S	
E1	
E2	
For Office Use Only	

Masters Project (MCS) Final Report

2019

Project Title	Coordination and computation paradigm on Software Defined Network
Student Name	R. D.Y. Thrilakshi
Registration No. & Index No.	14440816 & 2014MCS081
Supervisor's Name	Dr. D.N. Ranasinghe

For Office Use ONLY

Declaration

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Student Name: R.D.Y. Thrilakshi

Registration Number: 14440816

Index Number: 2014MCS081

Signature: Date: 2019/05/31

This is to certify that this thesis is based on the work of ~~Mr~~./Ms. R.D.Y.Thrilakshi under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name:

Signature: Date:

Coordination and computation paradigm on a Software Defined Network

A dissertation submitted for the Degree of Master of Computer Science

R. D. Y. Thrilakshi
University of Colombo School of Computing
2019

Acknowledgements

First and foremost, I am deeply indebted to my advisor Dr. D.N. Ranasinghe, senior lecturer at University of Colombo School of computing, for valuable advice, the guidance and support given me to continue my research work. In addition, I am thankful to have this opportunity to study regarding distributed systems with software defined networking, protocols and topologies. Finally, I am extremely grateful to my loving parents, family who extended the emotional support and constantly encouraged me during all times.

Abstract

Since SDN is highly programmable and decoupled the controller plane from the data plane, SDN has played a major role to provide some promising solutions to the traditional network. Mainly the controller has taken the flow management of the network by considering the whole network status and achieved performance improvements, high -flexibility and efficient configuration. Therefore, SDN has emerged as a trending approach for the network application development. Even though the SDN was able to minimize burning problems in a traditional network, SDN still is in its experimental level to support scalability, single point of failure etc. This thesis discusses the early practices of SDN by highlighting its architectural concepts with the evolution of software defined network. Further it examines the technologies which supports SDN such as OpenFlow, network virtualization and mininet. According to the statistics gained from the research evaluation, it has proven that the multiple controllers were able to reduce the so-called controller overhead than the single controller. Furthermore, the implementation of network function virtualization depicts the notion of SDN and simple emulation using mininet shows the practical aspect of this thesis.

Table of Contents

List of Figures 3

List of Tables 4

List of Abbreviations 5

Chapter 1 6

1. Introduction 6

1.1 Motivation 6

1.2 Objective 7

1.3 Scope 7

Chapter 2 8

2. Background 8

2.1 Traditional Network vs Software defined Network 8

2.2 Controller Throughput 10

2.3 Controller Response Time 10

2.4 What is OpenFlow? 10

2.5 About Mininet 12

2.6 Network Virtualization 12

2.7 Literature Review: 13

2.7.1 POX Controller 13

2.7.2 Open Daylight Controller 13

2.7.3 BEACON Controller 14

2.7.4 Kandoo 15

2.7.5 Hyper flow 16

2.7.6 DIFANE 17

2.7.7 SDN Architecture with logically centralized multiple controllers - ONIX 18

2.7.8 Hierarchical control plane architectures – Orion 18

2.7.9 Optimal Flow 19

Chapter 3 21

Experimental Setup 21

Experiment Result Evaluation 26

Chapter 4 39

Conclusion 39

Future work 40

References 41

List of Figures

Figure 1: Traditional network running distributed protocol ... 8
Figure 2: Control plane and forwarding plane of an SDN ... 9
Figure 3: How the request goes from host x to host y ... 9
Figure 4: OpenFlow switch communicates with a controller over a secure connection using the OpenFlow protocol ... 11
Figure 5: Open Daylight Framework ... 14
Figure 6: Kandoo's design ... 15
Figure 7: Hi-level overview of Hyper Flow ... 17
Figure 8: DIFANE flow management architecture ... 18
Figure 9: Network view of the hybrid hierarchical architecture ... 19
Figure 10: Single Controller Architecture ... 22
Figure 11: Multiple Controller Architecture using two controllers ... 23
Figure 12: Multiple Controller Architecture uisng three controllers ... 23
Figure 13: Open Daylight Controller ... 24
Figure 14: Total Time ... 28
Figure 15: Total Packets ... 28
Figure 16: Minimum Delay ... 29
Figure 17: Maximum Delay ... 29
Figure 18: Average Delay ... 30
Figure 19: Average Jitter ... 30
Figure 20: Delay Standard Deviation ... 31
Figure 21: Bytes received ... 31
Figure 22: Average Bit rate ... 32
Figure 23: Average Packet Rate ... 32
Figure 24: Packets Dropped ... 33
Figure 25: Average Loss-Burst Size ... 33
Figure 26: Average performance in Total Time ... 34
Figure 27: Average performance in total packets ... 35
Figure 28: Average performance in minimum delay ... 35
Figure 29: Average performance in maximum delay ... 35
Figure 30: Average performance in average delay ... 36
Figure 31: Average performance in average Jitter ... 36
Figure 32: Average performance in delay standard deviation ... 36
Figure 33: Average performance in bytes received ... 37
Figure 34: Average in average bit rate ... 37
Figure 35: Average in average packet rate ... 37
Figure 36: Average in packet dropped ... 38
Figure 37: Average in average loss-burst size ... 38

List of Tables

Table 1: Single controller experiment result 26
Table 2: Multiple controller experiment with two controllers 27
Table 3: Multiple controller experiment with three controllers 27
Table 4: Average performance matrix 34

List of Abbreviations

- SDN: Software Defined Network
- LAN: Local Area Network
- MAN: Metropolitan Area Network
- NFV: Network Function Virtualization
- VNF: Virtual Network Functions
- NAT: Network Address Translation

Chapter 1
1. Introduction

1.1 Motivation

The newly defined split architecture computing model called Software Defined Networks quickly has become a topic of interest in the domain. In situations where parallel applications are presented in large scale distributed systems, the software defined network is an emerging network architecture where network control plane is decoupled from forwarding plane and directly programmable with the aid of network virtualization. It supports to show the benefits of creating network infrastructure which would be more agile and flexible.

Another aspect of the distributed system is, full potential massively parallel systems requires programming models that deal with the concurrency of cooperation among very large-scale network. This has led to design and implementation of coordination models and their associated programming languages. However, they also differ in how they precisely define the notion of coordination, what exactly is being coordinated, how coordination is achieved. In such a situation, our motivation is to work out with the coordination and data driven computation using multiple controllers on a software defined network.

Since the SDN controller is directly programmable, our motivation opens the research work to apply the coordination computation paradigm on SDN controller. The SDN controller has raised many problems and one of the most voiced concern is the controller bottleneck. This newly focusing approach is to check whether it support to minimize the existing burning performance bottleneck in the controller with the large number of incoming requests. To prevent this kind of bottleneck, it is more generally to improve the performance of the controller. Therefore, the research is to find the architecture for the controller to create the kind of dynamic topology to control the requests of the data plane nodes with the help of coordination computation paradigm. The controller can change the data traffic rules on the fly if it wanted to reduce the traffic overhead in and among the data plane nodes (switches or routers).

This paper opens with the discussion of SDN by describing its background and network elements that are part of SDN architecture. In addition, it describes the utilized simulation tool which is called Mininet and OpenFlow protocol which is used by the network nodes to communicate within the network. Also, it describes the utilized existing controller architectures such as DIFANE, DEVFLOW, KANDOO etc.

In section 3, it is continuing the discussion on experimental set up to evaluate the network statistics using multiple controllers against the benchmarking single controller. Last section concludes the paper with the discussion and the possible future work.

1.2 Objective

Main objective of this research is to enhance the performance of the software defined network controller by offering comprehensive coordination and computation paradigm for SDN controller and control the requests of the data plane nodes. By enhancing the number of requests, our expectation to do an experiment to minimize the controller overhead and the latency of the controller through this approach.

Since the software defined network is having the ability to network program, it enables to separate the network devices' data plane from controlling plane. By having the periodical state of the network devices, the controller maintains the global view of the network. Having a logically centralized controller and multiple controllers has been involved in order to avoid network issues such as controller bottleneck and single point of failure. In this study we identify several inefficient points in Software defined network and propose SDN based controller architecture to avoid those inefficiencies through the concept of separating coordination and computation of the network. Since Mininet supports Open flow, this research work will demonstrate the existing behavior of the SDN and the new behavior of the SDN with coordination and computation paradigm with multiple controllers. This research will end up with comparison discussion with the effectiveness of using coordination and computation on SDN with multiple controllers.

1.3 Scope

This study improves the performance of the existing software defined network paradigm using multiple controllers. The controllers would handle the requests based on event categories and establish forwarding tables on network nodes. Specific controller will handle specific set of switches. This research is focus only on wired network.

This research will lead to a simulation work using mininet and comparison on benchmarking single controller SDN against multiple controller SDN using Open Daylight.

Chapter 2

2. Background

2.1 Traditional Network vs Software defined Network

Computer systems linking each other to share the data is known as a computer network and basically the network can be classified as local area network(LAN) which is in one geographical area, wide area network(WAN) which is in different geographical area and Metropolitan Area Networks(MANs) which is implemented for a city. In a traditional network, the protocols have been used to communicate among network nodes by determining the routing flows for the end hosts and periodically the protocol shares the network status of the switches over the network. In a situation like network failure, the protocol needs to propagate new routing paths to prevail the situation. Since there is no abstraction level visibility of the nodes, it is difficult to change the routing flows and difficult to debug the network fault such as packet losses and network looping.

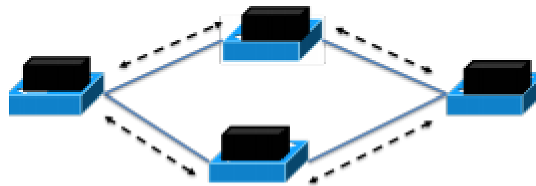

Figure 1:Traditional network running distributed protocol

Simply the SDN decouples the control plane from its forwarding plane while maintaining the whole network behaviour according to the decisions made by the logically centralized root controller. Rather than a single node in a network making its own forwarding decisions, the controller of the SDN is responsible for maintaining forwarding tables of the network nodes.

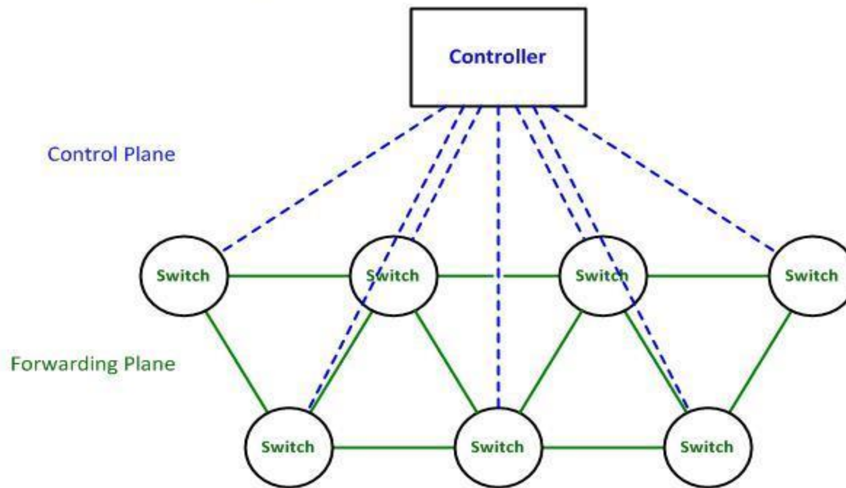

Figure 2:Control plane and forwarding plane of an SDN

Even though in a normally network the individual network node does not know how the entirely network looks like, in SDN, central controller can see the whole network and install forwarding decisions to each subordinate switch based on their destination path. Since the central controller is giving capability of controlling resources using convenient programming interface, as an example, if we want to move VM to one host to another, controller can instruct to any firewall to migrate. This removes the need of system administrator involvement of the network reconfiguration

.

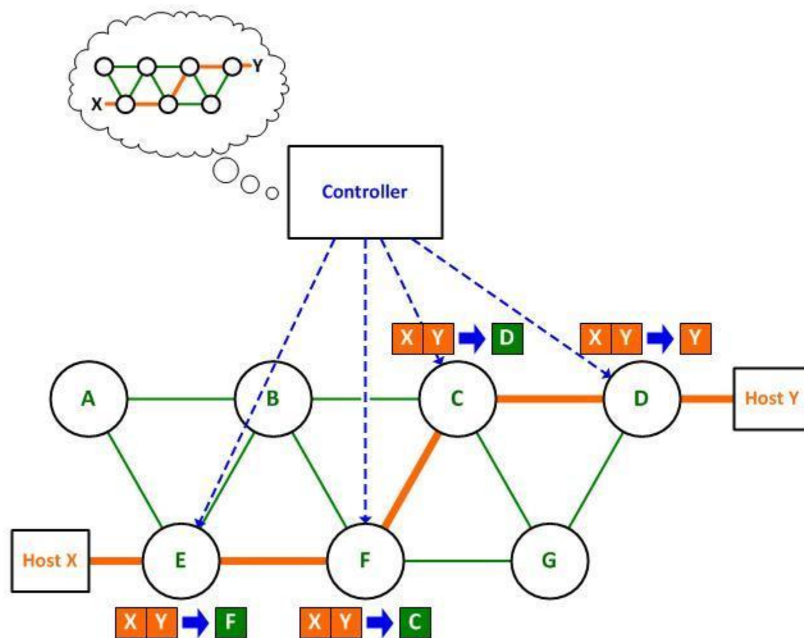

Figure 3:How the request goes from host x to host y

In the context of SDN, by taking the advantage of the network programmability of SDN, several efforts have been taken to achieve active networking behaviour. One of such effort is called SwitchWare that allows packet flowing by modifying the network dynamically and some attempts are there to program the software routers. Separation of control and data plane is one of the uniqueness of SDN architecture and it has achieved to take out the network intelligence from switches and put into controller since the controller has the whole network visibility. This brings the improvements in performance, optimization in network configuration and ability to define virtual network with less effort.

2.2 Controller Throughput

Throughput is the most important factor of the network and our concern is lead to an architecture where we can have maximum throughput from the controller. In that case we need to identify the no. of controllers is needed to handle the control load of the network. The controller named as NOX-MT shows that the maximum throughput can be gained by obtaining multiple threaded controllers [13].

Though the throughput should not be affected by the no. of switches or threads, however it degraded the performance of controller with the scheduling overhead within a controller. Since I/O handling increase and the shared resources are increased, I/O and job batching will be less effective [13].

2.3 Controller Response Time

The controller response time is corresponding to the load levels. Controllers such Maestro [3] and Beacon are affected from this workload. But some network applications like NOX-MT has portioned the network's MAC address into hash table to minimize the workload [13]. In SDN, controller response time also depends on flow completion time. Same as the no. of switches increase for the controller, it is increasing the response time as well.

2.4 What is OpenFlow?

OpenFlow is a protocol which is used to communicate among network nodes. In OpenFlow enabled network switch contains a flow table which is having flow entries to perform packet lookups and packet forwarding. The controller manages adding, updating and deleting flow table entries via OpenFlow channel both reactively and proactively. A flow entry defines a unique flow in a flow table by using its components such as “Match Fields”, “Priority”, “Counters”, “Instructions”, “Timeouts”

and “Cookie Flags”. If the transmitted packet matches any of the flow entry in the table, the instructions associated with the specific flow entry is executed according to the priority. Each flow table should configure the flow entry to handle the unmatched packets and such kind of situation is known as a table-miss. Normally if there is a table-miss, the packet will be transferred into the controller, drop the packet or redirect to the subsequent table. The controller maintains the responsibility of adding or removing the table-miss flow entry to the flow table since it is not a inbuilt entry of a flow table.

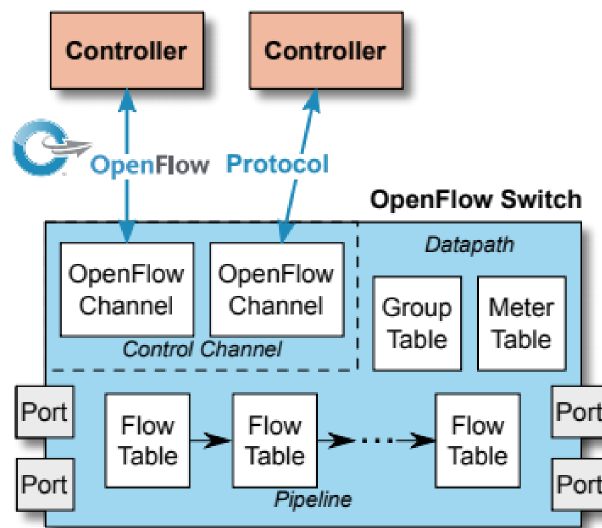

Figure 4:OpenFlow switch communicates with a controller over a secure connection using the OpenFlow protocol

For passing the network packets over the network, the OpenFlow [22] switches use the interface called OpenFlow port. The packet will be sent out via the output OpenFlow port and received via ingress OpenFlow port. Once the packet comes to the ingress OpenFlow port, it is processing through the OpenFlow pipeline before the packet transmitted to the output port. OpenFlow pipeline can decide how the packet should proceed into the network according to the output action.

2.5 About Mininet

Implementing a network with large number of network nodes is quite difficult and costly. In order to avoid this problems, virtual mode strategy was proposed to prototype and emulate the network. One such open source network simulator is named as Mininet. Mininet was created using Python and it uses python APIs for customization of user requirements [7]. Mininet network simulator can be used to simulate SDN switches and hosts where it emulates the OpenFlow network and end hosts within a single machine. It supports both common topologies and custom topologies [14]. Mininet switches which are running on Linux can support OpenFlow, but still non-Linux compatible switches or OpenFlow switches are not supported by mininet [7].

Mininet has the capability to emulate different kinds of network elements such as; host, layer-2 switches, layer-3 routers, and links. It works on a single Linux kernel and it utilizes virtualization for the purpose of emulating a complete network utilizing only a single system. However, the created host, switched, routers, and links are real-world elements although they are created by means of software[14].One of the key feature of mininet is its software-based Open Flow switches in a virtualized containers, providing the exact same semantics of hardware-based OpenFlow switches[16].

Characteristics of Mininet:

- Flexibility: can set up new topologies using programming languages.
- Applicability: even a prototype implementation can be used in a real network with or without having any modification in source codes.
- Interactivity: real time simulation
- Scalability: can be scaled up to large networks with hundreds or thousands of nodes on a computer.
- Realistic: prototype behave with high confidence, so that applications can use without any code modification.
- Share-able: easy to share the prototype with other collaborators.

2.6 Network Virtualization

Network virtualization is the concept of represent entire network nodes that may connect to create communication services on one or more virtual machines instead of having separate hardware for each network function.

2.7 Literature Review:

This review presents the discussion on existing OpenFlow enabled controllers such as POX, Beacon, Kandoo, Hyper flow, DIFANE, ONIX, ORION and Optimal Flow Controllers and it will focus on the methodologies they have used and important facts such as performance, scalability and reliability.

2.7.1 POX Controller

NOX was an open source development platform which has used C++ for the implementation where most of the SDN functionalities have been implemented using python. And, NOX has shown some drawbacks in backward incompatibility. In order to overcome this situation, the new framework called POX has come to the SDN platform. It is a variant for python development to write an OpenFlow controller which is easier than NOX. POX components have been developed using python and those functions are bundled with the mininet as well.

With the experimental study for the controllers, POX controller could not achieve the scalability since it did not support for multi-threading. Python-based controllers such as POX is more suitable for fast prototyping than for enterprise deployment. When packet messages coming, the POX controller detects invalid values of ARP header fields [22]. In a network, the response time of a controller is very important fact. The average response time has the correlation with the no. of connected host. But according to the researches which have been done in the past has stated that there is a smallest latency in POX controller than the other controllers such as Beacon controller [22].

2.7.2 Open Daylight Controller

Open Daylight is yet another OpenFlow support controller which offers ready-to-install network solutions. Because of the opensource nature of this controller, it minimizes the controller operation complexity. Due to that, it extends the lifetime of the infrastructure. When we consider the architecture of the Open Daylight, it has created a multilayered architecture. The controller layer is the most powerful layer since it controls the whole network traffic according to the flow tables. Open daylight [6] can run on any operating system like JAVA.

The implementation of the Open Daylight application has been done by considering two approaches as below.

- The API-Driven SAL (AD-SAL)
- The Model-Driven SAL (MD-SAL)

AD-SAL approach is considered as stateless and it is limited only for flow capable devices and services. The flow programming in this approach is reactive and handled by considering the received events. Since MD-SAL approach uses REST APIs for all the modules, not like AD-SAL, MD-SAL supports any device or any services. And, the flow programming of this approach is proactive without receiving any events from the network.

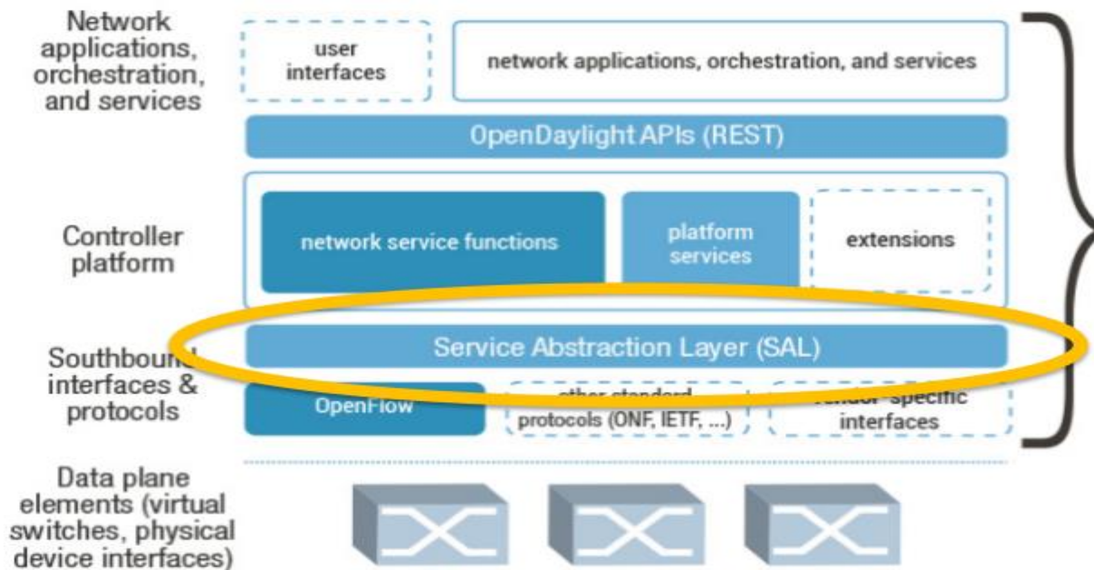

Figure 5: Open Daylight Framework

2.7.3 BEACON Controller

Beacon is the java based multithreaded controller [24]. In large scale network, multiple instance of the controller manages the controller bottleneck and managing the distributed control plane or one of the common approaches is to use multithreading [13]. Some of the useful features of the Beacon controller are run time modularity, fast and multithreaded. Also, the SDN controller performance evaluation statistics reveals that the maximum throughput can be gained from Beacon controller [13].

2.7.4 Kandoo

Realizing the overhead of the frequent events, Kandoo has implemented the alternative routing mechanism to minimize the controller overhead without modifying the switches. It has divided the controller layer into two, where the bottom layer consisted with the controller which does not have any interconnection between controllers and no knowledge on the network wide status and the top layer consisted of the logically centralized controller. The bottom layer helps to avoid the controller bottleneck on the top layer.

In Kandoo implementation, it provides local controllers which process events locally and the logically centralized controller which process nonlocal events. Logically centralized root controller takes care of the local controllers. Local controllers are switches that gives switch proxies to the root controller which can be implemented using OpenFlow switches. Since Kandoo is not considering the network wide state, if such requirement is there for the network, we need to go for the implementation such as Hyper flow or Onix. Kandoo gives the flexibility to network operator to configure the control plane based on the characteristics.

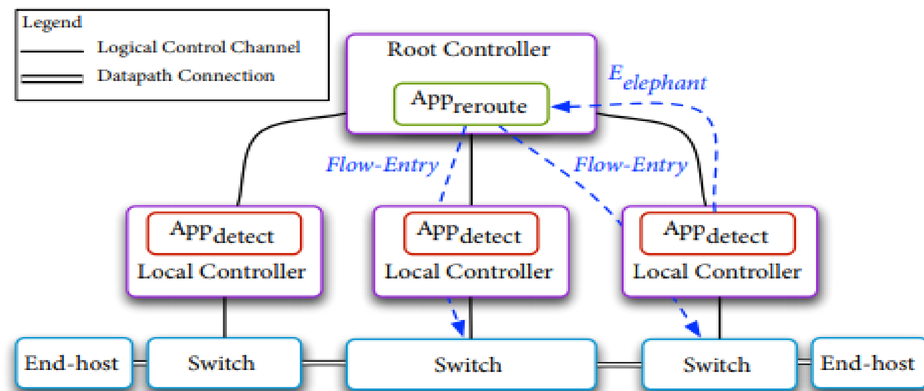

Figure 6:Kandoo's design

When we investigate the Kandoo implementation, it has mainly focused on two goals. First goal was, Kandoo should compatible with the OpenFlow and it should be able to distribute the applications without having any manual intervention and only Kandoo needs to know whether the control application is local or non-local. Therefore, the developers do not need to worry about how the applications are distributed over the network and they would see that applications are controlled by a logically centralized controller.

In Kandoo there are two applications called AppDetect and AppReroute where appdetect queries for the switches to identify the elephant flows and appreroute is to install flow entries on switches when needed.

In this architecture, one controller can control many switches and one switch will be controlled by one controller. In Kandoo architecture, the flow control is not always bottom up and the controller can also place sends the network topology to the local controllers by considering the event types. Basically, the event handling is the most important part in the Kandoo [28] Architecture, and it sets up the data path proactively while the elephant flow detection sets up adaptively. The implementation of Kandoo has been done using C, C++ and python in modular based to support the plugins and at any time the back end can be replaced using any other supplies. The application itself has developed with the repository and when the system boots up, it downloads the application informations from the repository and get the runnable applications. Kandoo has proven that the single node controller can perform the 1M packets per second using 512 switches on a single threaded xeonE7-4807.

2.7.5 Hyper flow

The network with a single controller has several drawbacks such as it can handle limited requests with the limited bandwidth and the time it takes to set up the flow paths will be significantly larger. Therefore, to get rid from the scalability problem, hyper flow is yet another distributed controller architecture which has push the network status into all controllers. Each controller should think as it is the only controller to the network while synchronizing with each other.

Hyper flow [29] can be identified as two major components such as C++ NOX controller application and the publisher/subscriber event propagation system. In this mechanism, switches are connected to the closest controllers and at any point of controller failure, switches will be connected to another controller by changing its configurations.

In hype flow, event loggers capture the events and publish to the publisher/subscriber and event players deserialized and replay captured events. And also, it uses command proxy to identify relevant switch for the request and send the response back to the place where it comes from. Publisher/Subscriber system maintains a network wide state using three channel types as data channel, control channel individual controllers implemented using wheekFS.

Hyper flow is not like Kandoo and it uses network wide statistics to modify the switches to reroutes the packets and significantly it handles few thousand packets per second. Therefore, Hyper flow is more resilient for network partitioning.

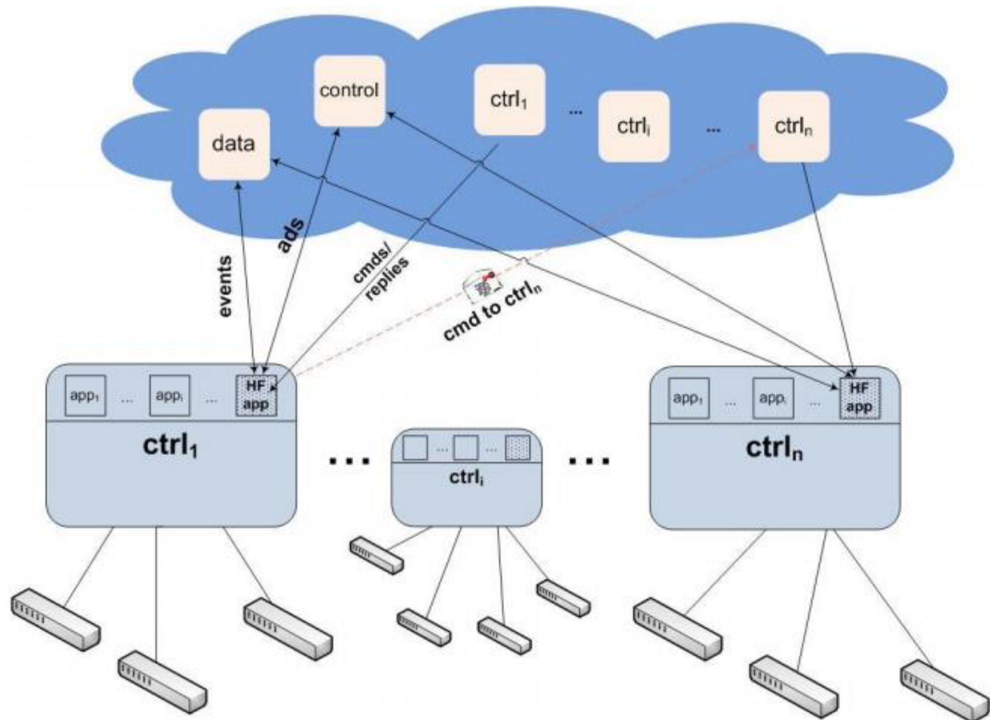

Figure 7:Hi-level overview of Hyper Flow

2.7.6 DIFANE

Difane provides the efficient, scalable network packet forwarding mechanism using the switches which has been installed set of rules to forward and drop the network packets. Here the challenge is, it is difficult to the change the rules every time. So that, the controller pre-compute the rules to determine which low-level rules to apply on which authority switches. DIFANE basically has two main ideas such that the controller should distribute rules among its authority switches by using partitioning algorithm and the packets are handled by the switches in the data plane. DIFANE [2] achieves flow-based management by installing the low-level rules in advance and it uses hi-level decisions to reduce the cache misses to improve the scalability.

In this architecture, it considers caching rules in the switch as an unnecessary burden for the switch because there may be a packet delay or some complexity on the switch when cache misses take place. So that, DIFANE uses some wildcard mechanism to handle cache misses effectively by keeping those cache misses in the data plane.

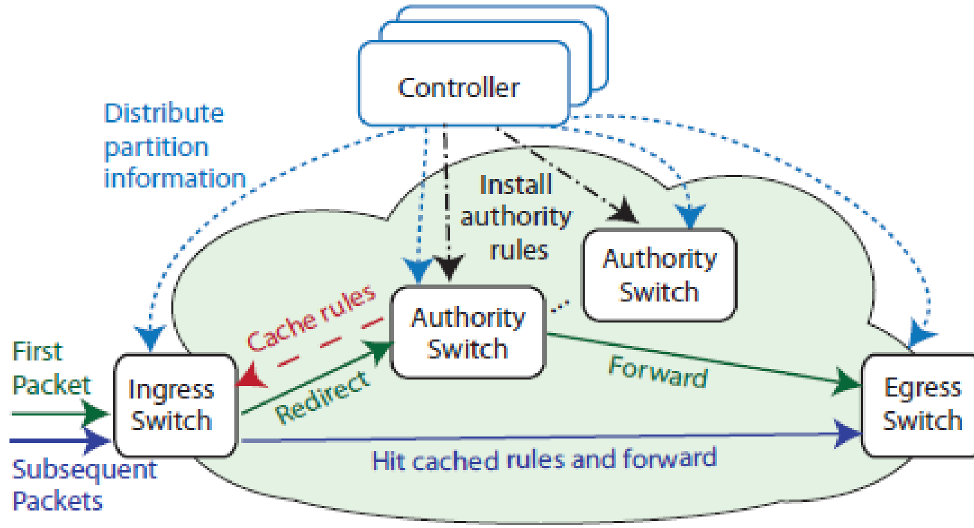

Figure 8: DIFANE flow management architecture

2.7.7 SDN Architecture with logically centralized multiple controllers - ONIX

In SDN, still there are issues where some critical requirements cannot be achieved. Using one controller, efficiency cannot be reached up to the expected level. Scalability is the most important factor that pushes network architectures into multiple controllers. The distributed control plan such as ONIX is running multiple ONIX instances and ONIX API [28] allows control applications to read and write the state of any network element. Also control logic records forwarding information from the switches. As a result of evaluation studies reveals that the ONIX provides scalability by partitioning the network logically by distributing the workload.

2.7.8 Hierarchical control plane architectures – Orion

There are two kind of SDN controller architectures such as the flat control architecture and the other one is hierarchical control plane architecture. Hierarchical control plane architecture has been introduced in order to improve the scalability problem in large scale network since the flat control architecture fails to minimize the computational complexity of the controller when the network size is getting large.
Orion is one of the systems which uses hybrid hierarchical control plane for large scale network. It reduces the controller plane computational complexity from super linear to linear by using abstracted hierarchical network views.

Orion is focusing on intra domain routing management system. Basically, it is consisted with three layers such as ‘Network Device Layer’, ’Area controller Layer’ and ‘Domain Controller Layer’. The whole network is referred as domain and the domain is divided into sub domains which the sub domain closes to each other and each domain is controlled by an SDN controller.

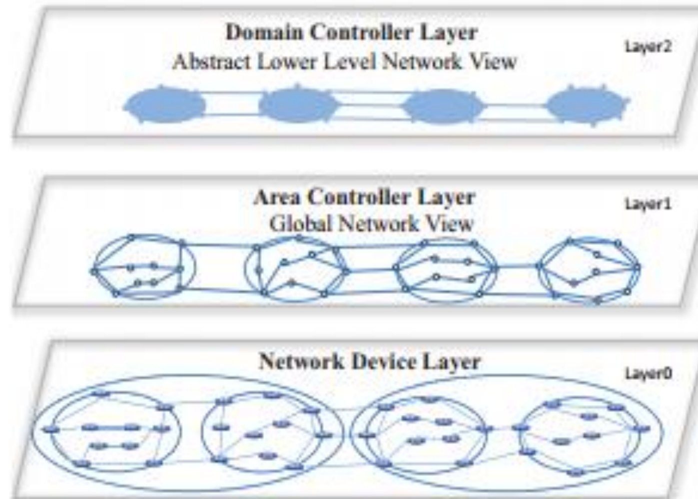

Figure 9:Network view of the hybrid hierarchical architecture.

The network device layer consisted of large amount of open flow switches and the middle layer is responsible to gain the device and link information from the network device layer. It processes the routing of the requests and create abstract view of the area network and send it to top level layer which is known as domain controller layer which synchronizes among area controllers through distributed protocol. This area division of Orion has gained the opportunity to reduce the computational complexity of large-scale network system.

2.7.9 Optimal Flow

Optimal flow controller is another controller system which has been used in Industrial controller system which facilitates to provide innovative applications like robust voltage control, renewable energy programs and electric vehicles. Those ICS requires the normal functioning even in the failures or disturbances such as cyber-attacks. By considering the issues, Optimal Flow is a proposed system for ICS where it monitors the single SDN domain and reroute the requests according to the integer linear programming (ILP) optimization problem.

While ILP provides shortest path routing decisions, Optimal Flow contains two interfaces to achieve hierarchical control plane. One interface is called northbound interface where it contains switched infrastructure which communicate through the open flow protocol and it exposes the edge ports into the upper tier. The southbound interface of the optimal flow is connected to the open flow controller, which monitored the SDN switches in the network. The optimal flow uses an algorithm to identify the affected flows due to disturbances and provision the flows according to the priority to disconnect the low priority flows. Also, it constructs the dependency network graph in order to update the network to avoid link congestion.

Chapter 3

Experimental Setup

This thesis conducts the practical implementation of an SDN using network function virtualization and it has used mininet emulator for prototyping of the SDN. The environment has been setup on MacOS 10.14.3 host machine with two virtual box hosted Ubuntu VMs. One Ubuntu VM contains open daylight controller and other one is a mininet installed server. Later in the implementation, separate VMs have been spawned in need of a new controller with different IP address and same set of configurations. The virtual box configured two network adapters; one adapter enabled NAT and the other one used as a host only adapter. To achieve the goal of performance evaluation, different network topologies were implemented in both single controller network and the multiple controller network with the aid of mininet which uses python scripts for network simulation.

Tools used:

1) Mininet: Python based network emulator to create virtual network which is topology-aware and OpenFlow-aware.

2) Miniedit: Miniedit is an experimental tool which comes for mininet with a simple GUI to demonstrate how the mininet can be extended. It has provided the flexibility to create and simulate the custom software defined network. Before running the miniedit, it is needed to start mininet VM and connect via SSH.

Basic commands used:

- $ sudo ~/mininet/examples/miniedit.py - To run Miniedit
- $ sudo ovs-ofctl dump-flows s1 -To check the flow table on switch1

Single Controller Architecture: Series 1

Single controller will be the benchmark for the statistic evaluation. In this network, there would be a single controller od1 with three OpenFlow enabled switches as S1, S2, S3 with 6 hosts naming h1, h2, h3, h4, h5 and h6.

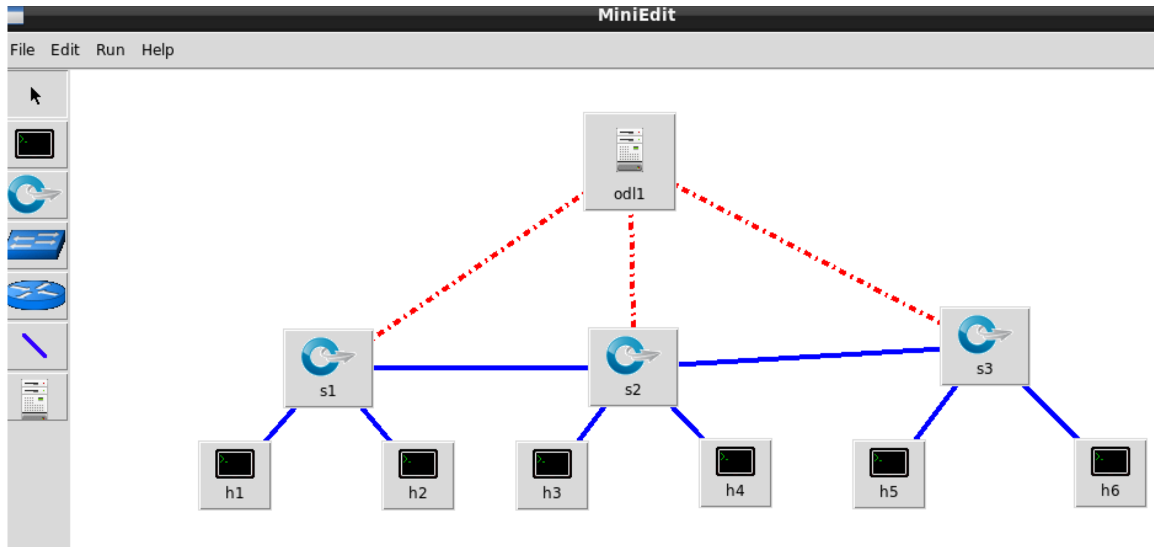

Figure 10: Single Controller Architecture

Multiple Controller Architecture (Two Controllers): Series 2

In this network, there would be two controllers as odl1 and odl2 to manage three switches as S1, S2 and S3 with the 6 hosts naming h1, h2, h3, h4, h5 and h6.

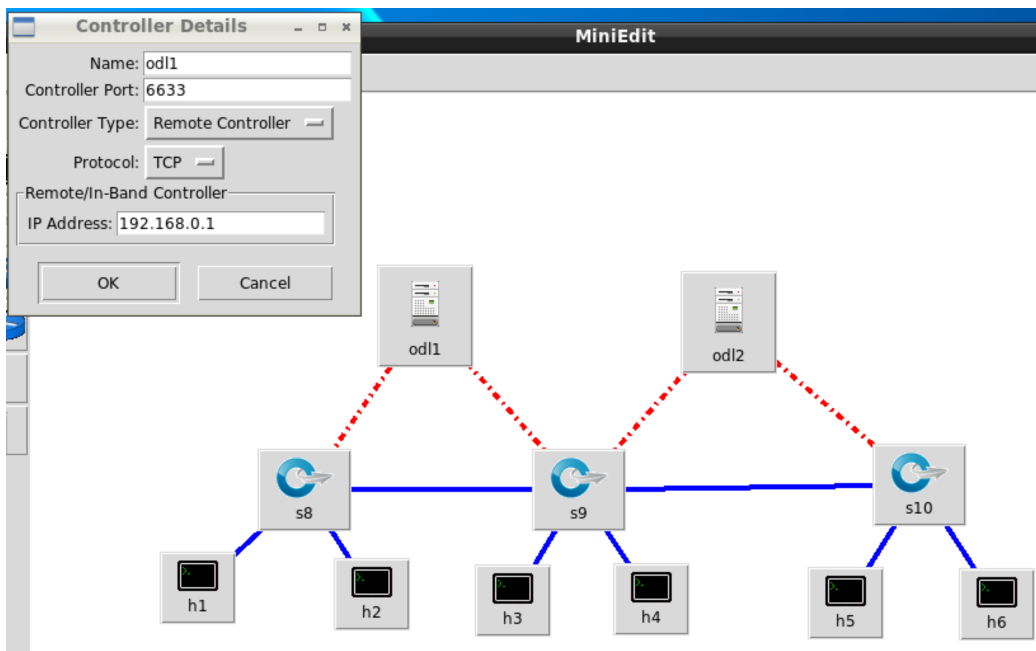

Figure 11: Multiple Controller Architecture using two controllers

Multiple Controller Architecture (Three Controllers): Series 3

In this network, there would be three controllers as odl1, odl2 and odl3 to manage three switches as S1, S2 and S3 with the 6 hosts naming h1, h2, h3, h4, h5 and h6.

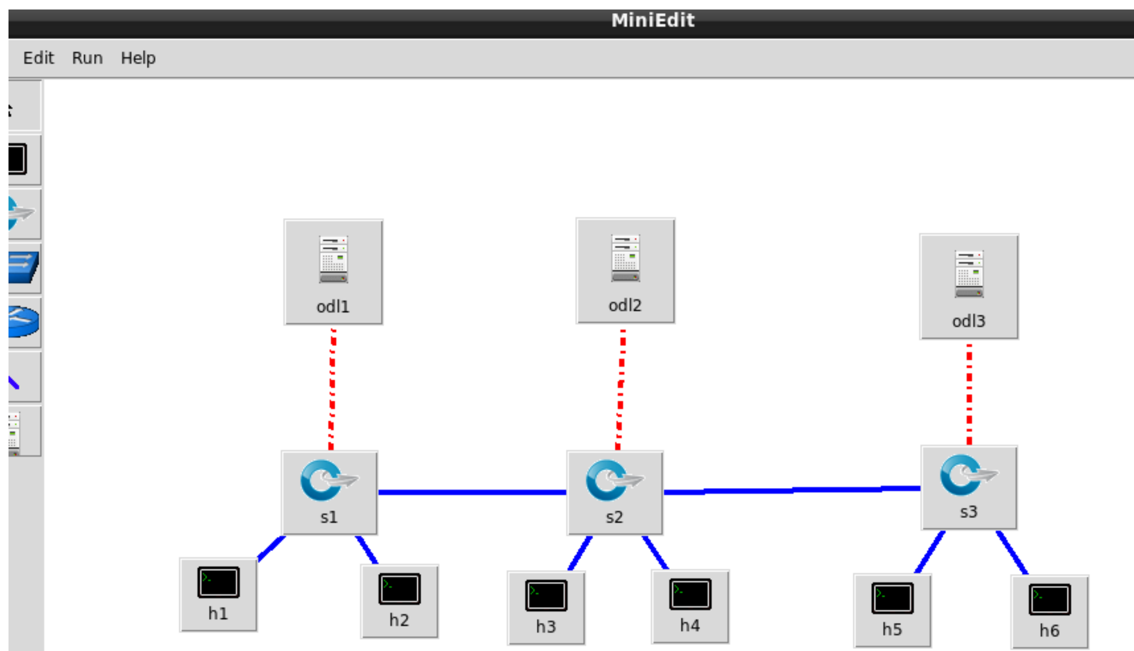

Figure 12: Multiple Controller Architecture uisng three controllers

IP Address Setup

odl1 - 192.168.0.2
odl2 - 192.168.0.3
odl3 - 192.168.0.4

S1 – 192.168.0.20
S2 – 192.168.0.21
S3 – 192.168.0.22

h1 - 192.168.0.11
h2 - 192.168.0.12
h3 - 192.168.0.13
h4 - 192.168.0.14
h5 - 192.168.0.15
h6 - 192.168.0.16

Open Daylight controller used for the experiment since it supports multiple controller architecture and contains easy to use web interface where we can see the topology related information. We send UDP packets from h1 to h6. Logs will be generated both sender and receiver side. All the logs are being generated by each end will be saved in h1. Thus, h1 listening for logs where h6 is listening for the UDP packets. We keep the bites capacity as unique (512) and we change the packet rate.

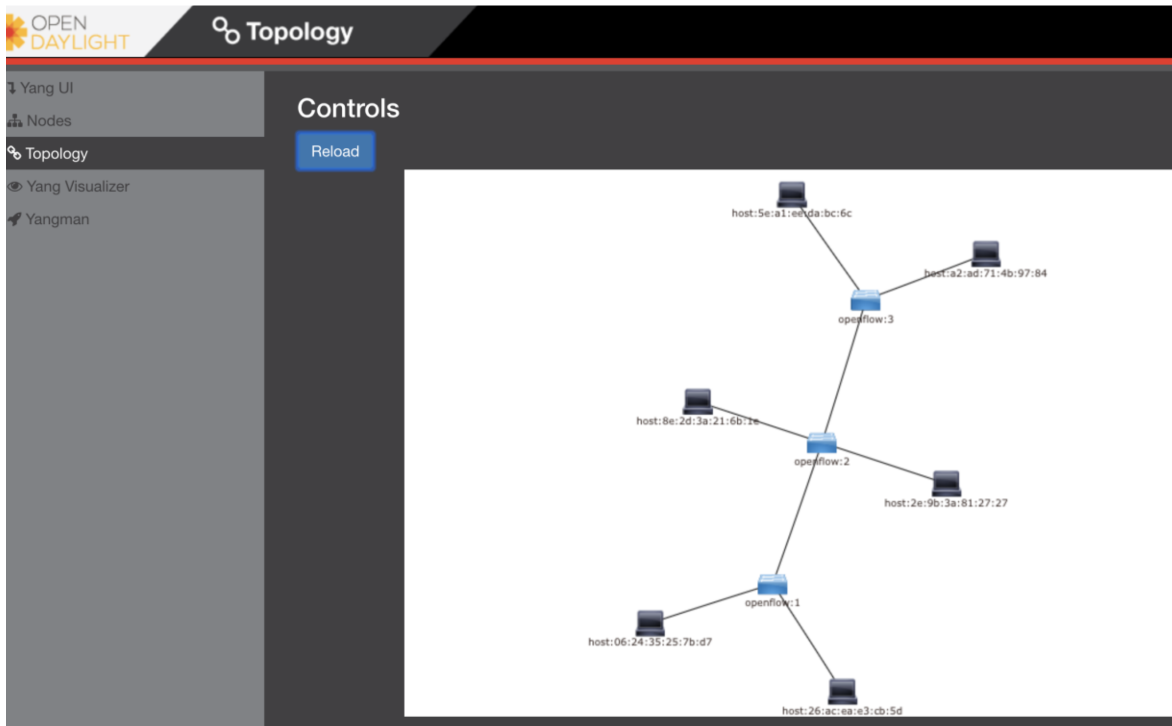

Figure 13: Open Daylight Controller

3) D-ITG Tool: D-ITG is a distributed internet traffic generator platform which supports both IPv4 and IPv6. Since D-ITG is compatible with the operating system Linux, this research used D-ITG to generate traffic on switches. Set of commands are there to send and receive network packets.

- $. /ITGLog - To start the log server
- $. /ITGRecv – To start the receiver
- $. /ITGSend <traffic_configuration_file> -l s<sender_log_file> -L 192.168.0.11 UDP -X 192.168.0.11 UDP -x <receiver_log_file> - To start the sender
- Ctrl+C - To close the receiver and the log server
- $. /ITGDec <receiver_log_file> – To decode the receiver log file.

Below is the script which was created to generate traffic simultaneously.

```
-a 192.168.0.16 -rp 1001 -C 20000 -c 512 -T UDP
-a 192.168.0.16 -rp 1002 -C 40000 -c 512 -T UDP
-a 192.168.0.16 -rp 1003 -C 60000 -c 512 -T UDP
-a 192.168.0.16 -rp 1004 -C 80000 -c 512 -T UDP
-a 192.168.0.16 -rp 1005 -C 100000 -c 512 -T UDP
```

rp – receiver port, -C – no. of packets per second, -c – no. of bits

Steps of the experiment. (All the the D-ITG commands will be listed below)

1)Start the mininet virtual machine with IP address 192.168.0.1

2)Start the Open Daylight controller VMs with IP address 192.168.0.2, 192.168.0.3, 192.168.0.4

3) Start miniedit using sudo ~/mininet/examples/miniedit.py

4)Load mininet model

5) Run below commands for each mininet model (Series1/Series2/Series3)

1. start the log server at h1

   ./ITGLog

2. start the UDP listener at h6

   ./ITGRecv

3. send packets from h1 to h6

   ./ITGSend traffic1 -l senderlog -L 192.168.0.11 UDP -X 192.168.0.11 UDP -x recv_log1

4. after sent out all the packets, we stop listeners at h6 and h1

5. Log file read for analysis

   ./ITGDec recv_log1

Experiment Result Evaluation

When it comes to the topic of network traffic, there are some QoS parameters to evaluate the performance of the network. Below are some parameters,

- **Latency** - the time it takes to transfer the packet from source to destination
- **Jitter** - the variation of latency
- **Loss** - the packet which failed to reach its destination

(Packet loss ratio = no. of packet loss/no. of packets sent)

- **Throughput - ability to carry data at a unit of time**

Among above QoS parameters, Latency and Jitter can be used to evaluate controllers since other ratios are related to the bandwidth.

After executing the experiment setup, below are the statistics which were gained during packet transmission.

Single Controller (Series 1):

packet size (One Controller)	20000	40000	60000	80000	100000
Total Time(Sec)	10.008404	10.006689	10.00717	10.007095	10.009816
Total Packets	23421	21249	30571	29433	31038
Minimum Delay(Sec)	0.000048	0.000054	0.000038	0.000032	0.000039
Maximum Delay(Sec)	0.021566	0.023077	0.022351	0.02301	0.022219
Average Delay(Sec)	0.01341	0.013235	0.013476	0.013301	0.013502
Average Jitter(Sec)	0.000254	0.000249	0.000252	0.000231	0.000251
Delay Standard Deviation(Sec)	0.004637	0.004813	0.00416	0.004219	0.004161
Bytes Received	11991552	10879488	15652352	15069696	15891456
Avg Bit Rate(Kbit/s)	9585.18621	8697.77246	12512.90984	12047.20931	12700.6978
Avg Packet Rate(pkt/s)	2340.133352	2123.479604	3054.90963	2941.213209	3100.756298
Packets Dropped(%)	42027 (64.21 %)	45805 (68.31 %)	37295 (54.95 %)	38669 (56.78 %)	37146 (54.48 %)
Avg Loss-Burst Size(pkt)	45.190323	53.076477	28.2324	33.450692	28.773044

Table 1: Single controller experiment result

As we discussed in the above experiment set up, we sent five simultaneous UDP traffic flows from h1 to h6 where each flow has different packet size. Table 1 shows the experiment result using single controller. By looking at the result, we couldn't see any correlation between packet rate against parameter values. Thus, we can say, that single controller architecture has no influence on individual flow, but overall SDN performance has been affected by larger amount of traffic. In next experiments will be carried out using multiple controllers against same switch configuration.

Multiple Controller (Series 2):

packet size (Two Controllers)	20000	40000	60000	80000	100000
Total Time(Sec)	10.010604	10.005742	10.006718	10.007027	10.011117
Total Packets	61110	55084	62954	63178	55811
Minimum Delay(Sec)	0.000032	0.000048	0.000032	0.000032	0.000046
Maximum Delay(Sec)	0.023242	0.022882	0.022307	0.022307	0.02163
Average Delay(Sec)	0.009875	0.009634	0.009854	0.009856	0.009578
Average Jitter(Sec)	0.000232	0.000227	0.000224	0.000223	0.000223
Delay Standard Deviation(Sec)	0.00586	0.005956	0.00586	0.005852	0.005923
Bytes Received	31288320	28203008	32232448	32347136	28575232
Avg Bit Rate(Kbit/s)	25004.14161	22549.4585	25768.64702	25859.5373	22834.80015
Avg Packet Rate(pkt/s)	6104.52676	5505.238892	6291.17359	6313.363599	5574.902381
Packets Dropped(%)	4160 (6.37 %)	11760 (17.59 %)	4788 (7.07 %)	4678 (6.89 %)	12075 (17.79 %)
Avg Loss-Burst Size(pkt)	3.88422	13.626883	4.483146	4.417375	12.777778

Table 2: Multiple controller experiment with two controllers

In this experiment, we placed two controllers in between three switches which connected using linear topology. By observing the results, we can see drastic improvement in overall SDN performance under high traffic.

Multiple Controller (Series 3):

packet size (Three Controllers)	20000	40000	60000	80000	100000
Total Time(Sec)	10.012921	10.005668	10.005086	10.007331	10.011471
Total Packets	43683	41423	42423	33686	46118
Minimum Delay(Sec)	0.000038	0.000051	0.000048	0.000057	0.000037
Maximum Delay(Sec)	0.033209	0.033555	0.033253	0.03387	0.032863
Average Delay(Sec)	0.013549	0.013477	0.013537	0.012769	0.013647
Average Jitter(Sec)	0.000276	0.000276	0.000275	0.000269	0.000267
Delay Standard Deviation(Sec)	0.006219	0.00653	0.006465	0.006687	0.006194
Bytes Received	22365696	21208576	21720576	17247232	23612416
Avg Bit Rate(Kbit/s)	17869.46766	16957.24943	17367.62763	13787.67785	18868.28899
Avg Packet Rate(pkt/s)	4362.663003	4139.953474	4240.143463	3366.132288	4606.515866
Packets Dropped(%)	14996 (25.56 %)	18831 (31.25 %)	18210 (30.03 %)	27176 (44.65 %)	14944 (24.47 %)
Avg Loss-Burst Size(pkt)	11.002201	14.463134	14.193297	27.450505	10.5686

Table 3: Multiple controller experiment with three controllers

Here we placed each switch with individual controller to observe the statistics. The results depict that, it has not improved as experiment series 2. But still there is a considerable amount of improvement in the SDN performance.

Below charts have been used to illustrate each parameter against packet rate on three controller architectures (Series 1, Series 2 and Series 3).

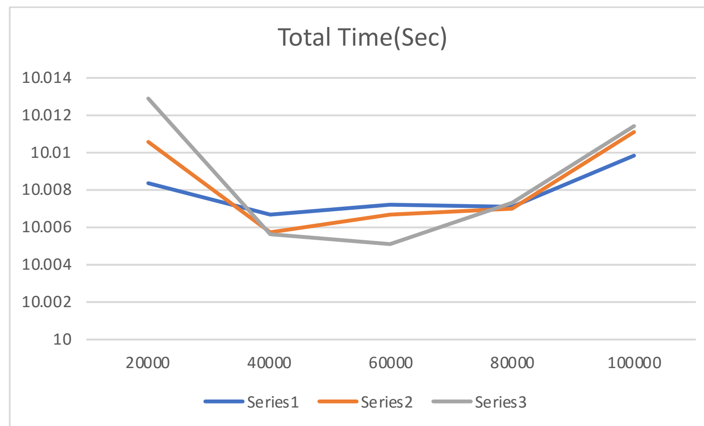

Figure 14: Total Time

According to the above chart, it shows total time taken for the whole experiment. Since the difference is in milliseconds, we cannot see any significant difference among series.

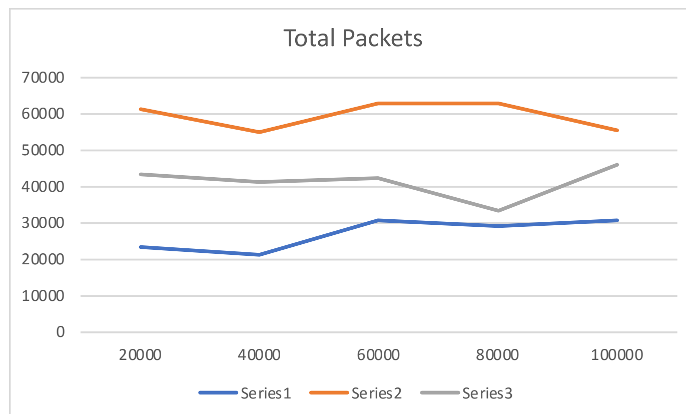

Figure 15: Total Packets

Figure 15 shows the total packets received to the destination h6. As we can see, two controller architecture has the best performance, three controller has intermediate performance and single controller has least performance.

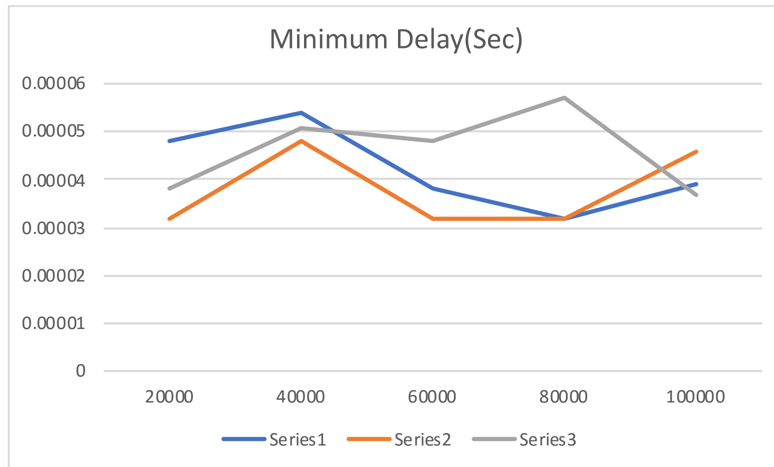

Figure 16: Minimum Delay

As we can see, there is no such big difference in minimum delay except in 60,000 and 80,000 packet rates.

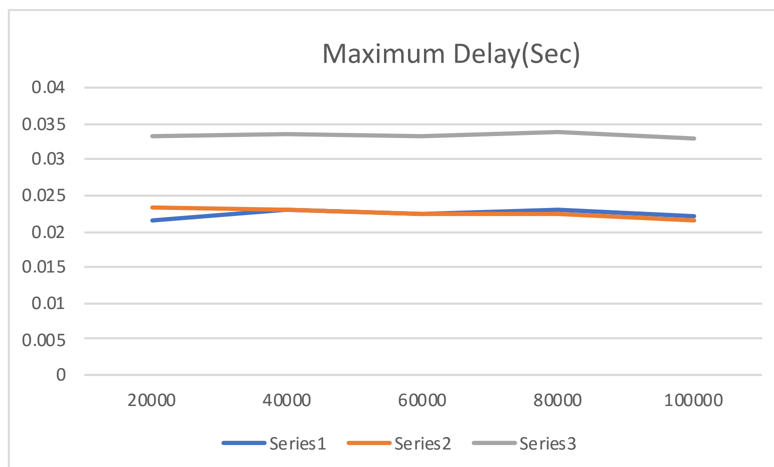

Figure 17: Maximum Delay

According to the figure 17 results, we can see that even though we have placed each controller on each switch, maximum delay has not minimized.

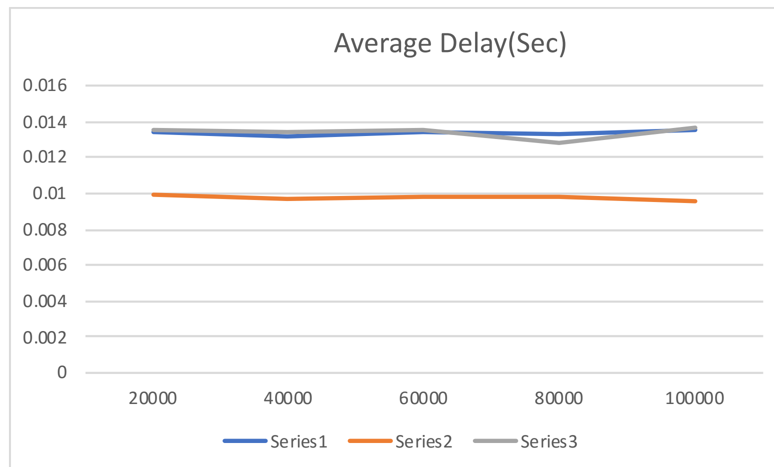

Figure 18: Average Delay

According to the figure 18, the set up with two controllers has given the least average delay throughout the experiment.

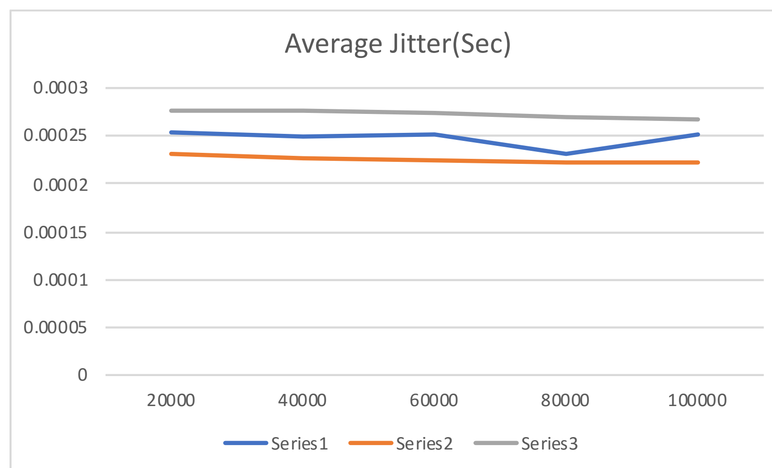

Figure 19: Average Jitter

It's better to gain low average Jitter in any network since low average Jitter implies low latency. According to the figure 19, we can see that series two set up has the best performance.

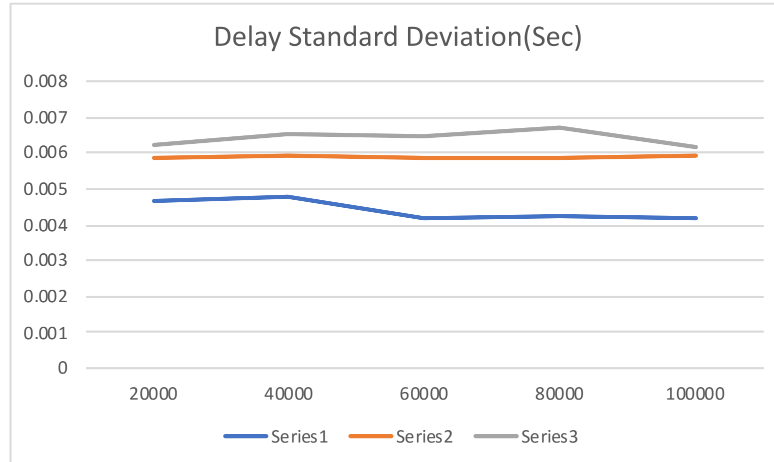

Figure 20: Delay Standard Deviation

This chart represents the variance of the delay. Series one set up has minimum variance of delay in the packet transmission.

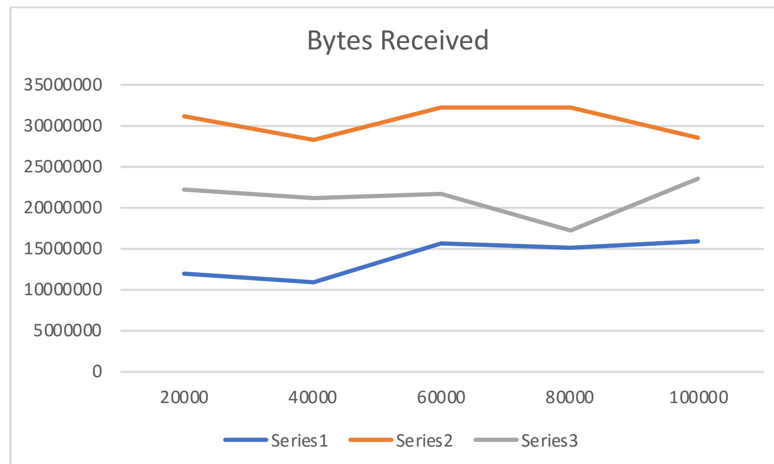

Figure 21: Bytes received

According to the figure 21, we can see that the best performance in bytes received has happened in series two set up.

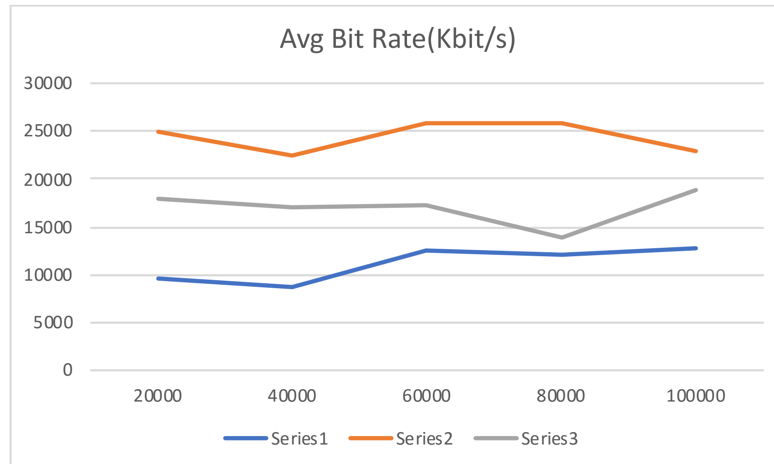

Figure 22: Average Bit rate

According to the figure 22, series two has performed best in bit rate and series 3 has performed intermediate and series 1 has lowest performance.

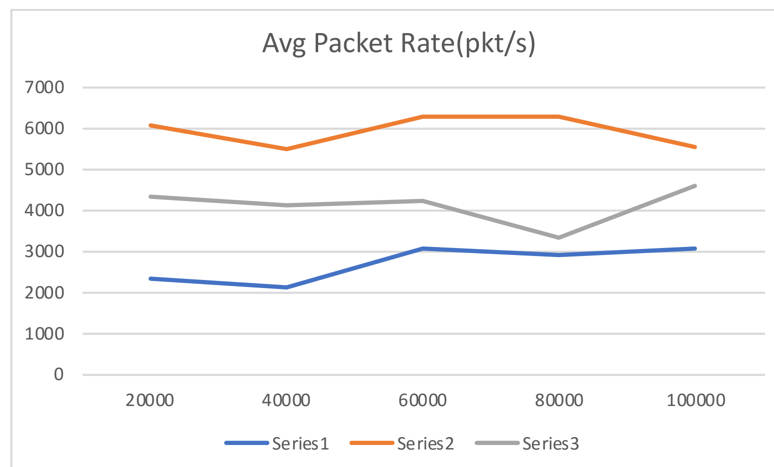

Figure 23: Average Packet Rate

According to the figure 23, series two has performed best in average packet rate, series 3 has performed intermediate and series 1 has lowest performance.

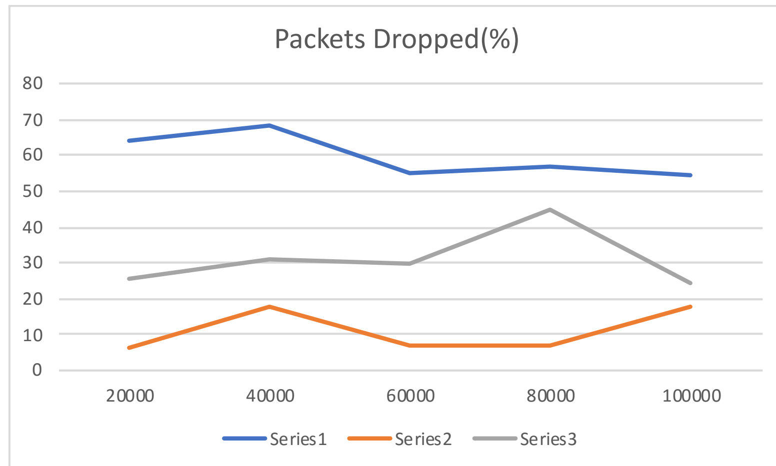

Figure 24: Packets Dropped

According to the figure 24, series two has given significant performance in packet drop, series 3 has performed intermediate and series 1 has lowest performance.

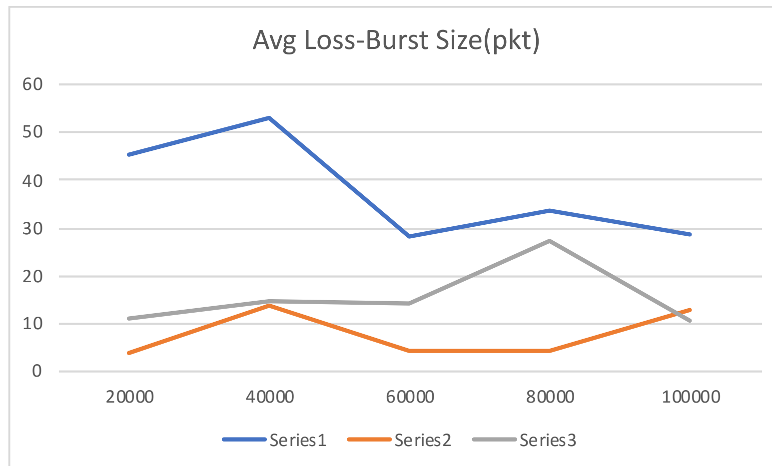

Figure 25: Average Loss-Burst Size

According to the figure 25, series two has performed best in lost burst size and series 3 has performed intermediate and series 1 has the lowest performance.

Below are average statistics which extracted according to the total results of different controller architectures.

Controller count	1	2	3
Number of flows	5	5	5
Total Time(Sec)	10.010869	10.011952	10.014212
Total Packets	135712	298137	207333
Minimum Delay(Sec)	0.000032	0.000032	0.000037
Maximum Delay(Sec)	0.023077	0.023242	0.03387
Average Delay(Sec)	0.013395	0.009766	0.013427
Average Jitter(Sec)	0.000254	0.000238	0.000285
Delay Standard Deviation(Sec)	0.004366	0.005889	0.006412
Bytes Received	69484544	152646144	106154496
Avg Bit Rate(Kbit/s)	55527.2826	121971.1353	84803.07467
Avg Packet Rate(pkt/s)	13556.46548	29778.1092	20703.87565
Packets Dropped(%)	200942 (59.69 %)	37461 (11.16 %)	94157 (31.23 %)
Avg Loss-Burst Size(pkt)	36.134149	7.48322	14.823205

Table 4: Average performance matrix

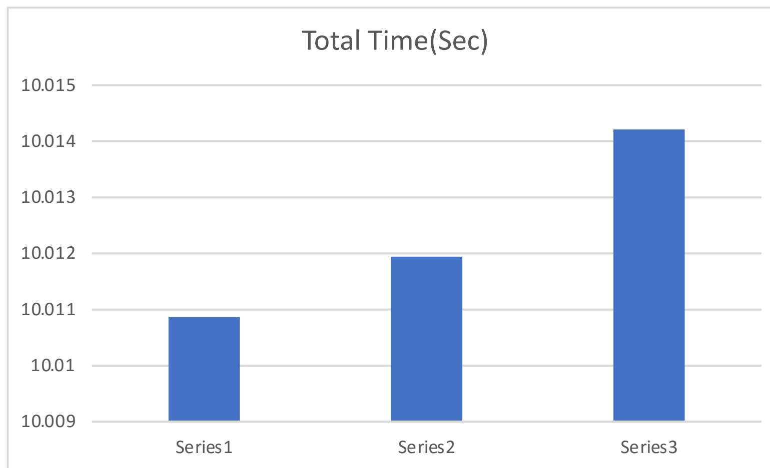

Figure 26: Average performance in Total Time

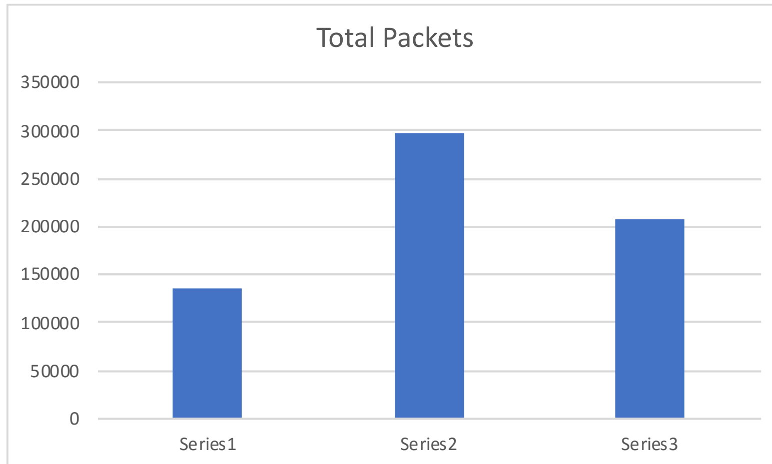

Figure 27: Average performance in total packets

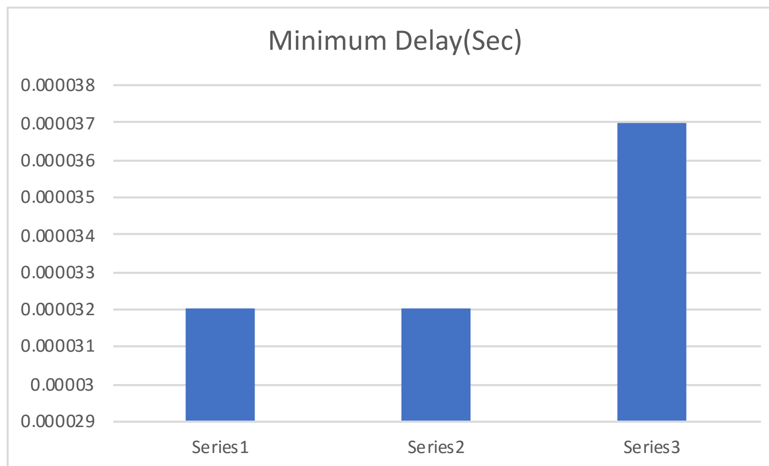

Figure 28: Average performance in minimum delay

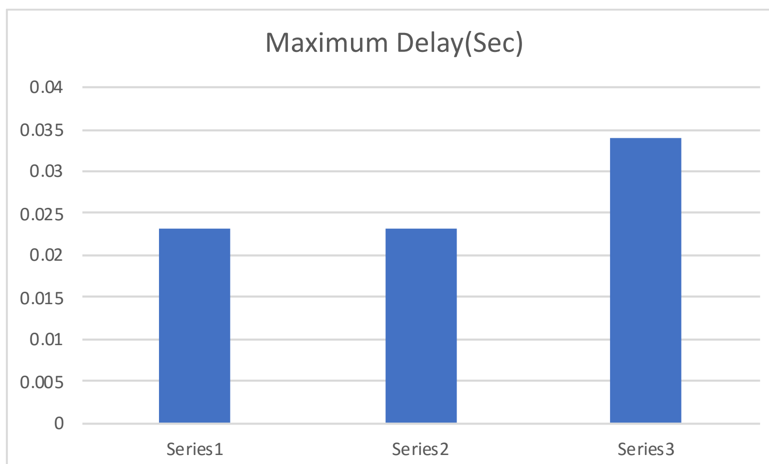

Figure 29: Average performance in maximum delay

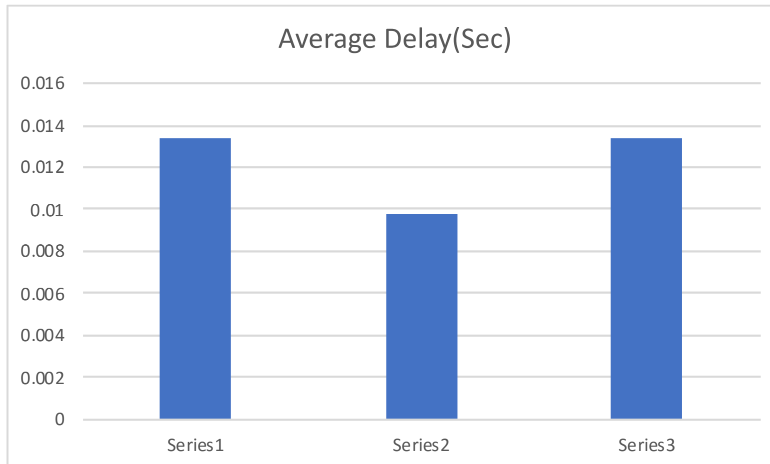

Figure 30: Average performance in average delay

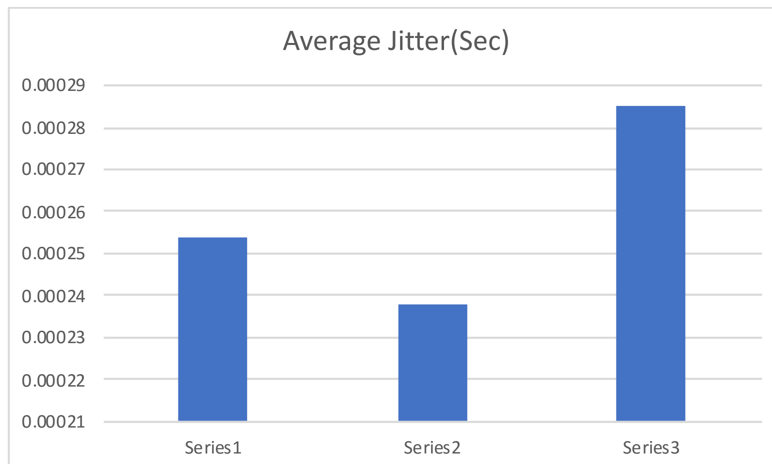

Figure 31: Average performance in average Jitter

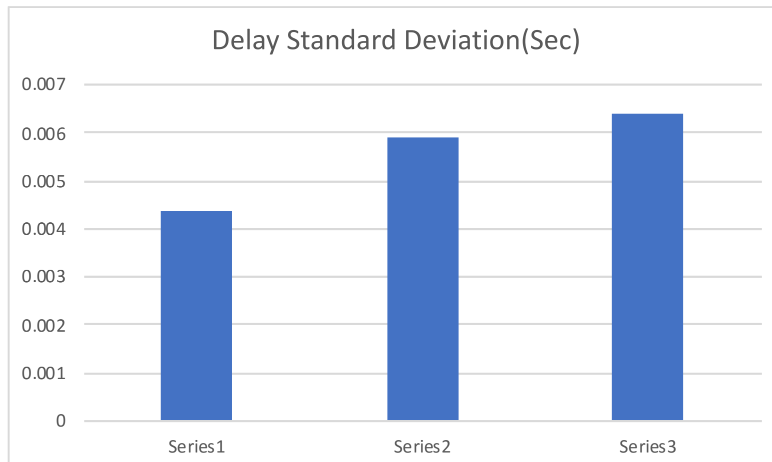

Figure 32: Average performance in delay standard deviation

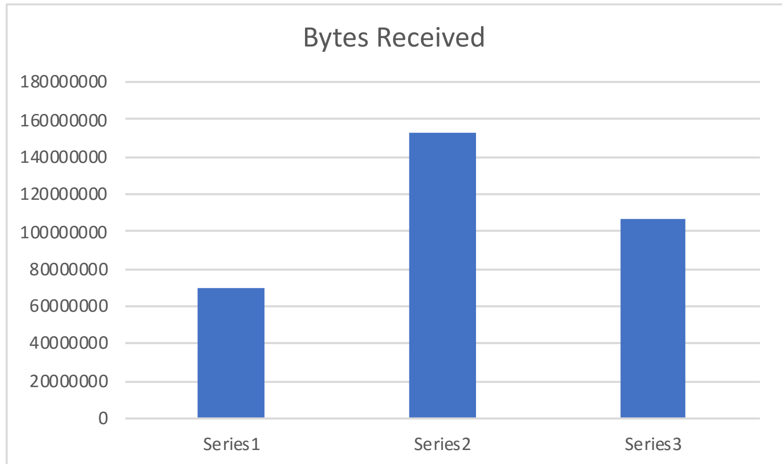

Figure 33: Average performance in bytes received

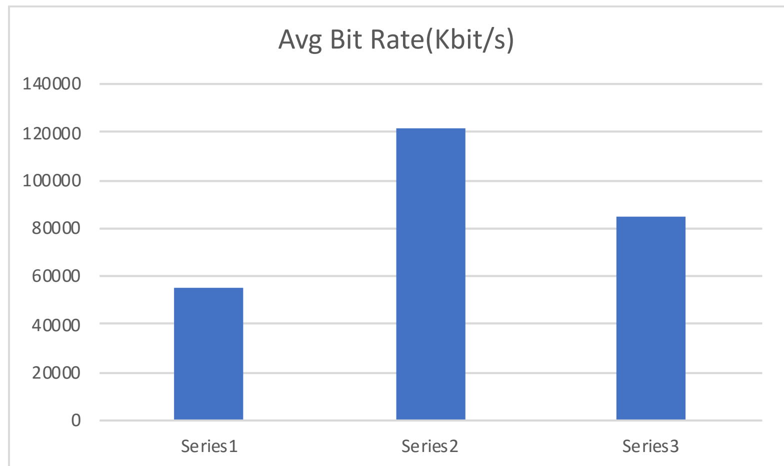

Figure 34: Average in average bit rate

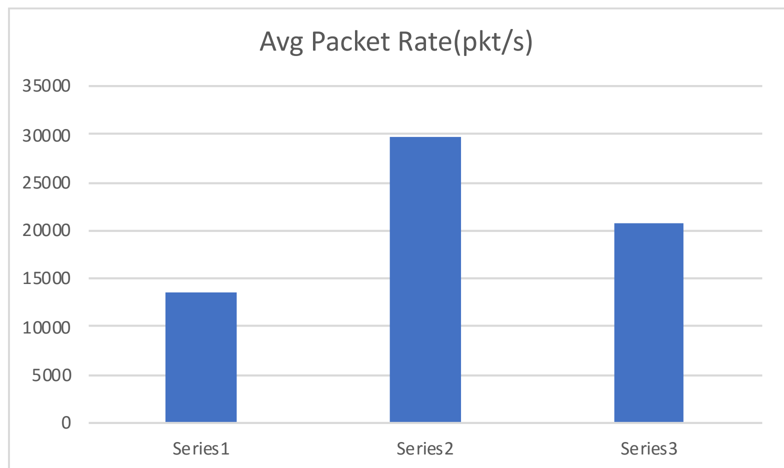

Figure 35: Average in average packet rate

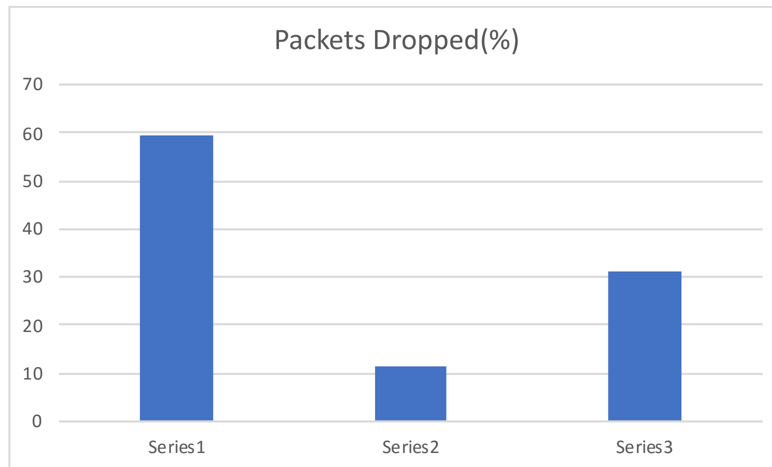

Figure 36: Average in packet dropped

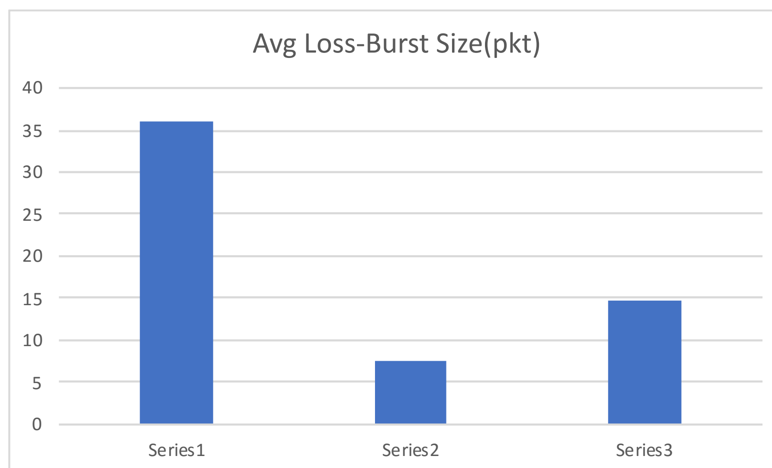

Figure 37: Average in average loss-burst size

Chapter 4

Conclusion

The SDN domain is becoming sophisticated day by day. It is very important to have a reliable network in any organization to have their business as usual. In this study, we discussed about how horizontally connected switches behaving with single controller and multiple controller under hefty traffic. The result shows the multiple controller architecture has improved performance in most important arias. According to the statistics, we saw that the packet loss rate has relationship with the latency because when the latency increases, the packet loss rate also get increased.

The packet loss rate is a major concern when it comes to the network reliability. Lesser the packet loss rate, higher the network reliability. We observed that the packet drop in a single controller architecture was 59%, where it has reduced the packet drop ratio into 11% in two controller architecture. The three controller architecture has an intermediate performance compared to other two. It has proven that, the number of controllers in a SDN is not the only fact which increase the performance. The most important thing is how they have been integrated. Having one controller for each switch has less performance than having one controller in between switches. This happens due to the high traffic in a switch at the time the single controller also busy. Here in our experimented two controller architecture, one switch has been updated by two controllers, which shows the best performances by sharing flow table management among two controllers.

By observing the network simulation and the evaluation results, we can find evidences which support to our hypothesis mentioned in this thesis. Therefore, we can conclude that the controller overhead can be mitigated using multiple controllers.

Future work

Though the multiple controllers were able to minimize the controller bottleneck, it is not the only concern of software defined work. But to a certain extent, we can predict that, multiple controllers can perform better than the logically centralized single controller. Also, if we can scale up the multiple controller architecture vertically, it would be more efficient and more scalable. We can deeply study how to scale up the multiple controller architecture vertically. And, despite of this research, which controller mechanism is suitable in which situation is still an open research topic because each controller has built on its own architectural perspectives. As a future work, we can run the test beds to determine the number of switches which can be handled at a same time using a single controller with respect to different network load. As well as we can research further on event management in software defined network using multiple controllers which assumed to be good in performance.

References

[1] E. Borcoci, "Control Plane Scalability in Software Defined Networking Control Plane Scalability in Software Defined Networking Acknowledgement," 2014.

[2] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, p. 351, 2012.

[3] Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro: A System for Scalable OpenFlow Control."

[4] T. Hu, Z. Guo, P. Yi, T. Baker, and J. Lan, "Multi-controller Based Software-Defined Networking: A Survey," *IEEE Access*, vol. 6, no. March, pp. 15980–15996, 2018.

[5] S. Avallone, S. Guadagno, D. Emma, A. Pescapè, and G. Ventre, "D-ITG distributed internet traffic generator," *Proc. - First Int. Conf. Quant. Eval. Syst. QEST 2004*, no. June 2014, pp. 316–317, 2004.

[6] S. Badotra and J. Singh, "OpenDaylight as a controller for Software Defined Networking," *Int. J. Adv. Res. Comput. Sci.*, vol. 8, no. 5, p. 7, 2017.

[7] C. Decusatis, A. Carranza, and J. Delgado-caceres, "Modeling Software Defined Networks using Mininet," no. 133, pp. 1–6, 2016.

[8] S. Defined, A. Fluent, and P. Networks, "Virtual Infrastructure for SDN in Enterprise Networks."

[9] D. Erickson, "The Beacon OpenFlow Controller."

[10] B. Freisleben and T. Kielmann, "Coordination Patterns for Parallel Computing."

[11] V. Implemented and W. Protocol, "OpenFlow Switch Specification List of Figures," pp. 1– 56, 2011.

[12] P. Ivashchenko, A. Shalimov, and R. Smeliansky, "High performance in-kernel SDN /OpenFlow controller Introduction / Motivation," pp. 3–5.

[13] J. S. Ivey, M. K. Riley, and G. F. Riley, "A Software-Defined Spanning Tree Application for ns-3," pp. 3–4.

[14] F. Keti, "Emulation of Software Defined Networks Using Mininet in Different Simulation Environments," pp. 205–210, 2015.

[15] W. Kim and S. Chung, "Proxy SDN Controller for Wireless Networks," vol. 2016, 2016.

[16] D. Kreutz, F. M. V Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, S. Member, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," pp. 1–61.

[17] J. Mccauley, A. Panda, M. Casado, T. Koponen, S. Shenker, and U. C. Berkeley, “Extending SDN to Large-Scale Networks,” pp. 1–2.

[18] N. Mckeown, T. Anderson, L. Peterson, J. Rexford, S. Shenker, and S. Louis, “OpenFlow: Enabling Innovation in Campus Networks,” 2008.

[19] C. Page, Separating Computation and Coordination in the Design of Parallel and Distributed Programs. 1998.

[20] G. A. Papadopoulos, “COORDINATION MODELS AND LANGUAGES Department of Software Engineering,” pp. 1–50.

[21] G. Salvaneschi, J. Drechsler, and M. Mezini, “Towards Distributed Reactive Programming.”

[22] A. Shalimov, D. Zimarina, and V. Pashkov, “Advanced Study of SDN / OpenFlow controllers.”

[23] J. Sommers and P. Barford, “Fast, Accurate Simulation for SDN Prototyping,” 2013.

[24] A. Tootoonchian, M. Casado, and R. Sherwood, “On Controller Performance in Software- Defined Networks.”

[25] V. Universiteit, “Coordination models and languages for parallel programming,” pp. 1–14.

[26] W. Xia, Y. Wen, S. Member, C. H. Foh, S. Member, D. Niyato, and H. Xie, “A Survey on Software-Defined Networking,” vol. 17, no. 1, pp. 27–51, 2015.

[27] E. Borcoci, “Network Function Virtualization and Software Defined Networking Cooperation Network Function Virtualization and,” 2015.

[28] S. H. Yeganeh and Y. Ganjali, “Kandoo: a framework for efficient and scalable offloading of control applications,” *HotSDN*, pp. 19–24, 2012.

[29] A. Tootoonchian, “Hyperflow.Pdf,” *Proc. 12th ACM Work. Hot Top. Netw.*, vol. 3, pp. 1–6, 2010.

[30] NAGA PRAVEEN KUMAR KATTA, “Building Efficient and Reliable Software-Defined Networks,” vol. 1, no. 1, pp. 17–29, 2016.