DISTRIBUTED LOAD BALANCING ORCHESTRATION USING SOFTWARE DEFINED NETWORKS (SDN) TECHNIQUE

by

Robert Muriithi Muna

A DISSERTATION

Submitted to

KCA UNIVERSITY

in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE IN DATA COMMUNICATION AND NETWORKS

in the

FACULTY OF COMPUTING AND INFORMATION MANAGEMENT

October 2016

Deed of Declaration

I hereby certify that this dissertation constitutes my own work, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Student Name: **Robert Muriithi Muna** Registration Number: **14/04135**

Sign: ________________________ Date: ______________________

I do hereby confirm that I have examined the master's dissertation of

Robert Muriithi Muna

and have certified that all revisions that the dissertation panel and examiners recommended have been adequately addressed

Sign: ________________________ Date: ______________________

Dr. Alice Njuguna

Dissertation Supervisor

ACKNOWLEDGEMENTS

I wish to thank my research supervisor Dr. Alice Njuguna of the Faculty of Computing and Information Management at KCA University. When I came across challenges in my research, Dr. Njuguna's office was always open. She steered me in the right direction by making sure that I did not go off course. She also consistently ensured that this research was my own work.

I would like also to thank my research co-supervisor, Mr. Peter Ngugi for his expertise in validating my research and his valuable comments and insightful criticism on this research. I am gratefully indebted to his unfailing support.

DEDICATION

I wish to dedicate this research to my dear wife and children for their support and continuous encouragement throughout my studies and research. This accomplishment would not have been possible were it not for their support and understanding. To that, I say, Thank You.

Abstract

Load balancing has been a challenge in large scale networks that process a lot of traffic and more so if access to the data is centralized. In best practice, user requests from multiple locations would require to have the same response time. The problem comes in if the requests are multimedia in nature and require quality guarantees. This causes a strain in the management of the server workloads as well as resources such as memory.

Traditional networks made use of centralized load balancers which were responsible for making decision as well as passing traffic so as to take care of the requests. However, with this kind of load balancers, the problem of managing very large networks whose traffic was unpredictable become obvious. Coupled with the fact that the load balancer performance could not be guaranteed, this necessitated the need for an alternative which came in form of Software Defined Networks (SDN).

SDN decoupled the decision making element from the traffic passing element such that network appliances could only pass traffic whereas the decision making was done by the controller. The communication between the appliances and the controllers was facilitated by the OpenFlow protocol standard. Initially the controllers were centralized and suffered the same fate that befell the traditional load balancers in regards to the management of network traffic. With distributed controllers, the traffic could be shared amongst the controllers and ease the workload. The distributed controllers also brought with them the problem of synchronisation of network amongst controllers.

This dissertation identifies the challenges brought about by having distributed controllers and comes up with an algorithm to meet these challenges.

Keywords: Openflow, OpenDayLight, Mininet, Iperf, VirtualBox, Controllers, Flow tables, data planes, control planes, Southbound Interface, Northbound Interface.

Table of Contents

Deed of Declaration i
ACKNOWLEDGEMENTS ii
DEDICATION iii
Abstract iv
Table of Contents v
Nomenclature 1
List of Figures 2
List of Tables 3
Abbreviations 4
Definition of Terms 5
CHAPTER ONE 7
1. Introduction 7
1.1 Background of the Problem 8
1.2 The Problem Statement 10
1.3 Aim 13
1.4 Specific Objectives 13
1.5 Significance of the Study 13
1.6 Scope and Limitation of the Study 14
CHAPTER TWO: Literature Review 15
2.1 State of Art 15
2.1.1 SDN 15
2.1.2 Openflow 17
2.1.3 Load balancing 18
2.2 State of Practice 21
2.3 Technological Development 23
2.3.1 DNS load balancing 23
2.3.2 ECMP load balancing 23
2.3.3 Dynamic Load Balancing 24
2.4 Critique of the Literature 24
2.5 Evaluation of the Current Methods 25
2.6 Conclusion 26
CHAPTER THREE 27
3.0 Introduction 27
3.1 Research Methodology 27
3.2 Tools to be adopted 28
3.3 Conceptual Design 30
3.3.1 Conceptual model 30
3.3.2 Flow Chart 32
3.4 Implementation Framework 34
3.4.1 Implementation Design 34
3.4.2 Framework Components 34
3.5 Tests to be done 35
CHAPTER FOUR: Implementation and Testing 36
4.1 Introduction 36
4.2 Hardware Requirement 36
4.3 Software Requirements 36
4.4 System Requirements 36
4.4.1 Installations and Configurations 37
4.5 Implementation of the algorithm 39
4.5.1 Variables 39

4.5.2 Algorithm 40
4.5.3 Algorithm Pseudocode 42
4.5.4 Mininet deployment 45
4.5.5 OpenDayLight 46
4.6 Data Analysis methods 48
CHAPTER FIVE: Conclusion and Future works 56
5.1 Conclusion 56
5.2 Suggestions for Further Studies 56
References 58

Nomenclature

Symbol	Meaning
∈	element of
∉	not element of
≠	not equal to
<=	less than or equal to
&&	and
∴	therefore

List of Figures

Figure 1: SDN Architecture. (Zhong, et al., 2015) ... 15
Figure 2: The OpenFlow flow entry (Trajano & Fernandez, 2016) ... 18
Figure 3: Conceptual Model ... 32
Figure 4: Flow Chart ... 33
Figure 5: Implementation design ... 34
Figure 6: Network adapter configuration ... 37
Figure 7: Identifying the Mininet IP address ... 38
Figure 8: OpenDayLight web interface ... 39
Figure 9: Distributed Controllers with switches and hosts ... 47
Figure 10: Iperf server node ... 48
Figure 11: Iperf Client node ... 48
Figure 12: Measuring bandwidth using a single controller ... 49
Figure 13: Measuring bandwidth using distributed controllers ... 50
Figure 14: Performance evaluation ... 51
Figure 15: Multiple threads on a single controller ... 52
Figure 16: Multiple threads on distributed controllers ... 53

List of Tables

Table 1: Distributed controllers components 34
Table 2: Comparative analysis 49
Table 3: Strain test analysis 52

Abbreviations

DNS	-	Domain Name Server
SDN	-	Software Defined Network
API	-	Application Programming Interface
AN	-	Active networks
PN	-	Programmable Networks
IP	-	Internet Protocol
CPU	-	Central Processing Unit
RAM	-	Random Access Memory
SSH	-	Secure Socket Shell

Definition of Terms

Software-Defined Networking (SDN): It is a technology in the field of computer networking that originated from a project that began at UC Berkeley and Stanford University around 2008. SDN allows network researchers to manage network services through abstraction of lower level functionality. This is achieved by decoupling the network control that makes decisions about where traffic is sent (the control plane) from the forwarding systems that forward traffic to the selected destination (the data plane) (Zhong, et al., 2015).

Southbound Interfaces: Southbound interfaces (or southbound APIs) are the connecting bridges between control and forwarding elements, thus being the crucial instrument for clearly separating control and data plane functionality. However, these APIs are still tightly tied to the forwarding elements of the underlying physical or virtual infrastructure (Kreutz, et al., 2015).

OpenFlow: A computer networking protocol that can be run on generic hardware, software and vendor specific devices. This protocol allows separating the control and data paths from the same device to different devices, thus giving more control over the flow of data through the network (Govindraj, et al., 2012).

Controller: A software program that is responsible for populating and manipulating the flow tables of the switches. By insertion, modification and removal of flow entries, the controller can modify the behavior of the switches with regard to forwarding (Braun & Menth, 2014).

Bandwidth: Bandwidth represents the capacity of the connection. The greater the capacity, the more likely that greater performance will follow, though overall performance also depends

on other factors, such as latency. In general, network bandwidth is a bit rate measure of available or consumed data communication resources expressed in bits/second (Mahanta, et al., 2013).

Data Planes: Represents the networking devices which are responsible for forwarding data (Kreutz, et al., 2015).

Control Planes: Represents the protocols used to populate the forwarding tables of the data plane elements (Kreutz, et al., 2015).

BGP: Border Gateway Protocol (BGP) is an interdomain routing protocol designed to provide loop-free routing between separate routing domains that contain independent routing policies (autonomous systems) (Cisco, 2011).

OSPF: Open Shortest Path First (OSPF) is a routing protocol developed for Internet Protocol (IP) networks by the interior gateway protocol (IGP) working group of the Internet Engineering Task Force (IETF). OSPF was created because the Routing Information Protocol (RIP) was, in the mid-1980s, increasingly unable to serve large, heterogeneous internetworks (Cisco, 2011).

CHAPTER ONE

1. Introduction

In an enterprise-scale network, the servers process a lot of user requests of which the expectations from the users is to have prompt response. However, with increase in the number of users and requests from multiple geographical locations, processing these requests becomes a challenge. For better management of the network traffic, load balancing is important.

Traditional networks relied on having dedicated hardware-based load balancers. The load balancers were tightly coupled in that the data planes (responsible for forwarding traffic) and the control planes (responsible for deciding on how to handle network traffic) were bundled in the appliance. This meant that the load balancer were not flexible to the changing network dynamics and thus could easily have become a single point of failure within the enterprise network (Senthil & Ranjani, 2015; Kreutz, et al., 2015).

The emergence of Software Defined Networks (SDN) made it possible to decouple the control planes from the data planes in that the control logic that was embedded in the hardware appliances was separated and the appliances became forwarding elements (Kreutz, et al., 2015). Through SDN controller, which had a global view of the network, the network became more agile and responsive to dynamic changes necessitated by the changing network demands.

However, since decisions in SDN were still being orchestrated by a single central controller, with the growth in the number of switches in the network, the need to scale the network as requests were being processed became evident. Coupled with time delays in the

establishment of flow tables, limitation in network bandwidth, and the processing constraints of the controller, there was need to distribute the implementation of the controller (Yao, et al., 2015).

According to Zhong, et al. (2015), “load balancing aims to optimize the utilization of the resource by maximizing the throughput, minimizing the response time, and avoiding overloading of any single resource.” The purpose for this research is to utilize the capabilities of SDN architecture in developing a distributed load balancing algorithm so as to orchestrate the scheduling of traffic in an enterprise-scale network.

1.1 Background of the Problem

The need to decouple the hardware and software components of devices or appliances did not start with the emergence of SDN. It pre-dates to the use of early telephony networks which separated the control and data planes so as to enable the deployment of new services and to simplify the management of the network. Through this separation, it became easier to deploy services on demand; promote innovation through the introduction of new services; as well as determine status of a circuit beforehand (Feamster, et al., 2013).

In the 1990, there was increasing need to have communication network programmable. Active networks (AN) and Programmable Networks (PN) provided a means in which programs could be executed on data packets. With AN, new routing mechanisms and network services could be implemented without modifying the hardware. AN used switches to perform custom computations on packets as the packets travelled through the switches. Unlike the AN, with PN, programs were installed inside the network nodes, which executed the programs on the packets. The motivation behind programmability of the network was to accelerate innovations; which is similar to the motivation behind SDN (Braun & Menth,

2014).

The advancement towards multiple programming models led to research on network virtualization. AN produced an architectural framework that allowed resource isolation in Central Processing Unit (CPU), memory, bandwidth as well as forwarding tables of which virtualization would depend upon. Network virtualization represented one or more logical network topologies on top of the same underlying infrastructure. Its main benefit was sharing. With sharing one could instantiate multiple logical routers and virtual networks on the same physical infrastructure. Users of a virtual network could get the view of their own logical network and topology that was separate from other logical networks that could be running on the same underlying physical infrastructure (Feamster, et al., 2013; Braun & Menth, 2014).

The programmability of the networks led to research on a project that began at UC Berkeley and Stanford University around 2008 and from where SDN evolved. The generic hardware could therefore be programmed using an Application Programmable Interface (API) or installation of any Network Operating System (NOS). SDN provided for the abstraction of lower level network functionality. This was achieved by decoupling the network control that made decisions about where traffic was sent from, to the forwarding systems that forwarded traffic to the selected destination. Through SDN, the network intelligence was logically centralized through a controller that maintained a global view of the entire network (Patel, et al., 2013; Zhong, et al., 2015; Gajjar, et al., 2016).

Some of the protocols used for network configuration were OpenFlow and NETCONF . However, Openflow was the most commonly used protocol in SDN through which a high level controller could control the behaviour of switches through a well defined interface.

Unlike traditional distributed architectures like the Border Gateway Protocol (BGP), Open Short Path First (OSPF) and the Intermediate System to Intermediate System (IS-IS) whose control topologies were peer to peer, Openflow's central-control model avoided the need to construct global policies from switch-by-switch configurations, and supported near-optimal traffic management (Senthil & Ranjani, 2015; Gajjar, et al., 2016). Kreutz et al. (2015) alluded that load balancing was envisioned as the first application of Openflow. Several algorithms and techniques were proposed so as to deal with load balancing but scalability was a major concern. Load balancing was meant to simplify the placement of network services in the network by putting into consideration both the network load and the available computing capacity of the respective resources.

SDN technologies increased the bandwidth capability, and significantly reduce operation and management complexity. However, some of the algorithms that were used that included round-robin scheduling algorithm and greedy scheduling algorithm suffered from low reliability, low scalability, and could not deal with mass-traffic (Zhong, et al., 2015).

1.2 The Problem Statement

In an enterprise network, the number of load balancers that would be required would grow as the network grew. The problem with that was that most load balancers in the traditional network were vendor specific, and therefore, it was difficult to configure the equipment and fix software bugs. By using these load balancers, it meant that there was over-dependence on the vendor to release new features so that certain services could be deployed. It also turned out that additional network devices had to be interoperable with the current vendor's solution. According to Gajjar et al. (2016), "When an enterprise invests in any vendor specific equipment, it has to incur an additional cost for hiring trained professionals who can configure and operate that equipment. The company may also have to pay the vendor to

provide the required training to their engineers to integrate the equipment with the existing network." In contrast, SDN adhered to open standard where the specifications used to develop and maintain the hardware and the software were shared (Gajjar, et al., 2016). The SDN load balancing provided for fault tolerance which improved performance and reduced the possibility of the servers becoming overloaded.

SDN controlled network devices were controlled from a single control plane that computed forwarding rules and pushed the rules down to the data plane using an agreed upon control protocol. The data plane then enforced those rules; and exceptions could then be pushed back to the control plane as they arose. However, as the network grew, it became difficult for the single controller to scale due to the number of switches that were managed by it (Yao, et al., 2015; Gajjar, et al., 2016).

Research by Guo, et al., (2014) indicated that having one centralized controller created problems that included poor scalability, poor responsiveness, and poor reliability, which significantly hindered and detained the deployment of SDN technology in large-scale production and data centre networks. So as to solve this problem, they resolved that by using multiple distributed controllers working together to achieve the function of the logically centralized controller, it would benefit the scalability and reliability of the distributed architecture while preserving the simplicity of the centralized system. They further noted that by using multiple controllers, there was the challenge of synchronizing the state among controllers due to the fact that each controller ran one domain simultaneously, and it only handled the local area switches in its domain. Each controller could therefore only obtain the state of other controllers through controller state synchronization. According to Wang et al. (2016) who had done extensive work on dynamic SDN controller in data centres, affirm that this could limit the network's ability to react to changes and could cause transient

congestion.

Yang et al. (2016) work on load balancing compared the centralized strategies against a distributed load balancing strategy in the design of scalable large distributed systems. They identified several problems in practice. Distributed schemes suffered from the ageing of load information. This was largely due to the non-preemptive message scheduler. Messages had to wait in the message queue causing delays in the processing of critical messages that contained the load information. This meant that the messages could get out-of-date when the execution time of the method being processed was long. This ageing of load information led to the load balancing runtime making poor load balancing decisions.

Gong et al. (2015) also proposed the use of dynamic load balancing in which the system checked the first packet of each flow sent to the controllers and determined forwarding rules according to the appropriate policy. Hence, load balancing would be carried out according to the current network conditions. However, this solution suffered from scalability limitations due to the deep involvement of the controllers in checking the first packet of each flow and a large number of rules in each switch. Another problem that they identified was route optimization in which there was the need for server selection services. Bandwidth intensive applications and content distribution networks (CDN) required an abstract view of the network in form of network maps and cost maps to help in selecting the best server instances.

The proposed solution aims at distributing the load amongst controllers including the introduction of the main controller which would have a global view of the entire network including the state of other controllers within the enterprise network. This ensures that the controllers are well synchronized and in case any of the controllers fails, the main controller

is able to decide on how traffic will be routed since it also has the flow table of the switches that were under the failed controller.

1.3 Aim

To develop a distributed load balancing algorithm that aims to achieve better switch forwarding speeds, flexibility in distributing load, and customizable reactions to load changes using SDN.

1.4 Specific Objectives

i. To identify the problems associated with distributed load balancing algorithms.

ii. To design a distributed load balancing algorithm based on SDN.

iii. To implement a distributed load balancing algorithm based on SDN.

iv. To simulate a distributed load balancing infrastructure using SDN.

v. To test the distributed SDN approach to load balancing as compared to the use of a centralized load balancing technique.

1.5 Significance of the Study

The use of OpenFlow based switch networks differed from legacy switching networks that depended on a distributed network of switches that would be programmed individually. With SDN, the load balancing capabilities of the OpenFlow based switches superseded the legacy switches since the legacy switches, which were proprietary, did not provide any load balancing (Govindraj, et al., 2012). Senthil and Ranjani (2015) pointed out that "the separation of the application logic from the forwarding substrate greatly facilitates the deployment and operation of new services and enables SDN to react gracefully to changing network demands and modifications to the substrate topology."

Compared with the traditional routers that stored routing tables containing destination network information and only calculated the next-hop network information without global network routing view, an SDN controller had the ability to show the global view of network at the beginning of network construction. By updating topology information of global network, the SDN controller could discover all paths between each source node to each destination node. By utilizing this superiority of global network view in SDN, the load condition of every global path could be evaluated (Chen-xiao & Ya-bin, 2016).

Feamster et al. (2013) alluded that a logically centralized controller had to cope with controller failure through replication, but replication introduced the potential for inconsistent state across replicas yet these replicas could have worked together to act like a single, logically centralized controller.

With this solution that makes use of distributed controllers, the controllers would ensure that the load apportioned to each controller is well managed in that no single controller would be overwhelmed by requests. In case of controller failures, the introduction of the main controller would ensure that the affected requests would be allocated to other available controllers within the network but dependent on their current load.

1.6 Scope and Limitation of the Study

The scope of the project will be limited to the design and development the algorithm that would best take care of load balancing in an enterprise network or data centre environment. The research will also test the single controller environment against the distributed controller environment. The algorithm will however not be implemented for the production environment due to lack of facilities and resources required in such an environment.

CHAPTER TWO: Literature Review

2.1 State of Art

2.1.1 SDN

Within the SDN framework of separation, the control plane was used to change the behaviour of the network without physical modification of the existing network infrastructure implementation. This allowed for the rapid deployment of new network architectures. As shown in the figure 1 below, the application layer provides innovative services and business applications. The control layer comprises of the controller which is an SDN software that sits on the server. For ease of use, the SDN software includes a uniform application program interface (API) that it uses to interact with other layers. The data layer includes network devices capable of providing hardware or switching operations which are software defined in the control layer and communicated through the OpenFlow standard protocol. The controller can thus calculate the routing path based on source and destination address, as well as collect real-time statistics from network devices within the network (Zhong, et al., 2015; Chen, et al., 2016).

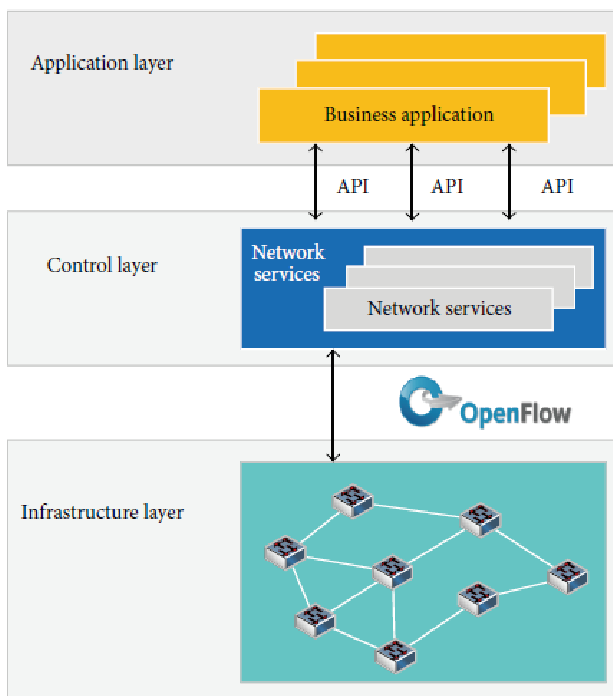

Figure 1: SDN Architecture. (Zhong, et al., 2015)

From the figure 1 above, it is evident that the control functions are removed from the network devices under the infrastructure layer and moved to an external entity under the control layer. The control layer thus allows for the programmability of the network devices through APIs whose decisions are flow based instead of destination based. The connecting bridges between the control layer and the infrastructure layer are referred to as southbound interfaces while those connecting the control layer and the application layer are referred to as northbound interfaces. An example of a southbound interface is Openflow. OpenFlow-enabled network devices are based on a pipeline of flow tables where each entry of a flow table has three parts, that is, a matching rule, actions to be executed, and statistics of matching packets. Flow abstraction therefore leads to unification of network devices behaviour which makes the network flexible due to the identical service policies received at the network devices (Kreutz, et al., 2015).

Schiff et al. (2016) identified that having logically centralized control plane offered better management of the network. However, they also noted that there was need to have the controller physically distributed. This would be achieved by having redundant controllers which would provide high availability in case any of the controllers failed. By having the controllers distributed, they would also be able to handle latency-sensitive and communication-intensive data plane as well as large SDNs operated by multiple administrators who could trigger policy changes concurrently. They also analysed that, with multiple controllers, there was the likelihood that they would try to install updates simultaneously which would cause a ripple effect resulting in synchronisation inconsistencies in the network state and distributed controllers.

2.1.2 Openflow

OpenFlow is a switching protocol that is commonly used in SDN. A generic switching architecture includes a control plane that makes decision on how to forward packets using a particular port and a forwarding plane that does the packet forwarding based on the decision made. The use of this generic solution leads to lack of scalability. However, this is mitigated in Openflow through SDN. The Openflow controller is capable of analysing packets based on a set of rules that are predefined. The Openflow switch can be software or hardware based. When it is software based, the controller can run on different device. However, when it is hardware based, the same device is capable of controlling switches in addition to hosting the controller. The other approach, in which Openflow can operate, is having a distributed architecture that has clusters of computers tasked in controlling the switching components by receiving flow information from the switches. This approach is suitable for scalable enterprise-scale network environment (Govindraj, et al., 2012).

The OpenFlow switches are dependent on the rules set by the controllers which are responsible for changing the forwarding behaviour of the switches. This means that they can alter the parameters of the forwarding table instantaneously due to the programmability of the network which would result in the orchestration of the network. The OpenFlow protocol allows for three methods of communication: (a) Communication between the controller and the switch which is responsible for retrieving of information from the switch as well as programming it. (b) Asynchronous communication which is not influenced by the controller and is responsible for relaying information to the controller about any change of state on the switches as well as the arrival of packets. (c) Symmetric communication in which communication can be initiated by either the switch or the controller. A good example is messages sent to find out if the channels are still alive (Braun & Menth, 2014).

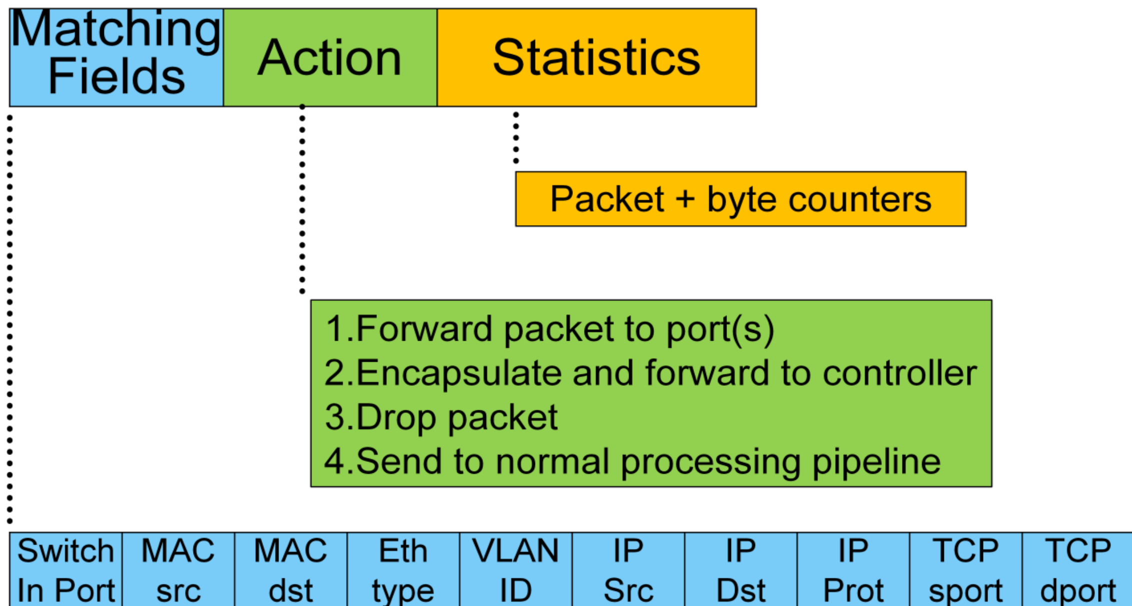

Figure 2: The OpenFlow flow entry (Trajano & Fernandez, 2016)

The figure 2 above shows the structure of a flow entry. When it comes to new flow entry, the Openflow controller is responsible for making decisions on how to handle a new flow entry in the switch. However, it first does a lookup on the flow table to identify if the flow table is matched. If not, it seek instructions from the controller. The controller can either drop the packet or add a flow entry that will assist the switch in forwarding subsequent packets from the same source. Another approach that can be used by the controller in handling new flow is by pro-actively populating the flow tables in the switches with information on all traffic that is expected. Through this method, the latency that is caused by the switches consulting the controllers is reduced significantly (Patel, et al., 2013; Du & Zhuang, 2015).

2.1.3 Load balancing

Load balancing configuration in an enterprise network or data centre environment in a traditional network consists of multiple servers behind a load balancer which services client's request. The load balancer helps in improving the performance by optimally using available resources in minimizing latency, response time and maximizing throughput.

However, the hardware has a rigid policy set. In contrast, SDN based load balancing, which is logically centralized, load balancing decisions are based on the load status of the entire network components (Namal, et al., 2013).

Chen et al. (2016) identified that "When a load balancer receives clients' requests for resources, it examines available resources in the pool. If proper resources are found, the load balancer re-routes the client to the server where the resources reside." Traffic spikes in the network are inevitable and can result in congestion of a link. The problem continues to persist even if there are redundant links to reach the servers. Load balancing algorithms are meant to address the issue of how the links are utilized. Load balancing can either be software based or hardware based whereby there are a dedicated hardware-based devices. There are also several load balancing algorithms available for the load balancer to distribute traffic among the servers (Gajjar, et al., 2016).

Software load balancing algorithms can broadly be categorized as either being static or dynamic. Static load balancing is predefined before transmission meaning that its routes cannot be changed since flows among hosts are allocated with calculated routes before data transmission and it does not refer to the states of the load nodes. The main aim of static load balancing is to decrease the overall execution time while minimizing the communication delays. The common static algorithms are *round-robin scheduling algorithm*, *weighted round-robin scheduling algorithm*, and *least-connection scheduling algorithm*. In comparison however, dynamic load balancing routing can schedule network traffic based on updated traffic statistics on each network device. Some of the dynamic load balancing algorithms include *honeybee foraging load balancing algorithm*, *biased random sampling load balancing algorithm* and *active clustering load balancing algorithm* (Li & Pan, 2013; Mondal, et al., 2016; Suguna & Barani, 2015; Saranya & Maheswari, 2015).

In round-robin, requests are served one after another through the passing of tokens. This means that when the request is sent to the last server, it starts again from the first server. This algorithm is therefore ideal where the servers are of equal specifications and the load is small. In weighted round robin, the load is distributed to the servers according to the weight that is assigned to it. In least connection algorithm, requests are sent to the server that has the least or no load at the time the connection is made. In dynamic load balancing algorithm, the decisions making is based on the current state of the system. If there is failure in any of the nodes, for instance, it would not stop the system but affect the performance of the system. This makes dynamic algorithm more resilient than static algorithms and can thus be better implemented in a dynamic environment (Suguna & Barani, 2015; Mondal, et al., 2016).

The *honeybee foraging load balancing algorithm* makes use of virtual servers that determine the queue to process based on the profit quality. If it is low, it goes to the forage where by identifying the state as either loaded, overloaded, balanced or under loaded. Based on the priority of the load, the virtual servers work by ensuring that neither is overloaded by allocating tasks that are waiting to other virtual servers thus reducing the response time. The *biased random sampling load balancing algorithm* makes use of virtual graph with which the load of each node represents connectivity. Each node represents a vertex in a directed graph. User requests are assigned to the node with the minimum in-degree which is selected using random sampling. Upon completion of the job, the node gets an increment of one in-degree thus indicating availability of resources. The disadvantage with this method is that as the number of servers increase, performance also decreases. The *active clustering load balancing algorithm* identifies similar nodes and groups them together. When a request is received by a node, it identifies its neighbour with which it balances the load. After processing the request, they are disconnected. By doing so, the throughput of the resources

is increased (Saranya & Maheswari, 2015).

In order to solve the problem associated with having a single controller, a distributed controller was proposed. However, as Yao et al. (2015) identified in their research on multi-controller, load balancing using multiple controllers is a challenging problem, especially for the enterprise-scale network of which the intended research is aimed at resolving. They further analysed that, the normal load balance in SDN was focused on traditional scheme that was adopted in traditional networks, and the distributed controller method was not considered. With the distributed network method, it would improve the enterprise-scale network performance considering that it has a global view of the entire network state. In a scenario where the controllers are distributed, when a controller is overloaded, it will signal to notify other controllers in the same cluster to deal with the extra requests. In case where there is load which still cannot be processed in the said cluster, the overloaded controller will seek assistance by signalling the super controller. In this way, the controller that is overloaded recovers to the normal levels (Yao, et al., 2015).

2.2 State of Practice

In practice, traditional networks are still widely used in routing and load balancing. Dedicated hardware load balancers like the *Barracuda load balancer* and *Coytepoint load balancer* are used in data centres and cloud environment. The migration to SDN is gradual but still not widely adopted in large-scale environment. Organisations like Google have made inroads in adopting SDN through their *B4 software defined wide area network (WAN)*. Other industry leaders have also developed their own SDN controllers like *Cisco ACI*, *Juniper Contrail*, *Big Switch's BNC*, *Brocade Vyatta Controller*.

Traditional inter-domain protocols for distributed architecture like BGP are used in routing.

The routing only occurs on destination IP prefix and can only influence routing for immediate neighbours but lack customization of routes by application or sender. This makes it difficult to send traffic along custom routes for particular applications such as video streaming or other types of applications that demand particular performance guarantees. However, through Routing Control Platform (RCP), additional information can be accommodated. Through storage of additional information, redundancy is eliminated in RCP since it only stores a single copy of each route. This accelerates lookups by having indexes that identify routes that can be affected by changes if for instance, the link failed. Standby replica RCP can also take over if the primary fails (Feamster, et al., 2013; Bondkovskii, et al., 2016; Caesar, et al., 2005).

In the traditional IP network, Guo et al. (2014), the function of load balancing is achieved by the Load Balancing Router (LBR). Specifically, when a new flow *f new* enters the network, *f new* can first go through LBR. In LBR, a specific server (for example, the least loaded server) can be selected as the destination server of *f new* based on the current network status. The IP address of the destination server is then written into the header of *p new* (that is, the packet of f new). Subsequently, *p new* can be strictly forwarded to its destination server through the path calculated by routing protocols (for example, OSPF). The forwarding decision is based on the network status at the time of selecting the destination server. However, the network status can change over time. In such conditions, during the forwarding process, there are some servers that are more suitable than the destination server selected by LBR before. Therefore, the load-balancing performance in the IP network is not optimal.

2.3 Technological Development

2.3.1 DNS load balancing

The most widely used load balancing technology was Domain Name Server (DNS) round robin load balancing which was a simple and effective method to manage traffic load. The DNS translated human readable domain names to IP addresses. The DNS server distributed user requests sequentially across the physical servers allocated IP addresses. This method demanded that new servers be added as the user requests grew. In the data centre environment, load balancer hardware devices were deployed so as to distribute the network traffic across multiple servers so as to avoid congestion. However, the DNS would not know whether any particular server that was in its list had failed or not. Load balancing devices therefore needed to have the ability to detect any server that had failed dynamically and remove it from the DNS list. There was also need for predictability. The DNS server also needed to know which server a user would be allocated beforehand so as to provided persistence (Senthil & Ranjani, 2015; Uppal & Brandon, 2010).

2.3.2 ECMP load balancing

ECMP refers to Equal-Cost Multi-Path which was a static load-balancing scheme that spread traffic equally across multiple paths. Switches which were ECMP enabled had multiple paths with which a packet used. The forwarding of a packet with multiple paths was decided by splitting load to each subnet across multiple paths based on selected fields in the packet header. This led to collision especially with large flows that resulted in creating a bottleneck. The collisions degraded the overall switch performance as well as overran the switch buffers. Just like DNS load balancing, ECMP was also affected by link failures thus leaving the network congested in asymmetric topologies (Akyildiz, et al., 2014; Ahmad, et al., 2015; Katta, et al., 2016).

2.3.3 Dynamic Load Balancing

Dynamic load balancing was an SDN-based solution that was adaptive to different network states and always kept the state updated. The servers were continually monitored and thus the status was always changing. The connections were distributed based on the server's performance; high performing servers would carry most of the load but would also be based on the node's response time. In large scale deployment, distributed controllers also caused the problems of synchronization. Hardware based mechanisms were proposed that saw the separation of physical servers or controller's operations split. The other method that was proposed was the operating system based mechanism that was based on the multi threading of the controller (Senthil & Ranjani, 2015; Mondal, et al., 2016; Akyildiz, et al., 2014).

2.4 Critique of the Literature

Zhong et al. (2015) work relating to traffic scheduling indicated that existing traffic scheduling algorithms like the round-robin scheduling algorithm suffered from low reliability and low scalability when dealing with mass traffic. If round robin is used in large-scale networks, there is the likelihood that a request can be forwarded to the farthest server. This would lead to inefficiency due to the traversal of the data across the network and the bandwidth consumption within the network. Hardware load balancers are capable of distributing the loads into different servers so as to get the best performance, but they suffer from inflexible policies imposed by the vendor (Yahya, et al., 2015).

Other algorithms like the extended Dijkstra’s algorithm have been very useful in determining the best routing path to send a packet from source to destination nodes. By incorporating the global view of the entire network through the SDN controller, the extended Dijkstra’s algorithm is capable of finding the shortest path determined by the summation of the node weights and the edge weights.

The node weight nw[v] of v is defined according to equation (1)

$$nw[v] = \frac{\sum_{f \in Flow(v)} Bits(f)}{Capacity(v)}, \quad (1)$$

where *Bits*(*f*) stands for the number of *f*'s bits processed by node *v* per second.

The edge weight ew[e] of e is defined according to equation (2).

$$ew[e] = \frac{\sum_{f \in Flow(e)} Bits(f)}{Bandwidth(e)}, \quad (2)$$

where *Bits*(*f*) stands for the number of *f*'s bits passing through edge *e* per second.

However, relying on a centralized controller in an enterprise scale deployment within the network, the communication between devices would take time and consume bandwidth as the network expands (Yahya, et al., 2015; Li & Pan, 2013; Jiang, 2014).

Braun and Menth (2014) pointed out that "an OpenFlow-based SDN approach cannot extend the data plane functionality without an upgrade of the switches, due to the fact that OpenFlow only provides a fixed set of network operations. The OpenFlow controller is only able to program the switch with its supported set of operations."

2.5 Evaluation of the Current Methods

Wang et al. (2011) proposed to load balance traffic by slicing the IP address space into trees that isolate a set of clients to a set of servers. This method assigned fixed number of clients to a server based on their weights. In order to implement this method, wildcards were used resulting to the reduction of forwarding performance and creating management issues. The solution however, did not consider network metrics and only part of the network passes through the controller.

Koerner & Kao (2012) proposed the use of OpenFlow to load balance multiple services.

This relied on SDN controllers of which each controller was responsible for load balancing the traffic of a specific service. The experiments conducted focused more on the architecture rather than the real world implementation of the solution. The disadvantage of this solution was that a single controller could not cater for more than one specific service and thus negated from the reason for using SDN controller which was to utilize available resources.

Li and Pan (2013) proposed a load balancer based on Openflow for Fat-Tree networks that would support multipath forwarding. This would iteratively find the best path from a source to a destination through load balancing the network and allowing the use of alternative paths at runtime and therefore minimizing network congestion. However, the proposed algorithm would work only on networks that operated on the Fat-Tree topology.

2.6 Conclusion

From the literature review, it was identified that load balancing has been a problem especially when dealing with large-scale network or a data centre environment. Different load balancing algorithms have been proposed in order to deal with the problem but they fall short of being the optimal solution to the problem.

CHAPTER THREE

3.0 Introduction

This chapter identifies the research method that would be ideal in implementing the proposed solution; the tools to be used in the implementation of the proposed solution; the conceptual design; the tests to be done; as well as the tools to be used in the evaluation of the results.

3.1 Research Methodology

The approach that was used for the purpose of this research was the simulation approach, which is a subset of quantitative research. The simulation approach entails the construction of an artificial environment using computer based tools within which relevant information and data is generated. This allows for the observation of the dynamic behaviour of the SDN controllers under controlled conditions (Kothari, 2004). The research demonstrates how distributed load balancing can be applied in enterprise scale networks. Due to the sheer scale of such a deployment, the use of simulation was identified so as to mimic the real world environment. Through simulation, the complexity and scale of the deployment was addressed through the use of computer based tools that gave close or nearly the same experience as when the systems would have been deployed in real environment.

The testing of network designs requires proper functioning of all the components so as to check for efficiency and reduce cost in deployment in the real environment. There are many simulation tools in use like NS2, OPNET, OMNET, QUALNET, among others. However, these tools do not have the capability of being implemented in the real environment and thus are only meant for testing purpose. Mininet simulator, however, is capable of creating "network of virtual hosts, OpenFlow enabled switches, OpenFlow controllers and secure links on a standard Linux environment and supports SDN and OpenFlow custom

topologies." Mininet is capable of creating the network topology through python programming language and executing commands on multiple nodes. In the proposed setup, there will be two scenarios. One of the scenarios will be based on having a single controller whereas the other scenario will be based on having distributed controllers (Bholebawa, et al., 2016).

3.2 Tools to be adopted

The main tool that would be used in the implementation of this research would be the controller. Identifying the right controller means analysing different controllers so as to identify the one that meets the requirement of the research. After analysing the controllers, other compatible tools are also identified. The following controllers were identified (Feamster, et al., 2013):

NOX: It is an open source controller that is implemented in C++ programming language. However, it supports version 1.0 of Openflow which is an earlier version of the Openflow protocol. Current version of Openflow is 1.4. NOX supports Openflow's low level facilities and semantics meaning that programming skills are required and the programmer writes the control programs that register events and then writes event handlers that respond to the events.

POX: It is an open source controller that is implemented in Python programming language. Similar to NOX, it supports only version 1.0 of Openflow. However, it does not scale well and therefore has performance issues and is mostly used only for experiments and demonstrations.

Ryu: It is an open source controller implemented in Python programming language. It supports different versions of Openflow including version 1.4. It can also be integrated into the cloud using Openstack and has relatively better performance.

Floodlight: It is an open source controller implemented in Java programming language and supports version 1.0 of the Openflow protocol. It makes use of Representational State Transfer (REST) API for integration and can be deployed in the cloud environment. It is well suited for the production environment where performance is required.

OpenDaylight: It is an open source controller implemented in Java programming language with robust codebase. It supports different versions of Openflow and has abstraction capabilities for Northbound Interfaces. It is widely used and accepted in the industry since it supports different cloud environments. It also supports modular functions for both Northbound and Southbound interfaces. However, it is quite complex and has a steep learning curve.

There are many considerations in identifying which SDN controller to use, that is: choice of programming languages which can sometimes affect the performance of the controller; the learning curve; user base and the support behind the controller; the APIs (southbound or northbound) that are supported; and whether the controller was intended for use in research, education or production. From the analysis of the above controllers, OpenDaylight was identified as the most appropriate controller to use in the implementation of this research. Therefore, the tools that were adopted are as follows:

OpenDaylight: It is a controller that is hosted by the Linux Foundation. The controller is written in Java programing language and uses Apache Karaf Open Service Gateway Initiative (OSGI) components. It provides for two northbound interfaces for applications: The OSGi interface and a web-based (REST) interface. It also supports the southbound interfaces through the OpenFlow protocol. The interfaces are represented in

a web-based interface called open**D**ay**L**ight **U**ser e**X**perience (DLUX), which is a collection of interfaces that have access to the controller's functions. Open Daylight Group Based Policies, which are accessible though the REST API and have a well-defined DLUX graphical interface, provide flexibility to define intents or connections at the transport layer as well as at the network and data-link layers (Bondkovskii, et al., 2016).

Mininet: Mininet is a network simulator used to create scalable SDNs using Linux processes. Mininet is used to simulate the network topology and test the traffic flows. A Python script is used to create the topology in Mininet and the traffic flows are received from a remote OpenFlow controller (Govindraj, et al., 2012).

Iperf: An open source performance measuring tool used to test the bandwidth and throughput between two hosts such that one host acts as a server and the other as a client. The performance metrics can be measured either with TCP or UDP packets (Govindraj, et al., 2012).

VirtualBox: It is a lightweight virtualization solution that supports multiple host instances and guest operating systems. It is the open source solution that is used in creating the Mininet instance (Watson, 2008).

3.3 Conceptual Design

3.3.1 Conceptual model

The model illustrated on figure 3 below demonstrates how the concept will be actualised. The server cluster houses all the applications to be used by the host. The servers hosting the data are virtualized so as to achieve redundancy. The virtual servers are logically connected

to each distributed controller within the network. The main controller is connected to the distributed controllers so as to take care of state synchronization. The main controller is also logically connected to every switch within the network so as to take care of failure in case any of the distributed controllers fails. The servers are connected to the OpenDaylight distributed controller through APIs. The controllers then connect to the switches through the OpenFlow southbound interface. The switches are then connected to the hosts. The setup is meant to demonstrate how the distributed controllers can be utilised in load balancing so that the traffic that is meant for hosts in its domain does not need to be routed to the main controller or any other controller but processed within the domain. The domain in this case refers to all components that are supported by a single distributed controller without the involvement of another controller or component in transmission and receiving of packets. This is meant to achieve better processing speeds of user requests. It also applies that traffic destined to another domain does not also have to pass through the main controller since the distributed controllers are logically connected and they have a global view of the flow table.

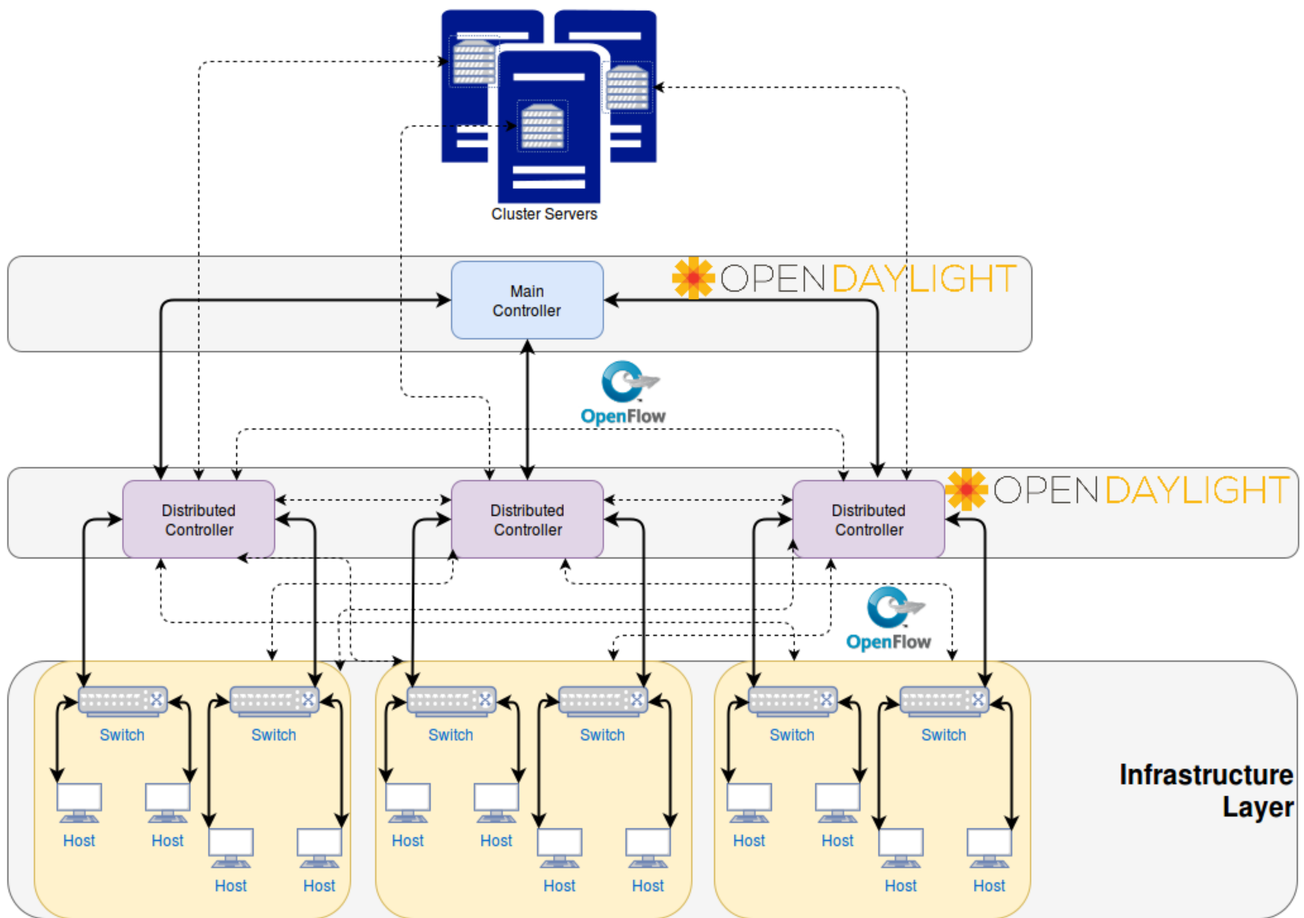

Figure 3: Conceptual Model

3.3.2 Flow Chart

The flow chart in figure 4 below shows a diagrammatic representation of how a user request through the distributed controllers will be processed. The flow starts with a user making a request from the end-user device. The end-user device then connects to the switch which checks from the flow entry as to whether the requests are destined for processing from its domain. From here on, there are two options:

(a) If it is within the domain, the distributed controller further checks on whether the request type is multimedia. If so, the controller check on whether the request exceeds the bandwidth threshold upon which it determines the routing protocol to use. If not, it processes the request based on the threshold set. If the request is not multimedia, it processes the request.

(b) If the request is not within its domain, the switch checks on the flow table on whether it belongs to a domain which is already on the flow table. If the domain exists it forwards the request to that domain for processing using the distributed controller attached to the domain. If not, it passes the request to the main controller which determines if it is a new flow entry or not. If it is, it can decide to terminate or process the user request. If it decides to process, it populates the flow tables of every switch through the distributed controllers within its global view from where the user request is forwarded for processing.

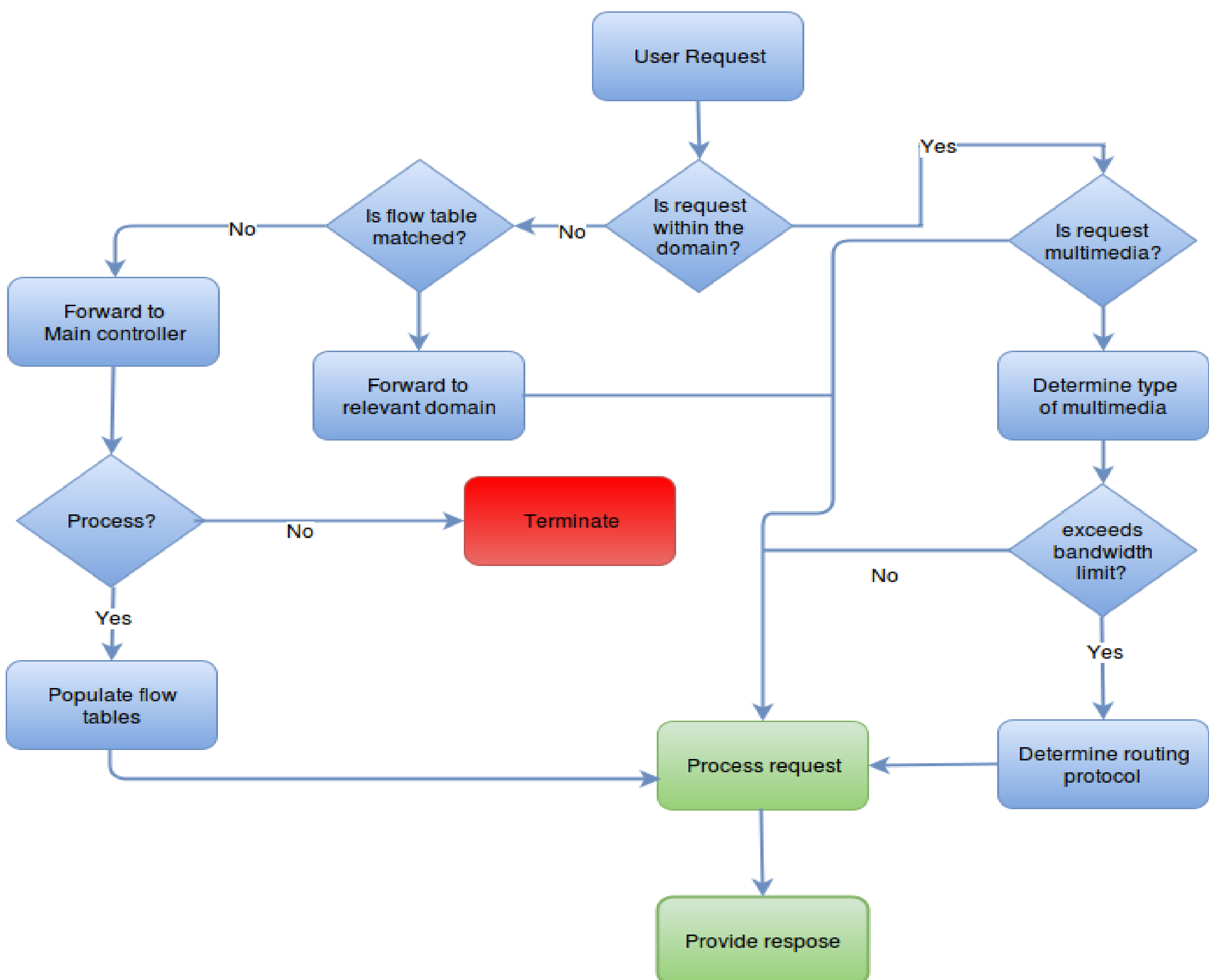

Figure 4: Flow Chart

3.4 Implementation Framework

3.4.1 Implementation Design

In the physical implementation design of the solution on distributed controllers as shown in the figure 5 below, the mininet network simulator is connected to the Ubuntu 14.04 OS. Java is then installed on the operating system. The distributed controller (OpenDaylight) is then installed on top of Java. The mininet is responsible for creating the controllers, virtual switches and virtual hosts from where the connection is made using the OpenFlow standard protocol.

Iperf measuring tool then connects with the two controllers as well as the xterm. Xterm is initiated from mininet and is responsible for providing two terminals that the Iperf uses for the host node as well as the server node.

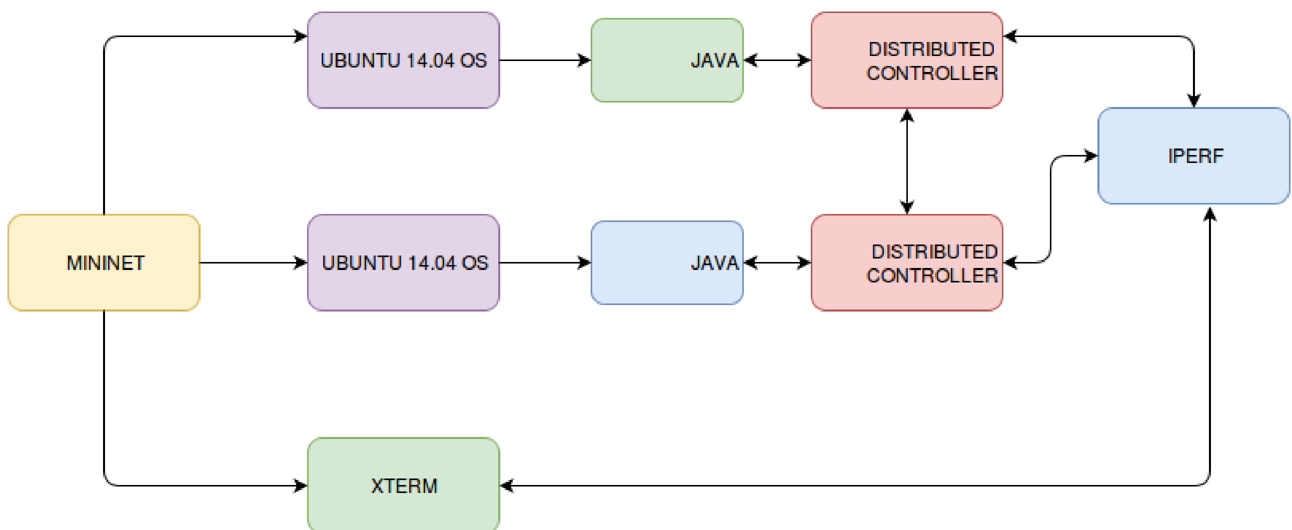

Figure 5: Implementation design

3.4.2 Framework Components

The table 1 below shows the components of the framework:

Table 1: Distributed controllers components

NO	COMPONENT DESCRIPTION	QUANTITY
1	Main controller	1
2	Distributed Controllers	3
3	Switches per distributed controller	2
4	Hosts per switch	2
5	Links between distributed controllers	3
6	Links between the main controller and distributed controllers	3
7	Links between the hosts and the distributed controllers	36

3.5 Tests to be done

To evaluate the research area, the effectiveness of having a distributed load balancing solution needs to be tested and the performance impact measured against a centralized set-up. Ubuntu 14.04 64-bit operating system will be used with *Virtualbox* installation for the simulation. The virtualbox will also be used to install the *Mininet* instance which installs the virtual switches and hosts. *OpenDaylight* is the controller to be used for the southbound interface based on the OpenFlow protocol. *Iperf* will be used for testing bandwidth performance in distributed controllers environment as well as the single controller environment (Röpke & Holz, 2016).

Iperf can only function between a server and a client but so as to have the same scenario as the enterprise environment, traffic is passed simultaneously and through multiple threads. The multiple threads represent multiple hosts making simultaneous access to resources on the network. These multiple threads are used to strain the network.

CHAPTER FOUR: Implementation and Testing

4.1 Introduction

In order to implement the project, there were hardware and software requirements that had to be met. Since the setup was done in a virtual environment running on a standard laptop, the requirements could not be compared to those needed in the production environment that uses dedicated servers. OpenDaylight, was the controller used, which did not come with any feature installations. The installation of the features were based on what ought to have been achieved in the research.

4.2 Hardware Requirement

Laptop

4.3 Software Requirements

(a) OS installation: Linux (Ubuntu 14.04)

(b) Java Virtual Machine (current version)

(c) Hypervisor (Virtualbox)

(d) OpenDayLight controller

(e) Mininet

(f) Iperf

4.4 System Requirements

CPU	4 cores
RAM	8 GB
Storage	80 GB

4.4.1 Installations and Configurations

4.4.1.1 Mininet

Mininet was installed as an appliance on the virtual machine. The virtual machine in this case, Virtualbox. After installation, the network adapter configurations on adapter 2 required to be changed so as to ensure that Mininet was accessible from outside the virtual environment as illustrated in figure 6 below. This was done by attaching the host-only adapter to the virtualbox network.

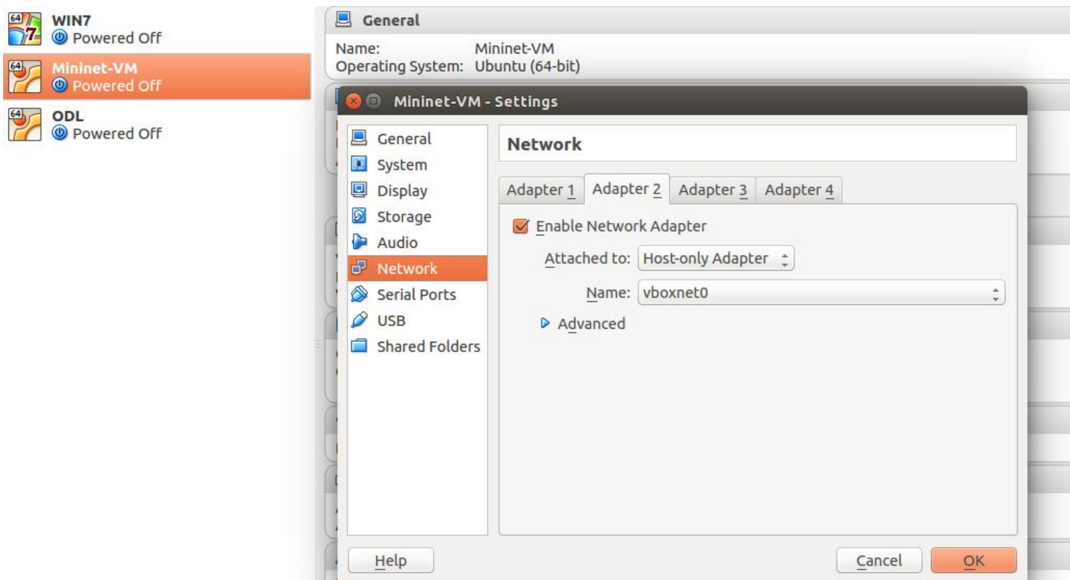

Figure 6: Network adapter configuration

After powering on Mininet, the network configurations were checked so as to identify the IP addresses that would be used to access the Mininet from outside the virtual environment as shown in figure 7 below. The essence of this was to ensure that SSH could be used to access the Mininet as well as ensuring that the OpenDayLight controller could be mapped from Mininet.

```
Ubuntu 14.04 LTS mininet-vm tty1

mininet-vm login: mininet
Password:
Last login: Sun Sep  4 06:23:09 PDT 2016 from 192.168.56.1 on pts/1
Welcome to Ubuntu 14.04 LTS (GNU/Linux 3.13.0-24-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
mininet@mininet-vm:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:fb:30:5e
          inet addr:192.168.56.102  Bcast:192.168.56.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1180 (1.1 KB)  TX bytes:684 (684.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:152 errors:0 dropped:0 overruns:0 frame:0
          TX packets:152 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:12144 (12.1 KB)  TX bytes:12144 (12.1 KB)

mininet@mininet-vm:~$
```

Figure 7: Identifying the Mininet IP address

Once this was achieved, the Mininet was then used to create virtual hosts, virtual switches, associated links and the controller.

4.4.1.2 OpenDaylight Controller

The OpenDayLight controllers were installed on Linux operating systems, that is, two separate Ubuntu 14.04 operating systems in a virtual environment. Associated pre-requisite software that was required for the OpenDayLight to run included Java. Once Java was installed, then OpenDayLight was installed on each virtual machine. The web interface default port used was 8181 which provided a log in page as shown in figure 8 below.

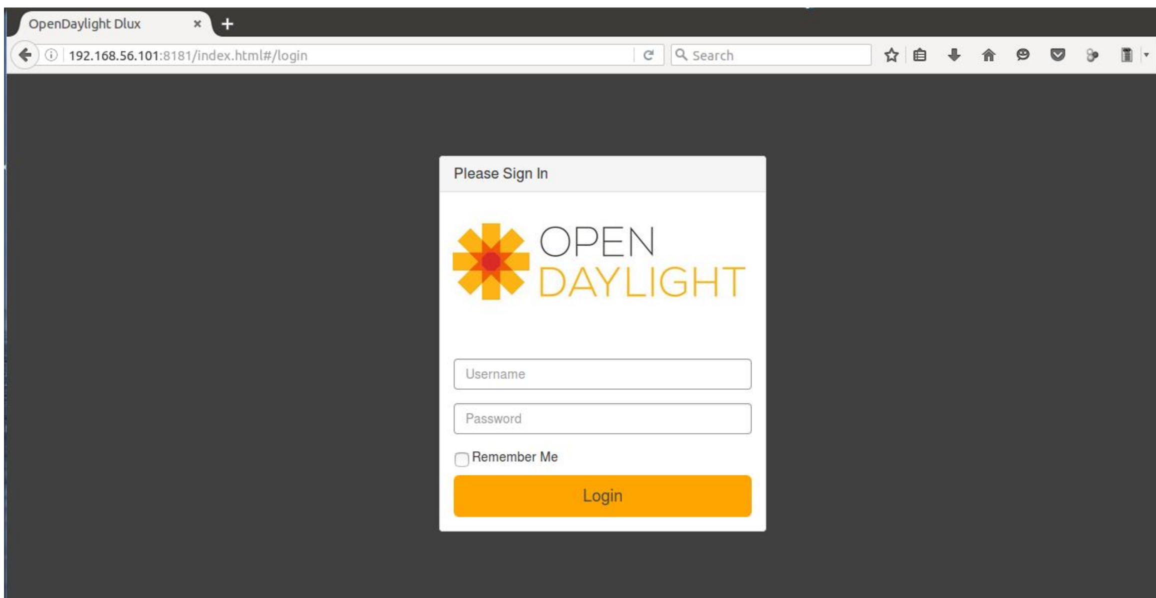

Figure 8: OpenDayLight web interface

Once logged in to the OpenDaylight controller, the virtual hosts and virtual switches created by Mininet were then visually accessible.

4.5 Implementation of the algorithm

4.5.1 Variables

U_{req} - User Request

R - Requirement from U_{req}

R_b - Bandwidth Requirement

T - Threshold

T_b - Bandwidth Threshold

M - Multimedia

C_0 - Main controller

$C_{1(a,b,c...n)}$ - Distributed controller

D - Domain

S - Switch

P_{req} - Process Request

P_{com} - Process complete

P_{res} - Provide Response

Min_L - Minimum Load

Max_L - Maximum Load

F_T - Flow Table

4.5.2 Algorithm

When ($U_{req} \in C_{1a}$)

{

If $U_{req} \neq M$

P_{req}

When ($P_{req} = P_{com}$)

$P_{com} = P_{res}$

else if

$U_{req} <= T_b$

P_{req}

When ($P_{req} = P_{com}$)

$P_{com} = P_{res}$

else if $U_{req} < Min_{L(C1b)}$ && $U_{req} > Max_{L(C1a)}$

Select $C_{1(b)}$

P_{req}

When ($P_{req} = P_{com}$)

$P_{com} = P_{res}$

else $U_{req} > Max_{L(C1b)}$ && $U_{req} > Max_{L(C1a)}$

Select $C_{1(a,b,c...n)}$ where $U_{req} < Max_{L}(C_{1(a,b,c...n)})$

P_{req}

When ($P_{req} = P_{com}$)

$P_{com} = P_{res}$

}

Where ($U_{req} \notin C_{1a}$)

{

If ($U_{req} \in C_{1(a,b,c...n)}$)

P_{req}

When ($P_{req} = P_{com}$)

$P_{com} = P_{res}$

Else if ($U_{req} \notin C_{1a}$) && ($U_{req} \notin C_{1(a,b,c...n)}$)

$C_{1(a,b,c...n)} \in C_0$

$C_{1(a,b,c...n)} \subseteq C_0 \therefore U_{req} \in C_0$

then $\exists\ F_{T(C1(a,b,c...n))}\ |\ C_{1(a,b,c...n)} \in C_0$

$\therefore\ U_{req} \in C_{1(a,b,c...n)}$

P_{req}

When ($P_{req} = P_{com}$)

$P_{com} = P_{res}$

}

The above algorithm can be explained as follows:

i. When user request is within the domain of a controller, then that request can be determined if it has multimedia elements in it. If it does not, then the user request is processed and the results submitted. If the request has multimedia elements, and are not beyond the allowed threshold, then the user request can be processed provided there is capacity for that. If the multimedia request is above the allowed threshold, the user request can be distributed to the controller(s) with the least load amongst the rest of the controllers.

ii. If the user request is destined for another domain, then the request is sent to the appropriate controller and the process as describe in (i.) above, applies. If the user request is not destined to any of the distributed controller, then the main controller takes charge in determining the destination of the request. The main controller then populates the flow tables in all the distributed controllers such that if there is any subsequent request of a similar nature then the request can be apportioned to the appropriate controller.

4.5.3 Algorithm Pseudocode

```
//get user requests from the host machines

//Domain denotes the switches, hosts and servers

// A distributed controller has control of a single domain but also has access to other domains

GET Ureq

SET D as Domain

SET C0 as Main Controller

SET C1 as Distributed Controller A

SET C2 as Distributed Controller B

SET CX as pool of Distributed Controllers

SET M as Multimedia

SET BT as Bandwidth Threshold

SET PR as Processed Request

SET ML as Maximum Load

SET FT as Flowtable


BEGIN
```

```
//Processing the Input
INIT Ureq
EXCEPTION
    //Where User request is processed within the domain
    WHEN Ureq : D && Ureq : C1
        IF Ureq == M
            IF Ureq <= BT
                COMPUTE Ureq
                SHOW PR
            ELSE
                IF C2 < ML
                    OBTAIN C2
                    //The two controllers to distribute the load
                    Ureq : C1 && Ureq : C2
                    COMPUTE Ureq
                    C2(RELEASE)
                    SHOW PR
                ELSEIF C1 = ML && C2 = ML
                    //Select from the pool of controllers
                    OBTAIN CX
                    Ureq : CX
                    COMPUTE Ureq
                    CX(RELEASE)
                    SHOW PR
                ENDIF
            ENDIF
```

```
                    ELSE
                              IF Ureq <= BT
                                        COMPUTE Ureq
                                        SHOW PR
                              ELSE
                                        OBTAIN C2
                                        //The two controllers to distribute the load
                                        Ureq : C1 && Ureq : C2
                                        COMPUTE Ureq
                                        C2(RELEASE)
                                        SHOW PR
                              ENDIF
                    ENDIF
          //Where user request is not within the domain but part of the Main Controller.
//We make use of Flowtable tables to populate and synchronize other controllers
          WHEN Ureq !: D && Ureq !: CX && Ureq : C0
                    BUT CX : C0
                    //Populate Flowtable so that the new flow entry
                    COMPUTE FT
                    FT : CX
                    DETERMINE CX
                    Ureq : CX
                    COMPUTE Ureq
                    SHOW PR
END
```

4.5.4 Mininet deployment

Mininet was used in creating controllers, switches, links between switches and hosts used in determining whether having a single controller was better than having distributed controllers by passing simultaneous traffic so as to measure bandwidth and latency as well as simulate multiple threads within both environments.

4.5.4.1 Creating a single controller

So as to simulate the single controller environment, a controller, c0, was created; sixteen switches, s1 through s16 were created; sixteen hosts were also created. This was done using the command below

mininet@mininet-vm:~$ sudo mn --controller=remote,ip=192.168.56.101 --topo=linear,16

Host two, h2, was made the server using the command

mininet> h2 python -m CGIHTTPServer &

Simultaneous traffic was then passed from server, h2 to the client, h12

4.5.4.2 Creating distributed controllers

So as to simulate the distributed environment, a python script, *controllers.py*, was written so as to create two controller, c1 and c2; six switches, s1 through s6; and sixteen hosts, h1 through h16.

Controllers.py

```
# Creating a network of switches connected to multiple controllers.

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

setLogLevel ('info')

# two remote controllers
```

```
c1 = RemoteController( 'c1', ip='192.168.56.101',  port=6633)
c2 = RemoteController( 'c2', ip='192.168.56.102',  port=6633)

cmap = {'s1': c1, 's2': c1, 's3': c1, 's4': c2, 's5': c2, 's6': c2}

class MultiSwitch ( OVSSwitch ):

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

topo = TreeTopo( depth=2, fanout=4 )
net = Mininet( topo=topo, switch=MultiSwitch, build=False )
for c in [ c1, c2 ]:
        net.addController (c)
net.build()
net.start()
CLI( net )
net.stop()
```

Host four, h4, was made the server using the command

mininet> h4 python -m CGIHTTPServer &

Simultaneous traffic was then passed from server, h4 to the client, h9 which were on different controllers.

4.5.5 OpenDayLight

Upon creation of the virtual switches, hosts, and controller, they were accessible from the OpenDayLight web interface as show on figure 9 below.

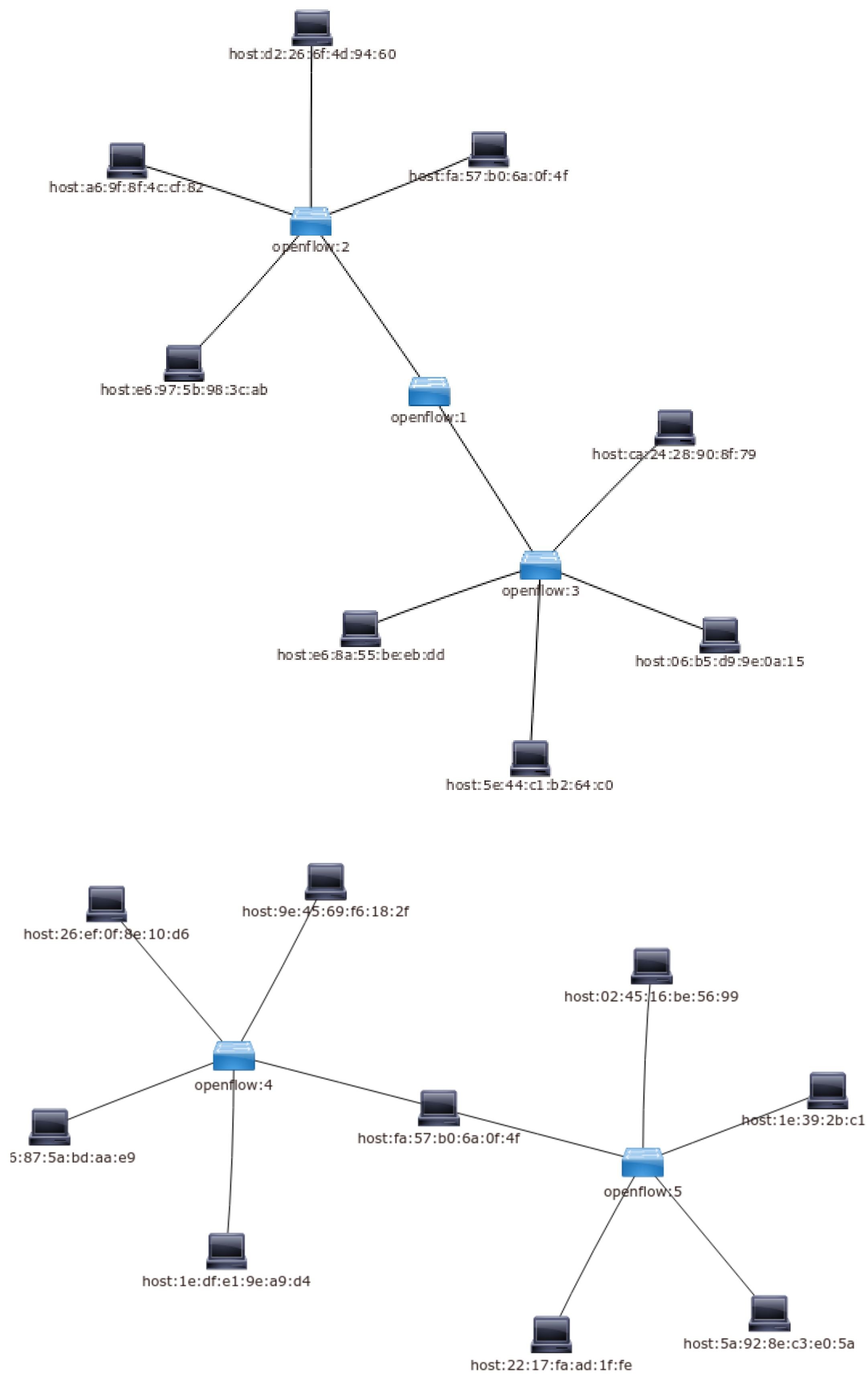

Figure 9: Distributed Controllers with switches and hosts

4.6 Data Analysis methods

The data was captured using *iperf* which was the measuring tool for simultaneous access between the server and the host. The tests were then categorized into four:

(a) **Single Controller tests**

In the single controller test scenario, two terminals were created from *Mininet* using the following command:

mininet> xterm h2 h12

After creating the two terminals, h2 which had earlier been identified as the server was also used as a server in *iperf* as illustrated in figure 10 below.

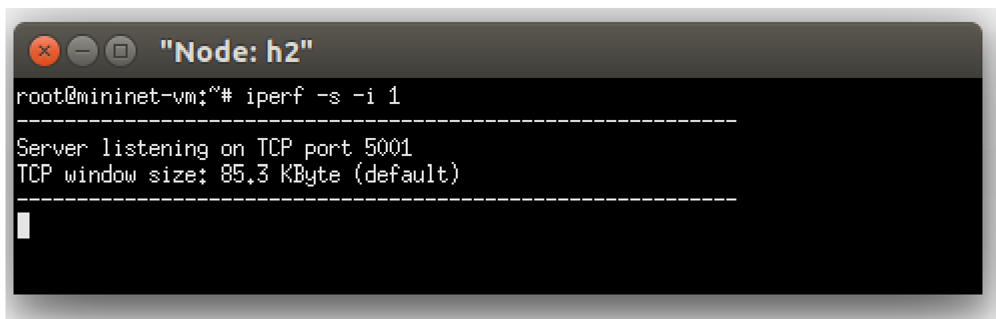

Figure 10: Iperf server node

From the above command, the interval time was set to one second. After setting up the iperf server, traffic needed to be passed from the iperf client so that iperf server could measure the bi-directional bandwidth simultaneously within fifteen seconds as shown figure 11 below. The *-d* argument was used to take care of bi-directional simultaneous bandwidth measurement.

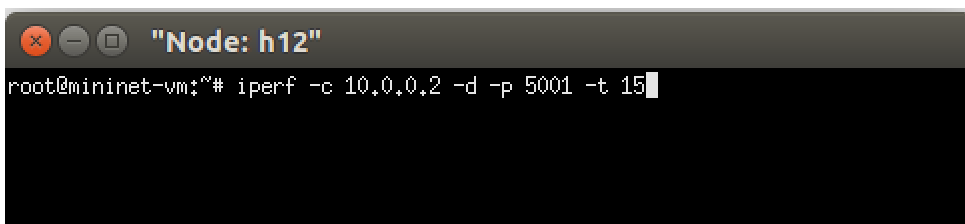

Figure 11: Iperf Client node

Upon running the command on host twelve, h12, the bandwidth was measure as shown in figure 12 below.

```
"Node: h12"
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5001
TCP window size:  246 KByte (default)
------------------------------------------------------------
[ 55] local 10.0.0.12 port 34674 connected with 10.0.0.2 port 5001
[ 57] local 10.0.0.12 port 5001 connected with 10.0.0.2 port 39288
[ ID] Interval       Transfer     Bandwidth
[ 55]  0.0-15.0 sec   164 MBytes  91.5 Mbits/sec
[ 57]  0.0-15.2 sec   124 MBytes  68.2 Mbits/sec
root@mininet-vm:~# iperf -c 10.0.0.2 -d -p 5001 -t 15
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5001
TCP window size:  162 KByte (default)
------------------------------------------------------------
[ 57] local 10.0.0.12 port 34676 connected with 10.0.0.2 port 5001
[ 56] local 10.0.0.12 port 5001 connected with 10.0.0.2 port 39290
[ ID] Interval       Transfer     Bandwidth
[ 57]  0.0-15.1 sec   112 MBytes  62.6 Mbits/sec
[ 56]  0.0-15.3 sec  81.1 MBytes  44.5 Mbits/sec
root@mininet-vm:~#
```

```
"Node: h2"
^Croot@mininet-vm:~# more result
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 56] local 10.0.0.2 port 5001 connected with 10.0.0.12 port 34676
------------------------------------------------------------
Client connecting to 10.0.0.12, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 58] local 10.0.0.2 port 39290 connected with 10.0.0.12 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 56]  0.0- 1.0 sec  7.98 MBytes  67.0 Mbits/sec
[ 58]  0.0- 1.0 sec  4.00 MBytes  33.6 Mbits/sec
[ 56]  1.0- 2.0 sec  7.97 MBytes  66.9 Mbits/sec
[ 58]  1.0- 2.0 sec  4.62 MBytes  38.8 Mbits/sec
[ 56]  2.0- 3.0 sec  8.49 MBytes  71.2 Mbits/sec
[ 58]  2.0- 3.0 sec  5.25 MBytes  44.0 Mbits/sec
[ 56]  3.0- 4.0 sec  7.39 MBytes  62.0 Mbits/sec
[ 58]  3.0- 4.0 sec  4.88 MBytes  40.9 Mbits/sec
[ 56]  4.0- 5.0 sec  7.68 MBytes  64.5 Mbits/sec
[ 58]  4.0- 5.0 sec  5.38 MBytes  45.1 Mbits/sec
[ 56]  5.0- 6.0 sec  7.79 MBytes  65.4 Mbits/sec
[ 58]  5.0- 6.0 sec  5.62 MBytes  47.2 Mbits/sec
[ 56]  6.0- 7.0 sec  7.39 MBytes  62.0 Mbits/sec
[ 58]  6.0- 7.0 sec  5.38 MBytes  45.1 Mbits/sec
[ 56]  7.0- 8.0 sec  6.90 MBytes  57.9 Mbits/sec
[ 58]  7.0- 8.0 sec  5.75 MBytes  48.2 Mbits/sec
[ 58]  8.0- 9.0 sec  6.12 MBytes  51.4 Mbits/sec
[ 56]  8.0- 9.0 sec  6.30 MBytes  52.8 Mbits/sec
[ 56]  9.0-10.0 sec  4.62 MBytes  38.8 Mbits/sec
[ 58]  9.0-10.0 sec  6.75 MBytes  56.6 Mbits/sec
[ 56] 10.0-11.0 sec  6.57 MBytes  55.1 Mbits/sec
[ 58] 10.0-11.0 sec  5.38 MBytes  45.1 Mbits/sec
[ 56] 11.0-12.0 sec  7.55 MBytes  63.3 Mbits/sec
[ 58] 11.0-12.0 sec  5.75 MBytes  48.2 Mbits/sec
[ 56] 12.0-13.0 sec  7.88 MBytes  66.1 Mbits/sec
[ 58] 12.0-13.0 sec  5.88 MBytes  49.3 Mbits/sec
[ 56] 13.0-14.0 sec  7.33 MBytes  61.5 Mbits/sec
[ 58] 13.0-14.0 sec  5.00 MBytes  41.9 Mbits/sec
[ 56] 14.0-15.0 sec  8.41 MBytes  70.6 Mbits/sec
[ 58] 14.0-15.0 sec  5.25 MBytes  44.0 Mbits/sec
[ 58]  0.0-15.0 sec  81.1 MBytes  45.3 Mbits/sec
[ 56]  0.0-15.3 sec   112 MBytes  61.6 Mbits/sec
root@mininet-vm:~#
```

Figure 12: Measuring bandwidth using a single controller

(b) **Distributed Controllers test**

Similar tests were also done with the distributed controllers whereby host four, h4, on controller, c1, was made the *iperf server* and host nine, h9, on controller, c2, was made the client. The bandwidth was then simultaneously measured over a period of fifteen seconds as with the previous test as shown in figure 13 below.

"Node: h9"

```
------------------------------------------------------------
Client connecting to 10.0.0.4, TCP port 5001
TCP window size:  187 KByte (default)
------------------------------------------------------------
[ 51] local 10.0.0.9 port 37853 connected with 10.0.0.4 port 5001
[ 53] local 10.0.0.9 port 5001 connected with 10.0.0.4 port 40161
[ ID] Interval       Transfer     Bandwidth
[ 51]  0.0-15.1 sec   106 MBytes  59.0 Mbits/sec
[ 53]  0.0-15.2 sec   192 MBytes   106 Mbits/sec
root@mininet-vm:~/mininet/custom# iperf -c 10.0.0.4 -d -p 5001 -t 15
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 10.0.0.4, TCP port 5001
TCP window size:  264 KByte (default)
------------------------------------------------------------
[ 51] local 10.0.0.9 port 37855 connected with 10.0.0.4 port 5001
[ 53] local 10.0.0.9 port 5001 connected with 10.0.0.4 port 40163
[ ID] Interval       Transfer     Bandwidth
[ 51]  0.0-15.1 sec  89.5 MBytes  49.8 Mbits/sec
[ 53]  0.0-15.3 sec   171 MBytes  94.2 Mbits/sec
root@mininet-vm:~/mininet/custom#
```

"Node: h4"

```
^Croot@mininet-vm:~/mininet/custom# more result
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 52] local 10.0.0.4 port 5001 connected with 10.0.0.9 port 37855
------------------------------------------------------------
Client connecting to 10.0.0.9, TCP port 5001
TCP window size:  238 KByte (default)
------------------------------------------------------------
[ 54] local 10.0.0.4 port 40163 connected with 10.0.0.9 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 52]  0.0- 1.0 sec  9.00 MBytes  75.5 Mbits/sec
[ 54]  0.0- 1.0 sec  9.88 MBytes  82.8 Mbits/sec
[ 52]  1.0- 2.0 sec  4.86 MBytes  40.8 Mbits/sec
[ 54]  1.0- 2.0 sec  11.8 MBytes  98.6 Mbits/sec
[ 52]  2.0- 3.0 sec  4.47 MBytes  37.5 Mbits/sec
[ 54]  2.0- 3.0 sec  10.6 MBytes  89.1 Mbits/sec
[ 52]  3.0- 4.0 sec  4.25 MBytes  35.7 Mbits/sec
[ 54]  3.0- 4.0 sec  10.8 MBytes  90.2 Mbits/sec
[ 52]  4.0- 5.0 sec  4.09 MBytes  34.3 Mbits/sec
[ 54]  4.0- 5.0 sec  11.6 MBytes  97.5 Mbits/sec
[ 52]  5.0- 6.0 sec  5.30 MBytes  44.5 Mbits/sec
[ 54]  5.0- 6.0 sec  10.8 MBytes  90.2 Mbits/sec
[ 52]  6.0- 7.0 sec  5.66 MBytes  47.5 Mbits/sec
[ 54]  6.0- 7.0 sec  11.8 MBytes  98.6 Mbits/sec
[ 52]  7.0- 8.0 sec  6.24 MBytes  52.3 Mbits/sec
[ 54]  7.0- 8.0 sec  12.9 MBytes   108 Mbits/sec
[ 52]  8.0- 9.0 sec  6.41 MBytes  53.8 Mbits/sec
[ 54]  8.0- 9.0 sec  13.0 MBytes   109 Mbits/sec
[ 52]  9.0-10.0 sec  6.40 MBytes  53.6 Mbits/sec
[ 54]  9.0-10.0 sec  13.0 MBytes   109 Mbits/sec
[ 52] 10.0-11.0 sec  6.20 MBytes  52.0 Mbits/sec
[ 54] 10.0-11.0 sec  10.6 MBytes  89.1 Mbits/sec
[ 52] 11.0-12.0 sec  5.71 MBytes  47.9 Mbits/sec
[ 54] 11.0-12.0 sec  9.38 MBytes  78.6 Mbits/sec
[ 52] 12.0-13.0 sec  6.36 MBytes  53.3 Mbits/sec
[ 54] 12.0-13.0 sec  11.8 MBytes  98.6 Mbits/sec
[ 52] 13.0-14.0 sec  7.10 MBytes  59.5 Mbits/sec
[ 54] 13.0-14.0 sec  11.8 MBytes  98.6 Mbits/sec
[ 52] 14.0-15.0 sec  5.34 MBytes  44.8 Mbits/sec
[ 54] 14.0-15.0 sec  11.8 MBytes  98.6 Mbits/sec
[ 54]  0.0-15.1 sec   171 MBytes  95.3 Mbits/sec
[ 52]  0.0-15.3 sec  89.5 MBytes  49.0 Mbits/sec
root@mininet-vm:~/mininet/custom#
```

Figure 13: Measuring bandwidth using distributed controllers

The results from the two tests were then tabulated for comparative analysis as shown in table 2 below.

Table 2: Comparative analysis

Time Interval (sec)	**Single Controller Bandwidth (Mbits/s) Server-Host (S-H)**	**Single Controller Bandwidth (Mbits/s) Host-Server (H-S)**	**Distributed Controller Bandwidth (Mbits/s) Server- Host (S-H)**	**Distributed Controller Bandwidth (Mbits/s) Host- Server(H-S)**
0.0 – 1.0	67.0	33.6	75.5	82.8
1.0 – 2.0	66.9	38.8	40.8	98.6
2.0 – 3.0	71.2	44.0	37.5	89.1
3.0 – 4.0	62.0	40.9	35.7	90.2
4.0 – 5.0	64.5	45.1	34.3	97.5
5.0 – 6.0	65.4	47.2	44.5	90.2
6.0 – 7.0	62.0	45.1	47.5	98.6
7.0 – 8.0	57.9	48.2	52.3	108.0
8.0 – 9.0	51.4	52.8	53.8	109.0
9.0 – 10.0	38.8	56.6	53.6	109.0
10.0 – 11.0	55.1	45.1	52.0	89.1
11.0 – 12.0	63.3	48.2	47.9	78.6
12.0 – 13.0	66.1	49.3	53.3	98.6
13.0 – 14.0	61.5	41.9	59.5	98.6
14.0 – 15.0	70.6	44.0	44.8	98.6

The results were then mapped to a line graph so as to show which of the two test demonstrated high bandwidth rate over the set time frame as shown in figure 14 below.

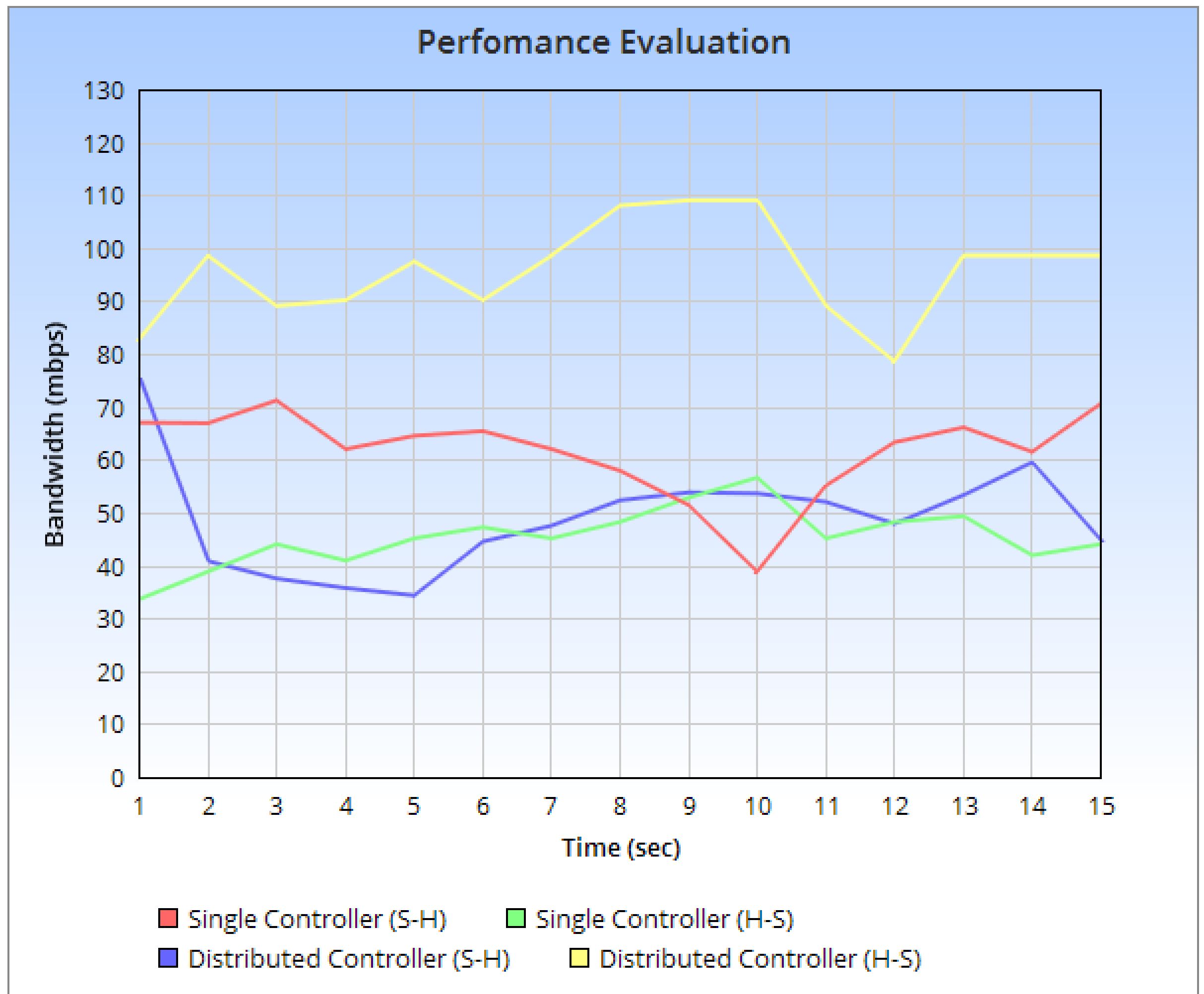

Figure 14: Performance evaluation

From the graph, the yellow and the blue lines denoted the distributed controllers whereas the red and green lines denoted the single controller. It was observed that the distributed controllers on average had higher throughput than the single controller and therefore could transfer more data over the two links as compared to using the single controller.

(c) **Strain Testing**

The strain test was meant to pass multiple threads from the *Iperf* client to the *Iperf*

server over a specific duration of time. This test was meant to determine how much traffic could be passed over the link. The *–P* argument was used to perform the test. The argument *–P* represented parallel connections while argument *–t* represented the time interval. The number of threads which were used were three.

For the single controllers, the results were as shown in figure 15 below.

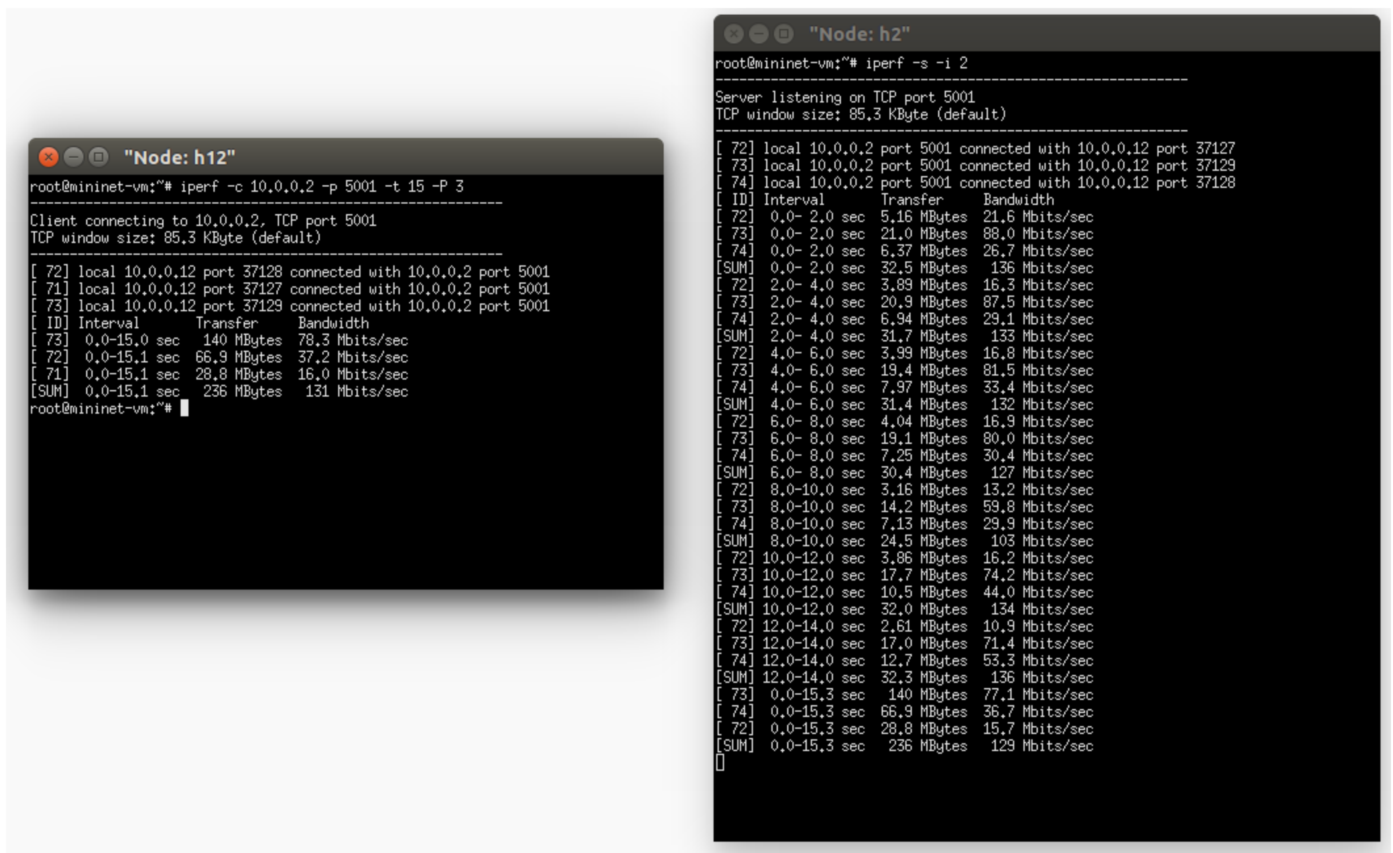

Figure 15: Multiple threads on a single controller

The results for the distributed controllers were as shown on figure 16 below.

```
"Node: h9"
root@mininet-vm:~/mininet/custom# iperf -c 10.0.0.4  -p 5001 -t 15 -P 3
------------------------------------------------------------
Client connecting to 10.0.0.4, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 53] local 10.0.0.9 port 40438 connected with 10.0.0.4 port 5001
[ 51] local 10.0.0.9 port 40436 connected with 10.0.0.4 port 5001
[ 52] local 10.0.0.9 port 40437 connected with 10.0.0.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 51]  0.0-15.0 sec   136 MBytes  75.7 Mbits/sec
[ 52]  0.0-15.1 sec  83.8 MBytes  46.6 Mbits/sec
[ 53]  0.0-15.2 sec  58.4 MBytes  32.2 Mbits/sec
[SUM]  0.0-15.2 sec   278 MBytes   153 Mbits/sec
root@mininet-vm:~/mininet/custom# 
```

```
"Node: h4"
root@mininet-vm:~/mininet/custom# iperf -s -i 2
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 52] local 10.0.0.4 port 5001 connected with 10.0.0.9 port 40438
[ 53] local 10.0.0.4 port 5001 connected with 10.0.0.9 port 40437
[ 54] local 10.0.0.4 port 5001 connected with 10.0.0.9 port 40436
[ ID] Interval       Transfer     Bandwidth
[ 52]  0.0- 2.0 sec  9.47 MBytes  39.7 Mbits/sec
[ 53]  0.0- 2.0 sec  6.99 MBytes  29.3 Mbits/sec
[ 54]  0.0- 2.0 sec  18.9 MBytes  79.4 Mbits/sec
[SUM]  0.0- 2.0 sec  35.4 MBytes   148 Mbits/sec
[ 52]  2.0- 4.0 sec  9.49 MBytes  39.8 Mbits/sec
[ 53]  2.0- 4.0 sec  7.19 MBytes  30.2 Mbits/sec
[ 54]  2.0- 4.0 sec  18.7 MBytes  78.4 Mbits/sec
[SUM]  2.0- 4.0 sec  35.4 MBytes   148 Mbits/sec
[ 52]  4.0- 6.0 sec  9.32 MBytes  39.1 Mbits/sec
[ 53]  4.0- 6.0 sec  8.25 MBytes  34.6 Mbits/sec
[ 54]  4.0- 6.0 sec  17.9 MBytes  75.2 Mbits/sec
[SUM]  4.0- 6.0 sec  35.5 MBytes   149 Mbits/sec
[ 53]  6.0- 8.0 sec  11.4 MBytes  48.0 Mbits/sec
[ 54]  6.0- 8.0 sec  15.5 MBytes  65.2 Mbits/sec
[ 52]  6.0- 8.0 sec  7.95 MBytes  33.3 Mbits/sec
[SUM]  6.0- 8.0 sec  34.9 MBytes   147 Mbits/sec
[ 53]  8.0-10.0 sec  15.9 MBytes  66.5 Mbits/sec
[ 54]  8.0-10.0 sec  17.5 MBytes  73.2 Mbits/sec
[ 52]  8.0-10.0 sec  6.36 MBytes  26.7 Mbits/sec
[SUM]  8.0-10.0 sec  39.7 MBytes   166 Mbits/sec
[ 52] 10.0-12.0 sec  5.49 MBytes  23.0 Mbits/sec
[ 53] 10.0-12.0 sec  13.2 MBytes  55.6 Mbits/sec
[ 54] 10.0-12.0 sec  18.9 MBytes  79.2 Mbits/sec
[SUM] 10.0-12.0 sec  37.6 MBytes   158 Mbits/sec
[ 52] 12.0-14.0 sec  6.86 MBytes  28.8 Mbits/sec
[ 53] 12.0-14.0 sec  14.2 MBytes  59.4 Mbits/sec
[ 54] 12.0-14.0 sec  20.5 MBytes  85.8 Mbits/sec
[SUM] 12.0-14.0 sec  41.5 MBytes   174 Mbits/sec
[ 53]  0.0-15.3 sec  83.8 MBytes  45.9 Mbits/sec
[ 54]  0.0-15.5 sec   136 MBytes  73.7 Mbits/sec
[ 52]  0.0-15.5 sec  58.4 MBytes  31.6 Mbits/sec
[SUM]  0.0-15.5 sec   278 MBytes   151 Mbits/sec
```

Figure 16: Multiple threads on distributed controllers

The results from the two tests for each thread were then tabulated on table 3 below.

Table 3: Strain test analysis

Controller	Thread 1 bandwidth (Mbits/s)	Thread 2 bandwidth (Mbits/s)	Thread 3 bandwidth (Mbits/s)	Average bandwidth (Mbits/s)
Single	78.3	37.2	16.0	**43.9**
Distributed	75.7	46.6	32.2	**51.5**

It was observed that after putting the strain on the network with multiple threads from the client to server, the average bandwidth for the distributed controller was higher than that of the single controller meaning that the distributed controller could pass more bandwidth on the network than the single controller.

(d) **Latency test**

The latency test was done from Mininet using the *round-trip time* (RTT) from the server to the client for both the distributed controllers as well as the single controller. This was

measured using the ping request over the same time interval, 10 seconds, in both scenarios.

For the single controller, the results were as follows:

mininet> h4 ping -c 10 h12

PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.

64 bytes from 10.0.0.12: icmp_seq=1 ttl=64 time=1.66 ms
64 bytes from 10.0.0.12: icmp_seq=2 ttl=64 time=0.249 ms
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=0.839 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=0.283 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=0.328 ms
64 bytes from 10.0.0.12: icmp_seq=6 ttl=64 time=0.382 ms
64 bytes from 10.0.0.12: icmp_seq=7 ttl=64 time=0.688 ms
64 bytes from 10.0.0.12: icmp_seq=8 ttl=64 time=0.229 ms
64 bytes from 10.0.0.12: icmp_seq=9 ttl=64 time=0.946 ms
64 bytes from 10.0.0.12: icmp_seq=10 ttl=64 time=0.265 ms

--- 10.0.0.12 ping statistics ---

10 packets transmitted, 10 received, 0% packet loss, time 9002ms
***rtt** min/**avg**/max/mdev = 0.229/**0.586**/1.660/0.437 ms*

For the distributed controllers, the results were as follows:

mininet> h4 ping -c 10 h12

PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.

64 bytes from 10.0.0.12: icmp_seq=1 ttl=64 time=0.311 ms
64 bytes from 10.0.0.12: icmp_seq=2 ttl=64 time=0.807 ms
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=0.596 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=0.224 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=0.739 ms
64 bytes from 10.0.0.12: icmp_seq=6 ttl=64 time=0.810 ms
64 bytes from 10.0.0.12: icmp_seq=7 ttl=64 time=0.412 ms
64 bytes from 10.0.0.12: icmp_seq=8 ttl=64 time=0.627 ms
64 bytes from 10.0.0.12: icmp_seq=9 ttl=64 time=0.653 ms
64 bytes from 10.0.0.12: icmp_seq=10 ttl=64 time=0.227 ms

--- 10.0.0.12 ping statistics ---

10 packets transmitted, 10 received, 0% packet loss, time 9001ms
***rtt** min/**avg**/max/mdev = 0.224/**0.540**/0.810/0.219 ms*

From the two tests, the average latency for the single controller was *0.586 milliseconds*

(ms) as compared to that of the distributed controller which was *0.540 milliseconds (ms)* over the same time interval. It was therefore observed that the distributed controllers had low latency as compared to the single controller.

CHAPTER FIVE: Conclusion and Future works

5.1 Conclusion

From the tests done, the distributed controllers were better at passing simultaneous traffic and multiple threads and also experienced less latency as compared to the single controller. This means that in large-scale environment such as an enterprise network covering different continents or a data centre environment, the distributed controllers would scale better. Critics can argue that since the single controller has the global view of the entire network and does not suffer from state synchronisation as the distributed controllers do, then it can be a better solution. However, with the introduction of the additional controller, dubbed, the main controller, in the distributed environment that takes care of state synchronisation and orchestration of failed controllers, then it can be argued that the distributed controllers can mitigate against this shortcomings.

Load balancing can therefore be implemented through the algorithm. The algorithm takes care of failover in cases where any of the distributed controllers fails and also makes decisions on how to address new flow into the network by assigning new flow to the distributed controller that is lightly loaded. This therefore ensures that the distributed controller need not worry about making certain decisions that are inherent with the current distributed controllers.

5.2 Suggestions for Further Studies

The current versions of the OpenFlow protocol have addressed the challenges posed by having static configurations between planes. A single switch can have access to more than one controller in three different states, namely, master, slave or equal states. In case the master controller fails, the other controller will transition to become the master. However, the role of shifting from one state to the other is left under the purview of the controller

since the Openflow protocol has no mechanism for transitioning between states. More research in this area is required so that the OpenFlow protocol can transition between states in case a controller fails so as to avoid situations in which a controller can become overloaded (Cheng, et al., 2016).

References

Ahmad, I., Karunarathna, S. N., Mika, Y. & Andrei, G., 2015. Load Balancing in Software Defined Mobile Networks. *Software Defined Mobile Networks (SDMN): Beyond LTE Network Architecture,* pp. 225-245.

Akyildiz, I. F. et al., 2014. A roadmap for traffic engineering in SDN-OpenFlow networks. *Computer Networks 71,* pp. 1-30.

Baskerville, R. L. & Wood-Harper, A. T., 2016. A critical perspective on action research as a method for information systems research. *In Enacting Research Methods in Information Systems,* Volume 2, pp. 169-190.

Bholebawa, I. Z., Rakesh, K. J. & Upena, D. D., 2016. Performance analysis of proposed OpenFlow-based network architecture using Mininet. *Wireless Personal Communications 86,* Issue 2, pp. 943-958.

Bhowmik, S., Tariq, M. A., Grunert, J. & Rothermel, K., 2016. Bandwidth-efficient content-based routing on software-defined networks. *In Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems,* pp. 137-144.

Bondkovskii, A., Keeney, J., van der Meer, S. & Weber, S., 2016. Qualitative comparison of open-source SDN controllers. *In Network Operations and Management Symposium (NOMS),* pp. 889-894.

Braun, W. & Menth, M., 2014. Software-Defined Networking using OpenFlow: Protocols, applications and architectural design choices. *Future Internet 6,* Issue 2, pp. 302-336.

Caesar, M. et al., 2005. Design and implementation of a routing control platform. *In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation,* Volume 2, pp. 15-28.

Cheng, G., Chen, H., Hu, H. & Lan, J., 2016. Dynamic switch migration towards a scalable SDN control plane. *International Journal of Communication Systems.*

Chen, L., Qiu, M., Dai, W. & Jiang, N., 2016. Supporting high-quality video streaming with SDN-based CDNs. *The Journal of Supercomputing,* pp. 1-15.

Chen-xiao, C. & Ya-bin, X., 2016. Research on Load Balance Method in SDN. *International Journal of Grid and Distributed Computing 9,* Issue 1, pp. 25-36.

Cisco, 2011. IP Routing: BGP Configuration Guide. *Cisco IOS Release 12.4T.*

Du, Q. & Zhuang, H., 2015. OpenFlow-Based Dynamic Server Cluster Load Balancing with Measurement Support. *Journal of Communications 10,* Issue 8.

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

Gajjar, D. et al., 2016. Round Robin Load Balancer using Software Defined Networking (SDN). *Capstone Team Research Project,* 22 April.

Gong, Y., Huang, W., Wang, W. & Lei, Y., 2015. A survey on software defined networking and its applications. *Frontiers of Computer Science 9,* Issue 6, pp. 827-845.

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

Guo, Z. et al., 2014. Improving the performance of load balancing in software-defined networks through load variance-based synchronization. *Computer Networks,* Issue 68, pp. 95-109.

Handigol, N. et al., 2009. Plug-n-Serve: Load-balancing web traffic using OpenFlow. *ACM Sigcomm Demo, 4,* Issue 5, p. 6.

He, X. et al., 2015. HyperFLOW: A Structured/Unstructured Hybrid Integrated Computational Environment for Multi-purpose Fluid Simulation. *Procedia Engineering, 126,* pp. 645-649.

Ho, C., Wang, K. & Hsu, Y., 2016. A fast consensus algorithm for multiple controllers in software-defined networks. *2016 18th International Conference on Advanced Communication Technology (ICACT),* January.pp. 112-116.

Jiang, J., 2014. Software Defined Networking and Dijkstra's Algorithm. *National Central University.*

Katta, N. et al., 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. *Proc. ACM Symposium on SDN Research.*

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

Koponen, T. et al., 2010. Onix: A Distributed Control Platform for Large-scale Production Networks. *OSDI ,* Volume 10, pp. 1-6.

Kothari, C., 2004. Research methodology: Methods and techniques. *New Age International.*

Kreutz, D. et al., 2015. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE,* 103(1), pp. 14-76.

Li, Y. & Pan, D., 2013. OpenFlow based load balancing for Fat-Tree networks with multipath support. *Proc. 12th IEEE International Conference on Communications (ICC'13),* June.pp. 1-5.

M.B., N., 2014. Dynamic Load Balancing in Software-Defined Networks. *Master's thesis, Aalborg University.*

Mahanta, D., Ahmed, M. & Bora, U., 2013. A study of Bandwidth Management in Computer Networks. *International Journal of Innovative Technology and Exploring Engineering,* 2(2).

Mondal, R., Ray, P., Nandi, E. & Sarddar, D., 2016. Load Balancing Algorithm with Total Task of Different Nodes. *International Journal of Computer Science and Information Technology & Security (IJCSITS),* 6(1).

Namal, S., Ahmad, I., Gurtov, A. & Ylianttila, M., 2013. Sdn based inter-technology load balancing leveraged by flow admission control. *Future Networks and Services (SDN4FNS), 2013 IEEE SDN,* pp. 1-5.

Open Networking Foundation, 2015. Openflow switch specification, version 1.3.5.

Patel, G., Athreya, A. & Erukulla, S., 2013. OpenFlow based dynamic load balanced switching.

Röpke, C. & Holz, T., 2016. On network operating system security. *International Journal of Network Management,* 26(1), pp. 6-24.

Saranya, D. & Maheswari, L., 2015. Load Balancing Algorithms in Cloud Computing: A Review. *International Journal of Advanced Research in Computer Science and Software Engineering,* July.5(7).

Saur, K. et al., 2016. Safe and Flexible Controller Upgrades for SDNs. *Proceedings of the Symposium on SDN Research (SOSR).*

Schiff, L., Schmid, S. & Kuznetsov, P., 2016. In-Band Synchronization for Distributed SDN Control Planes. *ACM SIGCOMM Computer Communication Review,* 46(1), pp. 37-43.

Senthil, N. & Ranjani, S., 2015. Dynamic Load Balancing using Software Defined Networks. *International Journal of Computer Applications.*

Suguna, S. & Barani, R., 2015. Simulation of Dynamic Load Balancing Algorithms. *Bonfring International Journal of Software Engineering and Soft Computing,* July.5(1).

Tootoonchian, A. & Ganjali, Y., 2010. HyperFlow: A distributed control plane for OpenFlow. *Proceedings of the 2010 internet network management conference on Research on enterprise networking,* April.p. 3.

Trajano, A. & Fernandez, M., 2016. uLoBal: Enabling In-Network Load Balancing for Arbitrary Internet Services on SDN. *ICN 2016 : The Fifteenth International Conference on Networks.*

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

Wang, R., Butnariu, D. & Rexford, J., 2011. OpenFlow-Based Server Load Balancing Gone Wild. *Hot-ICE,* Volume 11, pp. 12-12.

Wang, T., Liu, F., Guo, J. & Xu, H., 2016. Dynamic SDN Controller Assignment in Data Center Networks: Stable Matching with Transfers. *Proc. of INFOCOM.*

Watson, J., 2008. Virtualbox: bits and bytes masquerading as machines. *Linux Journal, 2008,* Issue 166, p. 1.

Yahya, W., Basuki, A. & Jiang, J., 2015. The Extended Dijkstra's-based Load Balancing for OpenFlow Network. *International Journal of Electrical and Computer Engineering (IJECE),* April, 5(2), pp. 289-296.

Yang, J., Ling, L. & Liu, H., 2016. A Hierarchical Load Balancing Strategy Considering Communication Delay Overhead for Large Distributed Computing Systems. *Mathematical Problems in Engineering, 2016.*

Yao, H., Qiu, C., Zhao, C. & Shi, L., 2015. A multicontroller load balancing approach in software-defined wireless networks. *International Journal of Distributed Sensor Networks,* Issue 454159.

Zhong, H. et al., 2015. An Efficient SDN Load Balancing Scheme Based on Variance Analysis for Massive Mobile Users. *Mobile Information Systems, 2015.*