TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

THESIS NO: 072/MSI/601

IP-Based Hashing Load Balancer in Software Defined Networking

By
Anup Bhattarai

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

NOVEMBER, 2019

IP-Based Hashing Load Balancer in Software Defined Networking

by

Anup Bhattarai

072/MSI/601

Thesis Supervisor

Baburam Dawadi

A thesis submitted in partial fulfillment of the requirements 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

November, 2019

COPYRIGHT©

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS, PULCHOWK

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

The undersigned certify that they have read, and recommended to the Institute of Engineering for acceptance, a thesis report entitled **"IP-Based Hashing Load Balancer in Software Defined Networking"** submitted by **Mr. Anup Bhattarai** in partial fulfillment of the requirement for the degree of **"Master of Science in Information and Communication Engineering"**.

…………………………………………………………………….

Supervisor, Baburam Dawadi

Lecturer

Department of Electronics and Computer Engineering

…………………………………………………………………….

External Examiner, Adesh Khadka

IT Director

Ministry of Finance, Government of Nepal

…………………………………………………………………….

Dr. Basanta Raj Joshi

Program Coordinator

Master of Science in Information and Communication Engineering

Department of Electronics and Computer Engineering

Date: 11/22/2019

DEPARTMENT ACCEPTANCE

The thesis entitled "**IP-Based Hashing Load Balancer in Software Defined Networking**" submitted by **Mr. Anup Bhattarai** 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. Surendra Shrestha

Head of the Department

Department of Electronics and Computer Engineering,

Pulchowk Campus,

Institute of Engineering,

Tribhuvan University,

Nepal.

ACKNOWLEDGEMENT

First and foremost, I would like to express my sincere gratitude to my supervisor **Baburam Dawadi** for his support, valuable suggestions and guidance for this thesis. I would like to show my greatest appreciation to the Department of Electronics and Computer Engineering, Institute of Engineering, Pulchowk Campus for providing me the platform and allowing me to work in this thesis. I would also like to thank to our Head of Department **Dr Surendra Shrestha** and Program Coordinator **Dr. Basanta Raj Joshi** for their support, guidance and providing me the platform to execute this thesis.

A special thank goes to **Prof. Dr. Sashidhar Ram Joshi**, **Prof. Dr. Subarna Shakya, Dr. Diwakar Raj Pant**, **Dr. Sanjeeb Prasad Pandey**, and all my respected Teachers for their encouragement, valuable suggestion and moral support.

Likewise, I would also like to thank my friends, colleagues for their support, motivation and encouragement regarding the project. I am grateful for their cooperation during the period of my thesis. Finally, I would like to thank all the people who are directly or indirectly related for completing this thesis.

ABSTRACT

Software-Defined Networking (SDN) is a new principle in the networking paradigm where control and management are centralized and decoupled from data plane, thus making the network programmable and uses open interfaces between the devices in the control plane (controllers) and those in the data plane. Load balancer is a system that distributes network or application traffic across a cluster of servers depending upon load balancing strategy. Here, IP hash-based load balancer algorithm has been deployed over SDN Framework and its performance over other load balancing algorithms; round robin, weighted round robin were evaluated and compared using HTTP server client model with exchange of different file sizes samples and standard dataset files. Further, the implemented model was evaluated using opensource network evaluation tool iPerf. The result showed that the performance of IP based Hash algorithm was observed to be slightly better among other in Software defined network.

Keywords— Software Defined Network, Load Balancer, round robin, weighted round robin, IP hash

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATION

API	Application Programming Interface
ARP	Address Resolution Protocol
CLI	Command Line Interface
GUI	Graphical User Interface
HTTP	Hyper Text Transport Protocol
IP	Internet Protocol
NETCONF	Network Configuration
NOS	Network Operating System
ONF	Open Networking Foundation
PPS	Packet Per Second
RR	Round Robin
SDN	Software Defined Networking
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VIP	Virtual IP
YANG	Yet Another Next Generation

CHAPTER ONE: INTRODUCTION

1.1 Background and Motivation

Software defined networking is an emerging topic in recent years. With concept to provide user-controlled management of forwarding in network nodes. SDN is an architecture purporting to be dynamic, manageable, cost-effective and adaptable, seeking to be suitable for the high bandwidth, dynamic nature of today's application. Open Networking Foundation (ONF) defines SDN as "In the SDN architecture, the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications." [1].SDN focuses on four key features:

- Separation of the control plane from the data plane.
- A centralized controller and view of the network.
- Open interfaces between the devices in the control plane (controllers) and those in the data plane.
- Programmability of the network by external applications.

Software Defined Networking (SDN) is a new approach to facilitate network management that has been gaining widespread attention. Traditional networking involves buying the hardware and software together from a vendor and configuring them as per the requirements. In traditional networking, it is not possible to separate the data plane from the control plane. Therefore, the customer had to rely on the vendor for software bug fixes, additional features, and licenses. It is not possible for the customer to modify the vendor software code as per needs of the network. The proprietary nature of most vendor software leads to slowed innovation and increased delay in deployment of new services. SDN tries to eliminate this vendor dependence and is a paradigm shift from closed vendor-specific networking to an open networking system. In SDN, a centralized controller deploys network flows in the bare-metal network switches. Therefore, the hardware for networking devices is available at a lower price and the software configuration can be modified as per needs of the customers. This generic piece of hardware can be programmed using an Application Programmable Interface (API) or installation of any Network Operating System (NOS). Some of the protocols used to network configuration are OpenFlow and NETCONF. OpenFlow enables the SDN controller to communicate with networking devices. Similarly, NETCONF protocol enables network configuration and automation using yet another Next Generation (YANG) as a data modeling language. The unbundling of hardware and software considerably reduces costs as networks grow. Through the use of unbundling, the customer can program the flows as per requirements of the network. [2]

Web browsing utilizes a client-server model where a client requests data from a server and the server responds.[2] Depending upon the type of service delivered, there can be millions of connections to the server at any time. In this case, it is necessary to ensure that the critical services like money transfer, online examinations, and business transactions are reliably delivered. A load balancer is a device which serves a key role in delivering these services and keeping the network infrastructure running.

1.2 Organization of The Thesis

This Thesis implements a load balancer in Software Defined Network using IP Hash based algorithm. The Thesis report is organized and presented Chapter wise.

Chapter I introduces with some background and motivation introduction with some description on problem statement and introduces the objective of this thesis.

Chapter II is regarding the Literature Review where related Theory of this Thesis including SDN and its traffic flow, Load balancer are explained. Further related works in SDN environment and research gap are further described in this chapter.

Chapter III includes the Methodology of the Thesis where the proposed Framework, Algorithm, Flowchart and Calculated are describer. Further Different Tools used and the data collection source are explained.

Chapter IV gives the result of the experiment done in this research and the discussion of the result.

Chapter V is regarding the limitation of this research. Chapter VI gives the conclusion and Future Recommendation. References section list all the reference sources used for the thesis preparation. In Annex all the experimental output and capture of experiment are listed along with the codes used in this research.

1.3 Problem Statement

Several companies manufacture dedicated load balancers like F5 Networks, A10 Networks, Kemp technologies, and Barracuda Networks. Unfortunately, the cost of buying a dedicated load balancer could be as high as $30,000 per device. These costs can be a financial constraint for companies in managing the network. Additional expenses are imposed in terms of receiving support and hiring trained professionals for the vendor specific equipment and further network administrator must reply on the vendor for software bug fixes, release of new features and standardization of protocols. Another problem in buying dedicated service networking equipment is that it does not offer flexibility in terms of configuration.

Similarly, there are some cases where user's session persistence is preferred like for case of shopping cart application where items in user's cart might be stored in browser level until user is ready to purchase them or can be case when an upstream server stores information requested by a user in its cache to boost performance. Changing which server received request from the client in middle of session may cause performance issue or transaction failure, repeated information fetch creating performance inefficiencies.

Open source controller, such as POX is used to deploy network flows in SDN framework and mitigate a load balancer offering flexibility and reduce constrain of traditional networking. Using IP hash-based load balancing approach user session can be preferred to redirect the user traffic to same servers for unique host.

1.4 Objectives

The objectives of this Thesis are:

1. To implement an IP based Hashing Load Balancer over Software Defined Networking environment.
2. To evaluate its performance and compare with round robin-based Load Balancer, weighted round robin-based Load Balancer.

CHAPTER TWO: LITERATURE REVIEW

2.1 Related Theory

2.1.1 Software Defined Network

SDN is an approach to computer networking that allows administrator to manage network service through abstraction of higher-level functionality.

SDN focuses on four key features:

- Separation of the control plane from the data plane.
- A centralized controller and view of the network.
- Open interfaces between the devices in the control plane (controllers) and those in the data plane.
- Programmability of the network by external applications.[3]

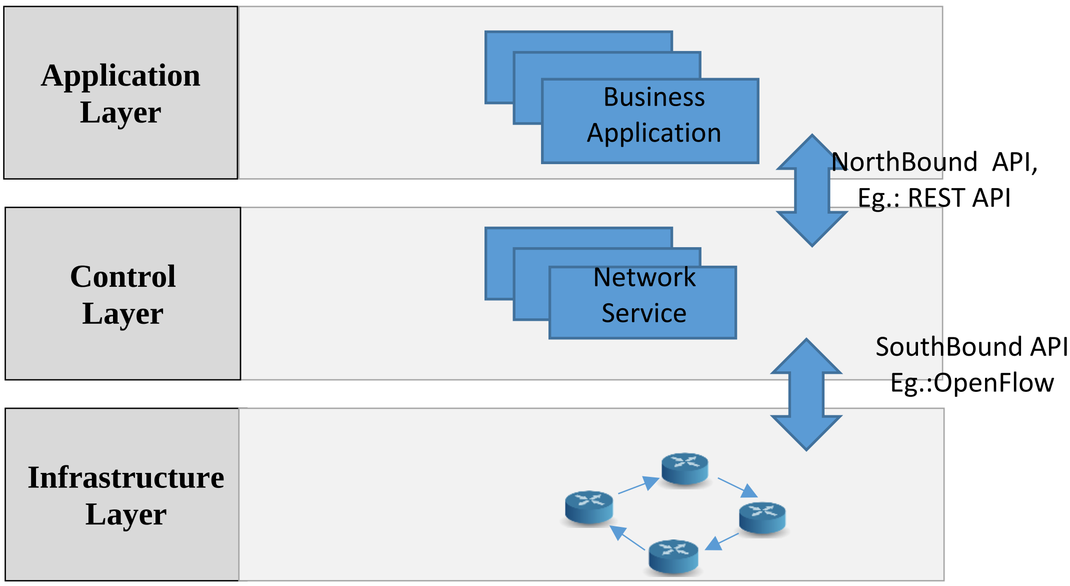

Figure 1 : Software Defined Networking Model

In the Figure 1, the lower part which is Infrastructure layer contains network equipment like router; switch etc. that forms data plans. The central tier is having controllers that manage data flows and define path in the network. Central tier is connected with the other tires using Application Programming Interface (API). The control plan communicates with the data plan through Open Flow protocol. Open Flow switch contain one or more rule tables, according to that rules action is perform to the traffics like drop, foreword or flood. Top of the architecture is called Application tier that manage network application like monitoring, access control and service provided by operators etc.

In Software Defined Networking (SDN), Northbound and Southbound APIs are used to describe how interfaces operate between the different planes - data plane, control plane and application plane.[4]

Southbound interface is the protocol specification that enables communication between controllers and switches and other network nodes, which is with the lower-level components. This further lets the router to identify network topology, determine network flows and implement request sent to it via northbound interfaces. Southbound interfaces define the way the SDN controller should interact with the data plane (also known as forwarding plane) to adjust the network, so it can better adapt to changing requirements. Southbound APIs allows the end-user to gain better control over the network and promotes the efficiency level of the SDN controller to evolve based on real-time demands and needs. In addition, the interface is an industry standard that justifies the ideal approach the SDN controller should communicate with the forwarding plane to modify the networks that would let it progressively move along with the advancing enterprise needs. To compose a more responsive network layer to real-time traffic demands, the administrators can add or remove entries to the internal flow-table of network switches and routers. Some of the popular southbound APIs are OpenFlow, Cisco, and OpFlex and other switch and router vendors that support OpenFlow include IBM, Dell, Juniper, Arista and more. OpenFlow is a well-known southbound interface.[5] With OpenFlow, entries can be added and removed to the internal flow-table of switches and potentially routers to make the network more responsive to real-time traffic demands.

Northbound interfaces define the way the SDN controller should interact with the application plane. Applications and services are things like load-balancers, firewalls, security services and cloud resources. Contradictory to southbound API, northbound interfaces allows communication among the higher-level components. While the traditional networks use firewall or load balancer to control data plane behavior. SDN installs applications that uses the controller and these applications communicate with the controller through its northbound interface. Experts say that it would be rather difficult to enhance the network infrastructure, as without a northbound interface the network applications will have to come directly from equipment vendors, which can make it harder to evolve. In addition, the northbound API makes it easier for network operators to innovate or customize the network controls and processing this task doesn't require help from expertise, as the API can be cleaned by a programmer who excels in programming languages like Java, Python, or Ruby.

2.1.2 SDN Traffic Flow

In SDN, incoming traffic can be classified as followings:

- Packets that do not match a flow table and are coming for the first time. This traffic goes through the controller and
- Packets that are already the part of an existing flow which does not have to go through controller.[6]

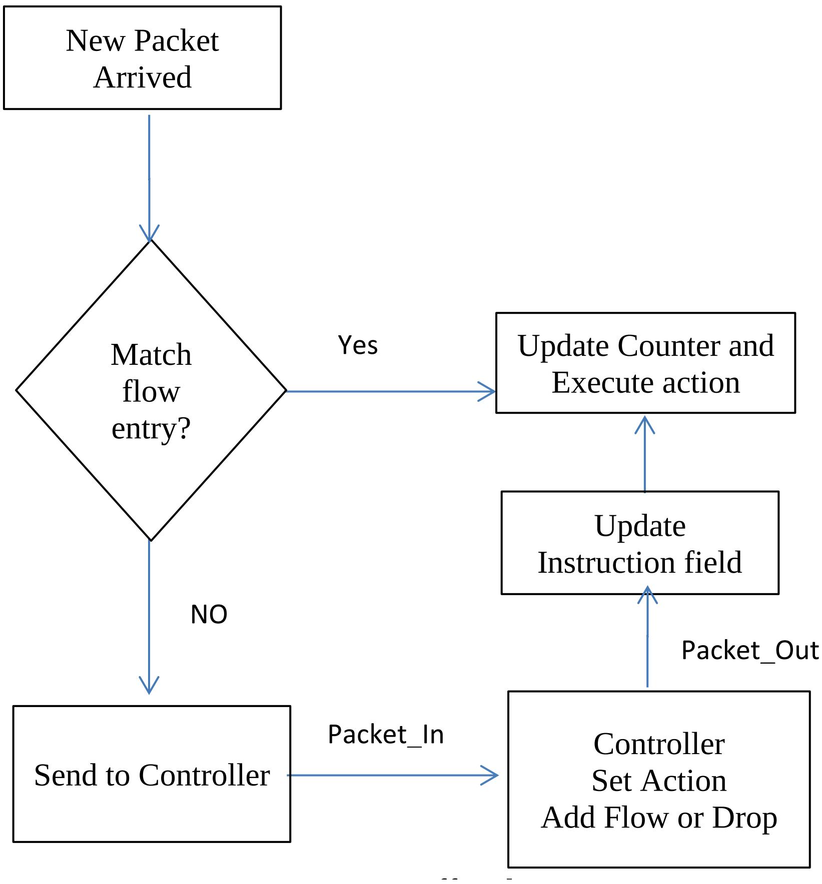

Figure 2 :Traffic Flow in SDN

Figure 2 shows the flowchart of flow of traffic in SDN. The packet that has been initiated for the first time and that has no match in the table goes to the controller. This type of traffic gets inspected by controller and sets an appropriate action. If the action is to add a flow the packet is returned back to the switch and flow is added to the flow table. Simultaneously, counters such as number of packets, number of bytes per flow starts and if flow is inactive timer starts the timeout.

2.1.3 Workflow between Client and Server in SDN

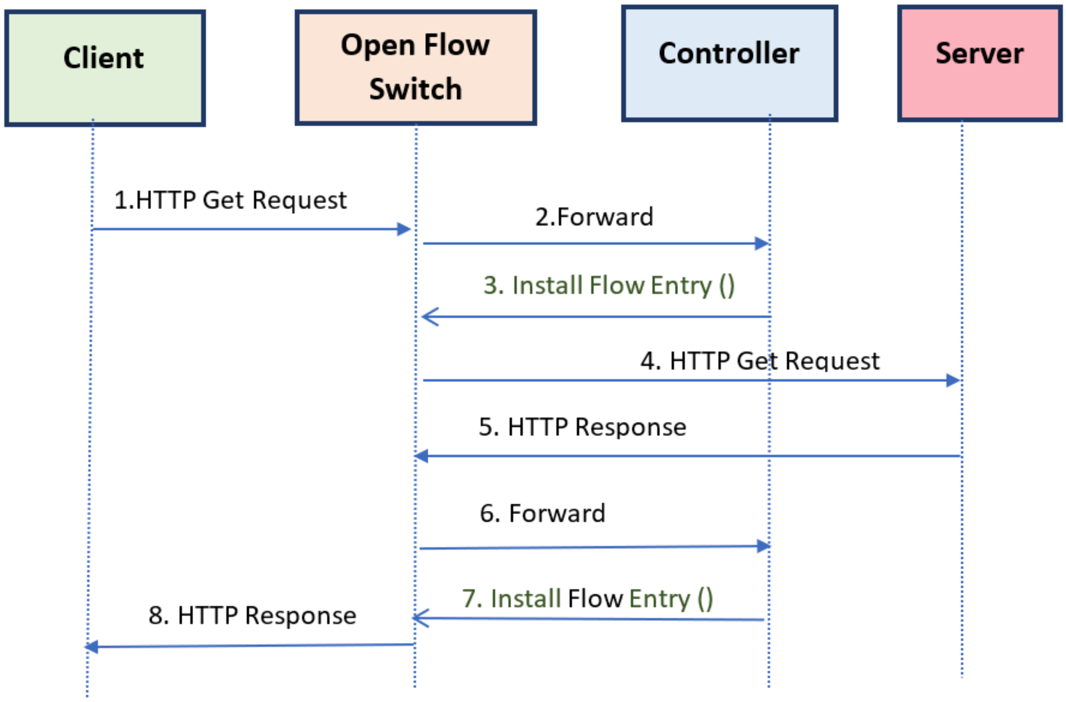

Figure 3 : New flow Arrival (HTTP GET and response)

When a new request from a client arrives at an OpenFlow switch, it does not match any existing flow. It is reported to the controller, which directs the switch to create a flow table entry for the new packet (Fig. 4, step 3). When the server responds to the request from the client, another flow entry is created at the switch, but in the reverse direction[7] The workflow discussed has been implemented on POX, a Python-based SDN controller [8]

The following two standard OpenFlow APIs implemented to cater most of the logic required in the workflow that we discussed above:

- OFPT_PACKET_IN
- OFPT_FLOW_MOD

OFPT_PACKET_IN message is used by the OpenFlow switch to report the arrival of a certain packet. Most typically, it takes place when the packet does not match any flow in the flow table. The reason code is also included in the message. In case of the absence of a matching flow entry, it is OFPR_TABLE_MISS. OFPT_PACKET_IN message is used in steps 2 and 6 of Figure. 4. OFPT_FLOW_MOD message is used in our system in steps steps 3 and 7 of Figure. 4. It is used by the controller to instruct the OpenFlow switch to create, delete, or modify a flow table entry. In each case, the command part of the message is one of the following:

- OFPFC_ADD
- OFPFC_DELETE
- OFPFC_MODIFY

OFPFC_ADD is used when a flow table entry should be created at the flow switch. Obviously, it takes place when a new flow arrives. Normally, OFPFC_DELETE is used when a flow terminates. When the controller wants to modify a flow table entry, they send a flow entry modification message OFPFC_MODIFY. OFPFC_MODIFY is not used in our system

When a flow table entry is made, an action is associated with it, the following actions is used in the system, both of which are mandatory in the OpenFlow standard:

- Output
- Drop

Output is the action to forward the packet. Drop is obviously to drop the given packet. In order to find a matching flow in the OpenFlow API, various fields can be used in the packet, such as IP protocol number, IPv4 addresses, IPv6 addresses, TCP/UDP port numbers, incoming switch port, Ether type, and MAC addresses.

2.1.4 Load Balancer

A load balancer is a device which serves a key role in delivering these services and keeping the network infrastructure running. There are several load balancing algorithms available for the load balancer to distribute traffic among the servers and deployed over conventional network. [9] Some of the different types of load balancing algorithms are round-robin, weighted round-robin, hashing based, and URL based. Further there are different vendor specific load balancer available designed on conventional based architecture and are vendor specific design and lacks flexibility.

A load balancer is a device which serves a key role in delivering these services and keeping the network infrastructure running. The load balancer is installed between the client and the server. The load balancer attempts to distribute client requests among the cluster of web servers efficiently. This allocation helps improve performance and reduce the possibility of a server being overloaded. Certain load balancers also provide failover capabilities. If a server fails, the load balancer provides fault tolerance by directing the requests to the remaining servers.[10]

2.1.4.1 Round Robin based Load Balancer

Round-robin load balancing is one of the simplest methods for distributing client requests across a group of servers. Going down the list of servers in the group, the round-robin load balancer forwards a client request to each server in turn. When it reaches the end of the list, the load balancer loops back and goes down the list again (sends the next request to the first listed server, the one after that to the second server, and so on)[10]

Steps:

- Client sends the request to towards the floating IP(load balancer IP).
- N= no of live server list
- Index=(index+1) mod N
- Destination server IP=server(index)

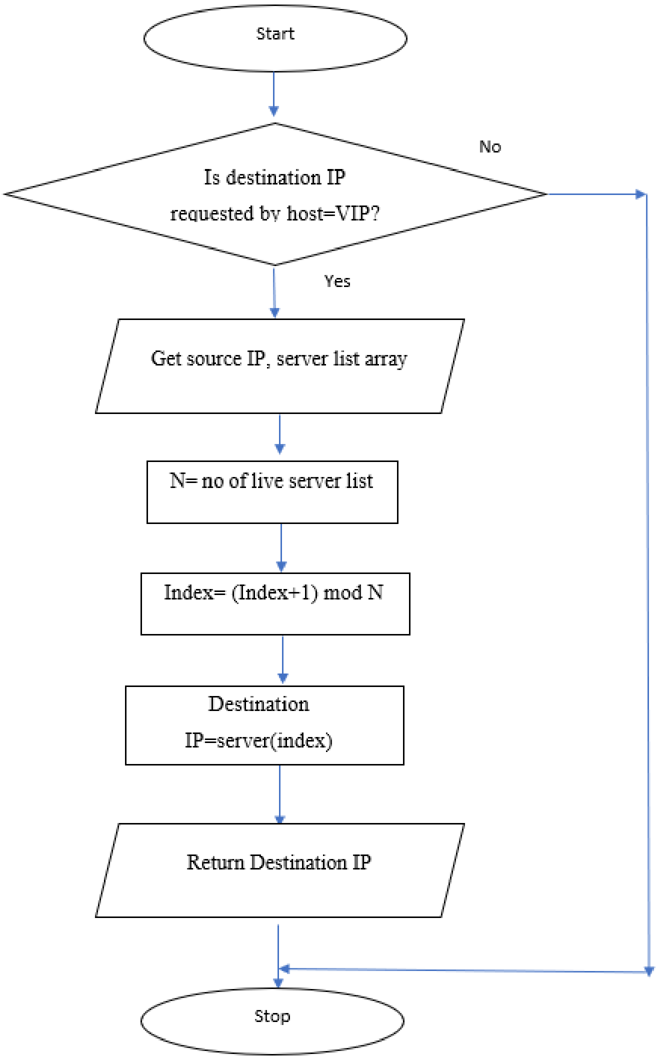

Figure 4 : Flowchart of Load Balancer using Round Robin

2.1.4.2 Weighted Round Robin Load Balancer

A weight is assigned to each server based on criteria chosen by the site administrator; the most commonly used criterion is the server's traffic-handling capacity. The higher the weight, the larger the proportion of client requests the server receives. If, for example, server A is assigned a weight of 2 and server B a weight of 1, the load balancer forwards 2 requests to server A for each 1 it sends to server B

Steps:

```
count = (count + 1) % 4
live_server_list[]=array of server IP
N=length of live_server_list[]
For x in live_server_list[N]
    if count < 2:
        if str(x) == '10.0.0.1':
         ipserver = x
         return ipserver
    elif count == 2:
         if str(x) == '10.0.0.2':
         ipserver = x
         return ipserver
    else:
       if str(x) == '10.0.0.3':
         ipserver = x
          return ipserver
```

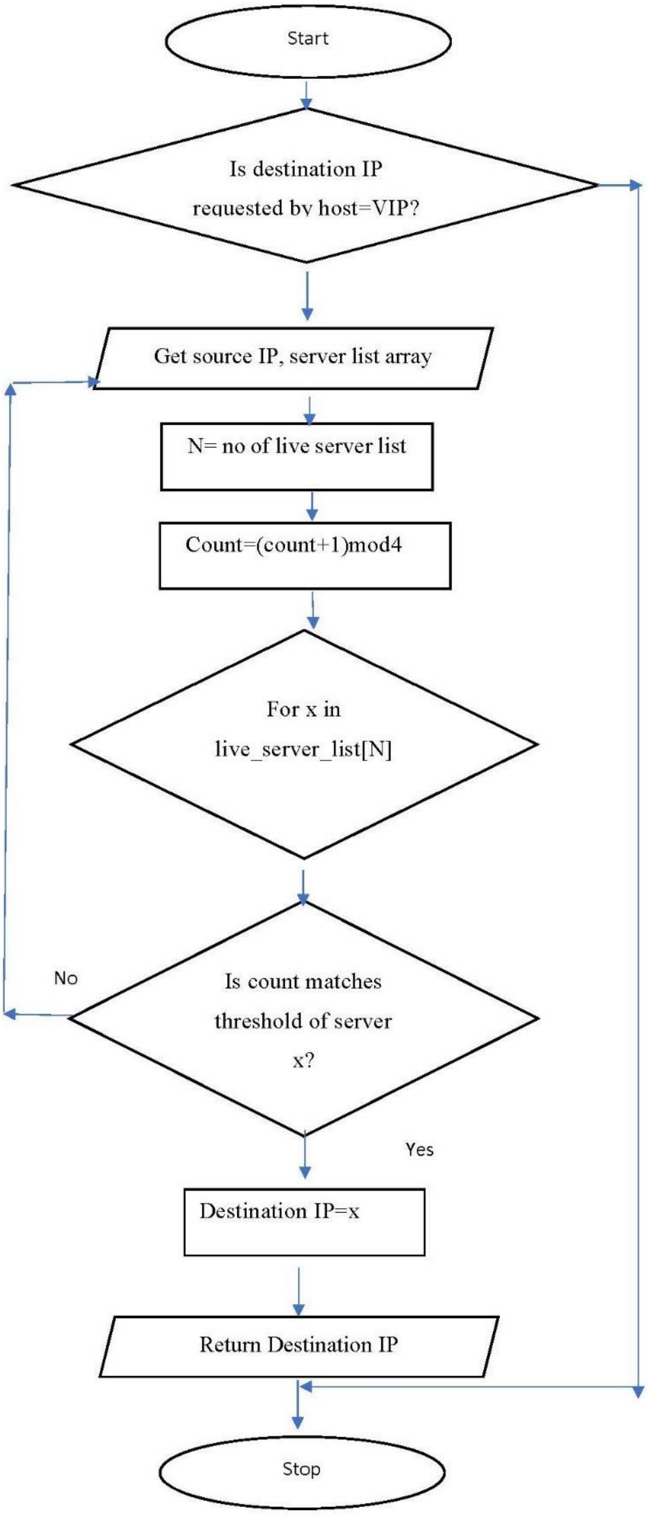

Figure 5 : Flowchart of Load Balancer using weighted Round Robin

2.2 Related works in SDN Environment

Software Defined Networking is not a revolutionary technology, it's an organizing principle in data networks. The rationale behind SDN is more important than its design. SDN allows a programmatic change control platform, which allows the entire network to be managed as a single asset, simplifies the understanding of the network and enables continuous monitoring in more detail. The fundamental shift in networking brought by SDN is the decoupling of the systems that decide where the traffic is sent (i.e. the control plane) from the systems that perform the forwarding of the traffic in the network (i.e. the data plane).

In 2011 the Open Networking Foundation was founded and now it has over 90 companies, among which are Google, Cisco, Dell, IBM, Intel, Facebook, Verizon, Arista, Brocade, etc... Google publicly announced to use SDN for their interconnecting their data centers in 2012.

In SDN architecture, limited researches have been proposed on network load balance.The round-robin load balancing algorithm is popular and simpler than the other algorithms and this was implemented in SDN by kaur.[11]. The round robin load balancing algorithm does not consider any state of the traffic and is classified as the static algorithm. "Traditional round robin uses a circular register and pointer to the last selected server to make dispatching decisions, that is, if Si was the last chosen node, a new request is assigned to Si+1, where i = (i+1) mod N and N is the number of server nodes" . The advantages of the round robin algorithm are the ease of management and its efficient operation. In addition to this, load balancing devices do not need significant memory and CPU resources to perform load balancing compared to other algorithms.

Similarly weighted Round robin, another approach was used by Kenji Hikichi in SDN [12] . and Sabiya, Japinder Singh [13] which calculates the weight of round robin scheduling of an external load balancer to distribute requests submitted to the applications among the servers.

2.3 Load Balancer based on IP Hash Strategy in SDN Framework

During the research, most of papers on load balancing algorithm deployed in SDN were found to be based on round-robin and weighted round robin strategies. Here Hash-based load balancing strategy is selected for implementation over SDN Framework and evaluate its performance in Software Defined Network environment compared to other deployed load balancing strategies in SDN.

Hash-based load balancing is a stateful algorithm being used for traffic load balancing. Ju-Yeon Jo and Kim [14] has explained the Hash based load balancing in Internet traffic. The hash value for a particular traffic flow is calculated based on the source IP address, destination IP address, source port number, destination port number, and URL. The hash value can be a combination of two or more parameters to identify a particular flow. The load balancer forwards the request to the server with the highest hash value, created using a combination of different parameters of packets. Any further requests with the same hash value are sent to the same server and hash values are usually cached for a certain amount of time. This algorithm creates unique hash values for the same type of traffic i.e. requested by the same host.

CHAPTER THREE: METHODOLOGY

3.1 Proposed Framework

The load balancing architecture consists of OpenFlow switch network with a POX controller and multiple servers connected to the ports of the OpenFlow switch. Each server is assigned static IP address and the POX controller maintains a list of live servers that are connected to the OpenFlow switch. Web service is running on each server on a well-known port 80.

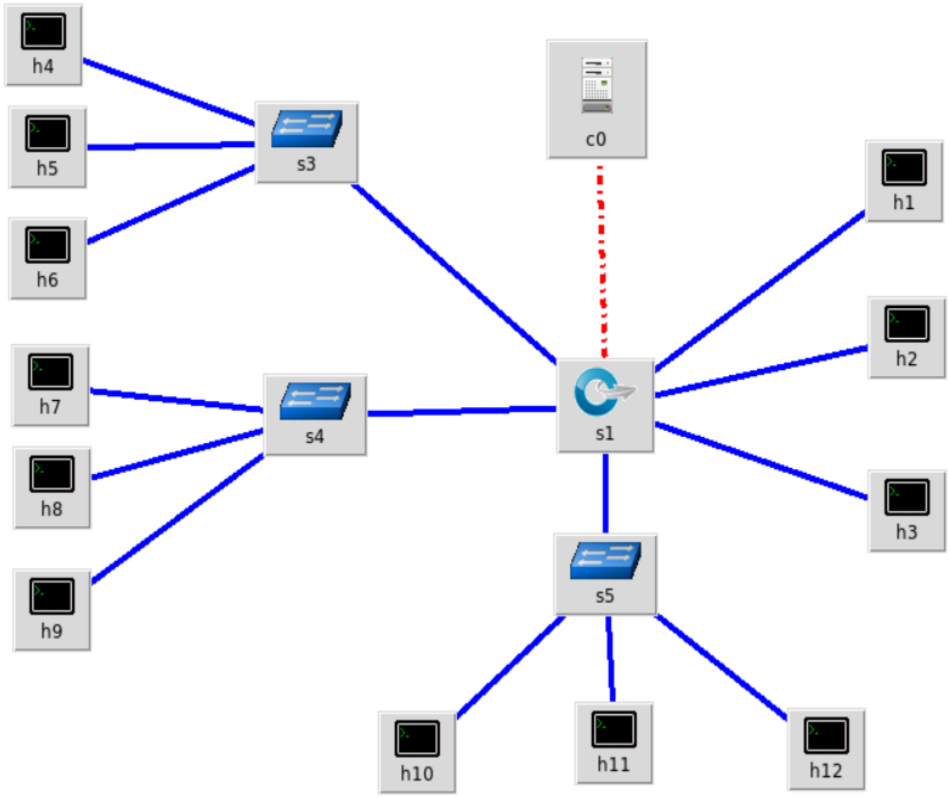

Figure 6 :Topology using MiniEdit

Using Mininet, the configuration which has been pushed on the virtual switch by the controller became simple and independent of the vendor and operating system.

After collecting the data, the software-based load balancer has been implemented using POX as follows. The virtual network topology was implemented using the CLI version of Mininet. The test topology is configured to have a total of six hosts connected to the Open vSwitch. The hosts are made to start with a dynamically assigned but easily readable IP and MAC addresses by using –-mac flag while creating the topology. The Open vSwitch has been configured to connect to the remote POX controller on TCP port 6633 and communicate using the OpenFlow protocol. When the POX controller is started at the command line, it invokes the load balancer program. The controller thereby implements the load balancer functionality in the SDN. The controller installs flow table entries in the Open vSwitch according to the criteria defined in the load balancer algorithm.

Second, the three hosts are configured as HTTP Web Servers and port 80 was opened to listen to HTTP GET Requests. While the remaining nine hosts are populated with the static ARP entry of the virtual IP address of the load balancer. This virtual IP address is be used by the clients to request web pages from the server. For every subsequent request made by the client to the virtual IP address, a different server would reply for the request.

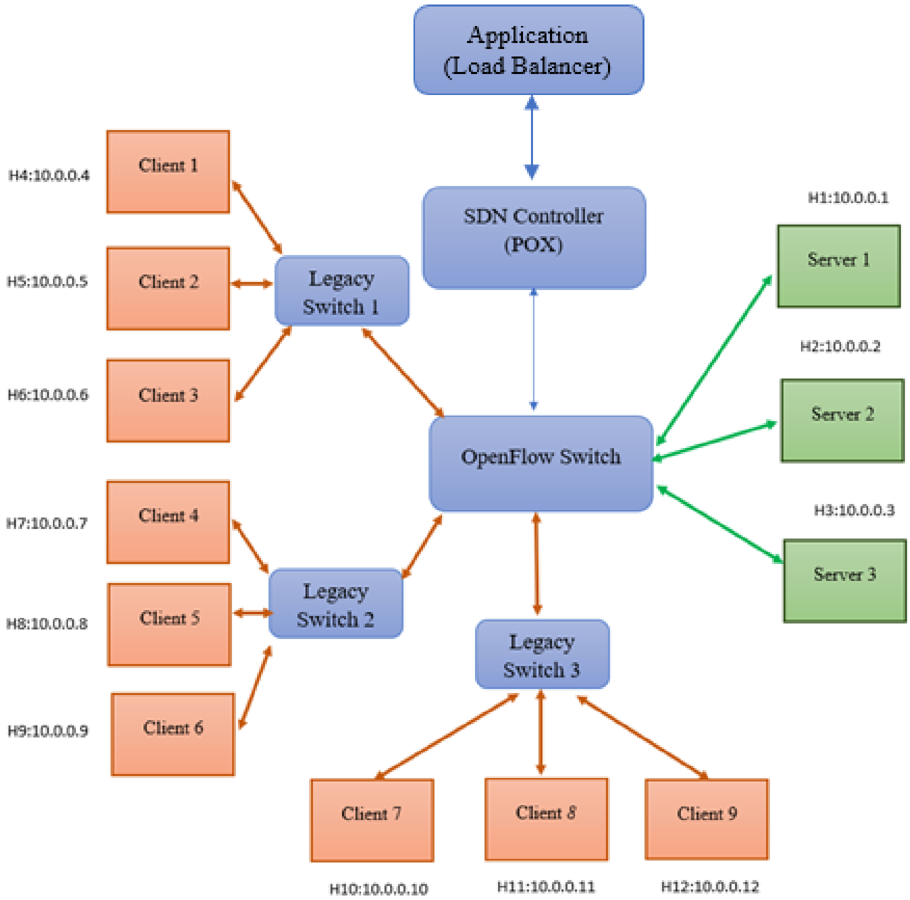

Figure 7 : Implemented network topology

For this Thesis, three different Algorithm has been experimented using Software defined Networking; Round Robin based load balancer, Weighted Round Robin based load balancer and our target load balancer which is Round Robin based load balancer. The controller code has been prepared for round robin, weighted round robin and IP hash-based load balancer initialized separately and the topology with nine host initialized. These controllers execute the load balancing function declared in the load balancing module when a new flow arrives at the switch. This function implements load balancing policy to spread the traffic across the available servers. i.e. for each flow, the controller determines which server is the next server in line to serve the client request based on the algorithm.

Figure 7 illustrates the flow diagram for packets arriving at the switch. Every switch maintains its own flow table. Each entry in the flow table contains the information about packet header and the action to perform. When a packet from a client arrives at the switch, the header information of the packet is compared with the switch's flow table entries. If there is a match, the actions associated with the flow entry are performed on the packet. If there is no match, the packet is forwarded to the controller.

For all packets sent from clients to the controller, the destination IP address of the packet is rewritten to the IP address of the selected server by the load balancer module. After modifying the packet's header, the packet is forwarded to the output port of the selected server. The load balancer module keeps track of this mapping information. When the servers send back a packet to the client, the load balancer module modifies the source IP address of the packet with the IP address of the load balancer.

An SDN controller which acts as a load balancer is configured with an IP address 10.0.1.1 (Virtual IP) and a set of hosts among which it distribute requests were configured with IP addresses, e.g., 10.0.0.1, 10.0.0.2, 10.0.0.3... Clients wanting to access the servers connecting on those hosts are provided with the IP address of the load balancer (SDN Controller), not the IP address of specific Servers. This Load balancer address is term as VIP (virtual IP) for representation. This VIP address is virtual and should be changed to Server address in SDN infrastructure transparently.

A simple HTTP server has been executed on the three host H1, H2 and H3 to act as HTTP server. The SDN controller maintains a list of servers currently connected to the OpenFlow switch and switch port statistics information. Remaining 9 hosts invoke the file request to HTTP server and the response time has been observed over the output file as well as Wireshark invoked inside the mininet environment and same type of traffic scenario with different variation of traffic size and iteration were observed and the result were used to measure performance.

3.2 Algorithm and Flowchart

IP hash-based Load Balancer:

When a Load balancer is configured to use the hash method, it computes a hash value then sends the request to the server. Hash load balancing is similar to persistence-based load balancing, ensuring that connections within existing user sessions are consistently routed to the same back-end servers even when the list of available servers is modified during the user's session.

For Hash, concept from CRC-32 hashing algorithm can be used.[15]. The basic idea behind CRC is simply to use the remainder generated by dividing one polynomial by another. For example, Ethernet uses a well-known divisor polynomial D(x) of degree 32 to allow a sender to view their data as a polynomial P(x) that is transmitted over a link alone with a 32-bit remainder that comes from dividing P(x) by D(x). So, once a sender and receiver agree on a divisor polynomial, the CRC algorithm can be implemented in either software or hardware. With reference to this we used conversion of source IP and Destination IP and XOR them and MOD with the number of live servers.

The load balancer computes the hash values using:

- The back-end server IP Address (X).
- One of the incoming Source IP(Y) & no of available server(N).
- The load balancer computes a new hash value (Z) based on (X) ,(Y) and (N).
- The hash value (Z) is stored in cache.
- The load balancer forwards the request to the server with designated hash value, by using the value (Z) from the computed hash values. Subsequent requests with the same hash value (cached) are sent to the same server.

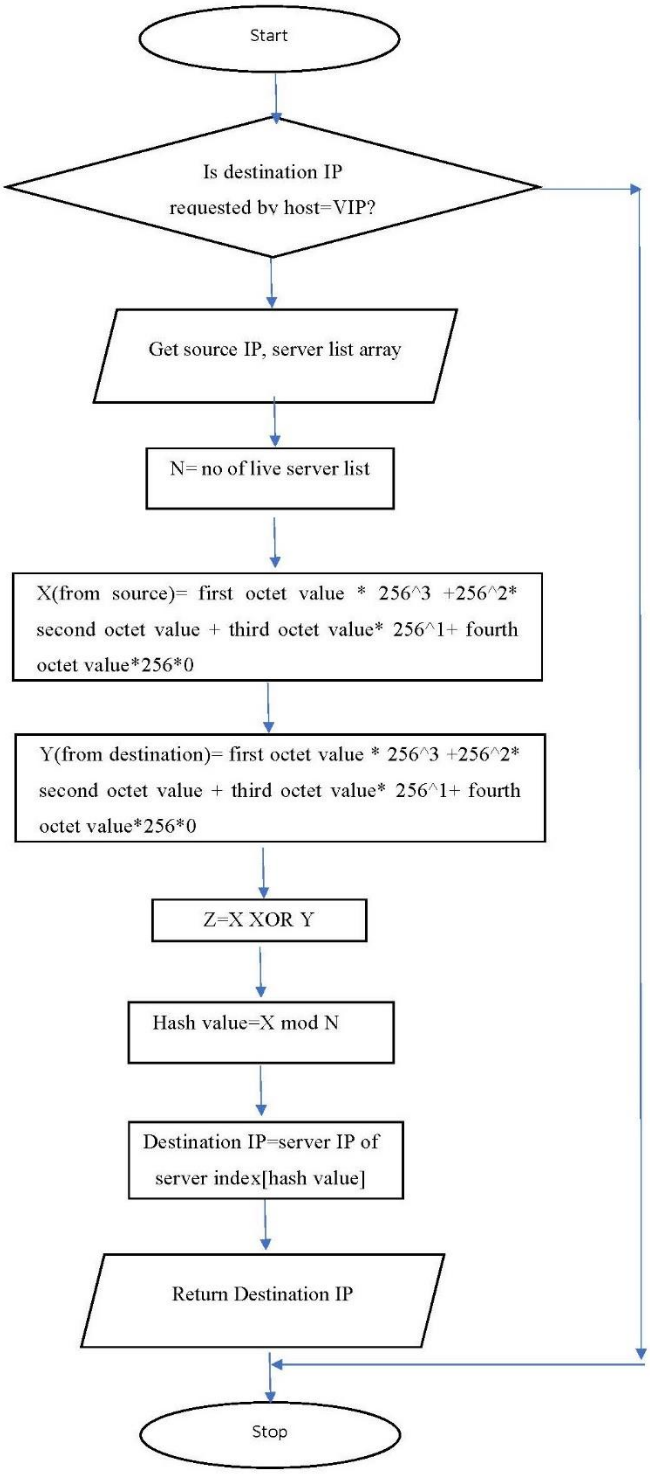

Figure 8 : Flowchart of Load Balancer using IP Hashing

3.3 Calculation

The selection of server for the incoming request is done based on the Hash value calculated using the combination of source IP and Destination IP. The source IP and Destination IP are fetched by the controller module during the network operation. The calculation of this hash value is carrier in the test topology using below approach.

Each octet of fetched IP address is separated and calculated as:

1. Decimal equivalent of Source IP octet 1= Value of first octet *256^3
2. Decimal equivalent of Source IP octet 2= Value of second octet* 256^2
3. Decimal equivalent of Source IP octet 2= Value of third octet* 256^1
4. Decimal equivalent of Source IP octet 2= value of fourth octet *256^0
5. Total Decimal equivalent Source = Decimal equivalent of IP octet 1+ Decimal equivalent of IP octet 2+ Decimal equivalent of IP octet 3+ Decimal equivalent of IP octet 4
6. Decimal equivalent of Destination IP octet 1 = Value of first octet *256^3
7. Decimal equivalent of Destination IP octet 2= Value of second octet* 256^2
8. Decimal equivalent of Destination IP octet 2= Value of third octet* 256^1
9. Decimal equivalent of Destination IP octet 2= value of fourth octet *256^0
10. Total Decimal equivalent Destination = Decimal equivalent of IP octet 1+ Decimal equivalent of IP octet 2+ Decimal equivalent of IP octet 3+ Decimal equivalent of IP octet 4
11. Total Decimal Equivalent= Total Decimal equivalent Source **XOR** Total Decimal equivalent Destination
12. Hash value=Total Decimal equivalent **MOD** no of uplink

Selection of server is based on the index number defined corresponding to the generated Hash Value.

3.4 Tools

Mininet Emulator:

Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM, cloud or native). We can easily interact with our 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.[16] The setup of this research experiment is done on mininet deployed on ubuntu 14.04.4 LTS virtualized over VMware Workstation 14.

POX as a SDN Controller:

SDN Controllers play a major role in the SDN environment. It is an application than handles all the flow between the switches and application modules through Northbound API. In brief, SDN controller is the one that handles all the controlling activities of the entire network. The default protocols used for communication are OpenFlow and Open Virtual Switch Database(OVSDB). Later, several Open Source controllers came into existence. This includes NOX, POX, Ryu, Floodlight, OpenDaylight, ONOS, Pyretic etc. For our research POX has been selected as the Open Flow Controller for control layer to infrastructure layer communication. NOC controller is the first opensource Open Flow controller developed which was created using C++ language. POX is a python version of NOX controller.[17]

Xming:

Xming is an X11 display server for Microsoft Windows operating systems. Xming provides the X Window System display server, a set of traditional sample X applications and tools, as well as a set of fonts. It features support of several languages and has Mesa 3D, OpenGL, and GLX 3D graphics extensions capabilities.

Putty and SSH Secur*e* Shell Client:

PuTTY is an SSH and telnet client, developed originally by Simon Tatham for the Windows platform. PuTTY is open source software that is available with source code and is developed and supported by a group of volunteers.

Similarly, SSH secure shell client is another SSH and telnet client which is a good option for secure system administration and is easy option for file transfers. This program is very simple to use and it wields a lot of power inside.

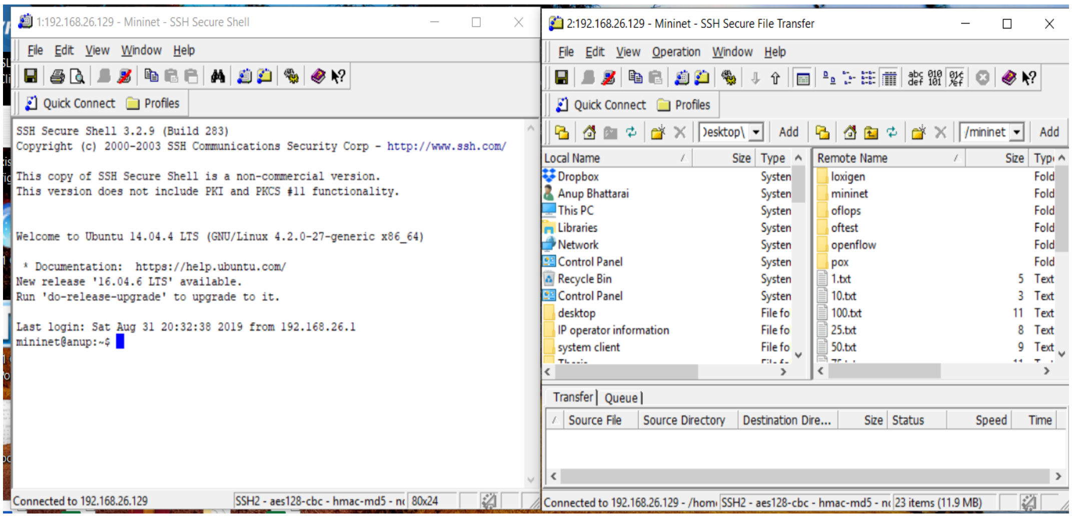

Figure 9 : Interface of SSH client for accessing Mininet running on VMWARE

VMware Workstation as Virtualization Software:

VMware Workstation Pro is the industry standard for running multiple operating systems as virtual machines (VMs) on a single Linux or Windows PC It is a cross-platform virtualization application which 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 VMware creates for guest operating system while it is running. During this research, Ubuntu image with mininet addon is virtualized using VMware Workstation 14.

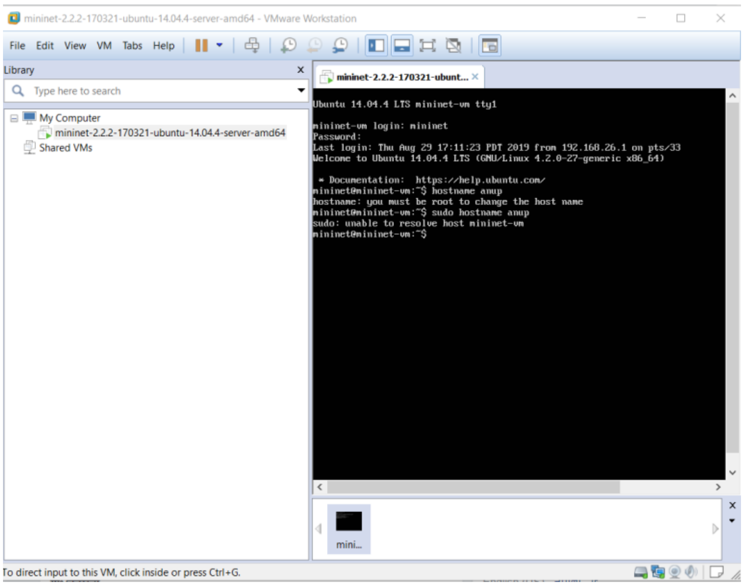

Figure 10: VMWARE workstation Interface

Wireshark

Wireshark is the world's foremost and widely-used free and open source network protocol analyzer.[18] 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

Miniedit

Miniedit is a simple network editor for Mininet. It is a GUI mininet network tree creation tool. It requires X windows system to be installed. During Virtualization for the research xming was deployed to run the miniet GUI interface.

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.[19]

3.5 Data Collection and Evaluation

Data required for the test has been generated using simulation and using IPERF, an open source network performance measuring tool, the network throughput test was calculated for analysis. Also, a simple HTTP server was loaded in server and HTTP GET request was used from hosts using python script to fetched test data files from server and the response time was noted for analysis. For the test file, different size file ranging from 1KB, 100KB, 1MB, 10MB, 100MB was used. Further some sample files from Standard dataset source like NIMS , DARPA was also used to test for our analysis.

CHAPTER FOUR: RESULT AND DISCUSSION

5.1 Round Robin Based Load Balancer load distribution:

During the experiment on traffic through Round robin load balancer using traffic of different sizes with 100 iteration from 9 different host simultaneously. The load distribution was observed to be uniformly distributed in a round robin manner with 33.33% load distributed in each server for the request generated from 9 different hosts as mentioned in Table 1 and Figure 11.

Table 1 : Traffic Distribution based on RR based Load Balancer

S.N	**Request redirected to Server**	**Count**	**Percent**
1	10.0.0.3	300	33.33%
2	10.0.0.2	300	33.33%
3	10.0.0.1	300	33.33%

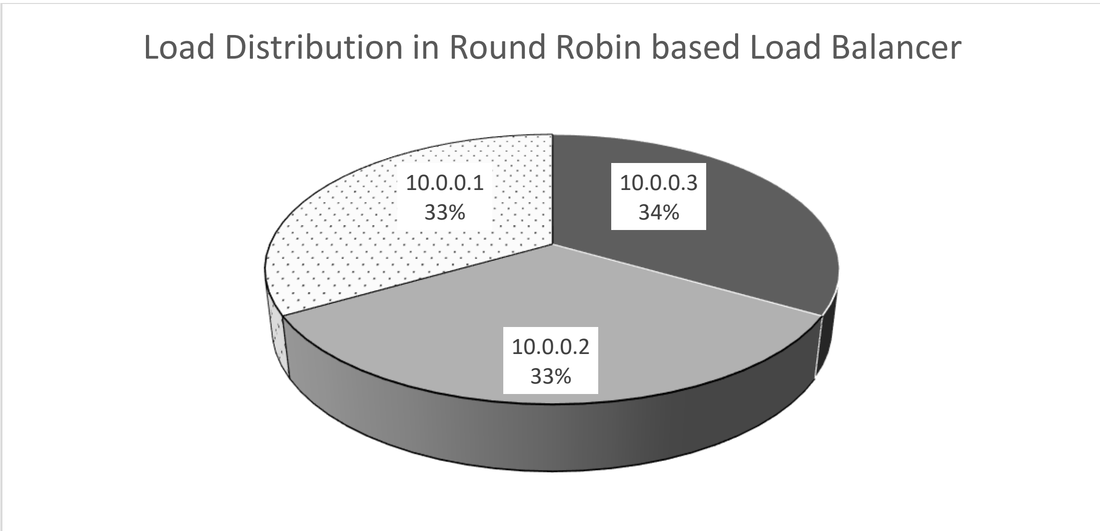

Figure 11: Traffic Distribution chart based on RR based Load Balancer

5.2 Weighted Round Robin Based Load Balancer load distribution:

During the experiment on traffic through Weighted Round robin load balancer using traffic of different sizes with 100 iteration from 9 different host simultaneously. The load distribution was observed to be distributed based on weight and selected on a round robin manner. The setup was done with first server to carry with weight 2 and weight 1 on remaining two servers. As per the observation, 50% load was distributed to first server and 25% each was distributed to remaining servers for the request generated from 9 different hosts as mentioned in Table 2 and Figure 12.

Table 2 : Traffic Distribution based on Weighted RR based Load Balancer

S. N	**Request redirected to Server**	**Count**	**Percent**
1	10.0.0.3	225	12.50%
2	10.0.0.2	225	12.50%
3	10.0.0.1	450	25.00%

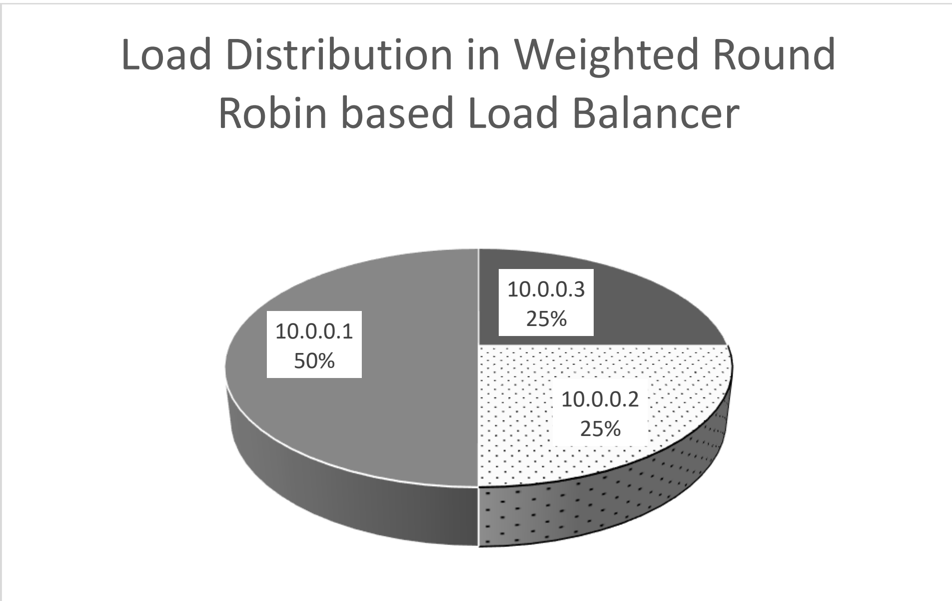

Figure 12: Traffic Distribution chart based on Weighted RR based Load Balancer

5.3 IP hash Based Load Balancer load distribution:

During the experiment on traffic through IP hash-based load balancer using traffic of different sizes with 100 iteration from 9 different host simultaneously. The load distribution was observed to be based on hash value. The setup was done such that each source IP was converted to decimal value and hashed with no of live server which is 3 in our case generating a unique value for that IP. Each of 9 host seems to be generated equally 3 set of unique value uniformly. Since the 9 host's hash value seems to match uniformly among the server selection criteria. As per the observation, each server seems to cater 33.33 percent of total traffic request generated by different host and each host was served by same server each time as mentioned in Table 3 and Figure 13.

Table 3 : Traffic Distribution based on IP hash-based Load Balancer

S.N	**Request redirected to Server**	**Count**	**Percent**
1	10.0.0.3	300	33.33%
2	10.0.0.2	300	33.33%
3	10.0.0.1	300	33.33%

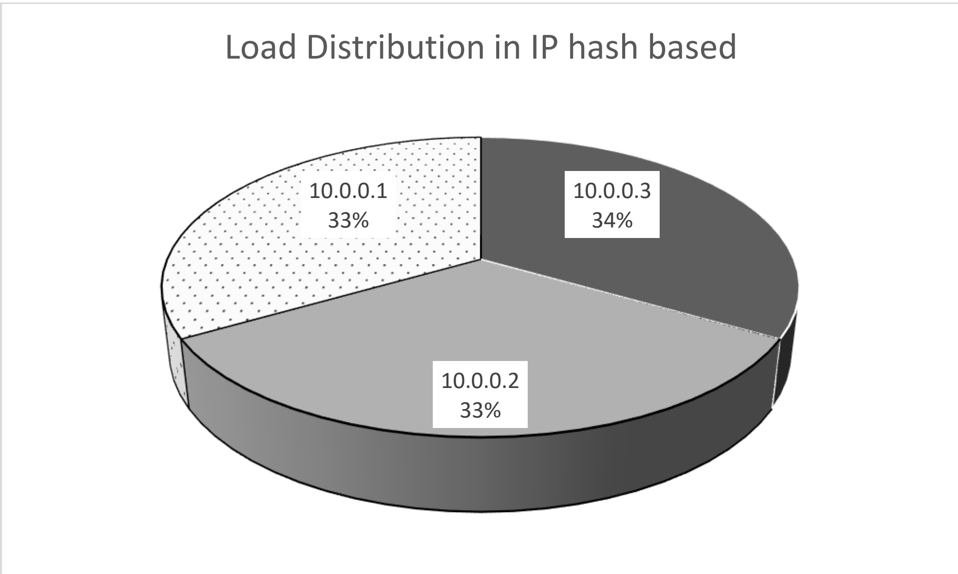

Figure 13: Traffic Distribution chart based on IP Hash based Load Balancer

5.4 Average Response time with different simultaneous load

Table 4 : Average Response time with 100*9 iteration on different simultaneous load

File Size	RR Based (sec)	Weighted RR Based(sec)	IP Hash Based(sec)
1 KB	0.904	1.165	0.834
100KB	1.023	1.149	1.043
1MB	0.956	1.164	0.893
10MB	1.179	1.382	1.146
100MB	2.804	3.546	2.347

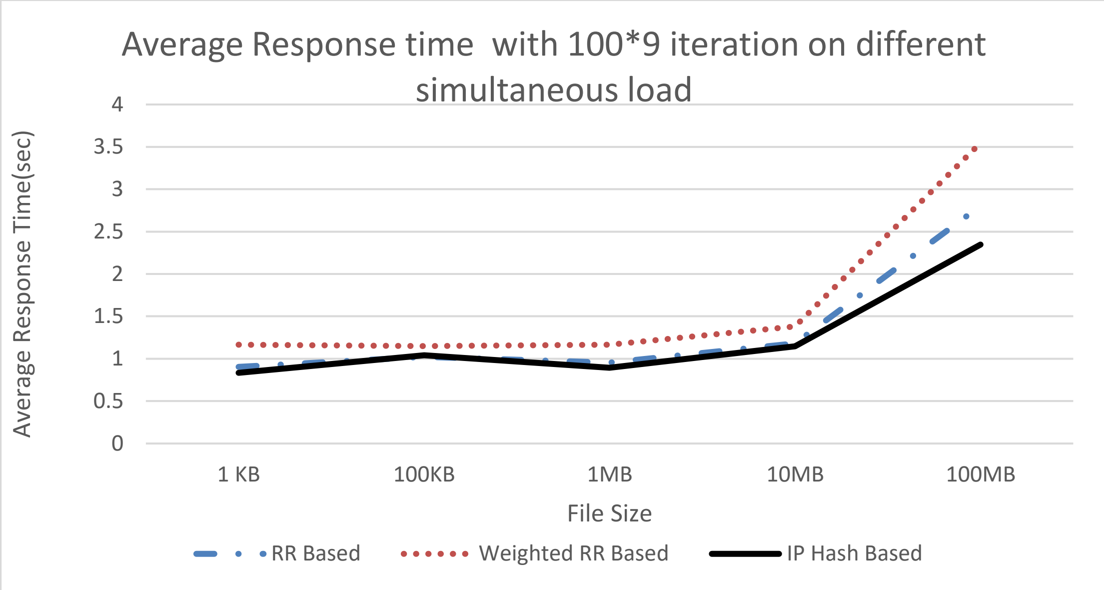

Figure 14: Average Response time with 100*9 iteration on different simultaneous load

Further during the observation of Average Response time with 100*9 iteration on different simultaneous load , the comparison table 4 and figure 14 , shows that IP hash algorithm has lower response time at lower size load and its performance is similar to round robin response time but at higher size file, the response time for IP hash algorithm seems to be better.

5.5 Average Response Time with same load at a time

Table 5 : Average Response Time with same load at a time

File Size	RR Based(sec)	Weighted RR Based(sec)	IP Hash Based(sec)
1 KB	0.010	0.0167	0.010
100KB	0.017	0.0267	0.010
1MB	0.070	0.120	0.043
10MB	0.200	0.340	0.213
100MB	2.306	1.603	1.556

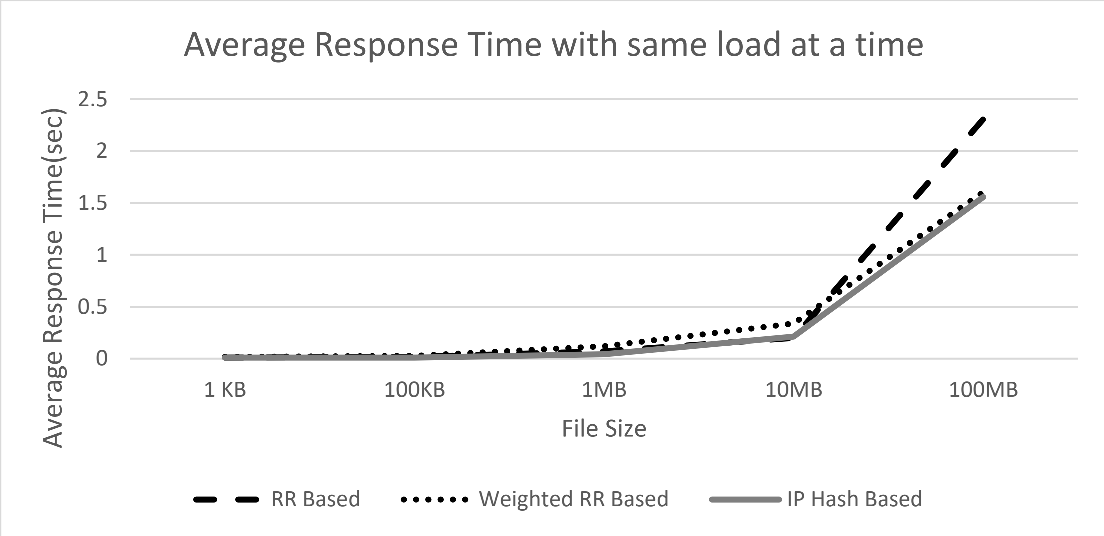

Figure 15: Average Response Time with same load at a time

During the repeated load of same size distributed at different instance, all the algorithm has similar average response time at low size but as the file size is increased that IP Hash Load balancer seems to have better result in terms of response time as mentioned in Table 5 and Figure 15.

5.6 Average Response Time with different File sample available from Standard Dataset source

Table 6 : Average Response time for file sample from standard Dataset source

File Name	File size	RR Based (sec)	Weighted RR Based (sec)	IP Hash Based (sec)
onlineBanking.csv	2KB	0.007	0.013	0.013
Primus.csv	120KB	0.013	0.023	0.010
NIMS.arff	5MB	0.137	0.150	0.147
DARPA99Week1	30MB	0.547	0.540	0.530
DARPA99Week2	135MB	2.250	2.150	2.137

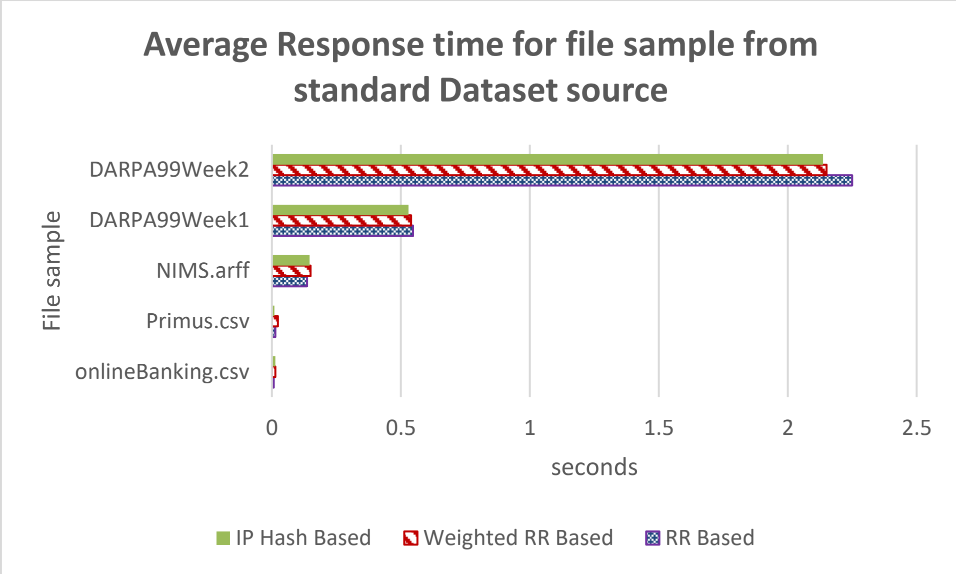

Figure 16: Average Response time for file sample from standard Dataset source

Similarly using a sample files from standard network data sources, we observed the average response time were almost equal for four dataset and for 1 dataset RR showed slightly greater response time. Among all these datasets the performance of IP hash based algorithm is observed to be slightly better as mentioned in Table 6 and Figure 16.

5.7 Turn Around Time with same size packet:

Table 7 : Turn Around Time with same size packet

Iteration	RR based(ms)	Weighted RR based (ms)	IP Hash based (ms)
100	17798.479	18618.557	18112.480
200	36933.612	36315.772	36484.797
500	90495.971	91688.535	90549.164
1000	180127.233	182532.450	182026.505

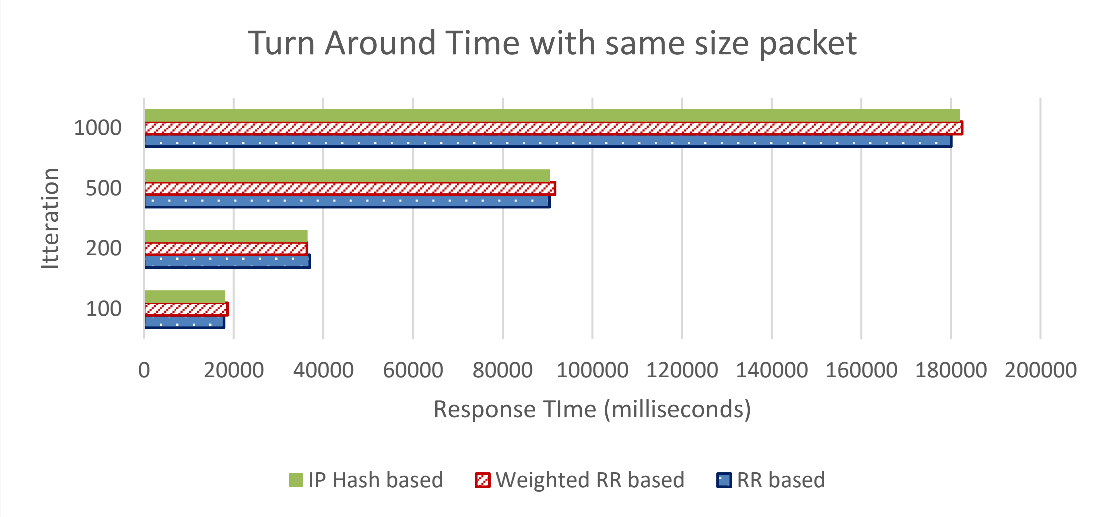

Figure 17: Turn Around Time with same size packet

For next phase using different iteration with value as 100, 200 ,500 and 1000 was used to check the Turnaround time. The observation shows that there is only small variance in turnaround time observed at different iterations as mentioned in Table 7 and Figure 17.

5.8 Analysis using IPERF tool:

Table 8 :Total transfer and Average speed from IPERF tool test

	total transfer (MB)	**average speed (Mbps)**
Round Robin	2067	566
Weighted RR	1996	550
IP Hash	2035	568

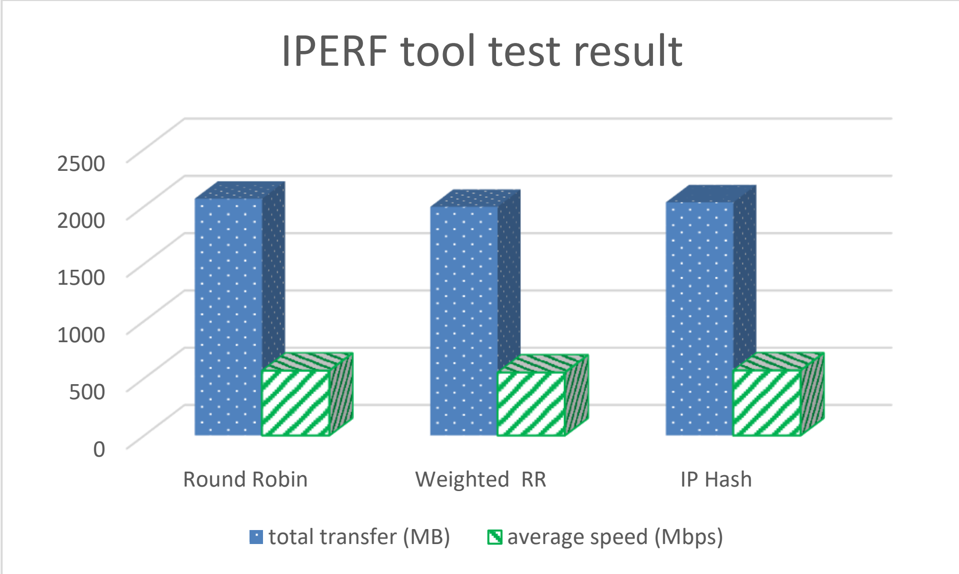

Figure 18: Total transfer and Average speed from IPERF tool test

Similarly, for the test using IPERF tool, the iperf was executed with server mode (iperf -s) on h1 h2 and h3 and as client (iperf -c 10.0.1.1) on host h4,h5…..h12. During IPERF test, the total volume usage and average speed of all three algorithm was near to each other and among them IP hash algorithm was found to be cater slightly higher bandwidth as shown in table 8 and Figure 18.

5.9 Latency Test:

Table 9 : Latency test

	Ping Reply (ms)
Round Robin	34.092
Weighted RR	34.930
IP Hash	33.001

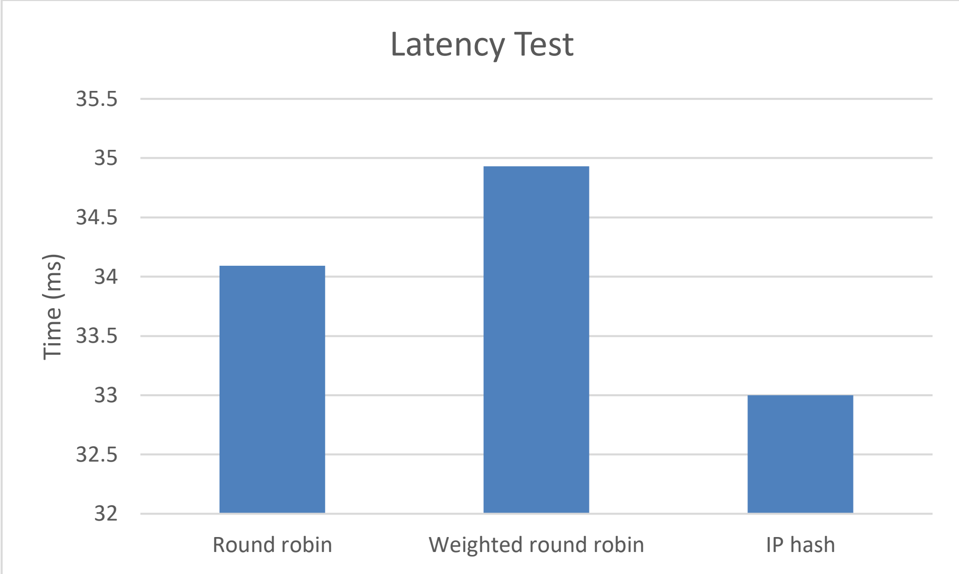

Figure 19: Latency Test

Similarly, during the latency test the ping response for the difference load balancing algorithm in SDN was observed to vary from 33.001 ms to 34.092 ms. The lowest latency was observed in IP hash algorithm among the three-load balancing algorithm. During these overall observations, there was some slight variation in difference performance parameters among three different load balancers tested using OpenFlow switch in software defined networking tested under different load scenarios. However, each of them has its own significance. Round Robin is the best approach on ideal condition and the simplest of the three methods. The weighted Load based algorithm is best when the server capacity or link capacity among the available server are not uniform. In such case, the weight can be applied on the traffic distribution to achieve the distribution as per the resource. And lastly, the significance of IP Hash Based load balancer is that this algorithm distributed the traffic among the server but does it with a logic using hash such that each client shall always be served by same server until the live server count is same. This reduces the requirement in servers where client authentication is required. Further any reverification of client every time while change of server during request also reduces the delay in this process while using IP hash-based load balancer in Software Defined Networking. Since the server selection was unique for each client, this can be considered as the reason for slightly better performance of IP hash algorithm among the other compared load balancer in SDN environment.

CHAPTER FIVE: LIMITATIONS

Performance analysis during implementation of IP hash-based load balancer is carried out and compared with other load balancer considering parameters like response time of server using simultaneous load, individual load response time and total turnaround time. Further to avoid overload, the variation of load balancer was based on CPU utilization of server. This thesis does not consider link bandwidths since optimization of routes incoming to the load balancer are regarded to be role of routers and the protocol used in them. In this research ,all the link from open flow switch used by load balancer to server are considered to be enough capacity for IP hash based strategy implementation as load balancer and servers are in practice generally placed and maintained inside same organization and opensource tool like catty are available for monitoring of individual links.

CHAPTER SIX: CONCLUSIONS AND FUTURE RECCOMENDATIONS

SDN is a new approach to facilitate network management that has been gaining widespread attention and load balancer is an important aspect for proper distribution of load across different application/network servers. A load balancer based on IP-HASH algorithm has been successfully implemented and its performance was evaluated and compared with two other load balancing strategy; round robin and weighted round robin strategy. In most of the tested samples the performance of all compared strategies; round robin, weighted round robin and IP hash based were similar while in higher file size samples, the response time of IP-hash algorithm was slightly better compared to two other algorithms. The key significance of IP-hash algorithm compared to other two strategies is that IP-hash algorithm attempts to select the same server for same client based on the hash value. With the compared evaluation of three strategies, IP-hash based algorithm can be a better choice to implement in SDN environment.

Further the SDN environment in mininet was also updated with dual stack model using IPV6 addressing along with IPV4 IP addresses and server were assigned dual stack IPv4 and IPV6 addresses. However, the hash value calculation was still implemented based on ipv4 addressing mode. Future works need to be done with more focus on IP hash-based load balancing in SDN environment in IPV6 only model.

REFERENCES

1. Opennetworking.org, *SDN Architecture Overview*. 2014. 1.1.
2. Deep Gajjar, H.K., Hardik Joshi, Dharmesh Mithbavkar,Dr. Levi Perigo, Jason Schnitzer, *Round Robin Load Balancer using Software Defined Networking (SDN)*. 2016.
3. Pradeep Kumar Sharma, D.S.S.T., *Strengthening Network Security: An SDN (Software Defined Networking) Approach.* International Journal of Electrical Electronics & Computer Science Engineering, 2018. ICSCAAIT-2018.
4. Barger, J. *https://www.econfigs.com/ccna-7-7-c-northbound-and-southbound-apis/*. [cited 2019.
5. Russello, J. *Southbound vs. Northbound SDN: What are the differences?* 2016 [cited 2019; Available from: http://blog.webwerks.in/data-centers-blog/southbound-vs-northbound-sdn-what-are-the-differences.
6. Swami, R., M. Dave, and V. Ranga, *Software-defined Networking-based DDoS Defense Mechanisms %J ACM Comput. Surv.* 2019. 52(2): p. 1-36.
7. Lim, S., et al. *A SDN-oriented DDoS blocking scheme for botnet-based attacks*. in *2014 Sixth International Conference on Ubiquitous and Future Networks (ICUFN)*. 2014.
8. *About POX*, *https://noxrepo.github.io/pox-doc/html/*
9. Bryhni, H., E. Klovning, and O. Kure, *A comparison of load balancing techniques for scalable web servers.* IEEE network, 2000. 14(4): p. 58-64.
10. NGINX, *What Is Load Balancing? How Load Balancers Work.* 2015.
11. Kaur, S., et al. *Round-robin based load balancing in Software Defined Networking*. in *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. 2015.
12. Hikichi, K., T. Soumiya, and A. Yamada. *Dynamic application load balancing in distributed SDN controller*. in *2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. 2016.
13. Sabiya, J.S., *Weighted Round-Robin Load Balancing Using Software Defined Networking*. ISSN: 2277 128X, 2016.

14. Ju-Yeon, J. and K. Yoohwan. *Hash-based Internet traffic load balancing*. in *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004.* 2004.
15. Prodanoff, Z.G. and R. King. *CRC32 Based Signature Generation Methods for URL Routing*. in *IEEE SoutheastCon, 2004. Proceedings.* 2004.
16. Team, M., *http://mininet.org/*.
17. Fancy, C. and M. Pushpalatha. *Performance evaluation of SDN controllers POX and floodlight in mininet emulation environment*. in *2017 International Conference on Intelligent Sustainable Systems (ICISS)*. 2017.
18. Combs, G., *https://www.wireshark.org/*.
19. iperf.fr, *https://iperf.fr/*.

APPENDIX A: EXPEREMENT OUTPUT

Round Robin based Load Balancer:

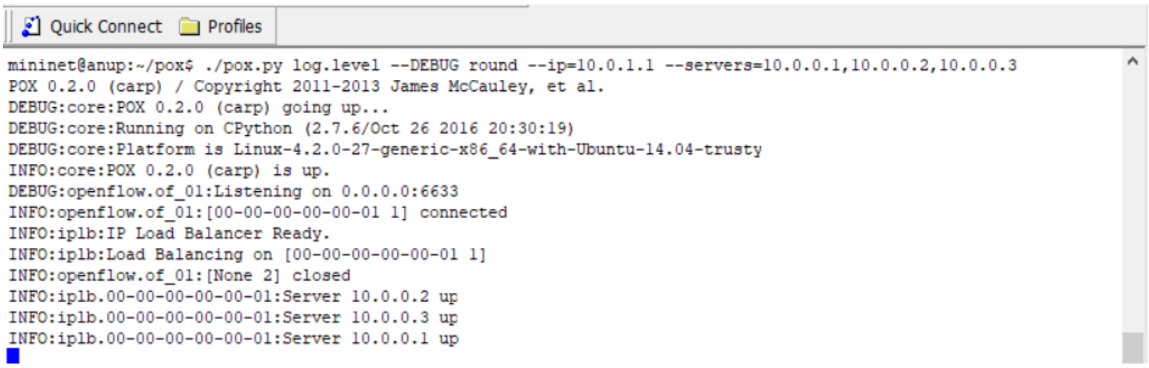

Figure 20: Initialization of Round robin-based controller

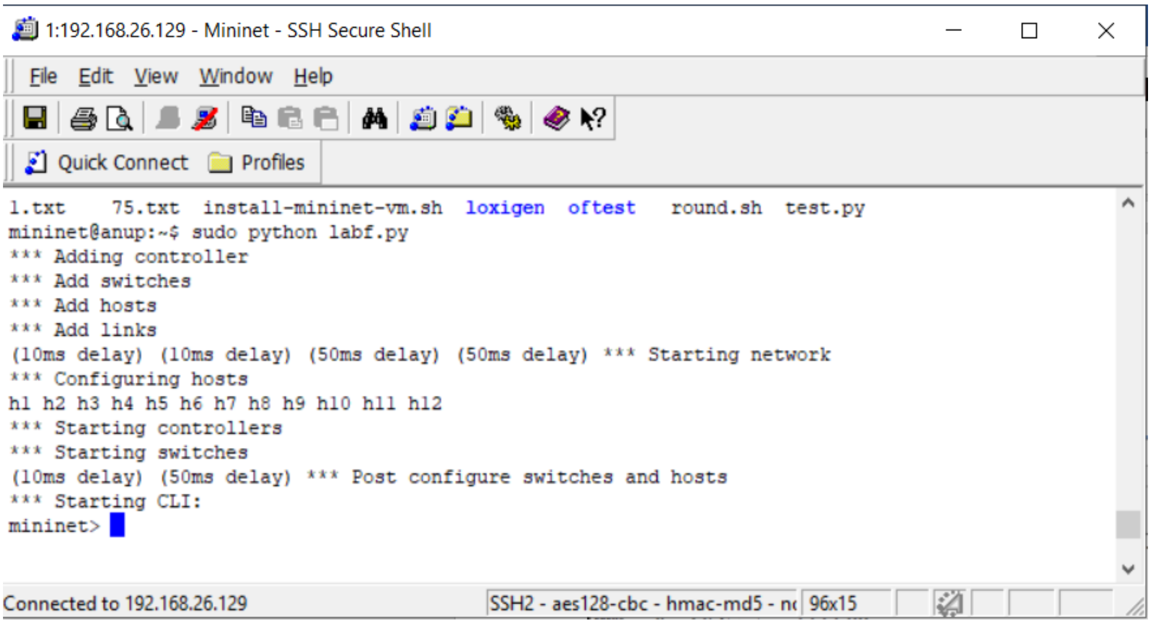

Figure 21: Initialization of Topology of Experiment network

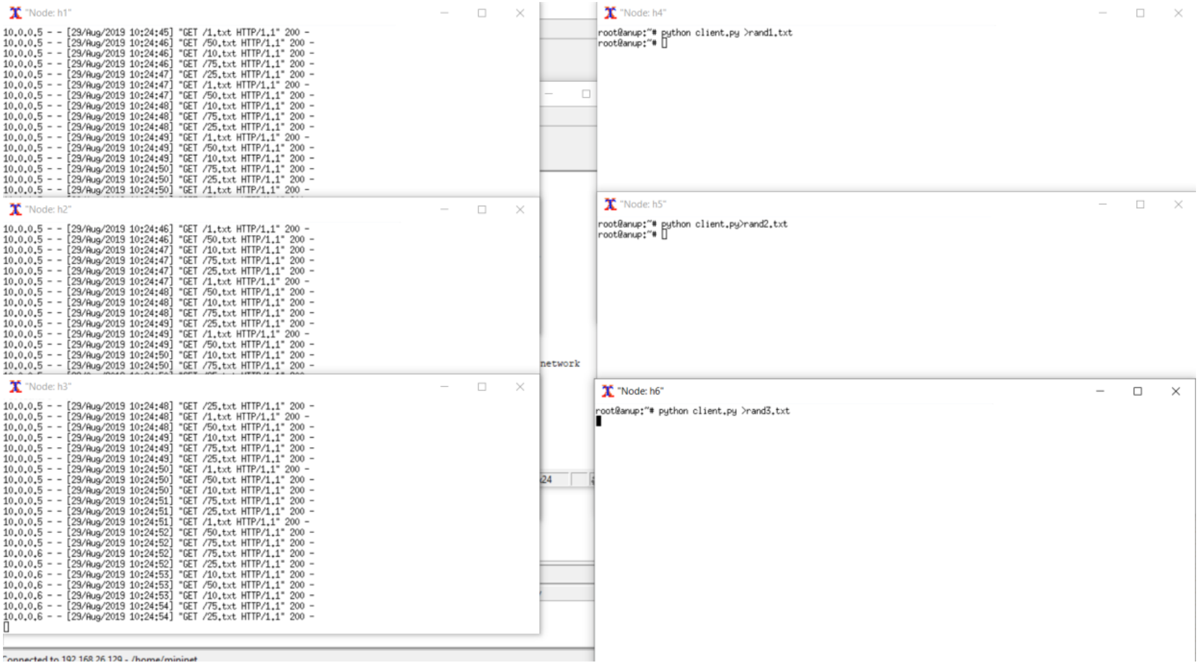

Figure 22: Generation of HTTP traffic towards Switch

```
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
```

Figure 23: Open flow load balancer traffic distribution based on Round Robin

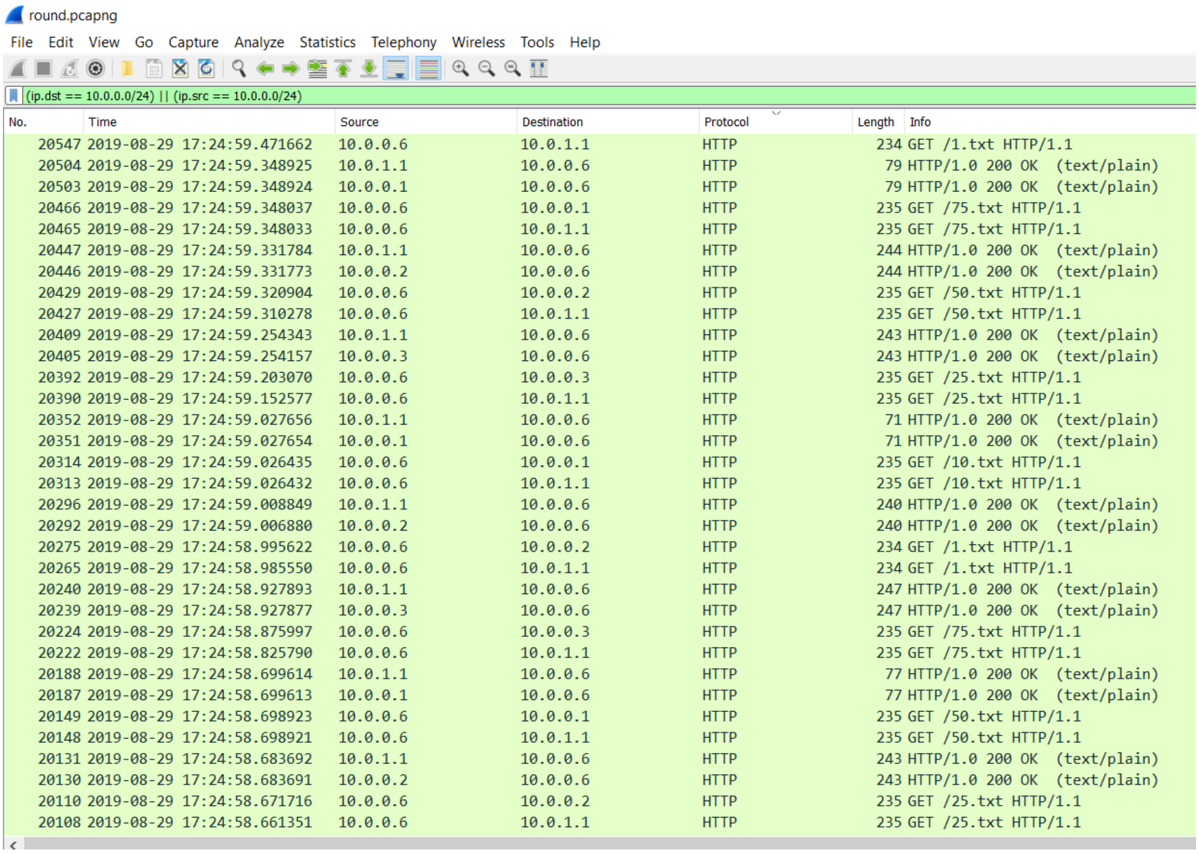

Figure 24: Capture of HTTP traffic exchange from Wireshark

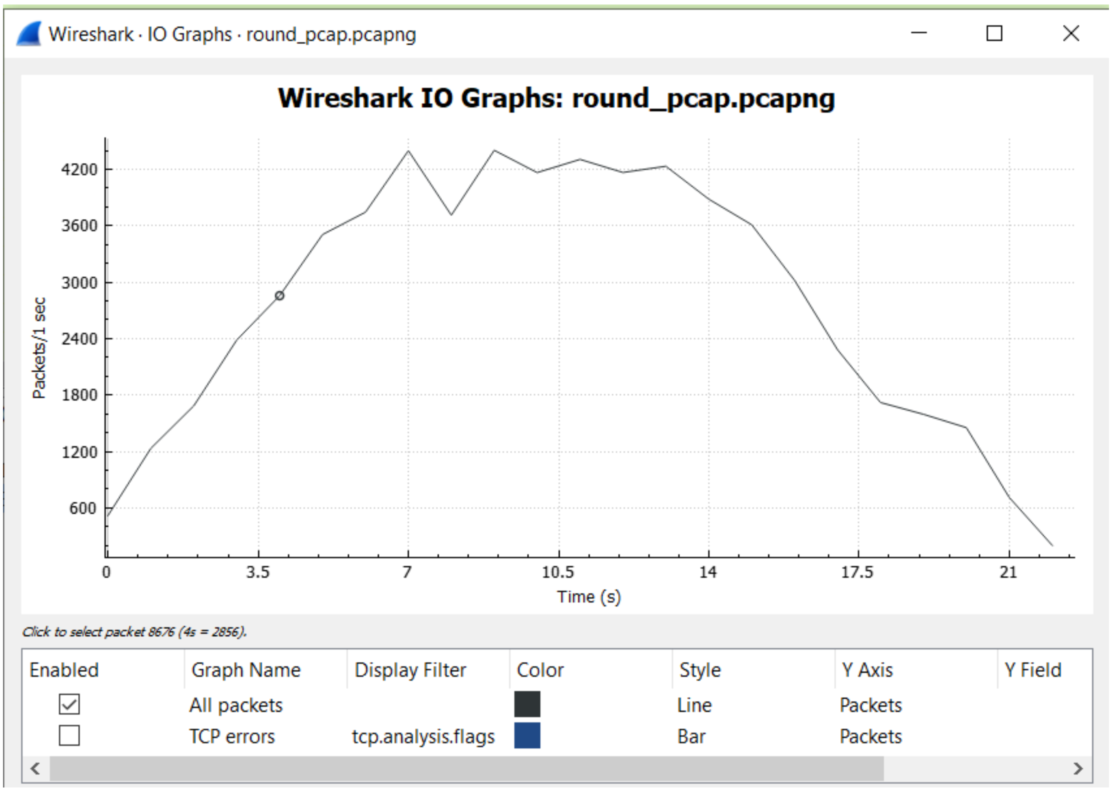

Figure 25: PPS graph from Wireshark during Round Robin Based experiment

Weighted round robin based Load Balancer:

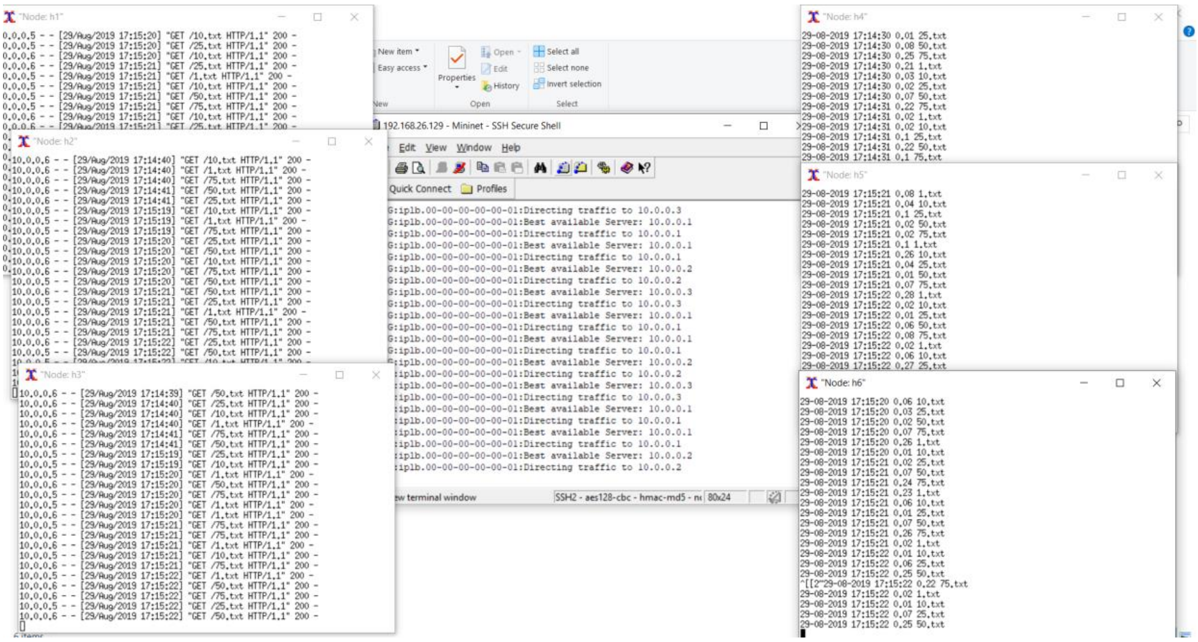

Figure 26: Generation of HTTP traffic towards Switch based on Weighted RR

Quick Connect Profiles

```
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.2
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Best available Server: 10.0.0.1
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.1
```

Figure 27: Open flow load balancer traffic distribution based on Weighted RR

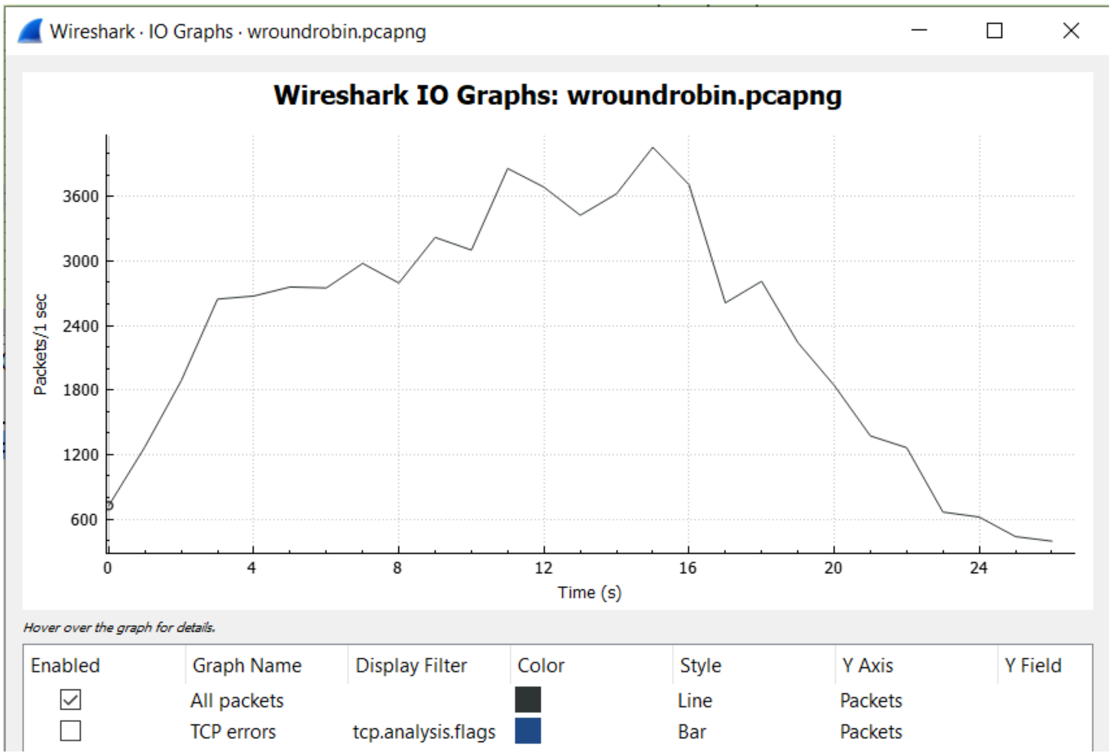

Figure 28: PPS graph from Wireshark during Weighted RR Based experiment

IP hash based Load Balancer:

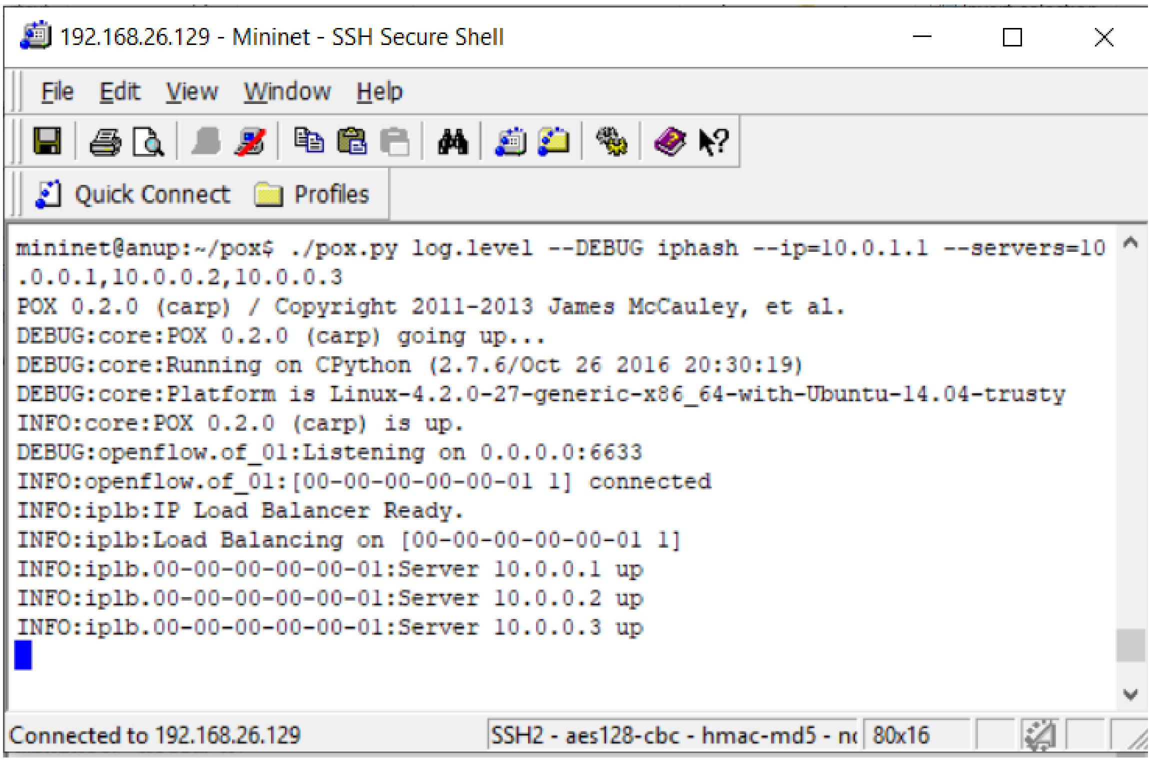

Figure 29: Initialization of IP hash based controller

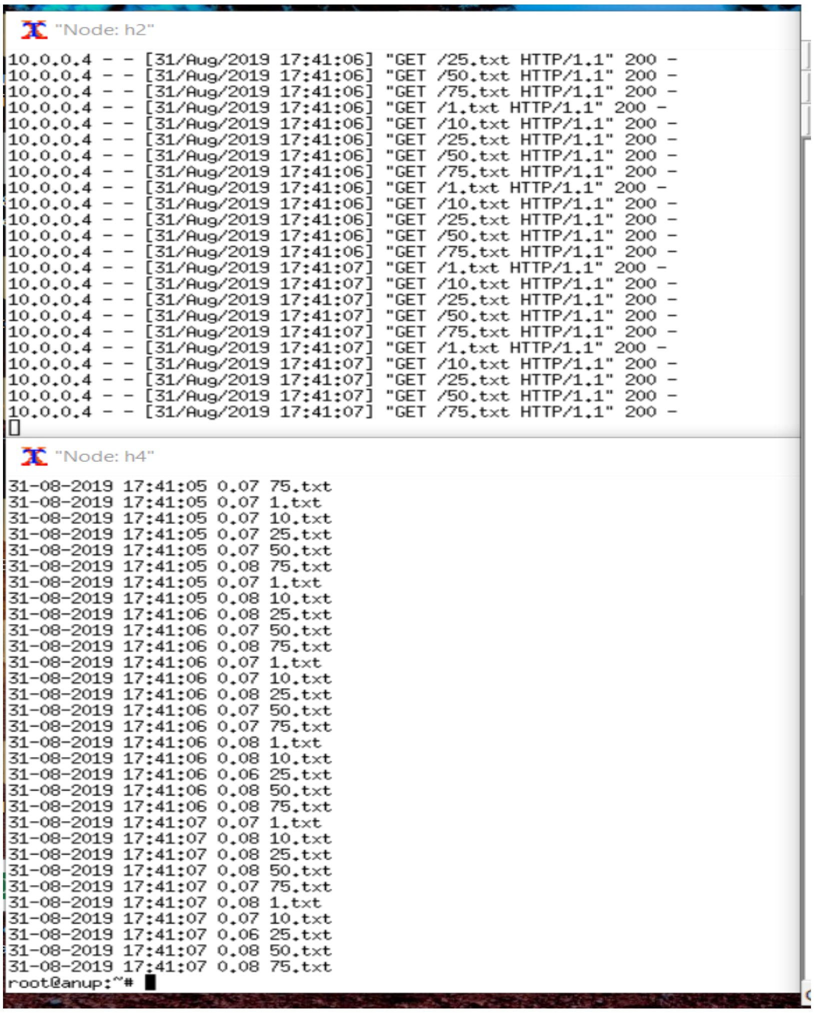

Figure 30: Generation of HTTP traffic towards Switch based on IP hash Based

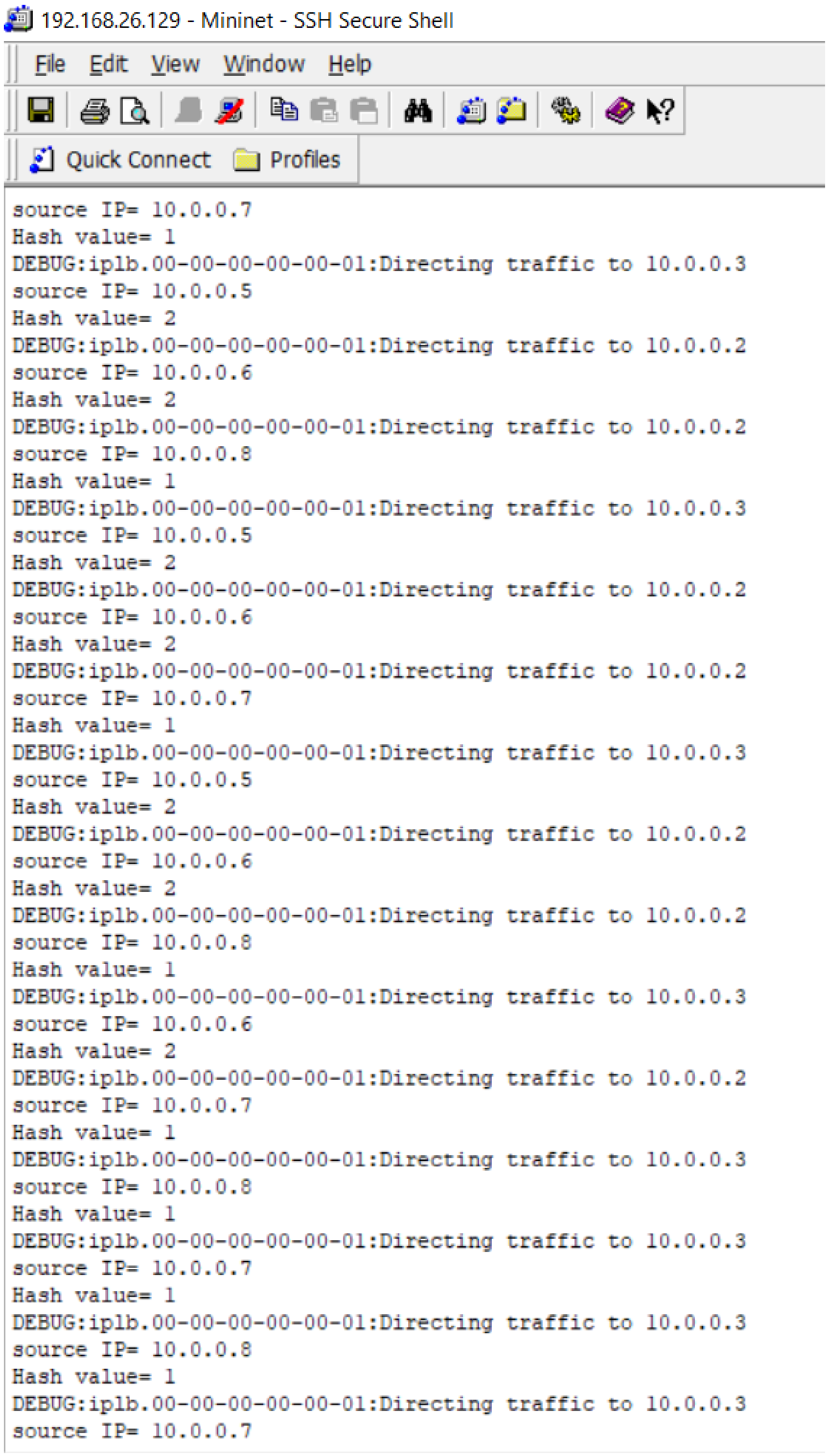

Figure 31: Open flow load balancer traffic distribution based on IP hash based


```
DEBUG:iplb.00-00-00-00-00-01:Destination IP: fc00::100
Hash value= 0
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Source IP: fc00::4
DEBUG:iplb.00-00-00-00-00-01:Destination IP: fc00::100
Hash value= 0
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Source IP: fc00::9
DEBUG:iplb.00-00-00-00-00-01:Destination IP: fc00::100
Hash value= 0
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Source IP: fc00::4
DEBUG:iplb.00-00-00-00-00-01:Destination IP: fc00::100
Hash value= 0
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Source IP: fc00::4
DEBUG:iplb.00-00-00-00-00-01:Destination IP: fc00::100
Hash value= 0
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Source IP: fc00::9
DEBUG:iplb.00-00-00-00-00-01:Destination IP: fc00::100
Hash value= 0
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Source IP: fc00::4
DEBUG:iplb.00-00-00-00-00-01:Destination IP: fc00::100
Hash value= 0
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Source IP: fc00::4
DEBUG:iplb.00-00-00-00-00-01:Destination IP: fc00::100
Hash value= 0
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Source IP: fc00::4
DEBUG:iplb.00-00-00-00-00-01:Destination IP: fc00::100
Hash value= 0
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Source IP: fc00::4
DEBUG:iplb.00-00-00-00-00-01:Destination IP: fc00::100
Hash value= 0
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Source IP: fc00::4
DEBUG:iplb.00-00-00-00-00-01:Destination IP: fc00::100
Hash value= 0
DEBUG:iplb.00-00-00-00-00-01:Directing traffic to 10.0.0.3
DEBUG:iplb.00-00-00-00-00-01:Source IP: fc00::4
DEBUG:iplb.00-00-00-00-00-01:Destination IP: fc00::100
```

Figure 32: load balancer traffic distribution based on IP hash based using dual IP stack

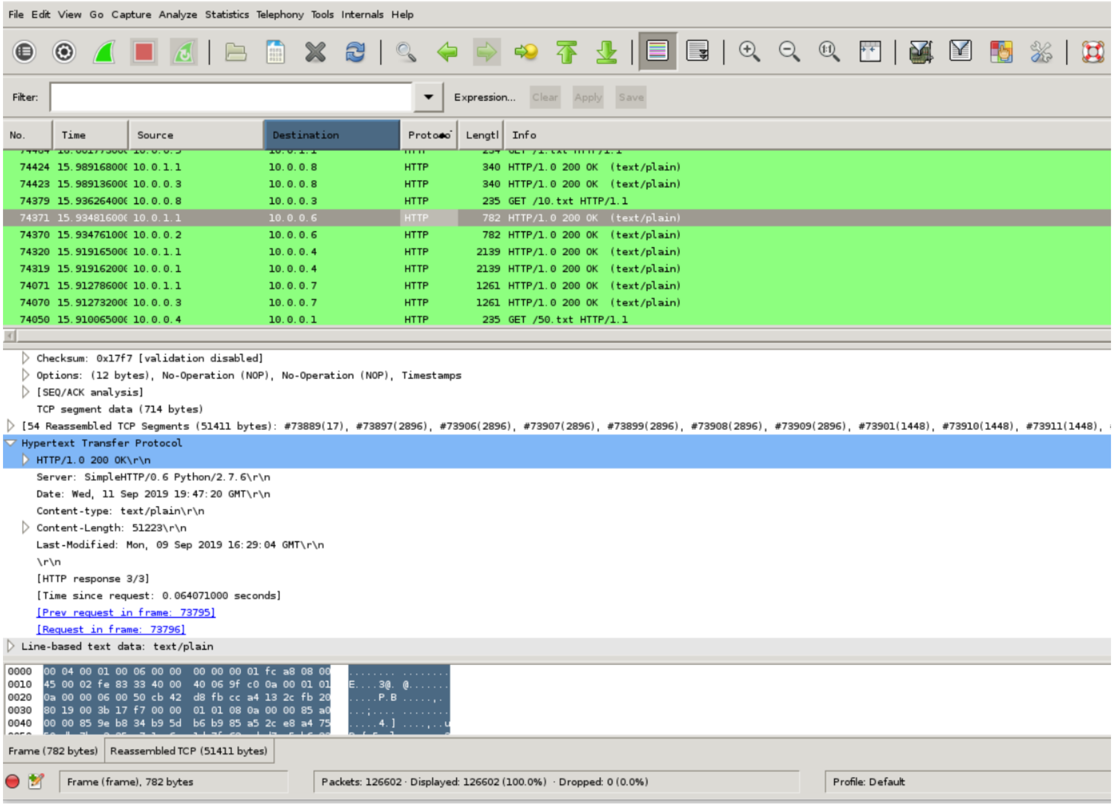

Figure 33: Capture of HTTP traffic exchange from Wireshark during IP hash based

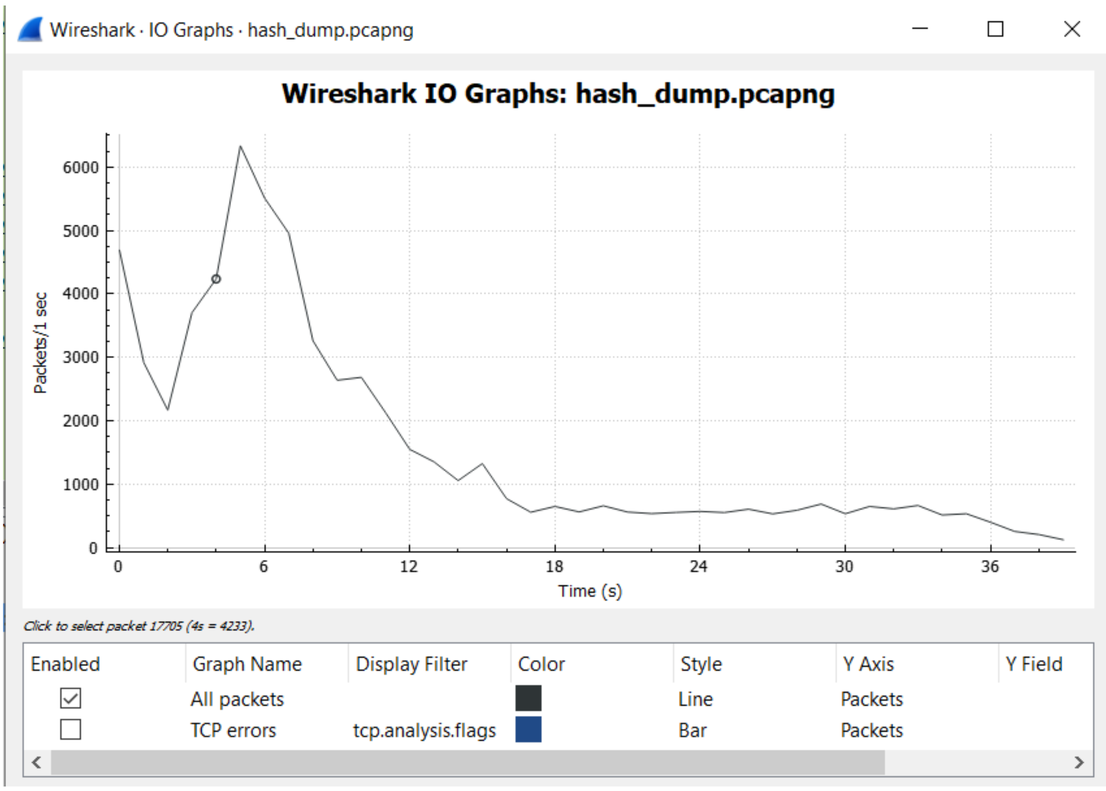

Figure 34: PPS graph from Wireshark during IP hash based experiment

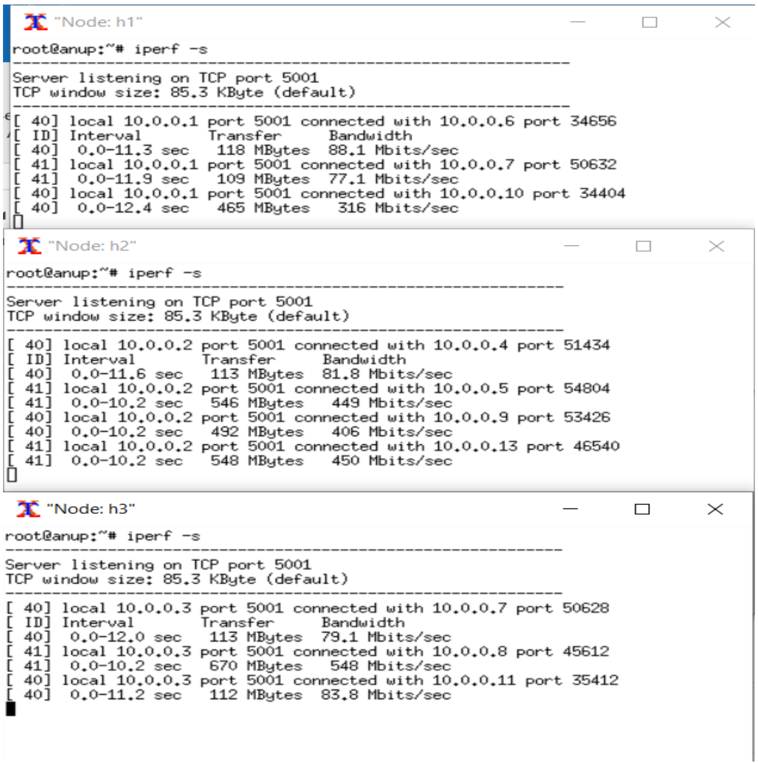

Figure 35: IPERF test in RR Based balancer

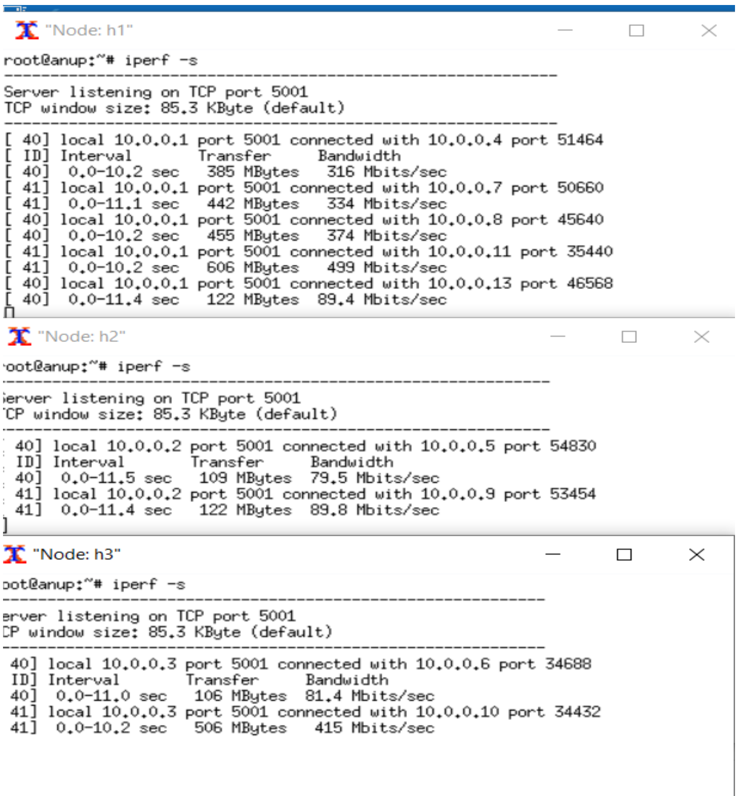

Figure 36: IPERF test in Weighted RR Based balancer

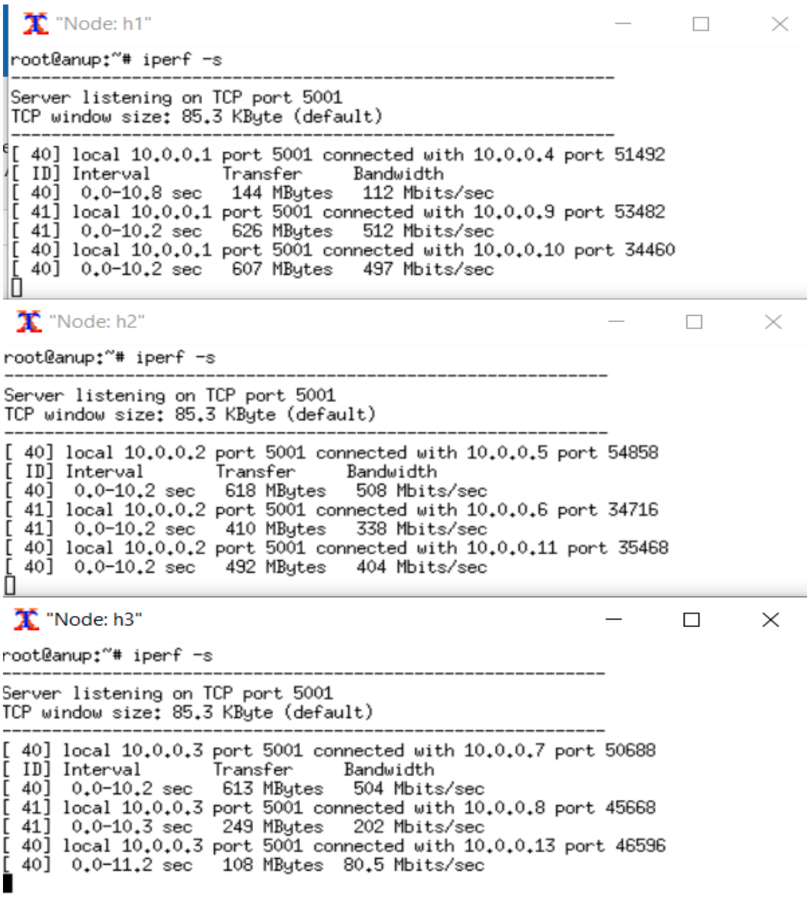

Figure 37: IPERF test in IP hash Based balancer

APPENDIX B: SOURCE CODE

1. **Implemented Hash based code for load balancer in SDN Framework:**

```
def key1 (self):
   ethp = self.first_packet
   ipp = ethp.find('ipv4')
   tcpp = ethp.find('tcp')
   return
ipp.srcip,ipp.dstip,tcpp.srcport,tcpp.dstport
  @property
  def key2 (self):
   ethp = self.first_packet
   ipp = ethp.find('ipv4')
   tcpp = ethp.find('tcp')
   return
self.server,ipp.srcip,tcpp.dstport,tcpp.srcp
ort
class iplb (object):
  def __init__ (self, connection, service_ip,
servers = []):
   self.service_ip = IPAddr(service_ip)
   self.servers = [IPAddr(a) for a in
servers]
   self.con = connection
   self.mac = self.con.eth_addr
   self.live_servers = {} # IP -> MAC,port
   self.last_server = 0
   try:
     self.log =
log.getChild(dpid_to_str(self.con.dpid))
   except:
     self.log = log
   self.outstanding_probes = {} # IP ->
expire_time
   self.probe_cycle_time = 5
   self.arp_timeout = 3
   self.memory = {} #
(srcip,dstip,srcport,dstport) ->
MemoryEntry
   self._do_probe()
 def _do_expire (self):
   t = time.time()
   for ip,expire_at in
self.outstanding_probes.items():
     if t > expire_at:
       self.outstanding_probes.pop(ip,
None)
       if ip in self.live_servers:
         self.log.warn("Server %s down", ip)
         del self.live_servers[ip]
   c = len(self.memory)
   self.memory = {k:v for k,v in
self.memory.items()
            if not v.is_expired}
   if len(self.memory) != c:
     self.log.debug("Expired %i flows", c-
len(self.memory))
  def _do_probe (self):
   self._do_expire()
   server = self.servers.pop(0)
   self.servers.append(server)
   r = arp()
   r.hwtype = r.HW_TYPE_ETHERNET
   r.prototype = r.PROTO_TYPE_IP
   r.opcode = r.REQUEST
   r.hwdst = ETHER_BROADCAST
   r.protodst = server
   r.hwsrc = self.mac
   r.protosrc = self.service_ip
   e = ethernet(type=ethernet.ARP_TYPE,
src=self.mac,
           dst=ETHER_BROADCAST)
   e.set_payload(r)
   #self.log.debug("ARPing for %s",
server)
   msg = of.ofp_packet_out()
   msg.data = e.pack()

msg.actions.append(of.ofp_action_output(
port = of.OFPP_FLOOD))
   msg.in_port = of.OFPP_NONE
   self.con.send(msg)
   self.outstanding_probes[server] =
time.time() + self.arp_timeout

core.callDelayed(self._probe_wait_time,
self._do_probe)
  @property
  def _probe_wait_time (self):
      r = self.probe_cycle_time /
float(len(self.servers))
   r = max(.25, r) # Cap it at four per
second
   return r
  def _pick_server (self, key, inport):
   #self.last_server = (self.last_server + 1)
% len(self.live_servers)
   #return
```

```
self.live_servers.keys()[self.last_server]
    #return
random.choice(self.live_servers.keys())
    h=0
    n=len(self.live_servers)

    x="abc"
        q=0
    x=str(key[0])
    y=str(key[1])
    h=x
    h=h[7:]
    h="fc00::"+h
    t="fc00::100"
    self.log.debug("Source IP: %s" % h)
    self.log.debug("Destination IP: %s" %
t)
    #print'source IP=',key[0],'destinaton
IP=',key[1]
    o = map(int, x.split('.'))
    p= map(int, y.split('.'))
    q = (16777216 * o[0]) + (65536 * o[1])
+ (256 * o[2]) + o[3]
    w = (16777216 * p[0]) + (65536 * p[1])
+ (256 * p[2]) + p[3]
    #print'decimal equivalent=',q
    r=q^w;
    #
    self.last_server=r%n
    print'Hash value=',self.last_server
    return
self.live_servers.keys()[self.last_server]
    #return
random.choice(self.live_servers.keys())
    """
    n=len(self.live_servers)
    print'source IP=',key[0]
    x="abc"
    q=0
    x=str(key[0])
    x=x.split('.')
    q=(int(x[0]) << 24) + (int(x[1]) << 16)
+ (int(x[2]) << 8) + int(x[3])
    print'decimal equivalent=',q
    self.last_server=q%n
    print'Hash value=',self.last_server
    return
self.live_servers.keys()[self.last_server]
    #return
random.choice(self.live_servers.keys())"""

  def _handle_PacketIn (self, event):
    inport = event.port
    packet = event.parsed
    def drop ():
      if event.ofp.buffer_id is not None:
        # Kill the buffer
        msg = of.ofp_packet_out(data =
event.ofp)
        self.con.send(msg)
      return None
    tcpp = packet.find('tcp')
    if not tcpp:
      arpp = packet.find('arp')
      if arpp:
        if arpp.opcode == arpp.REPLY:
          if arpp.protosrc in
self.outstanding_probes:
self.outstanding_probes[arpp.protosrc]
            if
(self.live_servers.get(arpp.protosrc,
(None,None))
                == (arpp.hwsrc,inport)):
              # Ah, nothing new here.
              pass
            else:
              # Ooh, new server.
              self.live_servers[arpp.protosrc] =
arpp.hwsrc,inport
              self.log.info("Server %s up",
arpp.protosrc)
        return
      return drop()
    ipp = packet.find('ipv4')
    if ipp.srcip in self.servers:
      key =
ipp.srcip,ipp.dstip,tcpp.srcport,tcpp.dstport
      entry = self.memory.get(key)
        self.log.debug("No client for %s",
key)
        return drop()
      entry.refresh()
        mac,port =
self.live_servers[entry.server]
      actions = []
actions.append(of.ofp_action_dl_addr.set_
src(self.mac))

actions.append(of.ofp_action_nw_addr.set
_src(self.service_ip))

actions.append(of.ofp_action_output(port
= entry.client_port))
      match =
of.ofp_match.from_packet(packet, inport)
      msg =
of.ofp_flow_mod(command=of.OFPFC_
ADD,

idle_timeout=FLOW_IDLE_TIMEOUT,
```

```
hard_timeout=of.OFP_FLOW_PERMAN
ENT,
                      data=event.ofp,
                      actions=actions,
                      match=match)
     self.con.send(msg)
   elif ipp.dstip == self.service_ip:
     key =
ipp.srcip,ipp.dstip,tcpp.srcport,tcpp.dstport
     entry = self.memory.get(key)
     if entry is None or entry.server not in
self.live_servers:
       if len(self.live_servers) == 0:
         self.log.warn("No servers!")
         return drop()
       server = self._pick_server(key,
inport)
       self.log.debug("Directing traffic to
%s", server)
       entry = MemoryEntry(server, packet,
inport)
       self.memory[entry.key1] = entry
       self.memory[entry.key2] = entry
       entry.refresh()

     mac,port =
self.live_servers[entry.server]
     actions = []

actions.append(of.ofp_action_dl_addr.set_
dst(mac))

actions.append(of.ofp_action_nw_addr.set
_dst(entry.server))

actions.append(of.ofp_action_output(port
= port))
     match =
of.ofp_match.from_packet(packet, inport)
     msg =
of.ofp_flow_mod(command=of.OFPFC_
ADD,

idle_timeout=FLOW_IDLE_TIMEOUT,

hard_timeout=of.OFP_FLOW_PERMAN
ENT,
                      data=event.ofp,
                      actions=actions,
                      match=match)
     self.con.send(msg)
_dpid = None
def launch (ip, servers):
  servers = servers.replace(","," ").split()
  servers = [IPAddr(x) for x in servers]
  ip = IPAddr(ip)
```

2. Mininet networkTopology Code:

```
from mininet.net import Mininet
from mininet.node import Controller,
RemoteController, OVSController
from mininet.node import
CPULimitedHost, Host, Node
from mininet.node import
OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call
def myNetwork():
    net = Mininet( topo=None,
               link=TCLink,
               build=False,
               ipBase='10.0.0.0/8'
               )
    info( '*** Adding controller\n' )
    c0=net.addController(name='c0',

controller=RemoteController,
                  ip='127.0.0.1',
                  protocol='tcp',
                  port=6633)
    info( '*** Add switches\n')
    s1 = net.addSwitch('s1',
cls=OVSKernelSwitch)
    info( '*** Add hosts\n')
    h1 = net.addHost('h1', cls=Host,
ip='10.0.0.1', defaultRoute=None)
    h2 = net.addHost('h2', cls=Host,
ip='10.0.0.2', defaultRoute=None)
    h3 = net.addHost('h3', cls=Host,
ip='10.0.0.3', defaultRoute=None)
    h4 = net.addHost('h4', cls=Host,
ip='10.0.0.4', defaultRoute=None)
    h5 = net.addHost('h5', cls=Host,
ip='10.0.0.5', defaultRoute=None)
    h6 = net.addHost('h6', cls=Host,
ip='10.0.0.6', defaultRoute=None)
    h7 = net.addHost('h7', cls=Host,
ip='10.0.0.7', defaultRoute=None)
    h8 = net.addHost('h8', cls=Host,
ip='10.0.0.8', defaultRoute=None)
    h9 = net.addHost('h9', cls=Host,
ip='10.0.0.9', defaultRoute=None)
    h10 = net.addHost('h10', cls=Host,
ip='10.0.0.10', defaultRoute=None)
    h11 = net.addHost('h11', cls=Host,
```

```
ip='10.0.0.11', defaultRoute=None)
    h12 = net.addHost('h12', cls=Host,
ip='10.0.0.12', defaultRoute=None)
    info( '*** Add links\n')
    net.addLink(h1, s1)
    net.addLink(h2, s1, delay='10ms')
    net.addLink(h3, s1, delay='50ms')
    net.addLink(h4, s1)
    net.addLink(h5, s1)
    net.addLink(h6, s1)
    net.addLink(h7, s1)
    net.addLink(h8, s1)
    net.addLink(h9, s1)
    net.addLink(h10, s1)
    net.addLink(h11, s1)
    net.addLink(h12, s1)
    info( '*** Starting network\n')
    net.build()
    info( '*** Starting controllers\n')
    for controller in net.controllers:
        controller.start()
    s1.cmd("sysctl
net.ipv6.conf.all.disable_ipv6=0")
    h1.cmd("ifconfig h1-eth0 inet6 add
fc00::1/64")
    h2.cmd("ifconfig h2-eth0 inet6 add
fc00::2/64")
    h3.cmd("ifconfig h3-eth0 inet6 add
fc00::3/64")
    h4.cmd("ifconfig h4-eth0 inet6 add
fc00::4/64")
    h5.cmd("ifconfig h5-eth0 inet6 add
fc00::5/64")
    h6.cmd("ifconfig h6-eth0 inet6 add
fc00::6/64")
    h7.cmd("ifconfig h7-eth0 inet6 add
fc00::7/64")
    h8.cmd("ifconfig h8-eth0 inet6 add
fc00::8/64")
    h9.cmd("ifconfig h9-eth0 inet6 add
fc00::9/64")
    h10.cmd("ifconfig h10-eth0 inet6 add
fc00::10/64")
    h11.cmd("ifconfig h11-eth0 inet6 add
fc00::11/64")
    h12.cmd("ifconfig h12-eth0 inet6 add
fc00::12/64")
    info( '*** Starting switches\n')
    net.get('s1').start([c0])
    info( '*** Post configure switches and
hosts\n')
    CLI(net)
    net.stop()
if __name__ == '__main__':
    setLogLevel( 'info' )
    myNetwork()
```

3. Code for simple HTTP server:

```
ps -ef | grep SimpleHTTPServer | awk
'{print $2}' | xargs kill -9
python -m SimpleHTTPServer 80
&>/dev/null &
ps -ef | grep SimpleHTTPServer | grep -v
grep
```

4. Code to check server response to be run from client:

```
import datetime
import requests
import time
url = "http://10.0.1.1/"
arr =
['1KB.txt','100KB.txt','1MB.txt','10MB.txt'
,'100MB.txt']
length = len(arr)
a = datetime.datetime.now()
for i in range(100):
        #time.sleep(1)
        f = arr[i%length]
        url1 = url + f
        try:
                r = requests.get(url1,
timeout=200)
                r.raise_for_status()
                respTime =
str(round(r.elapsed.total_seconds(),2))
                currDate =
datetime.datetime.now()
                currDate =
str(currDate.strftime("%d-%m-%Y
%H:%M:%S"))
                print(currDate + " " +
respTime + " " + f)
        except
requests.exceptions.HTTPError as err01:
                print ("HTTP error: ",
err01)
        except
requests.exceptions.ConnectionError as
err02:
                print ("Error connecting:
", err02)
        except
requests.exceptions.Timeout as err03:
                print ("Timeout error:",
err03)
        except
requests.exceptions.RequestException as
err04:
                print ("Error: ", err04)
```

```
b = datetime.datetime.now()
c = b - a
#print ("Total time for execution")
#print c.total_seconds() * 1000
```