SERVICE BASED LOAD BALANCE MECHANISM USING SOFTWARE-DEFINED NETWORKS

AHMED ABDELAZIZ ABDELLTIF OSMAN

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2017

SERVICE BASED LOAD BALANCE MECHANISM USING SOFTWARE-DEFINED NETWORKS

AHMED ABDELAZIZ ABDELLTIF OSMAN

THESIS SUBMITTED IN FULFILMENT
OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE AND
INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2017

UNIVERSITY OF MALAYA

ORIGINAL LITERARY WORK DECLARATION

Name of Candidate: **Ahmed Abdelaziz Abdelltif Osman**

Matric No:**WHA120043**

Name of Degree:**Doctor of Philosophy**

Title of Project Paper/Research Report/Dissertation/Thesis ("this Work"):

Service Based Load Balance Mechanism Using Software-Defined Networks

Field of Study:**Computer Networking**

I do solemnly and sincerely declare that:

(1) I am the sole author/writer of this Work;
(2) This Work is original;
(3) Any use of any work in which copyright exists was done by way of fair dealing and for permitted purposes and any excerpt or extract from, or reference to or reproduction of any copyright work has been disclosed expressly and sufficiently and the title of the Work and its authorship have been acknowledged in this Work;
(4) I do not have any actual knowledge nor do I ought reasonably to know that the making of this work constitutes an infringement of any copyright work;
(5) I hereby assign all and every rights in the copyright to this Work to the University ofMalaya ("UM"), who henceforth shall be owner of the copyright in this Work and that any reproduction or use in any form or by any means whatsoever is prohibited without the written consent of UM having been first had and obtained;
(6) I am fully aware that if in the course of making this Work I have infringed any copyright whether intentionally or otherwise, I may be subject to legal action or any other action as may be determined by UM.

Candidate's Signature Date:

Subscribed and solemnly declared before,

Witness's Signature Date:

Name:

Designation:

ABSTRACT

The proliferation of servers on the Internet has led to the emergence of server load balance as an important service in cloud aiming to optimize resource usage, maximize throughput and minimize the response time. Server load balance is the process and technology that distributes incoming requests among several servers in order tominimizethe response time and maximize the utilization of servers. The existing schemes of the load balance (dynamic or static) do not consider the service types as well as the size of the request. Besides, these schemes are implemented either in dedicated hardware devices called load-balancer or built intothe Operating System (OS) such as Linux Virtual Server (LVS). It is difficult to customize the built-in LB scheme during runtime. Additionally, load balancer experiences problems due to the same scheme being used for different type of services. In the cloud, most Service Providers (SP) host various kinds of services that require different load balancing schemes. This requires theinstallation of additional load-balancers for each service or a manual reconfiguration of the device to handle the new services. Such operation is time-consuming and expensive(Marc Koerner & Kao, 2012).

To address aforementioned problems, we proposed a service based load balance (SBLB) mechanism using Software-Defined Networks (SDN). The SDN controller is leveraged to provide online flow classification. The proposed mechanism is evaluated using benchmarking experiments and validated using a statisticalmodel. We investigate the impact of the different type of requests (compute and data) on SBLB mechanism in homogeneous and heterogeneous environments. The results demonstrate that SBLB can provide faster response time, higher throughput as compared to the other load balance solutions. For example, SBLB mechanism can reduce the average response time (ART) up to 5% and reply time (RT) up to 3% as compared to HAproxy load balance in the

homogeneous environment. In the heterogeneous environment, SBLB mechanism demonstrates 51% decrease in average response time and 47% decrease in reply time as compared to theHAproxy load balancer. Besides, our proposed mechanism also outperforms round robin algorithm in both environments. SBLB shows 7% increases in request per second (RPS) in the homogeneous environments and 20% increases in request per second in the heterogeneous environment as compare to aRound-Robin algorithm.

ABSTRAK

Kepesatan pelayan di Internet membawa kepada kemunculan pengimbangan beban pelayan sebagai suatu perkhidmatan yang penting dalam awan. Perkhidmatan ini bertujuan untuk mengoptimumkan penggunaan sumber, memaksimumkan pemprosesan dan meminimumkan masa tindak balas. Pengimbangan beban pelayan adalah proses dan teknologi yang mengedarkan permintaan di antara beberapa pelayan untuk mengurangkan masa tindak balas dan memaksimumkan penggunaan pelayan. Skim-skim pengimbangan beban yang sedia ada (dinamik atau statik) tidak mengambil kira jenis perkhidmatan serta saiz permintaan. Selain itu, skim yang sedia ada dilaksanakan sama ada dalam peranti perkakasan khusus dipanggil beban pengimbang atau dibina dalam Sistem Operasi (OS) seperti *Linux Virtual Server* (LVS). Ia adalah sukar untuk mengubah skim LB yang terbina dalam masa pemprosesan. Tambahan pula, perkakasan pengimbangan beban mengalami masalah akibat skim yang sama digunakan untuk pelbagai jenis perkhidmatan. Dalam awan, kebanyakan pembekal perkhidmatan (SP) menyediakan pelbagai jenis perkhidmatan yang memerlukan skim pengimbangan beban yang berbeza. Oleh demikian, ia memerlukan pemasangan tambahan pengimbangan beban bagi setiap perkhidmatan atau konfigurasi secara manual peranti untuk mengendalikan perkhidmatan baru. Operasi tersebut memakan masa dan mahal. Bagi menangani masalah di atas, kami mencadangkan satu mekanisme pengimbangan beban berasaskan perkhidmatan (SBLB) dengan menggunakan *Software-Defined Networks* (SDN). Pengawal SDN ditambahbaik untuk menyediakan klasifikasi aliran dalam talian. Mekanisme yang dicadangkan dinilai dengan menggunakan eksperimen penanda aras dan disahkan menggunakan pemodelan statistik.Kami menyiasat impak mekanisme SBLB ke atas permintaan yang berbeza. (pengkomputeran dan data) di persekitaran homogen dan heterogen. Keputusan menunjukkan bahawa SBLB dapat mempercepatkan masa tindak balas dan menawarkan pemprosesan yang lebih tinggi

berbanding dengan pengimbangan beban yang lain. Sebagai contoh, mekanisme SBLB boleh menurunkan purata masa tindak balas (ART) sebanyak 5% dan masa balasan (RT) sebanyak 3% berbanding pengimbangan beban HAproxy di persekitaran homogeny. Dalam persekitaran heterogen, mekanisme SBLB menunjukkan penurunan 51% dalam purata masa tindak balas dan 47% dalam masa balasan berbanding pengimbangan beban HAproxy. Selain itu, mekanisme yang dicadangkan juga mengatasi algoritma *round robin* dalam kedua-dua persekitaran. SBLB menunjukkan kenaikan 7% dalam permintaan sesaat (RPS) di persekitaran homogen dan peningkatan 20% RPS di persekitaran heterogen berbanding algoritma round robin.

ACKNOWLEDGEMENTS

All Praise and thanks to ALLAH for this help and guide me to complete this thesis. I owe my deepest gratitude to my supervisor Dr. ANG TAN FONG for his advice, guidance, and supervision from the very early stages of this study through to the completion of this thesis. This research will never be accomplished without his supervision. Ialso like to extend my gratitude to my Co-supervisors, Prof Dr. ABDULLAH GANI and for his deep commitments and continued help and support.

I would also like to express my sincerest gratitude and special appreciation to my Father. No words can express my real feelings, so I dedicate my first achievement in my life as a small gift to him. My hearty thanks must go to my mother for all the prayers that help me to go through this lonely journey of research.

I am very thankful and grateful to my uncle Salah for his great support in various possible ways. My special thanks to my wife Selwan for her endless love and support during my study journey

I would like to express my deep appreciation to my dear lab friends, who provided so much support and encouragement throughout this research and studies process. I am also grateful to the University of Malaya for giving me the opportunity to further my study at FCSIT, UM.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACL	:	Access Control Lists
ACO	:	Ant Colony Optimization
ADC	:	Application Delivery Controller
API	:	Application programing Interface
ARP	:	Address Resolution Protocol
ART	:	Average Response Time
AWS	:	Amazon Web Service
CC	:	Cloud Computing
CLS	:	Command Line Interface
CPU	:	Central Process Unit
CPU	:	Central Processing Unit
CR	:	Compute Request
URL	:	Uniform Resource Locator
CURL	:	Client Uniform Resource Locator
DC	:	Data Center
DLBM	:	Dynamic Load Balance Module
DNS	:	Domain Name System
DPI	:	Deep Packet Inspection
DR	:	DataRequest
DSR	:	Direct Server Return
FSM	:	Finite State Machines
FTP	:	File Transfer Protocol
FTP	:	File Transfer Protocol
FV	:	Flow Visor

GSLB : global server load balancing

GUI : Graphical User Interface

HTTP : Hypertext Transfer Protocol

HTTP : Hypertext Transfer Protocol

HTTPS : Hypertext Transfer Protocol Secure

IANA : Internet Assigned Numbers Authority

ICMP : Internet Control Message Protocol

LB : Load Balance

LBaas : Load-Balancing-as-a-Service

LLDP : Link Layer Discovery Protocol

MAC : Media Access Control Address

ML : Machine Learning

MM : Monitoring Module

NAT : Network Address Translation

NBI : North Bound Interface

NetPDL : Network Packet Description Language

NFV : Network Function Virtualization

NLB : Network Load Balance

NOX : Network Operating X

OF : OpenFlow

OFS : OpenFlow Slice Algorithm

OS : Operating System

OVS : Open Virtual Switch

OVS : Open vSwitch

OVSDB : Open vSwitch Database Protocol

OVX : Open VirteX

P2P : Peer-to-Peer

QoS : Quality Of Service

RAM : Random Access Memory

RAM : Random-access memory

RPS : Request Per Second

RR : Reply Rate

RRA : Round Robin Algorithm

RT : Reply Time

RTP : Real-time Transport Protocol

SBI : South Bound Interface

SBLB : Service Based Load Balance

SCM : Classification Module

SDDC : Software-Defined Data Center

SDN : Software Defined Networking

SIP : Session Initiation Protocol

SLB : Software Load Balance

SNAT : Secure Network Address Translation

SNMP : Simple Network Management Protocol ()

SP : Service Provider

TC : Traffic classification

TCP : Transmission Control Protocol

TR : Transfer Rate

TS : Table Service

UDP : User Datagram Protocol

VDC : Virtual Data Center

VE : Virtual Environment

VIP : Virtual IP address

VM : Virtual Machine

LIST OF APPENDICES

CHAPTER 1: **INTRODUCTION**

The expansion growth of cloud ecosystem (Fortis, Munteanu, & Negru, 2012) has influenced modern software vendors to develop their software on cloud environment and deploy them on a number of server instances. It resulted in an increase in the number of servers that hosted various types of the services. In turn, the servers can be accessed by multi-tenant users from different locations (Krebs, Momm, & Kounev, 2012). Such users expect to get a faster response regardless of the size of the request, type of services, or the number of requests. This raises the issue of balanced distribution, control, and utilization of the available resources over the system's environment. To address this issue, Load Balance (LB) (Randles, Lamb, & Taleb-Bendiab, 2010) emerged as a technique used for distributing the incoming traffic among various servers in order to minimize the response time and maximize the utilization of the servers.

This technique started in the mid-1990s (Bourke, 2001) due to the rapid increase of websites and additional servers that raised the need for the websites to be expanded. The development of LB in the cloud has been fast since the introduction of Domain Name System (DNS) LB (O'neil, Nerz, & Aubin, 2000) that can balance the load between multiple web servers over the Internet. It used the Round-robin scheduling scheme to distribute the traffic without considering the server status and capacity. However, this approach became ineffective after the tremendous growth of services. Due to the above drawback, the clustering technique (Randles et al., 2010) was then used to distribute single application into multiple servers that improve the scalability of the systems. In line with the popularity of clustering technique, the importance of LB then rose to a higher status, and several load balancer devices and applications have been introduced over the last decade. Nevertheless, this technique has some limitations

where it does not support dynamic multiple scheduling algorithms and uses only one LB schema for all the services and applications

The rapid growth of Cloud Computing (CC) (Armbrust et al., 2010) in the last few years, has led to a massive increase in service requests to cloud servers. Thus, there is a need for cloud load balancing technique that maximizes application performance and enhances the reliability of the services. There are many cloud providers that offer cloud load balancing service, including Amazon Web Services (AWS), Google, Microsoft Azure and Rackspace. The Load Balance as a Service (LBaaS) is the commonly used model in the cloud ecosystem. Nonetheless, most of the LBaaS solutions not concern about the mutual intervention among applications deployed on the same server. Therefore, a server with multiple deployed applications needs a proper scheduling policy to guarantee the effectiveness of load balancing. Software-Defined Networking (SDN) (Kreutz et al., 2015) is believed to provide an effective load balancing for the dynamism of the traffic requests. By integrating the service awareness with SDN, the issues of load balancing can then be addressed. This chapter is divided into eight sections. In section 1, a brief background of the load balance techniques in the cloud as well as the related service based traffic classification is presented. The section also discusses Software Defined Networking (SDN) as the technology to enhance the traffic management in the cloud. In section 2, the motivation of the study is presented. Section 3 then highlights the research gap and identifies the statements of the problem addressed in this study. The objectives and the scopes of the research are discussed in sections 4 and 5 respectively. The proposed methodology is then presented in section 6 while the contributions are listed in section 7. The chapter then concludes with Section 8 where the thesis layout is given.

1.1 Background

1.1.1 Load Balance in Cloud

The proliferation of servers on the Internet (Abdelzaher & Lu, 2000) led to the emergence of load balance as an important service in cloud aiming to optimize resource usage, maximize throughput and minimize the response time. Load balance in the cloud is the process and technology that distributes incoming requests among several servers in order to minimize the response time and maximize the utilization of servers (Chang & Tang, 2010a). Figure 1.1 illustrates the load balancing system architecture, whereas the load balancer is located in between the firewall and the server pool. Users send their requests to the server pool through a Virtual IP (VIP), and the load balancer will distribute the load among the servers in the pool upon receiving the requests.

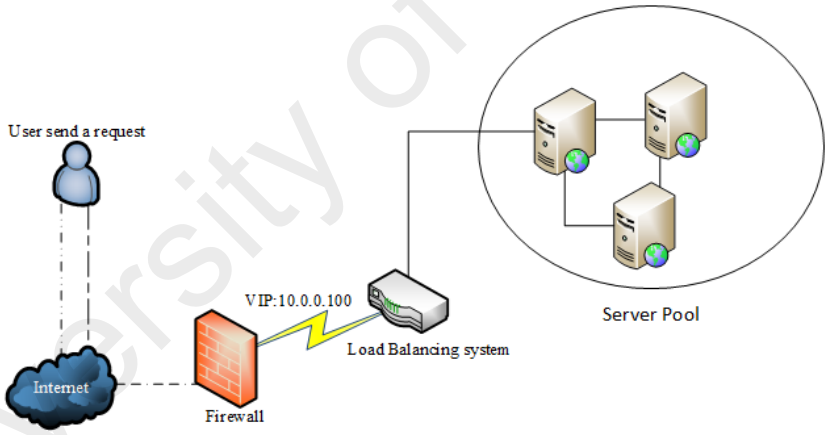

Figure 1.1 Load balance architecture in cloud environment

There are various types of load balance implementations namely; Application Load Balance (ALB) (H.-j. LIN, PENG, & LI, 2007), Hardware Load Balance (HLB) (Gandhi et al., 2015) and Application Delivery Controller (ADC) (Chiong, 2013). ALB is a software based load balance service that is integrated into the server's operating system such as Windows servers or Linux servers. ALB has the capabilities to distribute the network traffic between clustered servers or server farms. Application developers can customize the load balance policy (algorithm) based on the characteristics of the

related application as well as the resource information. HLB has emerged as a powerful solution that increases the system's scalability and availability (Saito, Bershad, & Levy, 2000). The hardware load balancer can distribute the load across multiple servers and redirect the load to other servers in case of server or application failure. The manageability function allows the administrator to dynamically scale-out the load balance by adding more servers to the server pool on the fly. HLB uses thespecialized processor to provide high capacity, but, it is expensive and characterized by alack of flexibility in terms of handling different types of traffic requests. HLB has evolved into Application Delivery Controllers (ADCs) over the past 10 years. Typically, in the data center, ADC is a load balancing device that sits between the firewall and the Web farm. ADC has the ability to inspect the packet headers and distribute the traffic to the selected servers based on this information. In addition, ADC has a monitoring system that allows the checking of the server's health status and ADC can provide content-aware service that is used for quality of service (QoS) and security purposes. However, ADC does not have the ability to identify exactly the types of the service. The importance of the service aware is essential in order to maximize resource utilization and optimize the execution time of the services.

1.1.2 Services Based Classification

Identifying the type of the service is a critical network processing task because of the complexity and dynamic characteristics of the network traffic. Besides, dramatic increases of services in various forms that are deployed in the cloud, along with dynamic communication protocols, have attracted numerous researchers to introduce several *service classification* solutions. The main objective of service classification is to identify which service is offered by the servers using different approaches of the Traffic Classification (TC) (Bernaille, Teixeira, Akodkenou, Soule, & Salamatian, 2006). TC is a mechanism that is used to identify the type of services or applications for different

purposes such as security, Quality of Service (QoS) and/or network statistics. There are a variety of traffic classification approaches which include Port-based classification (Dainotti, Pescape, & Claffy, 2012), Deep Packet Inspection (DPI) (Becchi, Franklin, & Crowley, 2008) and statistical information that are widely used with Machine Learning (ML) algorithms (Nguyen & Armitage, 2008). In the port-based technique, the port is used to define the service type. Even though the technique is fast, it is still not accurate. This is due to the fact that, most services are running on dynamic ports or communicate over the HTTP. Moreover, the port-based approach can lead to misclassification as new services may end up either reusing the port number or randomly selecting port numbers or in some cases a user selecting a port based on their preferences. Hence, this is currently no longer used because of the above-mentioned limitations. The DPI approach depends on the inspection of the actual payload of the packet to identify the type of service or application. However, the approach is more accurate, it is still slow and involves high computation cost.At the same time, it requires manual signature maintenance. Furthermore, most services today are transmitted over encrypted channels such as HTTPS in which the payload cannot be accessed. In order to to overcome this limitation, statistical information with Machine learning (ML) was then proposed as an alternative. This approach identifies the network features and uses them to define the related type of services. These features, attributes of flows or packets, are statistical information that can be calculated to label the respective traffic.

1.1.3 Software Defined Network (SDN)

In the past few years, Software Defined Network (SDN) (Nunes, Mendonca, Nguyen, Obraczka, & Turletti, 2014) was raised as a new generation network architecture to address the problems that were encountered by traditional networks such as security, traffic management, and virtualization. SDN architectures decouple network control and forwarding functions, enabling network control to become directly

programmable and the underlying infrastructure to be abstracted from applications and network services. Such abstraction provides unified cloud resources that can be managed by the SDN controller. SDN controller manages the edge devices (switch and router), defines network policy topology centrally, and manages multiple interfaces southbound protocols. SDN comes with several advantages(Feamster, Rexford, & Zegura, 2013): 1) Decoupling control plane from the data plane whereas a switch is responsible for forwarding the packet based on the instructions of thecontroller. 2) Centralizing control of the network by the controller that has a complete view of the network. 3) Open interfaces between the control plane and the data plane by using astandard protocol such as OpenFlow. 4) Programmability of the network which allows network administrators to develop their own network applications module on top of the controller.

1.2 Research Motivation

Grand View Research (G. V. Research, 2016), a market research company, reported that traffic in the data centre is expected to reach 9965 Exabyte by 2020 as compared to 4515 Exabyte in 2015, and market penetration of cloud-based data centre traffic is anticipated to increase from over 58% to over 75% by 2020. Such explosive growth of traffic is an evidence of the increasing use of load balance services that are required to balance the traffic on the cloud.

Another study from Microsoft search lab (Parveen Patel et al., 2013) compared the Internet traffic ratio and inter-service traffic ratio in one week for 8 data centers. The study discovered that 44% of the total traffic is VIP which represents the load balance traffic or SNAT, 30% is inter-service traffic while the rest, 26% is to the Internet traffic. Since most traffic must pass through the load balancer, the authors in (Poddar, Vishnoi, & Mann, 2015) showed that 70% of total VIP traffic is managed within the same data center. The load balancers are usually placed after a firewall and handle all VIP traffic.

The dedicated hardware load balancers (HLB) are too expensive and roughly cost about the US $80,000. The study was conducted in a data center containing 40.000 servers with 100 Tbps of internal data center traffic and more than 400 Gbps of external traffic. In order to handle this amount of traffic, 40 load balancers were required for external traffic and 10,000 load balancers deployed for the intra-DC traffic, and in turn, both incurred a cost of about (The US $3.2 million) and (The US $800 million) respectively.

Figure 1.2 shows the regional SDN segment forecast from 2015 to 2020 ($ billion). The graph illustrates the rapid growth of SDN in the Asian Pacific, North America and Europe. SDN has the potential to simplify network management and enable innovation and evolution of the computer networks. It is a natural fit for load balance because the controller has a global view of network resources and knowledge of application requirements to optimize the load. Thus, SDN brings new possibilities for improving balancing techniques that faced several problems in the traditional network

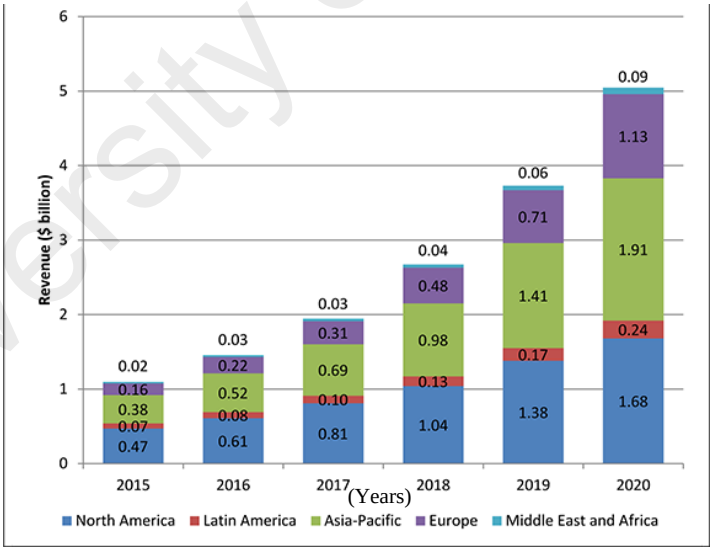

Figure 1.2Regional SDN segment forecast

load.

Today's solutions in the cloud for load balancing are effective but have limited flexibility in terms of customization. Typically, in the cloud environment, service providers host various types of services and applications with multi-tenants services that

require specific load balance schemes. Therefore, customizing the load balance system is quite hard and might also require different load balancers for each of the offered services, and that might be too costly

1.3 Statement of the Problem

The network congestion and server overloading have become a serious problem in most of the cloud environments due to the increment of network traffic. Consequently, load balancing for the cloud has turned out to be a very important research area (Handigol, Flajslik, Seetharaman, McKeown, & Johari, 2010). Several types of the load balance approaches and techniques using SDN technology are proposed. Such techniques/approaches aim to minimize the response time and maximize the throughput without causing overhead to the SDN Controller.

Typically, the load balance scheme is categorized into two; *Static* and *Dynamic*. The static scheme (Venkata Krishna, 2013) distributes the load without considering nodes capacity such as the server processor, RAM and links' bandwidth. However, thestatic scheme is simple to implement, with less overhead and suitable for homogenous servers but generally is not flexible, and is incapable of considering the dynamic changes to the attributes. For example, if one server received a huge number of tasks, after acertain time another task will be sent to the same server regardless of the capacity of the server or size of the task. Dynamic scheme (Guo et al., 2014) on the other hand, distributes the load based on the current status of network nodes. This means that at the run time, the load-balance system checks the server's load and links' capacity. Nevertheless, dynamic schema neglects the type and the size of the user request and can use only one algorithm (Handigol, Seetharaman, Flajslik, McKeown, & Johari, 2009) for all different services. For example, in thecloud, the load balancer is normally configured with one algorithm; least connections to deal with different services like HTTP and FTP that require adifferent schema. Typically, the request

processing time of each service is different, thus, taking into account the service type is important to provide optimal load balance.

The existing schemes (dynamic or static) do not consider the service types as well as the size of the request. Besides, these schemes are implemented either in dedicated hardware devices called load-balancer or built into the Operating System (OS) such as Linux Virtual Server (LVS)(W. Zhang, Jin, & Wu, 1999) or Microsoft Network Load Balancing (NLB) (Dutta, Vidovic, & Vrsalovic, 2003). It is difficult to customize the built into LB scheme during runtime. Additionally, load balancer experiences problems due to the same scheme being used for different type of services. In the cloud, most Service Providers (SP) host various kinds of services that require different load balancing schemes (Ragalatha P, 2013). In turn, needs installation of additional load-balancers for each service or a manual reconfiguration of the device to handle the new services. Such operation is time-consuming and expensive (Marc Koerner & Kao, 2012). In addition, to identify the type of the service for each request, online traffic classification method must be implemented. As SDN controller has the ability to view and monitoring network nodes via OpenFlow messages, therefore, designing and implementing online traffic classification can be achieved with minimum network overhead,

Although, several studies (Chou, Yang, Hong, Hu, & Jean, 2014), (Bays & Marcon, 2011), (Shang et al., 2013) have proposed load balancing solutions based on the SDN technology in the cloud. However, all these studies have neglected the various types of services that require different load balancing schema.

1.4 Statement of the Objectives

The aim of this study is to utilize the SDN technology in providing an effective service-based load balancing mechanism that maximizes the throughput and minimizes the response time. The objectives of this thesis are listed below:

1. To perform a gap analysis review on the approaches/techniques of the SDN load balancing schemes in the cloud.

2. To propose a service based load balancing mechanism with the SDN technologyto maximize throughput and minimize the response time.

3. To leverage the OpenFlow protocol for providing online traffic classification that identifies service types

4. To evaluate the proposed solution and compare the performance with the existing load balancing solutions.

1.5 Scope of the research

Server Load Balance (SLB) is widely implemented in the cloud for various purposes. For example, it can be used for Quality of Service (QoS) or during the DDoS attacks as a means of mitigation by leveraging the session's persistence. In addition, a load balancing system in software or hardware is usually designed with additional functions to address several issues such as server's scalability, availability, and security. Therefore, in this thesis; we focus on a load balancer that distributes the incoming traffic among a number of servers while considering the type of services. This is done so as to provide a dynamic load balancing scheme that minimizes the response time and maximizes the throughput. In order to achieve the research objectives during the given period, it is necessary to outline the scope and limitation of this research work;

- The study on the technology includes a load balancing system that can be implemented in thedata center.
- The OpenFlow protocol version deployed for this research is OpenFlow 1.3
- The evaluation of the proposed technique is carried out in the cloud environment using OpenStack.

- The metrics used for the experiment are Response Time (RT), Reply Time (RT), and Request per Second (RPS)

1.6 Proposed Methodology

The research is basically conducted as an empirical research study that is comprised of four phases which are as depicted inFigure1.3.

Phase 1: The research begins with a study of cloud environment to identify the current relevant issues and problems in terms of load balancing. Then, the existing load balancing solutions are explored. A gap analysis review on the approaches/techniques of the SDN load balancer in the cloud is performed. Subsequently, the problem statements and research objectives are defined.

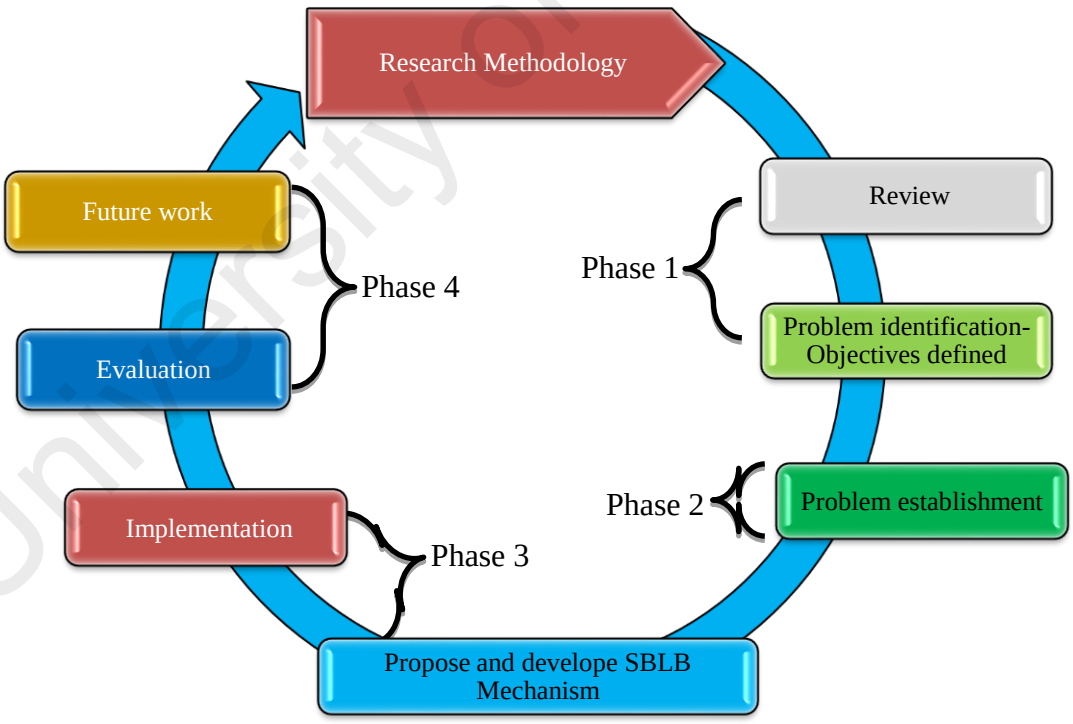

Figure 1.3 Research Methodology

Phase 2: In this phase, the problem is analyzed by conducting numbers of empirical experiments. Then, a mathematical representation of the load balancing

problems is formulated. After that, a Service Based Load Balance (SBLB) mechanism using the Software Defined Networks (SDN)architecture is proposed.

Phase 3: The proposed service based load balancing mechanism is developed, and a suitableSDN controller is selected for the implementation.

Phase 4: In this phase, several experiments are conducted to evaluate the performance of the SBLB. The proposed mechanism is evaluated based on several metrics in homogeneous and heterogeneous environments.The performance analysis is conducted between the proposed mechanism and existing load balance solutions. In addition, the future works are highlighted.

1.7 Contribution of the research

The research study provides information on the issues of SLB using SDN technology which provides several solutions to current cloud problems. Further, we will show the virtues of this new technology to enhance the performance of load balance.

- Designing and implementing dynamic load based on the service type.
- Besides, this study also enhances the OpenFlow Protocol by extending SDN controller and OpenFlow switch functions to support online traffic classification.
- A dynamic load balance algorithm that calculates the load of the each host based on request type is developed. This algorithm adjusts the parameters according to theservice type.

1.8 Layout of the thesis

Table 1.1 presents a summary of the contents of the thesis. This thesis includes seven chapters that are organized as follows:

Table 1.1 Summary of the Thesis Layout

Chapter	What?	Why?	How?
1	Introduction	• To give a brief background about the research • To show the motivation of the study • To define the problem and state the objectives • To describe the thesis layout	▪ Explain the research title ▪ Illustrate the motivation and formally writing the statement of problem and set the objectives
2	Literature Review	• To investigate the pros and cons of existing solutions • To review the currently used techniques • To present the open issues and challenges	▪ Perform a gap analysis review ▪ Explain in details the used architecture (SDN) ▪ Point out the issue that is addressed in the study
3	Problem Analysis	• To deeply understand the impact of user requests into load balancing system for the purpose of analyzing the problem	▪ Empirical study using simulation tools ▪ Implement/Conduct mathematical analysis
4	SBLB mechanism design and implementation	• Giving the clear understanding of the proposed system • Explain the system's architecture in details • Show implementation steps	▪ Elaborations of the system modules ▪ Explaining the system
5	Evaluation of the SBLB mechanism	• Presents the experiment's setup • Analysis, testing and comparison study of the proposed system to respective benchmarks	▪ Simulate the cloud environment (configuring Pools, Members, VIPs and controller modules) ▪ Explaining the tools used for evaluating the proposed solution ▪ Generating traffic and measure the metrics (*Response Time (RT), Reply Time (RT), Request per Second (RPS)*
6	Results and discussion	• Compared the results with existing load balance systems • Highlight the effectiveness of the proposed system	▪ Comparing of thethe performance of the proposed mechanism with the existing software load balancer such as (HAproxy). ▪ Compared SBLB algorithm with Round-Robin (RRA)
7	Conclusion	• Summary of the research findings • Show the limitations and future work	▪ point out the significance of the work reported in this thesis

Chapter 2presents a comprehensive review of the load-balance solutions that uses SDN. The chapter begins with a brief overview of load balancing schemes in the cloud, followed by the explanation of the SDN technology. Then, the load balancing

solutions in the SDN are presented, and the taxonomy is derived. A gap analysis review is conducted for the solutions. Besides, the different types of traffic classification techniques are investigated. The chapter ends with a number of open research issues and challenges with respect to load balancing in the cloud.

Chapter 3 presented an analysis of the problem to show the impact of user's requests on a load balancing system and the related host's response. For provingthat, several experiments were conducted in the Mininet, and various types of requests were generated.

Chapter 4 describes the development and implementation of the proposed service-based load balance mechanism. A presentation of thesystem architecture is given together with the functionalities of the three sub-modules namely; Service-based Classification, Load Balancing, and Monitoring.Besides, a use-case diagram is illustrated to show the interaction between the client and the proposed load balance mechanism.

Chapter 5 presents the data collection for examining the proposed SBLB. First, the experimental setup and the components of the experimentsare explained. The benchmarking that is used to evaluate the mechanism in homogenous and heterogeneous environments is presented. In addition, the data collection methods and the statistical model that are used to evaluate the proposed mechanism is presented in this chapter

Chapter 6 presents the performance of SBLB mechanism and compares it with other load balance solution.Three parameters namely Average Response Time (ART), Reply Time (RT), and Request Per Second (RPS) are used to evaluate the performance of SBLB. The experiments are carried out in homogeneous and heterogeneous environments.

Chapter 7 discusses the outcomes of the research and how the objectives have been achieved.Then, the limitations and delimitations of the proposed mechanism are discussed. The chapter ends with the suggestions of the future research directions.

CHAPTER 2: **LITERATURE REVIEW**

This chapter aims to conduct a review on Server Load Balancing (SLB) in the cloud and analyze existing solutions that are related to the problem. First, we discuss the development stages of the SLB in the cloud. Then, we present the fundamental concept of the SDN along with the comparison of traditional and SDN networks in terms of the load balancing service in the cloud. Subsequently, we classify the current SDN-SLB solutions and introduce a thematic taxonomy based on different criteria including; approaches/techniques, controller and controller modules, algorithms and experimental environment. We discussed in detail the latest existing SDN-SLB solutions. Such solutions depict the current system state of SDN-SLB in both academics and industries. Moreover, we present a discussion on traffic classification approaches that are implemented to provide service classifications. Finally, we summarize the chapter by discussing the challenges of SDN-SLB as well as open issues and future research direction that highlight the problem of the study.

The remainder of this chapteris organized as follows: In section 2.1, we present a brief background of SLB in the cloud. The fundamental concept of the SDN layer architecture is illustrated in section 2.2. Then, a comparison between traditional SLB and SDN-SLB architecture is presented in Section 2.3. In section 2.4, we discuss in details the thematic taxonomy of the SDN-SLB. Subsequently, the existing state-of-the-art SDN-SLB solutions are presented in section 2.5. Traffic classification approaches that are used for the identification of service types are then shown in section 2.6. Section2.7 discusses the challenges, future directions, and open issues related to SDN-SLB. The chapter is then concluded in section2.8.

2.1 Load balance Background

The proliferation of servers on the Internet led to the emergence of the SLB as a valuable service in the cloud (Radojević & Žagar, 2011). Its aim is to avoid overload of the server, optimize resource usage, maximize throughput and minimize the response time. SLB is simply a process and technology that distributes incoming request among several servers. This technique was initiated in the mid-1990s when a rapid increase of websites and additional servers were required to expand the applications (Wellman, 2004). From that time onwards, this technique has rapidly evolved and became one of the important service models in the cloud namely load balancing as services (LBaaS) (Rahman, Iqbal, & Gao, 2014). Besides, clustering technique (Gonzalez, Rojas, Ortega, & Prieto, 2002) was introduced to improve the scalability of the systems and help distribute thesingle application to multiple servers. In this section, we discuss the evolution of load balancing in the cloud that is illustrated in Figure 2.1, from simple Domain Name System (DNS) (Mockapetris & Dunlap, 1988) service to a dedicated hardware device that provides multi functions along with load balancing service.

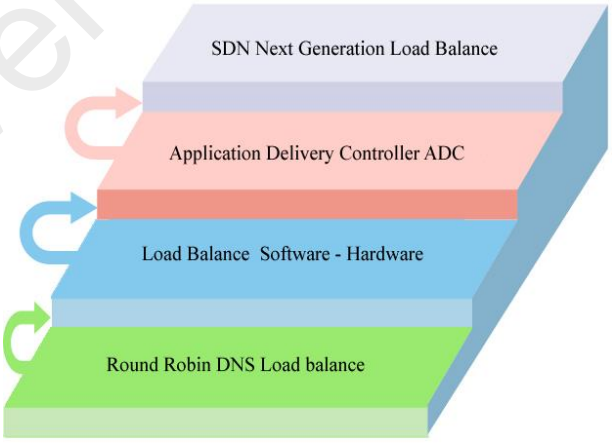

Figure 2.1 The evolution stages of load balance

2.1.1 Domain Name System (DNS)

Round-robin DNS is the first technique implemented to distribute the load across multiple servers over the Internet. The techniquewas used when a server is

overloadedby users'requests, and additional servers are deployed to balance the load and expand the applications. This process is executed in such a way that the user is obscured from the knowledge that there are multiple servers dealing with the requests (Cardellini, Colajanni, & Philip, 1999). In this approach, a single DNS name has a number of unique internal IP addresses, each of which represents a server that hosts the website. The usersends a request to a specific DNS system which consists of all servers' IPs that host the same website. The DNS system then selects an appropriate server in response to the user request. The DNS selects a server in a rotational and sequential manner, which is called Round-Robin DNS (Gulbrandsen & Esibov, 2000). When a user sends arequest to the website,the DNS forwards the request to the first IP. If the second user tries to access the website, the request will send to the second IP address, and other requests will be sent to the third IP and so forth.

Although this approach solves the scalability issue of the system that is hostedon distributed servers, it limits its reliability and efficiency (Bryhni, Klovning, & Kure, 2000). For instance, a number of servers that can add to the DNS system arelimited, and the system needs to be configured manually by the administrator through Command Line Interface (CLS) or Graphical User Interface (GUI). Another drawback is that the DNS works without the knowledge of the server's status (Bryhni et al., 2000), this means that the DNS can send a request to the server that is unavailable or overloaded. Recently, many solutions have been suggested to address this problem such as*lab named*(Schemer, 1995), a modified DNS solution that allowsreporting serverstatus periodically. But, the DNS only knows the servers based on IPs without considering the port and IP addresses which can be contained or stored in the caches of other name servers (McClain & Thatcher, 2004). Therefore, requests may keep on arrivingat the loaded server.

2.1.2 Hardware and Software load balance

2.1.2.1 Hardware load balancer (HLD) *(Load Balancer*)

Due to the restriction of the DNS load balance approach, HLD was introduced by several manufacturers in the mid-1990s (Ju, Xu, & Yang, 1995). Decoupling load balance function from application enables the DNS to use network layer techniques such as Network Address Translation (NAT) (Srisuresh & Egevang, 2001) or Direct Server Return (DSR) (Bansal, Warkhede, & Venketesan, 2012)(Kopparapu, 2002) to send inbound and outbound traffic to the servers. Such techniques are used to process the requests and replies to the client. In the case of using DSR, when the server responds, the load balancer will not translate the IP address of the server but will only change the MAC address. This approach is useful and performs better if the load balancing is the bottleneck. To configure DSR, the load-balancer and real server must be in the same Layer 2 domain, and loopback IP address must be configured in all physical servers. For the security purpose, NAT technique is used with a load-balancer, whereas IP and MAC addresses are translated. Therefore, users send their requests to a virtual IP (VIP) without knowing about the private IP addresses of the servers and get a response from the destination via NAT.

The load balancer introduced *server health-checking*(Kopparapu, 2002) that was not enabled in the DNS approach. By configuring time interval, the load balancer is capable of checking the availability of the server as well as its traffic load. For distributing the incoming traffic, load balancer implements a series of steps; first, it must ensure the availability of all servers by querying them via pings. If a server reply, it will be added to the available list, if a server fails to respond, the load balancer will consider this server as dead. Various query techniques are used, depending on the type of the load balancer or vendors' specifications. For example, healthy-check can be carried on layer 2 by sending Address Resolution Protocol (ARP) (Adelman, Kashtan, Palter, & Derrell,

2000) request to get the MAC address for a given IP address, or via layer 7, application layer, that is used for well-known applications such as *Hypertext Transfer Protocol* (HTTP) (Narendran, Rangarajan, & Yajnik, 2000) and *File Transfer Protocol* (FTP) (Narendran et al., 2000) servers.

2.1.2.2 Software Load Balancer (SLB)

Atypically, SLB can be implemented into server OS such as Windows Server 2016 (Huh & Seo, 2016) or Red Hat's High Availability Linux Server (Cash et al., 2016). Most of the existing solutions are focused on distributed network traffic between clustered servers or server farms. SLB is flexible for cloud visualization environment in which the servers have individual OSs, or share an operating system. SLB in a cluster environment that allows scaling of the network services where additional servers can be added dynamically to the cluster. SLB distributes the load between servers, while a server cluster provides fault tolerance in the system. For example, in Virtual Load Balancer that is implementedon a Linux server, the servers in a cluster listen to a "cluster IP" or VIP in addition to their physical IP address. The users send requests to the cluster IP. The system then selects a server from the cluster based on the load balancing policy. The NAT technique can be configured, but DSR is not used in SLB. One of the important features of SLB is that the application developers can customize the load balancing policy (algorithm) because of a variety of information about the server that can be used to determine which server the client should connect to. For example, least connection algorithm (Nuaimi, Mohamed, Nuaimi, & Al-Jaroodi, 2012) is widely implemented since the cluster has a count of how many sessions each server is already serving in a given session.

2.1.3 The Application Delivery Controller (ADC)

The proliferation of dynamic content led to the delivery of dynamic services, content-rich applications that need to understand the application-specific traffic. The traditional load balancer could not cope with these growing requirements. Therefore, LBH has evolved into Application Delivery Controllers (ADCs) (Salchow Jr, 2007) over the past ten years. Typically, in the data center, ADC is a device that sits between the firewall and a web farm to provide several tasks (Doron & Sekiguchi, 2013). One of these tasks is loading the traffic between web servers. ADC can inspect packet headers and distribute the traffic to a selected server based on this information. In addition, ADC comes with a monitoring system that allows the checking of the server's health status beyond the traditional health check approach that uses a *PingTool*(Jindal, Lim, Radia, & Chang, 2001). Such a monitoring system can be configured for specific health criteria to enhance the reliability and at the same time avoiding a potential disruption. Furthermore, ADCs provide more features, such as real-time and historical analysis for all traffics, with several metrics including round-trip time, latency (Ben-Shaul, Cidon, Kessler, Lev-Ran, & Unger, 2005) and bandwidth usage. Such features help administrators to identify the cause of a problem in the network and optimize the application server's performance. For example, the administrator can offload many of the computational-intensive tasks that affect the CPUs for a specific server and transfer these tasks to another server within the cluster. On the other hand, ADC can manage global server load balancing (GSLB) (Hsu, Cheung, & Jalan, 2013) i.e.load balancing of server clusters in different geographical locations, to provide high availability and faster response time. This section presented different phases of the development of the server load balance and showed the importance of this service in the cloud. Table 2.1 shows the pros and cons of the SLB approach mentioned above.

Table 2.1 Pros and cons of different SLB approach

SLB solutions	Pros	Cons
Domain Name System (DNS)	Allows to host a website into multi-servers	Presents a problem that limits its reliability and efficiency
Load balancer (Software and Hardware)	Provide healthy check for the servers and applications	Single point of failure
Application Delivery Controller (ADC)	Customize the load balancing policy (algorithm)	Enhance the reliability and avoiding a potential disruption.
SDN Load Balancer	Centralized view that provides effective monitoring of network resources	Causes delay due to PacketIn process time

2.2 SDN background

OpenFlow (OF) (McKeown et al., 2008), is the first implementation of SDN which was initiated in 2008 as a project at Stanford University by Professor Nick McKeown that put forward the concept of SDN (Haleplidis et al., 2015). In the same year, ACM SIGCOMM published a paper titled "OpenFlow: Enabling Innovation in Campus Networks" (McKeown et al., 2008). This paper introduced in detail the concept of OpenFlow. In December 2009, the first version of OpenFlow specification (1.0) was released to be used in commercial products. In March 2011, again Professor Nick McKeown et al., was responsible for the inception and establishment of Open Networking Foundation (ONF), which focused on the development of the SDN architecture. In April 2012, ONF released a white paper in SDN titled "Software-Defined Networking: The New Norm for Networks" (Fundation, 2012), whereas the three layer SDN architecture were introduced and gained widespread recognition in industry and academia. Figure 2.2 shows the SDN architecture. The ONF constitutes seven core members of organizations namely; Google, Facebook, Verizon, Deutsche

Telekom, Microsoft, Yahoo and currently has reached more than 100 members with several versions of OF being released under ONF such as 1.1, 1.2, 1.3, 1.4 and 1.5.

The OpenFlow concept is no longer just a research model that can remain within the boundaries of academia but has been rapidly moved to the production environment.

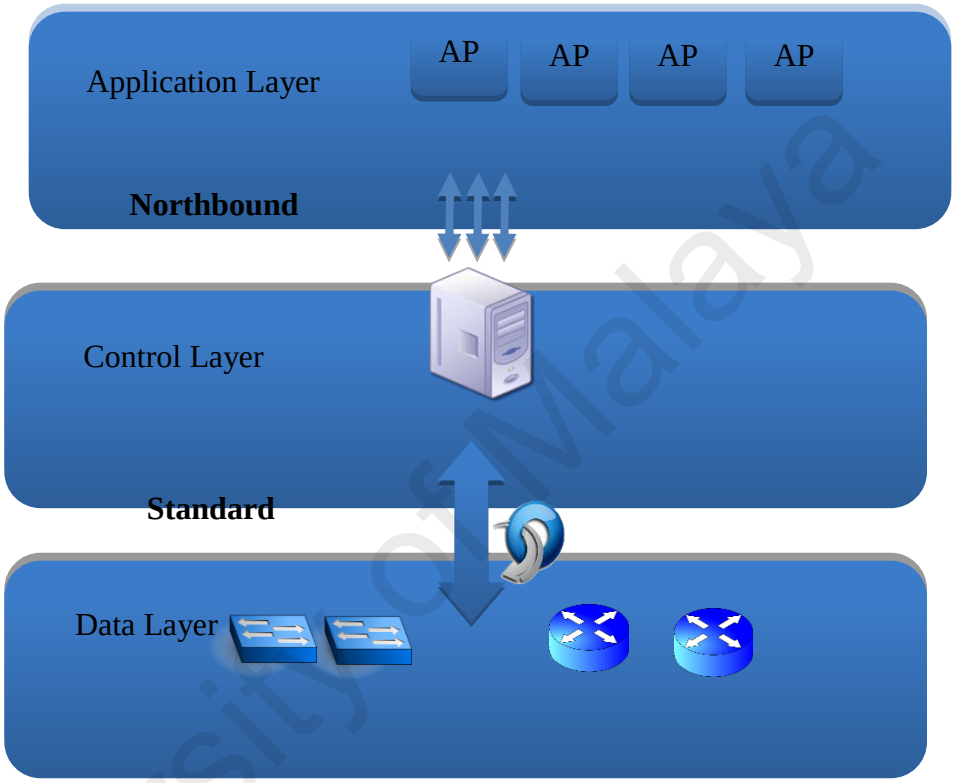

Figure 2.2 SDN architecture

In April 2012, Google announced that its backbone network has been fully operational in OpenFlow, with 10Gbps network link located in 12 data centers around the world. After the implementation of SDN, the utilization of the WAN lines has increased from 30% to near saturation. Later in April 2013, big companies such as Cisco and IBM, Microsoft, Big Switch, HP and Red Hat worked together to develop SDN applications and established OpenDayLight (Medved, Varga, Tkacik, & Gray, 2014) controller, which is an industrial-grade open source SDN controller.

2.2.1 SDN Architecture

The SDN architecture (Bozakov & Sander, 2013) consists of three main components: SDN controller, which is called the ***control layer***; SDN device (switch, routers), which refers to the ***data layer;*** and SDN ***application layer*** in which all network applications are executed . The main feature of the SDN architecture is that the controller and data layer are decoupled and abstracted from each other. In addition, programmability is a key feature that enables users to develop their own applications at the application layer using ***northbound interface*** that provides a programmable API and high-level policy applications and services. Moreover, the ***southboundinterface*** provides standard APIs that facilitate the communication between the controller and the switch via ***OpenFlow protocol***. In the next section, we discuss SDN architecture components in details.

2.2.1.1 Controller Layer

SDN controller is a network operating system (Clayman, Mamatas, & Galis, 2016) that views a comprehensive network topology and manages OpenFlow switch via a secure communication channel. It is responsible for managing, controlling, and manipulating flow tables (Kuźniar, Perešíni, & Kostić, 2015) in the switch. SDN controller communicates with two interfaces that include asouthbound and a northbound interface. The northbound interface provides programmable API that interacts with the application layer, while southbound interface communicates with the Data layer via a secured channel. A programmable API (Jarschel, Zinner, Hoßfeld, Tran-Gia, & Kellerer, 2014b) provides an abstract view of the network and delivers specific network functions in order to fulfill the network operator's needs. Server messages are interchanged between the controller and data layers via a southbound interface for establishing a connection and retrieving information. For example, SDN Controller manages the forwarding table for each switch based on the header of the PacketIn

message that is sent from the switch. The controller then replies to this message by sending “PacketOut” that informs the switch on how to deal with this packet based on the network policy. SDN supports two modes of deploying a controller, centralized mode whereas one controller can manage the entire network and distributed mode where two or more controllers control the whole network. Each controller, called the domain controller, is responsible for managing a number of switches and shares the network information with the other controller. Another mode of distributed controller (Schmid & Suomela, 2013) is the *Master/slave* mode whereas the *slave* controller serves as a back up to the *Master* controller in case of any failure. Two metrics are taken into account when measuring a controller’s performance; flow setup time and a number of flows per second that the controller can handle. These metricshave heavy influence when additional SDN controllers are deployed. To date, different types of SDN (compatible) controllers have been developed.

2.2.1.2 Data layer

Data layer consists of a set of networking equipment (such as switches, routers, and middlebox), known as OpenFlow switches, which communicate to formulate a single network. The OpenFlow switch is responsible for capturing, manipulating, and matching packets against flow table entries. The main function of SDN switch is to process the transit traffic based on the controller’s policy which decides what to do with packets headed to an ingress interface. It manages a number of flow tables, and each flow entry is associated with a set of instructions or actions that change a packet. When an incoming packet matches the rule in the flow entry, an action is required. The action might be forwarding a packet to a specified port or dropping the packet. OpenFlow involves two types of actions: required and optional (Shahmir Shourmasti, 2013). A required action must be supported in switches; whereas optional action is set based on the network requirements and could be a query by an OpenFlow controller. In addition,

OpenFlow switch supports multiple flow tables and a different group table that sometimes refers to an OpenFlow pipeline (El-Azzab, Bedhiaf, Lemieux, & Cherkaoui, 2011), in which a packet interacts with these flow tables. There are two types of SDN switches pure (OpenFlow-only) and hybrid (OpenFlow-enabled) (Azodolmolky, 2013). Pure OpenFlow switches have no legacy features or onboard control. These switches completely rely on the controller to forward decisions. Hybrid switches support OpenFlow as well as traditional operation and protocols. There are two approaches to manage flow tables in OpenFlow specification; Proactive Flow (P. Lin et al., 2013) in which the controller sets up flows in advance; Reactive Flow (Dusi, Bifulco, Gringoli, & Schneider, 2014), whereas controller responds to PacketIn events and dynamically updates the flow table.

2.2.1.3 Application Layer

Application layer consists of various network application services (Feamster, Rexford, & Zegura, 2014) that run on top of the SDN controller. It interacts with the controller through the northbound API interface. These application services can be used to configure the flows to be forwarded based on the changes in the network. For example, load balancing application distributes the traffic across multiple servers or paths according to the current load status. SDN applications communicate with SDN controller via APIs to manipulate network information. These APIs depend on the controller itself, whether the controller provides reach APIs that enable developers to design their applications or not. Usually, most of the open sources and commercial controllers provide REST-FUL-API (Zhou, Li, Luo, & Chou, 2014) that can easily be enabled to use any language. Recently, HP launches the network application store that includes various numbers of applications listed in its category.

2.2.1.4 Southbound Interfaces (SBI)

SBI (Ros & Ruiz, 2014)enables the SDN controller to manipulate the behavior of data plane and make changes according to real-time demands and needs. The main function of the SBI is to facilitate communication between a controller and a network switch (both physical and virtual) so that the switch can discover network topology, define network flows and implement requests relayed to it via Standard API. Several standards are available such as DevoFlow,OF-Config and Cisco's OpFlex. Cisco OpFlex is the most popular standardized southbound API for OpenFlow.

2.2.1.5 Northbound Interfaces (NBI)

NBI (Zhou et al., 2014) is a layer that sits between the SDN controller and high-level services and applications to enable the exchange of information between the controller and network applications. Each controller provides API interface to allow the user to interact with the lower level details of network functions. For example, controllers such as OpenDaylight, Floodlight, and Ryu define their own APIs that depend on the programming language deployed to develop the controller, but most of them provide REST-API. Therefore, Open Networking Foundation (ONF) created NBI working group that aims to develop standards for the interface that can be used by all the controllers. Recently, a number of the domain languages like Frenetic and Pyretic have been introduced to abstract the inner details of the controller and switch. In addition, the NBI is designed to be integrated with the cloud such as OpenStack and CloudStack.

2.3 SDN-SLB Server architecture Vs Traditional Load Balancing architecture

Over the past few years, the SDN architecture has emerged as a new network paradigm (Shukla, 2015) to provide management of the network services via abstraction. Such abstraction providesunified cloud resources that can be managed by

the SDN controller (Blial, Ben Mamoun, & Benaini, 2016). The separation of data plane from the control plane and management of network traffics via a controller that has an entire view of the network can provide optimal load balancing services (Godfrey & Stoica, 2005). Therefore, server load balancing is one of the challenges that can be addressed by the SDN (Handigol et al., 2010). In this section, we compare between the SDN architecture and traditional network architecture in terms of server load balancing and discuss the capabilities of the SDN in enhancing the load balancing services.Lastly, we highlighted the comparison between conventional SLB and SDN-SLB in terms of monitoring, scalability, configuration, innovation and management.

The SDN has the potential to simplify network management and enables innovation that brings about evolution to computer networks. It is a natural fit for load balancing since the controller has a global view of network resources and with knowledge of application requirements to optimize the load. Thus, the SDN brings new possibilities for improving load balancing techniques that faced several problems in the traditional network. Today's solutions in the cloud for the load balance are effective but have limited flexibility in terms of customization. Typically, in the cloud environment, service providers host various types of services and applications with multi-tenant services that require specific load balancing schemes. Consequently, customizing the load-balance for thousands of applications can be difficult and may require different load balancers for various services and in turn, can be too costly in terms of price.

Figure 2.3shows the SDN load balance architecture whereas the decoupling between SDN controller and OpenFlow enabled switch is presented. The controller has a complete view of the network components including switches, links, and server pools. The clustered controllers can be used for scalability (Wu, Huang, Kong, Tang, & Huang, 2015) to avoid a single point of failure. In the SDN network, when a new request is sent by the client, the switch checks the flow table; if there is any entrymatched, the switch carries out the action and forwards the flows to the corresponding server. If no match is found, the first packet of the flow is then sent as

Figure 2.3 SDN load balance architecture

PacketIn to the controller.

Sending packetsIn messages causes delays in responding to the user, but after the flow entry setup, traffic will flow normally between a server and the client. Thus, the SDN supports two modes of forwarding; *reactive mode*(Ding, Qi, Wang, & Chen, 2015) and *proactive mode* (Mayoral, Vilalta, Munoz, Casellas, & Martinez, 2015). Reactive mode is the default behavior of SDN whereas packetsautomatically are forwarded to the controller in case of the miss-match field presented in the flow table, while proactive mode setup the flow entry in advance.

Both the approaches have pros and cons in the load balancing system. Proactive mode minimizes the response time, but the forwarding is implemented without involving the controller that causes a bottleneck problem. The reactive mode allows the controller to make a decision based on the status of the current server. The status information of the links and servers are reported via statistical messages between the controller and the switch. Such messages can be configured to be sent periodically (W. Chen, Shang, Tian, & Li, 2015), for example, after every 5 seconds, the controller requests the network nodes for their status. The controller runs the load balancing module in the application layer on top of the controller to manage the server pool. The

controller executes this module that is responsible for handling the incoming packets based on the load policy.

Server Pool is a group of hosts that provides one or different services; each pool associated with VIP and specific type of the traffic, for example, TCP, UDP or ICMP. Two pools cannot share the same VIP, and each pool can be programmed with a specificalgorithm. This allows dynamic changes to the algorithm based on the traffic pattern or service type. In addition, it allows flexible, and dynamic scale-out/scale-in load balancing system by adding or removing a server from the pool. The main difference between SDN-SLB and traditional SLB is thatthe SDN provides programming *APIs* that facilitate more functions. Traditional load balancing is a pre-configured system carried out by the administrator, and it is limited to a certain load balancing policy.

Figure 2.4 illustrates the flow chart of the typically SDN-SLB PacketIn flow steps. Initially, the flow-table in the switch is empty and consists of single action called *Controller that* sends any packet to the controller. Before sending PacketIn to VIP, MAC address of VIP must be advertised to all servers and client. It can be done by using *Pingall* command that builds ARP table in the network. When the packet is sent to the VIP, the load balancing module is executed. The first step is to check the ARP in order to get the MAC addresses of the source and destination.

If the source MAC address belongs to the VIP, the ARP will reply. Otherwise, the controller will flood the packet. In the second step, the controller checks the IPv4 traffic whether it is TCP, UDP, or ICMP. After that, the controller selects the load balancing policy and the algorithm that handle this traffic. The current implementation of load balancing modules in most SDN controllers is designed to give one VIP for each server pool.

This means the load balancing module will select a server from a specific pool. This is one of the limitations of the current load balancing module. Another limitation is that each server pool is associated with one type of traffic such as TCP or UDP. In addition, each pool has only one algorithm that is configured before the load balancing module is executed. After the server is identified by the controller in respond to the user, a flow entry is then sent to the switch as a *packetOut*message and the rest of the packets will be transferred to the same server without involving the controller. This approach is called reactive mode whereas the controller is responsible for redirecting the PacketInbased on the application that runs on the controller.

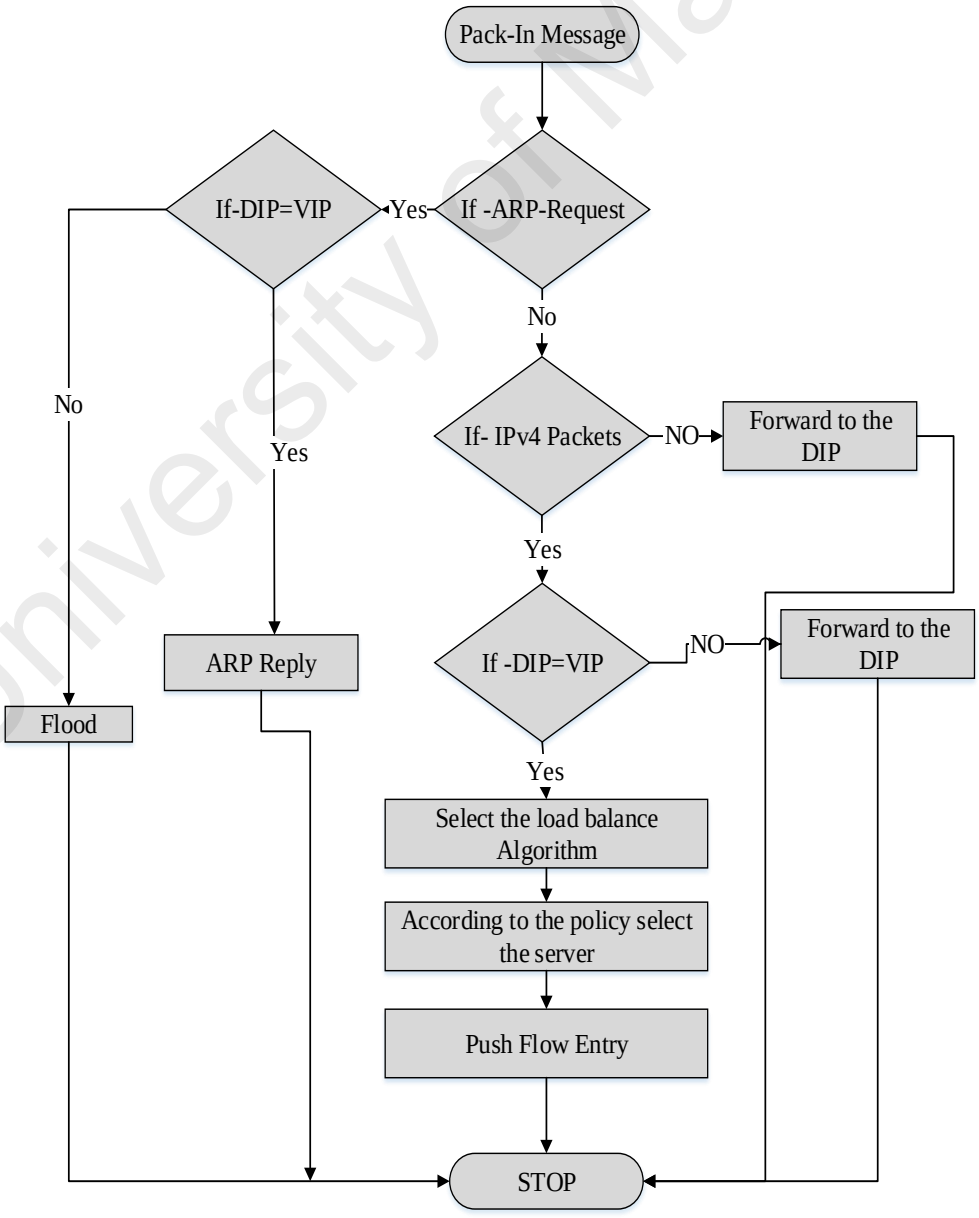

Figure 2.4The flow chart of the load balance PacketIn

In the second approach, known as proactive mode, the load balancer sends *PacketOut* in advance to the switch so to facilitate management of the incoming packet without involving the Controller. Figure 2.5 illustrates a conventional load balancing system (Okano, Ochi, Mochizuki, & Takaba, 2004) in a cloud environment whereas the load balancer located after the firewall maintains the VIP address. Users send their requests to the VIP without knowing about the IP addresses of the physical servers. The load balancing system which is either a software-oriented such as HAProxy (Tarreau, 2012) or hardware-oriented such as F5, represents a single point of failure. In addition, it is designed to serve a specific type of services that are pre-configured into the system. Thus, deploying new services requires additional load-balancers that may incur extra cost to the Service Provider (SP). We can summarize typical steps of traditional load balancing system as follows:

1. Load balancer receives the incoming requests from the clients.
2. Check the request's type (e.g. HTTP or FTP) and builds a request queue.
3. Checks the current load status of the servers in the server pool periodically using a server monitor tool.
4. Uses a load balancing strategy/algorithm/heuristic to select the appropriate server.

Table 2.2 showsa list of the comparisons between SDN load balance and traditional

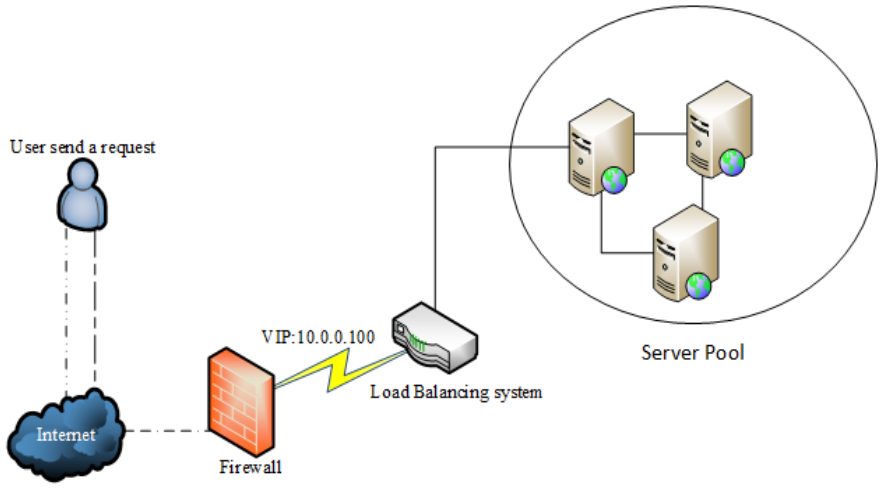

Figure 2.5 Traditional load balance architecture

load balance system in terms of monitoring, scalability, configuration, innovation and management.

Table 2.2 Comparison between conventional SLB and SDN-SLB

Issues	**Conventional Load Balancing System**	**SDN Load Balance**
Monitoring	Use extra tools that cause overhead in the system	Centralised monitoring with a dynamic global view
Scalability	Adds a load balancer for each new service	Auto-scales out and scale in
Configuration	Static configuration and error prone	Dynamic configuration with programmable API's
Innovation	Difficult to implement new load balancing algorithm	Innovates new load balancing schemes
Management	Difficult to manage for interconnection of many proprietary	unify cloud resources

2.4 Classification of SDN-SLB

In this section, we discuss a thematic taxonomy of the SDN-SLB. It is categorised based on four criteria namely; approaches/techniques, controllers, algorithms and experimental environments. The taxonomy is illustrated in Figure 2.6.

2.4.1 Approaches /Techniques

This section discusses various approaches and techniques that are used with the SDN network to provide server load balance.

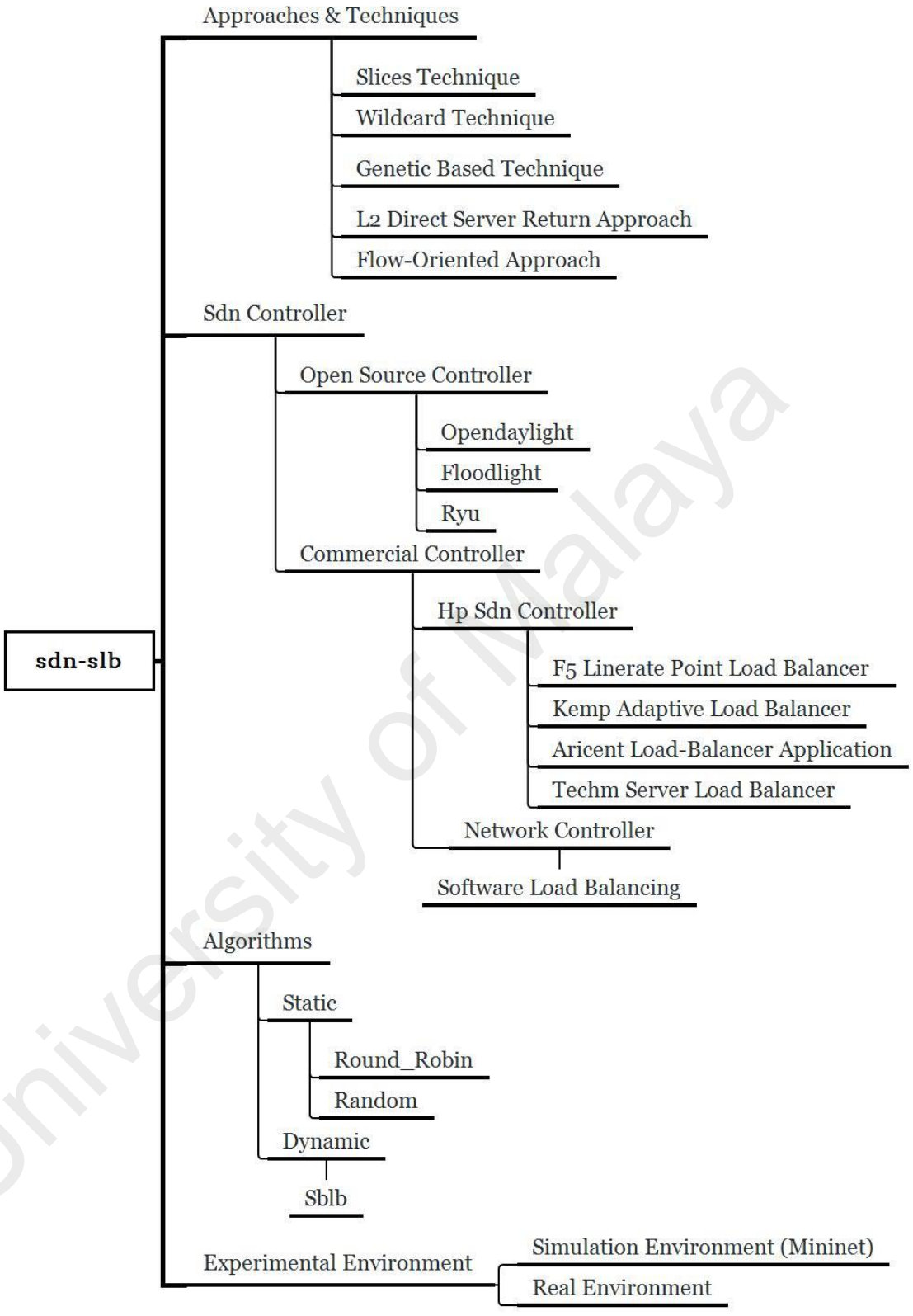

Figure 2.6 Taxonomy of SDN-SLB

2.4.1.1 Slices technique

Slices technique arose from the SDN network virtualization that introduced a virtual layer on top of the physical network infrastructure. This layer provides network virtualization by constructing several virtual networks. In turn, the virtual networks, contain virtual resources such as nodes, switches, and routers which also need to be controlled and managed. The control is implemented through a transparent proxy that acts as links between switches on one side and multiple controllers on the other side,(Kashiri, Tsagarakis, Van Damme, Vanderborght, & Caldwell, 2016). FlowVisor (Sherwood et al., 2009) and OVX (Al-Shabibi et al., 2014) are the examples of proxy controllers which are used to create a slice for each virtual network. Multiple service load balancing architecture that uses the slicing technique is proposed by (Marc Koerner & Kao, 2012). The idea aims at using multiple controllers where each controller is utilized per network service and is connected to the FlowVisor (FV) controller for the provision of load balancing strategies. This technique uses a slicing mechanism to manage different controllers in various parts of the network. In the FV slices, the mechanism depends on the header field information of the packet and will forward the packets based on respective policies. For example, a new request with destination port 80 is sent to the HTTP load balancing controller while a request with destination port 21 is sent to the FTP load balancing controller. Initially, the request arrives at FV which is responsible for managing all the network slices and for defining all the related services. Based on the packet header information (Costa & Costa, 2015), FV inserts flow entry into the switch, and the incoming packet is sent to the corresponding controller. This process defines the forwarding rule, and the path is then identified based on the load balancing algorithm. The round-robin algorithm is used as load balancing scheme that runs on the NOX controller. The module was written in C++ (Stroustrup, 1986), and the experimental evaluation was carried out in the "TU Berlin testbed" (Kwak & Jung,

2015). The drawback of this technique is that the proxy controller represents a single point of failure where it manages the entire traffic between multi-controllers and switches.

2.4.1.2 Wildcard Technique

Another technique used to provide load balancing is based on "wildcard rules", that directs incoming requests based on the clients' IP addresses. In the paper (R. Wang, Butnariu, & Rexford, 2011), the authors statedthatinsertingseparaterules for each packet flow leads to a huge number of rules in the flow table. This approach causes a heavy load on the controller since the controller needs to manipulate every *PacketIn* message. For overcoming, this problem, a new load balancing algorithm was proposed to calculate concise wildcard rules which are automatically adjusted for different load-balancing policies. The approach uses a proactive mode that inserts flow entry without involving the controller. The *partition algorithm* is proposed todetermine a minimal set of wildcard rules, and *transition algorithm* is used to change the rules for adapting new load balancing weights of the servers. For example, suppose the client traffic matching 0* indicates the rule that traffic should shift from server 1 to server 2. The controller needs to examine the next packet of each connection to decide whether to direct the traffic to the new server, server 2 or the previous server (server 1). In this case, the controller installs a rule directing all 0* traffic to the controller for further inspection; upon receiving a packet, the controller installs a high-priority micro-flow rule for the remaining packets of that connection. Mininet simulation is used with NOX controller and Open VSwitch OVS (Pfaff et al., 2015) to prototype the network, and Mongoose (Williams, 2015) is used as an emulator for the web server (Williams, 2015).

2.4.1.3 Genetic based technique

SDN load balance policy-based architecture with aGenetictechnique for distributing large data from clients to different servers was proposed by (Chou, Yang, Hong, Hu, &

Jean, 2014). The technique redirects the traffic flows to achieve optimal load-balance. This study assumed that there were a number of***N***flows, and each flow has a different load. In turn, each server in the server's pool had a different workload. In order tominimize the server's coefficient, the fitness function was proposed. The architecture consists of two main components: OpenFlow switch and SDN controller. On top of the Controller, three modules are built: *flow control module*, *a decisionmodule*, and *monitor module*. OpenFlow switch component includes *flowing modification module*, *OpenFlow handler module*, and *Packet-Mirror module*. Four load balancing algorithms were compared namely: Genetic-based, Load-based, Random and Round-robin. The Genetic-based showed significant performance compared to the other algorithms. The experiment was carried out in a simulation environment (Keti & Askar, 2015) with a simple topology.

2.4.1.4 L2 Direct Server Return

Typically, there are two common modes of operations for load balancing schemes which are NAT-based and Proxy-based. The mechanism of NAT works in layer-4 (MacDonald & Lowekamp, 2010) and it is based on rewriting the destination values of the IPs or MACs of the packets. The NAT and Proxy-based load balancers are responsible for observing and managing both incoming and outgoing network traffics.The authors (Michael Koerner & Kao, 2013) state that the performance of NAT in the conventionalnetwork is not suitable to provide an optimal load-balancing solution. For that, they proposed Layer 2 Direct Server Return model (L2DSR). The model implements a layer 2 concept (Jain & Paul, 2013) to improve the performance of OpenFlow switch that forwards the packet and replaces the source MAC address with its MAC address. The system architecture consists of Load Balancer Controller (LBC) that separates the servers from network to avoid addressing conflicts, and thus there is no need to use virtual IP (VIP). The LBC does not directly send any L2 traffic, and the

controller forces the switch to act as a device with an own non-transparent interface. In order to avoid layer three address conflicts, the broadcast of all MAC addresses is not flooded in the server network. In the case of no information available, the switch will forward the packet to the user according to the balancing algorithm and replace the source MAC address with the OpenFlow Switch MAC address. On the other hand, when the user replies, the switch then sends traffic with its IP and replaces the destination MAC address with the address of the server. The experiment was implemented in OFELIA (Suñé et al., 2014)testbed that consisted of three switches supporting OpenFlow version 1.0 (Consortium, 2009) and three servers. One of them is an NOX controller with a plug-in model written in C++. The solution was carried out with a round-robin algorithm.

2.4.1.5 Flow-oriented approach

The centralization of the SDN controller enables the traffic flow to be re-directed dynamically when any changes occur in the network. The flow-oriented load balancing approach using the SDN technology is proposed by (Bays & Marcon, 2011). The approach is built based on a number ofpolicies that dictate the direction of all data flow to intended servers. Data flow-oriented approach assumes that the communication between clients and servers is established when a client sends a request to the server, and the flow remains active even after a certain period of inactivity. These flows are distributed among existing servers via switches that redirect the packets according to a load balancing policy. Three policies are adopted in this study namely; *Random policy, Time slice policy,* and *weighted policy*. Random is a simplepolicy that randomly allocates the flows to servers and does not consider the capacity and current load of servers. This policy is not effective in heterogeneous distributed server farm (González-Vélez & Cole, 2010) that includes servers with different capacities. In the *time-slice policy*, each certain time controller will select a server for responding to the

request, and this selection is implemented randomly bearing in mind that each server has one slice-time. Thus, the only information required to be stored in this policy is "*a server with specificslice-timeperiod*", and this policy is not a concern for the current load of the servers. The Weighted balancing policyrecords the numbers of flows processed in each server. For the new incoming flows, the controller selects the server that has minimum load level; that is, a server with thesmallest number of connected or communicating clients. The controller uses the counter field (E.-D. Kim, Lee, Choi, Shin, & Kim, 2014) to calculate the number of flows that can be handled by each server. Table 2.3 shows the approaches and techniques with the related SDN layer and the flow modes that have been used.

Table 2.3 Approaches and techniques of SDN-SLB

Approaches/ Techniques	**SDN Layer**	**Managing flow table Mode**
Slices technique	Controller	Reactive mode
Wildcard technique	Data layer	Proactive mode
Genetic based technique	Application layer	Reactive mode
L2 Direct Server Return	Data layer	Reactive and Proactive modes
Flow-oriented approach	Application layer	Reactive mode

2.4.2 SDN Controller

The SDN controller is called the "network brain" (Jarschel, Zinner, Hoßfeld, Tran-Gia, & Kellerer, 2014a) which manages a collection of network application modules to perform different network tasks. Typically, these applications communicate with the core controller modules via an API (Zhou et al., 2014) to enhance more advanced capabilities. In this section, we discuss the server load balance modules that are widely implemented in various Open Source and Commercial SDN controllers. In the open source controller, the SLB moduleis carried out in application layer as part of the controller system, while in the commercial controller, it is a standalone application intergraded with the controller to utilize the APIs.

2.4.2.1 Open Source Controllers

Since the beginning of the SDN in Stanford University, several open source controllers have been introduced such as Trema (Khattak, Awais, & Iqbal, 2014), ONOS (Berde et al., 2014) and the Pox (Mccauley, 2014).These controllers come with network application modules that provide specific services like load balancing. This section focuses on well-known controllers that provide the server load balance service as a network application module.

OpenDaylight controller: OpenDaylight (Baucke, Mestery, Shaikh, & Wright, 2013) is an open source project which has focused on accelerating adoption of the SDN by providing a robust platform on which the industry can build and innovate. It can provide load balancing service in the cloud by integrating an SLB module into a cloud operating system such as OpenStack. OpenDaylight exposes the OpenStack (Sefraoui, Aissaoui, & Eleuldj, 2012) Neutron APIs service to manage the load balance services. Different bundles constitute the Neutron APIs (Denton, 2014) such as Northbound API and Neutron Southbound provider interface (SPI). The Neutron southbound interface provides collections of several classes that include load balancer, load balancer health, load balancer listener, and the load balancer pool. A Neutron northbound interface is used to create a VIP which will map a pool of servers within a subnet. The pools consist of members that are identified by an IP address. OpenDaylight comes with two types of load balancing services; 1) load balancing service that deals with balancing traffic to back-end servers based on the source address and source port for each incoming packet. This service is implemented in Hydrogen(Gomez, 2013), the first version of OpenDaylight with the basic round-robin and random algorithms, 2) used OVSDB protocol to create L3-L4 state-less load-balancer (Brandt, Khondoker, Marx, & Bayarou, 2014) in OVS and can be used in the virtual environment. Both types of load

balance systems can be used with a session-preserving and use active and proactive modes.

Flood Light Controller is a high-performance open source OpenFlow controller that is written in Java. It was developed on the basis of Beacon controller, an experimental OpenFlow controller from Stanford University, and it is now supported by a large developer community. Currently, Floodlight supports OpenFlow version 1.4 (Specification, 2013) and works with various physical and virtual OpenFlow-enabled

```
curl -X POST  d '{"id":"1","name":"vip1","protocol":"TCP","address":"10.0.0.100","port":"8"}' http:// 192.168.1.2:8080/quantum/v1.0/vips/

curl -X POST -d '{"id":"1","name":"pool1","protocol":"TCP","vip_id":"1"}' http://192.168.1.2:8080/quantum/v1.0/pools/

curl -X POST -d '{"id":"1","address":"10.0.0.1","port":"8","pool_id":"1"}' http:// 192.168.1.2:8080/quantum/v1.0/members/

curl -X POST -d '{"id":"1","address":"10.0.0.2","port":"8","pool_id":"1"}' http:// 192.168.1.2:8080/quantum/v1.0/members/

curl -X POST -d '{"id":"1","address":"10.0.0.3","port":"8","pool_id":"1"}' http:// 192.168.1.2:8080/quantum/v1.0/members/

curl -X POST -d '{"id":"1","address":"10.0.0.4","port":"8","pool_id":"1"}' http:// 192.168.1.2:8080/quantum/v1.0/members/
```

Figure 2.7 CURL configuration of Floodlight load balance

devices. Floodlight comes with built-in load balancer application for the ping, TCP, and UDP flows. Typically, these applications are designed for the developer to understand how to develop Floodlight module, and for the usage of APIs. The load balance module can be configured via a REST API (Zhou et al., 2014) that supports the basic creation of the VIP and Pools and adds members to that pool. However, this module has two limitations. It uses Static Flow Pusher module (Ivancic, Lumezanu, Balakrishnan, Dennis, & Gupta, 2014) that sets the flow timeout as 0. This means that the flow entry is not purged after use, and in turn, the flow-table will be overflowed over a certain time. The second limitation is that the module uses static simple load balancing policies such as Round Robin and Random schema.

In addition, health monitoring feature is not yet implemented with the module. Figure 2.7 shows the simple script that can be used to configure a load balancing module through CURL tool.CURL is a tool uses to transfer data to SDN controller via HTTP request. This request includes variables and values, and the IP addredd of the

controller must be included in each request . In figure 2.7,HTTP curl- POST used to addVIP name *vip1* and associated with specific IP address (10.0.0.10) and port (8). The pool is created , named (pool1) and this pool can only accept TCP traffic. The rest of the CURL codes are added to the host to that pool in which one service is used.

RYU Controller: the Ryu (Khondoker, Zaalouk, Marx, & Bayarou, 2014) controller has an advantage over other controllers due to its compatibility in supporting the higher versions of OpenFlow (v1.5) (Specifiation, 2014). It also supports the OpenState switch (Bianchi, Bonola, Capone, & Cascone, 2014) that provides State-Full forwarding in the data plane using Finite State Machines (FSM) (H. Kim et al., 2015). Such state machines are implemented in switches to reduce the need for relying on the remote controller and execute the logic of forwarding consistency. Server Load balancing based on FSM was introduced with the Ryu controller. In this module, the selection of the server is performed via a new group entry type that selects randomly one of the action buckets for a group entry.

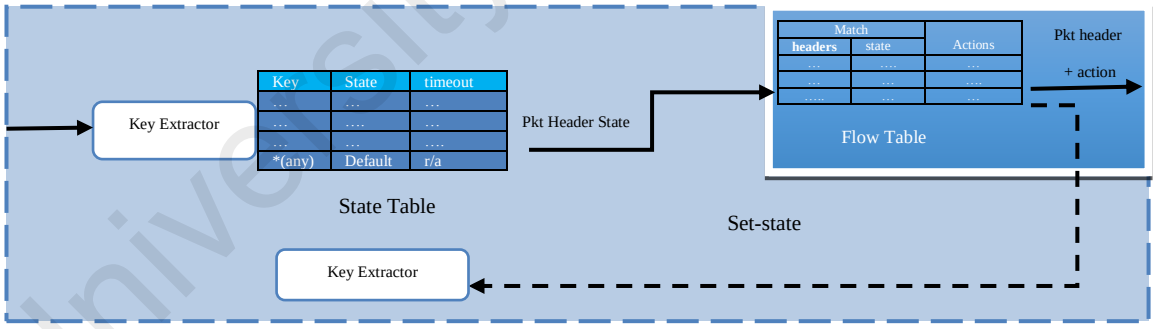

Figure 2.8 The OpenState process using FSM

Figure 2.8 presents the new state table that includes a key which is associated with a specificstate. For example in the load balancer, the incoming packet is first processed by a key extractor that uses the header fields. If an entry is found, then the corresponding state label is returned. Otherwise, it is set to 0 (default) state that is associatedwith the packet. By implementing this mechanism, the response time of the server is minimized.

2.4.2.2 Commercial Controller

SDN rapidly moved from research area to industry where big companies such as Microsoft, Cisco, and HP adopted this technology to provide solutions for the cloud. In effect, and according to Infonetics Research (I. Research), the SDN market will grow 129% and is expected to reach $18 billion by 2018. SLB is one of these solutions that areavailable in the market to address traditional load balancing issues. This section discusses several of SLB systems that are integrated with the commercial controller to provide load balancing services in the cloud.

F5 LineRate Point Load Balancer:F5 is a leading company for load balancing solutions that introduced *LineRate Point Load Balancer*(L. P. L. Balancer). It is a Layer 7 load balancer that can be used in a virtual environment for the Software-Defined Data Center (SDDC) environments (Kerravala, 2013). This solution aims to reduce the costs, on-demand application deployment, and automation. Moreover, the system is easy to configure through Graphical User Interface (GUI), Command Line Interface (CLI) or by utilizing the APIs. The main idea for this solution is that each application or services are paired with their own load balancing instances. This means that the system can be scaled accordingly when a new service or application is being deployed. Thus, it is based on the fully automated operation, and it supports multiple loads balancing algorithms. *LineRate Point Load Balancer* comeswith a number offeatures such as supporting HTTP/HTTPS, layer 4 TCP protocol (but it does not support UDP), application health monitoring and SSL offload (Jethanandani, Bashyam, Bagepalli, & Patra, 2006). In addition, the solution provides persistence when the users' request returns to the same server. Currently, it can be integrated with HP SDN VAN controller (Tourrilhes, Sharma, Banerjee, & Pettit, 2014) to monitor all aspects of the server's health and performs visibility on thousands of metrics via the REST API or Simple

Network Management Protocol (SNMP) (Frey, Bicket, Herbert, Malhotra, & Chambers, 2016).

KEMP Adaptive Load Balancer: the KEMP SDN Adaptive load balancer (K. S. l. balancer) is a dynamic load balance application with a delivery value that utilizes the HP VAN SDN Controller capabilities. KEMP can serve as a driver for Neutron LBaaS with the capability to manage features that are not supported by the OpenStack. On the other hand, KEMP provides a layer 4–7 load balance which is also integrated with the HP Virtual Application Network (VAN) SDN Controller to solve the problem of end-to-end visibility of network paths that was present in traditional networks. This load balancer has visibility of the upper layer application-level information (request load time, SSL TPS, application response throughout, just to name a few). Therefore, KEMP pulls information across the Northbound Interface (Shin, Nam, & Kim, 2012)(NBI), extending its visibility by adding the circuit information received from the controller. KEMP-HP combined with SDN controller can improve the performance of a new application across existing infrastructures and can be implemented in Hyper-V, VMWare.

AricentLoad-Balancer Application: Aricent's SDN Load Balancing application (Load-Balancer) is a dynamic and scaled SDN load-balancing solution which supports multiple OpenFlow switches. It comes with various features such as dynamic server configuration, wildcard forwarding, supporting various protocols and multiple flows from a client. In addition, it has northbound REST API that allows managing the distribution of traffic amongst servers using Weighted Round Robin (Devi & Uthariaraj, 2016), Weighted Least Connection (Yang & Yu, 2003), or custom algorithms. Thus,users can customize their own load algorithms dynamically and then integrate them accordingly into the SDN controller application. On the other hand, Aricent Load-Balancer enables the addition or removal of servers dynamically through the Web-based

graphical user interface (Y. Li & Brodlie, 2003) and manages the monitoring of available bandwidth in real-time.

Software Load Balancing: the Software Load Balancing (SLB) is an SDN load balancing system that was introduced by Microsoft in Windows Server 2016 (2016). It allows the distribution of tenant and tenant customer network traffic among virtual network resources with high availability and scalability. SLB provides links between virtual IP addresses (VIPs) that represent specific server pool in the cloud, and dynamic IP addresses (DIP), the IP addresses of the VMs (Maltz, Greenberg, Patel, Sengupta, &Lahiri, 2012) of the pool behind the VIP. The system stores all VIPs in the Multiplexer (MUX) that is managed by the Network SDN Controller and performs mapping among VM IPs and VIPs. When incoming traffic arrives, the MUX then checks the traffic, in which it includes the VIP as a destination, and maps and rewrites the traffic so that it will arrive at a particular DIP. One of the great features of SLB is implementing Direct Server Return that minimizes the server response time and provides low-latency. For example, when tenant VMs respond and send network traffic back, NAT is performed by the Hyper-V (Velte & Velte, 2009) host and the traffic then bypasses the MUX and goes directly to the edge router from the Hyper-V host. The system includes various features such as Health probe, full support of virtualization with high availability and scalability.

TechM Server Load Balancer: the TechM's is agent based SDN load balancer that can be deployed either on the servers or switches without the use of additional applications or agents. It is built on the rich set of RESTful APIs exposed on the various SDN controllers including HP VAN SDN Controller, ODL, and ONOS. It leverages end-to-end network visibility and network delay parameters for routing application traffic efficiently and dynamically. TechM provides load balancing using any L3 to L7 and supports several algorithms such as Round Robin, Least Connections, and Weighted

Round Robin. Table 2.4 shows commercial SDN-SLB solutions from different vendors. The table illustrates each solution and their features as well as the related implementation that the SDN controller can be integrated with.

Table 2.4 Commercial SDN-SLB solutions

Solutions	**Implementation**	**Vendor**	**Features**
LineRate Point Load Balancer	integrated with HP SDN	F5	Automated operation and supports multiple loads balancing algorithmsREST-based API, CLI and GUI interfaces
KEMP Adaptive Load Balancer	With HP VAN	KEMP	End-to-end visibility of network pathsLayer 4–7 load balancer
A recent Load-Balancer Application	OpenDayLight Floodlight HP VAN	Aricent	Add or remove servers from the load balancer on-the-flyBuilt upon custom Algorithm
Software load balancing	With Windows server 2016	Windows	Direct Server Return (DSR)Support VMs and multi-tenants
TechM Server LoadBalancer	With HP VAN ONOS,ODL	TechM	Rich set of RESTful APIs exposed SDN controllerPrior configurations and application signatures to determine balanced load paths and the servers

2.4.3 LB Algorithms

In order to select the best server to handle incoming request, various scheduling algorithmsare implemented in SDN load balance system. Typically, these algorithms can be divided into two types namely; *Static*(Leland & Hendrickson, 1994) and *Dynamic*(Shabtay, 2010) algorithms. For software and hardware load balancing system, these algorithms are pre-configured and assigned by the administrator. The configurations include selecting specific types of the traffic pattern and assigning one algorithm for them. For example, the administrator can configure the load balancer to handle all HTTP, FTP, and ICMP with Least Connections algorithm. Therefore, to change the algorithm, the administrator needs to reconfigure it or utilize the load balancer API manually. On the other hand, most load balancers are designed with a limited number of algorithms (Berde et al., 2014) such as *Random, Round Robin, and*

Weighted least connections(Khattak et al., 2014). Thus, current data centers are moving towards software load balancing systems that can provide flexible load balancing in terms of diversity of the algorithms. In this section, we discussed the SLB algorithms that were applied in the SDN environment; we focused on three static algorithms: *Round Robin, Random,* and one dynamic algorithm: *Server Based Load Balancing* (SBLB) that have been implemented in most SDN-LB related papers.

2.4.3.1 Round Robin

Round Robin Algorithm (RRA) (Singh, Goyal, & Batra, 2010) is a popular static algorithm that is implemented in SDN load balancing systems. It divides the incoming traffics between servers in a round robin manner. In fact, it is suitable for homogeneous environments (Pu Wang, Sahinoglu, Pun, Li, & Himed, 2011) and it produces good results. In the SDN, the clustered servers (Berde et al., 2014)are organized into pools, and each pool is designed with a specific algorithm. The request from a client is sent to the VIP that can handle all incoming traffics. In the case of not matching field in the flow table, the first packet is sent as PacketInto the controller to apply load balancing policy. First, the controller will check whether it is IPv4 or IPv6 traffic. If the traffic is IPv4, then again the controller will check on whether the type of the traffic is TCP or UDP. After that, the controller selects the server pool to handle the request. At this point, a load balancing algorithm will be selected. Thus in the SDN, each server pool can be run with a specific algorithm. If the number of the servers in the pool is little, the RRA will not be effective. For example, if we have only 3 or 4 servers in the pool, then there is a chance that more requests will end up on the same server while the current request is still being processed. This led to creating a queue of requests causing a delay in response time. For overcoming this problem, the work in (Drutskoy, Keller, & Rexford, 2013) proposed the scalability function in a virtualization environment. The system is designed to add a VM to a pool in case of all VMs are busy with other

requests while there is a new incoming request. The system can automatically add a new VM to handle this request based on a threshold value that determines the maximum load of each pool.

2.4.3.2 Random

This algorithm selects the servers in a random manner (Chang & Tang, 2010b). The process involves utilizing an underlying random number generator to choose the server in the pool for responding to the incoming requests. Unlike the RRA that distributes the request in order, the Random algorithm selects a server randomly. Therefore, the number of servers in one pool affects the performance of the algorithm. On the other hand, the capacity of the servers also affects the performance of this algorithm (Mccauley, 2014). For example in the heterogeneous server pool, the algorithm selects a server without considering its capacity; this leads to ineffective load distribution. Each time when a flow is established, the controller will choose a server in a random way. This is the simplest policy, where it is not necessary to store any data; however, the current load of each server is not considered.

2.4.3.3 Server-Based Load Balancing Algorithm (SBLB)

This is a dynamic algorithm based on the feedback of the current server status and adopted by various SDN solutions (Sefraoui et al., 2012)(Gomez, 2013). However, thesesolutions are different in terms of feedback implementation and specifically on how the controller can obtain the feedback from each server. The feedback includes the server's current load that is calculated based on three parameters: CPU occupancy rate, Memory occupancy rate, and the Bandwidth. These parameters are reported periodically to the controller, and then each server will gain a weighted measure that represents the current load. When a new request is sent, the controller then checks the weight of all servers in the pool and sends a request to the server that has themaximum value of

weight to handle the request. The following equation shows the calculation of the server load

$$L(S_i) = w_i L(S_{cpu}) + w_i L(S_{mem}) + \sum w_i = 1_{and}), \quad (2.1)$$

However, some papers consider other parameters, for example, the work in (Sefraoui et al., 2012) introduced three variables to get the current server load which includes; CPU, Mem and Response time. We noted that the way and the time in which the parameters are collected are different from solution to another. For example in the paper by (Baucke et al., 2013), the authors used SNMP protocol to collect the information every 5 seconds while another paper (Specification, 2013) used LLDP messages that are originally used for topology discovery.Actually, LLDPdoes not show acomplete status of the servers because it is designed to update the network topology, and used here to get some metrics information of the devices in the network. On the other hand, gathering the server information on the runtime causes overhead to the controller, especially with all the related information being reported to the controller in short time. Thus, implementing an optimal monitoring system for load balancing in SDN must be considered.

2.4.4 Experimental environment

The experimental environment is a vital part of the balanced load system. This section discusses the two main experimental environments that have been implemented in SD-SLB solutions. Frist, asimulationenvironment that is mostly carried out in the Mininet simulation. Although, the Mininet allows easy to customize the network topology, but it is limited to the server and link resources. The second experimental environment is a real environment that is implementedin a cloud environment such as OpenStack or that utilized a real Testbed.

2.4.4.1 Mininet

Mininet (Mininet) is an SDN simulation aimed at developing and testing of SDN solutions. It is allowed to create and manage prototype of large networks on a single computer. More than 100 researchers in more than 18 institutions use Mininet to develop and test SDN applications (Lantz, Heller, & McKeown, 2010). Mininet can create SDN elements such as Host, Switch, Links, and Controller. Such elements can be fully customizedin terms of resources and then shared with real network devices. In addition,SDN controller can run on aremote computer and easily can connect to Mininet with customized topology. Regarding, load balancing, Mininet provides three ways to create network topology; 1) via *CLI* that allows the customization of the topology and resources such as link bandwidth, speed, and delay, including the number of CPUs for various Hosts. 2) *Writing a script,* that utilizesPython API in which users can customize their topologies. For example, asimple script can simulate clustered servers distributed in the data center network contacted to SDN controller with numbers of "OF" enabled switches. 3) Via *Graphical User Interface* (UI), Mininet provides GUIs to create various network topologies connected to SDN controller with the option to use an OF-enable switch, legacy switch, or a router. Although, Mininet has been used in most studies of SDN server load balancing, nevertheless, Mininet has a number of defects(Ortiz, Londoño, & Novillo, 2016) such as a long time to setup and launch its program especially when the network size is large

2.4.4.2 Real environment

Real experimental environment means the the test that carried out on the physical servers, VMs server in OpenStack or using Remote Testbed that uses slice techniques for separating user traffic(Khondoker et al., 2014). Based on our review, a few papers have used real environment, and 90% of them implemented their simulation

environment such as Mininet. However, Mininet provides the ability to customize the server's resources (Bianchi et al., 2014) such as the amount of (CPUs), cores as well as links' bandwidth, delay and queue size, but is still limited to the computer resources that hosts the Mininet simulations. Such resources affect the load balancing measurement metrics such as response time (Specifiation, 2014). For example, in dynamic load balancing that depends on the server's load status which is indicated by CPU, RAM and bandwidth may give inaccurate results. On the other hand, using real experimental environment with a few number of servers (for example, two or three) is not reflective of the reliability of the system. Another issue of real environment is the lack of support for the latest version of "OF" specification. Therefore, we noted that in a real environment that implemented in (Poddar et al., 2015)using IBM cloud and in (Parveen Patel et al., 2013) that used Microsoft Azure cloud. The experiment, are used OpenFlow switch that supports 1.0. In addition, most testbeds currently support theold version of OpenFlow while simulation environments are usually updated with last OF specification (H. Kim et al., 2015). For example, the last version of Mininet 2.2.1 supports Open vSwitch 2.5 that can currently work with OpenFlow 1.5.

2.5 SDN-SLB: State-of-the-art

Several SDN solutions have been introduced to address the issues of load balancing. These solutions focused on separation of the data plane from the control plane and centralized SDN controller to provide dynamic load balancing. Figure 2.9 illustrates state-of-the-art topics that are discussed in this section. At the end of this section, the comparison of these solutions is presented in Table 2.5 based on the taxonomy discussed in section 2.4. In our discussion, we focus on the proposed solutions that used SDN with other technologies to provide server load balancing. The comparison parameters include; acontroller, algorithms, experimental environment, and Open Flow switch. The controller parameter indicates which controller and language are used for

implementing the load balancing system. The parameter “algorithms” refers to the proposed algorithms that are introduced with such solutions. Experimental environment parameter shows the simulation, generated traffic tools, topology, and measurement metrics. For OpenFlow switch, we focus on the type of the switch (physical/virtual) and OF specifications that have been used.

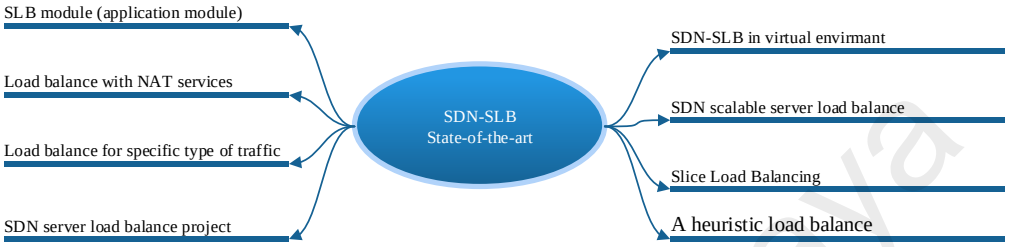

Figure 2.9State-of-the-art topics

2.5.1 SDN- SLB module (application module)

The first load balancing application based on OpenFlow switch was introducedby Uppal et.al (Uppal & Brandon, 2010). In this application module, each server has a static IP address, and a web server emulator is configured with the specific port. The server pools are connected to the OF switches that are managed by the NOX controller. When a new request arrives, OpenFlow switch will check the header of the packet and match it with the entries in the flow table. If the header of the packetmatch, the counter will then increment the number of the bytes and the necessary action is performed accordingly. If there is no match, the switch will send *PacketIn* message to the controller and then wait for *PacketOut* message that is sent from the controller to instruct the switch how to handle the flow. On top of the NOX controller, load balancing module is executed to determine how the switch can handle the flow by inserting new flow rule via OpenFlow protocol. Typically, clients send their requests to the VIP without knowing the physical IP address of the servers. The load balancing module is designed to modify the destination IP and MAC addresses and inserts the real MAC and IP addresses of the server that will handle the respective request. When a server responds to the client,

again the module will modify the packet header (VIP and MAC address) to show that the response is sent from the VIP. Servers report their current load to the NOX controller periodically. NOX canlisten to a separate thread on a UDP socket. When a new request is sent, the controller will check the servers' statuses and assign the request to the server with the lowest load and then increments that server's load to prevent a flooding of the flows going to the same server. In this work, a load balance module was written in C++ language. The solution has been carried out in a real environment with *HP 5412zl* OF switch that supports 1.0 specification. The *Latency* and *Throughput* of the algorithm show outperformance compared with other solutions.

Another load balance application module is a dynamic aware-load-balancing system that usedFloodlight controller (Shang, Chen, Ma, & Wu, 2013). In this work, the load-based scheme is introduced and compared with other two schemes such as round-robin and random. The controller selects the server with a minimum load according to the weighted least connections where the nodes are selected based on the number of active connections. To getting the load of the server, the number of active connections is divided by the weight of the server. The experimental test was implemented in the Mininet simulation.

2.5.2 Load balancing with NAT services

A dynamic load balancing technique with NAT function was proposed by (W. Chen, Li, Ma, & Shang, 2014). The authors argued that the load balancing systems in traditional network fail to achieve optimal load balance. For example, the DNS load balancing system is unable to know about the capacity of the servers and cannot reflect the current status of the server. In addition, load balancers that use NAT technology to map a public IP address for multi private IP addresses are relatively close to the bottom of the network infrastructure and expensive at the same time. In this solution. Floodlight controller was used with a virtual IP address and connected to the servers with a private

IP address. All servers share this virtual address, and the users only know the virtual IP address. All users' requests are sent to the virtual address and get a reply from the same virtual IP address. Thus, the controller can provide NAT services without any additional hardware devices. (W. Chen et al., 2014)proposed Server Based Load Balancer (SBLB) algorithm and compared it with random and round-robin algorithms. In SBLB, the dynamic feedback was developed to report the current server's load based on the Processor occupancy rate, memory occupancy rate, and response time. The author state that the ability of SDN to provide a global view of the network helps to deliver the content to the clients with high availability and performance.

2.5.3 Load balancing for specific type of traffic

Selin (Tourrilhes et al., 2014) proposed a server load-balance framework to enhance the QoS for video streaming services. The framework used OpenFlow protocol for providing dynamic server load balancing that monitors the loads at the video streaming servers continuously and redirects the client to the corresponding server with less load. This solution aims to utilize the server resources by developing controller application with two functions; *monitoring function* and *flow update function*. In the first function, the controller predefines threshold of load and keep track on whether the server load has exceeded this threshold or not. The monitoring function calculates the bandwidth usage between the servers and switches by dividing byte counts with a time interval. When bandwidth usage exceeds the threshold, then the author considered this scenario case as one where congestion has occurred. To avoid wrong decisions, the controller records non-congestion events after congestion detection. After the controller determines the threshold of a server, the controller then selects a new streaming server considering the two factors; *packet loss* of the link between server and client and *delay* that is measured based on the geographical location of the server. In the second function, an*update function,* after the previous flow has been detectedthen, the controller updates the flow

table by adding a new flow entry that redirects incoming packets to the new server. Additional PC was used which is called the "traffic loader."to simulate a client's request, and one node is used to monitor the severs's load, and sent it to the controller.

2.5.4 SDN server load balance first project

Aster*X(Kerravala, 2013) is the first project that is targeted to provide load balancing using SDN architecture. The project aimed at providing network load balance that can be implemented in the WAN, and server load balancing that is used in the data-center. **Aster*X** analyzed the load balance of the traditional network that carried out by load balancer device and pointed out to several limitations. The first limitation is that load balancer is a choke point whereas it is placed on the entry of the network and all traffic must pass through it. In addition, this can cause a problem of a single point of failure. The second weakness of convention load balance is that servers are static especially in the virtual data center that allows VMs to move around for making efficient use.Moreover, load balancer had been designed to work in a regular network structure such as data center network, but in an enterprise network, it is difficult to support such type of load balancing systems. To overcome all these limitations, Aster*X came up with new SDN load balance with specific characteristics. The load-balance is distributed throughout the network for high scalability and logically centralized by SDN controller that has the entire overview of the network. Moreover, load balance system must be flexible to allow each service used the schema according to the service requirements. To achieve such load-balance system using SDN, Aster*x proposed three SDN modules; *Flow Manager* that controls and manages routes flow according to the specific load-balancing algorithm, *Net Manager* that is responsible for tracking network topology, and *Host Manager* to monitor the servers' state. Aster*x supports proactive and reactive modes as well as individual and aggregated requests.

2.5.5 SDN-SLB in virtual environment

In atraditional network, load balancer has some restrictions in Virtual Environment (VE), whereas, in Virtual Data Center (VDC), VMs are distributed to provide various services. For example, one VM can host more than one application or services that need different load balancing schemes. Thus, (Jethanandani et al., 2006) introduced dynamic load balancing architecture for clustered servers in VE. The architectureconsists of three components: Floodlight SDN controller, OpenFlow switch that support OF 1.3, and SAN storage. A Load balancing module was developed to manage the balance between the servers. The load of the servers is calculated based on three parameters: CPU, RAM, and response time. Such parameters are reported periodically to the controller that uses weight to indicate loads of each server when a new request is sent to the VIP. To differentiate between various services that are hosted in one VM, a parameter R was then introduced. In this solution, the architecture is designed for a heterogeneousenvironment. Thus the processing ability of each VM is considered. On the other hand, the probability ofselecting a server is implemented to avoid overload of the server node. This architecture has been tested in the real environment that included four virtual machines and iSCSI shared storage.

2.5.6 SDN scalable server load balance

The scalability of server load balancing in the SDN is one of the issues that has been addressed by various researchers. *Ananta*(Parveen Patel et al., 2013) and *HAVEN* (Poddar et al., 2015)are examples of the solutions that are focused on the the scalability of load balance with multi-tenant in the cloud environments. *Ananta* is layer-4 load balancing system that scale-out the web services. *Ananta* divided the data plane functionality into three separate tiers that include*network layer* (layer-3), *multiplexer's* layer, and *state-full NAT layer*. Ananta can provide scaling and reliable load balancing. The proposed architecture consists of *Ananta Manager* (AM), *control planeof the*

system, Mux Pools as well as monitors of DIP health. The Host Agent provides Direct Server Return (DSR) and NAT function across the layer-2 domain. The Multiplexer (Mux) receives incoming traffic and forward it to the appropriate DIPs. In addition, the proposed system supports multi-tenants for the Quality of Service (QoS) by dividing CPU, memory, and bandwidth resources based on the tenants' weights. The implementation of the system was carried out in the public cloud using Windows Azure with 20 VMs as the server and 10 VMs as clients. The measurements metrics focused on *Fast-Path*, *response time* and *latency*.

The second solution is HAVEN that provides scale-up and scale-down services. In HAVEN, when the PacketInis received from the client, the controller will start to calculate the Score Computation (SC) of each Pool.If the total of SC is greater than a threshold value, the system will then start scaling up by providing more VMs to that server pool. On the other hand, if the SC is less than the threshold, the system will then start a scale down the process by reducing the resource utilization across all the active pool members. The solution was deployed in a real cloud environment using OpenStack with seven servers that connected to 14 OpenFlow-enabled switches. The OpenDaylight controller was installed and configured on the physical server to manage the load balance system. The systemwas compared with HAProxy, a software load balance system, in terms of response time, latency, and overheating of the system. However, the HAProxy performance overcomes the HAVEN in terms of latency due to first PacketIn processing, but HAVEN showed better results in the response time and less overhead compared to HAProxy.

2.5.7 Slice load balancing

In the OpenFlow-based slice, the authors (Jarschel et al., 2014a) proposed a new SDN load balancing solution called "slice load balancing" that classified the traffic flow into two types. The firsttype is known as aggressive flows that include the packets with

high rate and minimum arrival time. The second type is known as the normal flow that includes the rest of the flows except aggressive flows. To identify aggressive flows, *temporal locality* is used based on the window size, the number of packets, and the time of packets' arrival. An experimental test was carried out in the real environment that included eight OpenFlow-enabled switches and three nodes as well as two Floodlight controllers that were installedon a different host. OpenFlow Slice Algorithm (OFS) is proposed and compared with Random and Probability Stride algorithms.

2.5.8 A heuristic load balance

The authors (Khattak et al., 2014) stated that the scheme of dynamic load balancing for the multipath that has been utilized in the datacenter (with congestion control) are not effective enough to provide load-balance. Therefore, they proposed SDN-based LB with a heuristic method to manage the load balancing in the datacenter. The system aimed to integrate Ant Colony Optimization (ACO) with SDN controller to select the best path and best server. Two parameters were considered to perform load balancing; (CPU) cycles to execute a job in a note and delay traveling on the link. The results of the proposed system were compared with the static algorithm and round-robin algorithms. This comparison is not satisfactory since the round robin does not consider the dynamic load balance parameter such as CPU, RAM while ACO algorithm is dynamic and take into account these parameters.

Table 2.5 shows acomparison of the numbers of the SDN- LB solutions that used different controllers, algorithms and OpenFlow switches that implemented in real and simulation environment. Most of these studies used OpenFlow protocol with OVS or real switches. In the table, the name of the controller and programming language that are used to develop the load balance module are shown. The proposed algorithms that carried out in each solution are presented along with compared algorithms. In the experimental environments, we focused on measurement metrics, traffic generating

tools. In addition, the number of the hosting pool and version of the OpenFlow protocol are considered. These parameters are derivedfrom taxonomy mentioned in section 2.4.

Table 2.5 Comparison between various load balance solutions

No	Solutions	Controller		Algorithm		Experiment Environment				OpenFlow Switch	
		Controller Name	Model language	Compared Algorithm	Proposed Algorithm	Measurement Metrics	Traffic Tools	Simulation	Servers Pools	OF	Type of switch
1	(Uppal & Brandon, 2010)	Nox	C++	Random, Round-Robin	Load-Based	First Packet, Latency, Throughput	Zipf distribution	Real	3 server 1 Pool	1.0	HP 5412zl
2	(Shang et al., 2013)	Floodlight	Java	Randomized Round-Robin	Load based Algorithm	-	-	Mininet	-	1.0	OVS
3	(W. Chen et al., 2014)	Floodlight,	Java	Round-Robin Random	SBLB	response timeSystem utilization (CPU-RAM	Ping – Send Real Traffic	Real	3 server	1.0	-
4	(Yilmaz, Tekalp, & Unluturk, 2015)	OpenDaylight	Java	-	Load-balancing OpenFlow controller	Delay	Real video streaming app(VLC)	Real	2 servers	-	-
5	(Govindraj, Jayaraman, Khanna, & Prakash, 2012)	Floodlight	Java	-	-	throughput bandwidth,	Iperf	Mininet	6 servers	1.1	OVS
7	(Surya Prateek & Ying, 2013)	NOX	C++	-	*Flow Algorithm*	-	-	-	3 clients 3 servers	1.0	OVS
8	(Koushika & Selvi, 2014)	Floodlight	Java	Round-Robin	ACO algorithms	Response Time Delay	-	Mininet	2 server 1 client	1.0	OVS
9	(Peng Wang, Lan, & Chen, 2014)	Floodlight	Java	FLARE TSBN	OFS Algorithm	delay	Iperf	Mininet	-	1.1	OVS
10	(H. Zhang & Guo, 2014)	Floodlight	Java	Round-Robin	-	Average response time	httping	Mininet	3 servers	1.0	OVS
11	(Y.-J. Chen, Shen, & Wang, 2014)	OpenDaylight	Java	-	-	Service delay	-	Real	3 servers	1.0	TP-Link WR1043ND SDN switches
12	(Kaur, Singh, Kumar, & Ghumman, 2015)	POX	Python	*Random Round-Robin*	Comparing Round Robin with Random	Transactions Per second (TPS), Response Time (RT)	OpenLoad	Mininet	3 server	1.1	OVS

13	(Qilin & WeiKang, 2015)	OpenDayLight	Script Python Used API	-	-	-	Real traffic	Mininet	3 server	1.0	OVS
14	(Ghaffarinejad, 2015)	OpenDaylight	Java	*Random Round-Robin*	-	File transfer Split rate	Real traffic	Real Environmen t	3 Servers	1.1	-
15	(W. Chen et al., 2015)	Floodlight,	Java	Round-Robin Random	SBLB	Throughput Response time Servers' utilization	-	KVMvirtual Machines + Mininet	3 servers	-	OVS
16	(Yong, Xiaoling, Qian, & Yuwen, 2016)	Pox	Python	-	a hybrid load algorithm	*Throughput*	Spirent TestCenter SPT-2U	Mininet	6 servers	-	OVS
17	(Handigol et al., 2010)	NOX	-	-	-	-	-	Real	-	1.0	-
18	(Poddar et al., 2015)	OpenDayLight	Java	-	Scale out and scale in	Transactions Per second (TPS), ResponseTime (RT)	HTTPerf ping	Real	12	1.3	OVS

2.6 The important of the service based load balance

Due to the increasing number of users and extensions of service types, the low performance and poor scalability of a single server make it the bottleneck of network services. The service pool system has higher performance and extensibility and is more convenient for management and maintenance. The pools is a set of independent computers interconnected through a high-speed network and managed as a single service. The internal structure of pool is transparent to clients.

The purpose of the load balancing is to distribute the network load to each host in thepool as fairly as possible. In the circumstances of heavy load, load balancing ensures quick service response using the set of server nodes with scalability and high performance.

The basic principle in service-based load balance traffic classification is that knowing which service is offered at given "network coordinates" (IP address and TCP/UDP port pair). Therefore, service-based classification relies on the observation of how hosts usually interact and on the assumption that certain hosts, typically called servers, perform similar interactions, usually offering a service

Numbers of researches related to the dynamic load balancing schema are proppsed. For example, (Jian Liu et al. ,2005)proposed a load balancing based on dynamic feedback which considers the performance and the actual load at each node.(Qi Zheng et al. 2011) proposed another load balancing method based on the classification of contents, which fully consider the user requests and the differences between the server nodes. It classified the user requests and distributed them to each node fairly, thus guaranteeing that each node gets roughly the same amount of requests.However, these dynamic load balancing method usually focus on only one type of service. When various services coexist, they treated all services as the same and deal with them in the same way, which may not be desirable in real world scenarios

2.7 Identificationtypes of the services

For categorizingthe types of various services, Traffic Classification (TC) (Dainotti et al., 2012) is utilized with different approaches. TC is a method used to classify the network traffic for different purposes such as security, QoS and traffic management. This section aims to review the traffic classification approaches that are used to classify traffic with the goal of identifying the service type for load balancing purpose. A variety of approachesare proposed for network traffic, and so in this section, we discussed these approaches which include; port number, deep packet inspection (DPI), Statistical information based and Behavioural and Statistical Patterns.

2.7.1 Port-based Approach

The first and common approach that is used in the traffic classification is port-based (Valenti et al., 2013). This approach depends on examining the communication ports of TCP/UDP that exist in a given packet's header and compares it with well-known TCP/UDP port numbers that are assigned by the Internet Assigned Numbers Authority (IANA). This approach provides fast flow classification because a port number can be accessed easily and is not affected by encryption. Thus, Access Control Lists (ACL) and firewalls utilize this approach that can achieve high precision and speed-up the processing of the comparison between incoming packet and stored rules. In addition, this approach is mostly useful for well-known protocols such as SMTP and FTP that use port 25 and 21 respectively. Nonetheless, the port-based approach is unreliable and incapable of classifying all protocols in a modern network environment such as SDN. For instance, some protocols such as passive FTP, SHTTP and Peer-to-Peer (P2P) bypass or use a temporary port that hides the original port and associates the application with other ports to avoid traffic filters or to hack the system. Another example is the Session Initiation Protocol (SIP) that is implemented with Real-time Transport Protocol (RTP) and uses random ports to negotiate the terms of the call.

Moreover, this approach fails when it comes to implementation in tunnels or Network Address Port Translation (NAPT) whereas certain protocols can be identified. In this manner, port-based number as a classification approach are considered obsolete, and other approaches have been developed to address port-based limitations. One of these approaches is the Deep Packet Inspection (DPI) that differs from the port-based as it looks to the payload of the packet instead of the packet's header.

2.7.2 Deep packet Inspection (DPI) Approach

DPI is the approach (Bujlow, Carela-Español, & Barlet-Ros, 2013) that saves signatures of the application protocols in the database and match incoming packet or flow with the stored signatures making it a very accurate approach. It only checks the payload of the packet and ignores the header of the packet. Such an approach is not implemented for only the network traffic classification but also utilized to identify malicious data for security purpose. Because it is effective, the DPI can be used in all solutions where accuracy is crucial. Nevertheless, in terms of computational power and high-speed networks, the DPI approach may be unfeasible because of the inspection process for each packet. Consequently, this approach can affect the performance of the network, and therefore other mechanisms have been proposed to check only a part of a packet or few packets of each flow to avoid the overhead in the network. In addition to performance issues, DPI faces legal issues when it inspects the contents of a packet due to privacy and confidentiality bounds. The main disadvantage of the DPI approach is that when traffic is encrypted, then it is unable to access the payload contents to get packet's signature. Moreover, DPI cannot classify specific packets without payload which may be malicious. The modification of the protocol or upgrading of the application may lead to changes in the signatures of the packet which in turn can prevent the packet from being classified or affect the DPI's performance. For example, if the DPI approach depends on specific applications signatures, and if these

applications change or that the traffic generated by other applications use the same protocol DPI then the approach may fail to classify the packet.

2.7.3 Behavioral and Statistical Patterns Approach

To identify application-level protocols, the DPI approach that has been discussed in the previous section is not often considered as a valid option. Thus, a new approach has been developed, referred to as behavioral and statistical patterns (Gill, 2014), which collects host-related information and then associates it to one or more application types. Three methods are implemented in this approach; Heuristics, Social Behavior, and Behavioural Signatures. In heuristics method, the number of hosts that act both as servers is considered as well as an IP that uses the same transport port more than certain times. *Social Behaviour* isdesigned to capture the behavior of hosts in terms of role in the connection (server or client), the transport layer information and the average packet size. The last method, Behavioural Signatures or Statistical Fingerprints is mainly used in P2P traffic whereas failed connections, the ratio of incoming and outgoing connections, and the information on the use of unprivileged ports are collected. Although this approach can store archives and easily be applied to unknown protocols, it is still characterizedby less accuracy level compared to other approaches.

2.7.4 Statistical information Approach

Typically, statistical information approach (Soysal & Schmidt, 2010) relies on flow or packet level features such as flow duration and size, inter-arrival times, IP addresses, TCP and UDP port numbers, TCP flags and packet size. This approach sometimes calls flow features that are utilized, individually or combined, to calculate statistical values. It can use simple measures as average or variance, or complex measures such as the probability density function. Generally, such an approach requires a learning phase to build a reference model that can be used to classify unknown traffic.

Machine learning is always used to build the classification model. The advantage of this approach is that there is no packet payload inspection involved.

2.8 Challenges, Open issues, and future research direction

In this section, we discuss the open issues and challenges related to the server load balancing in the cloud computing environment. In effect, we highlight the key areas and future research direction that needs to be addressed. Some issues are due to SDN architecture itself, such as reactive flow mode and service chain, while other issues are common load balancing issues. Table 2.6 presents open issues with their related challenging factors, including the proposed solutions that can be applied in the future to overcome the problems.

Table 2.6 Open issues, challenges, and future research direction

Open issues	**Challenges**	**Future research direction**	**Addressed**
Monitoring	Overhead in the controller	Design accurate and timely statistical monitoring system without overhead in the controller	Partially
Scalability	Dynamically added members to server pools	Building Auto scale load balancing system	Addressed, but still some limitations are remain
Load balancing with different types of services	Dynamically apply load balancing algorithm based on application type	Adaptive load balancing using traffic classification approach for building application aware load balancing	Not addressed
Reactive Flow and load balance	Extra latency due to processing PacketIn	Minimize the flow setup rate in the controller	Not addressed
Multi-tenancy and load balancing	Customizing the server load balancing services based on Service Level Agreement (SLA)	Developing a load balancing system taking into account multi-tenant users in virtualization environment	Partially
Load balance and Services chain	Response time of the servers	Auto and dynamic setup of server load balancing service	Not addressed

2.8.1 Monitoring

The ability to determine the health of the server is a crucial part of the load balancing system (Okamoto, 2001). Its related metrics, such as CPU load, memory load, and I/O must be reported to the controller for calculating the current load of each server.

Without this knowledge, the load balancing functionality could perform incorrectly and send falsified connection requests to different devices that are overloaded. Therefore, the controller must continuously monitor performance metrics of the server. Thus, accurate and timely statistics on network resources at different aggregation levels (such as flow, packet, and port) must be implemented. This is so in order to quickly adapt forwarding rules in response to the changes in servers' workload. However, confidential monitoring solutions such sFlow(M. Wang, Li, & Li, 2004), JFlow (Myers, 1999) and NetFlow (Estan, Keys, Moore, & Varghese, 2004) that collect either complete or sampled traffic statistics, send them to a central collector that imposes significant measurement overhead. Therefore, these approaches may not be as efficient solutions to be applied in SDN systems, such as large-scale data center networks. Even the SDN solutions that have been implemented to collect static information of network must seek more efficient monitoring mechanisms in order to achieve both high accuracy, and low overhead to provide effective load balancing mechanism.

2.8.2 Scalability

We can look to the scalability(Zhou et al., 2014) issue of load balancing in SDN from two perspectives; 1) the number of server pools that are managed by a single controller; 2) the number of server members in the single pool. Typically, load balancing in the cloud is required to create Virtual IP Address (VIP) that accepts all users' requests and distribute them to the physical server based on the specificpolicy such as static or dynamic. The server is added to the server pool associated with a specificprotocol, for example, TCP, UDP or ICMP. The number of server pools that are associated with VIP affects the performance of the load balancing, for instance,managing a large number of server pools with a different type of protocols may cause controller overhead (Tootoonchian, Gorbunov, Ganjali, Casado, & Sherwood, 2012). In addition, a number of server members in one pool also cause low-latency for

network performance especially in dynamic load balancing in which each server reports the load metrics to the controller periodically.

2.8.3 Load balances with different type of services

Currently, in virtualized data centers (Urgaonkar, Kozat, Igarashi, & Neely, 2010), one server can host different services and more services will be deployed by differentusers, and they will be moving around. Therefore, implementing one load balancing scheme for all services is not efficient. For example, when a user sends a request to the web server then the response must be returned as quickest as possible, and in this case, load balancing system must consider the CPU and RAM of the serverwithout considering the bandwidth of the channel. While a large file transfer such as FTP requires more bandwidth, load balancing system must consider this metric for the load-balance system. For this reason, different types of services require different sort of load balancing schemes. Typically, adaptive schemes in load balancing (Lee & Riley, 2005), used traffic classification to identify users' request. Based on the type of user requests, load balancing system can change the schema or adjust the load balancing parameters. In addition, load balancing as a service (LBasS) in the cloud provides load balance for multi-tenancy with different services that require different Service-Level Agreement (SLA). Currently, SDN load balancing solutions focus on providing dynamic load balancing via application controller without identifying the flow type that can be implemented in the data plane layer. Qosmos Company (Qosmos, 2016) produced a real-time traffic classification system that can be implemented in Open vSwitch to recognize traffic up to Layer 7 using Deep Packet Inspection DPI (Bujlow et al., 2013). Other solutions provided by (OVS, 2016) have added additional interface of DPI engine on top of Open vSwitch to manage the flow classification. The integration of these techniques with load balancing applications can produce effective LB systems.

2.8.4 Reactive Flow and load balance

There are two approaches to managing flow table in OpenFlow specification such as a) proactive flow (P. Lin et al., 2013) in which the controller sets up flows in advance and b) reactive flow (Dusi et al., 2014) where the controller responds to the PacketIn messages and dynamically update the flow table. In the load balancing system, response time is an important factor to measure the performance of the system. Thus, sending every packet that does not match in the flow table can cause extra latency due to processing packetIn messages in the controller. In addition, reactive mode costs extra overhead to the network due to frequently updating of flow tables (Kuźniar et al., 2015). On the other hand, setting up the flows in advance is not suitable for a load balancing system especially in the dynamic load balancing where the server load reports regularly to the controller. A good example, in this case, is that, after every 5 seconds, the flow table is updated based on the server load. To addressed this problem, a hybrid approach where some traffic is handled proactively and some are handledreactivelycan be used. Another solution is based on OpenState (Bianchi et al., 2014; Capone, Cascone, Nguyen, & Sanso, 2015), a Stateful data plane, that uses state machines implemented in the switches to reduce the need to rely on the remote controllers. Currently, only RYU controller can support this approach.

2.8.5 Multi-tenancy and load balance

One of the key challenges of load balancing in the cloud is to ensure the availability, scalability, and performance of all applications and tenant infrastructure, while continuously providing customized services per each tenant (Bezemer & Zaidman, 2010).To cope with dynamically increasing demands from multiple tenants, cloud service providers need to manage load balancing for the applications and services dynamically. Load balancing service in the cloud can be provisioned by sharing a load-balancer between different tenants or provide dedicated load balancing systems for each

tenant or even for each tenant's application. Sharing a load-balancer between different tenants may introduce tight interdependency between each tenant application. In addition, Service Level Agreement (SLA) (Pankesh Patel, Ranabahu, & Sheth, 2009) must be considered for the multi-tenancy (Tsai, Sun, Shao, & Qi, 2010) load balance especially when using LBaaS in the cloud environment. For example, a single cloud may provide multiple services, and each of these services can use a subset of the tenants. In such a tenant-partitioned deployment, the load balancers themselves need to be tenant-aware, in order to be able to route the requests to the proper tenant clusters. In other words, the load balancer has to be tenant-aware as well as service-aware.

2.8.6 Server Load balance and services chain

Service chain (John et al., 2013) refers to an ordered networking services such as firewall, load balancing, and IDS, into the path of applications. The traditional network service chains (Jung, 2011) are more complex, static, and error-prone because they require careful planning of the respective topology that is difficult to manage due to its traditional network. Although SDN and Network Function Virtualization (NFV) (Matias, Garay, Toledo, Unzilla, & Jacob, 2015) enable easy, agile, and manage deployment of service chain, still, in terms of server load balancing, several factors must be considered including position of SLB services into a service's chain, the length of a service chain, and auto configuration of the services.

In traditional networks, the load balancer (Tian, Zhao, Zhong, Xu, & Jing, 2011) is located after the firewall and provides additional services such as the NAT process (Gilly, Juiz, & Puigjaner, 2011). Therefore the position of the SLB into services chain is critical and has an effect on the performance of the services. The second factor is the length of a service chain that affects the process of the load balancing. For example, if the service chain is prolonged and it includes numbers of the services then the communication between users and servers will take longer time resulting into large

response time. In addition, the auto-configuration(L. E. Li & Woo, 2011) of the SLB service allows dynamically setting up the load balancing policy that always changes according to the tenant requirements

2.9 Conclusion

In this chapter, we have reviewed SDN solutions in terms of the server load balancing. First, we introduced the SDN architecture and then compared it with the traditional network architecture while focusing on the SLB service in the cloud. In the process, we developed the SLB taxonomy that is divided into four parts. In the first part, we pointed out the approaches and techniques that are integrated with the SDN to provide SLB solutions. In the second element of the taxonomy, we presented various open source and commercial controllers that facilitate SLB. We focused on the Floodlight controller that is used in this study. The load balance algorithms, both static and dynamic that have been implemented in the existing SD-SLB solutions were also discussed. In addition, we reviewed the experiments' environments that are used in the SD-SLB solutions; it is shown that most of the solutions use Mininet simulation. Further, the approaches that are utilized to recognize the type of services have been illustrated as well. Finally, as a result of this review, we have highlighted a number of open issues and the related challenges of the SDN-SLB especially the SLB with different types of services in the cloud. In the next chapter, we will discuss the solution that addresses the problems.

CHAPTER 3: **SERVICE BASED LOAD BALANCE MECHANISM: PROBLEM ANALYSIS**

This chapter aims to analyses the problem that was highlighted in chapter 1 and conducts a deep investigation to show the impact of the user's requests in the SLB system. We focused on the impact of the different types of requests that use the same Load Balancing (LB) scheme. This scheme used different algorithms that aim to select the best server for handling the incoming request. In order to analysis the problems, various experiments are performed with statistical analysis.

This chapter is divided into four sections; in section 3.1, we explain the LB system discretion that includes pre-steps involved in conducting the experiment, definitions and the configuration of Virtual IP (VIP). The experimental setup and the related network model adopted in these experiments are discussed as well. In addition, experimental model along with the performance metrics, prototype application, and the benchmark is presented in this section. In section 3.2 we analyze various empirical experiments to show the influence of the users' requests on the load balancing system. This section is further divided into subsections that focus on different metrics which include Average Response Time (ART), Reply Time (RR) and Request per Second (RPS). In section 3.3, we illustrated some example of the impact of the user's requests on the hosts' performance. This chapter is concluded in section 3.4.

3.1 LB System Description

The problem has been analyzed through a series of experiments conducted in Mininet(Kuźniar et al., 2015), SDN simulation environment. Floodlight (Govindraj et al., 2012) controller is the controller of choice in this study which was installed on a separate computer and configured to connect the network mode in Mininet. Before we explore on the experiments, pre-steps are discussed to explain how the load balancer works.

3.1.1 System Definitions

In this section, we discuss the details of the configuration and operation of load balancing in the cloud using SDN and outlining the system definitions. In the SDN network, a controller can connect to one OF(McKeown et al., 2008) switch or more, and this switch is connected to another OF switch or a normal switch. Each switch connects to a number of Hosts which are called *Host Pool.* A Host Pool consists of several host nodes that provide different services such as HTTP, FTP. Each host pool can be viewed as one virtual server by the users. The SDN controller receives a request from a user (as PacketIn message)(Phemius & Bouet, 2013), then, chooses the best host to process the request, and updates the flow table in the switch based on the load balancing policy. The SDN load balancing system components are listed in Table 3.1 below.

Table 3.1 SDN-SLB Components

Components	**Definition**
Host Pool	The container of the Host associated with specific VIP
Members	Number of the Hosts in one Pool
Host	The server that is hosted in the pool
VIP	Acts as the proxy between the users and the server Pool
Controller	SDN controller (Floodlight)
OpenFlow switch	OpenFlow device (OVS)

Typically, before starting the load balancing system, the configuration of pool and VIP is important. Therefore, we utilized the Restful API that is provided by Floodlight controller to configure the load balancing system. Figure 3.1 shows the load balancingconfiguration in the Floodlight controller.

```
curl -X POST  d '{"id":"1","name":"vip1","protocol":"TCP","address":"10.0.0.100","port":"8"}' http:// 192.168.1.2:8080/quantum/v1.0/vips/
curl -X POST -d '{"id":"1","name":"pool1","protocol":"TCP","vip_id":"1"}' http://192.168.1.2:8080/quantum/v1.0/pools/
curl -X POST -d '{"id":"1","address":"10.0.0.1","port":"8","pool_id":"1"}' http:// 192.168.1.2:8080/quantum/v1.0/members/
curl -X POST -d '{"id":"1","address":"10.0.0.2","port":"8","pool_id":"1"}' http:// 192.168.1.2:8080/quantum/v1.0/members/
curl -X POST -d '{"id":"1","address":"10.0.0.3","port":"8","pool_id":"1"}' http:// 192.168.1.2:8080/quantum/v1.0/members/
curl -X POST -d '{"id":"1","address":"10.0.0.4","port":"8","pool_id":"1"}' http:// 192.168.1.2:8080/quantum/v1.0/members/
```

Figure 3.1 Configuration of the load balancing system

In the first step, the VIP is created and given a unique name; this name is associated with the IPv4 address and TCP or UDP protocol with aspecific port. Then, the Pool is set up. The last step is to add the Host to the Pool; each host has a name and ID linked with a physical IP address of the Host. This configuration is different from the controller to other. For example, OpenDayLight(Medved et al., 2014) controller defines the load balancing policy in the configuration steps while Floodlight allows the selection of policy inside the module.

3.1.2 Experimental setup and network model

In our experiments, we configured the load balancing system and identified two server pools, HTTP service pool, and FTP service pool. Each one includes five host members. Two hosts are selected as clients to send a request to the VIP that can accept the request and distribute the traffic based on the implemented policy. We used Mininet for the network emulation; The SDN simulation allows developing SDN solutions, as well as creatingand managing the prototype of the system. Mininet can create SDN network elements such as host, switch, links and controller on a standard Linux environment. It allows customizing the network topology in a single desktop/laptop. A simple network can be created by using a command-line tool; thatis “mn.” Mininet has an API that allows customizingthe topology via a python script. To design a custom network topology, we developed a Python script which in turn created a network topology as required. Figure 3.2 below shows the network topology that was deployed in our experiments. The topology consists of the two pools that connect with traditional

switchesand two hosts that are used for sending the requests. Each pool includes five hosts with adifferent type of services. The first pool runs HTTP service and second pool runs FTP service. SDN controller is connected with OF switch that links to three normal switches. H12 and H11 used as aclient to send a request to the hosts

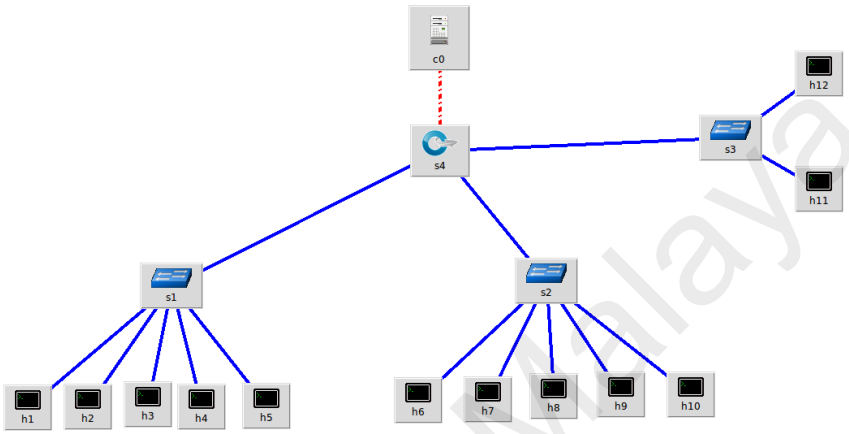

Figure 3.2 The network topology used in Mininet

Table 3.2 shows the specification of the VM that used to run Mininet simulation. Floodlight controller and OVS are installed in same VM.

Table 3.2 System specification of the Mininet

Software and Hardware	**Specifications**
Processor	Core i7
RAM	2 GB
Operation System	Ubuntu 14.04
SDN controller	Floodlight 1.0
Open VSwitch	Version 2.3 support OF 1.0

In order to launch a customized network topology, the following command in Figure 3.3 is executed in Mininet CLI:

```
Sudo mn --custom ~/mininet/custom/Loadbalance.py --topo LBtopo
```

Figure 3.3 Mininet command to run a custom topology

After execution of the above command, a network model is created with the followingparameters:

- 12 virtual hosts , each having an IP address ranging from 10.0.0.1 to 10.0.0.13
- 1 OpenFlow VS with supporting OF 1.3 and two traditional switches.
- MAC address of each host being set as equal to its IP address.
- The first five hosts (1~5) represent Pool-1 and the second batch of hosts (6~10) represent Pool-2, 11 and 12 are the clients that send traffics to these pools.
- The Controller is run on the local host (127.0.0.1) and connects to the switch via port (6653).

3.1.3 Experimental Model

In this section, we explain the experimental model that includes performance metrics, prototype application, and benchmark tools. For ensuring accuracy and reliability of obtained data, the execution process for each value is repeated 30 times. In addition, experiments are performed for 15 different sample workloads. The confidence interval's attribute shows the possible range of the sample means with 95% confidence for the sample space of 15 values in each experiment. A simple script is developed for data collection after each experiment for analysis based on the parameters and metrics that are used inthis analysis. Various types of metrics are adopted in this analysis.

Table 3.3The list of the parameters and metrics

Parameters	**Metrics**	**Definition**	**Types**
Response Time (RT)	Average Response Time (ART)	Time spent between sending the last byte of the request and receiving the first byte of the response	Time
Throughput	Reply Time (RT)	Time spent between receiving the first byte of response and the last byte of the response	Time
	Request per second (RPS)	Total number of requests processed per second	Number(requests)

Our objective is to propose a mechanism of SBLB that can minimize the response time and maximize the throughput of the system. Consequently, these two parameters are selected, where the average response time (ART) is used as a metric

representing the response time while for the throughput we used four metrics named Reply Time (RT), and Request per Second (RPS). Table 3.2 shows the definitions of parameters that are associated with the metrics as well as the type of the metric.

We used the default prototype load-balance application by the Floodlight controller. To enable the application, we changed the Floodlight configuration properties to load the application when the controller is running. The application has been used with basic load balancing scheme that includes Round-Robin and Random algorithms. The application used *static-flow-pusher* API to insert flow entry into a flow table. This API used 0 *idle-time-out*, meaning that the flows' rules will remain in the flow table and applied for all incoming traffic. For example, when a client sends a request to the VIP that selects one host, say H1, after some time, if the same client sends a request, then, the switch will forward it to the same host H1. We changed the *Idle-time-out* to become 5 seconds. This means that after 5 seconds, the flow entry will be removed to allow new request being handled by another host.

Three different types of the benchmark tools were deployed in these experiments in the Mininet simulation environment. We used these to generate various types of traffic in terms of different number, types, and size of the requests with respect to the number of concurrent users in the system. For example, Figure 3.4 presents a different number of requests with different connection rates. Typically, a connection rate is less than the number of the connections. As depicted in Figure 3.4, when the number of requests is 190, the connection rate is recorded as 180. While in the first three requests of 50, 60 and 70, the request rate is not much different. To avoid the increasing of dropped packets due to the high number of requests.We opted to stick with only 15 samples for each experiment.

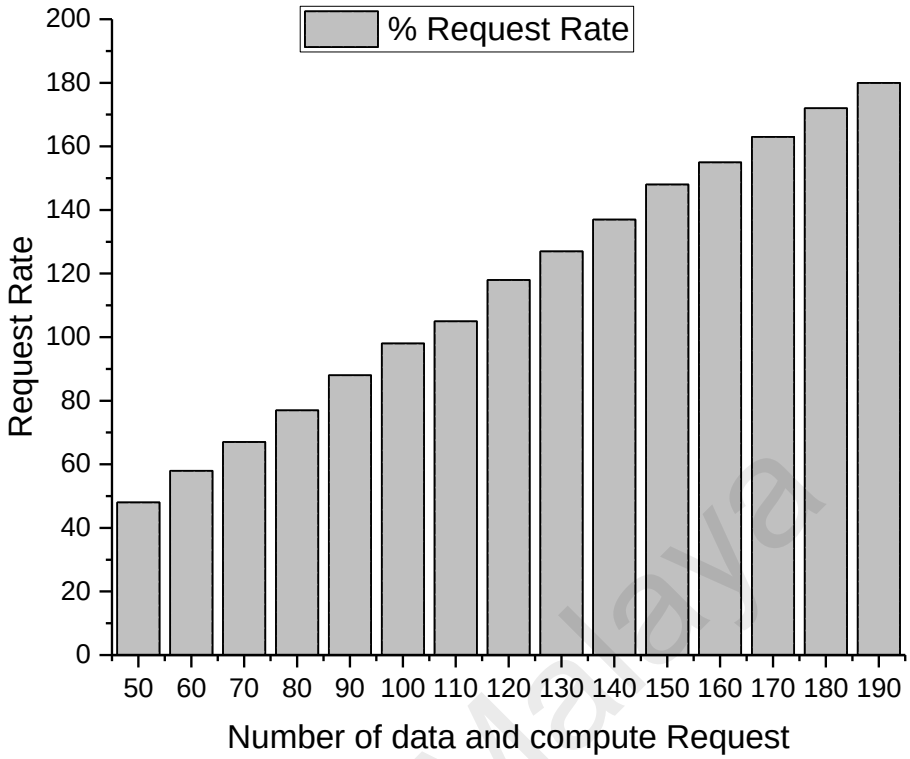

Figure 3.4Relation between request rateand number of the request

3.2 Empirical Analysis of user's request into load balancing system

In this section, we discuss our empirical findings concerning the impact of user's requests on the performance of the load balancing system. We first present the average response time. This is followed by; reply time and request per second and finally, the impact of the user request on the host are presented.

3.2.1 Analysis of the Average Response Time (RT)

Average response time is one of the critical metrics in load balancing systems. It is interpreted as the amount of time taken to return the results of a request to the user. The response time is affected by various factors, for instance, bandwidth, the number ofusers who access the system at the same time, thenumber of requests and average thinking time. to get faster responses, a high number of requests per second must be processed.Thus, we can calculate the response time as follows:

$$RT = U_N/R_N - Tt \qquad \textbf{(3.1)}$$

Whereas $U_N$is a number of the concurrent users and R_N= a number of the requests per second, T_t = Thinking time per request. For example, if we assume that the maximum number of users that can access the system at the same time are 4000, and the maximum requests that can be handled per second is 1000. Moreover, if the average thinking time per request is 3 seconds then we can calculate the response time as follows:

$$ART = (4000/ 1000) - 3\ sec. = 4 - 3\ sec = 1\ second$$

The response time is 1 second. In the SDN environment that uses reactive mode, the thinking time for the first packet may take more than 3 seconds because of PacketIn setup rate.

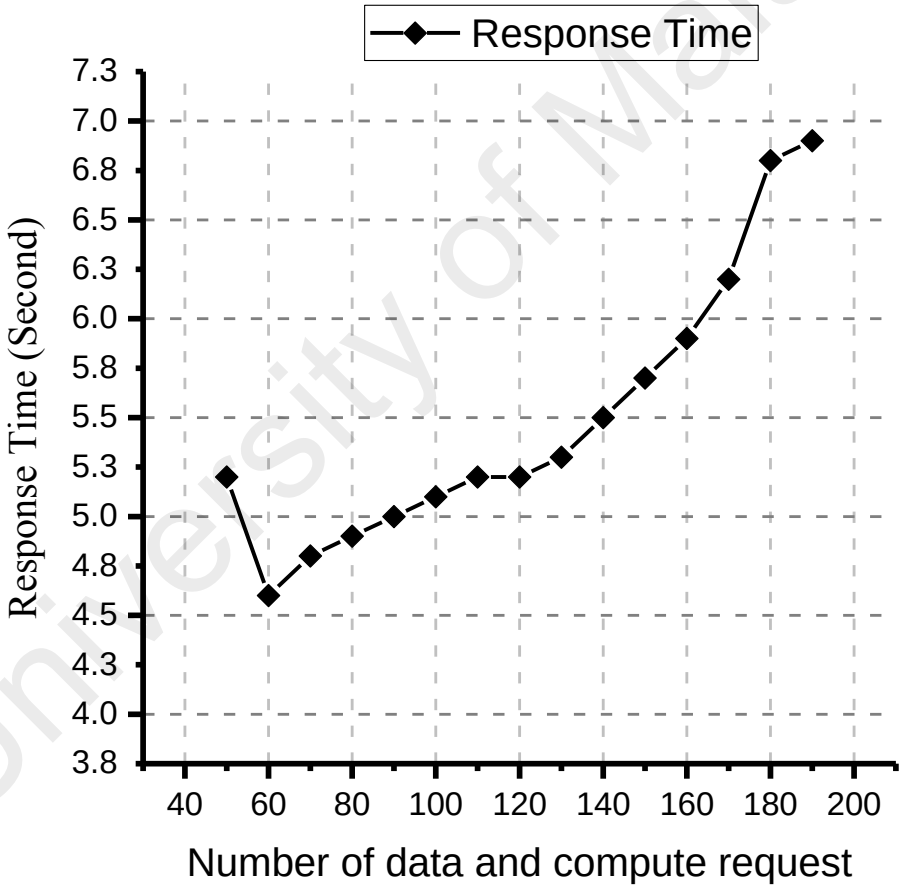

Figure 3.5Impact of the number of different request with ART

There are several factors that affect PacketIn setup rates such as the type of controller used, the number of switches connected to the controller and the packet forwarding rate of the switch(Kuźniar et al., 2015). Figure 3.5 presents the effect of the

number of requests to response time. We increase the number of requests gradually from 50-190 requests per second. The first average response time was recorded as 5.2s.tThis wasdue to the additional time for processing packetIn. The default flow-entry action of the Floodlight controller is "controller" which means "send the packet to the controller."We can note that the ART decreased from 5.2s to 4.6s when the number of requests increased from 50 to 60 requests. However, the ART started increasing gradually to reach 6.9s with the increasing number of requests from 60 to 190. The request rate, in this case, increased from 48.2 with the lowest number of requests to 180.1 with the 190 requests and the 6.9s ART.

Table 3.4 Average response time with increasing number of the requests

Number of requests	**Request Rate**	**ART**
50	48.2	5.2
60	58.1	4.6
70	67.5	4.8
80	77.6	4.9
90	88.9	5.0
100	98.6	5.1
110	105.3	5.2
120	118.5	5.2
130	127.8	5.3
140	137.1	5.5
150	148.5	5.7
160	155.4	5.9
170	163.7	6.2
180	172.2	6.8
190	180.1	6.9

Therefore, when the number of the requests increases the response time increased as well. Table 3.3 shows 15 samples of the different number of requests with various request rates. The request rate illustrates that the number of the rate for each sample is usually less than the number of the requests. For example when we send 70 requests then the request rate is 67.5. The maximum request rate is 180.1 of 190 requests per/second. This is simply due to the fact that, not all requests are being processed at the same time.

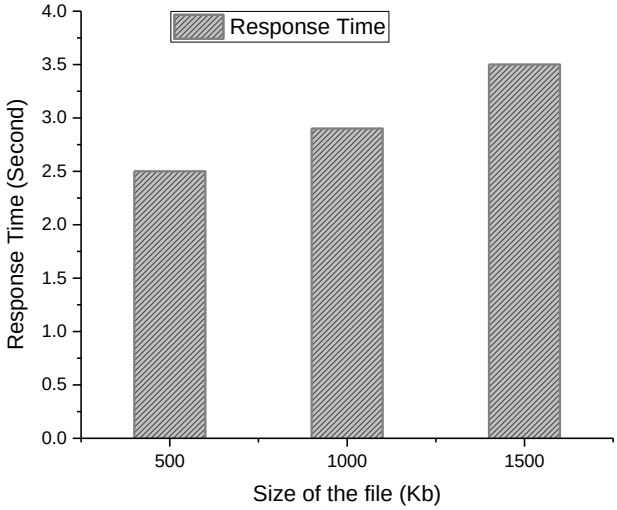

Figure 3.6 Size of the file and response time

The second metric that can affect the ART is the size of the request. We implemented the second experiment with three different sizes of the request and captured the ART for first 5 experiments. Figure 3.6 illustrates that when the size of the request is 500K, then the response time is recorded as 2.5s with 50 requests per second. Thus, as the size of request increases further, the RT then starts to increase as well. The results for different request sizes along with a various number of requests for each sizeare presented in Table 3.4.

Table 3.5The impact of the request size into RT

File size	Number of requests	Request rate	Standard deviation	ART
500K	10	9.9	0.4	1.5
	20	19.6	1.3	1.0
	30	29.1	2.5	1.2
	40	38.3	3.9	2.1
	50	47.4	5.8	2.5
1000k	10	9.9	0.4	1.2
	20	19.6	1.3	1.2
	30	29.1	2.4	1.1
	40	38.3	4	1.8
	50	47.4	5.8	2.9
1500K	10	9.9	0.4	1.2
	20	19.6	1.3	0.9
	30	29.1	2.4	1.2
	40	38.3	4	2.1
	50	47.4	5.8	3.5

3.2.2 Analysis of the Reply Time (RT)

It is sometimes called Transfer time which equals the time between the first byte of response and the last byte of response. Typically, it is less than Average Response Time ART and impacted by the user requests. The following equation (Equation 3.2) illustrates the formula involved in calculating the RT.

$$RT = T_{fb} + T_{lb} \quad (3.2)$$

Whereas T_{fb} is the time of the first byte of the response,while $T_{lb}$is the time of the last byte of response, RT is normally measured in seconds.Figure 3.7 shows the percentage of the RT for 50 requests per second. In the 50 requests, 50% of the requests could be replied within 1055 msec while with the 80% of the requests, the RT increased slightly to 1109 msec. We note that when the percentage of the requests is 90%, the RT then is increased sharply to reach 3013msec. This sometimes happens due to the capacity of the host or availability of the bandwidth. However, in general, we found that when the number of the requests increases then the RT increases as well.

In this experiment, we recorded the RT for the specific number of requests and presented it in the percentage form

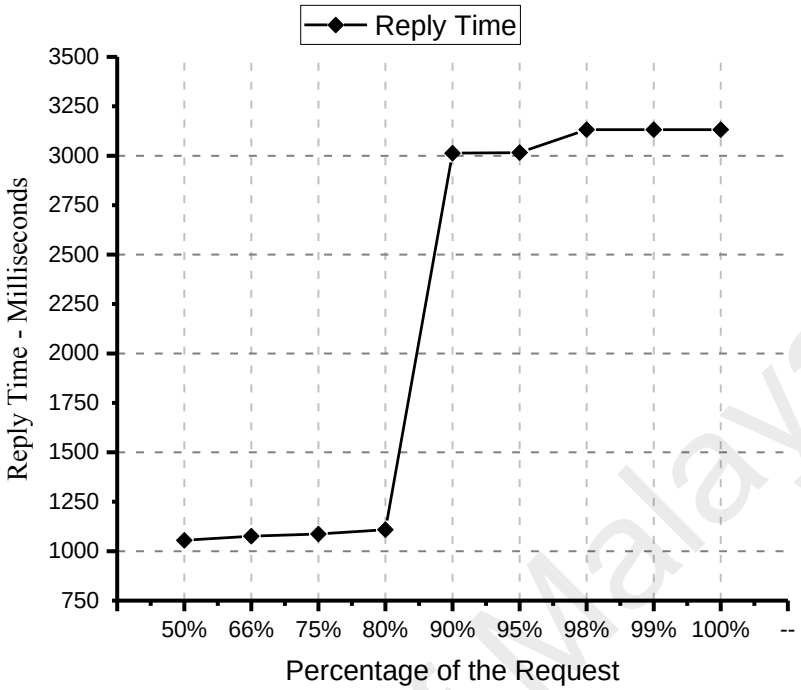

Figure 3.7 Reply Time per percentage

For example, with the 70 requests, 50% of the requests are replied to within1064*ms*,while the total RT of the 70 requests is recorded as 7063ms. We notethat in Table 3.5, the average difference of the RT between 50 requests and 100 requests is 44%. This indicates that the number ofrequests impacts the RT.

Table 3.6 The percentage of the request and RT of different request number

Number of requests	percentage	Reply Time (msec)
50	50%	1055
	66%	1076
	75%	1087
	80%	1109
	90%	3013
	95%	3016
	98%	3132
	99%	3132
	100%	3132
60	50%	1095
	66%	3017
	75%	3028
	80%	3048
	90%	7018
	95%	7031
	98%	7034
	99%	7055
	100%	7055
70	50%	1064
	66%	3006
	75%	3025
	80%	3058
	90%	7018
	95%	7037
	98%	7044
	99%	7063
	100%	7063
80	50%	1045
	66%	1108
	75%	3012
	80%	3029
	90%	7023
	95%	7051
	98%	7062
	99%	7073
	100%	7073
90	50%	1074
	66%	3007
	75%	3019
	80%	3030
	90%	7022

	95%	7036
	98%	7067
	99%	7081
	100%	7081
100	50%	1022
	66%	1057
	75%	3007
	80%	3009
	90%	3065
	95%	7019
	98%	7022
	99%	7074
	100%	7090

3.2.3 Request per second (RPS)

To understand the impact of requests on the load balancing system, this section discusses the RPS that measures how many requests are processed in the load balance system per second. This can be calculated by the given equation 3.4.

$$RPS = \left(\frac{\mathrm{R1} + \mathrm{R2} + \cdots + \mathrm{RN}}{\sum T}\right) \qquad 3.4$$

If we can take the HTTP service as an example, then one request to the web site may call 20 to 100 images per page, and the size of these images are varied (e.g. 2 MB to 50MB per images). In this case, the number of the RPS will increase. By contrast, a request that targeted simple text pages will produce a higher number of the request per second. This is due to the fact that simple text can be processed by the web server itself while a big size of the image requires expensive processing that takes some time before the response is sent.

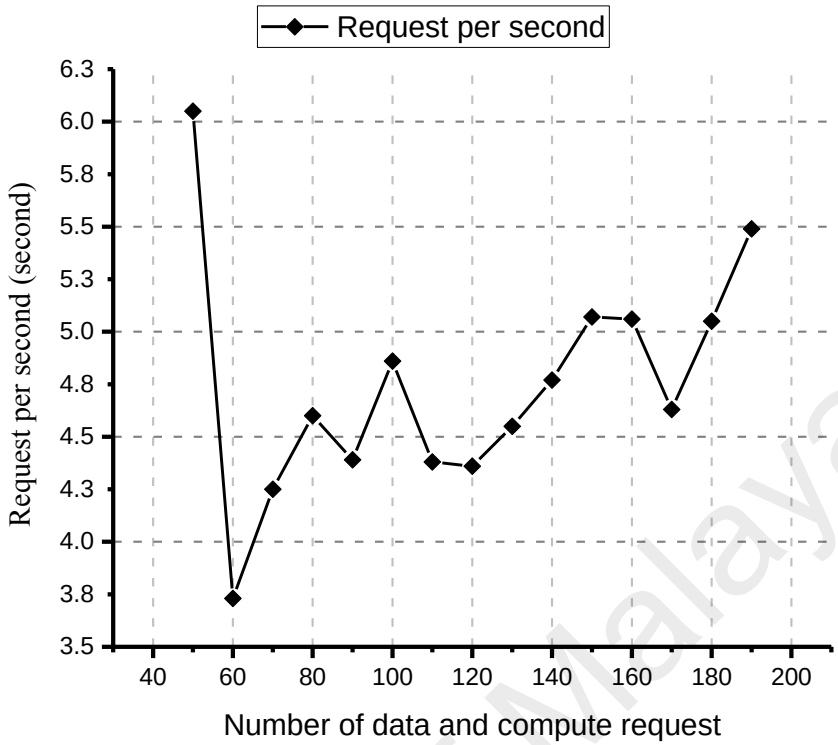

Figure 3.8The request per second

Figure 3.8 shows the variety of the RPS in the conducted experiment that placed different number and types of the requests to analyze the impact of these requests on the RPS. It can be seen, in 80 requests per second, the RPS recorded a value of 4.6 while in 70 requests the RPS was observed to be 4.25. Table 3.6 presents 15 samples of the experiment that show the RPS and time per request. In addition, the capacity of the host to handle the requests affect the RPS as well. Thus, in the next section, we will discuss the impact of the requests on a host.

Table 3.7The impact of the type of the request into RPS

Concurrent users	Request number	Time of test(Seconds)	Requests per second(mean)	Time per(MS)
10	50	8.266	6.05	165.32
10	60	16.097	3.73	268.285
10	70	16.467	4.25	235.242
10	80	17.378	4.60	217.22
10	90	20.517	4.39	227.97
10	100	20.555	4.86	205.552
10	110	25.134	4.38	228.492
10	120	27.539	4.36	229.493
10	130	28.563	4.55	2019.715
10	140	29.34	4.77	209.572
10	150	29.576	5.07	197.174
10	160	31.637	5.06	197.731
10	170	35.661	4.63	215.803
10	180	35.661	5.05	198.114
10	190	34.596	5.49	182.087

3.3 Impact of the requests on a host load

For understanding the impact of the user requests on a host's load; three factors must be considered. These factors are; *Observation time* OT, the total amount of time that the server is being monitored, *Busy time* BT which is the total amount of time that the server is active during OT) request a total number of requests that have been completed during OT.

Therefore, from the above factors, we can calculate the CPU Utilisation using the following equation.

$$U = BT/OT \qquad 3.5$$

The equation can show the percentage of CPU capacity during a specific period of time. For example, if we observed the host for 60 seconds OT and during this time say 90 requests (R) were completed, then the busy time of the host is actually 48 seconds BT. So, CPU utilization is 80% which is simply 48/60 x 100%.

In addition, we can get the Average Request Time ART, meaning the amount of time that a request need to be processed, sometimes it is called the Average Reply Time ART. Equation 3.6 shows that.

$$ART = BT/R \quad 3.6$$

Whereas BT and Rare 48 and 90 respectively, therefore the ART is 0.53/sec. Moreover, we can get the *throughput* of the system by dividing the number of completed requests by the observation time as we can see in the equation 3.7. By using the above case, the throughput is 1.5 request per second, where R=90 requests and OT=60sec. This means that the host can handle the average of 1.5 requests in every second.

$$Throughput = R/OT \quad 3.7$$

To know the capacity of the CPU for handling the number of servers, we can use equation 3.8.

$$CP = 1/ART \quad 3.8$$

So in the above example, we can say it is 1/0.53 = 1.875 request/sec. Typically, when a number of requests are sent to the host, then the host will create a queue in which the requests will wait to be processed. To get the length of this queue, we can use equation 3.9.

$$Q = U\,(1 - U) \quad 3.9$$

Giving (0.8/1-0.8) = 4 requests, where U is the CPU utilization. In turn, this shows the average number of requests during a specific period of time.

3.4 Conclusion

This chapter aimed to analysis the problem of the impact of theusing some load balance schema with adifferent type of the servicesand the related host's response. Several experiments were conducted in the Mininet, and various types of requests were generated. First, we covered the setup for the experimental environment and then developed the Python script which allowed the creation of a custom topology. Such

script configured two server pool connected to the Floodlight controller that runs the default load balancing application. Two hosts were used as clients to generate the required traffic.

Various types of requests were generated to show their impact on the load balancing system. The type of the requests, the size of the request, and the number of the requests were analyzed. Enhancing Httperf tool was utilized to generate the traffic, where this tool can generatedifferent types of the treffic such HTTP, FTP.In this chapter, we analyzed the problem by simulating two pools (HTTP and FTP) and used two hosts as clients to send requests to VIP.In each experiment, we used to send data requests constituting of 50% as simple HTTP requests while the other 50% is containing the FTPrequests. Three parameters were utilized to study the influence of the requests on the load-balancing system; these parameters include; average response time, reply time and request per second. Moreover, we presented a study on the impact of the requests on the host and showedby means of illustration some examples demonstrating the effect of the requests on CPU, and throughput of the host.

CHAPTER 4: **SERVICE BASE LOAD BALANCE (SBLB): DESIGN AND IMPLEMENTATION**

In this chapter, we present the system design and implementation of Service Based Load Balance mechanism (SBLB). The proposed mechanism aims to minimize the response time and maximize the throughput. First, we discuss the application modules of the SDN controller. Subsequently, with the use of schematic diagrams, the system architecture is illustrated. Then, the functionality of each module is discussed in details. Finally, the system design and implementation of the proposed SBLB mechanism are presented.

This chapter is divided into four sections. In section 4.1, we present the steps to develop application modules and utilize the core modules of Floodlight controller. The comprehensive description of the system architecture of the SBLB mechanism is presented in section 4.2. In section 4.3, we discuss the building blocks of the system architecture of SLB mechanism that includes three sub-modules namely service classification, load balance, and monitoring. In section 4.4, we show the use-case diagram that illustrates the process flow of SBLB. The conclusion of the chapter is presented in section 4.5.

4.1 Development of the Modules in Floodlight

Currently, there are several SDN controllers available, e.g., NOX(Tavakoli, Casado, Koponen, & Shenker, 2009), Beacon(Boero, Cello, Garibotto, Marchese, & Mongelli, 2016), Floodlight, OpenDaylight, BVC and HP VAN Controller(Tourrilhes et al., 2014). Some are Open-source while others are commercial and proprietary. Typically, the standard southbound interface such as OpenFlow can be used with all controllers, but the southbound interface is different from one controller to another. It depends on the technology that is used to develop the controller such as platform, framework, programming language and operating system that the controller supports. In this

study,we used Floodlight controller version 1.2 with OF specification 1.3. Floodlight is an open source controller, Apache-license, and Java-based. The Floodlight core architecture is modular, with components including topology management, device management (MAC and IP tracking), path computation, infrastructure for web access (management), counter store (OpenFlow counters), and a generalized storage abstraction for state storage. The Floodlight controller realizes a set of common functionalities to control and inquire an OpenFlow network, while applications on top of it realize different features to solve different user needs over the network. Floodlight can be configured to load different modules to accommodate different applications. A lot of applications have been built to work with Floodlight through three main i.e. APIs, REST applications, Module applications, and OpenStack (Karacali & Tracey, 2016) applications. We utilized these three APIs to build our SBLB mechanism. The steps are illustrated in Figure 4.1. First, we define the scope and functionalities of the modules by defining the interfaces dependencies and constraints of each module. For example, in our SBLB module, *ILoadBalancerService*, *IOFMessageListener* and *IFloodlightModule* interfaces are implemented, and all dependencies that allowexploiting other core functions of the controller are declared in the *initfunction*. In the second step, we define the event and how the module handles the PacketIn messages. In addition, we define the store and thread to collect server's information. This function is implemented in *monitoring* module that collects the host's information every five seconds and sends it to the *load balance* module.

After that, we created our Java (Gosling, 2000) classes and imported the libraries that are used in each class. The subclass such as *LBVIP*, *LBMembers*, and *LBPool*are constructed with necessary variables.

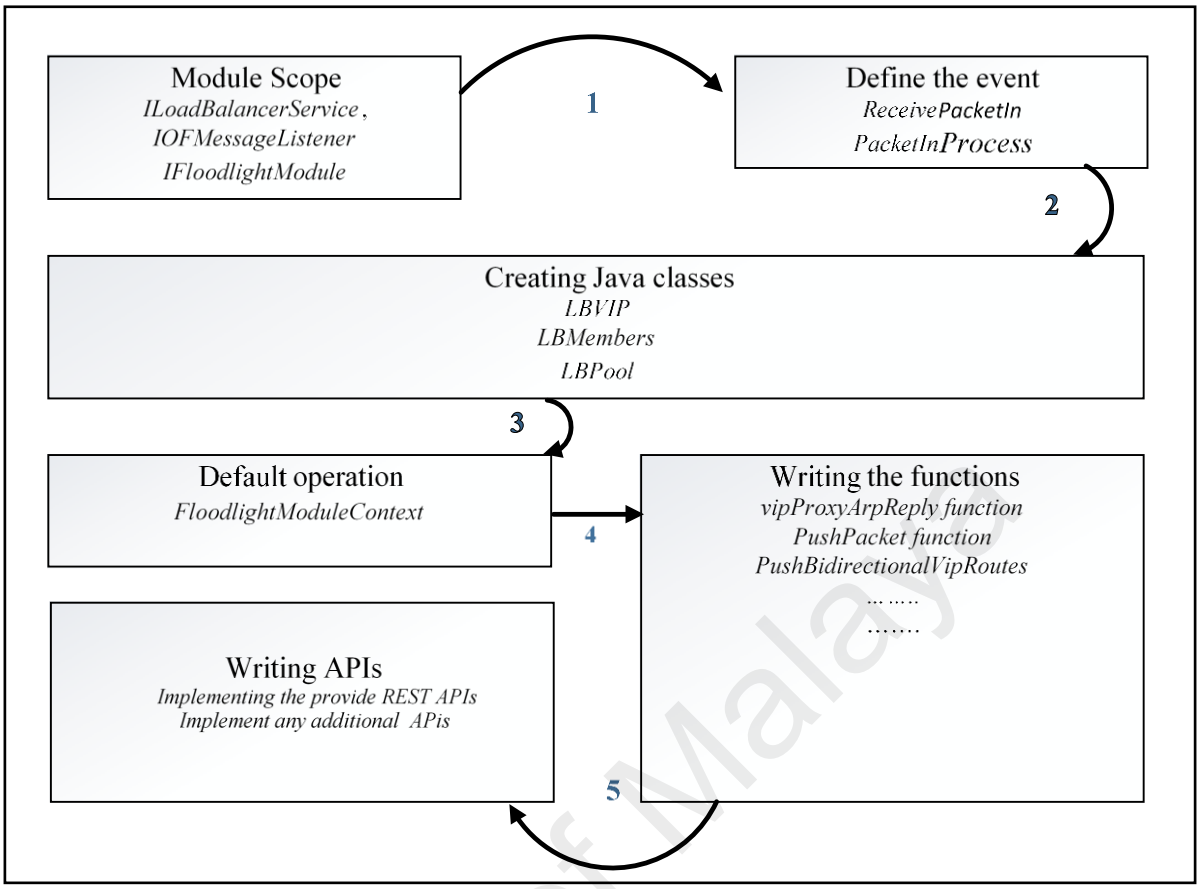

Figure 4.1 Floodlight modules processing steps

In step four, we write a basic operation that allows the modules to be loaded and initialize the data structure by taking*FloodlightModuleContext* as input. The last step is concerned with writing the functions of the modules. We listed the main functions with abrief explanation of each one below:

1. **Receive function**: in this function, the controller receives the message from the switch. When the message arrives at the switch, and there is no flow entry is setup, the controller will check if the message is PacketInmessage or not, if yes, the packet is sent to processPacketInfunction.
2. **Process PacketIn function**: Once the PacketIn message is received, the message is verified to check if it is an IPv4 traffic or not, if yes, the packet is parsed to get the details such as *sourceIP*, *desinationIP*, *traffic pattern* (TCP, UDP, and ICMP) and

the type of services. Then, the server member is selected from the pool to handle the PacketIn message based on the load balance policy.

3. **VipProxyArpReply function**: this function is responsible for broadcasting the VIP and MAC among the clients by implementing *Pingall* that builds the ARP table in the network.
4. **PushPacket function:** this function sends the PacketOut from the controller to the switch and instructs it on how to deal with the packet.
5. **PushBidirectionalVipRoutes function:** itis responsible for getting source IP of the PacketIn (VIP) and changes it to physical host IP that handles the incoming request. This operation is the reversed process of the PacketOut which is sent to the client.

4.2 System Architecture of SBLB

Our system architecture consists of two parts; the application modules that run ontop of the SDN controller and servers pools that connect to the controller through OpenFlow switches. For the implementation of the SDN application modules, we use the Floodlight controller which is one of the well-known SDN controllers written in Java. For the implementation of the server pools part, we used RESTFULL (Representation State Transfer) API to communicate with SDN controllers and OpenFlow switches. In addition, since we need dynamically modify the OpenFlow table, we use Open vSwitch [62] which is the OpenFlow reference implementation.

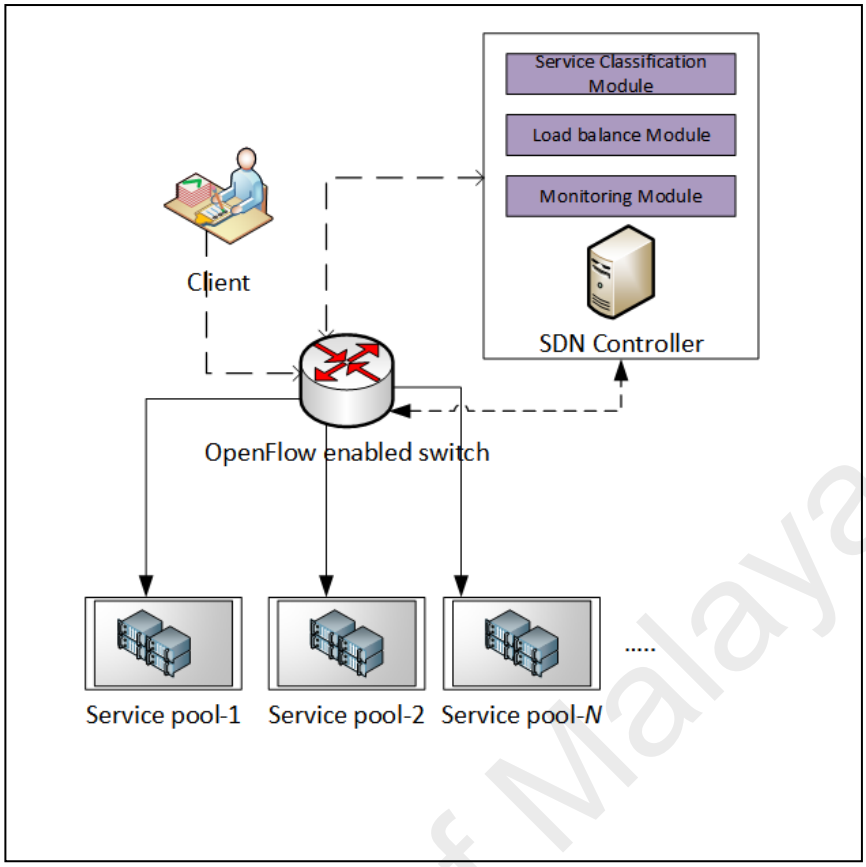

Figure 4.2 SBLB system architecture

Figure 4.2 shows the SBLB system architecture. The application modulesconsist of three functional modules namely Service Classification Module (SCM), Dynamic Load Balance Module (DLBM) and Monitoring Module (MM). These modules run ontop of each SDN controller. Monitoring module runs on *intfunction* of Floodlight to collect hosts information every five seconds and sends it to the load balance module. The main function of SCM is to identify the type of request to define the service type and send it to load balance module. DLBM is the main module that manages the load balance system by adjusting the corresponding parameters. A controller works as a master controller that handles all packetIn messages coming from OpenFlow switch. In addition, the controller manages the host pools and maintains the host's load in real time. Each server has a static IP address connected to the OpenFlow switch, and each pool has a virtual IP with virtual MAC-address. All users send their requests to virtual MAC-address, without knowing the physical address of the host, the OpenFlow switch.

The switch checks its flow table once a request arrives to find a matching entry. If the client packet header is matched, the switch carries out the actions in the flow entry. If there are no flow entries matches, the switch sendsPacketIn message to the controller that executes the modules. Then, the controller inserts the corresponding flow entry to the switch through OpenFlow protocol. In the following section, the building blocks of the system are discussed in details.

4.3 The building blocks of the proposed load balance mechanism.

In this section, the system building block that includes three modules namely service classification module, dynamic load balance module and monitoring module are discussed in depth. Load balance is the main module that receives the type of request from the service classification module and gets the current server load from the monitoring module.

4.3.1 Service Classification Module(SCM)

This section aims to discuss the implementation of service classification module. To support real-time traffic classification in Floodlight controller in order to identify the request type, we developed an online service classification module based on statistical information that is collected from both hosts and clients during the communication. In this section, we will discuss the following:

- The method of identifying the type of request.
- *MemoryStorageSource* service.

4.3.1.1 The method of identifying the type of Request

Various approaches are proposed for service classification. In our module, we adopted the approach that is implemented in a traditional network using Network Packet Description Language (NetPDL). For using this approach, OpenFlow protocol is leveraged by adding an additional function that can classify the flows. Service classification relies on observation of how hosts and clients communicate with each

other in the network. In the case of a client-server network, typically, a large number of clients connect to a single host or multiple hosts that provide the same service. Thus, we identify the host or the VIP of the load balance system as the main actor. The aim of the service classification is to identify which service is offered at an IP address, port, and protocol. Consequently, a classifier can infer that all future sessions that contain these three factors will be directed toward the host that provides such service. As we see inFigure4.4, when client A establishes a session and starts sending a request to VIP, the service classification module will first extract the packet information to get the value of the factor and save it into the Service Table (ST) that is stored in *MemoryStorageSource* service. When a new request is sent to VIP, the classification module first checks the ST. If the three values are matched, the service name is sent to the load balance module to adjust the parameters according to the request type.

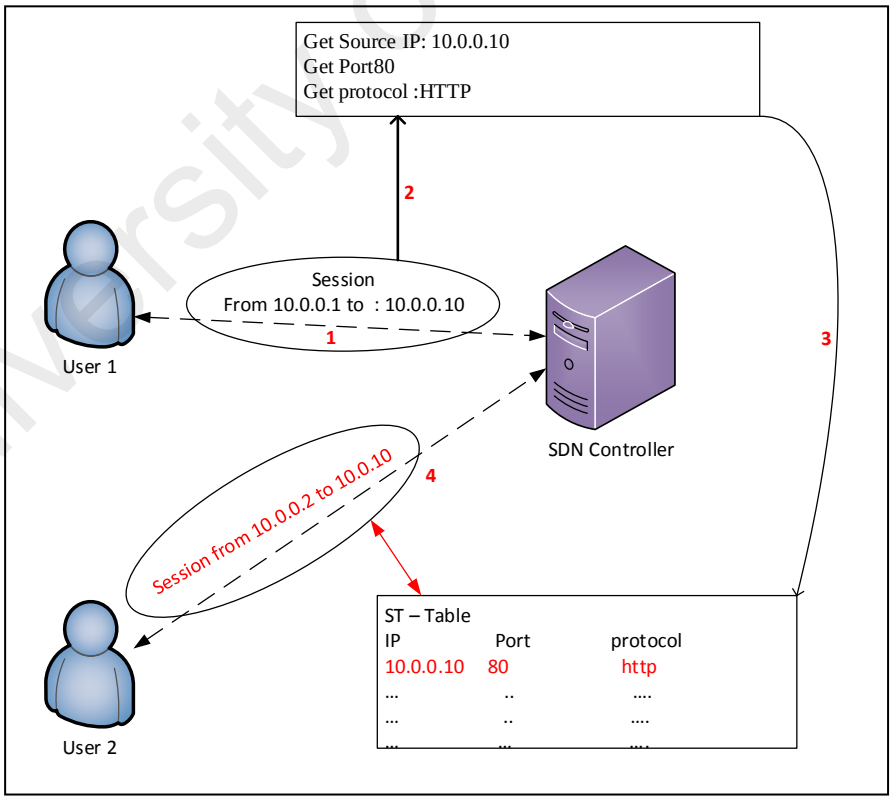

Figure 4.3 Service classification process

For example, if the classifier module recognizes that server 1 with IP address (192.168.5.2) that runs a web service on TCP port 8080, it can classify all sessions established to this IP address, port, and protocol as HTTP services. Our classification approach is different from the port-based approach that focuses on the port number. This approach extracts the two values; IP address and port from the header of the field while the protocol is extracted from the payload of the packet. The process of storing and retrieving values from the ST will be discussed in the next section.

4.3.1.2 MemoryStorageSource service

Floodlight controller provides *MemoryStorageSource* service that allows storing some information about the network temporarily. It is the memory of NoSQL storage utilized for storing and retrieving information of the PacketIn for service classification purpose. To use this service when the controller starts, we configure it in Floodlight configuration file. In addition, we import two other dependencies, I*DebugCounterService* and *IRestApiService* that this *MemoryStorageSource* service uses. To create ST, we call *IStorageSourceService* interface in our SBLB module to be able to create, delete and modify data in the memory storage source. For retrieving data from ST, we implement an*IStorageSourceListener* interface that allows sharing all data in other modules.Initially, we configure *MemoryStorageSource* service to run when the first PacketIn is sent to the controller. Floodlight Controller will check if the destination IP is VIP of the Load balance module, then we parse the PacketIn and get the three parameters (Source IP, Port number, and Protocol). Then, the PacketIn will be classified through the service-based method discussed in the previous section. After that, any incoming packet with this "known service" can subsequently be classified directly into the information stored in the ST as described above without any further processing (e.g., payload inspection). Since Service Table ST is stored in memory that has limited space, we introduced *Service Idle Timeout* SIT to remove the entry of ST that does not match

for certain time. Such SIT will reduce the number of service entries in the ST and speed the processing time of classifying incoming packets.

4.3.2 Dynamic Load-balancing Module (DLM)

This section aims to explain the implementationof the dynamic load balancing module that calculates and distributes the incoming traffic to the server with the consideration of service type. A load of each host is calculated based on the three parameters; CPUutilization, Memory utilization and Bandwidth utilization. By considering the type of service, we divide the request into two types *compute request* (CR) and *data request* (DR) and then adjust the above parameters accordingly. CR refers to a request that does not need many network resources such as *Bandwidth*. For example, simple HTTP request (static) that does not includeserver-side scripting (e.g. PHP, ASP, and JSP) may consume less bandwidth. Conversely, DR is a request that relies more on bandwidth. To identify the type of request, the PacketIn is captured by *Service* C*lassification Module* in which the packet is classified and sent back to this module. Therefore, in this module two tasks are implemented; calculating the load of each host and selecting the best server to handle the incoming request. Finally, we integrate these two tasks into our proposed mechanism.

4.3.2.1 Calculating the load of hosts

This section we explained the calculation of a load of each host according to theservice type. We assume that the clients send various requests to the pools that provide different services. Inside each pool, we have different hosts with various workloads. The complete set of SBLB parameters that are used in this section is listed in Table 4.1.

Table 4.1SBLB Symbols parameters

Symbols	Description

P	Pools
H	Hosts
R	Request
HL	Host Load
HC	Host Capacity
CPU_r	CPU ratio
MEM_r	Memory ratio
$Band_r$	Bandwidth ratio
α	CPU weight
β	Bandwidth weight
μ	Memory weight

It is assumed that we have a set of Pools, P= (P_1, P_2 .....P_N), that are associated with the set of services, S= (S_1, S_2....S_N). In each pool, we have a number of host members with a different load, HL = (HL_1, HL_2 …… HL_N). Clients can send a request to VIP Pool. Let's say that R is a set of requests need to be scheduled: R= (R_1, R_2…….RN). First, we can calculate the load of the pool in equation 4.1

$$PLi = \sum_{i=0}^{n} \mathrm{HL}i \qquad (4.1)$$

We can simply get the load of the each host member in the pool as equation 4.2:

$$\mathrm{HL_i} = CPU_r + MEM_r + Band_r (4.2)$$

Since we have heterogeneous hosts, thus, we consider the capacity of each host, so the following equation 4.3 is used

$$\mathrm{HC_i} = CPU_r + MEM_r + Band_r (4.3)$$

To calculate the CPU proportion of the each host, we utilized */proc* filesystem that is provided by Linux kernels. A simple java class called*CpuRamRatio* is devoted to getting the CPU and memory utilization and report it to the Floodlight Controller every five seconds. The formula to get the CPU ratio from */proc file is* 100.0 * (1.0 - *IdleTime* / *TotalTime*). The ratio of the memory is calculated based on this formula 100.0 * (*TotalMemoryUse* + *MemorySwap/ TotalMemory*). In terms of the bandwidth, we utilized an OpenFlow switch to reporting the bandwidth of each link, this function is explained in section 4.3.3.2.

To calculate the load of the hots, we proposed α, β and μ values. This value is multiplied with each load balance factors to adjust the parameters based on the type of the request. Meanwhile, different types of requests are available (*N* types). In this research, we divided the users request into two type ComputeRequests CR, and Data Requests DR. Thus, we can define the different weight values for each type of requests. In order to calculate the load of the each host based on the different type of request, we used the following equation 4.4.

$$HL_i = \alpha * CPU_r + \mu * MEM_r + \beta * Band_r \quad (\alpha, \beta, \mu) <= 1 \quad (4.4)$$

In which the total of the three parameters must be less than 1. Different types of therequest will have different weight values for CPU, memory, and bandwidth. For example, in thecase of the compute request the load of the hosts are calculated as follow:

$$HL_i = \alpha * CPU_r + \mu * MEM_r + \beta * Band_r \quad (\alpha = \mu, \alpha > \beta \text{ and } \alpha + \beta + \mu = 1) (4.5)$$

In the above equation, the value of the bandwidth is less because this type of the request does not need much bandwidth. However, in case of the request is the data request, the load of the hosts is calculate based on equation 4.6

$$HL_i = \alpha * CPU_r + \mu * MEM_r + \beta * Band_r \quad (\alpha = \mu, \mu < \beta \text{ and } \alpha + \beta + \mu = 1) \quad (4.6)$$

To achieve the load balancing over the proposed mechanism, we proposed a new load balancing algorithm that considers both real-time host's load and type of the request.

4.3.2.2 SBLB Algorithm

This section explains dynamic SBLB algorithm that calculates hostsload according to theservice type. Figure 4.5 presents the pseudo-code for the proposed load balance mechanism. The input consists of multiple requests from different clients. When the controller has received the requests, the requests will be classified into two different classes namely, compute request and data request based on the type of request. Then, the controller selects the host with minimum load to process the client request

depending on the type of the request. This information is updated each time the loads on the servers change.

Algorithm 1 : Service-Based Load Blanca Algorithm (SLBA)

Input: Set of the incoming request (R_1,R_2……..$R_{N)}$
Set of the Host (H_1,H_2,……H_N)
Current host status (CPU, RAM, BAN)
Output: Select best Host to handle incoming request (H*i*)
W*i*←(α, β, μ)
Foreach 1 in *ST.R***Then**
  If R*i*= ComputeRequest **then**
    G*etResourceUtilization*(CPU, RAM, BAN)
    H*i.cpu*←CPU
    H*i.ram*←RAM
    H*i.band*←Band
L(H*i*)= $\alpha *CPU_r + \beta *RAM_r + \mu *Band_r$ ($\alpha = \mu$, $\alpha > \beta$ and $\alpha + \beta + \mu = 1$)
    H(C) ; *// *calculate the capacity of the Host*
    P(H*i*) ; *// *Probability of the task for the host*
Elseif R*i*= DataRequest **then**
L(H*i*) = $\alpha *CPUr + \beta *RAMr + \mu *Bandr$ ($\alpha = \mu$, $\mu < \beta$ and $\alpha + \beta + \mu = 1$)
    H(C) ; *// *calculate the capacity of the Host*
    P(H*i*) ; *// *Probability of the task for the host*
Endif
End

Figure 4.4Pseudo-code of the SBLB algorithm

4.3.2.3 Selecting the best server

This section explained how to select a host to response for user request after the calculation of a load of thepool. If we always choose the host with the smallest load for request distribution, all the requests may be assigned to the same host in a short time, the host's load will increase quickly, and then requests will be assigned to the host with the second-smallest load.

(4.7) $$P(S_k) = \frac{C(S_k)}{\text{sum}\, C(S_i)}.$$

The cluster system will generate jitter. Therefore, we use the random probability distribution method. When distributing requests, we firstly choose those hosts with the smaller load as candidate hosts constituting the hosts set for distribution, then, we

allocate the requests according to the probabilities of hosts in the set, ensuring that the distribution of requests is uniform and the jitter is avoided.

4.3.3 Monitoring Module (MM)

In this section, we discussed the statistics collection method that used for monitoring hosts and links. We devoted monitoring module that upsized APIs of the controller andruns on the top of the Floodlight controller.

For the purpose of obtaining the network and host status, we gather two types of data from the network nodes; 1) resource utilization of the hosts that include server information such as CPU utilization and memory utilization. 2) Link bandwidth statistics that are reported by switches periodically. This information is collected by controller every five seconds. In this section, we explain in details how we implement these tasks.

4.3.3.1 Statistics collection service

To carry out these tasks, we implement statistics collector services in which all information are gathered. Figure 4.6 shows two functions that collect the values of resources and bandwidth of the links and calculate the real state of each node in the network.

Algorithm 2 Statistics Collection Services

Input 1 : M a set of all Pool Members
Input 2 : CPU, Mem: Values of M CPU and Memory utilization
Input 3 : DatapathId , OFPortNo : all switches with port numbers

```
1: function RESOUCESUTILIZATION(M,CPU,MEM)
2:     COLLECT CPU (M)
3:     COLLECT Mem (M)
4:     /*Notify listeners of update()*/
5:
6:     while r ≠ 0 do                    ▷ We have the answer if r is 0
7:         a ← b
8:         b ← r
9:         r ← a mod b
10:    end while
        NOTIFY()
11: end function
12: function BANDWIDTHUTILIZATION(M,DATAPATHID,OFPORTNO)
13:    count RX (DatapathId,OFPortNo)
14:    count TX (DatapathId,OFPortNo)
15: end function
```

Figure 4.5Statistics collector algorithm

4.3.3.2 Bandwidth Statics information

OpenFlow specification provides many statistics messages that allow the controller to query the switch for information such as flow stats, meter stats, queue stats, aggregate stats, table stats, and port stats. The ability to collect this information is great, but it must be reported at the real time. For example, when the controller receives a status reply message, the values within the message are most likely out of date and do not reflect the real-time state of the switch anymore. To avoid this, we utilize collect switch statistics service in Floodlight to gather bandwidth values every five seconds. But the problem is that OpenFlow switch provides raw byte counters without providing the times of these counters. Therefore, we collect and modify some statistics that depend on time such as bandwidth. A bandwidth is a number of packets that can travel through a link for a given amount of time. Therefore, it refers to Byte per the second

amount. The following equation is used to calculate the bandwidth of links each 5 seconds

$$Bw = C/T \quad (4.8)$$

Where C is the size of counter and T is time. Since time is not given, we count the byte counters returned at two points in time $C_i = T_1 <----> T$. The difference between these two counters divided by the time passed between the points of each counter value returns the bandwidth

For calculating the bandwidth, there are two methods:

- Compute the bandwidth frequently to ensure that we have real-time bandwidth consumption.
- Less frequent update to avoid network overhead.

The first approach produces more errors due to computing timestamps, but it is accurate than the second approach, while the second approach is not accurate but it does not cause network overhead. To avoid these two issues, we adopted 5s as a rate to collect links information to avoid causing overhead for the Floodlight controller

The statistics module in Floodlight is not enabled by default. This means we must enable it in the Floodlight default properties file to make it start when the controller runs. Figure 4.7 shows the commands that enable statistics module with an interval of 5s.

```
net.floodlightcontroller.statistics.StatisticsCollector.enable=True
net.floodlightcontroller.statistics.StatisticsCollector.collectionIntervalPortStatsSeconds=5
```

Figure 4.6Enable the statistical function

To integrate this module in our SBLB mechanism, we implement *Thread* that handles the statistics request and response.The Thread will be initialized via

IThreadPoolService, Floodlight service that is provided by another module. Three parameters will be collected *portStatsInterval_1, portStatsInterval_2, TimeUnit* per second. *Statics collation service* implements a *Runnable class* that contains the run function, which will be invoked by the executor at the times and interval set as described above. To get Bandwidth statistics counter of each link, we pass two parameters the *DatapathId* and *Pot number that* return the *rxBytesCounted* and *txBytesCounted.*

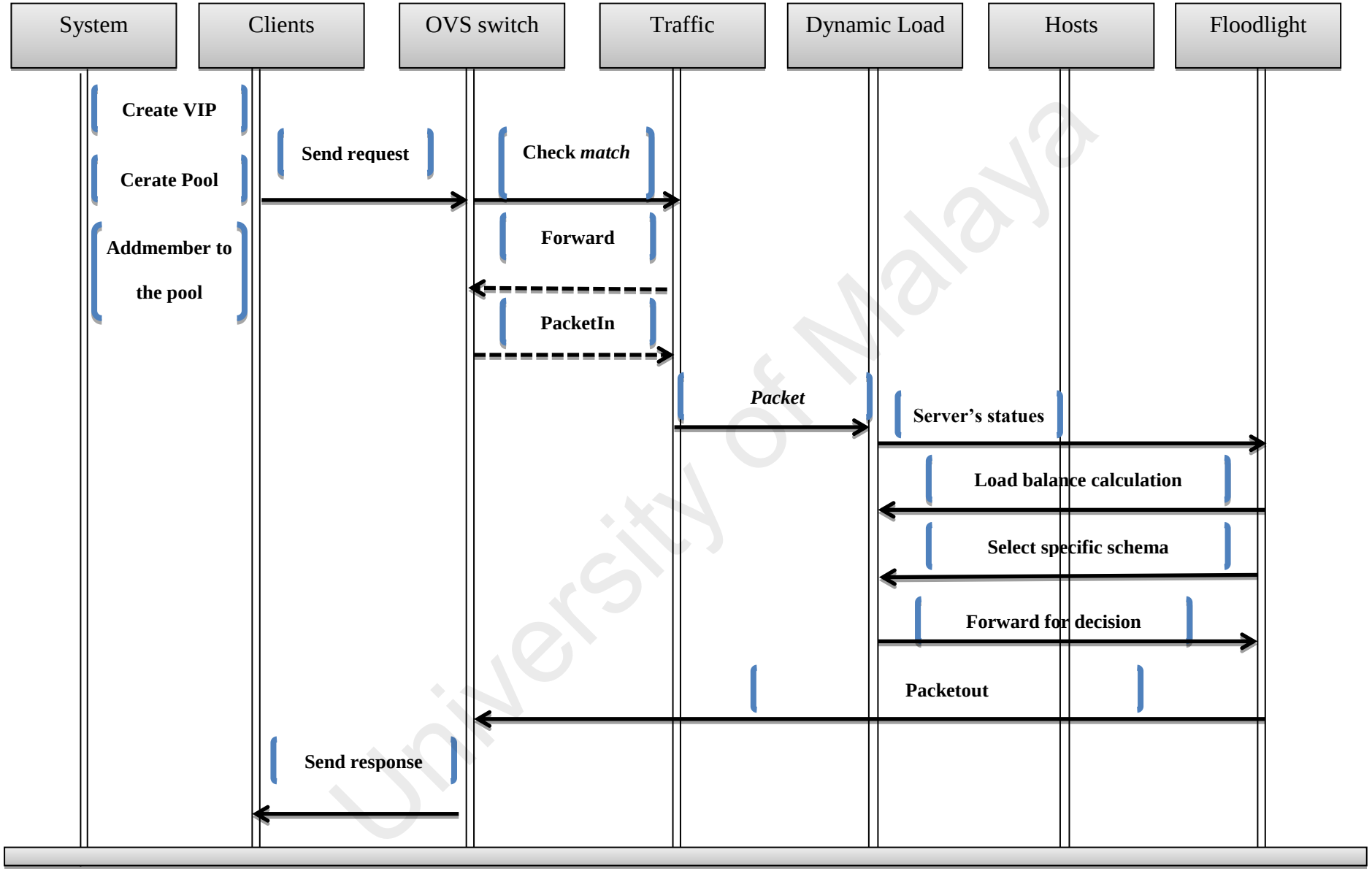

Figure 4.7 System flow sequence

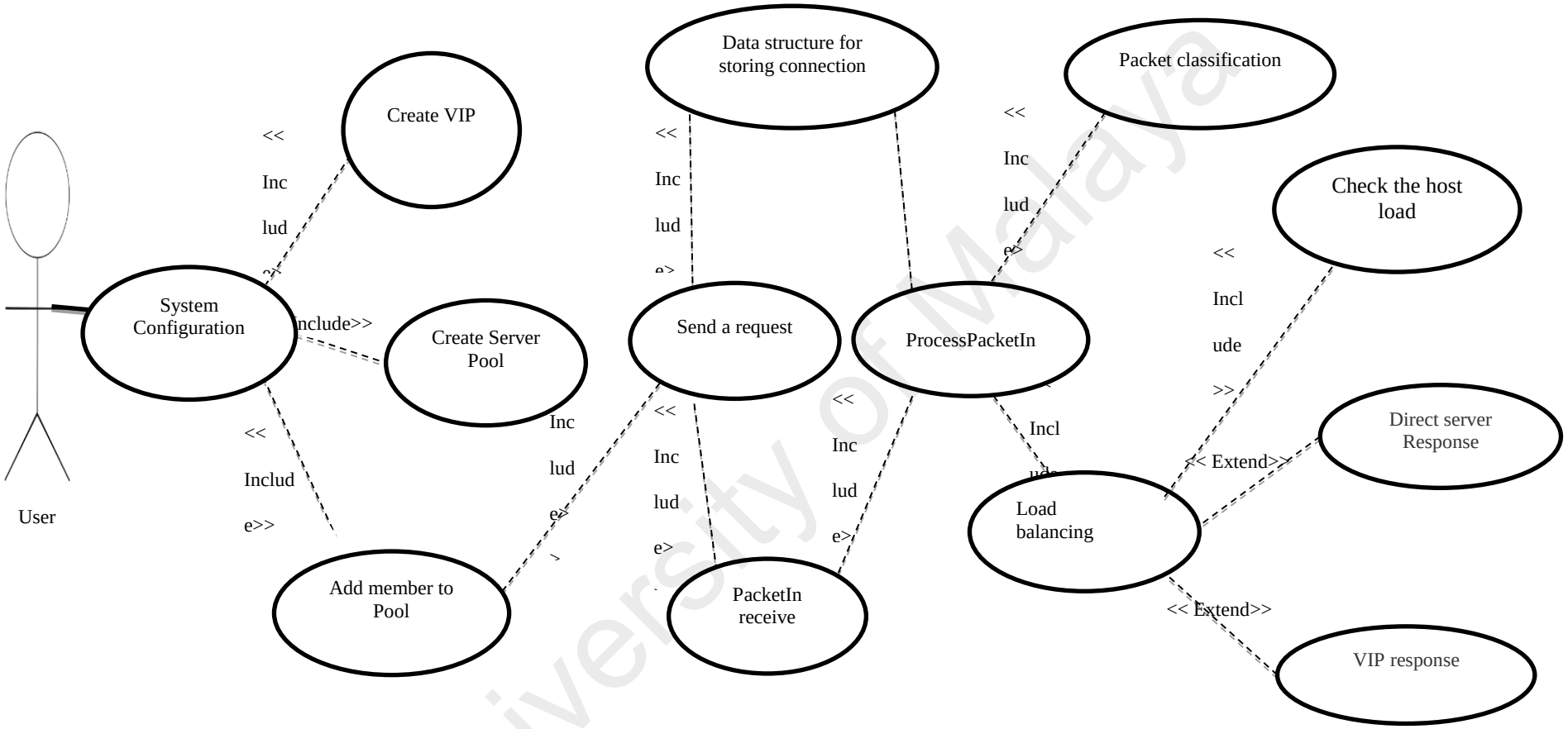

Figure 4.8 Use-Case diagram

4.3.4 Use-case and flow-sequence diagram

This section explained the use-case diagram that shows the interaction of the user with SBLB as well as the flow sequence diagram in which all SBLB steps are explained in details. Figure 4.8 shows the use case diagram for SBLB mechanism. The use case diagram is used to identify the interactions between the proposed load balance mechanism and the users. In the use case diagram, the use cases or processes are drawn as an oval shape whereas the actors or the users are represented as a stick figure. The compulsory procedure is shown using the <<*Include*>> relationship whereas the <<*extend*>> relationship indicates the optional procedure for the SBLB processes.

4.3.4.1 System configuration Process

In the beginning, the user must configure the server pools and added VIP for each pool as well as define the types of traffic such as TCP or UDP. The configuration depends on a number of the server pool and the number of members added to each pool. In our proposed system, we have 4 server pools; two is TCP and two for UDP traffic. Figure 4.6 show the script of CURL command that adds pools and sends it as JSON

message to Floodlight controller.

Figure 4.9Send JSON message to Floodlight

4.3.4.2 Create host pool

Floodlight Controller provides*IRestApiService* services that handle the mid-level details of the configuring Restful API. In the first line of the script, we created TCPVIP1 for TCP traffic associated with L3 VIP, the MAC Address of this VIP is created in *LBVip* class inside the module. We must define the port which is port number

8, a unique port for each VIP based on the Floodlight Rest API configuration, and then define the URI that includes controller IP.The server pool must implement with a unique name, to associate this pool with VIP, the ID of TCPVIP1 is added with the same type of traffic. Also, the URI of the *pool* is addedwith the physical IP of the controller.

4.3.4.3 Added member to Pool

Clustered servers are the member of each pool, for example, HTTP servers are added to TCPVIP1 pool while FTP servers are added to TCPVIP2 pool. The members will be added by their L3 IP address and associated with a same port of VIP. The user's request sends to VIP; then classification module will identify the request type. Based on the request type, parameters will be adjusted, and the best host will be selected to handle this request.

4.3.4.4 Send request

We set the default action in the switch is (*Controller*) so that the first traffic is sent to the controller. The controller will check the type of messageand ignore all messages except *PacketIn* message. Then, we will send the message to *ProcessPacketIn*function with the same parameters. When the PacketIn message received by the process function, it will extract the message using *IFloodlightProviderServic*e and get the payload which includes all header information of the packet. Since the VIP only consists of MAC address and is not known to another computer in the network, we need to advertise the MAC address with VIP by implemented *pingall* command. When we *pingall*,thePacketIn will send to the controller thattrigger the *Forwarding* moduleto send all MAC address to all port in the switch for recognizing each other. In the process PacketIn function, ARP message will be checked, and *vipProxyArpReply* procedure will be implemented to send MAC address associated with VIP to all network devices. Therefore, clients have the ability to

communicate with VIP. For normal IPv4 traffic, the destination IP address and the protocol type such as TCP and UDP are identified.

4.3.4.5 Data structure for storing

To get the information of the user who sends the request and identifiesthe request send to which VIP, we implemented Class named *IPClient* that define; *IPv4Address*,*IpProtocol*, *TransportPort*of the source (client) as well as *TransportPort* of the host (destination). This parameter also helps to maintain the session between the host and the client as well as used for classifying the request

4.3.4.6 PacketIn receive

utilizePacketIn receive function; we extend the *IOFMessageListener* interface in our module; Itincludes three parameters; *IOFSwitch, OFMessage, FloodlightContext.*This function will check the type of message, in our module, we will ignore all messages except *PacketIn*message. Then, we will send the message to *ProcessPacketIn*function with same parameters.

4.3.4.7 Process PacketIn

When the PacketIn message comes to process function, it will extract the message using *IFloodlightProviderServic*e and get the data which includes all header information of the packet. Frist, in process PacketIn function, ARP message will be checked, and *vipProxyArpReply* procedure will be implemented to send MAC address associated with VIP to all network devices. Therefore, clients have the ability to communicate with VIP. For normal IPv4 traffic, first, will check the destination IP address and identify the protocol type such as TCP and UDP.

4.3.4.8 Packet classification

For classifying the packet, the TCP/UDP packet header must be analyzed as we mentioned early using statistical information that collects from both host and clients during the communication. In this process, the controller extracts the three values; IP

address, ports, and protocols from the header of the field. To reduce the transmission overhead and increase the performance of controller during checking of the PacketIn, we will check only the three values mentioned above. To store information of the packet and used for incoming packet we used *MemoryStorageSource* that can store all packet information and retrieved by the controller to select appropriate parameters for load balancing.

4.3.4.9 Load balancing

In this process, three functions are implemented namely; schema selection, host load calculation, and decision making. In the first process, after received the type of the traffic from the classification module, the schema of the data or compute request is selected. The load balance module will calculate the load of the each host based on equation 4.5 or 4.6 that are used based on the type of the traffic. After that, the best host to handle the request is selected based on question 4.7.

4.3.4.10 Check the host load

Our proposed system implemented the dynamic load balance, whereas the hosts report their load ratio such as CPU, RAM, and the controller collects the link bandwidth that is reported by the switches periodically. This information is sent to the controller every 5 seconds to avoid the controller be overheated. In addition, 5 second is default timeout that used in Floodlight controller. This means automatically after 5 seconds flow entry will be removed.

4.3.4.11 Direct host response and VIP response

These two processes are optional, where we can use one of them to response to the clients. In the first option, the hostcan senda response to the client directly by showing the IP address of the host in the *PacketOut* message. This can decrease the response time. However, it is not secure and is not commonly used in load balance system.In the second option, the VIP response process where the Floodlight controller

will change the distention IP of the *PacketOut to the VIP address*, so the client does not know which host response to.

4.3.5 Conclusion

In this chapter, we propose Service Based Load Balance SBLB mechanism that takes into account the type of the service. SBLB aims to minimize the response time and maximize the throughput. The development process of the module using Floodlight controller is discussed. Then, the system architecture is presented to illustrate the characteristics of the proposed mechanism. The SBLB system architecture consists of the three main modules is discussed in details. We explained the load balance module that dynamically adjusts the load balance parameters based on service type. The operation of the classification module that responsible for classifying the incoming request using Service Table ST is discussed. The function of the monitoring module that is reported the links status and the current host's load is described. In addition, the SBLB algorithm is presented as well to show how it is dynamically select the hosts based on the type of the services. Lastly, the use-case and flow-sequence diagrams are illustrated to show the interaction between client and the proposed load balance mechanism.

CHAPTER 5: **EVALUATION**

This chapter discusses the evaluation of the proposed solution. A data collection technique is used to analyze the results of the SBLB mechanism. A statistical model is implemented to evaluate the accuracy of the data. In addition, the chapter presents the experiment setup and tools used to evaluate the performance of the SBLB. Three metrics namely Average Response Time (ART), Reply Time (RT), and Request Per Second (RPS) are used for the SBLB evaluation.

The chapter begins with the evaluation of the results for real and simulation environments. Then, the data and compute request in homogeneous and heterogeneous environments are analyzed. Thisis followed by the results of SBLB and HAproxy load balancer software. After that, a performance evaluation of SBLB and Round-Robin in homogeneous and heterogeneous environments is presented.

This chapter is organized as follows: Section 5.1 describes performance evaluation that includes the experimental setup and components of the experiments. Section 5.2 presents the data collection method. Section 5.3 discusses the statistical model that is implemented for evaluating the results. Section 5.4 discusses the performance analysis of SBLB. This section consists of four subsections. Section 5.4.1 shows the comparison of the results of Mininet with OpenStack. Section 5.4.2 presents data collection for analyzingcompute, and data request in homogeneous and heterogeneous environments. Section 5.4.3 shows the comparison in between SBLB and HAproxy load balance in the homogeneous and heterogeneous environments. In section 5.4.4, we discuss the evaluation of the SBLB and round-robin algorithm in homogeneous and heterogeneous environments. Finally, Section 5.5 concludes the chapter.

5.1 Performance Evaluation

In this section, we explain the experimental setup and the components that are used for the performance evaluation. In addition, we highlight the data collection method and statistical tools that are used to verify the correctness of the data.

SBLB is a dynamic load balance mechanism that uses to adjust the load according to services types in a different environment. The proposed mechanism are evaluatedin ahomogeneous and heterogeneous environment with adifferent type of requests. In ahomogeneousenvironment, all hosts have thesame specification in term of CPU, RAM, and bandwidth while in aheterogeneousenvironment, these specifications are varied.The CPU, RAM and bandwidth ratio are used to calculate the host load in each request.

Table 5.1 shows the configuration of hosts in the experiments for thehomogeneous and heterogeneous environment. Unlike the experiments in chapter three that are carried out in asimulation system, Mininet, for analyzing the problem, these experiments are implemented in acloud environment using OpenStack. Due to the limit of the resources, we implemented the experiments with thespecification that illustrates in table 5.1.

Table 5.1 Specification of hosts in OpenStack environment

Specification	homogeneous	heterogeneous
CPU	3.40.0 GHz	2.60GHz -3.40 GHz
RAM	512MB	256MB - 512MB
Bandwidth	100Mbps	50Mbps – 100Mbps

To evaluate thedifferent type of services in SBLB, we used enhanced HHTperf tool that can generate HTTP, FTP, and video streaming traffic. We assumed that HHTP is a compute request that relativelygenerates a small data with high intensity. In addition, we assumed that FTP and video streaming services are data request,a larger amount of data that requires some computation work.

5.1.1 Experimental Setup

We implement SBLB mechanism in the cloud environment that is installed and configured on a single desktop computer where the OpenStack is implemented. Multiple VMs were implemented in one physical machine to represent the cloud environment. We have deployed OpenStack using *devStack* script. The specifications and the software versions of the computer are listed in Table 5.1.

Table 5.2 Systems specification of the computer

Software and Hardware	**Specifications**
Processor	Core i7
RAM	12 GB
Operation System	Ubuntu 14.04
SDN controller	Floodlight 1.2
OpenStack	Grizzly
Open VSwitch	Version 2.9 support OF 1.3

5.1.2 System Topology

Figure 5.1 shows the network topology that configures in an OpenStack cloud environment. Five host pool are created with three different hosting pool. Each pool includes five hosts. Floodlight controller is installed in remote VM connected to OVS inside OpenStack. Host A and B are used as a client to generate traffic using enhanced HTTPerf tool. The tool can generate three types of the traffic namely HTTP, FTP, and video streaming.

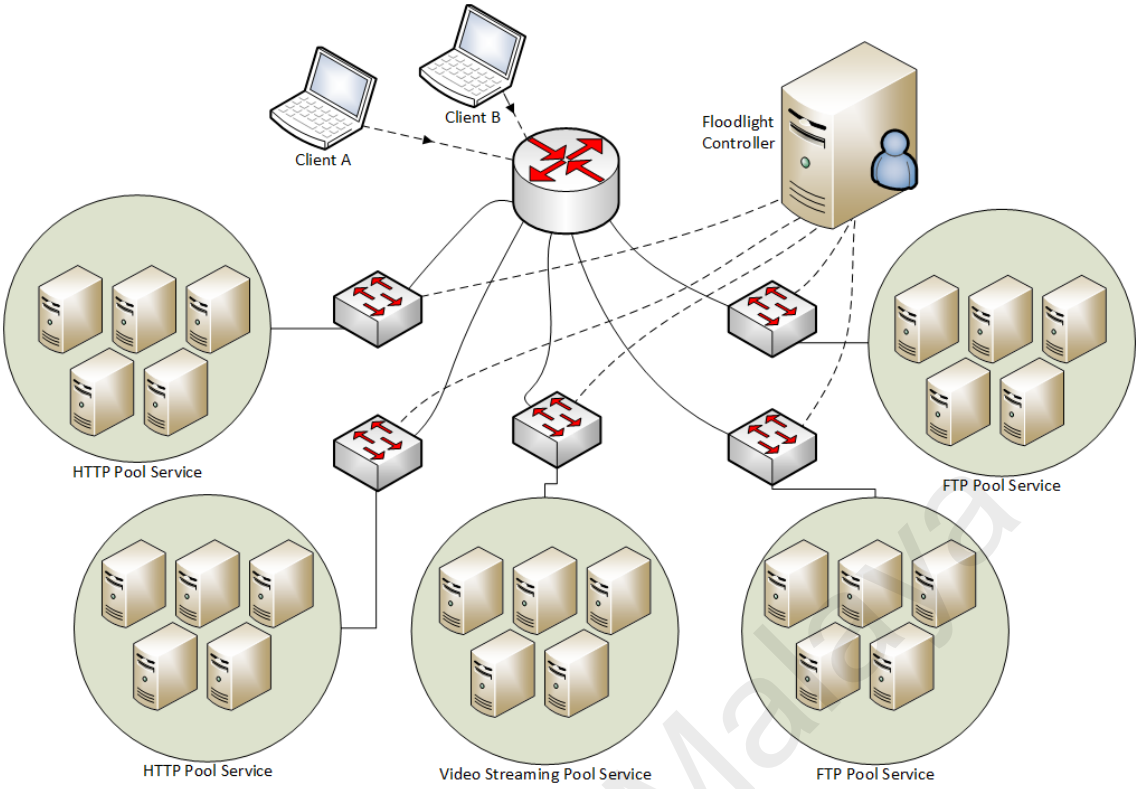

Figure 5.1 System topology

5.1.3 The components of the experiments

The components of the experiments are summarized below:

5.1.3.1 Floodlight Controller

In this experiment, we used Floodlight controller version 1.2 that supports OF 1.3. The controller is open source written in Java language and supports FULL-REST API that uses HTTP requests to program the controller. In appendix 2, the code of the FULL-REST API to configure hosts pool of the controller is presented

Two options are available to develop an application in this controller. The first option is writing a java code in *(/floodlight/src/main/*resources) directory and configure the modules to start when the controller is running by adding the modules in *floodlightproperties* file. The second option is using another language such as python or simple script by utilizing the REST API. In this study, we develop SBLB modules using java and implement the proposed algorithm. Also, we have written some scripts using CUR

L and REST API to configure the server pools and added the members to that pool with a specific type of traffic. For example, we added five server pools, and each one can handle a specific type of services such as HTTP, FTP, Video stream service.

5.1.3.2 Open VSwitch

OVS is an open source software layer switch that supports OF and OVSDB management protocol. It is a virtual switch that can be placed in physical server or VM. In this experiment, we configured OVS to manage the hosts inside OpenStack by configuring plug-in. All hosts and the controller are connected to OVS interfaces to direct traffic among hosts. The range of IP addresses is 192.168.0.1 - 192.168.0.30.

5.1.3.3 OpenStack:

We implement OpenStack Grizzly that was recommended by Floodlight project in our experiment with Neutron v2.0.

5.1.3.4 SBLB Application modules and performance metrics

We install Floodlight controller in separate VM using Oracle VirtualBox to make the controller connect to OpenStack. We configure the following steps:

- Configure DevStack script with *Neutron Plugin* to install OpenStack Grizzly
- Configure the Floodlight resource file that is located in (src/main/resources/neutron.properties)
- Install OpenVswitch and configure it for running on each of the nova-compute nodes.

5.2 Data collection method

The results of the experiments are obtained via testing the SBLB mechanism in both simulation environments that are implemented in Mininet and in a real environment that is carried out in OpenStack. Moreover, we compared between different type of requests by dividing the requests into two types; compute request and data request to study the

impact of user's request on our proposed mechanism. In addition, we conducted another experiment in a conventional network using HAproxy, load balancer software, for the comparison purpose.

The above-mentioned experiments are carried out in two different scenarios. In the first scenario, we used homogeneous server (hosts with the same specification) while in the second scenario we implemented on theheterogeneous server (hosts with different specification and various link capacity). These scenarios aim to test and evaluate the proposed mechanism in different environments.

5.3 Statistical model

As mentioned earlier in the previous section, the results are obtained by conducting several experiments in different scenarios. The data is collected for all the benchmark tools that are used in this study in 15 experiments. Each experiment is tested 15 times to evaluate the metrics based on sample statistic.

In the data sample, the measurement of the central tendency is calculated based on the sample mean (-X) that can achieve a better point estimate of the population compared to median or mode. A sample includes a range of intervals determined by the specified confidence level, a statistic, and a margin of error. The level of confidence is the probability that the metric is truly captured by the confidence range. The common Confidence Levels (CL) are 90%, 95%, and 99%. According to the sample central limit theorem, the sample size that is less than 30 (n>= 30), then approximately 95% of the sample means within 1.96 standard deviations should be used. To calculate the margin of error in the sample, we used equation 5.1 below:

$$M = Z * (\frac{\partial}{\sqrt{n}}) 5.1$$

Where M is the margin of error, and Z is value based on a percentage of the confidence interval and ∂ is standard deviation, and N is the number of samples. The confidence

interval estimates for each sample mean (X) of the primary data are calculated with a 95% confidence interval using the following equation.

$$\mu = \overline{X} \pm Z\left(\frac{\partial}{\sqrt{n}}\right) 5.2$$

We also performed paired sample T-test to ensure that there is a significant difference between the results when we compare SBLB mechanism with existing solutions. The question 5.3 is used to calculate the T value and P value.

$$t = \frac{(\sum D)/N}{\sqrt{\frac{\sum D^2 - (\frac{(\sum D)^2}{N})}{(N-1)(N)}}} \quad 5.3$$

$$r = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_i (x_i - \overline{x})^2}\sqrt{\sum_i (y_i - \overline{y})^2}} \quad 5.4$$

Correlation analysis between the simulation and real environment results are also calculated based on equation 5.4. The following section presents the data collected during different experiments for the evaluation of the SLBM as applied to the different environments.

5.4 Performance Analysis

In this section, we discuss the performance analysis of SBLB mechanism. First, we validate the results that are obtained from the simulation and real environments. In addition, we analyze the impact of data and compute requests in the homogeneous and heterogeneous environments. Then, we compare SBLB with HAproxy software load balance in the homogeneous and heterogeneous environments. Lastly, SBLB is compared with round-robin.

5.4.1 Data Collected for SBLB mechanism that is carried out in simulation and real environments

We conducted the experiments in two different environments; simulation environment that is carried out in Mininet and real cloud environment that is implemented in theOpenStack environment. This section verifies the correctness of results obtained from these two environments during the execution of SBLB modules. The first step is to collect the simulation data; then, we compare the data of the simulation, under the same conditions with data that is generated when using real OpenStack environment. Then, we validate the results by employing statistical analysis tools for the statistical validation; in this case, we use a confidence interval and Pearson coefficient. Average response time, reply time and request per second are the metrics that are used for validating the results. The topology used in this experiment includes five pools with different types of services, each pool connected with OF switch and one switch connected to Floodlight controller. In both environments, the controller is installed on a separate computer that manages the network remotely (Remote Controller). We create a simple script that collects the results in runtime and saves them in CSV file. To prevent requests from queuing and being delayed, we increased the timeout to 15 seconds in between each sample.15 data samples are used for each experiment with an incremental request that is started by 50 requests per second up to 190 requests. In Mininet, it is noticed that when the client sends more than 190 requests, the dropped packet is increased and few packets are processed. This is due to the limitation of the buffer of sending and receiving data. Therefore, the maximum number of requests is 190 requests in our experiment

Table 5.3 Average response time of Mininet and OpenStack

Number of requests	Mininet	OpenStack
50	4.12	4.88
60	4.64	5.00

70	4.88	5.28
80	4.94	5.57
90	5.07	5.69
100	5.15	5.91
110	5.21	6.00
120	5.24	6.04
130	5.32	6.26
140	5.50	6.34
150	5.70	6.48
160	5.91	6.64
170	6.20	6.71
180	6.80	6.74
190	6.90	6.79
Mean	5.438782	6.0226376
SD	0.762228	0.6307562
Correlation Coefficient	0.913830188	
CI	0.385734	0.3192008

Table 5.2 shows the average response time in Mininet and OpenStack environment. The first column of the table represents a number of requests that are sent in each experiment, while the second and third columns show the average response time. As we can see, there is no big difference between the two results. The R value of the correlation coefficient (cc) shows 0.91 which means there is a strong positive correlation between the two data traces.

Table 55.4 Reply Time of Mininet and OpenStack

Number of requests	**Mininet**	**OpenStack**
50	2.81	2.72
60	3.26	2.87
70	3.30	2.88
80	3.35	2.89
90	3.41	3.02
100	3.45	3.58
110	3.46	3.65
120	3.60	3.81
130	3.63	3.93
140	3.65	4.12
150	3.93	4.22
160	4.41	4.23

170	4.57	4.24
180	4.62	4.58
190	4.67	4.60
Mean	3.742052	3.69385
SD	0.57073	0.67005
Correlation Coefficient	0.890022617	
CI	0.288824	0.33909

Table 5.3 illustrates the reply time through Mininet and OpenStack. In the bottom of the table, the average reply time of the Mininet and OpenStack are 3.7 and 3.6respectively. This small amount of difference validates the results obtained from both experiments. We notice that after 160 requests, Mininet environment recorded high reply time compared with OpenStack environment because of the buffer of the receiver. For example, in a real environment, each host has its own buffer that has a queue that stores the incoming requests. Conversely, in asimulation environment, this buffer cannot handle all incoming requests because of its buffer's limitation.

Table 5.5 Request per Second of Mininet and OpenStack

Number of requests	**Mininet**	**OpenStack**
1000	175.48	175.20
1000	140.69	140.80
1000	172.73	173.20
1000	159.08	160.10
1000	170.58	171.20
1000	180.01	182.20
1000	179.74	180.40
1000	173.61	173.50
1000	167.07	167.20
1000	160.31	159.20
1000	165.74	165.10
1000	131.51	130.00
1000	130.82	132.10
1000	146.04	146.20
1000	130.44	130.10
Mean	158.923	159.10
SD	18.2446	18.692

Correlation Coefficient	0.998854644	
CI	9.23286	9.4593

Table 5.4 shows the number of requests that can be handled per second in both simulation and real environment. In this experiment, we send 1000 requests in each sample trace and record the RPS. As we can see, The R value of the correlation coefficient (cc) shows0.99, which indicates strong positive correlation.

5.4.2 Data Collected to analysis data and compute request in homogeneous and heterogeneous environments

In this data collection, the comparison between the data request and compute request in a homogeneous environment is conducted. The compute request is relatively a small data with high intensity. This type of requests does not need much bandwidth and thus are characterized by fast CPU processing time. One example of this request is HTTP requests whereas HTTP GET must be returned as quickly as possible and should concern only on the capacity of the host, especially CPU. As we mentioned in chapter 4, the calculation of the host's load is implemented based on the type of the request. In the caseof the compute request, the W value of the CPU and RAM is large compared to the W value of the bandwidth.

The data request is a larger amount of data that requires some computation work. It represents real-time audio/video services or FTP file in which a big size file is to be transferred. One example of this type of the request is downloading files using protocols such as FTP. This type of request should be concerned on the bandwidth of the links. Therefore, the weightvalue is large.

In this experiment, the compute request includes simple HTTP GET while data request consists of FTP with the size of 500MB, real-time audio and video. In the first test, we send only data request in a heterogeneous environment and record the response

time, reply time and request per second. We carried out the same test in a heterogeneous environment. In turn, these two tests are applied to data request as well.

Table 5.6The average response time of the compute and data request in homogeneous environment using

Number of requests	**Data request in SBLB**	**Compute request in SBLB**
500	6.33	4.21
600	6.34	4.22
700	6.40	4.41
800	6.63	4.50
900	6.70	4.55
1000	6.85	4.55
1100	6.99	4.67
1200	7.02	4.69
1300	7.14	4.70
1400	7.25	4.86
1500	7.40	4.87
1600	7.59	4.88
1700	7.60	4.92
1800	7.69	4.96
1900	7.73	5.07
Mean	7.04	4.67
SD	0.49538	0.26262
T-test	16.4131	
CI	0.25069	0.1329
P Value	0.00001	

In Table 5.5, the average response time of the compute and data request are illustrated. The compute request shows less ART compared to a data request in all 15 samples traces. According to T test table, thecritical value of the samples is 2.08 (appendix A). The calculated t exceeds the critical value (16.4131>2.08), and so this means that the two results are significantly different.

Table 5.7Reply time of the data and compute request in homogenous environment

Number of requests	Data request in SBLB	Compute request in SBLB
500	3.11	2.03
600	3.15	2.05
700	3.27	2.23
800	3.29	2.26
900	3.30	2.30
1000	3.50	2.61
1100	3.77	2.67
1200	3.80	2.71
1300	3.85	3.01
1400	3.91	3.26
1500	3.92	3.27
1600	3.92	3.33
1700	4.03	3.40
1800	4.05	3.51
1900	4.06	3.60
Mean	3.66	2.82
SD	0.35119	0.5564
T-test	4.9794	
CI	0.17773	0.28157
P Value	0.000029	

The reply time (RT) of the data and compute request are presented in Table 5.6. The Mean shows 25.9% difference between data and computes request. The RT of the data request ranges between 3.11 to 4.06 second for 500 and 1900 request respectively. The statistical analysis shows the significant difference, whereas T value is (4.9794>2.064) and P value shows 0.000029 < 0.05

Table 5.8Request per second of the data and compute request in homogenous environment

Number of requests	Data request in SBLB	Compute request in SBLB
1000	148.3	168.8
1000	110.9	129.1
1000	146.6	166.1
1000	116.5	136.5
1000	142.4	161.9
1000	133.8	152.9
1000	107.9	128.5
1000	126.9	146.8
1000	131.6	150.7
1000	134.8	154.9
1000	155.6	174.7
1000	138.1	157.4
1000	151.6	170.6
1000	156.0	174.9
1000	152.9	173.2
Mean	136.9	156.5
SD	15.90	15.80
T-test	3.3804	
CI	8.02119	7.9957
P Value	0.002148	

The request per second (RPS) for both data and compute requests are presented in Table 5.7. T value shows 3.3804 which is a value less than the critical value (2.048). This means that there are significant differences between the two values. In addition, P value is 0.002148 < 0.05.

Table 5.8 and 5.9 present the average response time and reply time of data and compute requests in aheterogeneous environment. The impact of the data request on average response time clearly appears where the mean of the 15 experiments shows 6.10 while compute request shows only 3.60. Similarly, the reply time in a heterogeneous environment of the data and compute requests shows 4.83 and 3.24 respectively. The T-Testshows a significance difference between data and compute requests in heterogeneous environment

Table 5.9The average response time of the compute and data requests in heterogeneous environment

Number of requests	Data request in SBLB	Compute request in SBLB
500	5.25	3.04
600	5.38	3.05
700	5.47	3.07
800	5.58	3.14
900	5.58	3.39
1000	5.96	3.51
1100	5.98	3.59
1200	6.08	3.70
1300	6.17	3.70
1400	6.18	3.71
1500	6.55	3.75
1600	6.60	3.96
1700	6.68	4.08
1800	6.70	4.09
1900	7.27	4.17
Mean	6.10	3.60
SD	0.58	0.39
T Test	13.802	
CI	0.29	0.20
P Value	0.00001	

Table 5.10The reply time of the compute and data requests in heterogeneous environment

Number of requests	Data request in SBLB	Compute request in SBLB
500	3.98	2.42
600	4.05	2.55
700	4.23	2.59
800	4.32	2.63
900	4.51	2.82
1000	4.52	2.97
1100	4.76	2.97
1200	4.80	3.20
1300	4.95	3.27
1400	4.96	3.41
1500	5.12	3.82
1600	5.33	3.89
1700	5.45	3.96
1800	5.69	4.01
1900	5.75	4.07
Mean	4.83	3.24
SD	0.56658	0.59107
T Test	7.5324	
CI	0.28673	0.29912
P value	0.00001	

Table 5.10 shows the request per second of the compute and data requests in a heterogeneous environment. The average RPS of the data request is 158.3 while compute request is 134.0. The difference between them is almost 24 request per second. The T-testproves the significant difference of the data and compute requests in a heterogeneous environment.

Table 5.11Request per second of the compute and data requests in theheterogeneous environment.

Number of requests	Data request in SBLB	Compute request in SBLB
1000	136.5	161.3
1000	124.4	149.6
1000	152.0	175.2
1000	152.4	173.5
1000	113.6	139.5
1000	128.9	151.3
1000	128.2	152.8
1000	135.9	161.6
1000	132.9	154.6
1000	144.8	170.1
1000	102.4	127.6
1000	139.9	165.8
1000	151.9	175.8
1000	114.7	139.3
1000	151.6	176.5
Mean	134.0	158.3
SD	15.6	15.11
T-test	4.3369	
CI	7.87395	7.64413
P value	0.000169	

5.4.3 Data Collection for the comparison of SBLB mechanism and HAproxy in homogeneous and heterogeneous environments

In this section, we compare SBLB with theHAproxy load balancer. HAproxy is an open source software load balancer that is widely used to provide TCP/HTTP load balancing. In this experiment, HAproxy is configured with five pools; each pool includes five hosts. Hosts are configured to run on the OpenStack, and HAproxy is installed in a separate computer. Three metrics are used to measure the results namely; Average response time, reply time and request per second.

Table 5.12The average response time in SBLB mechanism and HAproxy load balancer in a homogeneous environment.

Number of requests	SBLB	HAproxy
500	4.88	5.95
600	5.00	6.06
700	5.28	6.10
800	5.57	6.11
900	5.69	6.22
1000	5.91	6.22
1100	6.00	6.23
1200	6.04	6.33
1300	6.26	6.37
1400	6.34	6.57
1500	6.48	6.62
1600	6.64	6.73
1700	6.71	6.95
1800	6.74	7.00
1900	6.79	7.17
Mean	6.02	6.44
SD	0.63076	0.37816
T Test	2.2129	
CI	0.3192	0.19137
P Value	0.035198	

The average response time (ART) is the first indicator and provides us with an idea about the performance of the SBLB when we compare it with other solutions. Thus, we start with the presentation of the data for the ART in table 5.11. In the benchmark, the values of other parameters are set as follows; the connection rate is set to 100 per second, and the session is configured to be 10 sessions that represent the number of concurrent users. The interval time between each sample is set to five seconds to ensure that the responses are received by the users. The user's requests that are sent from different clients that contain 50% of compute request and 50% of thedata request. These requests include different services such as HTTP, FTP, and video stream. For validating the results, a T-test is used, and the P value is calculated. The T-test shows 2.2129 that indicates a significant difference between the two values, and we found that the P value is 0.00001 <0.05.

Table 5.13The Reply Time in SBLB and HAproxy in homogeneous environment

Number of requests	**SBLB**	**HAproxy**
500	2.72	3.01
600	2.87	3.20
700	2.88	3.43
800	2.89	3.43
900	3.02	3.45
1000	3.58	3.52
1100	3.65	3.59
1200	3.81	3.78
1300	3.93	3.79
1400	4.12	4.04
1500	4.22	4.12
1600	4.23	4.27
1700	4.24	4.35
1800	4.58	4.73
1900	4.68	4.76
Mean	3.69	3.83
SD	0.67005	0.53135
T Test	0.619	
CI	0.33909	0.2689
P Value	0.540873	

Table 5.12 shows the Reply Time (RT) of SBLB and HApoxy for different types of request. The two experiments are carried out under the same conditions. The number of sessions for the experiments was set as (10, 5, 2). The ten indicates the number of sessions and each session consists of five calls that are spaced out by two seconds. The collected data shows that the RT increases as the number of requests increase. This is because the connection rate is increased by 100 requests in each experiment. The statistical analysis shows that there is no significant difference in terms of the RT between the SBLB and HAproxy in thehomogeneous environment through all the fifteen experiments. This result indicates that the performance of HAproxy and SBLB mechanism in thehomogenous environment are almost the same.

Table 5.14 Request per Second (RPS) of SBLB and HAproxy load balancerin homogeneous environment

Number of requests	**SBLB**	**HAproxy**
1000	175.5	169.2
1000	140.7	133.2
1000	172.7	161.2
1000	159.1	151.2
1000	170.6	162.3
1000	180.0	172.4
1000	179.7	172.2
1000	173.6	170.2
1000	167.1	159.0
1000	160.3	155.4
1000	165.7	161.2
1000	131.5	126.3
1000	130.8	125.3
1000	146.0	140.2
1000	130.4	126.0
Mean	158.9	152.353
SD	18.2446	17.5825
T-test	1.0026	
CI	9.23286	8.89783
P Value	0.32465	

Table 5.13 presents the data collection for SBLB and HAproxy in terms of RPS. The request per second indicates the throughput of the load balance system to handle a certain request per second. Based on the statistical analysis, we found that there isno significant difference between SBLB and HAproxy in ahomogeneous environment. This is due to the factors that affect RPS such as the capacity of the host, and the link utilization is not considered in this experiment.This experiment is conducted in aheterogeneous environment for SBLB and HAproxy to achieve realistic and representative results in our evaluation. The links are configured with 100 Mb/s of throughput while others only have 50 Mb/s. The same is used for the latency that ranges from 10ms to 20 ms. In addition, the CPU and the RAM of the hosts vary. For example, some hosts are configured with 2GHz and 1024 RAM, while others are set to 1GHz and 512 RAM. We used the same metrics that were implemented in the homogeneous

environment to show the differences between SBLB and HAproxy in both environments.

Table 5.15The average response time of SBLB and HAproxy in heterogeneous environment

Number of requests	**SBLB**	**HAproxy**
500	4.91	6.83
600	5.06	7.58
700	5.19	7.64
800	5.24	7.88
900	5.35	7.96
1000	5.35	8.15
1100	5.36	8.66
1200	5.50	8.77
1300	5.51	8.84
1400	5.77	8.89
1500	5.85	8.96
1600	6.01	9.03
1700	6.10	9.17
1800	6.38	9.38
1900	6.45	9.55
Mean	5.60	8.49
SD	0.47018	0.77329
T-test	12.3459	
CI	0.23794	0.39133
P Value	0.00001	

In table 5.14, the average response time of SBLB and HAproxy in aheterogeneous environment for 15 experiments are presented. The statistical information at the bottom shows that the value of the T-test exceeds the critical value (12.3459>2.069). So, the means are significantly different, and P value is (0.00001 < 0.05)

Table 5.16The reply time of SBLB and HAproxy in heterogeneous environment

Number of requests	**SBLB**	**HAproxy**
500	2.55	5.16
600	2.57	5.34
700	2.76	5.80
800	3.57	6.02
900	3.81	6.12
1000	4.04	6.29
1100	4.17	6.31
1200	4.21	6.45
1300	4.24	6.62
1400	4.30	6.62
1500	4.35	6.63
1600	4.42	6.70
1700	4.48	6.90
1800	4.55	6.94
1900	4.62	7.01
Mean	3.91	6.33
SD	0.71836	0.55634
T-test	10.3036	
CI	0.36353	0.28154
P Value	0.00001	

In terms of reply time (RT), Table 5.15 presents the data collection for SBLB and HAproxy in the heterogeneous environment. The mean shows 47.2% difference in between two values. The absolute value of the calculated T test exceeds the critical value (10.3069>2.056) which proves there is a significant difference between the two results.

Table 5.17 Request per second of the SBLB mechanism and HAproxy in heterogeneous environment

Number of requests	SBLB	HAproxy
1000	143.0	129.3
1000	158.3	142.2
1000	162.4	148.4
1000	154.4	141.1
1000	170.2	155.8
1000	153.7	140.3
1000	156.5	140.7
1000	172.0	156.3
1000	140.9	127.5
1000	133.6	117.5
1000	134.8	119.7
1000	179.8	166.7
1000	177.5	164.9
1000	162.2	147.5
1000	130.7	115.3
Mean	155.3	140.9
SD	15.9	16.4
T-test	2.44639	
CI	8.07261	8.30505
P Value	0.010484	

Table 5.16 shows the request per second of the SBLB and HAproxy in the heterogeneous environment. The T value shows 10.3069 which is a value greater than the critical value (2.056). This means that there are significant differences between the two values. In addition, P value is 0.010484 < 0.05.

5.4.4 Data Collected for performing comparison of SBLB and RRA in homogeneous and heterogeneous environments

In this section, we compare the performance of SBLB and Round-Robin Algorithm (RRA) in homogeneous and heterogeneous environments. RRA is load balance algorithm that is widely used to distribute the load of the server. It divides the incoming traffic in between hosts in a round robin manner. In fact, it is suitable for the homogeneous environments with a large number of hosts (P. Wang et al., 2011). In

these experiments, we used five pools with adifferent type of services, and each pool consists of five hosts. Three parameters are used in this comparison; average response time, reply time and request per second. The following tables show the results of SBLB and RRA with different number and type of requests.

Table 5.18Average response time of the SBLB and RRA in homogeneous environment

Number of requests	**SBLB**	**RRA**
500	4.88	7.21
600	5.00	7.30
700	5.28	7.38
800	5.57	7.57
900	5.69	7.67
1000	5.91	7.92
1100	6.00	8.01
1200	6.04	8.29
1300	6.26	8.35
1400	6.34	9.14
1500	6.48	9.15
1600	6.64	9.29
1700	6.71	9.42
1800	6.74	9.74
1900	6.79	10.47
Mean	6.02	8.46
SD	0.63076	1.009368
T-test	7.9403	
CI	0.3192	0.5108013
P value	0.00001	

Table 5.17 presents the results of average response time (in seconds) of the SBLB and RR algorithms in the homogeneous environment for 15 different experiments. The results show that the superior performance of the SBLB in the 15 experiments. The mean of ART for the SBLB is 6.02 while the mean of ART for the RRA is 8.46. Such a significant difference in between the two values indicates that SBLB performs better in the homogeneous environment as compared to RRA. The T value shows 7.9403 which is a value greater than the critical value (2.145). This means that there is a significant difference in between the two results.

Table 5.19 Reply time of the SBLB and RRA in homogeneous environment

Number of requests	**SBLB**	**RRA**
500	2.72	5.26
600	2.87	5.52
700	2.88	6.03
800	2.89	6.27
900	3.02	6.50
1000	3.58	6.79
1100	3.65	7.45
1200	3.81	7.56
1300	3.93	8.14
1400	4.12	8.21
1500	4.22	8.48
1600	4.23	8.50
1700	4.24	8.63
1800	4.58	9.35
1900	4.68	9.39
Mean	3.69	7.47
SD	0.67005	1.343103161
T-test	21.0588	
CI	0.33909	0.679691491
P value	0.00001	

Table 5.18 shows the results of the reply time (in seconds) of the SBLB and RRA in the homogeneous environment for all 15 different experiments. Thestatistical information in the bottom of the table shows that there is a significant difference of reply time over the 15 experiments. The means of the SBLB and RRA are 3.69 and 7.47

respectively. It is observed that the value of the T-test exceeds the critical value (9.7481>2.145), and P value is (0.00001 < 0.05).

Table 5.20 Request per second of the SBLB and RRA in homogeneous environment

Number of requests	**SBLB**	**RRA**
1000	190.7	175.5
1000	155.6	140.7
1000	185.9	172.7
1000	172.5	159.1
1000	186.8	170.6
1000	192.3	180.0
1000	191.4	179.7
1000	186.0	173.6
1000	180.1	167.1
1000	173.8	160.3
1000	179.3	165.7
1000	145.4	131.5
1000	145.1	130.8
1000	160.3	146.0
1000	145.2	130.4
Mean	172.7	158.9
SD	17.7646	18.24457542
T-test	2.0957	
CI	8.98994	9.232859409
P value	0.045272	

Table 5.19 presents the results of the request per second (in number) of the SBLB and RRA in the homogeneous environment for 15 different experiments. Based on the statistical analysis, we found that there is asignificant difference in between SBLB and HAproxy in the homogeneous environment. Such a difference clearly appears in the means attribute. For example, RRA shows 158.9 RPS while SBLB algorithm shows 172.7 RPS. The difference in between the two values is almost 14 requests per second. Moreover, the value of T-test exceeds the critical value (2.0957 > 2.0145), and P value is (0.00001< 0.05).

Table 5.21 Average response time of the SBLB and RRA in heterogeneous environment

Number of requests	**SBLB**	**RRA**
500	4.91	9.23
600	5.06	9.48
700	5.19	9.72
800	5.24	9.76
900	5.35	10.01
1000	5.35	10.06
1100	5.36	10.86
1200	5.50	10.98
1300	5.51	11.05
1400	5.77	12.00
1500	5.85	12.21
1600	6.01	12.48
1700	6.10	12.58
1800	6.38	12.74
1900	6.45	12.95
Mean	5.60	11.07
SD	0.47018	1.3188889
T-test	15.1187	
CI	0.23794	0.6674376
P value	0.00001	

Table 5.20 shows the results of average response time (in second) of the SBLB and RRA in the heterogeneous environment for 15 different experiments. According to the statistical information, the mean attribute of ART for SBLB algorithm is 5.60, while

the mean attribute of ART for RRA is 11.07. In these experiments, the critical value is 2.145 (appendix A). The absolute value of the calculated T exceeds the critical value (15.118 > 2.145), and so, the means are significantly different.

Table 5.22 Reply time of the SBLB and RRA in heterogeneous environment

Number of requests	**SBLB**	**RRA**
500	2.55	7.12
600	2.57	7.73
700	2.76	7.89
800	3.57	8.32
900	3.81	8.32
1000	4.04	8.35
1100	4.17	8.39
1200	4.21	8.47
1300	4.24	8.81
1400	4.30	8.98
1500	4.35	8.98
1600	4.42	9.15
1700	4.48	9.30
1800	4.55	9.32
1900	4.62	9.50
Mean	3.91	8.58
SD	0.71836	0.6656832
T-test	18.4739	
CI	0.36353	0.336876
P value	0.00001	

Table 5.21 depicts the results of the reply time (in seconds) of the SBLB and RRA in the heterogeneous environment for 15 different experiments. In the bottom of the table, the statistical analysis presents the means of RT of SBLB that shows 3.91. The mean of RT of RRA shows 8.58. Such a significant difference in between the two means is presented by a T value that exceeds the critical value (18.493>2.145),and a P value is (0.00001 < 0.05).

Table 5.23 Request per second of the SBLB and RRA in heterogeneous environment

Number of requests	SBLB	RRA
1000	173.2	143.0
1000	188.9	158.3
1000	194.2	162.4
1000	186.3	154.4
1000	202.5	170.2
1000	186.0	153.7
1000	188.9	156.5
1000	204.7	172.0
1000	173.7	140.9
1000	166.3	133.6
1000	167.9	134.8
1000	212.9	179.8
1000	210.7	177.5
1000	196.5	162.2
1000	165.4	130.7
Mean	187.9	155.3
SD	15.9	16.0
T-test	5.591	
CI	8.06016	8.0726101
P value	0.00001	

Table 5.22 shows the results of the request per second (in number) of the SBLB and RRA in the heterogeneous environment for 15 different experiments. The average request per second of the SBLB algorithm is 187.9 while RRA is 155.3. The T-test value illustrates the significant difference in between the two results where P value is less than (0.05).

5.5 Conclusion

This chapter presented the data collection to evaluate the proposed SBLB based on three parameters: average response time, reply time and request per second. The results are collected by sampling the evaluation parameters with 15 different experiments. The data collection is carried out by sampling the parameters considering two factors; (1) Type of the request (data or compute) (2) the number of requests (100 to 1900 requests).

The SBLB mechanism is tested on the real environments, and the benchmarking is used to evaluate the mechanism in homogeneous and heterogeneous environments. Multiple VMs were implemented in one physical machine to represent the cloud environment.

We started by explaining the experimental setup as well as components of the experiments. The data collection methods and the statistical model that were used to evaluate the proposed load balance mechanism were presented in this chapter. The results are presented in four steps. In the first step, we evaluated the simulation results by comparing it with the real environment. This result showed that the simulation and real results were closely matched. In the second step, we illustrated the results that showed the impact of the data and compute requests on SBLB mechanism. In the third step, we showed the results that were used for performing an evaluation of SBLB mechanism by comparing the proposed mechanism with HAproxy load balancers. Lastly, the data collection that compared the SBLB and Round-Robin algorithms were presented in this chapter.

It is concluded that SBLB mechanism leveraged the load balance service in the cloud and successfully implemented dynamic load balance based on the type of service.

CHAPTER 6: **CHAPTER RESULTS AND DISCUSSION**

This chapter evaluates the performance of SBLB mechanism and compares it with other load balance solution. Three parameters namely Average Response Time ART, Reply Time RT, and Request Per Second RPS are used to evaluate the performance of SBLB. These parameters are related to our main objective that aims to minimize the response time and maximize the throughput.

First, we analyze the result that was collected from Mininet, the SDN simulation tools, and then, the real test was conducted in OpenStack, a real cloud computing environment. The result was analyzed based on the data and compute request to show the impact of SBLB. In these experiments, SBLB modules are run in the Floodlight controller, and the requests are sent to the hosts. The three parameters namely ART, RT, and RPS are captured. The chapter also focuses on the comparison of the SBLB mechanism results with existing software load balance application such as HAproxy. Besides, SBLB algorithm is also compared with Round Robin algorithm. All the experiments mentioned above are carried out in homogeneous and heterogeneous environments.

This chapter includes five sections. Section 6.1 presents the analysis of SBLB mechanism in the simulation and real environments. The analysis of the data and compute request are presented in section 6.2. The comparison of SBLB with Round-Robin Algorithm(RRA) is discussed in section 6.3 while the comparison of SBLB with HAproxy load balancer software is presented in section 6.4. Section 6.5 concludes the chapter by highlighting the significance of SBLB mechanism.

6.1. Analysis of SBLB mechanism in simulation and real environment

This section analyses the results obtained from the simulation environment that is implemented in Mininet, and real cloud environment that was carried out in OpenStack. The results are presented in Table 5.2, 5.3 and 5.4 in the previous chapter. In these

experiments, a few numbers of requests are sent in each data trace. This is due to the limit of buffer for sending and receiving data when we use enhanced HTTPerf. Therefore, the maximum number of the requests is 190 request per second.

We configure the hosts to use a different type of services such as HTTP, FTP,and video streaming service. EnhancingHTTPerf is used to generate the traffic. This tool can generate various types of request to measure the performance of the SBLB in both environments. We discover that the relationship between the number of requests and the ART and RT are exactly linear. Thus, when we increase the number of requests, these two parameters are increased as well.

Figure 6.1 presents the comparison of average response time through simulation and real environments. In the figure, the y-axis shows the ART that is measured in seconds and the x-axis represents the number of requests of the six different data traces. The results of the Mininet for all six data traces are closer to the results obtained from the OpenStack. The difference in ART for Mininet and OpenStack are 0.63s, 0.62s, and 0.76sin the last three data traces. This small amount of difference validates the results collected from the simulation when compared to the results of real environments.

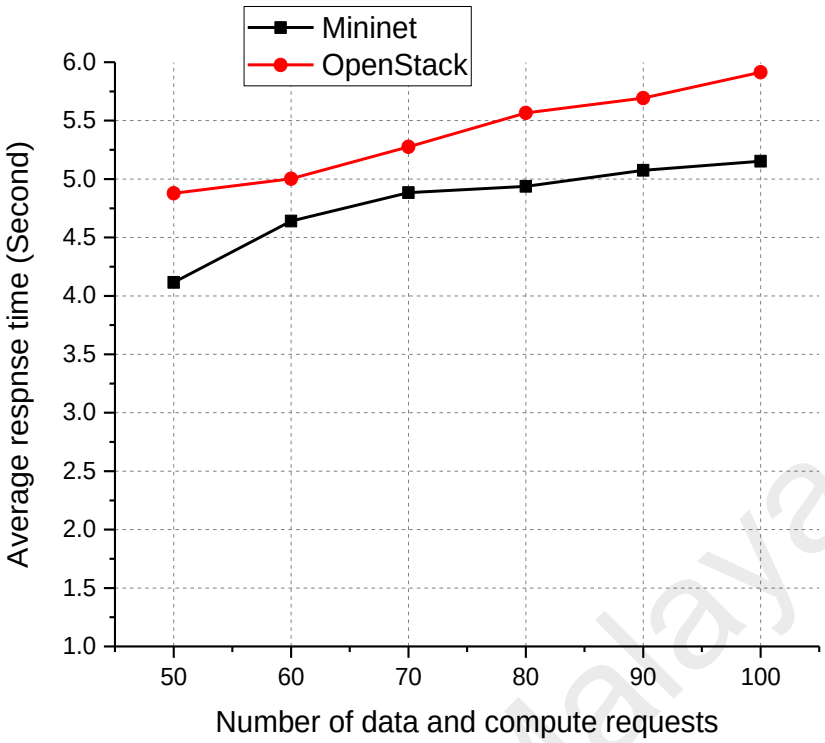

Figure 6.1 ART of the Mininet and OpenStack for SBLB.

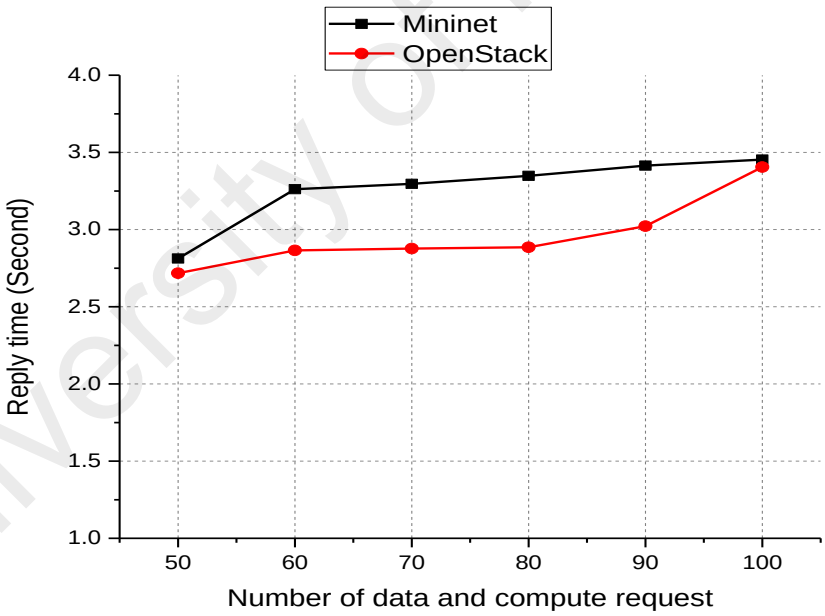

Figure 6.2 RT of the Mininet and OpenStack SBLB

Figure 6.2 shows the reply time of Mininet as compared to OpenStack for sixsamples trace. We can note that two results are close to each other. The RT of both environments is presented in between 2.5s and 3.5s. For example, the mean in Mininet and OpenStack is 3.6s and 3.7s, with the difference of 0.10s. This small amount of difference validates the simulation results with the ones collected from real

environments. In 100 requests, Mininet shows higher RT compared to OpenStack; this is due to the limit of virtual host's capacity that is used in Mininet.

Figure 6.3, presents the request per second for Mininet and OpenStack for six samples trace. The aim of this experiment is to calculate the number of requests that both environments can achieve per second. We divided the number of requests 1000 by the total time of the experiments. The average difference between the simulation and real environment is two requests per second in all data trace. This small amount of difference validates the simulation results with the ones collected from real environments.

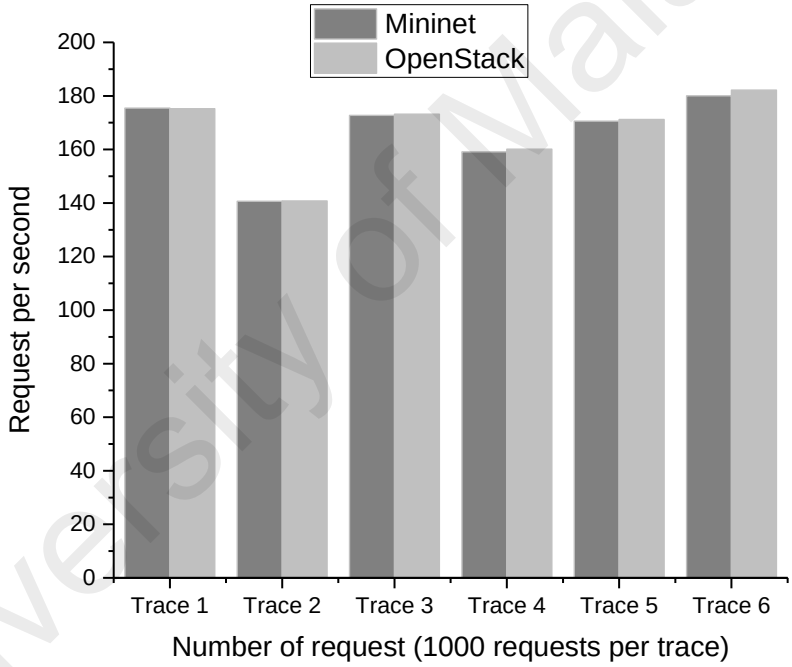

Figure 6.3 RPS of the Mininet and OpenStack for SBLB.

6.2. Collection data of compute and data request in SBLB mechanism

In this section, we analyze the ART, RT,and RPS for data and compute request in homogeneous and heterogeneous environments. The aim of this analysis is to compare the different type of request running in thehomogeneous and heterogeneous environment on SBLB mechanism.

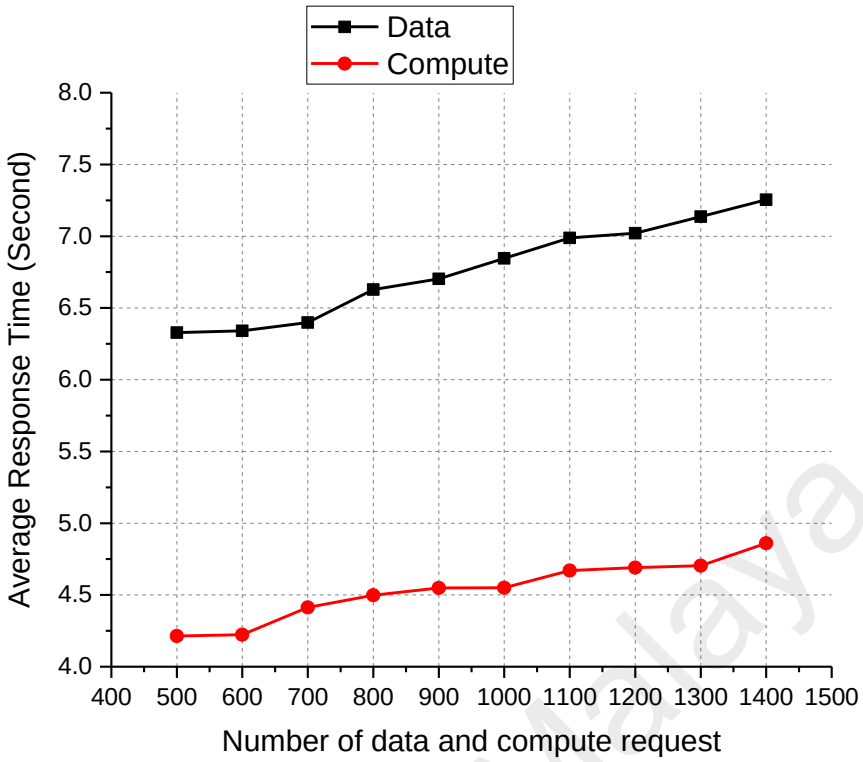

Figure 6.4 ART of data and compute request in thehomogeneous environment for SBLB.

Figure 6.4 presents average response time of data and compute request in the heterogeneous environment. The y-axis shows the ART in seconds and x-axis represents the number of the request. The ART of the compute request ranges from 4.21 to 5.07 seconds, while in the data request, the ART ranges from 6.33 to 7.73 seconds. The highest value of the ART of the compute requests is 4.74s that is lesser thandata request by 2.39s. This big difference indicates that data request has a great impact on SBLB. Because of data request consumes more resources and the hosts take more time to process the request.

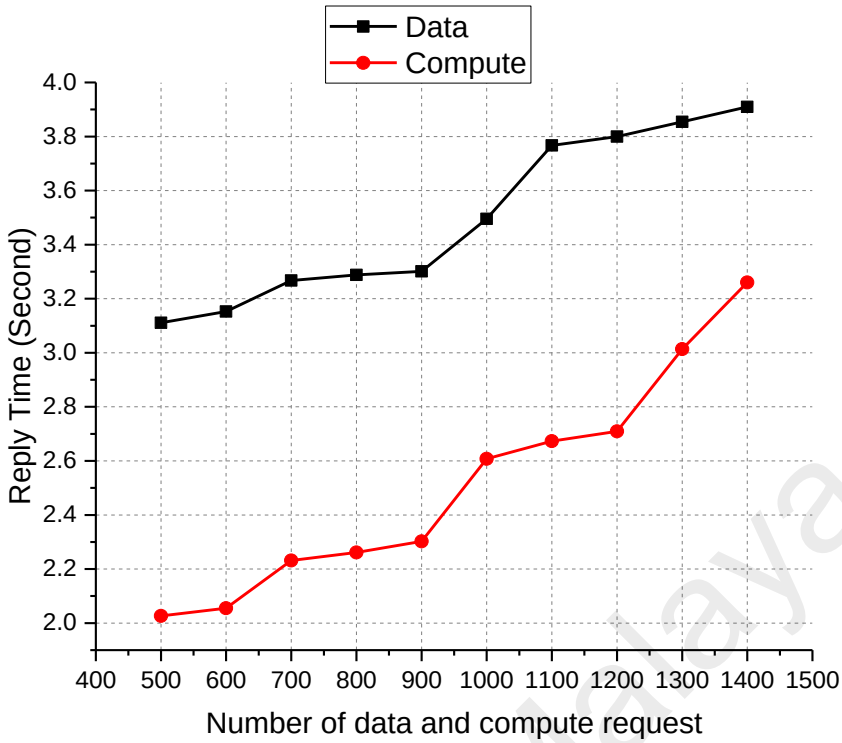

Figure 6.5 RT of data and compute request in thehomogeneous environment for SBLB.

Figure 6.5 shows the reply time of data and compute request in the homogeneous environment. The x-axis represents the number of different requests, and they-axis represents the reply time per second. As depicted from Figure 6.5, when the compute request is 1400, the RT is 3.26 second. The RT for 700 data request achieved almost the same value which is 3.27 second. This value indicates that compute request in SBLB mechanism can achieve twice the number of request compared to the data request. In addition, we notice that RT of the compute request is increased rapidly in the homogeneous environment. For example, with 500 requests, the RT is 2.03 and reaches 3.26 in 1400 requests. In turn, RT of the data request starts from 3.11 up to 3.26 in 1400 requests. This is because of the default idle-time-out of the flow table in Floodlight is set to five seconds. This means, after five seconds if there is no flow to be matched, the flow entry will be removed automatically. The data requests are considered as *Elephant Flow* that remains in the flow table. But, the compute requests are *Normal Flow* that is removed from the flow table after 5 seconds.

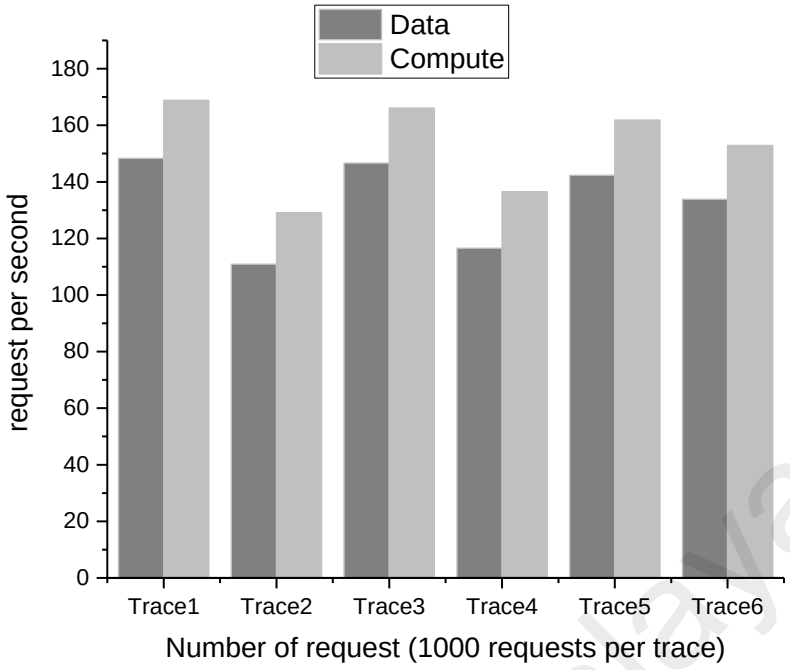

Figure 6.6 RPS of data and compute request in thehomogeneous environment for SBLB.

Figure 6.6 shows the request per second of data and compute request using SBLB mechanism in the homogeneous environment. As depicted from the graph, the RPS of the data request is less than the compute request. For example, in the first and second experiments, the compute request shows 168.8 and 129.1 while the data request shows only 148.3 and 110.9. Moreover, the average of the RPS in data request is 136.9 while the average of RPS in compute request is 156.5. Typically, compute request can produce more request per second compared to the data request. The results indicate that different type of requests has agreat impact on RPS in the homogeneous environment.

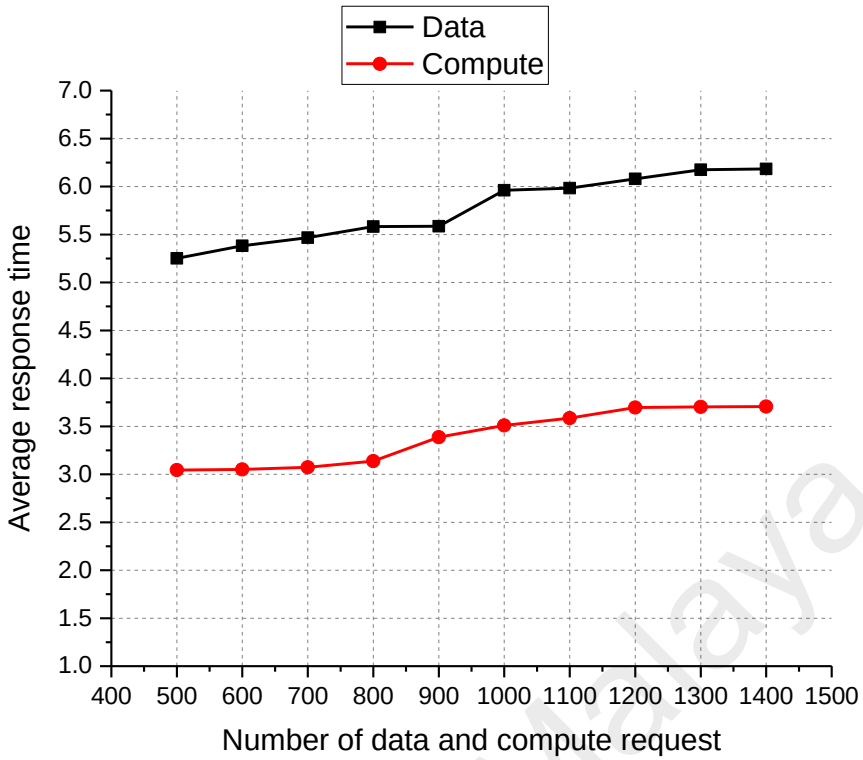

Figure 6.7 ART of data and compute request in aheterogeneous environment for SBLB.

Figure 6.7 presents the average response time of the data and compute request based on various numbers of the request in the heterogeneousenvironment. The graph shows that the ART increases when the number of requestsincreases. For instance, when the client sends 500 requests, the ART is 5.25 seconds, with the 1400 requests, the ART shows 6.18 seconds. Two points are noteworthy in this graph. First, the clear differences of ART between data and compute requests over the 10 trace samples. For example, the highest value of ART of the compute request is 3.71 seconds in 1400 requests, while the lowest value of data request is 5.25 in 500 requests. This is because data requests consume more resources in the hosts, and need longer time to be processed. Secondly, the number of requests does not greatly affect the difference of relevant ART values. In 500 requests, the difference between the data and compute request is 2.21 seconds, and in the 1400 requests, the difference between data and compute requests is 2.48 seconds.

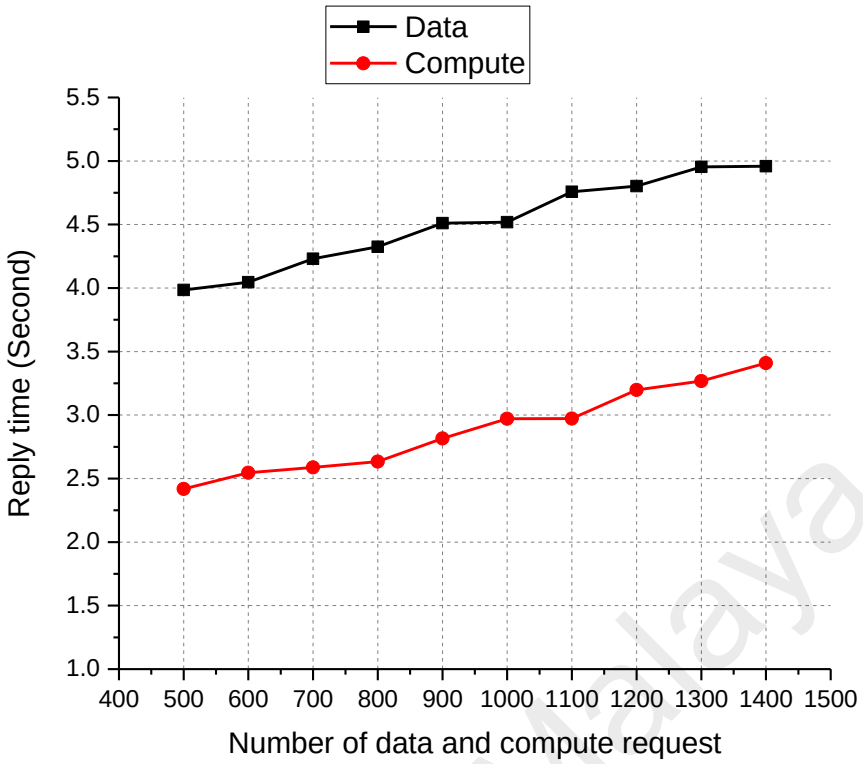

Figure 6.8 RT of data and compute request in heterogeneous for SBLB

Figure 6.8 shows reply time of the data and compute requests based on different numbers of the request in the heterogeneousenvironment. Similar to the ART in the heterogeneousenvironment, the differences between data and compare request clearly appear. For example, the rate of differences between data and compute request in 500, and 1400 requests are 48.7% and 37.0% respectively. This significant variance between the values indicates that the data request needs longer time to handle user request compared to the compute request. Throughout experiments that started by 500 requests up to 1400 requests, the differences between data and compute request remained steady, around the average of 1.63 seconds.

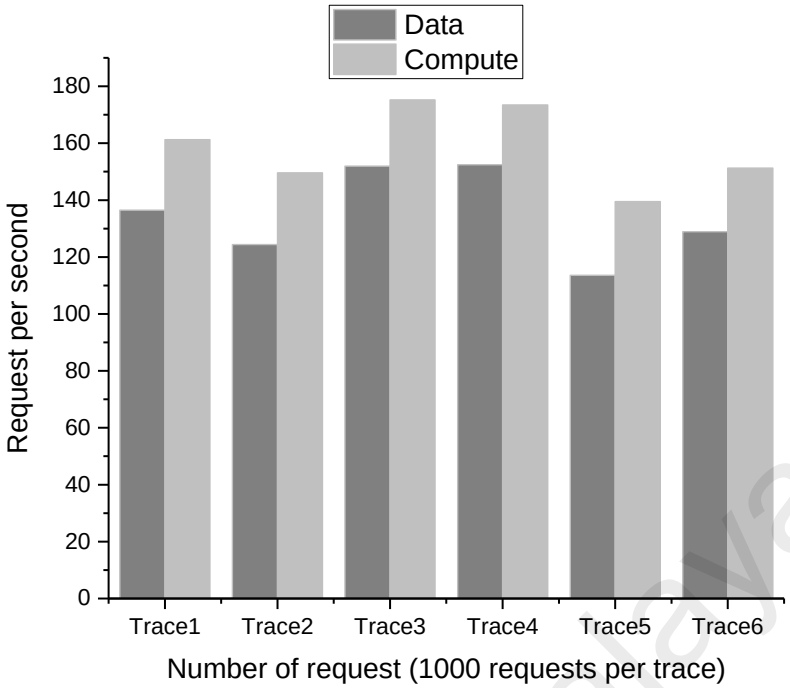

Figure 6.9 RPS of data and compute request in heterogeneous environment for SBLB

In Figure 6.9, the request per second of data and compute request in the heterogeneous environment is presented. The graph shows clearly the significant differences between data and the compute request in terms of RPS in the heterogeneous environment. For example, the minimum difference between data and compute requests is 21.1 request per second that appears in the fourth experiment, while,compute request is recorded as 173.5 RPS, while data request shows 152.4 RPS. Thus, the average differences between data and compute requests over 10 experiments is 24.3 RPS. This big amount of differences between data and compute request prove that the compute request can perform better that data request in the heterogeneous environment compared to homogeneous environment.

6.3. Comparison between SBLB and RRA

In this section, we compare between SBLB and Round-robin algorithm. The comparisons are carried out in homogeneous and heterogeneous environments.

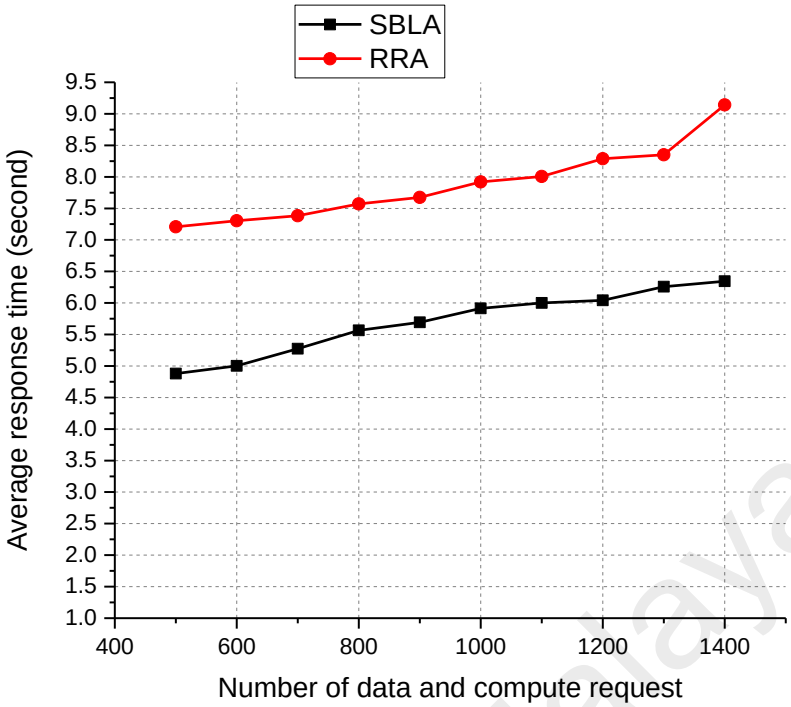

Figure 6.10 ART of the SBLB and RRA in homogeneous environment

In Figure 6.10, the results of the average response time of the server-based load balance algorithm and round robin algorithm in 10 different experiments are presented. The ART for SBLB is less than RRA in a different number of requests. For example, in 500 and 600 requests, the ART of SBLB mechanism is 4.88 and 5.0 while RRA shows 7.21 and 7.30. The mean of ART for SBLB mechanism and RRAis 6.02 and 8.46 respectively. This significant difference demonstrates that our proposed algorithm performs better than RRA in the homogeneous environment.

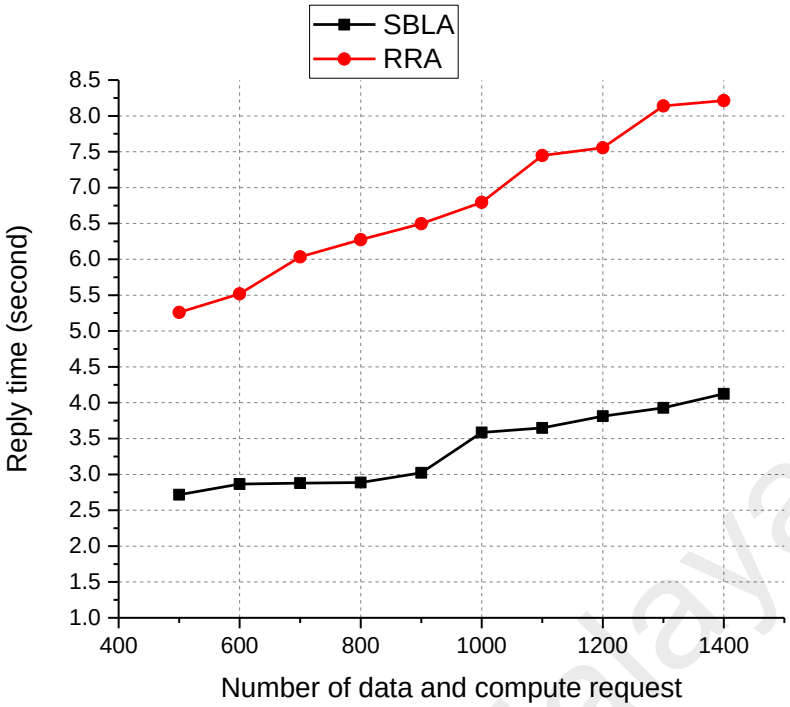

Figure 6.11 RT of the SBLB and RRA in homogeneous environment

The results of reply time for SBLB and RRA in homogeneous environments are presented in Figure 6.11. The results compare between SBLB algorithmand RRA in 10 different experiments. The graph shows that the RT of SBLB algorithm is less than RRA. In addition, the reply time of SBLB is almost steady compared to RRA that increases sharply. For example, with 500 requests, the reply time of RRA shows 5.25 seconds and increases to 8.21 seconds with 1400 request. Although, previous studies (Kaur et al., 2015) prove that RRA performs well in the homogeneous environment, but SBLB appears to outperform RRA. This could be due to the small number of hosts involved in these experiments.

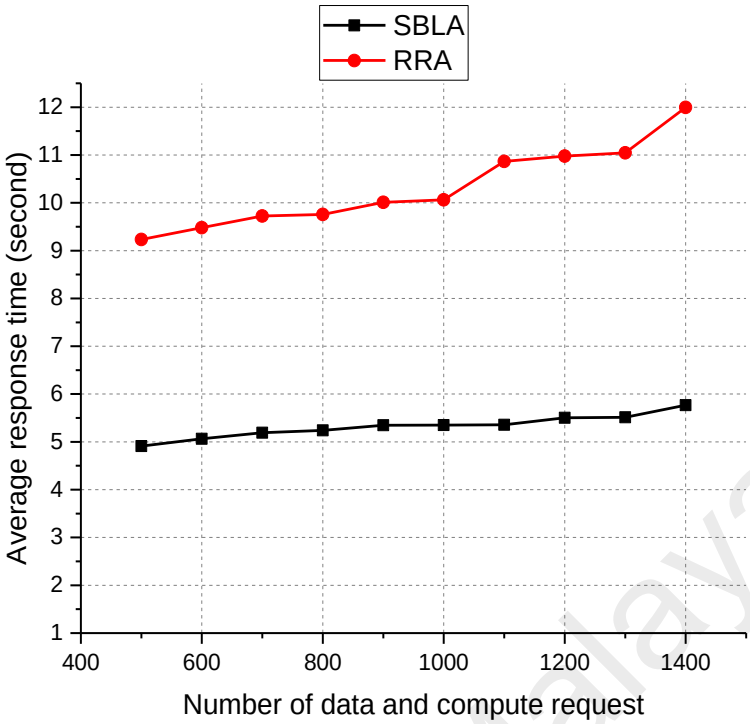

Figure 6.12 ART of the SBLB and RRA in aheterogeneous environment.

Figure 6.12 presents the comparison of average response time in between SBLB and RRA in 10 different experiments. The graph illustrates that SBLB performs better than RR over all experiments. As we can see, ART of SBLB is range between 5 to 6 seconds while RRA shows 9.23 seconds in the first experiment and increases to 12 seconds with 1400 requests. This significant difference proves that our proposed algorithm performs effectively as compared to RRA in the heterogeneous environment. Moreover, increasing the number of requests has a slight impact on the average response time of SBLB algorithm in the heterogeneous environment. In contrast, the number of requests affects RRA.

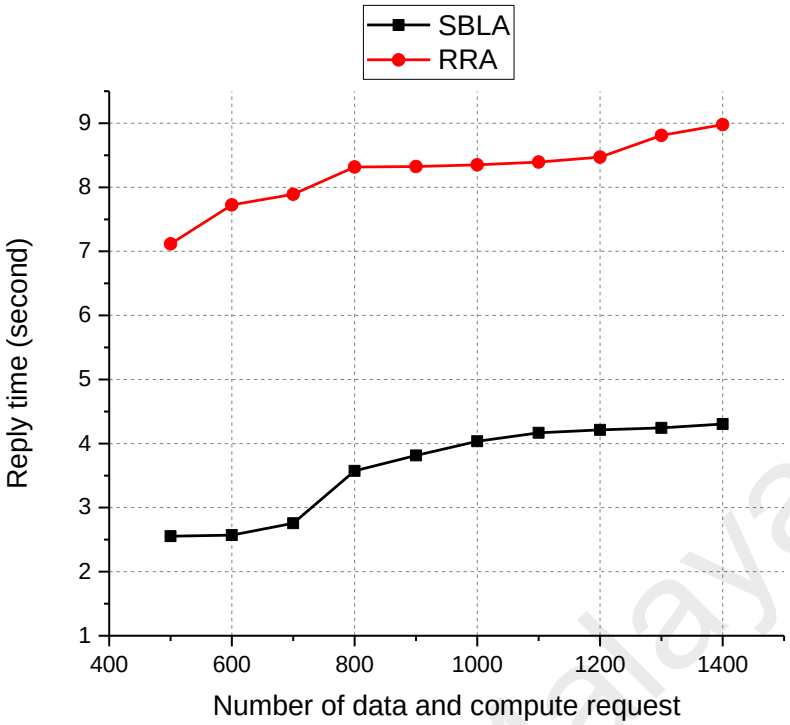

Figure 6.13 RT of the SBLB and RRA in aheterogeneous environment.

Figure 6.13 shows the result of comparison between the reply time of SBLB and Round-robin algorithm in the heterogeneous environment. The graph illustrates that the reply time of SBLB is less than the reply time of the RRA in the heterogeneous environment. For example, the RT for RRA is 7.0s, 7.7s, and 7.9s higher thanthe RT of SBLB in the first three data traces respectively. The high reply time for RRA in the heterogeneous environment is because of sending an incoming request to the host that is already loaded. SBLB performs better because it can dynamically calculate the host load and adjust the parameters according to the type of the service

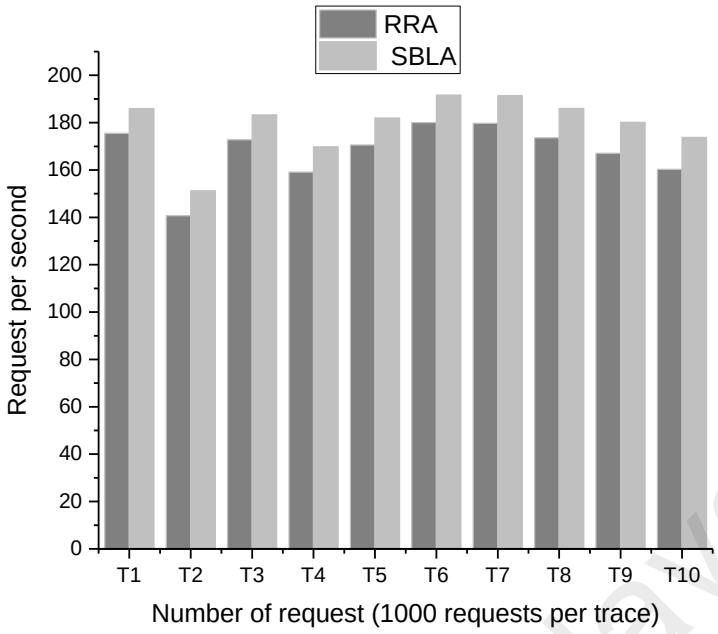

Figure 6.14 RPS of the SBLB algorithmand RRA in homogeneous environment

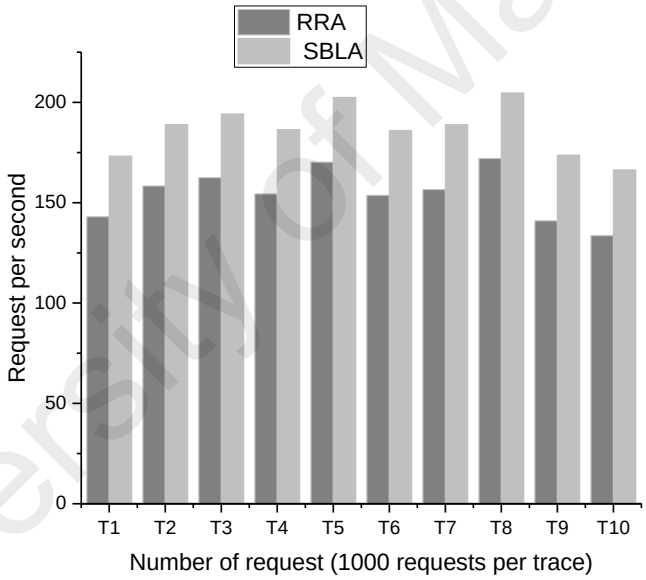

Figure 6.15 RPS of the SBLB and RRA in heterogeneous environment

The request per second of SBLB and round-robin algorithm in the homogeneous and heterogeneous environmentsis presented in Figure 6.14 and 6.15s. The mean of the RPS for SBLB and RRA in ahomogeneous environment is 171.4 and 185.9 respectively,and the different between the two values is 14.5 RPS.

In the heterogeneous environment, we can see that the SBLB performs better than RRA and the differences of RPS in all experiments are clearly illustrated. For example, in Table 5.14, the mean of RPS for SLBA is 158.9 and for RRA is 171.4. So,

the average difference between RRA and SLBA is 13 requests per second. This number indicates the superior performance of SLBA over the RR. This is because RRA does not assume the capacity of the host and link when a new incoming request is sent, while SLBA dynamically calculates a load of each server based on the type of service.

6.4. Comparison between SBLB mechanism and HAproxy load balancer software

In this section, we compare the performance of SBLB mechanism with HAProxy load balancer software in the homogeneous and heterogeneous environment. The parameters used for the comparison are average response time, reply time and request per second. Ten different experiments are used for these comparisons. Although in the homogeneousenvironment there are slight differences between our proposed mechanism and HAproxy load balancer software but in the heterogeneousenvironment, the differences obviously appear. Thus, the results of SBLB are much better in the heterogeneous environment because of the parameters weights that are assigned according to the type of request.

First, we analyze the average response time, reply time and request per second by comparing the results obtained from SBLB mechanism and HAproxy in the homogeneous environment. The aim of this comparison is to highlight the performance differences between SBLB mechanism and HAproxy load balance.

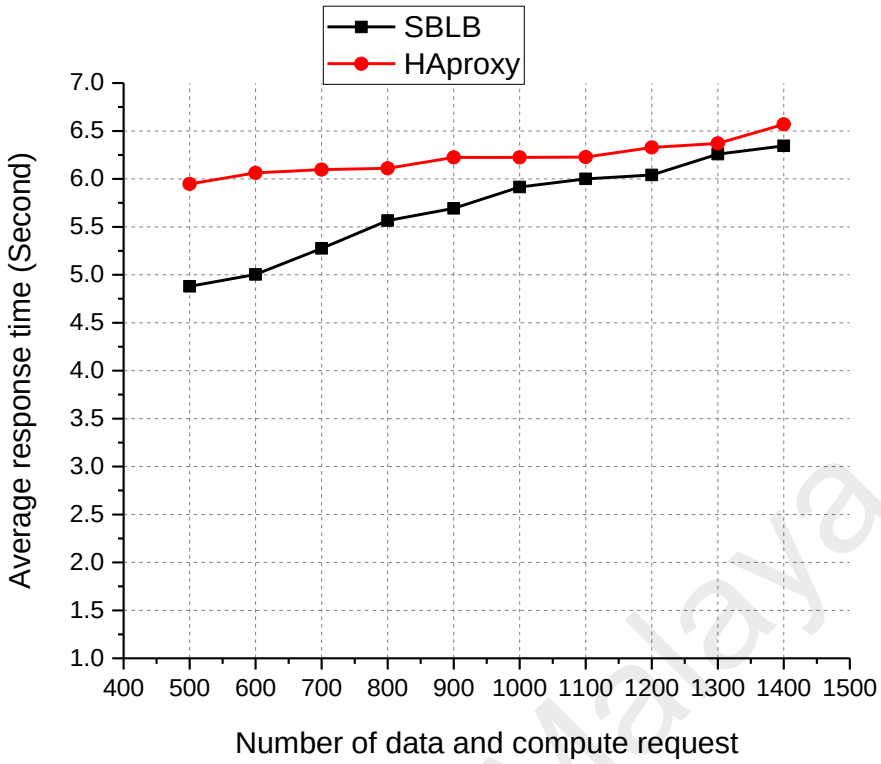

Figure 6.16 ART of the SBLB and HAproxy in homogeneous environment

Figure 6.16 compares the ART of SBLB and HAproxy. The x-axis and y-axis coordinates show the ART and the number of requests. Figure 6.16 clearly shows that SBLB achieved better results in all 10 experiments compared with HAproxy. This is because the SBLB module calculates the load of the host and adjusts the parameters according to the type of service. For example, if the client sends a data request to the controller, it will first check the type of service and use the data request equation (equation 4.5) to calculate the load of all hosts that provide this type of service. While in the HAproxy, the load balancer only used one schema for all type of services.

Figure 6.17 depicts the comparison of SBLB and HAproxy in terms of reply time in 10 experiments. The graph illustrates that the RT of SBLB is better than HAproxy in first five experiments

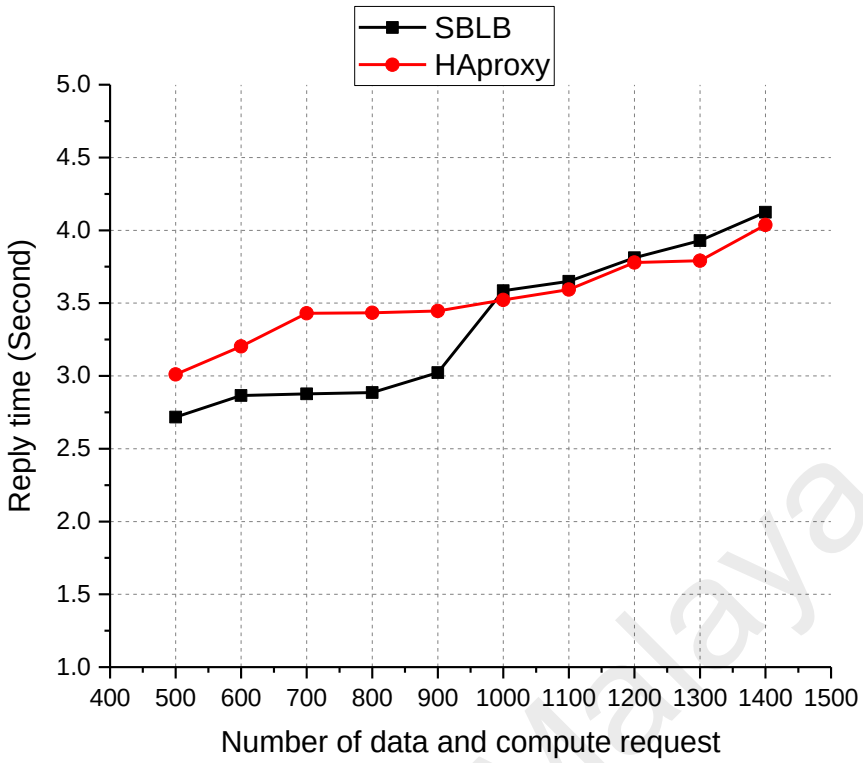

Figure 6.17 RT of the SBLB and HAproxy in homogeneous environment

For instance, when there are 500 requests, the RT in SBLB and HAproxy are recorded as 2.72 and 3.01 respectively. We notice that after 1000 requests, the RT of the SBLB and HA proxy are almost the same. In thehomogeneous environment, all hosts have the same specification in term of CPU, RAM, and requests are divided equally among the hots. When the requests are increased (1000 requests), the five hosts reach their maximum capacity to process more request. This small difference is also due to both solutions used a dynamicscheme that takes into account the host's load in the homogeneous environment.

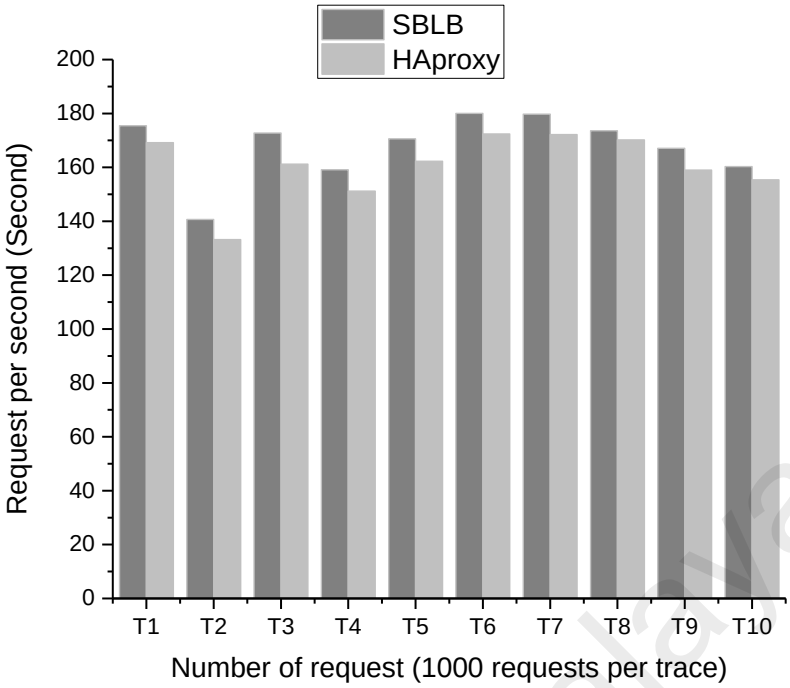

Figure 6.18 RPS of the SBLB and HAproxy in homogeneous environment

Figure 6.18 shows request per second for the SBLB and HAproxy load balancer for 10 different experiments. As we can see, the number of the RPS are diverse, but all RPS are ranging from 130 to 180 requests per second. The traffic generating tool calculates the RPS every 10 seconds by dividing the total time, and reply rate e.g. suppose that in 10 seconds, the reply rate is 1566 byte, then the RPS should be 156.6 requests per second. As we can see, the difference between SBLB and HAproxy in term of RPS in the homogeneous environment is not as big as in the heterogeneous environment. For example, the highest difference shows 11.5 RPS in the third experiment. This is because both solutions used dynamic schema and the host's resources are the same. But SBLB considers the type of request and calculate the host load base on that, while in HAproxy this factor is not considered.

In this section, we analyze the average response time, reply time and request per second by comparing the results of the SBLB and HAproxy in the heterogeneous environment.

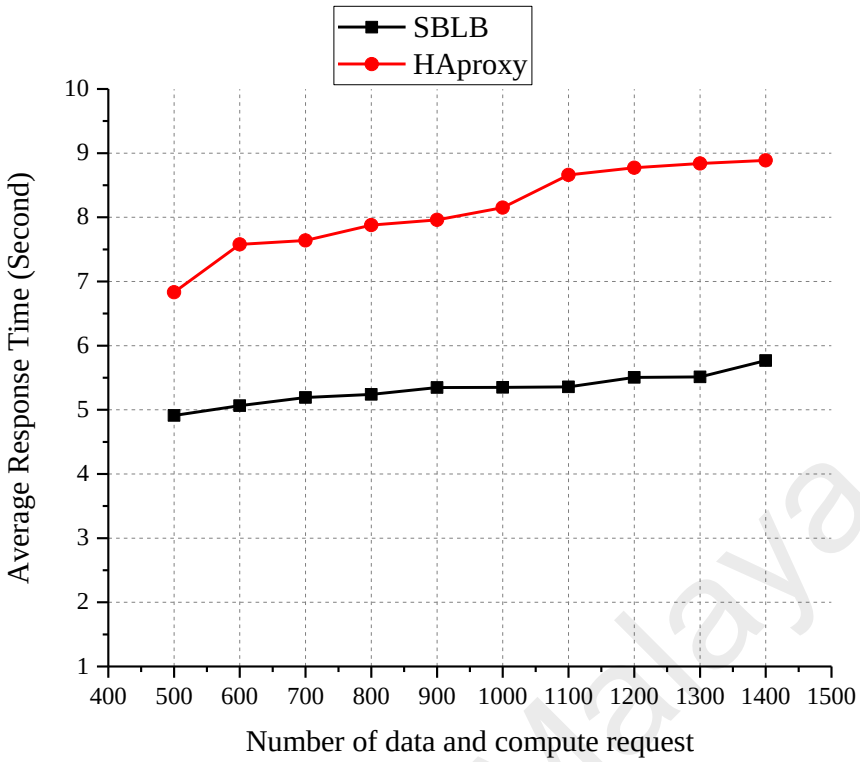

Figure 6.19 ART of the SBLB and HAproxy in heterogeneous environment

Figure 6.19 shows the ART of the SBLB and HAproxy in the heterogeneous environment for 10 experiments. The y-axis shows the ART in seconds and x-axis represents the number of requests. We notice from Figure6.19 that SBLB performed better than HAproxy load balancer in all 10 experiments. For example, when clients send a total of 500 requests, the ART of the SBLB shows 4.91 seconds as compared to HAproxy load balancer that shows 6.83 seconds for the same number of requests. The ART in both solutions is increased when the number of requests increases. The peak value of ART is presented in 1400 requests for SBLB and HAproxy. SBLB shows better performance as compared to HAproxy.

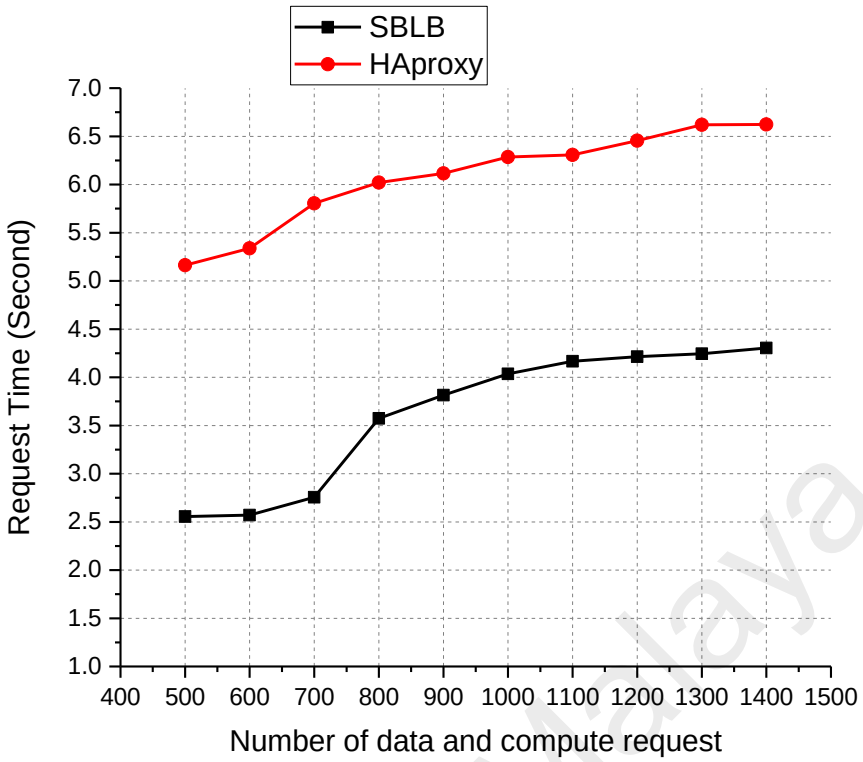

Figure 6.20 RT of the SBLB and HAproxy in heterogeneous environment

Figure 6.20 illustrates the reply time of SBLB and HAproxy in the heterogeneous environment. The result shows that the reply time of SBLB is shorter as compared to HAproxy load balancer over 10 experiments. For example, the RT of SBLB with 500, 600 and 700 requests are 2.55s, 2.57s, 2.76s respectively, while in HAproxy, RT show 5.16s, 5.34s, 5.80s for the same number of requests. Although, the RT of SBLB increases after 800 requests but the significant difference between SBLC and HAproxy remains.

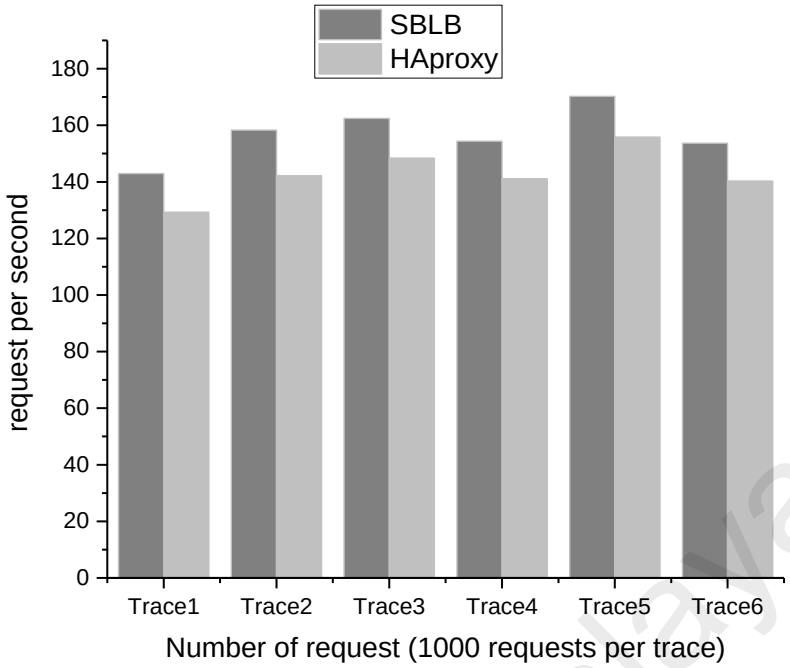

Figure 6.21 RPS of the SBLB and HAproxy in heterogeneous environment

In Figure 6.21, the request per second of the SBLB and HAproxy in the heterogeneous environment is presented. We observed from the results that SBLB could produce a high number of requests as compared to HAproxy and the differences between them are clearly visible. The average difference between SBLB and HAproxy is 14.5 request per second. This is because in the heterogeneous environment the bandwidth of the link and hosts resource vary. In SBLB, the controller calculates the host load and adjusts the load balance parameters according to the type of request, while HAproxy uses a simple dynamic load balance without taking into account this factor.

6.5. Conclusion

In this chapter, the experimental result of SBLB is discussed to prove the efficiency of the proposed mechanism on the basis of the average response time, reply time and request per second.The experiments were investigated based on two criteria such as (1) number of request, (2) type of request. In addition, the experiments are carried out in homogeneous and heterogeneous environments.

First, we validated our results by comparing between simulation and real environment. Then, we analyzed the data and compute request in homogeneous and heterogeneous environments. We observed that compute request showed an average of 4.67s in average response time and 2.82s in reply time in the homogeneousenvironment. These numbers are decreased by 1.07 in ART and 0.42 in RT in the heterogeneous environment. Then, our proposed load balance algorithm was compared with Round-robin algorithm. SBLBhas clearly outperformed the RRA in homogeneousand heterogeneous environments. Lastly, we compare between SBLB and HAproxy load balancer software. The comparison showed that in the homogeneousenvironment the performance of HAproxy and SBLB mechanism is the same in terms of ART and RT. But in the heterogeneousenvironment, the SBLB mechanism shows less ART and RT as compared to HAproxy.

In summary, SBLB performs better in the heterogeneous environment as compared to homogeneous environment. Moreover, the analysis of the data and compute request reveals that the performance of the proposed mechanism shows less ART and RT in the heterogeneous environment.

CHAPTER 7: **CONCLUSION**

This chapter aims to present the epilogue and summary of the research that is carried out in this study. First, the objectives of the study are re-examined to make sure that each objective is accomplished within the scope of the study. Secondly, we discussed the contributions of the studyin details. Moreover, the scopes, limitations, and delimitations of the study are presented. Lastly, the suggestions for the future research directions are highlighted.

The organization of this chapter is as follows. Section 7.1 explains how the objectives have been achieved. In section 7.2, the contributions of the research are provided. Section 7.3 shows the scope as well as the limitations and delimitations. Section 7.4 highlights the open issues and future research direction.

7.1 Re- examining the objectives of the research

This study aims to solve a problem of using single load balance scheme with different types of request. The proposed solution aims to provide a load balance mechanism that can minimize response time and maximize the throughput. In the following, we show how the research objectives that are presented in Chapter 1 are accomplished.

In the first objective, we performed a gap analysis review on the approaches/techniques of the load balancing solutions in the cloud. A survey was conducted to study the state-of-the-art of load balance solution that utilizes SDN. Based on the study, we listed the limitations of the existing load balancers that are widely used in the cloud. Besides, the open source and commercial SDN load balancing solutions are revised and classified. Based on this classification, a thematic taxonomy is proposed after studying more than 200 papers that explained the concept and implementation of the SDN. Over and above, the study focused on SDN-SLB solutions that include 30 papers and was further summarized into 18 solutions as presented in Table 2.7 in

chapter 2. We used qualitative analysis to identify the open research issues of the server load balance in SDN. This ledto defining the research problem that was analyzed and proved in chapter 3.In the second objective, a service based load balancing mechanism was presented. The mechanism is designed to provide load balance using software defined network. The mechanism was geared to provide load balance based service to minimize the response time and maximize the throughput. Three modules are developed to run on the top of the selected SDN controller (Floodlight). The first module was service classification that categorized the request into two types namely computer request and data request, based on the type of the service. The second module was that dynamically load balance the request based on the request type. This module was designed to receive the type of request from the service classification module as well as the status of the hosts and network link from monitoring module. Based on the information and the load balance mechanism, the module distributed the incoming request to the best hosts. The third module is the monitoring module that periodically reported the status of the hosts and network links to the controller. In order to avoid the overhead of the controller, the current load of the link bandwidth and hosts status are sent every five seconds.

In the third objective, we leveraged the OpenFlow protocol to provide online traffic classification that can identify the type of request. The classification approach that relied on port member and protocol type, as well as, IP address for identifying the type of the request was proposed. Then, we utilized*MemoryStorageSource* service that was provided by Floodlight controller to build service table in which PacketIn information is stored.In the last objective, the proposed load balance mechanism was evaluated and compared with existing load balancing solutions. Some experiments were carried out in a different environment to evaluate the mechanism. First, we used *Mininet* simulation to analyze the proposed solution; the network topology was simulated to

connect to Floodlight controller that is installed on theremote computer and is connected remotely to the *Mininet.* In the real cloud environment, we used OpenStack to create VMs that works as hosts, and two additional computers are used as a client to send the traffic. We evaluated the result obtained from both environments, and different type of requests are analyzed and evaluated. In addition, we compared the proposed load balance mechanism with existing software load balance that was implemented in the traditional network. The comparison of the proposed mechanism with another load balance algorithm was carried out as well.

7.2 Contributions of the study

This section presents several contributions that were highlighted in chapter 1. The contributions were described as follows:

7.1.1 Taxonomy of SDN Server Load Balance

We reviewed more than 150 papers that discussed the implementation of load balance in SDN. Based on this review, a thematic taxonomy was proposed to provide a conceptual knowledge in terms of the server load balance in SDN. The taxonomy was categorized based on four parameters; approach/techniques, controller, algorithms, and experiment's environment. This work was presented in Chapter 2.

7.1.2 Studying the impacts of the request on load balance system

In the second contribution, we investigated the impact of user's request on load balance system by analyzing the request from adifferent perspective. We found that several factors could affect the server load balance. First, we studied the impact of the number of the request on average response time and reply time . We proved that by conducting several experiments, the number of the request had a significant effect on the load balance system. The other two factors that impact load balance were the size and type of the request. In addition, we studied the impact of the request on hosts load

by calculating Request per Second (RPS) which each server can handle during a specific time.

7.1.3 Proposing Service Based Load Balance (SBLB) Mechanism

The SBLB mechanism represented the main contribution of the study. In this mechanism, three module are implemented and integrated into Floodlight controller. Such modules were written in Java language, the language that is used to develop Floodlight controller, and additional shell scripts were used to configure the server pools and VIP. In the service classification module, we developed a hybrid approach to identify the type of the request that is divided into two types; *computer* and *data* request. Moreover, Service Table (ST) was created to store information about the *packetIn* in the controller. This information is utilized to provide service classification. In the load balance module, the parameters of the hosts were calculated based on the type of the service provided by the monitoring module. In turn, the parameters were adjusted according to the request type. The monitoring module is responsible for collecting the hosts and link utilization periodically and send to the Floodlight controller.

7.1.4 Enhancing OpenFlow protocol to provide traffic classification

In this contribution, the OpenFlow protocol was enhanced to provide online traffic classification that is used to identify the type of request. We utilized the *receive function* to parse the PacketIn information that includes the number of the port, protocol and IP address of the server. Based on this information, when the PacketIn is sent to the controller, the Service Table (ST) is verified to get the type of the service and send it to the load balance module. This table includes all Host IP address associated with protocol and port number. The ST is created when the controller is initially run. During the running of the system hosts, we can dynamically add hosts, and the controller can detect the IP, service type and port number and then saves them in the ST.

7.1.5 Evaluation and validation of the proposed solution

In the last contribution, the results were evaluated and validated using different statistical approach. Several experiments wereconducted in the simulation and real environment toevaluate the response time and throughput of the SBLB mechanism. The comparison between SBLB mechanism and exciting software load balance was carried out in the homogeneous and heterogeneous environment in terms of of average response time, reply time and request per second. We analyzed thedifferent type of the request in various conditions.

7.3 Limitations and Delimitations

In this section, we highlighted the limitations and delimitations of the study. First, we presented the delimitations that include the boundaries that were set by the researcher in this study. The following were delimitations;

- During reviewing the papers related to SDN load balance, several problems were raised, such as server load balance with multi-tenancy in the cloud, virtualization, and server load balance, the impact of the service chain on load balance. However, we focused on the problem of the load balancing that used the same schema for a different type of the service. Thus, we proposed service based load balance mechanism that can minimize the response time and maximize the throughput of the system.
- The second delimitation was that the availability of sufficient resources for the experiment. Nevertheless, we implemented the experiment in the real environment using OpenStack, but few numbers of the VMswere used as hosts. This number is limited to five servers per pool.
- Another delimitation was that thousands of the services are provided in the cloud, but we selected the common and known services. However, the proposed

mechanism can be used for any number of the services, and load balance parameters can be adjusted according to the service requirements.

- There are several metrics which are used to measure the effect of the load balance system. In this research, we focused on Average Response Time (ART), Reply Time (RT) and Request per Second (RPS).

Secondly, the limitations of the study can be summarized as follows:

- We used virtual switch (Open VSwitch) instead of the real OpenFlow switch.
- The hosts are simulated to run adifferent type of services by developing Python script that runs on each host when they initially run.
- Due to the limit of the resource of the hosts, the interval time between sending and receiving the request was configured to be five seconds.
- The main objective of the study focused on load balance mechanism that canminimize the response time and maximize the throughput. Traffic classification module is developed. However, this research was not concerned with the related accuracy, precision, and recall of traffic classification.

7.4 Future research directions

In the following, we present possible future research directions that further studies can be conducted to extend the SBLB mechanism. We identify four specific areas of research that may effectively enhance the SBLB mechanism.

- First, this study only focused on load balance that minimizes the response time and maximizes the throughput. The scalability of the load balance is not addressed in this study. For example, Adding hosts to existing pool dynamically when all members of the pool are overloaded could be used with the proposed mechanism.

- One possible feature that may be added is to use multiple controllers with SBLB mechanism to avoid asingle point of the failure problem and to check how the other controllers can balance the load when the master controller goes down.
- Other future works include proposing different traffic classification approach to identify the application instead of the service. So, the load balance can be based on the type of the application instead of the service

7.5 Conclusion

This chapter aims to present the epilogue and summary of the research that is carried out in this study. First, the objectives of the study are re-examined to make sure that each objective is accomplished within the scope of the study. Secondly, we discussed the contributions of the studyin details. Moreover, the scopes, limitations, and delimitations of the study are presented. Lastly, the suggestions for the future research directions are highlighted.

The organization of this chapter is as follows. Section 7.1 explains how the objectives have been achieved. In section 7.2, the contributions of the research are provided. Section 7.3 shows the scope as well as the limitations and delimitations. Section 7.4 highlights the open issues and future research direction.

REFERENCES

Abdelzaher, T. F., & Lu, C. (2000). *Modeling and performance control of internet servers. In Decision and Control, 2000. Proceedings of the 39th IEEE Conference on* (Vol. 3, pp. 2234-2239). IEEE.

Adelman, K. A., Kashtan, D. L., Palter, W. L., & Derrell, D. P. I. (2000). *U.S. Patent No. 6,078,957*. Washington, DC: U.S. Patent and Trademark Office.

Al-Shabibi, A., De Leenheer, M., Gerola, M., Koshibe, A., Parulkar, G., Salvadori, E., & Snow, B. (2014, August). *OpenVirteX: Make your virtual SDNs programmable*. In *Proceedings of the third workshop on Hot topics in software defined networking* (pp. 25-30). ACM.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., . . . Stoica, I. (2010). A view of cloud computing. *Communications of the ACM, 53*(4), 50-58.

Azodolmolky, S. (2013). *Software Defined Networking with OpenFlow*: Packt Publishing Ltd.

Balancer, K. S. l., https://kemptechnologies.com/sdn-adaptive-load-balancing/,*[accessed: 27/06/2016]*.

Balancer, L. P. L, https://linerate.f5.com/,*[accessed: 25/05/2016]*.

Bansal, D., Warkhede, P. R., & Venketesan, T. (2012). *U.S. Patent No. 8,266,204*. Washington, DC: U.S. Patent and Trademark Office.

Baucke, S., Mestery, K., Shaikh, A., & Wright, C. (2013). OpenDaylight: *An Open Source SDN for your OpenStack Cloud*. *An Open-Stack Summit, Hong Kong*.

Bays, L. R., & Marcon, D. S. (2011). *Flow based load balancing: Optimizing web servers resource utilization*. Journal of Applied Computing Research, 1(2), 76-83.

Becchi, M., Franklin, M., & Crowley, P. (2008, September). *A workload for evaluating deep packet inspection architectures*. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on* (pp. 79-89). IEEE.

Ben-Shaul, I., Cidon, I., Kessler, I., Lev-Ran, I., & Unger, O. (2005). *U.S. Patent No. 6,976,090*. Washington, DC: U.S. Patent and Trademark Office.

Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., ... & Parulkar, G. (2014, August). *ONOS: towards an open, distributed SDN OS*. In Proceedings of the third workshop on Hot topics in software defined networking (pp. 1-6). ACM.

Bernaille, L., Teixeira, R., Akodkenou, I., Soule, A., & Salamatian, K. (2006). *Traffic classification on the fly*. *ACM SIGCOMM Computer Communication Review, 36*(2), 23-26.

Bezemer, C. P., & Zaidman, A. (2010, September). M*ulti-tenant SaaS applications: maintenance dream or nightmare*?. In Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE) (pp. 88-92). ACM.

Bianchi, G., Bonola, M., Capone, A., & Cascone, C. (2014). *OpenState: programming platform-independent stateful openflow applications inside the switch*. ACM SIGCOMM Computer Communication Review, 44(2), 44-51.

Blial, O., Ben Mamoun, M., & Benaini, R. (2016). An Overview on SDN Architectures with Multiple Controllers. *Journal of Computer Networks and Communications, 2016*.

Boero, L., Cello, M., Garibotto, C., Marchese, M., & Mongelli, M. (2016). *BeaQoS: Load balancing and deadline management of queues in an OpenFlow SDN switch*. Computer Networks, 106, 161-170.

Bourke, T. (2001). *Server load balancing*: " O'Reilly Media, Inc.".

Bozakov, Z., & Sander, V. (2013). *OpenFlow: A Perspective for Building Versatile Networks* Network-Embedded Management and Applications (pp. 217-245): Springer.

Brandt, M., Khondoker, R., Marx, R., & Bayarou, K. (2014). *Security Analysis of Software Defined Networking Protocols*—OpenFlow, OF-Config, and OVSDB. Paper presented at the The 2014 IEEE Fifth International Conference on Communications and Electronics (ICCE 2014), DA NANG, Vietnam.

Bryhni, H., Klovning, E., & Kure, Ø. (2000). *A comparison of load balancing techniques for scalable web servers*. *Network, IEEE, 14*(4), 58-64.

Bujlow, T., Carela-Español, V., & Barlet-Ros, P. (2013). Comparison of Deep Packet Inspection (DPI) Tools for Traffic Classification: Universitat Politècnica de Catalunya.

Capone, A., Cascone, C., Nguyen, A. Q., & Sanso, B. (2015, March). *Detour planning for fast and reliable failure recovery in SDN with OpenState*. In Design of Reliable Communication Networks (DRCN), 2015 11th International Conference on the (pp. 25-32). IEEE.

Cardellini, V., Colajanni, M., & Philip, S. Y. (1999). *Dynamic load balancing on web-server systems*. IEEE Internet computing, 3(3), 28.

Cash, S., Jain, V., Jiang, L., Karve, A., Kidambi, J., Lyons, M., . . . Patel, N. (2016). *Managed infrastructure with IBM Cloud OpenStack Services*. IBM Journal of Research and Development, 60(2-3), 6: 1-6: 12.

Chang, H., & Tang, X. (2010, December). *A load-balance based resource-scheduling algorithm under cloud computing environment*. In *International Conference on Web-Based Learning* (pp. 85-90). Springer Berlin Heidelberg.

Chen, W., Li, H., Ma, Q., & Shang, Z. (2014, June). *Design and implementation of server cluster dynamic load balancing in virtualization environment based on OpenFlow.* In Proceedings of The Ninth International Conference on Future Internet Technologies (p. 9). ACM.

Chen, W., Shang, Z., Tian, X., & Li, H. (2015). *Dynamic server cluster load balancing in virtualization environment with OpenFlow*. International Journal of Distributed Sensor Networks, 11(7), 531538.

Chen, Y. J., Shen, Y. H., & Wang, L. C. (2014, December). *Traffic-Aware Load Balancing for M2M Networks Using SDN*. In Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on (pp. 668-671). IEEE.

Chiong, J. (2013). *U.S. Patent Application No. 13/791,760.*

Chou, L. D., Yang, Y. T., Hong, Y. M., Hu, J. K., & Jean, B. (2014). *A genetic-based load balancing algorithm in openflow network.* In Advanced Technologies, Embedded and Multimedia for Human-centric Computing (pp. 411-417). Springer Netherlands.

Clayman, S., Mamatas, L., & Galis, A. (2016, April). *Efficient management solutions for software-defined infrastructures.* In Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP (pp. 1291-1296). IEEE.

OpenFlow Switch Consortium. (2009). OpenFlow Switch Specification Version 1.0. 0.

Costa, V. T., & Costa, L. H. M. (2015). *Vulnerabilities and solutions for isolation in FlowVisor-based virtual network environments*. Journal of Internet Services and Applications, 6(1), 18.

Dainotti, A., Pescape, A., & Claffy, K. C. (2012). *Issues and future directions in traffic classification*. IEEE network, 26(1), 35-40.

Denton, J. (2014). *Learning OpenStack Networking (Neutron)*: Packt Publishing Ltd.

Devi, D. C., & Uthariaraj, V. R. (2016). Load Balancing in Cloud Computing Environment Using Improved Weighted Round Robin Algorithm for Nonpreemptive Dependent Tasks. *The Scientific World Journal, 2016*.

Ding, W., Qi, W., Wang, J., & Chen, B. (2015). *OpenSCaaS: an open service chain as a service platform toward the integration of SDN and NFV*. Network, IEEE, 29(3), 30-35.

Doron, E., & Sekiguchi, M. (2016). *U.S. Patent No. 9,386,085*. Washington, DC: U.S. Patent and Trademark Office.

Drutskoy, D., Keller, E., & Rexford, J. (2013). *Scalable network virtualization in software-defined networks*. Internet Computing, IEEE, 17(2), 20-27.

Dusi, M., Bifulco, R., Gringoli, F., & Schneider, F. (2014, August). *Reactive logic in software-defined networking: Measuring flow-table requirements*. In Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International (pp. 340-345). IEEE.

Dutta, P. P., Vidovic, N., & Vrsalovic, D. F. (2003). *U.S. Patent No. 6,546,423*. Washington, DC: U.S. Patent and Trademark Office.

El-Azzab, M., Bedhiaf, I. L., Lemieux, Y., & Cherkaoui, O. (2011, November). *Slices isolator for a virtualized OpenFlow node*. In Network Cloud Computing and Applications (NCCA), 2011 First International Symposium on (pp. 121-126). IEEE.

Estan, C., Keys, K., Moore, D., & Varghese, G. (2004). *Building a better NetFlow*. ACM SIGCOMM Computer Communication Review, 34(4), 245-256.

Feamster, N., Rexford, J., & Zegura, E. (2013). *The road to SDN*. *Queue*, *11*(12), 20.

Feamster, N., Rexford, J., & Zegura, E. (2014). *The road to SDN: an intellectual history of programmable networks*. ACM SIGCOMM Computer Communication Review, 44(2), 87-98.

Fortis, T. F., Munteanu, V. I., & Negru, V. (2012, June). *Towards a service friendly cloud ecosystem*. In Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on (pp. 172-179). IEEE.

Frey, C. A., Bicket, J., Herbert, K. P., Malhotra, V. S., & Chambers, B. A. (2016). *Methods for exchanging network management messages using udp over http protocol*: US Patent 20,160,094,688.

Fundation, O. N. (2012). S*oftware-defined networking: The new norm for networks*. ONF White Paper2, 2-6.

Gandhi, R., Liu, H. H., Hu, Y. C., Lu, G., Padhye, J., Yuan, L., & Zhang, M. (2015). *Duet: Cloud scale load balancing with hardware and software*. ACM SIGCOMM Computer Communication Review, 44(4), 27-38.

Ghaffarinejad, A. (2015). *Comparing a Commercial and an SDN-Based Load Balancer in a Campus Network.* Arizona State University.

Gill, J. (2014). *Bayesian methods: A social and behavioral sciences approach* (Vol. 20): CRC press.

Gilly, K., Juiz, C., & Puigjaner, R. (2011). *An up-to-date survey in web load balancing.* World Wide Web, 14(2), 105-131.

Godfrey, P. B., & Stoica, I. (2005, March). *Heterogeneity and load balance in distributed hash tables*. In *INFOCOM 2005.* 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE (Vol. 1, pp. 596-606). IEEE.

Gomez, L. (2013). A Brief Introduction to SDN and OpenDaylight.

González-Vélez, H., & Cole, M. (2010). *Adaptive structured parallelism for distributed heterogeneous architectures: A methodological approach with pipelines and farms*. Concurrency and Computation: Practice and Experience, 22(15), 2073-2094.

Gonzalez, J., Rojas, H., Ortega, J., & Prieto, A. (2002). *A new clustering technique for function approximation*. Neural Networks, IEEE Transactions on, 13(1), 132-142.

Gosling, J. (2000). *The Java language specification*: Addison-Wesley Professional.

Govindraj, S., Jayaraman, A., Khanna, N., & Prakash, K. R. (2012). Openflow: Load balancing in enterprise networks using floodlight controller. *The university of Colorado*.

Gulbrandsen, A., & Esibov, L. (2000). A DNS RR for specifying the location of services (DNS SRV).

Guo, Z., Su, M., Xu, Y., Duan, Z., Wang, L., Hui, S., & Chao, H. J. (2014). *Improving the performance of load balancing in software-defined networks through load variance-based synchronization.* Computer Networks, 68, 95-109.

Haleplidis, E., Pentikousis, K., Denazis, S., Salim, J. H., Meyer, D., & Koufopavlou, O. (2015). *Software-defined networking (SDN): Layers and architecture terminology* (No. RFC 7426).

Handigol, N., Flajslik, M., Seetharaman, S., McKeown, N., & Johari, R. (2010, November). *Aster* x: Load-balancing as a network primitive*. In 9th GENI Engineering Conference (Plenary) (pp. 1-2).

Handigol, N., Seetharaman, S., Flajslik, M., McKeown, N., & Johari, R. (2009). *Plug-n-Serve: Load-balancing web traffic using OpenFlow*. ACM Sigcomm Demo, 4(5), 6.

Hsu, I. P. S., Cheung, D. C. Y., & Jalan, R. R. (2013). *U.S. Patent No. 8,504,721*. Washington, DC: U.S. Patent and Trademark Office.

Huh, J.-H., & Seo, K. (2016). *Design and test bed experiments of server operation system using virtualization technology*. Human-centric Computing and Information Sciences, 6(1), 1.

Ivancic, Franjo, Cristian Lumezanu, Gogul Balakrishnan, Willard Dennis, and Aarti Gupta. "*Network Testing*." U.S. Patent Application 14/270,445, filed May 6, 2014.

Jian Liu, Lei Xu, Weiming Zhang, A load balancing algorithm based on dynamic feedback, Computer Engineering and Science, 25 (2003), 65-68

Jain, R., & Paul, S. (2013). *Network virtualization and software defined networking for cloud computing: a survey*. Communications Magazine, IEEE, 51(11), 24-31.

Jarschel, M., Zinner, T., Hoßfeld, T., Tran-Gia, P., & Kellerer, W. (2014a). I*nterfaces, attributes, and use cases: A compass for SDN*. Communications Magazine, IEEE, 52(6), 210-217.

Jarschel, M., Zinner, T., Hoßfeld, T., Tran-Gia, P., & Kellerer, W. (2014b). *Interfaces, attributes, and use cases: A compass for SDN*. *I*EEE Communications Magazine, 52(6), 210-217.

Jethanandani, M., Bashyam, M., Bagepalli, N., & Patra, A. (2006). *U.S. Patent Application No. 11/383,093*.

Jindal, A., Lim, S. B., Radia, S., & Chang, W. L. (2001). *U.S. Patent No. 6,324,580*. Washington, DC: U.S. Patent and Trademark Office.

John, W., Pentikousis, K., Agapiou, G., Jacob, E., Kind, M., Manzalini, A., ... & Meirosu, C. (2013, November). *Research directions in network service chaining*. In Future Networks and Services (SDN4FNS), 2013 IEEE SDN for (pp. 1-7). IEEE.

Ju, J., Xu, G., & Yang, K. (1995). *An intelligent dynamic load balancer for workstation clusters*. ACM SIGOPS Operating Systems Review, 29(1), 7-16.

Jung, J. J. (2011). Service chain-based business alliance formation in service-oriented architecture. *Expert Systems with Applications*, *38*(3), 2206-2211.

Karacali, B., & Tracey, J. M. (2016, April). *Experiences evaluating openstack network data plane performance and scalability*. In Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP (pp. 901-906). IEEE.

Kashiri, N., Tsagarakis, N. G., Van Damme, M., Vanderborght, B., & Caldwell, D. G. (2016). *Proxy-Based Sliding Mode Control of Compliant Joint Manipulators*Informatics in Control, Automation and Robotics (pp. 241-257): Springer.

Kaur, S., Kumar, K., Singh, J., & Ghumman, N. S. (2015, March). *Round-robin based load balancing in Software Defined Networking*. In Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on (pp. 2136-2139). IEEE.

Kerravala, Z. (2013). *The Software-Defined Data Center is Key to IT-as-a-Service*. Cell, 301, 775-7447.

Keti, F., & Askar, S. (2015, February). *Emulation of Software Defined Networks Using Mininet in Different Simulation Environments.In Intelligent Systems, Modelling and Simulation (ISMS), 2015 6th International Conference on (pp. 205-210). IEEE.*

Khattak, Z. K., Awais, M., & Iqbal, A. (2014, December). *Performance evaluation of OpenDaylight SDN controller*. In Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on (pp. 671-676). IEEE..

Khondoker, R., Zaalouk, A., Marx, R., & Bayarou, K. (2014, January). *Feature-based comparison and selection of Software Defined Networking (SDN) controllers*. In Computer Applications and Information Systems (WCCAIS), 2014 World Congress on (pp. 1-7). IEEE.

Kim, E. D., Lee, S. I., Choi, Y., Shin, M. K., & Kim, H. J. (2014, February). A *flow entry management scheme for reducing controller overhead*. In Advanced Communication Technology (ICACT), 2014 16th International Conference on (pp. 754-757). IEEE.

Kim, H., Reich, J., Gupta, A., Shahbaz, M., Feamster, N., & Clark, R. J. (2015, May). Kinetic: *Verifiable Dynamic Network Control*. In NSDI (pp. 59-72).

Koerner, M., & Kao, O. (2012, June). *Multiple service load-balancing with OpenFlow.* In High Performance Switching and Routing (HPSR), 2012 IEEE 13th International Conference on (pp. 210-214). IEEE.

Koerner, M., & Kao, O. (2013, October). *Optimizing openflow load-balancing with l2 direct server return*. In Network of the Future (NOF), 2013 Fourth International Conference on the (pp. 1-5). IEEE.

Kopparapu, C. (2002). *Load balancing servers, firewalls, and caches*: John Wiley & Sons.

Koushika, A. M., & Selvi, S. T. (2014, April). *Load valancing Using Software Defined Networking in cloud environment*. In Recent Trends in Information Technology (ICRTIT), 2014 International Conference on (pp. 1-8). IEEE.

Krebs, R., Momm, C., & Kounev, S. (2012). *Architectural Concerns in Multi-tenant SaaS Applications.* CLOSER, 12, 426-431.

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

Kuźniar, M., Perešíni, P., & Kostić, D. (2015, March). *What you need to know about SDN flow tables*. In International Conference on Passive and Active Network Measurement (pp. 347-359). Springer International Publishing..

Kwak, B., & Jung, H. (2015). Virtualized Testbed Development using Openstack.

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

Lee, Y. J., & Riley, G. F. (2005, March). *A workload-based adaptive load-balancing technique for mobile ad hoc networks*. In Wireless communications and networking conference, 2005 IEEE (Vol. 4, pp. 2002-2007). IEEE.

Leland, R., & Hendrickson, B. (1994, May). *An empirical study of static load balancing algorithms*. In Scalable High-Performance Computing Conference, 1994., Proceedings of the (pp. 682-685). IEEE.

Li, L. E., & Woo, T. (2011). *U.S. Patent Application No. 12/571,271*.

Li, Y., & Brodlie, K. (2003, December). *Soft Object Modelling with Generalised ChainMail—Extending the Boundaries of Web-based Graphics*. In Computer Graphics Forum (Vol. 22, No. 4, pp. 717-727). Blackwell Publishing.

LIN, H.-j., PENG, H., & LI, J. (2007). *Design and realization of tendency load-balance for application server* [J]. Computer Engineering and Design, 14, 034.

Lin, P., Hart, J., Krishnaswamy, U., Murakami, T., Kobayashi, M., Al-Shabibi, A., ... & Bi, J. (2013, August). *Seamless interworking of SDN and IP*. In ACM SIGCOMM computer communication review (Vol. 43, No. 4, pp. 475-476). ACM.

Load-Balancer, A. https://saas.hpe.com/marketplace/sdn/aricent-sdn-load-balancer-application. *[accessed: 28/06/2016]*.

MacDonald, D., & Lowekamp, B. (2010). *NAT behavior discovery using session traversal utilities for NAT* (STUN) (No. RFC 5780).

Maltz, D. A., Greenberg, A. G., Patel, P. K., Sengupta, S., & Lahiri, P. (2012). *U.S. Patent No. 8,160,063*. Washington, DC: U.S. Patent and Trademark Office.

Matias, J., Garay, J., Toledo, N., Unzilla, J., & Jacob, E. (2015). *Toward an SDN-enabled NFV architecture*. IEEE Communications Magazine, 53(4), 187-193.

Mayoral, A., Vilalta, R., Munoz, R., Casellas, R., & Martánez, R. (2015, July). *Performance analysis of SDN orchestration in the cloud computing platform and transport network of the ADRENALINE testbed*. In Transparent Optical Networks (ICTON), 2015 17th International Conference on (pp. 1-4). IEEE.

Mccauley, J. (2014). Pox: A python-based openflow controller.

McClain, C. B., & Thatcher, J. E. (2004). *U.S. Patent No. 6,772,214*. Washington, DC: U.S. Patent and Trademark Office.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., . . . Turner, J. (2008). *OpenFlow: enabling innovation in campus networks*. ACM SIGCOMM Computer Communication Review, 38(2), 69-74.

Medved, J., Varga, R., Tkacik, A., & Gray, K. (2014, June). *Opendaylight: Towards a model-driven sdn controller architecture*. In A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on (pp. 1-6). IEEE.

Mininet,http://mininet.org/, *accessed: 28/06/2016]*.

Mockapetris, P., & Dunlap, K. J. (1988). *Development of the domain name system* (Vol. 18, No. 4, pp. 123-133). ACM.

Myers, A. C. (1999, January). J*Flow: Practical mostly-static information flow control*. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (pp. 228-241). ACM.

Narendran, B., Rangarajan, S., & Yajnik, S. (2000). *U.S. Patent No. 6,070,191*. Washington, DC: U.S. Patent and Trademark Office.

Nguyen, T. T., & Armitage, G. (2008). A *survey of techniques for internet traffic classification using machine learning*. IEEE Communications Surveys & Tutorials, 10(4), 56-76.

Nuaimi Al, K., Mohamed, N., Al Nuaimi, M., & Al-Jaroodi, J. (2012, December). *A survey of load balancing in cloud computing: Challenges and algorithms*. In Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on (pp. 137-142). IEEE.

Nunes, B. A. A., Mendonca, M., Nguyen, X.-N., Obraczka, K., & Turletti, T. (2014). *A survey of software-defined networking: Past, present, and future of programmable networks*. *IEEE* Communications Surveys & Tutorials, 16(3), 1617-1634.

O'Neil, Kevin, Robert Nerz, and Robert Aubin. "*System for balancing loads among network servers*." U.S. Patent Application No. 10/162,419.

Okamoto, K. (2001). *U.S. Patent Application No. 09/900,891*.

Okano, T., Ochi, A., Mochizuki, T., & Takaba, K. (2004). *Load balancing system*: Google Patents.

Ortiz, J., Londoño, J., & Novillo, F. (2016, October). *Evaluation of performance and scalability of Mininet in scenarios with large data centers*. In Ecuador Technical Chapters Meeting (ETCM), IEEE (Vol. 1, pp. 1-6). IEEE.

OVS, D. i.,http://kspviswa.github.io/dpi-enabled-ovs/,*[accessed: 28/07/2016]*.

Patel, P., Bansal, D., Yuan, L., Murthy, A., Greenberg, A., Maltz, D. A., . . . Wu, H. (2013). *Ananta: cloud scale load balancing*. ACM SIGCOMM Computer Communication Review, 43(4), 207-218.

Patel, P., Ranabahu, A. H., & Sheth, A. P. (2009). Service level agreement in cloud computing.

Pfaff, B., Pettit, J., Koponen, T., Jackson, E. J., Zhou, A., Rajahalme, J., ... & Amidon, K. (2015, May). *The Design and Implementation of Open vSwitch*. In *NSDI* (pp. 117-130).

Phemius, K., & Bouet, M. (2013, October). *Monitoring latency with openflow*. In Network and Service Management (CNSM), 2013 9th International Conference on (pp. 122-125). IEEE.

Poddar, R., Vishnoi, A., & Mann, V. (2015, January). *HAVEN: Holistic load balancing and auto scaling in the cloud*. In Communication Systems and Networks (COMSNETS), 2015 7th International Conference on (pp. 1-8). IEEE.

Qilin, M., & Weikang, S. (2015, June). A *Load Balancing Method Based on SDN.* In Measuring Technology and Mechatronics Automation (ICMTMA), 2015 Seventh International Conference on (pp. 18-21). IEEE.

Qi Zheng, Gguangping Zhou, Content classification load balancing algorithm in cluster, Computer Systems and Applications, 20 (2011), 47-50

Qosmos.,http://www.qosmos.com/sdn-nfv/dpi-module-for-vswitch/,*[accessed: 22/08/2016]*.

Radojević, B., & Žagar, M. (2011, May). *Analysis of issues with load balancing algorithms in hosted (cloud) environments*. In MIPRO, 2011 Proceedings of the 34th International Convention (pp. 416-420). IEEE.

Ragalatha P, M. C., Sundeep Kumar. K. (2013). *Design and Implementation of Dynamic load balancer on OpenFlow enabled SDNs*. IOSR Journal of Engineering, 3(8), 32-41.

Rahman, M., Iqbal, S., & Gao, J. (2014, April). *Load balancer as a service in cloud computing*. In Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on (pp. 204-211). IEEE.

Randles, M., Lamb, D., & Taleb-Bendiab, A. (2010, April). *A comparative study into distributed load balancing algorithms for cloud computing.* In Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on (pp. 551-556). IEEE.

Research, G. V, http://www.grandviewresearch.com/industry-analysis/software-defined-networking-sdn-market-analysis,*accessed: 28/06/2016]*.

Research, I,http://www.infonetics.com/pr/2016/Carrier-SDN-Market-Highlights.asp,*[accessed: 27/06/2016]*.

Ros, F. J., & Ruiz, P. M. (2014, August). *Five nines of southbound reliability in software-defined networks.* In Proceedings of the third workshop on Hot topics in software defined networking (pp. 31-36). ACM.

Saito, Y., Bershad, B. N., & Levy, H. M. (2000). *Manageability, availability, and performance in porcupine: a highly scalable, cluster-based mail service.*ACM Transactions on Computer Systems (TOCS), 18(3), 298.

Salchow Jr, K. (2007). Load Balancing 101: T*he Evolution to Application Delivery Controllers.F5 White Paper*.

Schemers, R. (1995, September). lbnamed: *A Load Balancing Name Server in Perl*. In LISA (pp. 1-12).

Schmid, S., & Suomela, J. (2013, August). *Exploiting locality in distributed SDN control.* In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (pp. 121-126). ACM.

Sefraoui, O., Aissaoui, M., & Eleuldj, M. (2012). *OpenStack: toward an open-source solution for cloud computing*. International Journal of Computer Applications, 55(3).

Shabtay, L. (2010). Dynamic load balancer: Google Patents.

Shahmir Shourmasti, K. (2013). Stochastic Switching Using OpenFlow.

Shang, Z., Chen, W., Ma, Q., & Wu, B. (2013, November). *Design and implementation of server cluster dynamic load balancing based on OpenFlow*. I*n Awareness Science and Technology and Ubi-Media Computing (iCAST-UMEDIA), 2013 International Joint Conference on (pp. 691-697). IEEE.*

Sherwood, R., Gibb, G., Yap, K. K., Appenzeller, G., Casado, M., McKeown, N., & Parulkar, G. (2009). *Flowvisor: A network virtualization layer. OpenFlow Switch Consortium, T*ech. Rep, 1-13.

Shin, M. K., Nam, K. H., & Kim, H. J. (2012, October). *Software-defined networking (SDN): A reference architecture and open APIs*. In ICT Convergence (ICTC), 2012 International Conference on (pp. 360-361). IEEE.

Shukla, V. S. (2015, March). *SDN transport architecture and challenges*. In Optical Fiber Communications Conference and Exhibition (OFC), 2015 (pp. 1-3). IEEE.

Singh, A., Goyal, P., & Batra, S. (2010). *An optimized round robin scheduling algorithm for CPU scheduling*. IJCSE) International Journal on Computer Science and Engineering, 2(07), 2383-2385.

Soysal, M., & Schmidt, E. G. (2010). *Machine learning algorithms for accurate flow-based network traffic classification: Evaluation and comparison*. Performance Evaluation, 67(6), 451-467.

Specifiation, O. S. (2014). v1. 5, *Open Network Foundation*, September 27, 2013.

Specification, O. S. (2013). Version 1.4. 0, October 14, 2013.

Srisuresh, P., & Egevang, K. (2001). *Traditional IP network address translator (Traditional NAT):* RFC 3022, January.

Stroustrup, B. (1986). *The C++ programming language*: Pearson Education India.

Suñé, M., Bergesio, L., Woesner, H., Rothe, T., Köpsel, A., Colle, D., . . . Channegowda, M. (2014). *Design and implementation of the OFELIA FP7 facility: The European OpenFlow testbed*. Computer Networks, 61, 132-150.

Surya Prateek, S., & Ying, Q. (2013). *Cloud Server with OpenFlow: Load Balancing.* Paper presented at the 1st International Workshop on Cloud Computing and Information Security.

Tarreau, W. (2012). *HAProxy-the reliable, high-performance TCP/HTTP load balancer*.

Tavakoli, A., Casado, M., Koponen, T., & Shenker, S. (2009). *Applying NOX to the Datacenter.* Paper presented at the HotNets.

Tian, W., Zhao, Y., Zhong, Y., Xu, M., & Jing, C. (2011, September). *A dynamic and integrated load-balancing scheduling algorithm for Cloud datacenters. In Cloud Computing and Intelligence Systems (*CCIS), 2011 IEEE International Conference on (pp. 311-315). IEEE.

Tootoonchian, A., Gorbunov, S., Ganjali, Y., Casado, M., & Sherwood, R. (2012). *On Controller Performance in Software-Defined Networks*. Hot-ICE, 12, 1-6.

Tourrilhes, J., Sharma, P., Banerjee, S., & Pettit, J. (2014). T*he Evolution of SDN and OpenFlow: A Standards Perspective*. IEEE Computer Society, 47(11), 22-29.

Tsai, W. T., Sun, X., Shao, Q., & Qi, G. (2010, November). *Two-tier multi-tenancy scaling and load balancing*. In e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on (pp. 484-489). IEEE.

Uppal, H., & Brandon, D. (2010). *OpenFlow based load balancing*. CSE561: Networking Project Report, University of Washington.

Urgaonkar, R., Kozat, U. C., Igarashi, K., & Neely, M. J. (2010, April). *Dynamic resource allocation and power management in virtualized data centers*. In Network Operations and Management Symposium (NOMS), 2010 IEEE (pp. 479-486). IEEE.

Valenti, S., Rossi, D., Dainotti, A., Pescapè, A., Finamore, A., & Mellia, M. (2013). *Reviewing traffic classification Data Traffic Monitoring and Analysis* (pp. 123-147): Springer.

Velte, A., & Velte, T. (2009). *Microsoft virtualization with Hyper-V*: McGraw-Hill, Inc.

Wang, M., Li, B., & Li, Z. (2004). *sFlow: Towards resource-efficient and agile service federation in service overlay networks*. In Distributed Computing Systems, 2004. Proceedings. 24th International Conference on (pp. 628-635). IEEE.

Wang, P., Lan, J., & Chen, S. (2014). *OpenFlow based flow slice load balancing. Communications*, China, 11(12), 72-82.

Wang, P., Sahinoglu, Z., Pun, M.-O., Li, H., & Himed, B. (2011). *Knowledge-aided adaptive coherence estimator in stochastic partially homogeneous environments*. Signal Processing Letters, IEEE, 18(3), 193-196.

Wang, R., Butnariu, D., & Rexford, J. (2011). O*penFlow-Based Server Load Balancing GoneWild. Hot-ICE, 11, 12-12.*

Wellman, B. (2004). *The three ages of internet studies: ten, five and zero years ago*. *New media* & society, 6(1), 123-129.

Williams, A. (2015). *A comparison of the performance and scalability of relational and document-based web-systems for large scale applications in a rehabilitation context.* arXiv preprint arXiv:1510.00216.

Wu, J., Huang, Y., Kong, J., Tang, Q., & Huang, X. (2015). *A Study on the Dependability of Software Defined Networks.* Paper presented at the International Conference on Materials Engineering and Information Technology Applications (MEITA 2015).

Yang, L.-H., & Yu, S.-S. (2003). *A variable weighted least-connection algorithm for multimedia transmission*. Journal of Shanghai University (English Edition), 7(3), 256-260.

Yilmaz, S., Tekalp, A. M., & Unluturk, B. D. (2015, February). *Video streaming over software defined networks with server load balancing*. In *Computing, Networking and Communications (ICNC), 2015 International Conference on* (pp. 722-726). IEEE.

Yong, W., Xiaoling, T., Qian, H., & Yuwen, K. (2016). *A dynamic load balancing method of cloud-center based on SDN*.China Communications, 13(2), 130-137.

Zhang, H., & Guo, X. (2014, November). S*DN-based load balancing strategy for server cluster.In Cloud Computing and Intelligence Systems (CCIS), 2014 IEEE 3rd International Conference on* (pp. 662-667). IEEE.

Zhang, W., Jin, S., & Wu, Q. (1999). *Creating Linux virtual servers.* Paper presented at the LinuxExpo 1999 Conference.

Zhou, W., Li, L., Luo, M., & Chou, W. (2014, May). *REST API design patterns for SDN northbound API.* In *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on* (pp. 358-365). IEEE.

LIST OF PUBLICATIONS AND PAPERS PRESENTED

ISI Journal paper

Ahmed Abdelaziz, TF Ang, M Sookhak, S Khan, A Vasilakos, CS Liew, Survey on Network Virtualization Using OpenFlow: Taxonomy, Opportunities, and Open Issues (2016) KSII Transactions on Internet and Information Systems 10 (10), 4902-4932

Suleman khan, Abdullah Gani, Ainuddin Wahid Abdul Wahab, **Ahmed Abdelaziz** et. al, Towards an Applicability of Current Network Forensics for Cloud Networks: A SWOT Analysis, IEEE Access, Accepted 09 November 2016

Suleman khan, Abdullah Gani, Ainuddin Wahid Abdul Wahab, **Ahmed Abdelaziz**, et. al, Software-defined Network Forensics: Motivation, Potential Locations, Requirements, and Challenges, IEEE Networks, Accepted 23 August 2016

Thomas, Bimba Andrew, Ahmed abdelaziz et al. "Towards Knowledge Modeling and Manipulation Technologies: A Survey." International Journal of Information Management (2016).

Akhunzada, A., Gani, A., Anuar, N. B., **Ahmed Abdelaziz**. , Khan, M. K., Hayat, A., & Khan, S. U. (2016). Secure and dependable software defined networks. Journal of Network and Computer Applications , 61, 199-221.

Conference paper:

Ahmed Abdelaziz, Ang Tang Fong, "Application-Aware load balance system in cloud using SDN," 4th International Conference on Computer Science and Computational Mathematics, ICCSCM 2015, Malaysia ,Langkawi

Suliman Kan , Ahmed Abdelaiz , FML: A novel Forensics Management Layer for Software Defined Networks" , Confluence 2016 Conference-India 2016