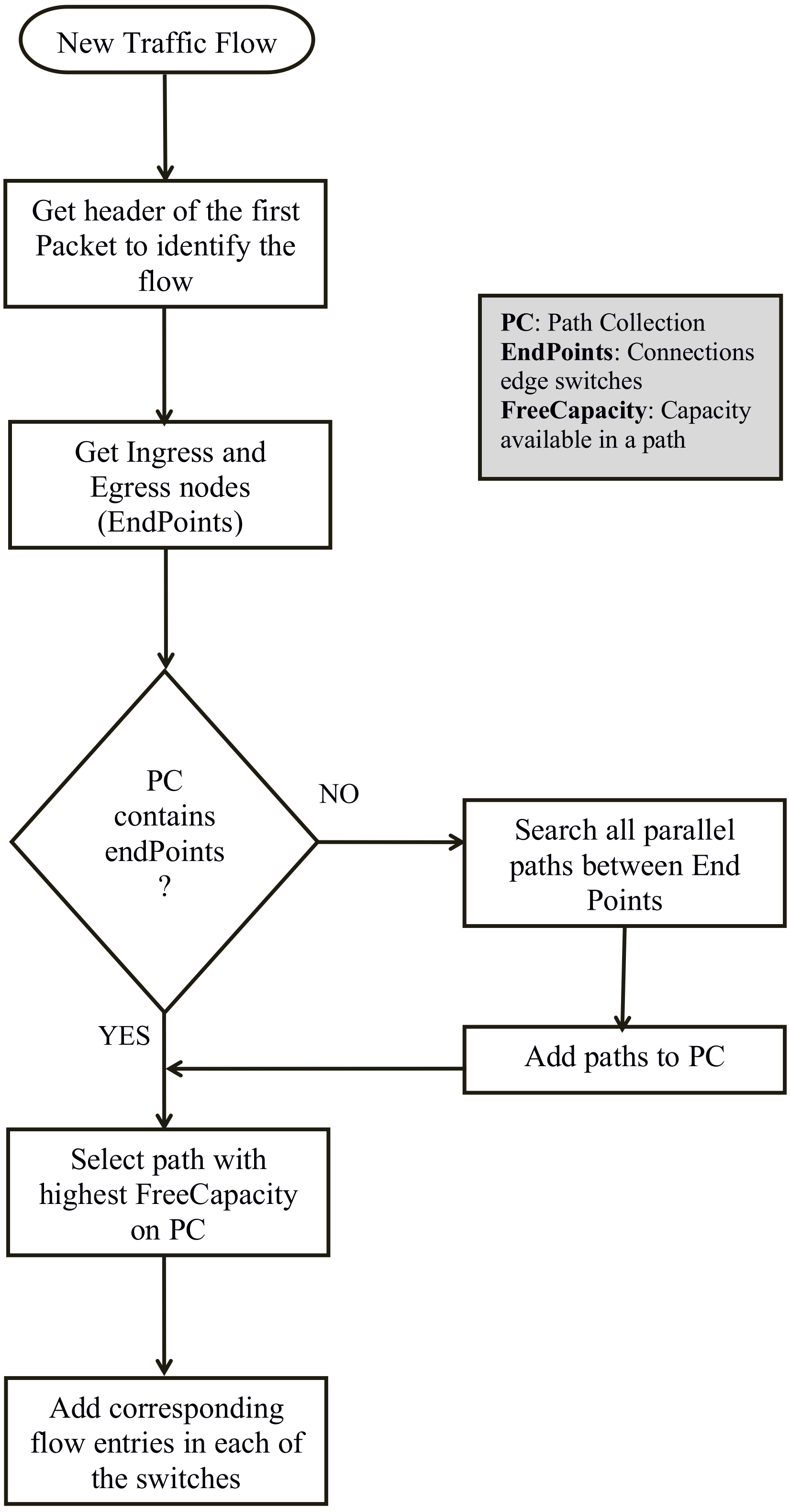

New Traffic Flow
Get header of the first Packet to identify the flow
Get Ingress and Egress nodes (EndPoints)
PC: Path Collection
EndPoints: Connections edge switches
FreeCapacity: Capacity available in a path
PC contains endPoints ?
NO
Search all parallel paths between End Points
YES
Add paths to PC
Select path with highest FreeCapacity on PC
Add corresponding flow entries in each of the switches

Figure 5 : New flow routing

After the addition of new flows entries into the path collection entries and in the openflow switches, then the controller needs to go for rerouting flows in order to balance the load in the network.

4.2.2 Rerouting flows mechanism

After the flow has initiated its communication between the two hosts, and starts to transmit traffic, the algorithm starts to capture information about the network state in order to adapt to the conditions of every single moment. Once congestion is detected at any link, appears the need of rerouting some of the current flows. To accomplish the best accommodation of resources possible, this algorithm is focused on reroute the flows with lowest traffic.

The process conducted is explained in two parts. The first one, illustrated in Figure 6, explains the first step, where the link which is overcrowded is detected and, afterwards, a FlowsCollection is established (LFC), containing all the flows that are using that link. Subsequently it selects the flow with lowest bandwidth usage (assigning it to the variable PendingFlow), and checks out if there is any other parallel route for this flow with enough free capacity to carry its traffic. In the case that there is another possible path with sufficient capacity, the flow will be routed through that route, sending the corresponding flow entries to each of the OpenFlow switches.

Once the flow with the lowest bandwidth usage have been routed across another path, the process starts again, and, in the case that the congestion still exists, the same procedure will be followed, moving the lightest flows along another routes.

After congestion is detected, no route with enough free capacity have been found to reroute any of the flows. In this case the different flows is analyzed to Figure out which is the best way to allocate them. Afterwards, the algorithm iterate for every flow and analyzing if the CB (Contributed Bandwidth) of the selected flow is higher than the bandwidth than the flow that the controller is trying to allocate. Inquiring the free resources from the controller, the available in the Path if the selected flow is moved along another route and see if that will be enough to carry the bandwidth the flow needs.

The main objectives of two mechanism new flow detection and rerouting the flow are to move the traffic flows with lower used bandwidth to the paths with lowest capacity. Thus, assuring that the heaviest flows are allocated to the highest-capacity paths. The algorithm takes into account the network topology character and traffic bandwidth request.

Rerouting flow mechanism is shown in the Figure 6. It starts with free capacity check , link congestion check and finally rerouting pending flow through the path.

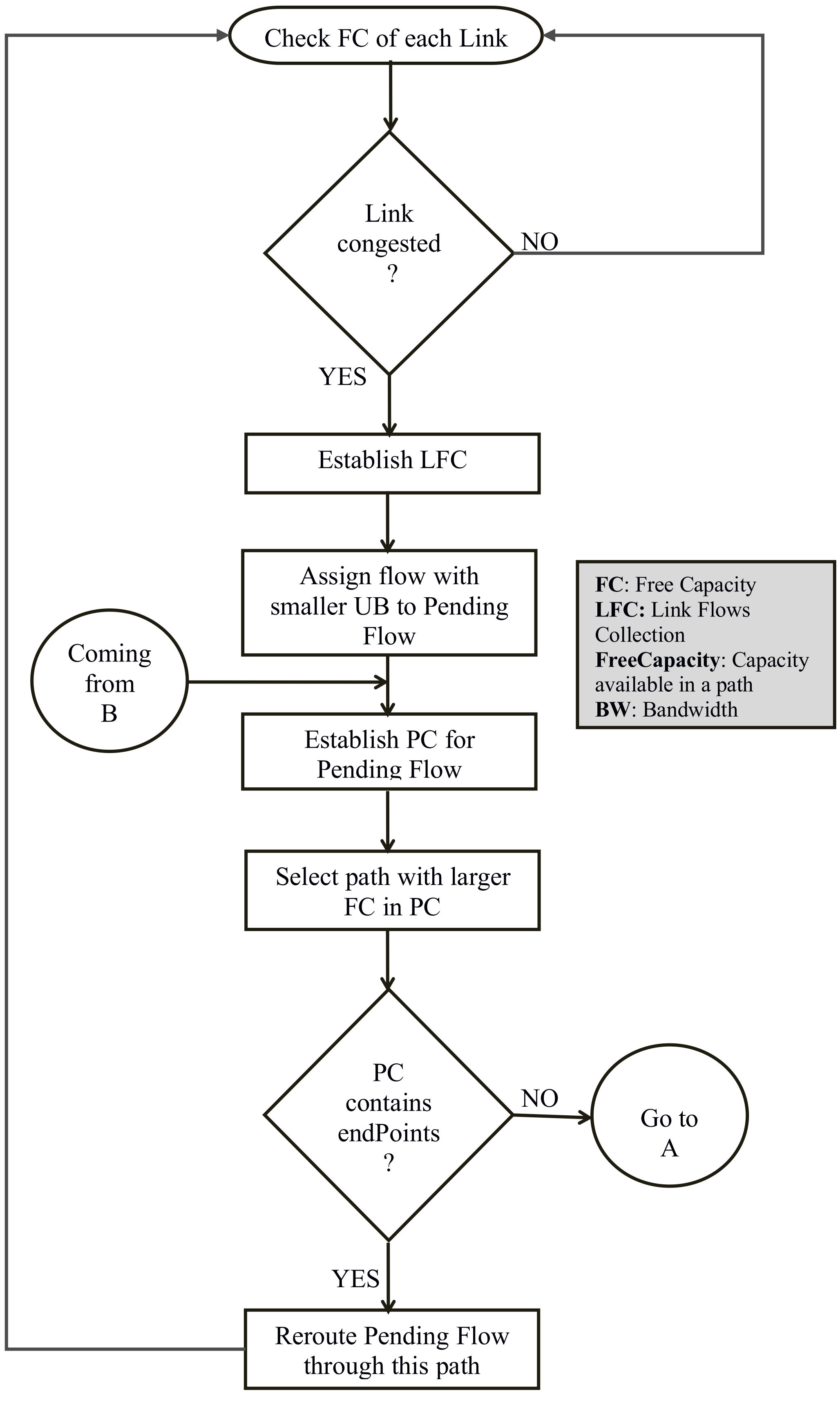

Check FC of each Link
Link congested ?
NO
YES
Establish LFC
Assign flow with smaller UB to Pending Flow
Coming from B
Establish PC for Pending Flow
Select path with larger FC in PC
PC contains endPoints ?
NO
Go to A
YES
Reroute Pending Flow through this path
FC: Free Capacity
LFC: Link Flows Collection
FreeCapacity: Capacity available in a path
BW: Bandwidth

Figure 6: Rerouting flow

With the rerouting flow mechanism employed in the SDN controller, QoS is achieved.

4.3 Tools

Different tools and languages to be used for completion of this thesis are discussed in this section.

4.3.1 Mininet

Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM, cloud or native), in seconds, with a single command. Programmer can easily interact with your network using the Mininet CLI (and API), customize it, share it with others, or deploy it on real hardware. Mininet is useful for development, teaching, and research. Mininet is also a great way to develop, share, and experiment with OpenFlow and Software-Defined Networking systems.

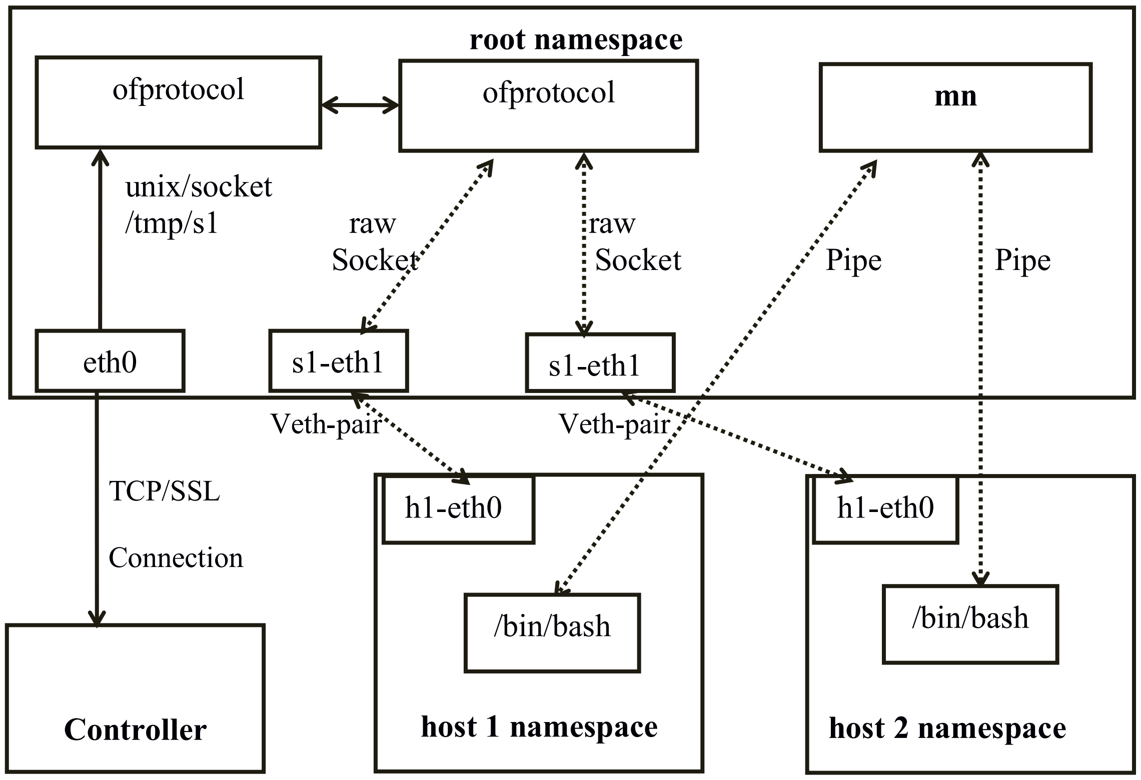


Figure 7 : Mininet

4.4.2 SDN Controller

An SDN controller is an application in SDN that manages flow control to enable intelligent networking. SDN controllers are based on protocols, such as OpenFlow, that allow servers to tell switches where to send packets.

The controller is the core of an SDN network. It lies between network devices at one end and applications at the other end. Any communications between applications and devices have to go through the controller. The controller also uses protocols such as OpenFlow to cofigure network devices and choose the optimal network path.

4.3.3 OpenFlow Switch

OpenFlow provides a mechanism for SDN. When a packet from a client arrives at an OpenFlow switch, packet header information is compared with flow Table entries of the switch. Each flow entry consists of a set of flow rules, defined on basis of packet header fields for packet matching, an action to be performed on the packets matching the flow rules, & flow statistics. A packet header includes Port id, VLAN tag, Ethernet type, source & destination address, IP protocol type, User Datagram Protocol/Transmission Control Protocol (UDP/TCP) source & destination port.

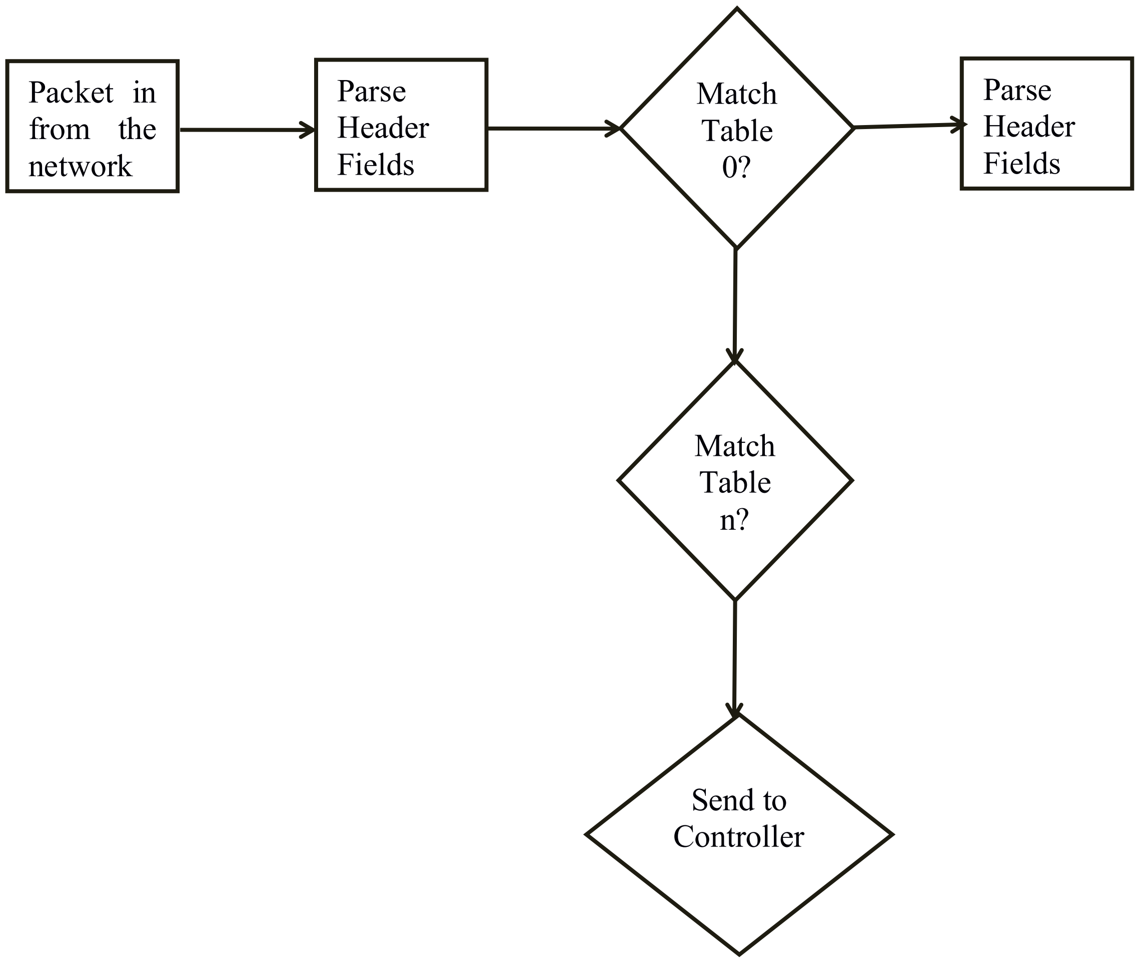


Figure 8 : Matching a flow in OpenFlow

Open vSwitch uses different kinds of flows for different purposes. OpenFlow Controller uses OpenFlow flows to define a switch’s policy. In a conventional switch, packet forwarding (data plane) and high level routing (control plane) occur on the same device. In SDN, the data plane is decoupled from the control plane.

4.3.4 Wireshark

Wireshark is a Free and open source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development, and education. Wireshark lets the user put network interface controllers that support promiscuous mode into that mode, so they can see all traffic visible on that interface, not just traffic addressed to one of the interface's configured addresses and broadcast/multicast traffic. However, when capturing with a packet analyzer in promiscuous mode on a port on a network switch, not all traffic through the switch is necessarily sent to the port where the capture is done, so capturing in promiscuous mode is not necessarily sufficient to see all network traffic. Port mirroring or various network taps extend capture to any point on the network[19].

4.3.5 IPERF

IPERF is a commonly used network testing tool that can create Transmission Control protocol (TCP) and User Datagram Protocol (UDP) data streams and measure the throughput of a network that is carrying them. Iperf allows the user to set various parameters that can be used for testing a network, or alternatively for optimizing or tuning a network. Iperf has a client and server functionality, and can measure the throughput between the two ends [20].

4.3.6 Java & Python Programming Language

Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture[21].

Python is a widely used high-level, general-purpose, interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than possible in languages such as C++ or Java. Python supports multiple programming paradigms, including object-oriented. For any experienced programmer in any programming language, python can be easy to use and learn[22].

4.3.7 Oracle Virtual Box

VirtualBox is a cross-platform virtualization application. It extends the capabilities of existing computer so that it can run multiple operating systems (inside multiple virtual machines) at the same time. It allows to run more than one operating system at a time. Virtual machine (VM) is the special environment that VirtualBox creates for guest operating system while it is running. The key features of oracle virtual box are portability, no hardware virtualization required and guest additions.

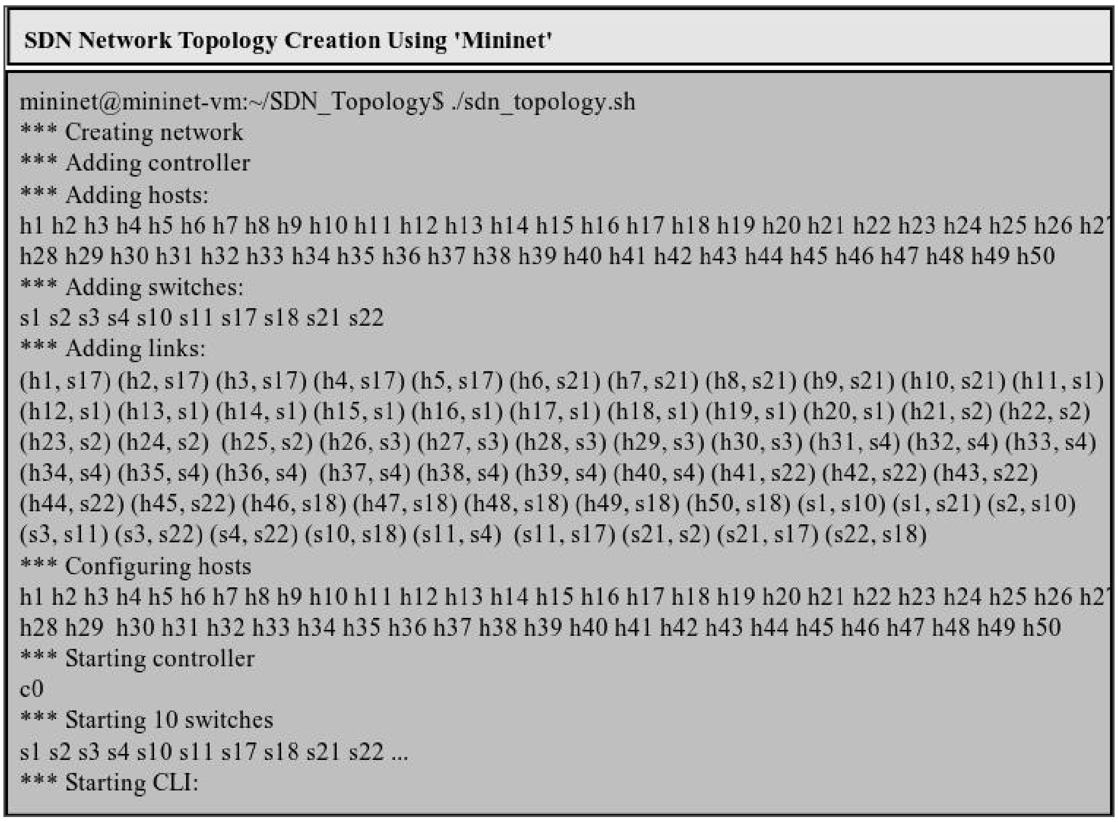
4.3.8 Gnome Connection Manager

Gnome connection manager (GCM) is very useful when connecting multiple remote machine over ssh. It is a GUI for remote management. Remote activity is something that usually done by a system administrator. The remote protocol that might be used is SSH and/or telnet. It is a tabbed SSH Connection for gtk+ environment. With this, remote screens can be manage to make the administrator easier to operate them. It can store passwords for easy access to hosts. It supports multiple ssh tunnels for each host. It can connect to multiple hosts with just one click. It’s free and the source is included in the download.

CHAPTER 5: EXPERIMENTS AND OUTPUTS

5.1 SDN Network Topology setup

The SDN network topology is created using mininet. Eight hosts are created with ten OpenFlow switches. The remote floodlight controller [IP=192.168.124.5, port=6653] is connected to the SDN network topology. The Figure 9 shows SDN network topology creation using mininet.



SDN Network Topology Creation Using 'Mininet'

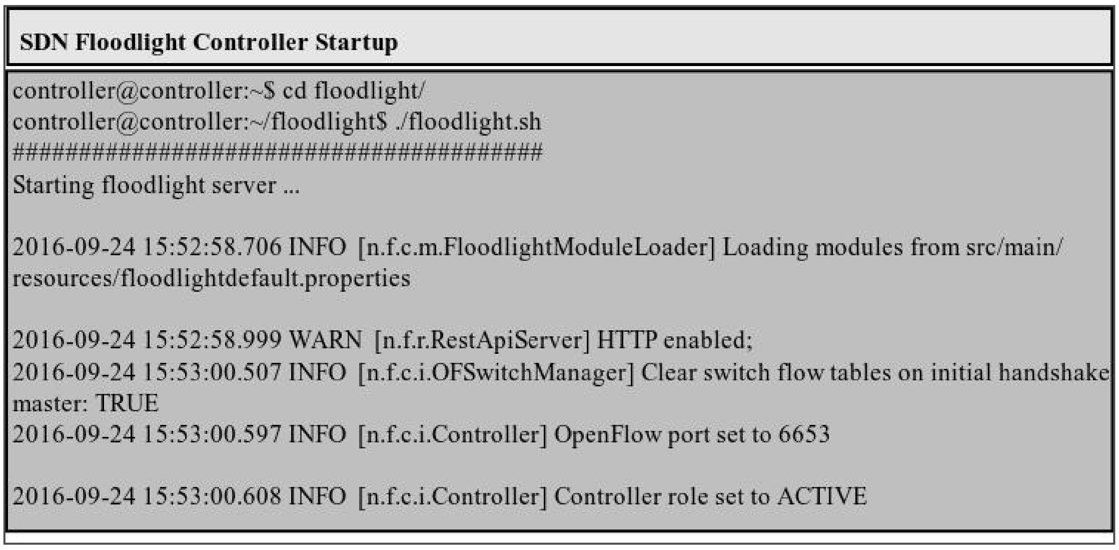
```
mininet@mininet-vm:~/SDN_Topology$ ./sdn_topology.sh
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25 h26 h2
h28 h29 h30 h31 h32 h33 h34 h35 h36 h37 h38 h39 h40 h41 h42 h43 h44 h45 h46 h47 h48 h49 h50
*** Adding switches:
s1 s2 s3 s4 s10 s11 s17 s18 s21 s22
*** Adding links:
(h1, s17) (h2, s17) (h3, s17) (h4, s17) (h5, s17) (h6, s21) (h7, s21) (h8, s21) (h9, s21) (h10, s21) (h11, s1)
(h12, s1) (h13, s1) (h14, s1) (h15, s1) (h16, s1) (h17, s1) (h18, s1) (h19, s1) (h20, s1) (h21, s2) (h22, s2)
(h23, s2) (h24, s2)  (h25, s2) (h26, s3) (h27, s3) (h28, s3) (h29, s3) (h30, s3) (h31, s4) (h32, s4) (h33, s4)
(h34, s4) (h35, s4) (h36, s4)  (h37, s4) (h38, s4) (h39, s4) (h40, s4) (h41, s22) (h42, s22) (h43, s22)
(h44, s22) (h45, s22) (h46, s18) (h47, s18) (h48, s18) (h49, s18) (h50, s18) (s1, s10) (s1, s21) (s2, s10)
(s3, s11) (s3, s22) (s4, s22) (s10, s18) (s11, s4)  (s11, s17) (s21, s2) (s21, s17) (s22, s18)
*** 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 h2
h28 h29  h30 h31 h32 h33 h34 h35 h36 h37 h38 h39 h40 h41 h42 h43 h44 h45 h46 h47 h48 h49 h50
*** Starting controller
c0
*** Starting 10 switches
s1 s2 s3 s4 s10 s11 s17 s18 s21 s22 ...
*** Starting CLI:
```

Figure 9 : SDN network topology creation using mininet

5.2 SDN Controller setup

Floodlight is an OpenFlow controller altogether with a collection of applications built on its top. The controller by itself contains a bunch of interfaces which keep track of the state of the network. It also provides a set of services which can be used from the extensible modules. The controller core has a Java Application Programming Interface (API) to expose the controller functionalities, consenting the expansion of the controller functionalities. Thus, the network manager can adapt the behavior of the network by writing a custom module. Furthermore the Java API, Floodlight also provides a REST API, therefore any REST applications, written in any language, can retrieve information and invoke services by sending http REST commands to the controller REST port.

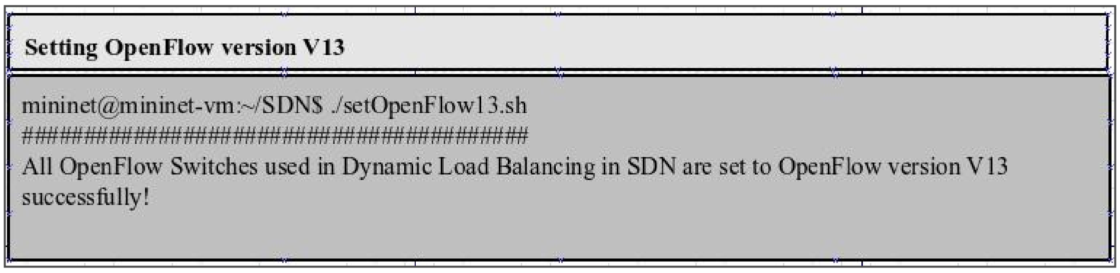
Once the floodlight controller is started, HTTP will be enabled. All flow Tables on initial handmaster will be cleared. OpenFlow port to 6653 is set. The controller role is set to ACTIVE.



```
SDN Floodlight Controller Startup

controller@controller:~$ cd floodlight/
controller@controller:~/floodlight$ ./floodlight.sh
##########################################
Starting floodlight server ...

2016-09-24 15:52:58.706 INFO  [n.f.c.m.FloodlightModuleLoader] Loading modules from src/main/
resources/floodlightdefault.properties

2016-09-24 15:52:58.999 WARN  [n.f.r.RestApiServer] HTTP enabled;
2016-09-24 15:53:00.507 INFO  [n.f.c.i.OFSwitchManager] Clear switch flow tables on initial handshake
master: TRUE
2016-09-24 15:53:00.597 INFO  [n.f.c.i.Controller] OpenFlow port set to 6653

2016-09-24 15:53:00.608 INFO  [n.f.c.i.Controller] Controller role set to ACTIVE
```

Figure 10 : SDN Floodlight controller startup

After the SDN floodlight controller and SDN topology are created, then SDN switches need to be set with OpenFlow version V13 in order to get full operability. The following Figure shows the successful setting of OpenFlow version V13 all SDN switches after calling the script by ./setOpenFlow13.sh.



```
Setting OpenFlow version V13

mininet@mininet-vm:~/SDN$ ./setOpenFlow13.sh
##########################################
All OpenFlow Switches used in Dynamic Load Balancing in SDN are set to OpenFlow version V13
successfully!
```

Figure 11 :   Setting OpenFlow version V13

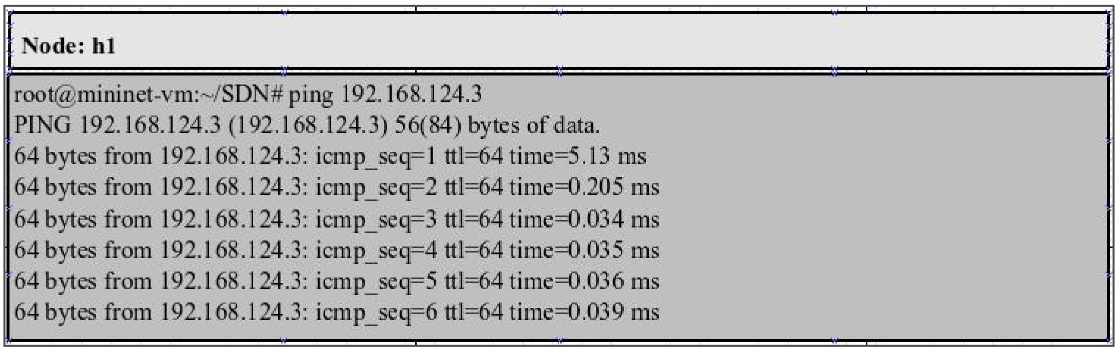
The calling of user defined script setOpenFlow13.sh will install protocols=OpenFlow13 in all SDN switches that are created by mininet emulator.

5.3 Path selection before Load Balancing

A load balancer selects the appropriate least cost effective link between the source and destination. If the network is without a load balancer, then the link utilization factor of that network is poor.

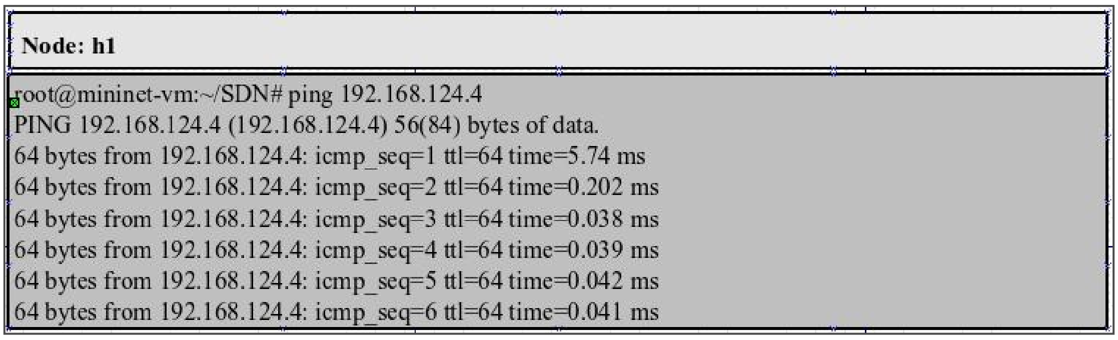
Before the implementation of load balancer in the customized SDN network topology, two things are observed. The first thing is, bandwidth utilization of link between source host and destination host. The second thing is latency of packets to reach destination.

From the first console of host h1, host h3 (192.168.124.3) is made. Also from the second console of host h1, host h4 192.168.124.4 is pinged. They are shown in the Figure 12 and Figure 13 respectively.



```
Node: h1
root@mininet-vm:~/SDN# ping 192.168.124.3
PING 192.168.124.3 (192.168.124.3) 56(84) bytes of data.
64 bytes from 192.168.124.3: icmp_seq=1 ttl=64 time=5.13 ms
64 bytes from 192.168.124.3: icmp_seq=2 ttl=64 time=0.205 ms
64 bytes from 192.168.124.3: icmp_seq=3 ttl=64 time=0.034 ms
64 bytes from 192.168.124.3: icmp_seq=4 ttl=64 time=0.035 ms
64 bytes from 192.168.124.3: icmp_seq=5 ttl=64 time=0.036 ms
64 bytes from 192.168.124.3: icmp_seq=6 ttl=64 time=0.039 ms
```

Figure 12 : h1 ping h3



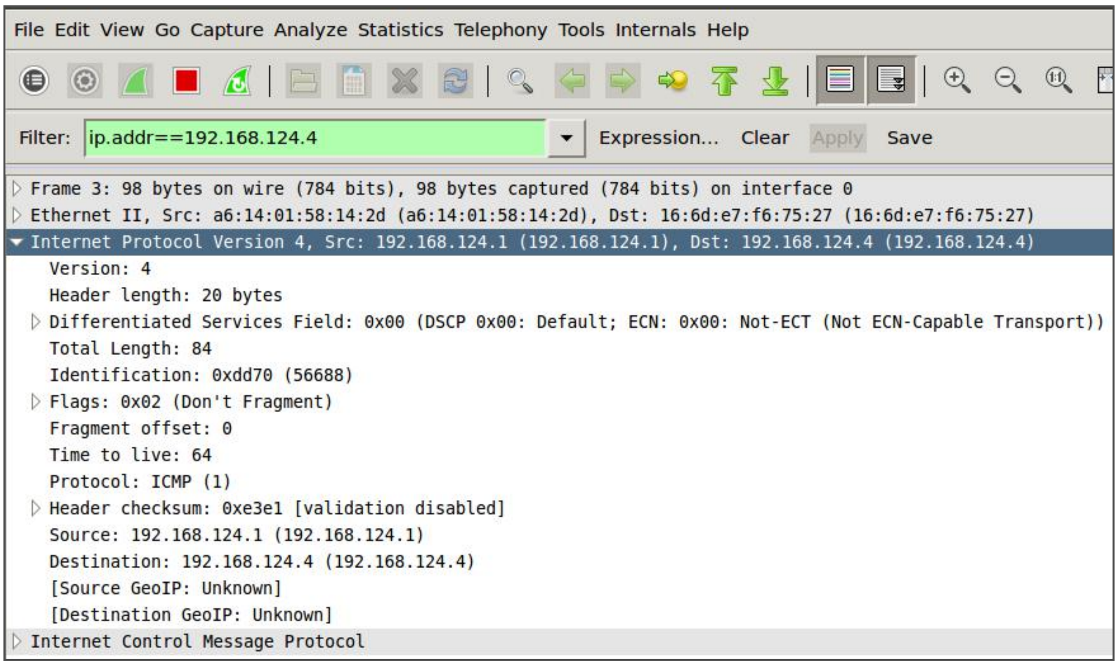
```
Node: h1
root@mininet-vm:~/SDN# ping 192.168.124.4
PING 192.168.124.4 (192.168.124.4) 56(84) bytes of data.
64 bytes from 192.168.124.4: icmp_seq=1 ttl=64 time=5.74 ms
64 bytes from 192.168.124.4: icmp_seq=2 ttl=64 time=0.202 ms
64 bytes from 192.168.124.4: icmp_seq=3 ttl=64 time=0.038 ms
64 bytes from 192.168.124.4: icmp_seq=4 ttl=64 time=0.039 ms
64 bytes from 192.168.124.4: icmp_seq=5 ttl=64 time=0.042 ms
64 bytes from 192.168.124.4: icmp_seq=6 ttl=64 time=0.041 ms
```

Figure 13 : h1 ping h4

In the custom topology, the path from h1 to h3 and the path from h1 to h4 can be same . The paths may be s1-s21-s2 or s1-s10-s2. When h1 first pings h3, the path s1-s10-s2 is selected. After h1 pinging h3, h1 to ping h4 is made. Again, the same path s1-s10-s2 is chosen even though s1-s10-s2 is unused. This situation can be observed from the wire-shark. The process to analyze the packets in wire-shark is as follows.

- Go to capture
- Select Interfaces
- Select “s1-eth4”
- Start the capture

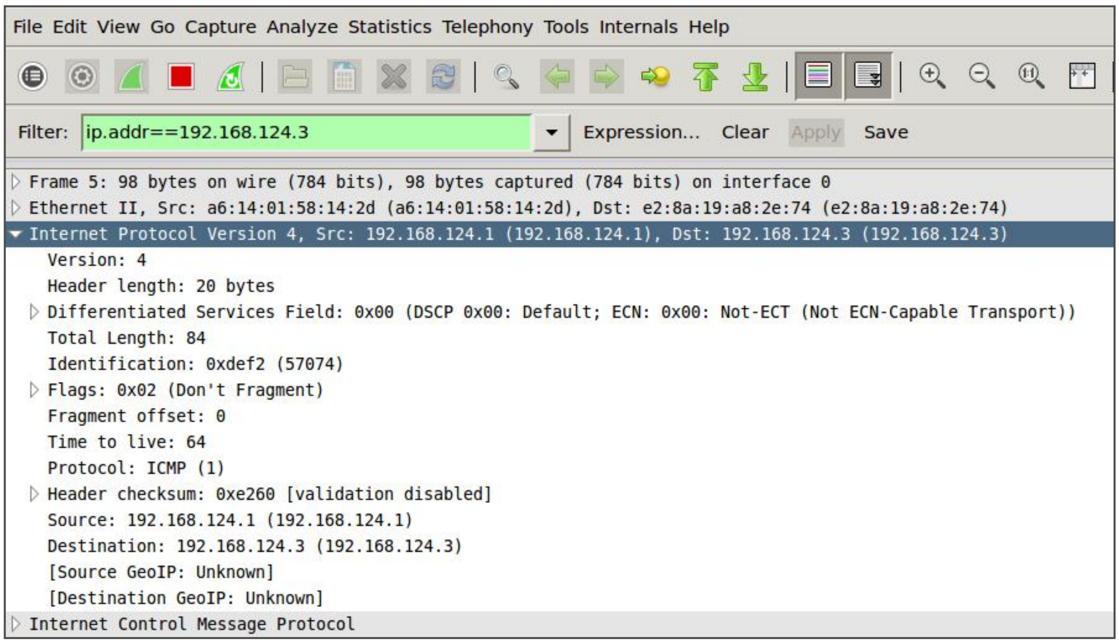
In the filter section of wireshark, ip.addr=192.168.124.4 packets are filtered. The packets of 192.168.124.4 can be observed in the interface s1-eth4.


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: ip.addr==192.168.124.4
Expression... Clear Apply Save

```
▷ Frame 3: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▷ Ethernet II, Src: a6:14:01:58:14:2d (a6:14:01:58:14:2d), Dst: 16:6d:e7:f6:75:27 (16:6d:e7:f6:75:27)
▽ Internet Protocol Version 4, Src: 192.168.124.1 (192.168.124.1), Dst: 192.168.124.4 (192.168.124.4)
    Version: 4
    Header length: 20 bytes
  ▷ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
    Total Length: 84
    Identification: 0xdd70 (56688)
  ▷ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
  ▷ Header checksum: 0xe3e1 [validation disabled]
    Source: 192.168.124.1 (192.168.124.1)
    Destination: 192.168.124.4 (192.168.124.4)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▷ Internet Control Message Protocol
```

Figure 14 : Wire-shark capture in interface s1-eth4 for packets of 192.168.124.4

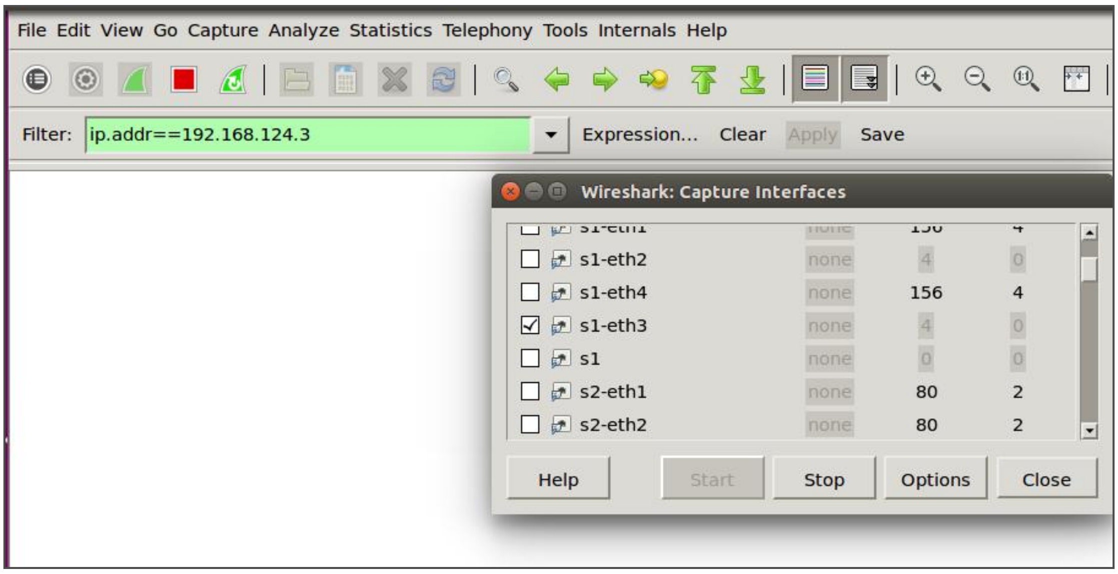
Again, same process is done to check the packets of 192.168.124.3 in the interface s1-eth4. Also the packets of 192.168.124.3 can be observed in the same interface s1-eth4. This is shown in the Figure 15. But there is no ping packets of 192.168.124.3 and 192.168.124.4 in the interface 192.168.124.3. This means that unused path is not properly utilized.


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: ip.addr==192.168.124.3
Expression... Clear Apply Save

```
▷ Frame 5: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▷ Ethernet II, Src: a6:14:01:58:14:2d (a6:14:01:58:14:2d), Dst: e2:8a:19:a8:2e:74 (e2:8a:19:a8:2e:74)
▽ Internet Protocol Version 4, Src: 192.168.124.1 (192.168.124.1), Dst: 192.168.124.3 (192.168.124.3)
    Version: 4
    Header length: 20 bytes
  ▷ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
    Total Length: 84
    Identification: 0xdef2 (57074)
  ▷ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
  ▷ Header checksum: 0xe260 [validation disabled]
    Source: 192.168.124.1 (192.168.124.1)
    Destination: 192.168.124.3 (192.168.124.3)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▷ Internet Control Message Protocol
```

Figure 15 : Wireshark capture in interface s1-eth4 for packets of 192.168.124.3

From Figure 15, it is observed that ping packets for node h3 [ip=192.168.124.3] is not observed in the interface s1-eth3 before load balancing.

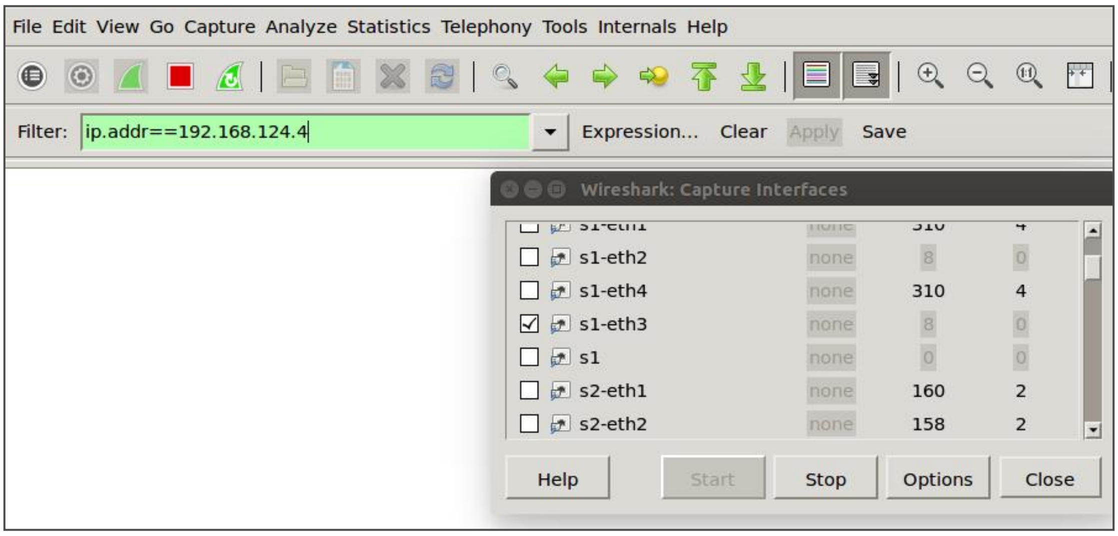

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help
Start
Stop
Options
Close

Filter: ip.addr==192.168.124.3

☐	s1-eth2	none	4	0
☐	s1-eth4	none	156	4
☑	s1-eth3	none	4	0
☐	s1	none	0	0
☐	s2-eth1	none	80	2
☐	s2-eth2	none	80	2

Figure 16 : Wireshark capture in interface s1-eth3 for packets of 192.168.124.3

From Figure 16, it is also observed that ping packets for node h4 [ip=192.168.124.4] is not observed in the interface s1-eth3 before load balancing. This means that, interface s1-eth3 is not used even it is free before the implementation of load balancer in software defined networking.

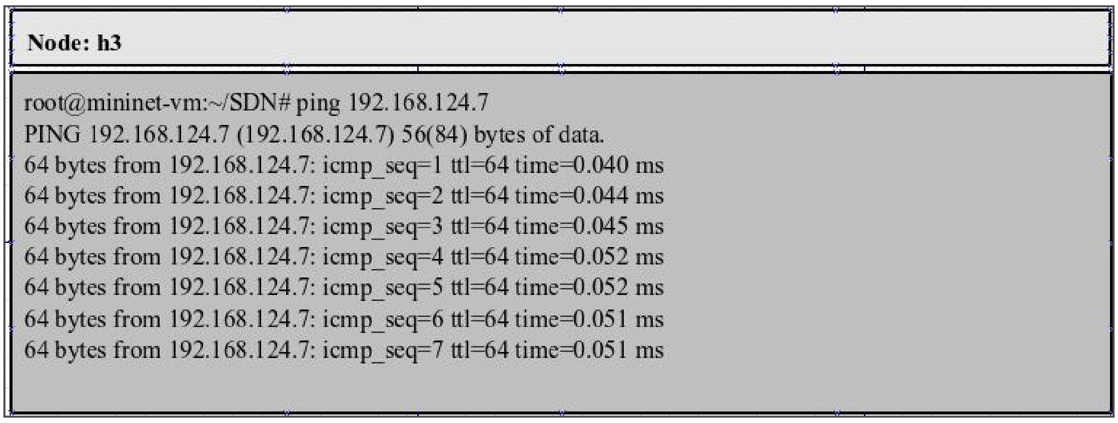

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help
Start
Stop
Options
Close

Filter: ip.addr==192.168.124.4

☐	s1-eth2	none	8	0
☐	s1-eth4	none	310	4
☑	s1-eth3	none	8	0
☐	s1	none	0	0
☐	s2-eth1	none	160	2
☐	s2-eth2	none	158	2

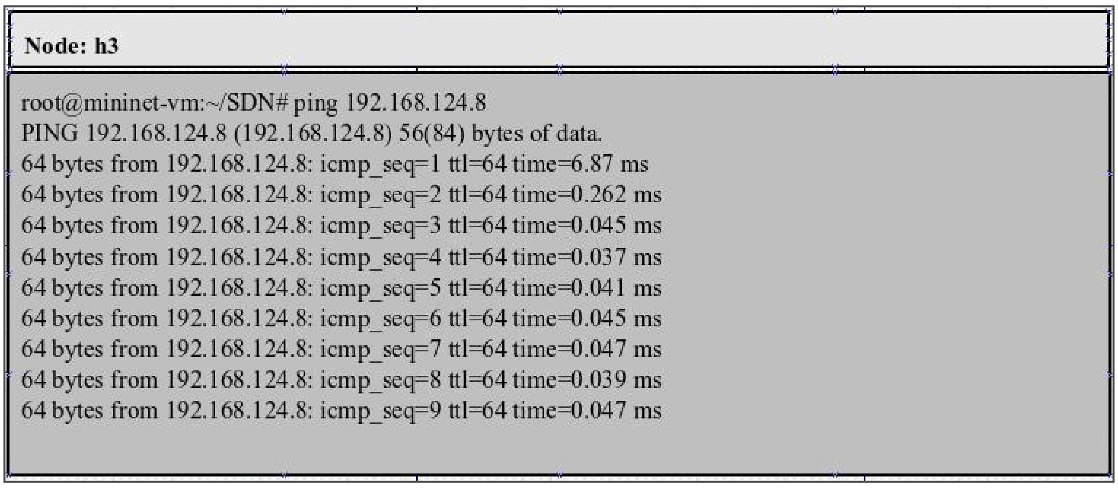
Figure 17 : Wireshark capture in interface s1-eth3 for packets of 192.168.124.4

Now, another situation that is considered for observation is that from the first console of host h3, ping 192.168.124.7 (host h7) is done. In addition, from the second console of host h3, ping 192.168.124.8 is made. They are shown in the Figure 18 and Figure 19 respectively.



```
Node: h3

root@mininet-vm:~/SDN# ping 192.168.124.7
PING 192.168.124.7 (192.168.124.7) 56(84) bytes of data.
64 bytes from 192.168.124.7: icmp_seq=1 ttl=64 time=0.040 ms
64 bytes from 192.168.124.7: icmp_seq=2 ttl=64 time=0.044 ms
64 bytes from 192.168.124.7: icmp_seq=3 ttl=64 time=0.045 ms
64 bytes from 192.168.124.7: icmp_seq=4 ttl=64 time=0.052 ms
64 bytes from 192.168.124.7: icmp_seq=5 ttl=64 time=0.052 ms
64 bytes from 192.168.124.7: icmp_seq=6 ttl=64 time=0.051 ms
64 bytes from 192.168.124.7: icmp_seq=7 ttl=64 time=0.051 ms
```

Figure 18 : h3 ping h7

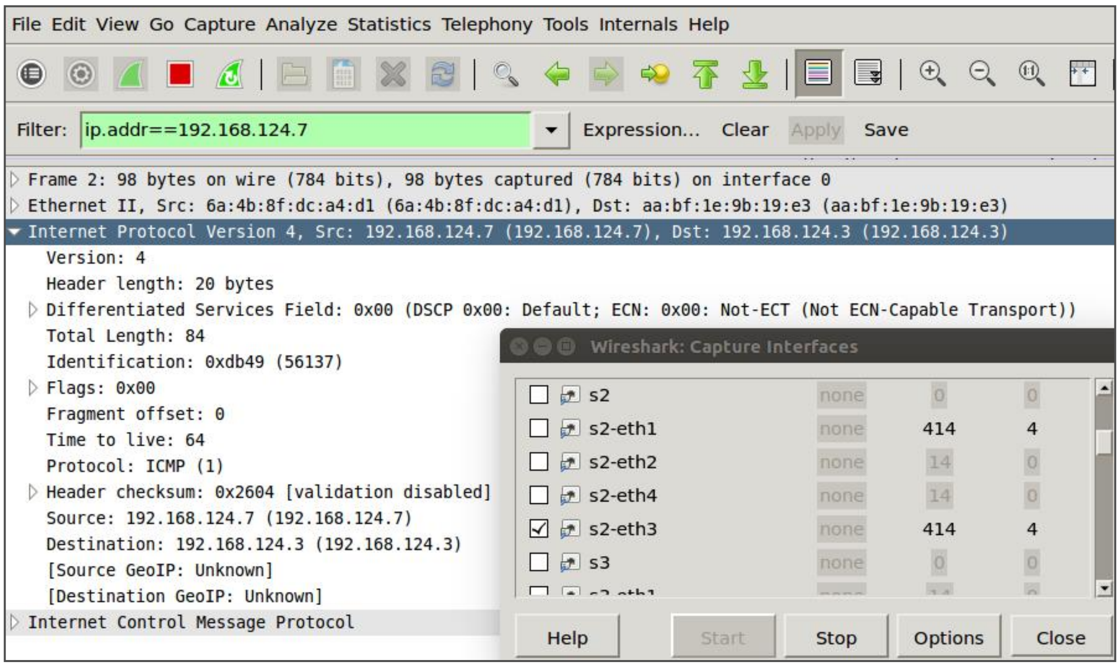


```
Node: h3

root@mininet-vm:~/SDN# ping 192.168.124.8
PING 192.168.124.8 (192.168.124.8) 56(84) bytes of data.
64 bytes from 192.168.124.8: icmp_seq=1 ttl=64 time=6.87 ms
64 bytes from 192.168.124.8: icmp_seq=2 ttl=64 time=0.262 ms
64 bytes from 192.168.124.8: icmp_seq=3 ttl=64 time=0.045 ms
64 bytes from 192.168.124.8: icmp_seq=4 ttl=64 time=0.037 ms
64 bytes from 192.168.124.8: icmp_seq=5 ttl=64 time=0.041 ms
64 bytes from 192.168.124.8: icmp_seq=6 ttl=64 time=0.045 ms
64 bytes from 192.168.124.8: icmp_seq=7 ttl=64 time=0.047 ms
64 bytes from 192.168.124.8: icmp_seq=8 ttl=64 time=0.039 ms
64 bytes from 192.168.124.8: icmp_seq=9 ttl=64 time=0.047 ms
```

Figure 19 : h3 ping h8

In the custom topology, the path from h3 to h7 and the path from h3 to h8 can be same. The Figure 18 and the Figure 19 show pinging of node h7 (ip address 192.168.124.7) and node h8 (ip address 192.168.124.8) respectively from the node h3 (ip address 192.168.124.3) in the custom topology. The path may be s2-s10-ss18-s22-s4 or s2-s21-s17-s11-s4. When h3 first pings h7, the path s2-s21-s17-s11-s4 is selected. After h3 pinging h7, h3 to ping h8 is made. Again, the same path s2-s21-s17-s11-s4 is chosen even though s2-s10-ss18-s22-s4 path does not bear any network traffic or traffic with less congestion. This condition is undesirable.

When node h3 pings nodes h7, the packets of 192.168.124.7 can be observed in the interface s2-eth3 before load balancing which is shown in the Figure 20. In other words, the wireshark capture filter of ip address 192.168.124.7 at the interface s2-eth3 can be observed. In the Internet Protocol Version 4 field of wireshark for ip.addr filter, current source and destination ip addresses are listed.

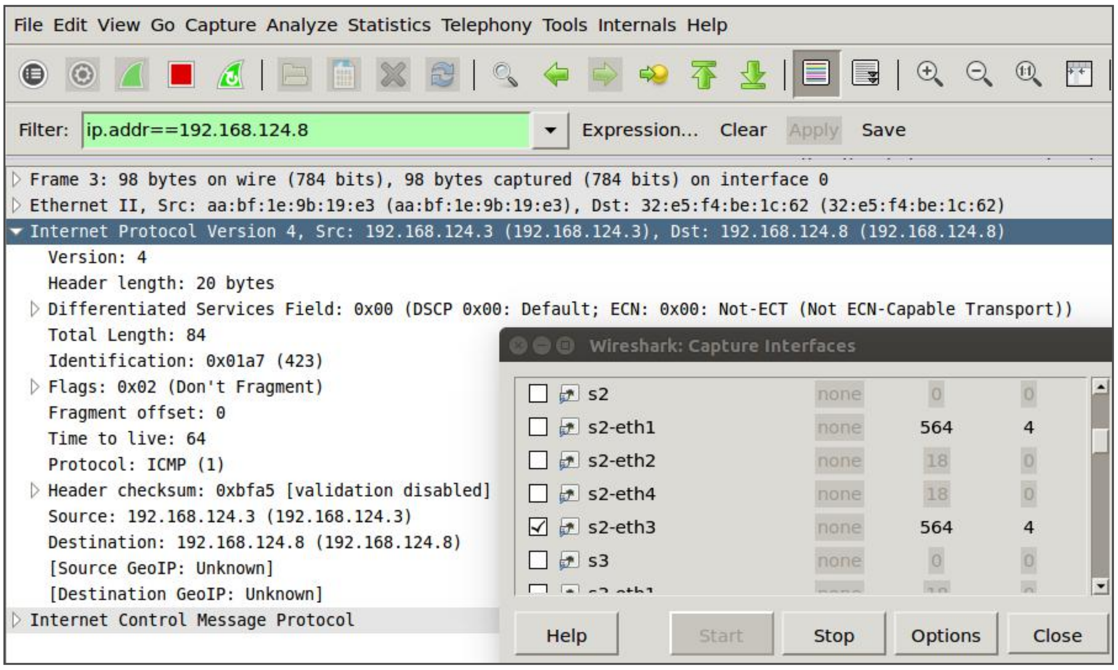

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: ip.addr==192.168.124.7
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help Start Stop Options Close

```
▷ Frame 2: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▷ Ethernet II, Src: 6a:4b:8f:dc:a4:d1 (6a:4b:8f:dc:a4:d1), Dst: aa:bf:1e:9b:19:e3 (aa:bf:1e:9b:19:e3)
▼ Internet Protocol Version 4, Src: 192.168.124.7 (192.168.124.7), Dst: 192.168.124.3 (192.168.124.3)
    Version: 4
    Header length: 20 bytes
  ▷ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
    Total Length: 84
    Identification: 0xdb49 (56137)
  ▷ Flags: 0x00
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
  ▷ Header checksum: 0x2604 [validation disabled]
    Source: 192.168.124.7 (192.168.124.7)
    Destination: 192.168.124.3 (192.168.124.3)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▷ Internet Control Message Protocol
```

☐	s2	none	0	0
☐	s2-eth1	none	414	4
☐	s2-eth2	none	14	0
☐	s2-eth4	none	14	0
✓	s2-eth3	none	414	4
☐	s3	none	0	0

Figure 20 : Wireshark capture in interface s2-eth3 for packets of 192.168.124.7

The packets of 192.168.124.8 can be observed in the interface s2-eth3 before load balancing when h3 pings h8 which is shown in the Figure 21.

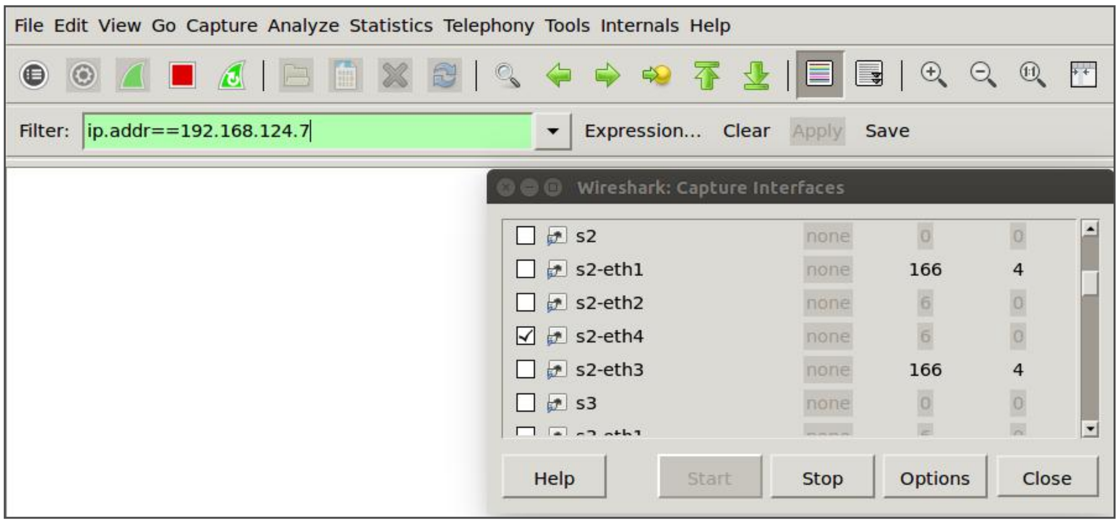

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: ip.addr==192.168.124.8
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help Start Stop Options Close

```
▷ Frame 3: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▷ Ethernet II, Src: aa:bf:1e:9b:19:e3 (aa:bf:1e:9b:19:e3), Dst: 32:e5:f4:be:1c:62 (32:e5:f4:be:1c:62)
▼ Internet Protocol Version 4, Src: 192.168.124.3 (192.168.124.3), Dst: 192.168.124.8 (192.168.124.8)
    Version: 4
    Header length: 20 bytes
  ▷ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
    Total Length: 84
    Identification: 0x01a7 (423)
  ▷ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
  ▷ Header checksum: 0xbfa5 [validation disabled]
    Source: 192.168.124.3 (192.168.124.3)
    Destination: 192.168.124.8 (192.168.124.8)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▷ Internet Control Message Protocol
```

☐	s2	none	0	0
☐	s2-eth1	none	564	4
☐	s2-eth2	none	18	0
☐	s2-eth4	none	18	0
✓	s2-eth3	none	564	4
☐	s3	none	0	0

Figure 21 : Wireshark capture in interface s2-eth3 for packets of 192.168.124.8

When node h3 pings node h7, the packets of 192.168.124.7 can not be observed in the interface s2-eth4 before load balancing which is shown in the Figure 22. In other words, the wireshark capture filter of ip address 192.168.124.7 at the interface s2-eth4 cannot be observed before load balancing in SDN.

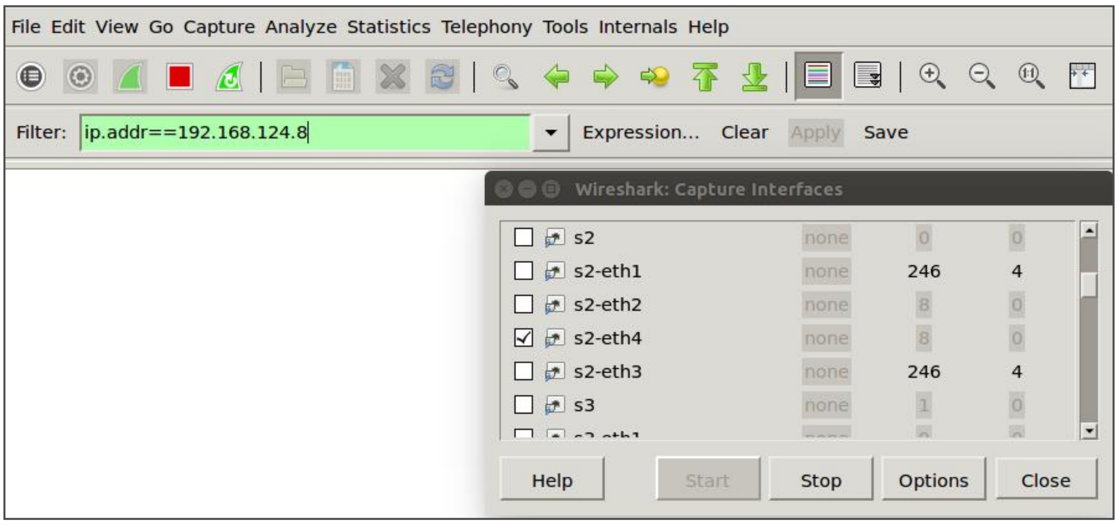

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help Start Stop Options Close

Filter: ip.addr==192.168.124.7

☐	s2	none	0	0
☐	s2-eth1	none	166	4
☐	s2-eth2	none	6	0
✓	s2-eth4	none	6	0
☐	s2-eth3	none	166	4
☐	s3	none	0	0

Figure 22 : Wireshark capture in interface s2-eth3 for packets of 192.168.124.8

The packets of 192.168.124.8 can not be observed in the interface s2-eth4 before load balancing when h3 pings h8 which is shown in the Figure 23.


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help Start Stop Options Close

Filter: ip.addr==192.168.124.8

☐	s2	none	0	0
☐	s2-eth1	none	246	4
☐	s2-eth2	none	8	0
✓	s2-eth4	none	8	0
☐	s2-eth3	none	246	4
☐	s3	none	1	0

Figure 23 : Wireshark capture in interface s2-eth4 for packets of 192.168.124.8

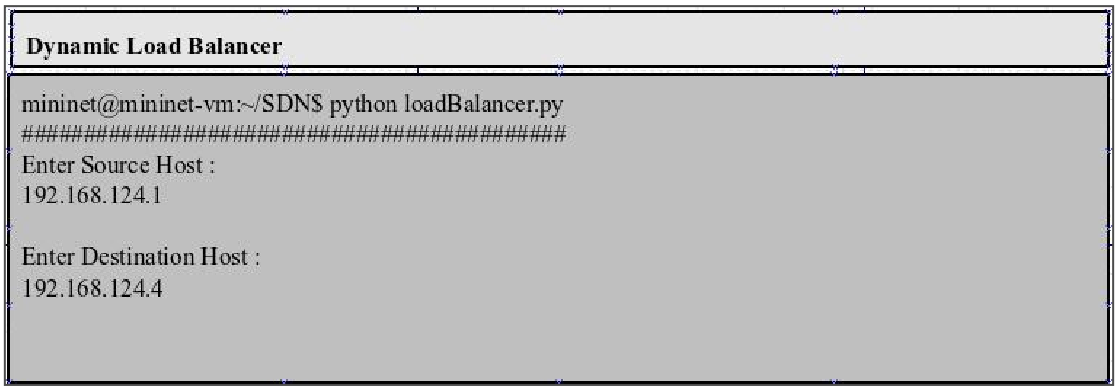
This shows that network paths are not properly utilized before load balancing.

5.4 Path selection after Load Balancing

The REST APIs Dynamic load balancer is used to collect operation information of the topology and its devices. It enables statistics collection in case of Floodlight (TX i.e. Transmission Rate, RX i.e. Receiving Rate, etc). It finds information about hosts n dynamic load balancing, the work load is calculated and distributed among the servers at runtime. The controller assigns new requests to the servers based on the load information collected. As shown in figure 3, a set of clients and servers are connected to a network. The controller connected to the network communicates via the OpenFlow protocol and has a set of defined load balancing algorithms.

connected such as their IP, Switch to which they are connected, MAC Addresses, Port mapping, etc 3. It obtains path/route information from source Host to destination Host i..e. the hosts between load balancing has to be performed. It finds total link cost for all these paths between source Host 1 and destination Host 2. The flows are created depending on the minimum transmission cost of the links at the given time. Based on the cost, the best path is decided and static flows are pushed into each switch in the current best path. Information such as In-Port, Out-Port, Source IP, Destination IP, Source MAC, Destination MAC is fed to the flows. The program continues to update this information every 1 second thereby making it dynamic.

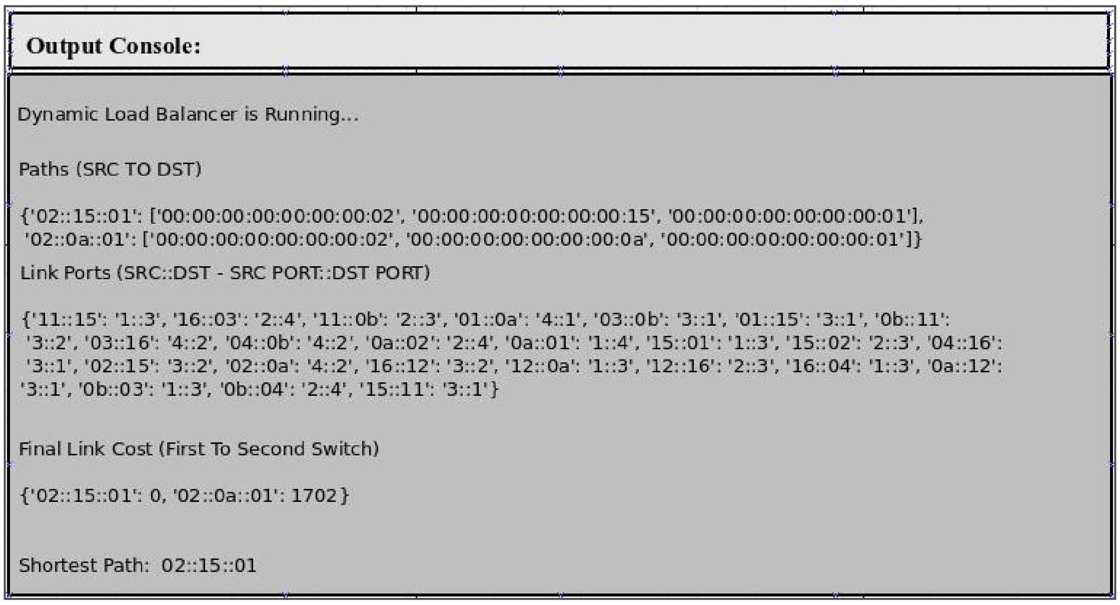
The Figure 24 shows dynamic load balancer initialization and its output to find the shortest possible path exists between source and destination node.



Dynamic Load Balancer

```
mininet@mininet-vm:~/SDN$ python loadBalancer.py
#############################################
Enter Source Host :
192.168.124.1

Enter Destination Host :
192.168.124.4
```

Figure 24: Initialization of dynamic load balancer



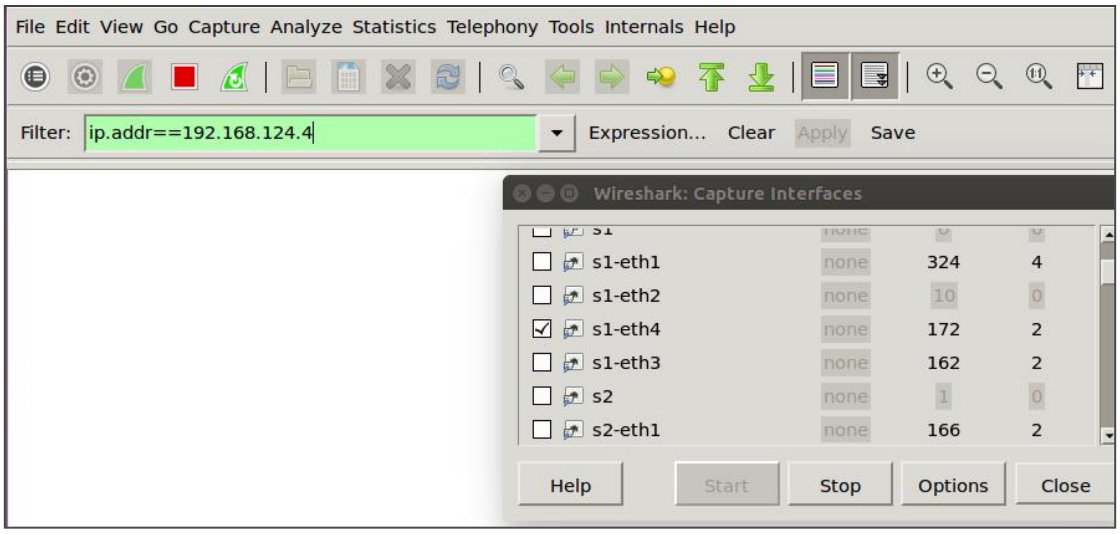
Output Console:

```
Dynamic Load Balancer is Running...

Paths (SRC TO DST)

{'02::15::01': ['00:00:00:00:00:00:00:02', '00:00:00:00:00:00:00:15', '00:00:00:00:00:00:00:01'],
 '02::0a::01': ['00:00:00:00:00:00:00:02', '00:00:00:00:00:00:00:0a', '00:00:00:00:00:00:00:01']}
Link Ports (SRC::DST - SRC PORT::DST PORT)

{'11::15': '1::3', '16::03': '2::4', '11::0b': '2::3', '01::0a': '4::1', '03::0b': '3::1', '01::15': '3::1', '0b::11':
 '3::2', '03::16': '4::2', '04::0b': '4::2', '0a::02': '2::4', '0a::01': '1::4', '15::01': '1::3', '15::02': '2::3', '04::16':
 '3::1', '02::15': '3::2', '02::0a': '4::2', '16::12': '3::2', '12::0a': '1::3', '12::16': '2::3', '16::04': '1::3', '0a::12':
'3::1', '0b::03': '1::3', '0b::04': '2::4', '15::11': '3::1'}


Final Link Cost (First To Second Switch)

{'02::15::01': 0, '02::0a::01': 1702}


Shortest Path:  02::15::01
```

Figure 25 : Output of dynamic load balancer showing shortest path for h1 to h4

After the dynamic load balancer is invoked, all the paths that exist between source node and destination nodes are determined. The load balancer then lists paths from source to destination along with the link ports connections. Finally, link costs between each path is calculated and listed. Ultimately, the shortest path between source and destination is chosen.

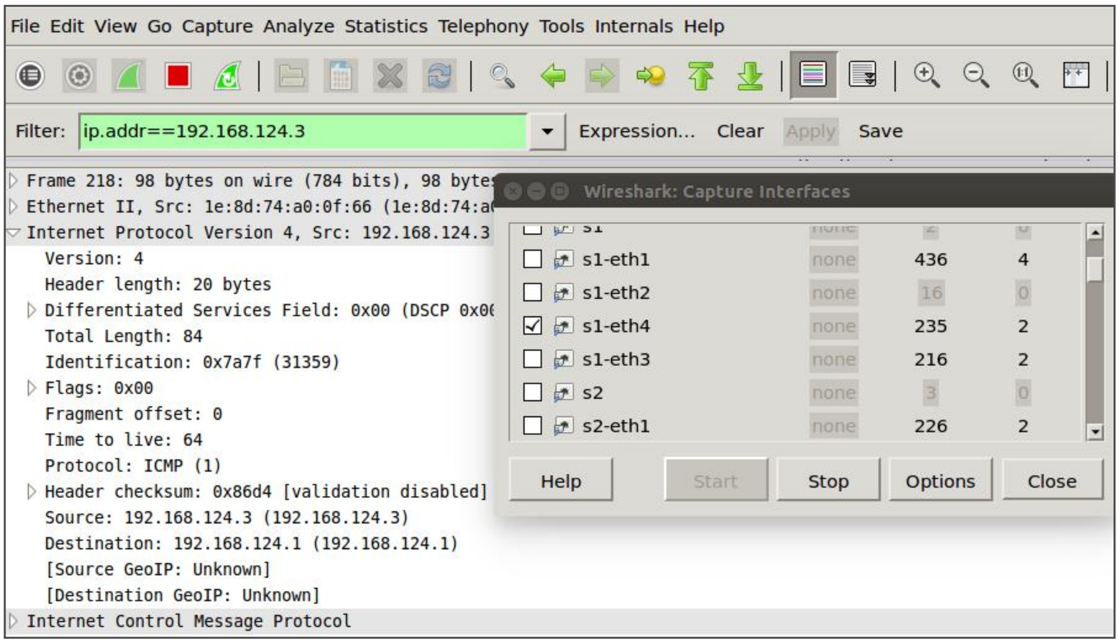
Here in the Figure 26, the paths between node h1 and node h4 are 02::15::01 and 02::0a::01. Between these 2 paths, the path 02::15:01 is chosen as its path cost is 0. The packets of 192.168.124.4 can not be observed in the interface s1-eth4 after load balancing when h1 pings h4 which is shown in the Figure 26.


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help Start Stop Options Close

Filter: ip.addr==192.168.124.4

☐	s1-eth1	none	324	4
☐	s1-eth2	none	10	0
✓	s1-eth4	none	172	2
☐	s1-eth3	none	162	2
☐	s2	none	1	0
☐	s2-eth1	none	166	2

Figure 26 : Wireshark capture in interface s1-eth4 for packets of 192.168.124.4


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help Start Stop Options Close

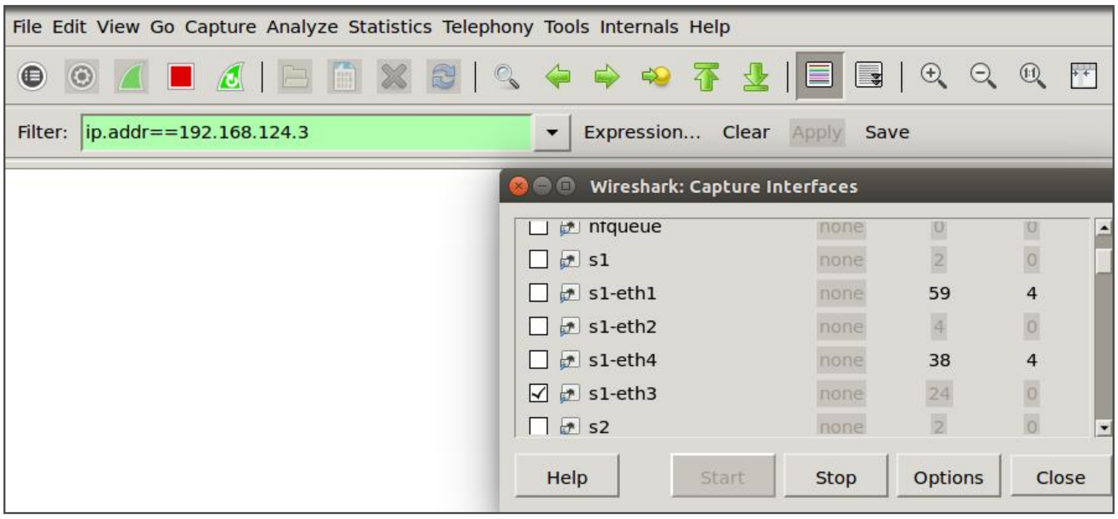
Filter: ip.addr==192.168.124.3

```
▷ Frame 218: 98 bytes on wire (784 bits), 98 byte
▷ Ethernet II, Src: 1e:8d:74:a0:0f:66 (1e:8d:74:a
▽ Internet Protocol Version 4, Src: 192.168.124.3
    Version: 4
    Header length: 20 bytes
  ▷ Differentiated Services Field: 0x00 (DSCP 0x0
    Total Length: 84
    Identification: 0x7a7f (31359)
  ▷ Flags: 0x00
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
  ▷ Header checksum: 0x86d4 [validation disabled]
    Source: 192.168.124.3 (192.168.124.3)
    Destination: 192.168.124.1 (192.168.124.1)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▷ Internet Control Message Protocol
```

☐	s1-eth1	none	436	4
☐	s1-eth2	none	16	0
✓	s1-eth4	none	235	2
☐	s1-eth3	none	216	2
☐	s2	none	3	0
☐	s2-eth1	none	226	2

Figure 27 : Wireshark capture in interface s1-eth4 for packets of 192.168.124.3

The packets of 192.168.124.3 can not be observed in the interface s1-eth3 after load balancing when h1 pings h3 which is shown in the Figure 28.

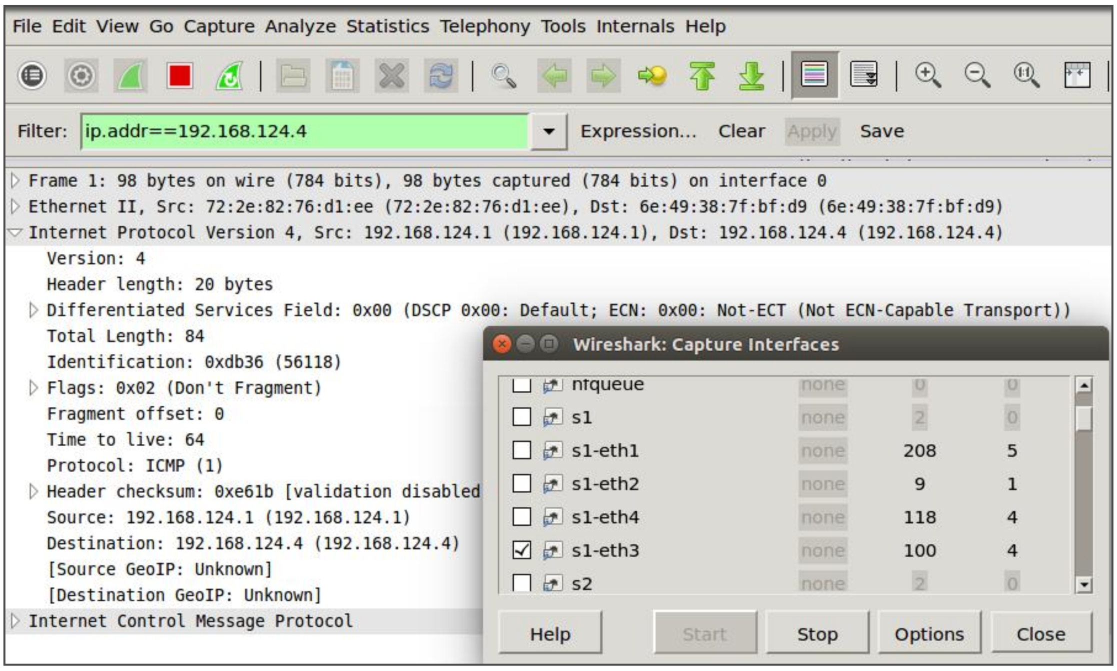

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help Start Stop Options Close

Filter: ip.addr==192.168.124.3

☐	nfqueue	none	0	0
☐	s1	none	2	0
☐	s1-eth1	none	59	4
☐	s1-eth2	none	4	0
☐	s1-eth4	none	38	4
☑	s1-eth3	none	24	0
☐	s2	none	2	0

Figure 28 : Wireshark capture in interface s1-eth3 for packets of 192.168.124.3

But, the packets of 192.168.124.4 can be observed in the interface s1-eth3 after load balancing when h1 pings h4 which is shown in the Figure 29. In other words, the wireshark capture filter of ip address 192.168.124.4 at the interface s1-eth3 can be observed after load balancing in SDN.


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help Start Stop Options Close

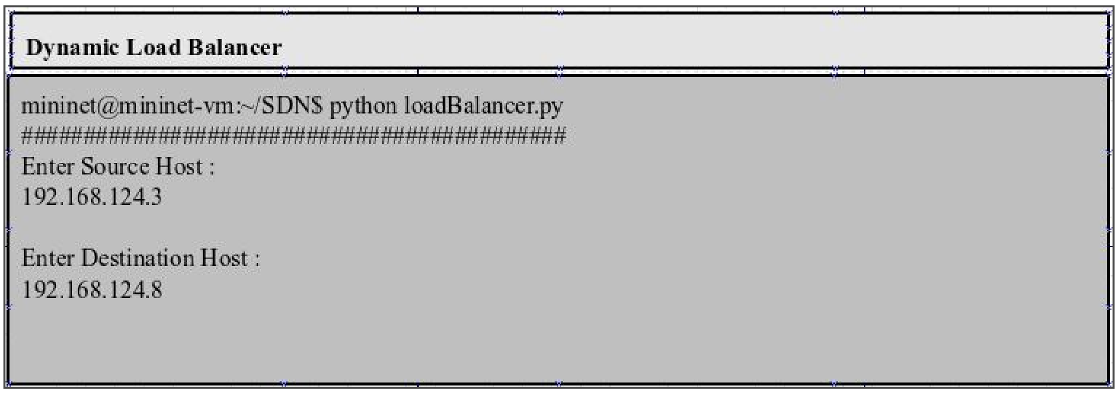
Filter: ip.addr==192.168.124.4

```
▷ Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▷ Ethernet II, Src: 72:2e:82:76:d1:ee (72:2e:82:76:d1:ee), Dst: 6e:49:38:7f:bf:d9 (6e:49:38:7f:bf:d9)
▽ Internet Protocol Version 4, Src: 192.168.124.1 (192.168.124.1), Dst: 192.168.124.4 (192.168.124.4)
    Version: 4
    Header length: 20 bytes
  ▷ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
    Total Length: 84
    Identification: 0xdb36 (56118)
  ▷ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
  ▷ Header checksum: 0xe61b [validation disabled
    Source: 192.168.124.1 (192.168.124.1)
    Destination: 192.168.124.4 (192.168.124.4)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▷ Internet Control Message Protocol
```

☐	nfqueue	none	0	0
☐	s1	none	2	0
☐	s1-eth1	none	208	5
☐	s1-eth2	none	9	1
☐	s1-eth4	none	118	4
☑	s1-eth3	none	100	4
☐	s2	none	2	0

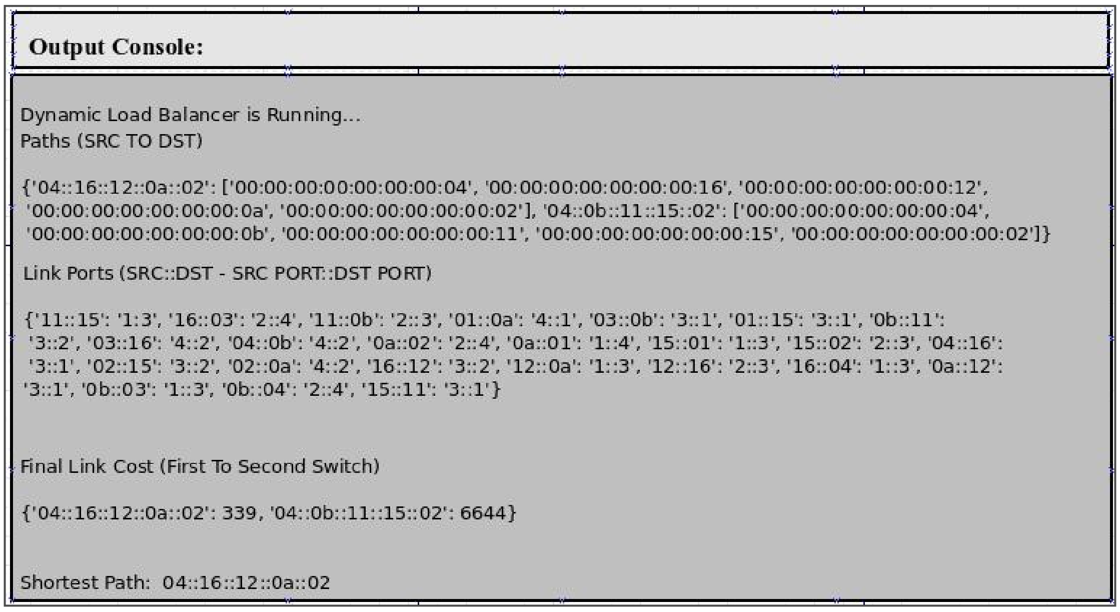
Figure 29 : Wireshark capture in interface s1-eth3 for packets of 192.168.124.4

This shows that network path between host h1 to host h3 is properly utilized after the implementation of dynamic load balancer in SDN network.



```
Dynamic Load Balancer

mininet@mininet-vm:~/SDN$ python loadBalancer.py
############################################
Enter Source Host :
192.168.124.3

Enter Destination Host :
192.168.124.8
```

Figure 30 : Initialization of dynamic load balancer for path h3 to h8



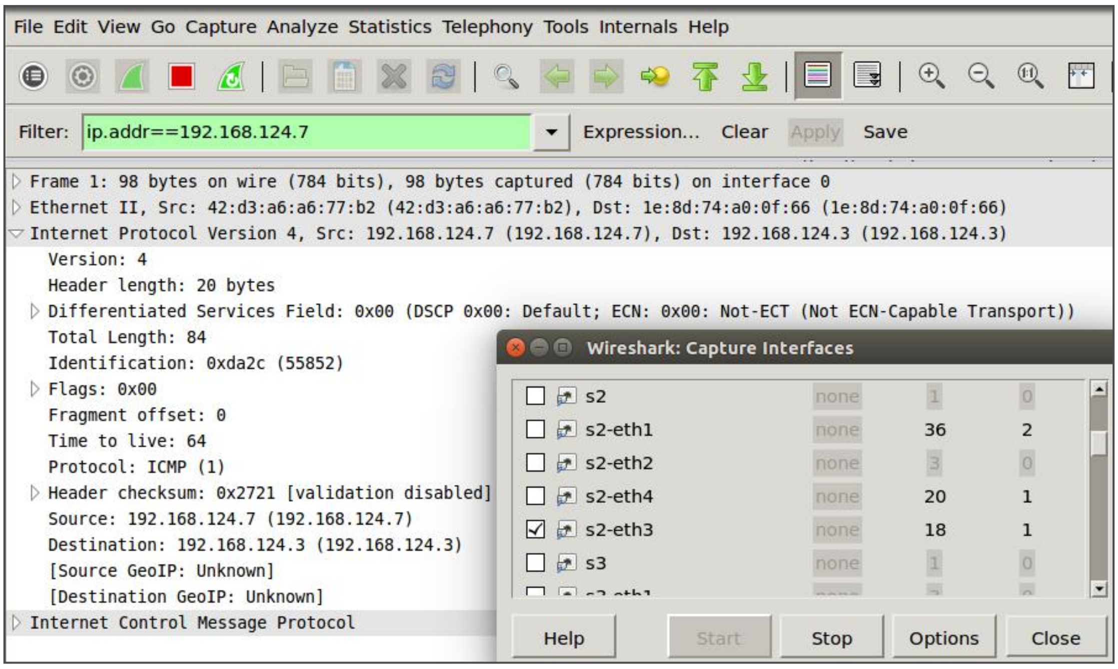
```
Output Console:

Dynamic Load Balancer is Running...
Paths (SRC TO DST)

{'04::16::12::0a::02': ['00:00:00:00:00:00:00:04', '00:00:00:00:00:00:00:16', '00:00:00:00:00:00:00:12',
 '00:00:00:00:00:00:00:0a', '00:00:00:00:00:00:00:02'], '04::0b::11::15::02': ['00:00:00:00:00:00:00:04',
 '00:00:00:00:00:00:00:0b', '00:00:00:00:00:00:00:11', '00:00:00:00:00:00:00:15', '00:00:00:00:00:00:00:02']}

Link Ports (SRC::DST - SRC PORT::DST PORT)

{'11::15': '1:3', '16::03': '2::4', '11::0b': '2::3', '01::0a': '4::1', '03::0b': '3::1', '01::15': '3::1', '0b::11':
 '3::2', '03::16': '4::2', '04::0b': '4::2', '0a::02': '2::4', '0a::01': '1::4', '15::01': '1::3', '15::02': '2::3', '04::16':
 '3::1', '02::15': '3::2', '02::0a': '4::2', '16::12': '3::2', '12::0a': '1::3', '12::16': '2::3', '16::04': '1::3', '0a::12':
'3::1', '0b::03': '1::3', '0b::04': '2::4', '15::11': '3::1'}


Final Link Cost (First To Second Switch)

{'04::16::12::0a::02': 339, '04::0b::11::15::02': 6644}


Shortest Path:  04::16::12::0a::02
```

Figure 31 : Output of dynamic load balancer showing shortest path for h3 and h8

Once the dynamic load balancer is invoked, all the paths that exist between source node h3 and destination node h8 are determined. The load balancer then lists paths from source h3 to destination h8 along with the link ports connections. Finally, link costs between each path is calculated and listed. Ultimately, the shortest path between source h3 and destination h8 is chosen. Through that path, the communication between source and destination takes place.

The Figure 31 shows that there exists 2 paths between node h3 and node h8. They are represented as 04::16::12::0a::02 and 04::0b::11::15::02. Out of these available 2 paths, the path 04::16::12::0a::02 is chosen as its path cost is 339 which is lesser than the path cost of 04::0b::11::15::02. In this way, the shortest path is chosen wisely by the load balancer which improves the quality of service in the network. This is the sole objective of implementing the dynamic load balancer in SDN.

Packets of 192.168.124.7 can be observed in the interface s1-eth3 after load balancing when h3 pings h7 which is shown in the Figure 32. In other words, the wireshark capture filter of ip address 192.168.124.7 at the interface s2-eth4 can be observed after load balancing in SDN.

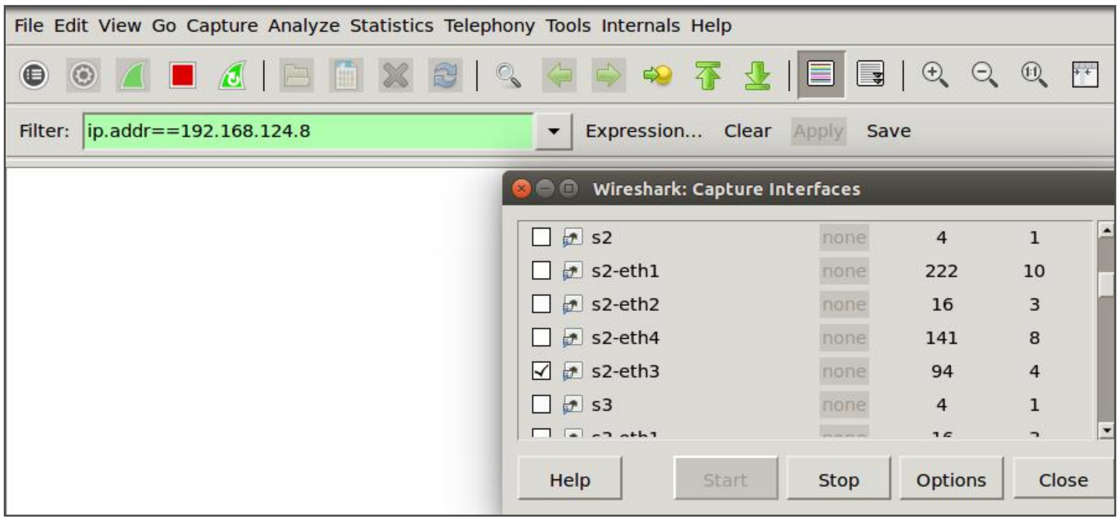

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: ip.addr==192.168.124.7
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help Start Stop Options Close

```
▷ Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▷ Ethernet II, Src: 42:d3:a6:a6:77:b2 (42:d3:a6:a6:77:b2), Dst: 1e:8d:74:a0:0f:66 (1e:8d:74:a0:0f:66)
▽ Internet Protocol Version 4, Src: 192.168.124.7 (192.168.124.7), Dst: 192.168.124.3 (192.168.124.3)
    Version: 4
    Header length: 20 bytes
  ▷ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
    Total Length: 84
    Identification: 0xda2c (55852)
  ▷ Flags: 0x00
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
  ▷ Header checksum: 0x2721 [validation disabled]
    Source: 192.168.124.7 (192.168.124.7)
    Destination: 192.168.124.3 (192.168.124.3)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▷ Internet Control Message Protocol
```

☐	s2	none	1	0
☐	s2-eth1	none	36	2
☐	s2-eth2	none	3	0
☐	s2-eth4	none	20	1
☑	s2-eth3	none	18	1
☐	s3	none	1	0

Figure 32: Wire-shark capture in the interface s2-eth3 for packets of 192.168.124.7

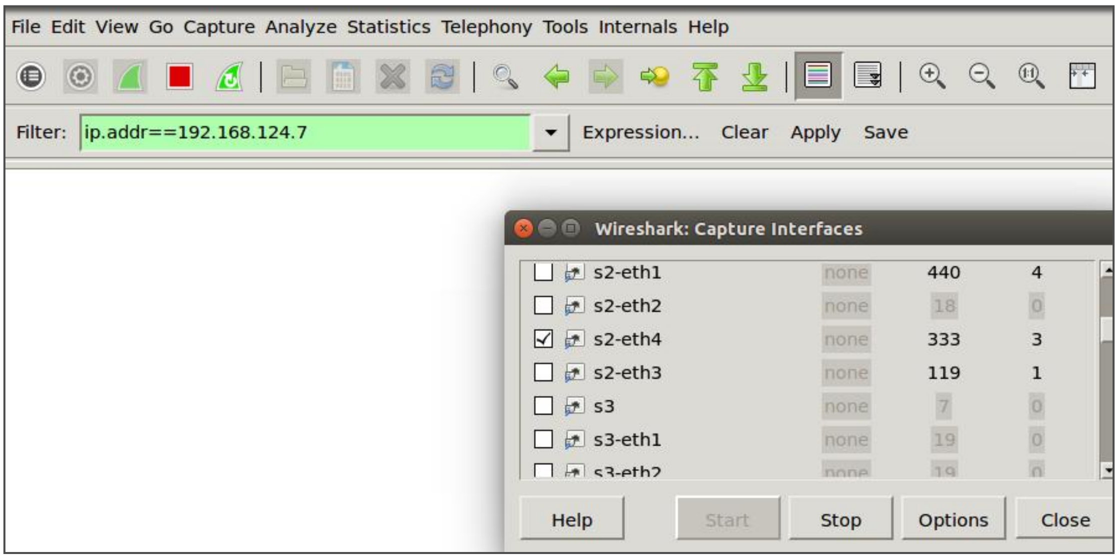
Packets of 192.168.124.8 can not be observed in the interface s1-eth3 after load balancing when h3 pings h8 which is shown in the Figure 33.


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: ip.addr==192.168.124.8
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help Start Stop Options Close

☐	s2	none	4	1
☐	s2-eth1	none	222	10
☐	s2-eth2	none	16	3
☐	s2-eth4	none	141	8
☑	s2-eth3	none	94	4
☐	s3	none	4	1

Figure 33: Wire-shark capture in the interface s2-eth3 for packets of 192.168.124.8

Packets of 192.168.124.7 can not be observed in the interface s1-eth4 after load balancing when h3 pings h7 which is shown in the Figure 34.

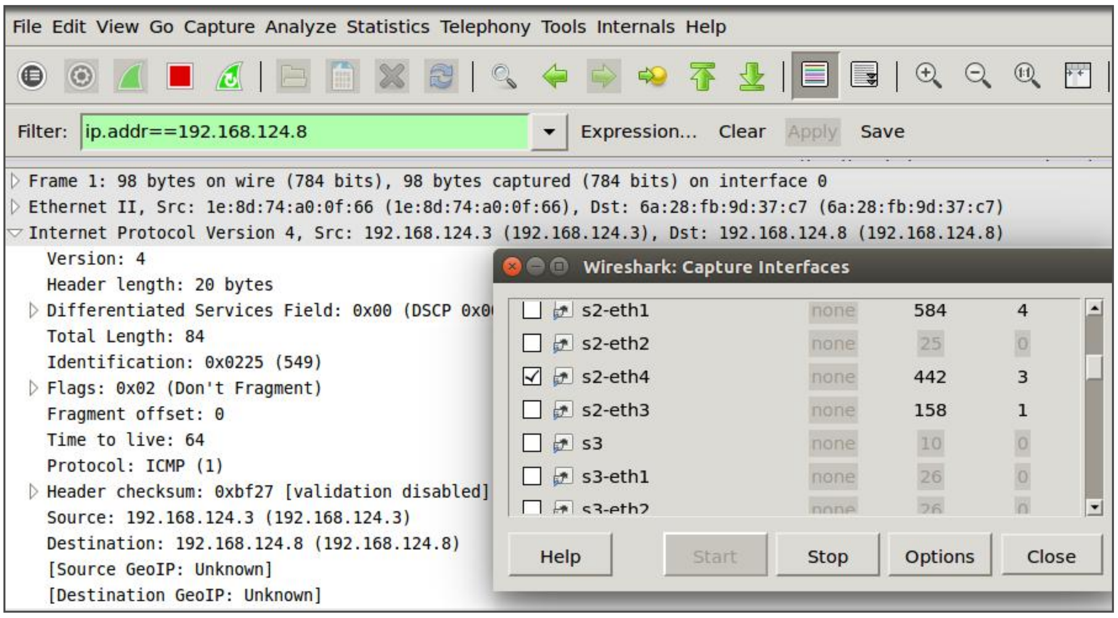

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help Start Stop Options Close

Filter: ip.addr==192.168.124.7

```
[ ] s2-eth1   none  440  4
[ ] s2-eth2   none   18  0
[✓] s2-eth4   none  333  3
[ ] s2-eth3   none  119  1
[ ] s3        none    7  0
[ ] s3-eth1   none   19  0
[ ] s3-eth2   none   19  0
```

Figure 34 : Wire-shark capture in the interface s2-eth4 for packets of 192.168.124.7

When h3 pings h8, packets of 192.168.124.8 can be observed in the interface s1-eth4 after load balancing. This is shown in the Figure 35.


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply Save
Wireshark: Capture Interfaces
Help Start Stop Options Close

Filter: ip.addr==192.168.124.8

```
▷ Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
▷ Ethernet II, Src: 1e:8d:74:a0:0f:66 (1e:8d:74:a0:0f:66), Dst: 6a:28:fb:9d:37:c7 (6a:28:fb:9d:37:c7)
▽ Internet Protocol Version 4, Src: 192.168.124.3 (192.168.124.3), Dst: 192.168.124.8 (192.168.124.8)
    Version: 4
    Header length: 20 bytes
  ▷ Differentiated Services Field: 0x00 (DSCP 0x0
    Total Length: 84
    Identification: 0x0225 (549)
  ▷ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
  ▷ Header checksum: 0xbf27 [validation disabled]
    Source: 192.168.124.3 (192.168.124.3)
    Destination: 192.168.124.8 (192.168.124.8)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
```

```
[ ] s2-eth1   none  584  4
[ ] s2-eth2   none   25  0
[✓] s2-eth4   none  442  3
[ ] s2-eth3   none  158  1
[ ] s3        none   10  0
[ ] s3-eth1   none   26  0
[ ] s3-eth2   none   26  0
```

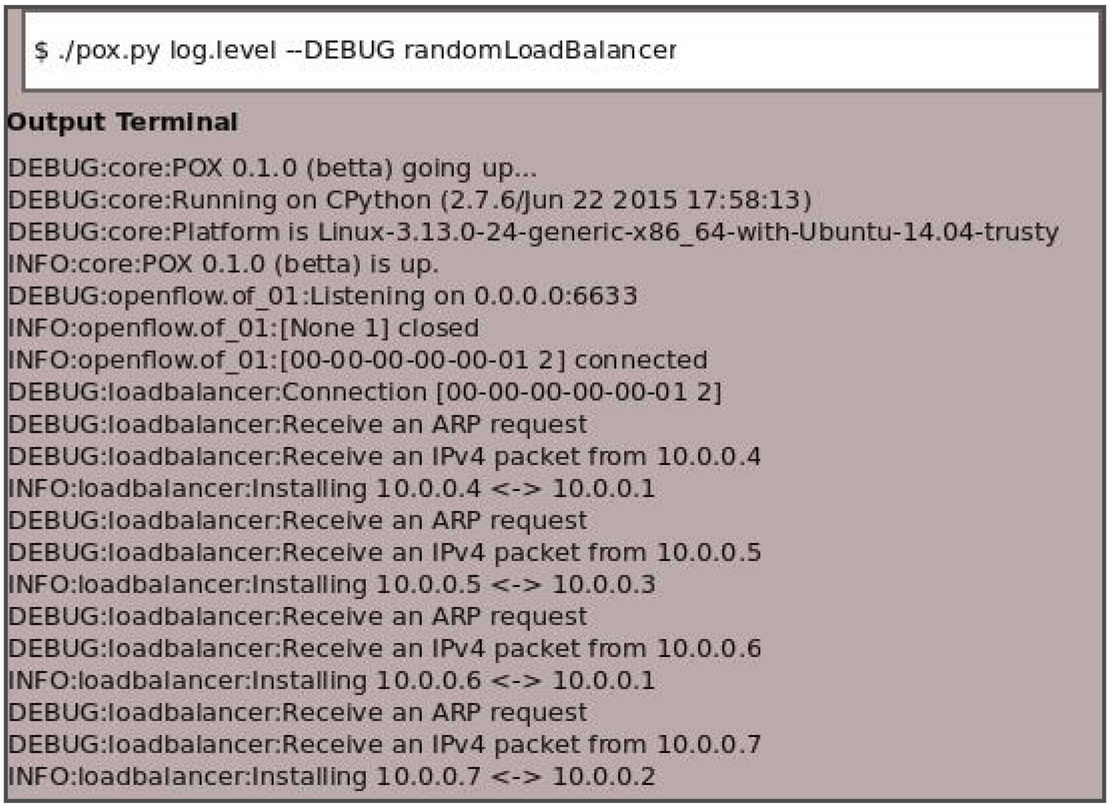
Figure 35: Wire-shark capture in the interface s2-eth4 for packets of 192.168.124.8

To sum up, when node h3 pings node 8, ping packets of 192.168.124.8 go through the interface of s2-eth4 and when node h3 pings node 7, ping packets of 192.168.124.7 go through the interface of s2-eth3. Though there exists the common path between node h3 to node h8 and node h3 to node7, the link which is less utilized is chosen while going packets from node h3 to node h7 or node h8. This is achieved by the implementation of the customized dynamic load balancer in the SDN. The network path is selected appropriately after the implementation of the dynamic load balancer in the SDN.

5.5 Server Load Balancing

Clients request access for web-server after which random load balancing mechanism configured in POX controller gives random server selection.

The core methods used for POX controller are _init_(self,ip,mac,port), _init_(self,connection), get_next_server(self) and handle_arp(self,packet,in_port). A function _init_(self,ip,mac,port) sets IP,MAC of load balancer as 10.0.0.254 and 00:00:00:00:00:FE.



```
$ ./pox.py log.level --DEBUG randomLoadBalancer
```

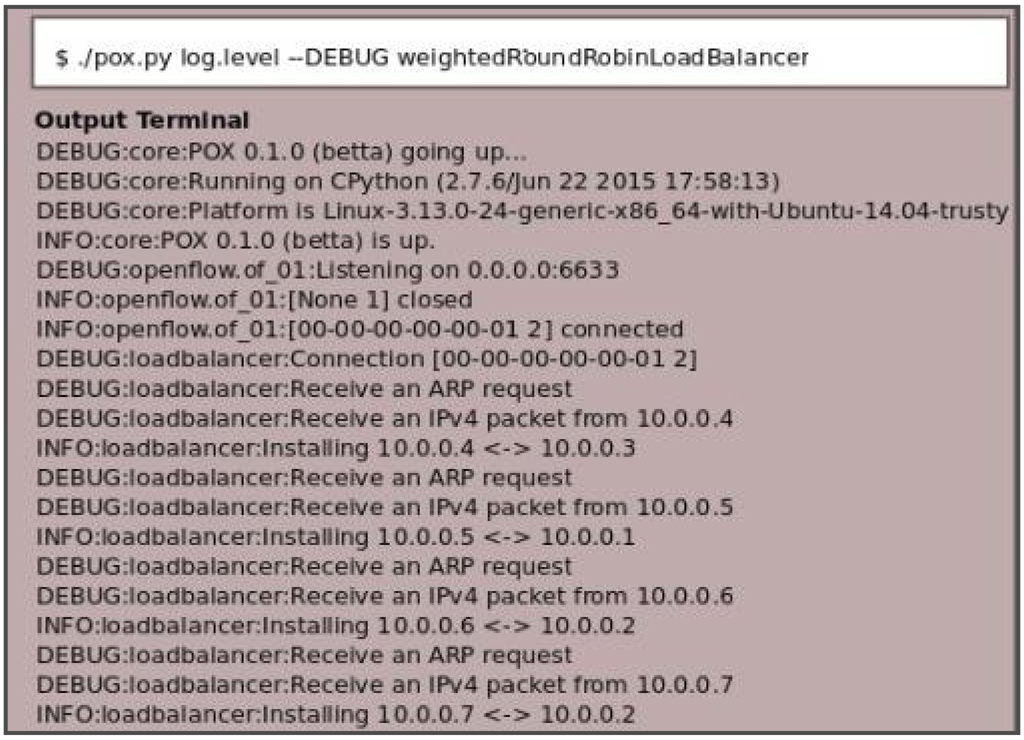
Output Terminal

```
DEBUG:core:POX 0.1.0 (betta) going up...
DEBUG:core:Running on CPython (2.7.6/Jun 22 2015 17:58:13)
DEBUG:core:Platform is Linux-3.13.0-24-generic-x86_64-with-Ubuntu-14.04-trusty
INFO:core:POX 0.1.0 (betta) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[None 1] closed
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
DEBUG:loadbalancer:Connection [00-00-00-00-00-01 2]
DEBUG:loadbalancer:Receive an ARP request
DEBUG:loadbalancer:Receive an IPv4 packet from 10.0.0.4
INFO:loadbalancer:Installing 10.0.0.4 <-> 10.0.0.1
DEBUG:loadbalancer:Receive an ARP request
DEBUG:loadbalancer:Receive an IPv4 packet from 10.0.0.5
INFO:loadbalancer:Installing 10.0.0.5 <-> 10.0.0.3
DEBUG:loadbalancer:Receive an ARP request
DEBUG:loadbalancer:Receive an IPv4 packet from 10.0.0.6
INFO:loadbalancer:Installing 10.0.0.6 <-> 10.0.0.1
DEBUG:loadbalancer:Receive an ARP request
DEBUG:loadbalancer:Receive an IPv4 packet from 10.0.0.7
INFO:loadbalancer:Installing 10.0.0.7 <-> 10.0.0.2
```

Figure 36 : Output of Random Load Balancer

Initially, random load balancer controller is attached with 00-00-00-00-00-01 MAC address with IP address 10.0.0.254. When clients demands for access to the web server HTTP, then it receives an ARP request. In the Figure 36; 10.0.0.4 station demands the request. Then POX controller installs 10.0.0.4 with 10.0.0.1 web HTTP server at port 80 using random selection logic. Consider second case in the Figure 36, 10.0.0.5 station demands the request. Then POX controller installs 10.0.0.5 with 10.0.0.3 web HTTP server at port 80 using random selection logic. In third case, 10.0.0.6 station demands the request. Then POX controller installs 10.0.0.6 with 10.0.0.1 web HTTP server at port 80 using random selection logic. This shows that there is random selection of server without recognizing the server load demand load weight.

Weighted Round-Robin load balancing provides greater efficiency compared to random load balancing.



```
$ ./pox.py log.level --DEBUG weightedRoundRobinLoadBalancer

Output Terminal
DEBUG:core:POX 0.1.0 (betta) going up...
DEBUG:core:Running on CPython (2.7.6/Jun 22 2015 17:58:13)
DEBUG:core:Platform is Linux-3.13.0-24-generic-x86_64-with-Ubuntu-14.04-trusty
INFO:core:POX 0.1.0 (betta) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[None 1] closed
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
DEBUG:loadbalancer:Connection [00-00-00-00-00-01 2]
DEBUG:loadbalancer:Receive an ARP request
DEBUG:loadbalancer:Receive an IPv4 packet from 10.0.0.4
INFO:loadbalancer:Installing 10.0.0.4 <-> 10.0.0.3
DEBUG:loadbalancer:Receive an ARP request
DEBUG:loadbalancer:Receive an IPv4 packet from 10.0.0.5
INFO:loadbalancer:Installing 10.0.0.5 <-> 10.0.0.1
DEBUG:loadbalancer:Receive an ARP request
DEBUG:loadbalancer:Receive an IPv4 packet from 10.0.0.6
INFO:loadbalancer:Installing 10.0.0.6 <-> 10.0.0.2
DEBUG:loadbalancer:Receive an ARP request
DEBUG:loadbalancer:Receive an IPv4 packet from 10.0.0.7
INFO:loadbalancer:Installing 10.0.0.7 <-> 10.0.0.2
```

Figure 37 : Output of Weighted Round Robin Load Balancer

Weighted Random load balancer controller is attached with 00-00-00-00-00-01 MAC address with IP address 10.0.0.254. When clients demands for access to the web server HTTP, then it receives an ARP request. In the Figure 37, 10.0.0.4 station demands the request. Then POX controller installs 10.0.0.4 with 10.0.0.3 web HTTP server at port 80 using weighted round robin selection logic. Consider second case in the Figure 37, 10.0.0.5 station demands the request. Then POX controller installs 10.0.0.5 with 10.0.0.1 web HTTP server at port 80. In third case, 10.0.0.6 station demands the request. Then 10.0.0.6 client is served with 10.0.0.2 web HTTP server at port 80 . There is appropriate distribution of load. With weighted round robin load balancing, the network will be stable when the demand of load from clients is huge as there is balancing of load on demand.

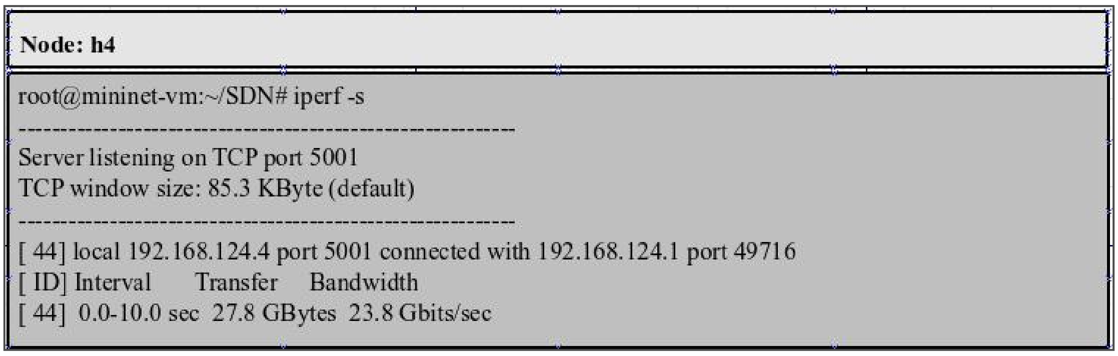
CHAPTER 6: RESULTS ANALYSIS AND DISCUSSION

The implementation of dynamic load balancer results decrease in latency of packets to reach from source to destination. It is due to the proper path selection for packets to travel from source to destination. The data transfer and bandwidth measurement of the network path is done before and after load balancing. Along with that, ping test is done before and after load balancing to find latency improvement in the SDN network.

6.1 Bandwidth Test Analysis

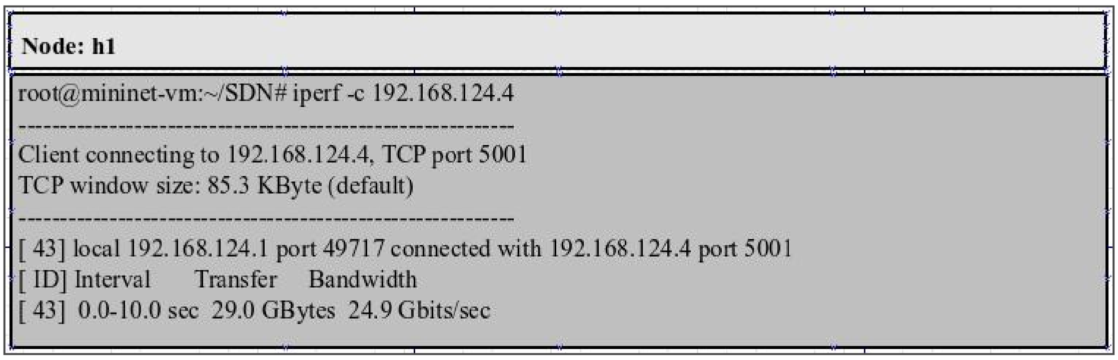
Iperf test provides amount of data transfer and bandwidth between source and destination nodes. Individual 25 iperf tests have been performed before and after load balancing. Table 1 and Table 2 show data populated with amount of data transfer and bandwidth between the node h1 and h4 before and after load balancing successively.

Consider the situation in which node h1 pings node h4 before using load balancer. Then the data transfer and bandwidth measurement between node h1 and node h4 is shown in the Figure 38 and Figure 39.



```
Node: h4
root@mininet-vm:~/SDN# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 44] local 192.168.124.4 port 5001 connected with 192.168.124.1 port 49716
[ ID] Interval       Transfer     Bandwidth
[ 44]  0.0-10.0 sec  27.8 GBytes  23.8 Gbits/sec
```

Figure 38: Data transfer and bandwidth measurement in h4 IPERF server



```
Node: h1
root@mininet-vm:~/SDN# iperf -c 192.168.124.4
------------------------------------------------------------
Client connecting to 192.168.124.4, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 43] local 192.168.124.1 port 49717 connected with 192.168.124.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 43]  0.0-10.0 sec  29.0 GBytes  24.9 Gbits/sec
```

Figure 39 : Data transfer and bandwidth measurement in h1 IPERF client

For observation, node h4 is set to iperf server and node h1 is iperf client. In the same way node h8 is set ot iperf server and node h3 is set to iperf client.

The Table 1 and Table 2 are populated with the iperf tests for ping between h1 and h4.

Table 1: iPerf h1 to h4 before Load Balancing

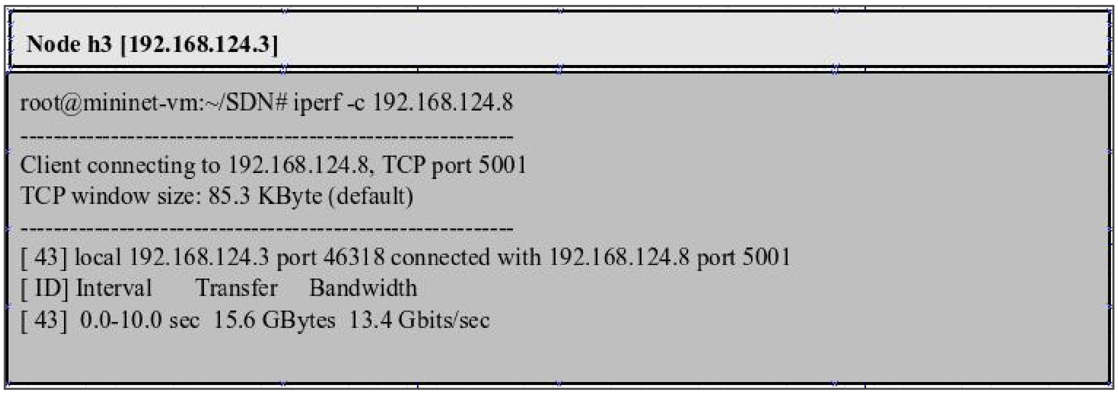
No. Of Observations	Transfer (Gbytes)	B/W(Gbits/s)
1	27.8	23.8
2	27.11	23.31
3	27.23	23.5
4	26.92	22
5	26.82	24.12
6	27.11	22.11
7	27.34	23.34
8	26.89	22.34
9	26.97	21.21
10	27.43	23.45
11	26.87	22.23
12	27	23.21
13	27	22.42
14	27.32	23.95
15	26.34	23.67
16	26	23.76
17	26.9	22.95
18	29.45	23.19
19	28	24.12
20	26	22.11
21	26.32	23.34
22	27.9	23.34
23	27.8	23.89
24	28.9	22.89
25	27.9	23.11
	Average: 27.25	**Average: 23.09**

Table 2: iPerf h1 to h4 after Load Balancing

No. Of Observations	Transfer (Gbytes)	B/W(Gbits/s)
1	54.11	46.23
2	53.91	45.54
3	53.78	46.5
4	54.11	46.12
5	54.23	46.71
6	53.98	45.76
7	54.12	45.65
8	53.76	45.98
9	53.87	46.45

46		
10	54.23	46.67
11	53.56	45.78
12	54.26	46.23
13	53.22	46.23
14	54.13	45.65
15	54.78	45.67
16	53.25	46.61
17	54.89	45.98
18	54.23	46.21
19	54.89	45.55
20	53.23	46.23
21	53.56	45.11
22	53.19	46.81
23	53.29	45.23
24	54.12	46.34
25	53.89	46.12
	Average: 53.94	**Average: 46.05**

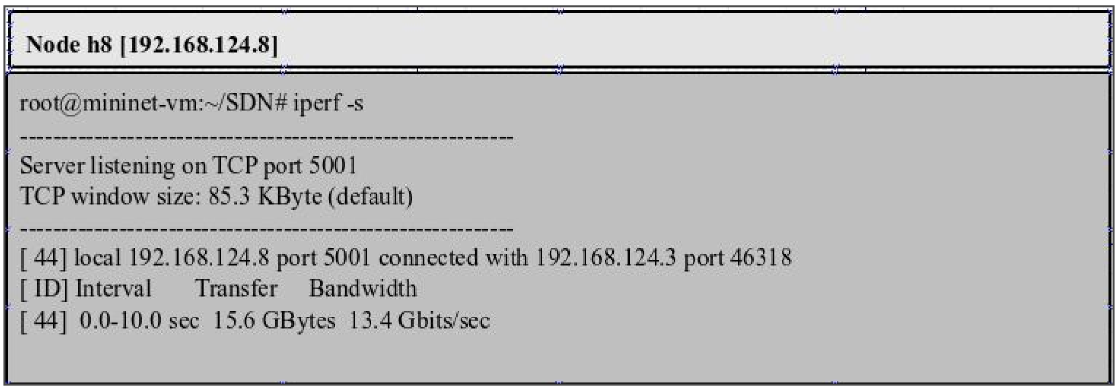
The Table 1 shows iperf tests between node h1 and h4 for 25 observations before the implementation of load balancer in software defined network. The Table 1 consists of average transfer of 27.25 GBytes for packets and 23.09 Gbits/s bandwidth utilization of the links in the network before the implementation of customized load balancer in the SDN. The Table 2 shows average transfer of 53.94 GBytes for packets and 46.05 Gbits/s bandwidth utilization after the implementation of load balancer.



Node h3 [192.168.124.3]

```
root@mininet-vm:~/SDN# iperf -c 192.168.124.8
------------------------------------------------------------
Client connecting to 192.168.124.8, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 43] local 192.168.124.3 port 46318 connected with 192.168.124.8 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 43]  0.0-10.0 sec  15.6 GBytes  13.4 Gbits/sec
```

Figure 40: Data transfer and bandwidth measurement in h3 IPERF client



```
Node h8 [192.168.124.8]

root@mininet-vm:~/SDN# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 44] local 192.168.124.8 port 5001 connected with 192.168.124.3 port 46318
[ ID] Interval       Transfer     Bandwidth
[ 44]  0.0-10.0 sec  15.6 GBytes  13.4 Gbits/sec
```

Figure 41: Data transfer and bandwidth measurement in h8 IPERF server

The Table 3 and Table 4 are populated with the iperf tests for ping between h3 and h8.

Table 3: iPerf h3 to h8 before Load Balancing

No. Of Observations	Transfer (Gbytes)	B/W(Gbits/s)
1	15.6	13.4
2	15.41	12.98
3	15.31	13.3
4	15.7	12.8
5	15.2	13.32
6	15.8	12.98
7	15.23	12.76
8	15.11	13.87
9	15.46	12.67
10	15.76	12.98
11	15.34	13.45
12	15.34	13.56
13	15.11	13.54
14	14.99	13.78
15	14.67	13.82
16	16.11	12.67
17	15.23	12.87
18	15.14	13.11
19	16.12	13.56
20	15.12	14.02
21	16.23	13.82

No. Of Observations	Transfer (Gbytes)	B/W(Gbits/s)
22	15.24	13.39
23	15.76	12.88
24	15.87	12.72
25	15.98	13.62
	Average: 15.47	**Average: 13.62**

Table 3 and Table 4 show data populated with amount of data transfer and bandwidth between the node h3 and h8 before and after load balancing successively. iPerf enables to check the amount of bytes transferred and the rate i.e. Bandwidth. It shows how the average transferred data increases after load balancing. The average transfer for node h3 to h8 before load balancing is 15.47 Gbytes and average bandwidth utilization is 13.62 Gbits/s. After load balancing average transfer for node h3 to h8 is 15.47 Gbytes and average bandwidth utilization is 13.62 Gbits/s.

Table 4: iPerf h3 to h8 after Load Balancing

No. Of Observations	Transfer (Gbytes)	B/W(Gbits/s)
1	31.12	25.7
2	30.46	26.3
3	30.34	24.52
4	31.6	26.12
5	31.78	26.31
6	30.35	25.89
7	31.7	24.78
8	32.56	26.23
9	32.11	26.65
10	29.45	25.27
11	29.99	25.82
12	30.25	25.82
13	31.02	25.39
14	31.76	25.95
15	31.89	24.98
16	29.03	24.78
17	31.89	24.95

No. Of Observations	Transfer (Gbytes)	B/W(Gbits/s)
18	31.65	24.39
19	31.23	26.02
20	30.78	26.08
21	31.78	26.11
22	31.56	26.38
23	31.67	26.51
24	31.89	.25.17
25	31.67	25.48
	Average: 31.18	**Average: 25.68**

From the Table 1, Table 2, Table 3 and Table 4, it is observed that data transfer rate as well as bandwidth utilization of links increased after the implementation of path load balancer in software defined networking compared with the no path load balancing.

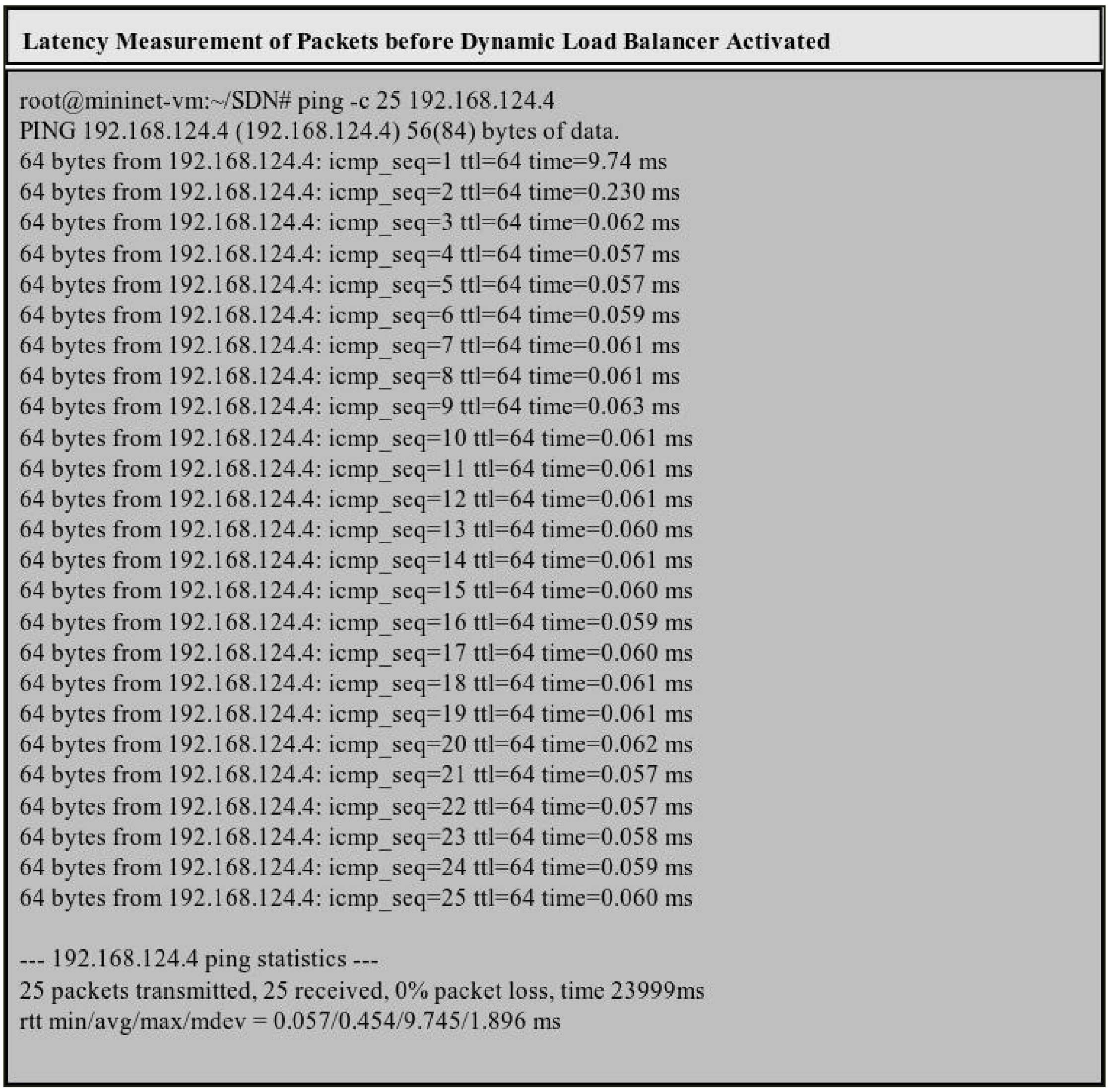
The results show that use of path load balancer effectively utilize the links in SDN.

6.2 Latency Test Analysis

Latency test gives 4 parameters of ping statistics. They are rtt minimum, rtt average, rtt maximum and rtt mdev. It determines the quality of the links. Streaming media, voice, video communications and online gaming require more than just raw speed. There is difference between bandwidth test analysis and latency test analysis. Bandwidth is the total amount of data that can flow through the channel in a given period of time. On the other hand, latency is the amount of time it takes for the data that enters the channel or links at one end to exit at the other. If the link is short and not so congested, then the packets exits the bottom of the link almost as quickly. This means latency of packet in that particular link will be minimum.

Most operating systems contain a utility called "ping". The time in milliseconds is the ping rate between source host and the destination host or the server. In the majority of cases, the ping rate is equivalent to the effective latency between the effective latency between the source and the destination host.

The latency measurement of packets from node h1 to node h4 before dynamic load balancer implementation is shown in Figure 42. The ping statistics of node h4 from node h1 are observed and for the first observation in our case, it is found to be as rtt min/avg/max/mdev =0.057/0.454/9.745/1.895 ms.


Latency Measurement of Packets before Dynamic Load Balancer Activated

```
root@mininet-vm:~/SDN# ping -c 25 192.168.124.4
PING 192.168.124.4 (192.168.124.4) 56(84) bytes of data.
64 bytes from 192.168.124.4: icmp_seq=1 ttl=64 time=9.74 ms
64 bytes from 192.168.124.4: icmp_seq=2 ttl=64 time=0.230 ms
64 bytes from 192.168.124.4: icmp_seq=3 ttl=64 time=0.062 ms
64 bytes from 192.168.124.4: icmp_seq=4 ttl=64 time=0.057 ms
64 bytes from 192.168.124.4: icmp_seq=5 ttl=64 time=0.057 ms
64 bytes from 192.168.124.4: icmp_seq=6 ttl=64 time=0.059 ms
64 bytes from 192.168.124.4: icmp_seq=7 ttl=64 time=0.061 ms
64 bytes from 192.168.124.4: icmp_seq=8 ttl=64 time=0.061 ms
64 bytes from 192.168.124.4: icmp_seq=9 ttl=64 time=0.063 ms
64 bytes from 192.168.124.4: icmp_seq=10 ttl=64 time=0.061 ms
64 bytes from 192.168.124.4: icmp_seq=11 ttl=64 time=0.061 ms
64 bytes from 192.168.124.4: icmp_seq=12 ttl=64 time=0.061 ms
64 bytes from 192.168.124.4: icmp_seq=13 ttl=64 time=0.060 ms
64 bytes from 192.168.124.4: icmp_seq=14 ttl=64 time=0.061 ms
64 bytes from 192.168.124.4: icmp_seq=15 ttl=64 time=0.060 ms
64 bytes from 192.168.124.4: icmp_seq=16 ttl=64 time=0.059 ms
64 bytes from 192.168.124.4: icmp_seq=17 ttl=64 time=0.060 ms
64 bytes from 192.168.124.4: icmp_seq=18 ttl=64 time=0.061 ms
64 bytes from 192.168.124.4: icmp_seq=19 ttl=64 time=0.061 ms
64 bytes from 192.168.124.4: icmp_seq=20 ttl=64 time=0.062 ms
64 bytes from 192.168.124.4: icmp_seq=21 ttl=64 time=0.057 ms
64 bytes from 192.168.124.4: icmp_seq=22 ttl=64 time=0.057 ms
64 bytes from 192.168.124.4: icmp_seq=23 ttl=64 time=0.058 ms
64 bytes from 192.168.124.4: icmp_seq=24 ttl=64 time=0.059 ms
64 bytes from 192.168.124.4: icmp_seq=25 ttl=64 time=0.060 ms

--- 192.168.124.4 ping statistics ---
25 packets transmitted, 25 received, 0% packet loss, time 23999ms
rtt min/avg/max/mdev = 0.057/0.454/9.745/1.896 ms
```

Figure 42: Latency measurement of packets for h1ping h4 before balancing

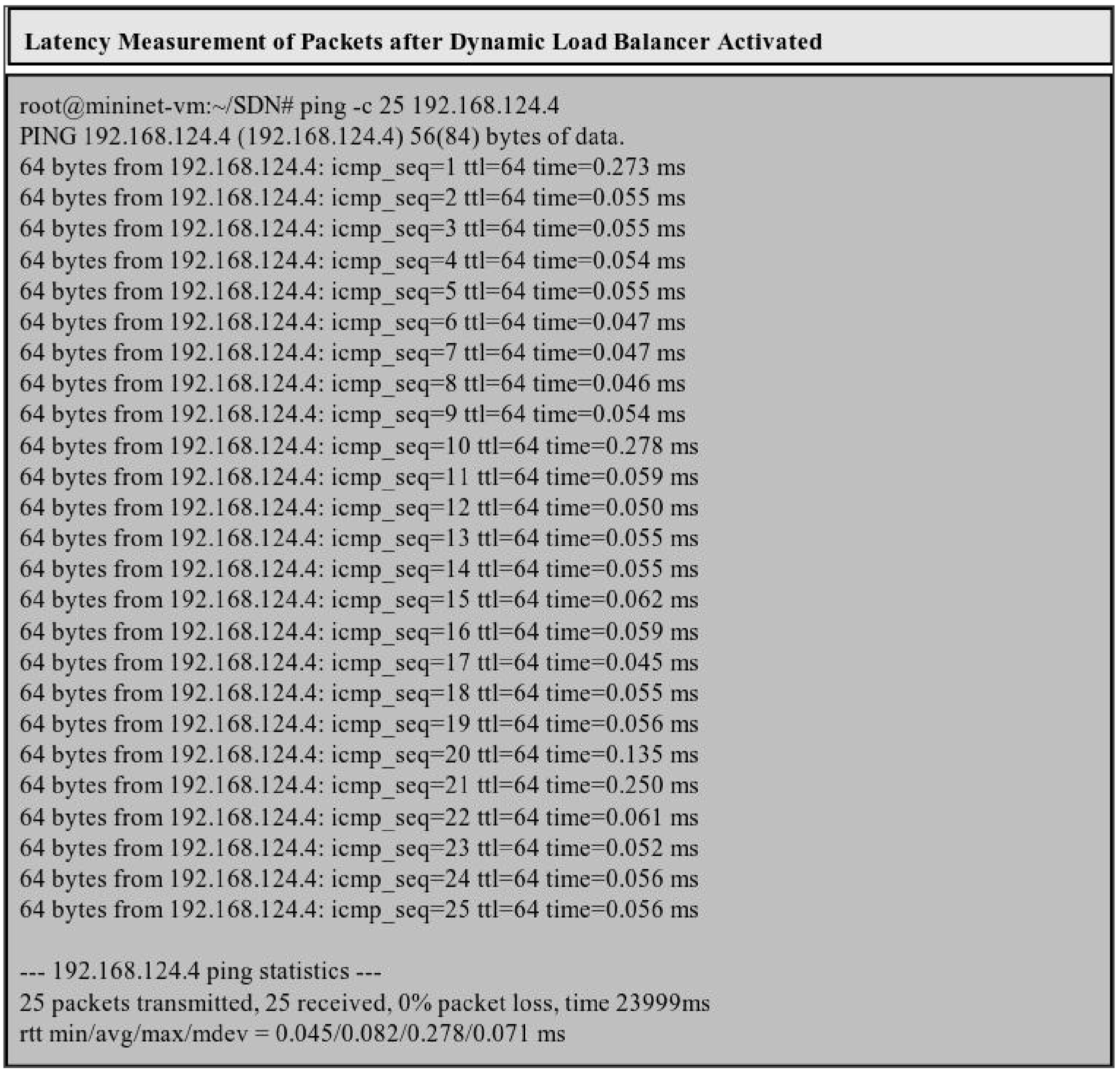
25 Ping tests have been performed on 25 packets/each test from h1 to h4 before load balancing. For the first observation , 25 packets are sent from node h1 to node h4. A ping test is simply a way for source to send a small packet to the destination and to measure the amount of time it takes to get there. The time = 23999 ms at the end of each ping reply is the measured time that last packet took to get to the destination. In this case 23999 ms. Similarly for the second observation, 25 packets are sent from node h1 to node h4. Ping statistics is found to be as rtt min / avg / max / mdev = 0.057 / 0.454 / 9.745 / 1.895 ms. This is recorded in Table 5. Rest 23 observations are performed similar to observation 1 and their results are recorded accordingly in the Table 5.

The average rtt min/avg/max/mdev for ping from h1 to h4 before load balancing in SDN is given by 0.0423/0.445/9.332/1.424 ms where each observation includes 25 packets sent from source h1 to destination h4.

Table 5 : Ping h1 to h4 before Load Balancing (in ms)

Observation	**Min**	**Avg**	**Max**	**Mdev**
1	0.057	0.454	9.745	1.896
2	0.052	0.4224	9.033	1.645
3	0.037	0.4061	9.098	1.019
4	0.052	0.487	9.89	1.957
5	0.042	0.398	9.078	1.579
6	0.036	0.453	9.01	1.02
7	0.038	0.4456	9.34	1.34
8	0.043	0.489	9.34	1.45
9	0.042	0.432	9.21	1.43
10	0.047	0.435	9.891	1.56
11	0.037	0.345	9.034	1.03
12	0.041	0.543	9.45	1.33
13	0.046	0.435	9.045	1.35
14	0.041	0.467	9.034	1.54
15	0.039	0.467	9.32	1.34
16	0.047	0.478	9.45	1.09
17	0.037	0.432	9.13	1.43
18	0.041	0.478	9.89	1.36
19	0.042	0.489	9.34	1.51
20	0.036	0.399	9.24	1.78
21	0.038	0.411	9.891	1.67
22	0.042	0.478	9.034	1.35
23	0.047	0.378	9.45	1.54
24	0.037	0.478	9.045	1.34
25	0.041	0.437	9.32	1.09
	Average: 0.042	**Average: 0.445**	**Average:9.33**	**Average:1.42**

The latency measurement of packets from node h1 to node h4 before dynamic load balancer implementation must be improved after the implementation of the load balancer in SDN which is the main objective of this thesis. The latency measurement of packets after dynamic load balancer implementation is shown in the Figure 41.


Latency Measurement of Packets after Dynamic Load Balancer Activated

```
root@mininet-vm:~/SDN# ping -c 25 192.168.124.4
PING 192.168.124.4 (192.168.124.4) 56(84) bytes of data.
64 bytes from 192.168.124.4: icmp_seq=1 ttl=64 time=0.273 ms
64 bytes from 192.168.124.4: icmp_seq=2 ttl=64 time=0.055 ms
64 bytes from 192.168.124.4: icmp_seq=3 ttl=64 time=0.055 ms
64 bytes from 192.168.124.4: icmp_seq=4 ttl=64 time=0.054 ms
64 bytes from 192.168.124.4: icmp_seq=5 ttl=64 time=0.055 ms
64 bytes from 192.168.124.4: icmp_seq=6 ttl=64 time=0.047 ms
64 bytes from 192.168.124.4: icmp_seq=7 ttl=64 time=0.047 ms
64 bytes from 192.168.124.4: icmp_seq=8 ttl=64 time=0.046 ms
64 bytes from 192.168.124.4: icmp_seq=9 ttl=64 time=0.054 ms
64 bytes from 192.168.124.4: icmp_seq=10 ttl=64 time=0.278 ms
64 bytes from 192.168.124.4: icmp_seq=11 ttl=64 time=0.059 ms
64 bytes from 192.168.124.4: icmp_seq=12 ttl=64 time=0.050 ms
64 bytes from 192.168.124.4: icmp_seq=13 ttl=64 time=0.055 ms
64 bytes from 192.168.124.4: icmp_seq=14 ttl=64 time=0.055 ms
64 bytes from 192.168.124.4: icmp_seq=15 ttl=64 time=0.062 ms
64 bytes from 192.168.124.4: icmp_seq=16 ttl=64 time=0.059 ms
64 bytes from 192.168.124.4: icmp_seq=17 ttl=64 time=0.045 ms
64 bytes from 192.168.124.4: icmp_seq=18 ttl=64 time=0.055 ms
64 bytes from 192.168.124.4: icmp_seq=19 ttl=64 time=0.056 ms
64 bytes from 192.168.124.4: icmp_seq=20 ttl=64 time=0.135 ms
64 bytes from 192.168.124.4: icmp_seq=21 ttl=64 time=0.250 ms
64 bytes from 192.168.124.4: icmp_seq=22 ttl=64 time=0.061 ms
64 bytes from 192.168.124.4: icmp_seq=23 ttl=64 time=0.052 ms
64 bytes from 192.168.124.4: icmp_seq=24 ttl=64 time=0.056 ms
64 bytes from 192.168.124.4: icmp_seq=25 ttl=64 time=0.056 ms

--- 192.168.124.4 ping statistics ---
25 packets transmitted, 25 received, 0% packet loss, time 23999ms
rtt min/avg/max/mdev = 0.045/0.082/0.278/0.071 ms
```

Figure 43: Latency measurement of packets for h1ping h4 after balancing

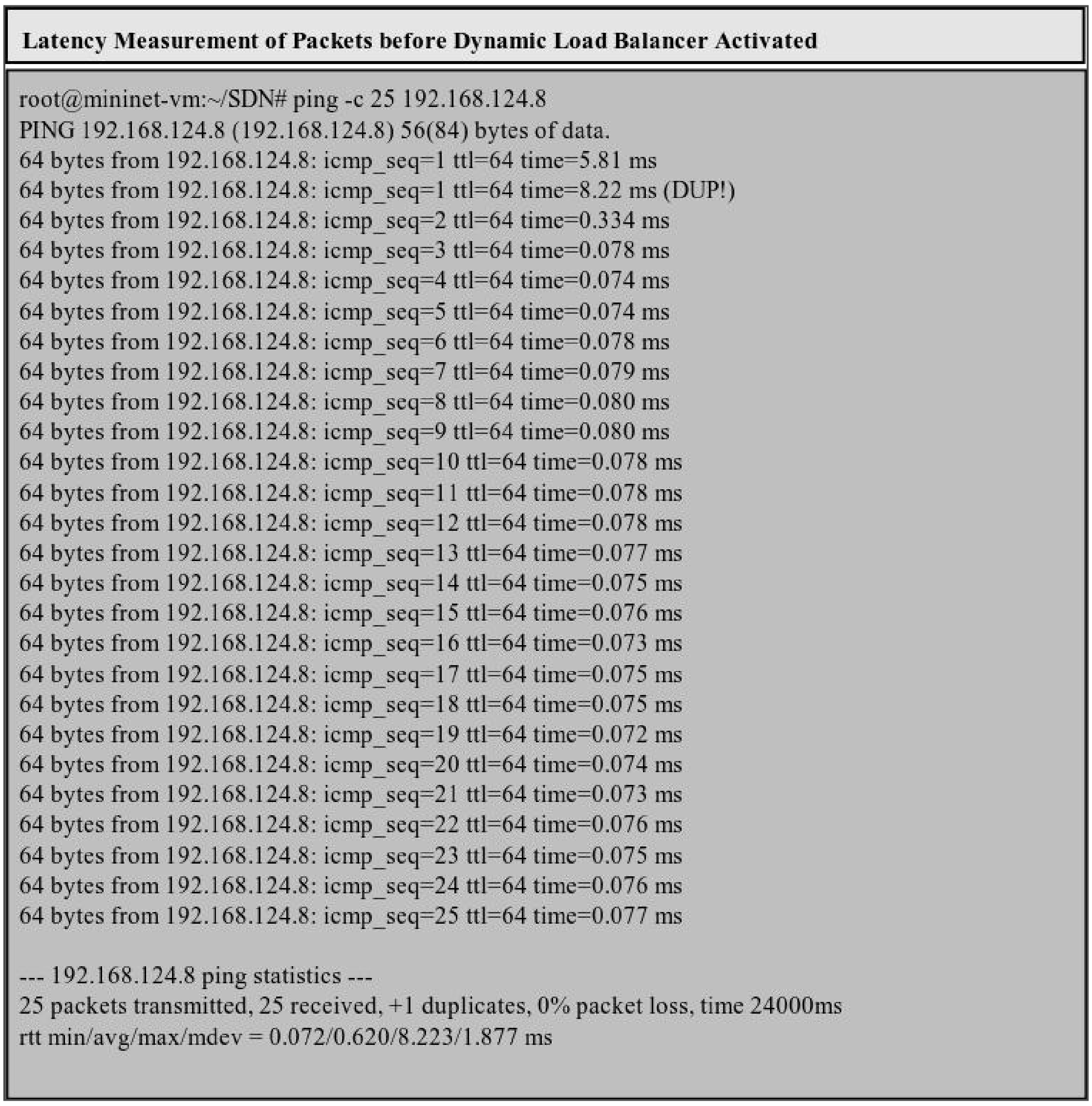
25 Ping tests have been performed on 25 packets/each test from h1 to h4 after load balancing. For the first observation , 25 packets are sent from node h1 to node h4. A ping test is simply a way for source to send a small packet to the destination and to measure the amount of time it takes to get there. The time = 23999 ms at the end of each ping reply is the measured time that last packet took to get to the destination. In this case 23999 ms. Similarly for the second observation, 25 packets are sent from node h1 to node h4. Ping statistics is found to be as rtt min / avg / max / mdev = 0.047 / 0.071 / 0.111 / 0.014 ms. This is recorded in Table 6. Rest 23 observations are performed similar to observation 1 and their results are recorded accordingly in the Table 6.

The average rtt min/avg/max/mdev for ping from h1 to h4 after load balancing in SDN is given by 0.04282/0.07472/0.21284/0.03364 ms where each observation includes 25 packets sent from source h1 to destination h4.

Table 6 : Ping h1 to h4 after Load Balancing (in ms)

Observation	**Min**	**Avg**	**Max**	**Mdev**
1	0.047	0.081	0.320	0.053
2	0.047	0.071	0.111	0.014
3	0.049	0.072	0.095	0.009
4	0.045	0.082	0.278	0.071
5	0.050	0.067	0.152	0.018
6	0.046	0.073	0.217	0.019
7	0.046	0.074	0.067	0.023
8	0.034	0.076	0.153	0.013
9	0.034	0.071	0.276	0.071
10	0.047	0.072	0.251	0.081
11	0.043	0.073	0.155	0.021
12	0.044	0.077	0.251	0.009
13	0.047	0.078	0.257	0.071
14	0.043	0.073	0.218	0.018
15	0.037	0.076	0.267	0.013
16	0.038	0.075	0.289	0.071
17	0.037	0.079	0.167	0.021
18	0.039	0.077	0.199	0.027
19	0.034	0.072	0.201	0.028
20	0.041	0.073	0.221	0.001
21	0.044	0.077	0.205	0.037
22	0.047	0.078	0.276	0.049
23	0.043	0.072	0.251	0.019
24	0.044	0.082	0.155	0.013
25	0.047	0.067	0.289	0.071
	Average: 0.042	**Average: 0.074**	**Average:0.21**	**Average:0.03**

Again, consider the situation to ping node h8 [ip=192.168.124.8] from node h3 [ip=192.168.124.3]. The path from node h3 to node h8 is longer than the path from node h1 to h4. This example is considered in order to validate the result of dynamic load balancer for the improvement of latency of packets in the overall SDN network.



Latency Measurement of Packets before Dynamic Load Balancer Activated

```
root@mininet-vm:~/SDN# ping -c 25 192.168.124.8
PING 192.168.124.8 (192.168.124.8) 56(84) bytes of data.
64 bytes from 192.168.124.8: icmp_seq=1 ttl=64 time=5.81 ms
64 bytes from 192.168.124.8: icmp_seq=1 ttl=64 time=8.22 ms (DUP!)
64 bytes from 192.168.124.8: icmp_seq=2 ttl=64 time=0.334 ms
64 bytes from 192.168.124.8: icmp_seq=3 ttl=64 time=0.078 ms
64 bytes from 192.168.124.8: icmp_seq=4 ttl=64 time=0.074 ms
64 bytes from 192.168.124.8: icmp_seq=5 ttl=64 time=0.074 ms
64 bytes from 192.168.124.8: icmp_seq=6 ttl=64 time=0.078 ms
64 bytes from 192.168.124.8: icmp_seq=7 ttl=64 time=0.079 ms
64 bytes from 192.168.124.8: icmp_seq=8 ttl=64 time=0.080 ms
64 bytes from 192.168.124.8: icmp_seq=9 ttl=64 time=0.080 ms
64 bytes from 192.168.124.8: icmp_seq=10 ttl=64 time=0.078 ms
64 bytes from 192.168.124.8: icmp_seq=11 ttl=64 time=0.078 ms
64 bytes from 192.168.124.8: icmp_seq=12 ttl=64 time=0.078 ms
64 bytes from 192.168.124.8: icmp_seq=13 ttl=64 time=0.077 ms
64 bytes from 192.168.124.8: icmp_seq=14 ttl=64 time=0.075 ms
64 bytes from 192.168.124.8: icmp_seq=15 ttl=64 time=0.076 ms
64 bytes from 192.168.124.8: icmp_seq=16 ttl=64 time=0.073 ms
64 bytes from 192.168.124.8: icmp_seq=17 ttl=64 time=0.075 ms
64 bytes from 192.168.124.8: icmp_seq=18 ttl=64 time=0.075 ms
64 bytes from 192.168.124.8: icmp_seq=19 ttl=64 time=0.072 ms
64 bytes from 192.168.124.8: icmp_seq=20 ttl=64 time=0.074 ms
64 bytes from 192.168.124.8: icmp_seq=21 ttl=64 time=0.073 ms
64 bytes from 192.168.124.8: icmp_seq=22 ttl=64 time=0.076 ms
64 bytes from 192.168.124.8: icmp_seq=23 ttl=64 time=0.075 ms
64 bytes from 192.168.124.8: icmp_seq=24 ttl=64 time=0.076 ms
64 bytes from 192.168.124.8: icmp_seq=25 ttl=64 time=0.077 ms

--- 192.168.124.8 ping statistics ---
25 packets transmitted, 25 received, +1 duplicates, 0% packet loss, time 24000ms
rtt min/avg/max/mdev = 0.072/0.620/8.223/1.877 ms
```

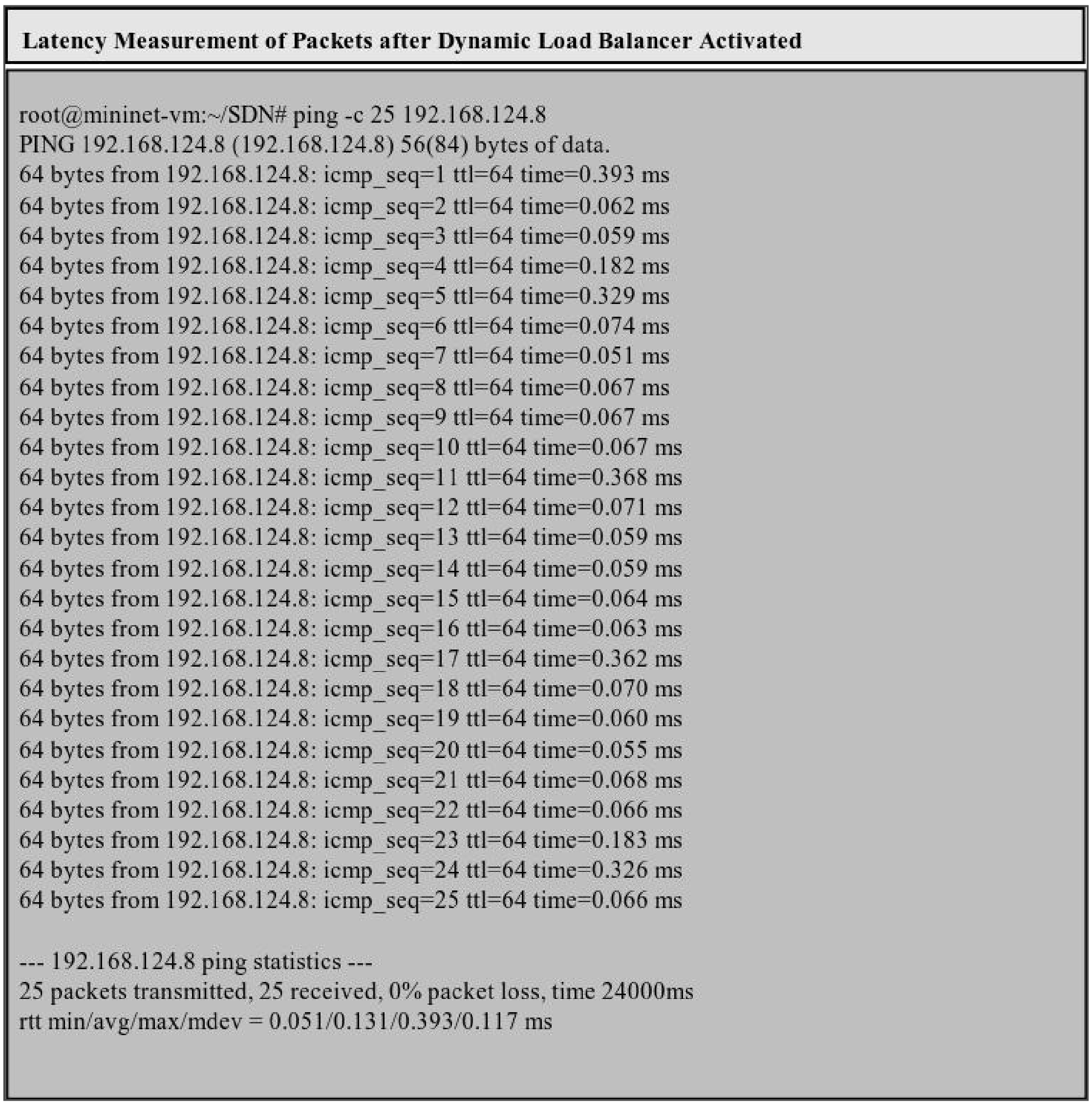
Figure 44: Latency measurement of packets for h3 ping h8 before balancing

Ping statistics from node h3 to h8 before load balancing in shown in Figure 44.

Table 7 : Ping h3 to h8 before Load Balancing (in ms)

Min	Avg	Max	Mdev
0.072	0.620	8.233	1.877
0.052	0.624	8.033	1.645
0.037	0.661	8.098	1.019
0.052	0.534	7.89	1.956
0.042	0.687	8.078	1.579
Average:0.051	**Average:0.6252**	**Average:8.0664**	**Average:1.6152**

The latency measurement of packets from node h8 to node h8 after dynamic load balancer implementation is shown in Figure 45.


Latency Measurement of Packets after Dynamic Load Balancer Activated

```
root@mininet-vm:~/SDN# ping -c 25 192.168.124.8
PING 192.168.124.8 (192.168.124.8) 56(84) bytes of data.
64 bytes from 192.168.124.8: icmp_seq=1 ttl=64 time=0.393 ms
64 bytes from 192.168.124.8: icmp_seq=2 ttl=64 time=0.062 ms
64 bytes from 192.168.124.8: icmp_seq=3 ttl=64 time=0.059 ms
64 bytes from 192.168.124.8: icmp_seq=4 ttl=64 time=0.182 ms
64 bytes from 192.168.124.8: icmp_seq=5 ttl=64 time=0.329 ms
64 bytes from 192.168.124.8: icmp_seq=6 ttl=64 time=0.074 ms
64 bytes from 192.168.124.8: icmp_seq=7 ttl=64 time=0.051 ms
64 bytes from 192.168.124.8: icmp_seq=8 ttl=64 time=0.067 ms
64 bytes from 192.168.124.8: icmp_seq=9 ttl=64 time=0.067 ms
64 bytes from 192.168.124.8: icmp_seq=10 ttl=64 time=0.067 ms
64 bytes from 192.168.124.8: icmp_seq=11 ttl=64 time=0.368 ms
64 bytes from 192.168.124.8: icmp_seq=12 ttl=64 time=0.071 ms
64 bytes from 192.168.124.8: icmp_seq=13 ttl=64 time=0.059 ms
64 bytes from 192.168.124.8: icmp_seq=14 ttl=64 time=0.059 ms
64 bytes from 192.168.124.8: icmp_seq=15 ttl=64 time=0.064 ms
64 bytes from 192.168.124.8: icmp_seq=16 ttl=64 time=0.063 ms
64 bytes from 192.168.124.8: icmp_seq=17 ttl=64 time=0.362 ms
64 bytes from 192.168.124.8: icmp_seq=18 ttl=64 time=0.070 ms
64 bytes from 192.168.124.8: icmp_seq=19 ttl=64 time=0.060 ms
64 bytes from 192.168.124.8: icmp_seq=20 ttl=64 time=0.055 ms
64 bytes from 192.168.124.8: icmp_seq=21 ttl=64 time=0.068 ms
64 bytes from 192.168.124.8: icmp_seq=22 ttl=64 time=0.066 ms
64 bytes from 192.168.124.8: icmp_seq=23 ttl=64 time=0.183 ms
64 bytes from 192.168.124.8: icmp_seq=24 ttl=64 time=0.326 ms
64 bytes from 192.168.124.8: icmp_seq=25 ttl=64 time=0.066 ms

--- 192.168.124.8 ping statistics ---
25 packets transmitted, 25 received, 0% packet loss, time 24000ms
rtt min/avg/max/mdev = 0.051/0.131/0.393/0.117 ms
```

Figure 45: Latency measurement of packets for h3 ping h8 before balancing

The Table 8 shows decrease in packets transmission latency after load balancing.

Table 8: Ping h3 to h8 after Load Balancing (in ms)

Min	Avg	Max	Mdev
0.051	0.131	0.339	0.117
0.047	0.171	0.311	0.114
0.049	0.172	0.395	0.109
0.045	0.166	0.381	0.111
0.050	0.167	0.352	0.118
Average: 0.0484	**Average: 0.1614**	**Average: 0.3556**	**Average: 0.1138**

The Figure 46 and Figure 47 show data transfer load balancing and after load balancing in SDN and average latency before load balancing and after load balancing in SDN.

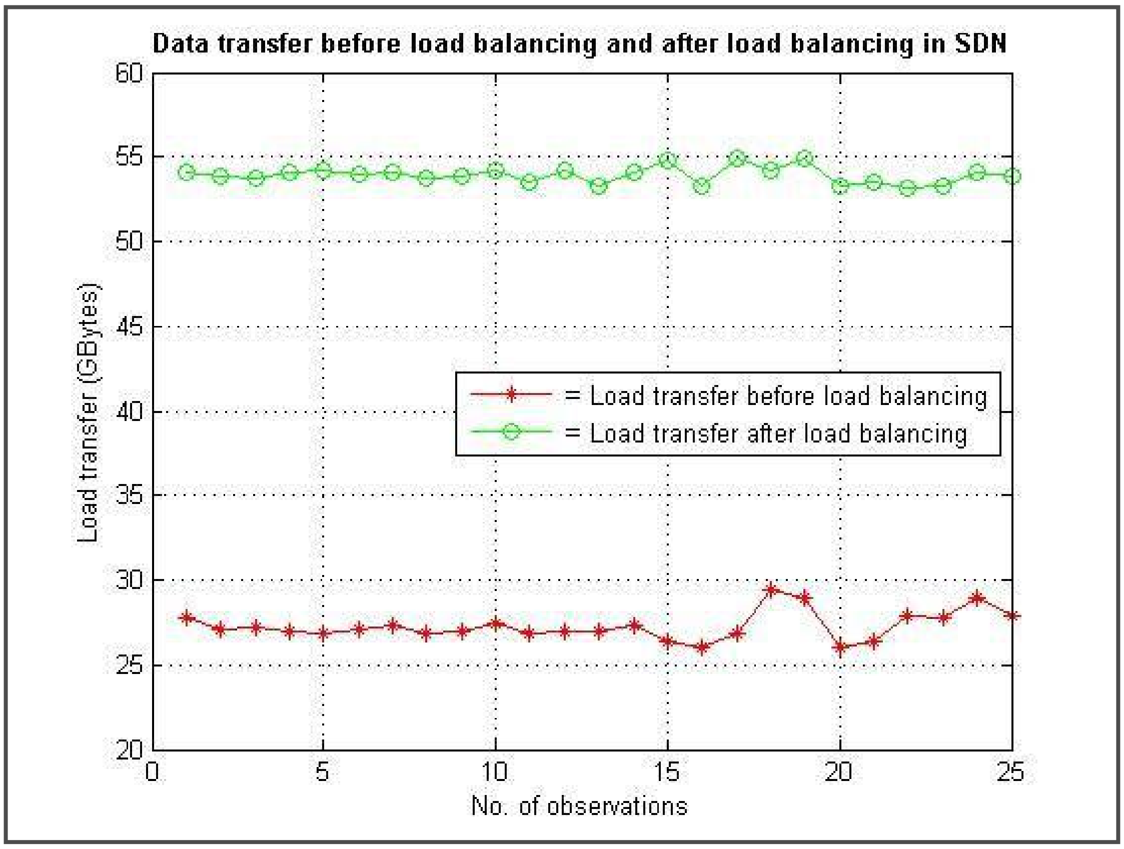


Figure 46: Data transfer before load balancing and after load balancing in SDN

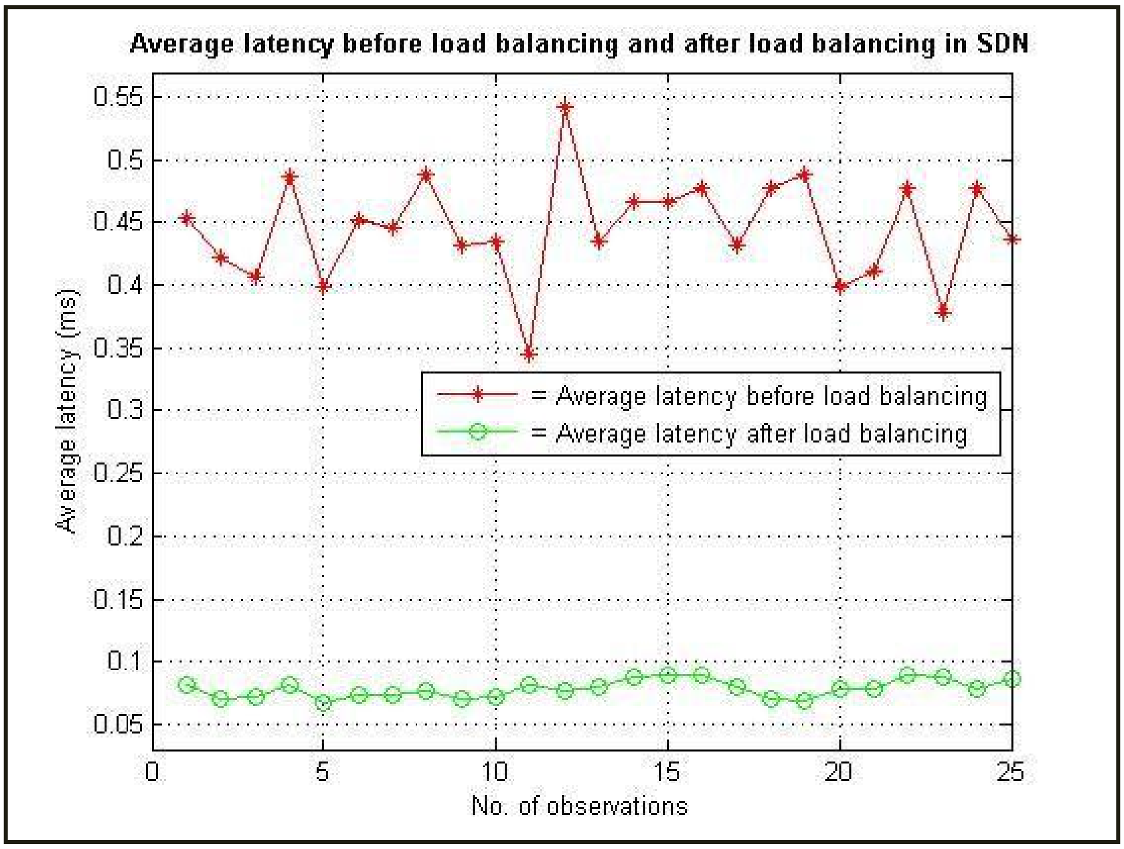


Figure 47: Average latency before load balancing and after load balancing in SDN

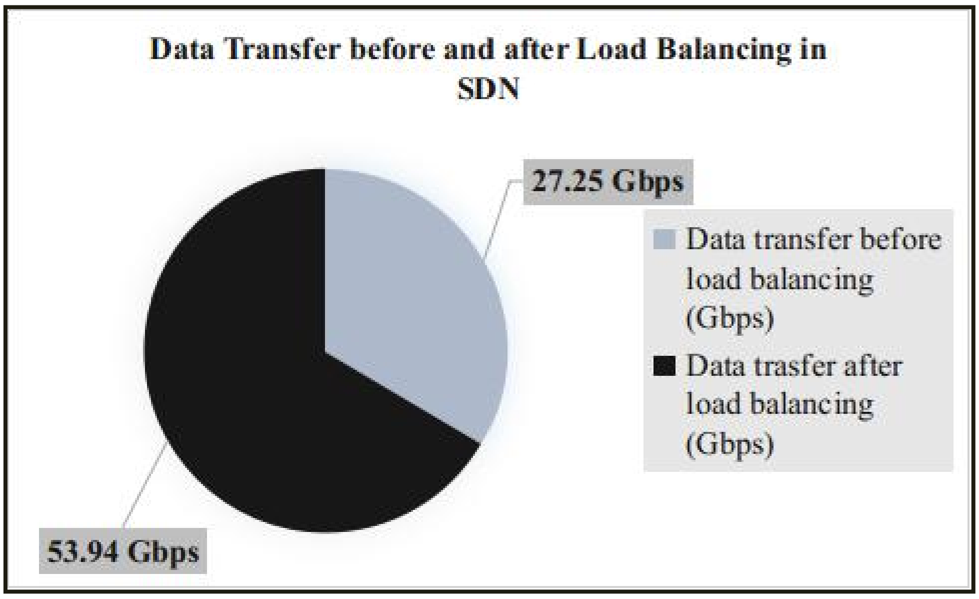


Figure 48: PI-Chart Representation for average data transfer before and after load balancing in SDN

The average data transfer before the implementation of load balancer is 27.25 Gbps. After the implementation of dynamic load balancer, average data transfer is 53.94 which is shown in Figure 48.

The average latency before the implementation of load balancer is 0.074 ms. After the implementation of dynamic load balancer, average latency is 0.445 ms which is shown in Figure 49.

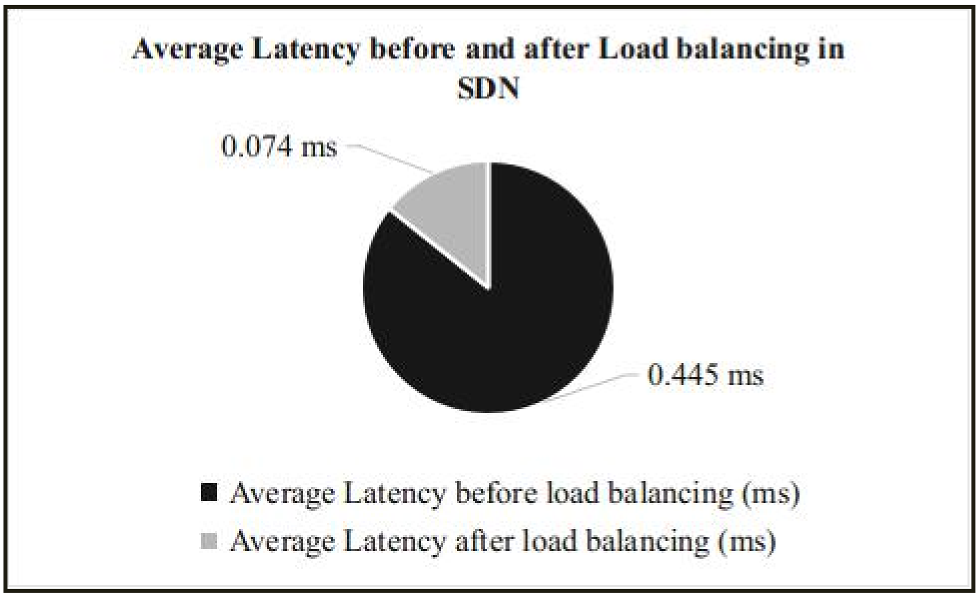


Figure 49: PI-Chart Representation for average latency before and after load balancing in SDN

The Figure 48 and Figure 49 show the effectiveness of implementing the load balancer.

CHAPTER 7: EPILOGUES

7.1 Conclusion

The dynamic load balancer which can balance server load as well as network path loads using SDN controller is successfully implemented. The implementation of the dynamic load balancer in SDN efficiently utilizes available link’s load as well as server’s load.

7.2 Future Enhancements

The thesis work can further be extended by investigating machine learning approach for load balancing. Also in future, load balancing can be evaluated by considering more parameters like response time of server and high availability of computing system.

REFERENCES

1. N. Mckeown, S. Shenker, T. Anderson, L. Peterson, J. Turner, H. Balakrishnan, J. Re. "Openflow: Enabling Innovation In Campus Networks.", Acm Sigcomm Computer Communication Review38.2, (2008), pp. 69-74.
2. T. Wing Chim, and Y. K.L. "Traffic distribution over equal-cost-multi-paths.", IEEE International Conference on Communications IEEE, (2004), pp. 465–475.
3. H. Nikhil, et al. "Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow." Acm Sigcomm Demo (2009).
4. R. Wang, D. Butnariu, and J. Rexford. "OpenFlow-based server load balancing gone wild", Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services USENIX Association, (2011), pp. 12-12.
5. Y. Hu, W. Wang, X. Gong, X. Que, & S. Cheng, "BalanceFlow: Controller load balancing for OpenFlow networks.", Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on IEEE, (2012), pp. 780-785.
6. Y. Li, D. Pan. "OpenFlow based load balancing for Fat-Tree networks with multipath support", Proc.12th IEEE International Conference on Communications (ICC'13), Budapest, Hungary. (2013), pp. 1-5.
7. S. Bhandarkar, K. Khan, "Load Balancing in Software Defined Network Based on Traffic Volume.", Advances in Computer Science and Information Technology (ACSIT), (2015), pp. 72-76.
8. Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing Tables in software-defined networks," Technion, Tech. Rep. TR12-05, 2012.
9. Yossi Kanizo, David Hay, and Isaac Keslassy. Palette: Distributing Tables in software-defined networks. In INFOCOM, pages 545–549,2013.
10. DAI, Wei, Guochu SHOU, Yihong HU, and Zhigang GUO. "R-SDN: A RECUSIVE APPROACH FOR SCALING SDN."
11. Shi, Lei, Bin Liu, Changhua Sun, Zhengyu Yin, Laxmi Bhuyan, and H. Jonathan Chao. "Load-balancing multipath switching system with flow slice." Computers,

IEEE Transactions on 61,no. 3 (2012): 350-365.

12. Prete, Luca, Fabio Farina, Mauro Campanella, and Andrea Biancini. "Energy efficient minimum spanning tree in OpenFlow networks." In Software Defined Networking (EWSDN), 2012 European Workshop on, pp. 36-41. IEEE, 2012.

13. Dixit, Advait, Fang Hao, Sarit Mukherjee, T. V. Lakshman, and Ramana Kompella. "Towards an elastic distributed SDN controller." In ACM SIGCOMM Computer Communication Review, vol. 43, no. 4, pp. 7-12. ACM, 2013.

14. Kim, Hyojoon, J. R. Santos, Y. Turner, M. Schlansker, J.Tourrilhes, and Nick Feamster. "Coronet: Fault tolerance for software defined networks." In Network Protocols (ICNP), 2012 20th IEEE International Conference on, pp. 1-2. IEEE, 2012.

15. Long, Hui, Yao Shen, Minyi Guo, and Feilong Tang."LABERIO: Dynamic load-balanced routing in OpenFlow-enabled networks." In Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on, pp. 290-297. IEEE, 2013.

16. Hata, Hiroaki. "A study of requirements for sdn switch platform." In Intelligent Signal Processing and Communications Systems (ISPACS), 2013 International Symposium on, pp. 79-84.IEEE, 2013.

17. Y. Zhou, M. Zhu, L. Xiao, Li Ruan, W. Duan, D. Li, R. Liu, “A Load Balancing Strategy for SDN Controller based on Distributed Decision.”, IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications, (2014).

18. J. Li, X. Chang, Y. Ren, Z. Zhang, & G. Wang, "An Effective Path Load Balancing Mechanism Based on SDN." Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on IEEE, (2014), pp. 527-533.

19. https://www.wireshark.org/ (22nd August, 2016)

20. https://iperf.fr/ (27th August, 2016)

21. https://www.java.com/ (27th August,2016)

22. https://www.python.org/ (29th August,2016)