Evaluation of Multiple Controller based Software Defined Networks Architecture over Single Controller Software Defined Architecture

A Thesis

presented to

the Faculty of the Department of Engineering Technology

University of Houston

In partial fulfillment

of the Requirements for the Degree

Master of Science

in Engineering Technology

By

Sonal Harsh

December 2016

Acknowledgements

I would like to pay my debt of gratitude towards my advisor Dr. Ricardo Lent, for his valuable advice and support towards the completion of my research work. I thank him for his patience, support, motivation and sharing his immense knowledge in helping me advance my related research and studies.

Also, I offer appreciation to my thesis committee which includes finest professors from Department of Network Communication Dr. Driss Benhaddou and Dr. Xiaojing Yuan, for reviewing my work and providing their insightful comments and feedback.

I would like to extend my appreciation and gratitude towards my classmates Govind Prasad and Nidhi Shah for constantly guiding me and supporting me.

Finally, I would offer my sincere gratitude towards my parents who constantly encouraged and patiently supported me during all times.

Abstract

Software Defined Networks are currently the most promising researched area in the field of computer networking. Many research implementation is undergoing in making advances in the field of Software Defined networks. The rise of demand for moving from traditional networks to SDNs has given rise to many challenges. Software-defined Networks are continuously advancing, which demands the need to solve the issues of scalability, transmission delays, and packet loss. The large the network, more the delay as with the increase in the network causes network congestion and transmission delays. In larger networks, single controller architecture will be inefficient to manage the network. To tackle this issue, Software Defined Architecture with multiple controllers has been introduced in the recent researches.

Multiple controller SDN architectures will be efficient to manage larger networks, decrease transmission loss and avoid fault tolerance. As the network expands the load on the single controller increases, so it becomes difficult for single controller architecture to efficiently manage the network. Multiple controllers do not help to balance the load between them, it will tackle the network congestion issue by distributing the load between them.

In this paper, we evaluate the performance of Multiple controller SDN architectures in comparison to single controller SDN architecture. The Multiple controller architectures have better load management, less transmission delay, and less packet loss ratio. The multicontroller SDN architecture is a more efficient solution than single controller architecture when it comes to wider networks. In the experiment, we have used POX SDN controller to implement the Software Defined

Architecture, mininet[2] as a network emulator to test complex topologies and D-ITG to test the result of intense network traffic on topologies in different SDN architectures.

Contents

Introduction

1.1 Background and Motivation

In SDN[1] control plane and data plane are separated in order to provide open interfaces. Software defined Architecture can be defined into three layers namely Application layer, Control Plane layer, and data plane layer. The SDN architecture enables the network control directly programmable and it's centrally managed through SDN Controllers which maintains a global view of the entire network.

There are two different interfaces in the SDN architecture namely northbound interfaces and southbound interfaces. Northbound helps to communicate within controller with the APIs on how the network service or device should be. It enables different applications on the SDN architecture.

The other SDN interface i.e. southbound which provides the communication between the network plane and control plane. It gives an opportunity to automate the networking tasks by providing some enormous possibilities for network programming. SDN will eventually advance in the creativity and flexibility in the field of networking.

The most important and critical requirements are not fulfilled in the early proposed single controller SDN architectures which are efficiency which is not enough in just one centralized a controller as it will not be able to manage the load and data in an effective manner, high availability will be an issue in single controller architecture as bigger the network less available the controller which will lead to redundancy and security issues. The scalability is also one of the reasons to move to multiple controllers architecture.

The whole network performance will be jeopardized with the single controller architecture. If the controller fails the control plane will lose the connectivity with the whole network, thereby making the whole fail. Clearly, multiple controllers will help to overcome these issues. Multiple controllers will provide better availability, better performance, and effective load balancing.

Many researchers have worked in the field of SDN to enhance the traditional single controller SDN architecture to push to multiple controller SDN architectures to solve these issues. In this thesis, we are evaluating the performance of the two different architecture to support the hypothesis that multiple controllers architecture handles the network better than single controller architecture. We will be focusing on providing the evaluation of the results of the analysis between Single controller SDN architecture and Multiple controller architectures.

1.2 Objective

The three most critical requirements- efficiency, scalability, and availability, are unachievable in an SDN-enabled centralized network, which was the main objective of the early proposed SDN architectures, using just one controller. These issues can be resolved with multicontroller architecture. The ulticontroller architecture can be efficiently used to avoid failure issue which is very significant with a single controller. Clearly, multiple controllers can team up to understand the network issues and even if one controller fails, other controller or controllers will be there to take over the network traffic.

The purpose is to experimentally prove the reasons to move to the multiple controller architectures as the network widens. With our evaluation, we want to highlight the benefits of multicontroller architecture over single controller architecture. The POX SDN controller is used to evaluate the performance. The topology has been set up in mininet using miniedit. To understand the network traffic performance D-ITG is used to fill the capacity of the network. We have analyzed the various merits like load jitter, average delay, bitrates to understand the difference in performance of the single controller SDN architecture and multiple controller SDN architecture.

1.3 Contributions

The thesis presents the review of SDN architectures and their drawbacks. The deep understanding of contributions in the field of Software Defined Networks is done to support the proposed hypothesis. Then the reasons push us to think that multicontroller topology is better than single controller topology explained and justifies using experimental setup. So, we have done the complete evaluation of both the architectures that will present detailed different aspects that will emphasize the benefits of multicontroller architecture over Single controllers.

1.4 Thesis Organization

Chapter 2 will present the background of the work done in the related field that leads the way for the reason to work on the evaluation of single controller and multiple controller architectures.

Chapter 3 provides the architecture and different topology, explains the controllers used in order formulate SDN architecture, the methodology used to implement the experimental setup.

Chapter 4 explains the experimental setup and testbed to perform the network analysis on the system and there presenting the evaluation of the experiment in the further chapters

Chapter 5 and Chapter 6 will provide results and analysis of comparison of two architectures to showcase the needs of moving to multicontroller scenario.

Chapter 7 concludes the thesis and discusses possible future work in this area.

Chapter 2

2.1 Background

2.1.1 SDN Architecture

The traditional single SDN architecture has 6 major components explained in the below section

Management plane is the first is the management plane, which is a set of network applications that manage the control logic of a software-defined network. SDN-enabled networks use programmability instead of command line interface to give flexibility and easiness for implementing new applications and services, which includes routing, load balancing, policy enforcement, or a custom application from a service provider. Orchestration and automation of the network are also possible via existing API[4].

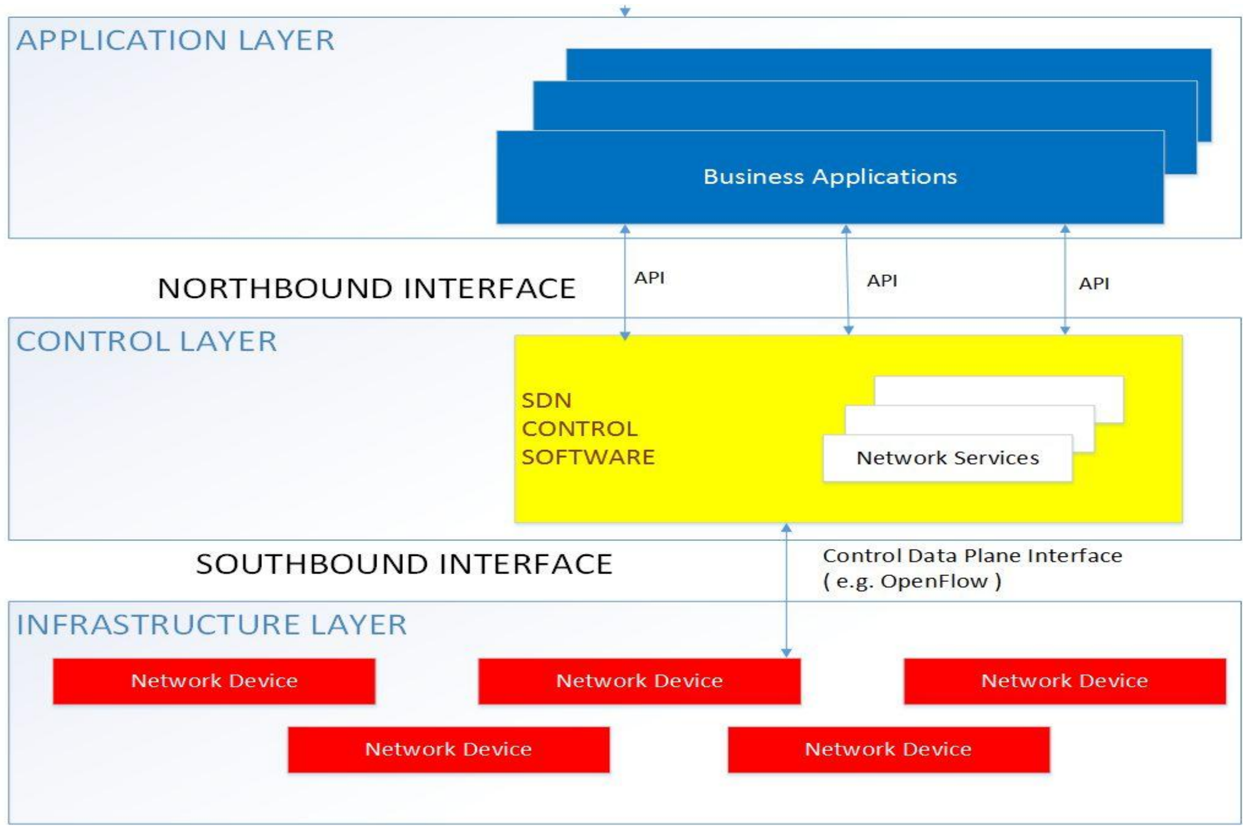
The control plane is the second and the most intelligent and important component layer of a basic SDN architecture. This layer contains a controller, that manages the whole packet transfer by forwarding various rules and policies to the infrastructure layer through the one of the interface Southbound interface [4].

The third layer is an infrastructure layer which is also called as data plane , represents the forwarding devices on the network (routers, switches, load balancers, etc.). It uses the southbound APIs to interact with the control plane by receiving the forwarding rules and policies to apply them to the corresponding devices [3].

Fourth, the northbound interfaces are mostly a set of open source application programming interfaces(APIs) that allows communication between the control layer and the management layer [3].

The east-west interfaces are the fifth component of the SDN architecture but are not yet standardized. The east-west interface is one of the components will be important for multiple controllers of different SDN architectures to communicate between them. They use notification system or messaging or distributed routing protocols like BGP and OSPF.

The southbound interfaces are the sixth and the last component of the SDN architecture which allow interaction in between data plane and control plane which is a most important element for the functioning of SDN networks. The southbound interfaces can be defined as protocols i.e OpenFlow protocols, that permit the controller to push policies to the forwarding plane. The OpenFlow protocols are the southbound API for SDN-enabled networks are most widely accepted and used.



2.1.1 OpenFlow

Open Networking Foundation (ONF) [5], which is backed by the IT industry leaders like Facebook, Cisco, Google, HP, and others have normalized OpenFlow. This is the reason it's important to understand the OpenFlow to grasp the notion of SDN. OpenFlow is nothing but just an instantiation of SDN, as there are lot many present and some under development southbound APIs, available for SDN. OpFlex [6], is one of the which distributes some of the complexity of managing the network to the infrastructure layer to improve the scalability. On the other hand, ForCES [7] proposes a flexible method to ameliorate the management of traditional networks without using a logically centralized controller, while ROFL [8], which relies on OpenFlow, provides an API for software developers to enable full development of new applications [1].

2.2 Literature Review

In SDN, various research has been done and in progress to implement the multiple controller architecture. In multicontroller architecture, more than one controllers are used to perform the task of networking to achieve some level of scalability. Another idea was proposed [10] before implementing multiple controllers, which is installing replicated controllers to remove the single point of failure. Nevertheless, this method has many disadvantages such as using passive controllers that will be active, just in case the main controller fails. In a logically centralized architecture, all the controllers have the same responsibilities, and they split the charge equally. They are always aware of every change in the network, and they share the same information instantly, thanks to the network synchronization.

Further I would describe all different works being done in the field of multicontroller Software Defined enhancement. Physically distributed architecture corresponds to the multicontroller platform to enhance the basic SDN architecture and it varies on the placement of the controllers and how they are communicating between them.

The approach of implementing replicated controller [3] before multicontroller approach was introduced to overcome single point of failure issue. This approach has several disadvantages like redundancy of controllers as the passive controllers are being used as active to make sure the controller can be swapped. All the controllers have same responsibilities. They are always aware of every change in the network, and they share the same information instantly, thanks to the network synchronization. A physically distributed architecture can be either logically centralized or logically distributed. Logically centralized means that we take advantage of the concept of a multicontroller design, but at the same time, we always consider that we have a single controller. In other words, we take the charge, and we distribute it among the multiple controllers; however, for the underlying layer, it is like there is just one controller that commands the whole network.

When we talk about distributed controller, ONIX [13] is one the controllers is most talked about because of its distributed controller plane with cluster of physical servers. ONIX has 4 different components for network control, namely; physical infrastructure like routers, switches, etc., the connectivity infrastructure, the control logic which is dependent on top of ONIS's API, and lastly ONIX which provides the control logic programmatic access to the entire network.

ONIX provides three methods to improve the scalability of its network. First is by partitioning the network logically, in other words, by distributing the workload on multiple ONIX instances. Second, ONIX can allow multiple nodes to show up as a single node in the upper layer, which is called aggregation. Third, ONIX allows data state applications that can be used to improve the consistency and the durability of the network.

In a logically distributed architecture, the controllers are physically and logically distributed. Additionally, every controller has just a view of the domain it is responsible for, and it can take decisions for it, unlike a logically centralized design, where each controller decides based on the global network view. In a word, a logically centralized architecture stays near to the initial tendency of SDN, which is using a single controller, or a multicore controller to improve the performance. On the other hand, a logically distributed architecture goes away from the first tendency of SDN, by making several controllers have several responsibilities inside the network. Controllers like Beacon [10] and NOX [11] have used multithreading techniques to split a single controller logically to increase its performance. In this case, it is so obvious, since we have a single controller, that we are not talking about a multicontroller architecture.

Some researchers have proposed BalanceFlow[11], a controller load balancing architecture for wide-area OpenFlow networks. BalanceFlow approach undertakes the use of CONTROLLER X for switches and cross-controller communication and selects a controller to be a super controller to smoothly distribute the flow requests thereby avoiding transmission delay. BalanceFlow can adjust a load of each controller dynamically. The problem with BalanceFlow is the centralized control problem due to the extreme need of a super controller. Also, when there the stress of load balancing increases on the super controller, it reduces the performance of

In the lieu of BalanceFlow, researchers proposed HybridFlow approach to overcome the scalability issue on the super controller [12]. HybridFlow integrates the method of distributing the load and centralized control by managing a cluster of controllers using a super controller. Each cluster contains multiple controller and switches which oversee the controller in the cluster. Each cluster shares the information with the super controller. Super controller based on the load on the cluster divide the flow between the internal cluster and external cluster per the threshold. Hybrid Flow method is best amongst the approach but it doesn't provide the solution of the super controller fails. Once the super controller is stops responding it will break the centralized control. The distributed load balancing is done on Multipath routing algorithm. In this method, every controller uses an algorithm to make a routing decision. The algorithm uses the estimated bandwidth of flow usage and the amount of free bandwidth. It tests the bandwidth of the system on various parameters. Before load balancing, it does a lot of calculation to make the load balancing work. The effect of the bandwidth. HyperFlow uses a "publish/subscribe" messaging paradigm to propagate information in the control plane. It makes sure to keep the ordering of events published by the same controller. Also, to have less overhead it minimizes the traffic required for intercontrollers. This "publish/subscribe" system runs on the top of WheelFS [20], which is a distributed file system and can deliver flexible wide area storage for distributed application.

ONOS [21]

ONOS is one of the distributed SDN controller which two different prototypes. The first prototype has global view of the network, the fault tolerance, and the scalability. The network view has three components namely Titan [22] which is graph database, Cassandra [23] a key value store and a graph API to expose network state i.e. Blueprint [24]. ONOS can reassign the work to other instances and can add supplementary instances in case of failure or to distribute for better scalability. The second prototype was proposed to improve the previous one. The global network is intact but the effort has been made to solve excessive data storage by making remote operations fast, another approach by focusing on reducing the number of approaches. RAMCloud [25], has been introduced following the previous approach to implement Titan/Cassandra system with Blueprints graph on top of it. To remove the intercontrollers notification problems they adopted based on Hazelcast [26], in which communications will go through the channels installed at the top of all instances of the control plane.

Inter-SDN controller communication

Communication intercontrollers are the method used to allow exchanging information among the multiple controllers of a software-defined networks. Intercontroller communication can be implemented in different ways. One is horizontal approach which is SDN east-west interface. SDN east-west interface is used to exchange information between SDN domains that are under the control of single or multiple network operators. A session needs to be established between the two controllers by using either BGP or Session Initiation Protocol (SIP) over Transmission Control Protocol (TCP) to exchange information [9]. BGP can help inter-SDN communication for east-west type interface.

BGP is feasible for the peer SDN data. BGP is one of the solutions for load balancing between multiple controllers but the only feasible east-west interface. BGP is feasible for the peer SDN data. It cannot be used when it must communicate with more than just neighboring controllers. To establish session SIP i.e. Session Initiation Protocol can be used. SIP is a request- response protocol for initiating and managing communication. SIP is most likely to handle multimedia messages; thus, text-based messages are difficult to retrieve when integrated with SDN controller. Even though BGP is better than SIP approach, it's not achievable with other than east-west inter SDN interface.
SDNi[29] is one of the project under implementation by TCS (Tata consultancy Services) to implement the intercontroller communication. This aims at developing the communication between controllers which is deployable on Opendaylight-helium controller using BGP4 protocol.

DISCO [22]

DISCO is a distributed multi domain SDN controller which has two parts as an intradomain and an interdomain. The intradomain handles monitoring of the network and managing the flow prioritization. Intradomain tackle multiple network issue with the set of modules. The interdomain provides the communication among the multiple controllers and has two modules. First is the Messenger module to build channels between controllers to share information amongst each other. In this approach the AMPQ [28] protocol has been used which provides routing, prioritized querying, and messaging with orientation.

Chapter 3 System Architecture

We developed two different topologies using mininet and miniedit. The topology as we discussed has been created with POX controller as the remote controller to establish an SDN framework. After that we developed topology with more than one controller using python programming in mininet. We have created the topology using mininet and running 2 remote pox controllers one on 10.0.0.49:6633 and other on 10.0.0.50:6634.

To achieve the goal of implementing the distributed framework in SDN we created a communication channel in between the two POX controllers. The outbound communication has been implemented using the native messenger component of POX. The messenger component creates a system to inform one POX controller about the other and able to exchange the messages encoded in JSON. The messenger supports a straightforward TCP socket transport and an HTTP transport. Connections are established when there is a request to check and its temporary, which gets deleted when there is no communication.

3.1 SDN Architecture with Communication Channel

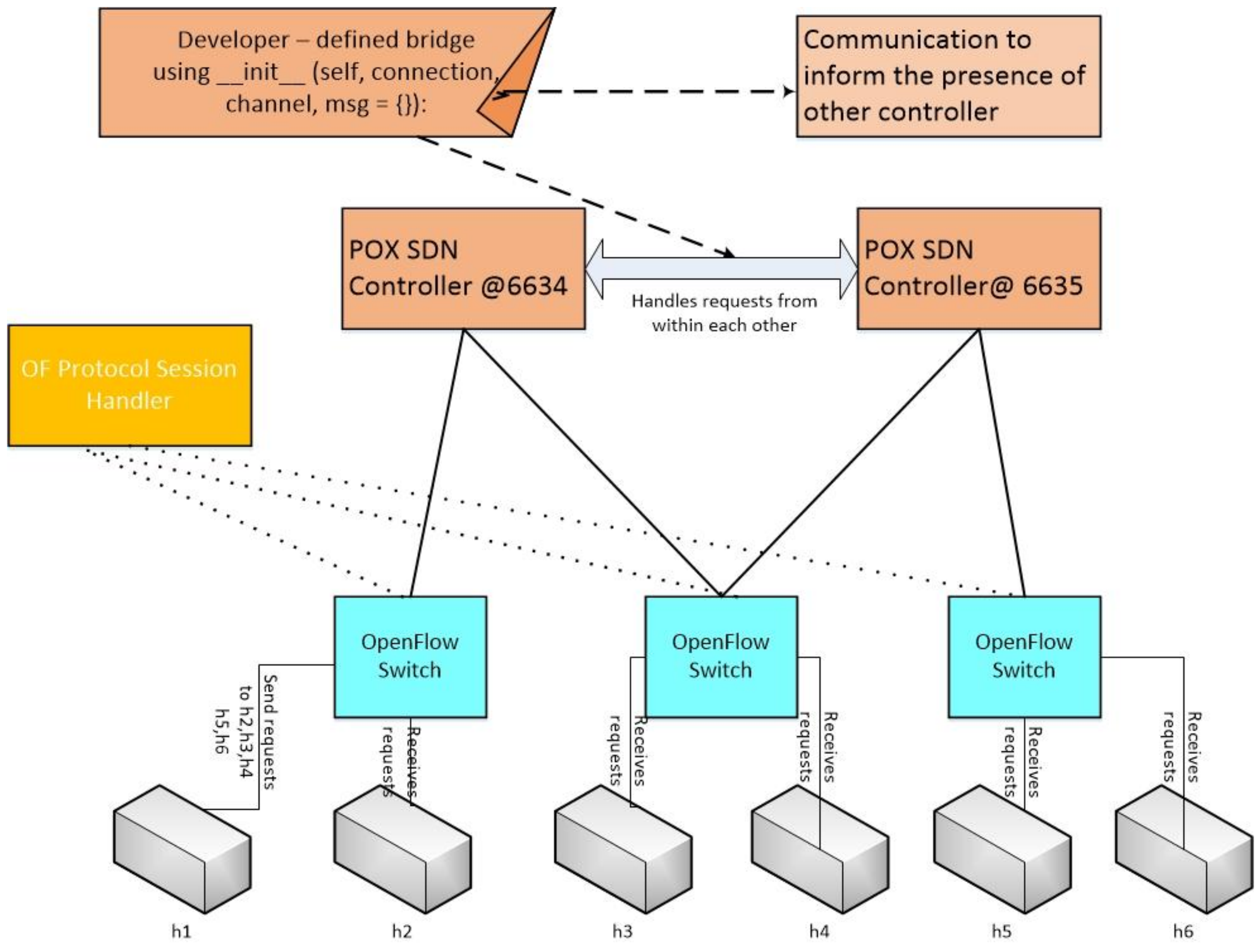

Developer – defined bridge using __init__ (self, connection, channel, msg = {}):
Communication to inform the presence of other controller
POX SDN Controller @6634
POX SDN Controller@ 6635
Handles requests from within each other
OF Protocol Session Handler
OpenFlow Switch
OpenFlow Switch
OpenFlow Switch
Send requests to h2,h3,h4 h5,h6
Receives requests
Receives requests
Receives requests
Receives requests
Receives requests
h1
h2
h3
h4
h5
h6

Figure 3.1 Distributed SDN System Architecture

In this architecture, all the requests from different hosts will be sent to the respective switches. Now all the different requests accumulated in the switches s1, s2, and s3 will be sent to the controllers. The advantage of this topology is that there are two controllers to handle all the load. The load is divided between the two controllers and have less load on a single controller. The controller is POX controller and the switches are OpenFlow switches in mininet. The hosts are in the kernel of the mininet which helps to simulate the test bed.

3.2 Mininet Topology using python script

```
Three directly connected switches plus two host for each switch:

   host --- switch --- switch --- host

Adding the 'topos' dict with a key/value pair to generate our newly defined
topology enables one to pass in '--topo=mytopo' from the command line.
"""

from mininet.topo import Topo

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )

        # Add links
        self.addLink( leftHost, leftSwitch )
        self.addLink( leftSwitch, rightSwitch )
        self.addLink( rightSwitch, rightHost )


topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Chapter 4

Testbed

4.1 Experimental Setup

We have set up experimental testbed on mininet network emulator. We have used POX [14] SDN controller to make the SDN topology. The mininet uses a python script to simulate the different topology. We have used two different topologies to evaluate the performance between single controller SDN architecture and multiple controller SDN architecture. The miniedit has been used to implement the GUI of the topology. In the topology, three openVswitches are used to setup the OpenFlow protocol. To check the network performance, 6 different clients have been set up to transfer the data and evaluate the performance of the network. POX controller instances have been used to simulate the SDN network. The emulator helps to generate network traffic using different tools. D-ITG, a network traffic generator, and the monitor is used to fill the network capacity.

The network performance is analyzed taking into consideration different parameters like transmission delay, Load jitter, packet loss rate. The performance between the different architectures has been evaluated using D-ITG traffic generator and monitor tool.

The testbed is simulated using VM [15] workstation player. The VM i.e. a virtual machine helps to download the Ubuntu operating system to implement the architecture on a windows machine. The virtual machine creates the instance of the operating system on the other operating system without affecting the configuration of the primary operating system. We have used 3 different servers to emulate the two controllers and one mininet topology.

4.2 Mininet

Mininet is used to emulate the network. Mininet is perhaps a network emulation orchestration system which runs router, SDN switches, different end-hosts, and links between all the devices on a Linux kernel system. It is used for emulating all the SDN system the hosts in mininet behaves nothing but like a real machine in which you can run different arbitrary programs. We can also SSH into the mininet using putty and bridge the network to the hosts. It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system, and user code.

Mininet is designed to easily create virtual software-defined networks consisting of an OpenFlow controller, a flat Ethernet network of multiple OpenFlow-enabled Ethernet switches, and multiple hosts connected to those switches. It has built-in functions that support using different types of controllers and switches. We can create also complex custom scenarios using the Mininet Python API. Compared to any other network emulator mininet is easy to use and open source. Due to its' many inbuilt features, it makes it favorable for research work in the field of SDN. We don't have any other emulator with the capability of developing python programs to implement the topology.

There are other emulators like Emulab [24], VINI [25], GENI [26], FIRE [27] which provides a global implementation of the SDN architecture. Though it's not very suitable for experimental and research purposes as it is heavy on the system and difficult to implement the infrastructure. While Mininet runs surprisingly well on a single laptop by leveraging Linux features (processes and virtual Ethernet pairs in network namespaces) to launch networks with gigabits of bandwidth and hundreds of nodes (switches, hosts, and controllers).

4.2.1 Single Controller Mininet Topology

For testing purposes, we created two topologies. We created single controller topology using Miniedit[17]. We have used POX remote controller *c0* to establish SDN architecture the SDN in the miniedit. The switches are the OpenFlow switches *s1, s2, s3*. There are 6 ghosts naming *h1, h2, h3, h4*, *h5*, and *h6*. In the single controller, all the OpenFlow switches are communicating with the single controller.

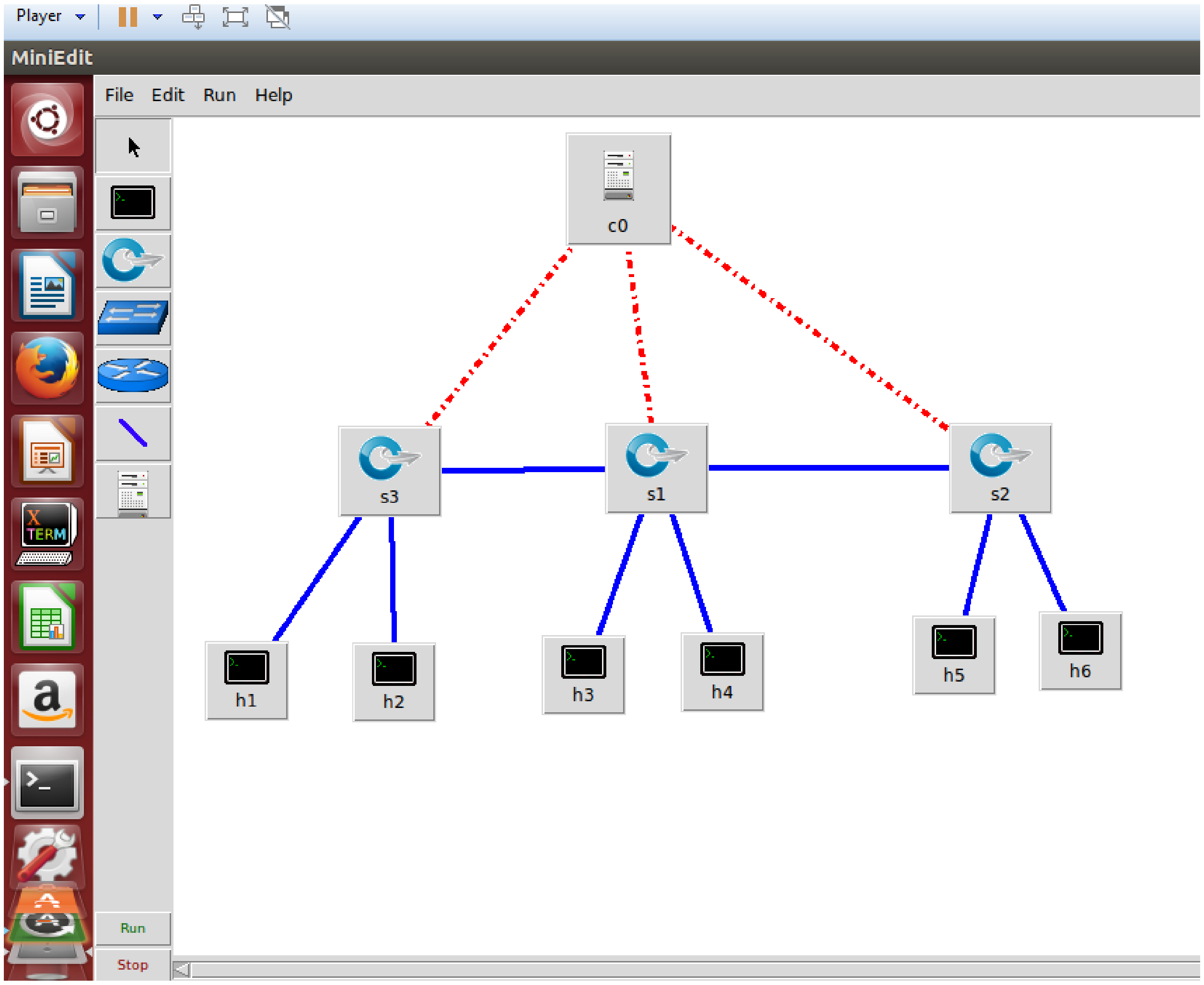


Fig 4.1 Single Controller Topology

4.2.1 Two and Three Controller Mininet Topology

To compare the performance, we created another topology i.e. multiple controllers' topology using miniedit. We have used POX remote controller *c0, c1 and c2* to establish SDN architecture the SDN in the miniedit. The switches are the OpenFlow switches *s1, s2, s3*. There are 6 ghosts naming *h1, h2, h3, h4, h5*, and *h6*. In this topology, OpenFlow switches have two controllers to manage the network traffic.

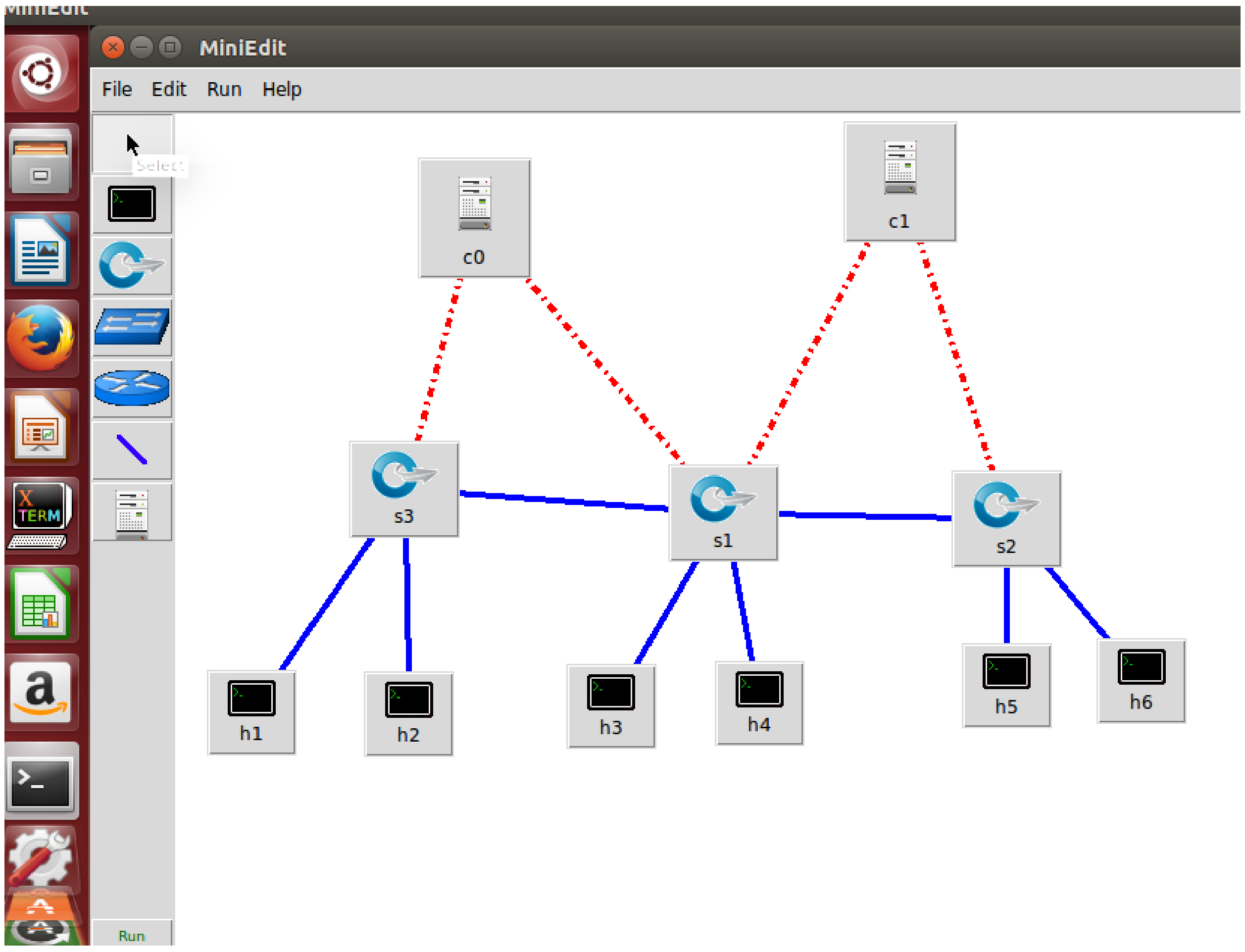


Fig 4.2 Multiple Controllers Topology

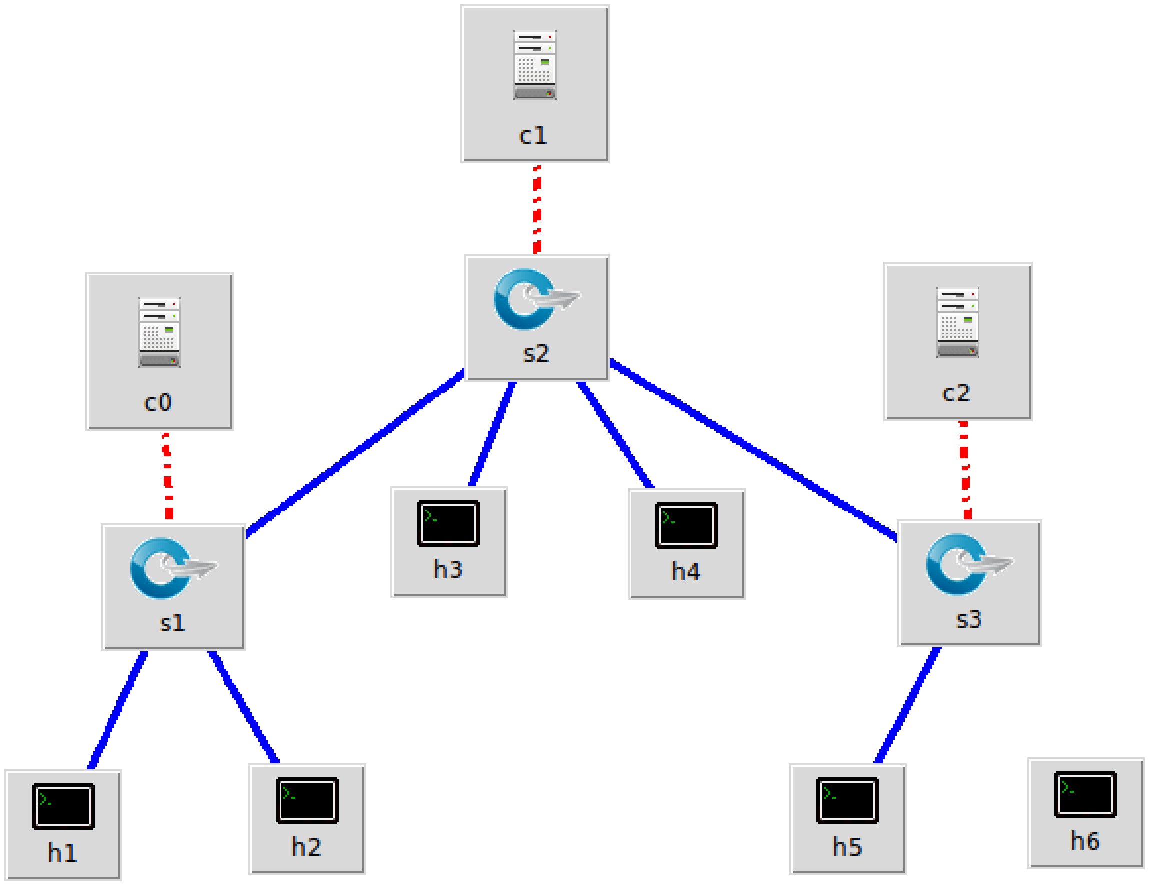


Figure 4.3 Three Controller Topology

4.3 POX Controller

POX is one of the many controllers developed for the SDN implementation. POX is a sister project of NOX and became more rapid than NOX[20]. POX is python based OpenFlow controller. It is an open source development platform. The main advantage if POX is that it is Python-based Software Defined Networking controller to develop networking applications. The reason we used POX is it's readily available, easy to learn and develop applications. It can run anywhere as it can bundle with PyPy[19] runtime which is an easy and free install. The POX is one of the controllers which is dedicated to creating an archetypal, modern Software Defined Controller. There are different SDN controllers available to use like ONOS [21], Floodlight[22], NOX, etc.

The reason we choose POX over other controllers as different controllers have a different requirement. Ryu [23] is one of the python based SDN controllers but cannot establish inter-controller communication. For future work Ryu controller, won't be efficient and doesn't support multicontroller architecture. ONOS is one of the controllers which supports distributed architecture but has a lot of dependencies. It's still underdeveloped and makes it difficult develop new applications on top of it. Floodlight is one of the most widely used SDN controllers but the API is Java-based and cannot be developed using python.

POX can be immediately used as a basic SDN controller by using the stock components that come bundled with it. POX frameworks are for both sides of OpenFlow, one is for switch side and another one is for controller side. The aggregator is a switch, allowing other controllers to connect to it and send it OpenFlow commands and so forth. But underneath, it implements this by controlling other OpenFlow switches.

4.4 D-ITG Tool

It's a distributed Internet Traffic Generator tool which can support both IPv4[21] and IPv6[22] traffic generation. It can produce the traffic at the packet level. We have used D-ITG to fill the capacity of the switches to understand the behavior of the topology. It can replicate the appropriate stochastic processes for both IDT (Inter-Departure Time) and PS (Packet Size) random variables. The D-ITG tool is used as it helps to create multiple flows simultaneously which will affect the performance of the network. It can be used on each used for sending and receiving packets using xterm [23]. The ease of the tool makes it more user-friendly and quick to learn. This tool is the best to prove the hypothesis. Also, it can be easily installed and used with mininet and give better results than any other traffic generator.

The tool helps to send and receive the packets with following commands

To simultaneously generate more than one flow, you have to prepare a script file like those shown in the following examples. Three UDP flows with different constant bit rate and remote log:

1. start the log server on the log host:

   ```
   $ ./ITGLog
   ```

2. start the receiver on the destination host:

   ```
   $ ./ITGRecv
   ```

3. create the script file:

   ```
   $ cat > script_file <<END
   -a 10.0.0.3 -rp 10001 -C 1000 -c 512 -T UDP
   -a 10.0.0.3 -rp 10002 -C 2000 -c 512 -T UDP
   -a 10.0.0.3 -rp 10003 -C 3000 -c 512 -T UDP
   END
   ```

   In the above example 10.0.0.3 is the destination of the packets
   10001 is the port number the receiver is listening.
   -C 1000 implies 1000 packets every second
   -c 512 says 512 bites and UDP is the time communication protocol between sender and receiver.

4. start the sender:

   ```
   $ ./ITGSend script_file -l send_log_file -L 10.0.0.4 UDP -X 10.0.0.4 UDP -x recv_log_file
   ```

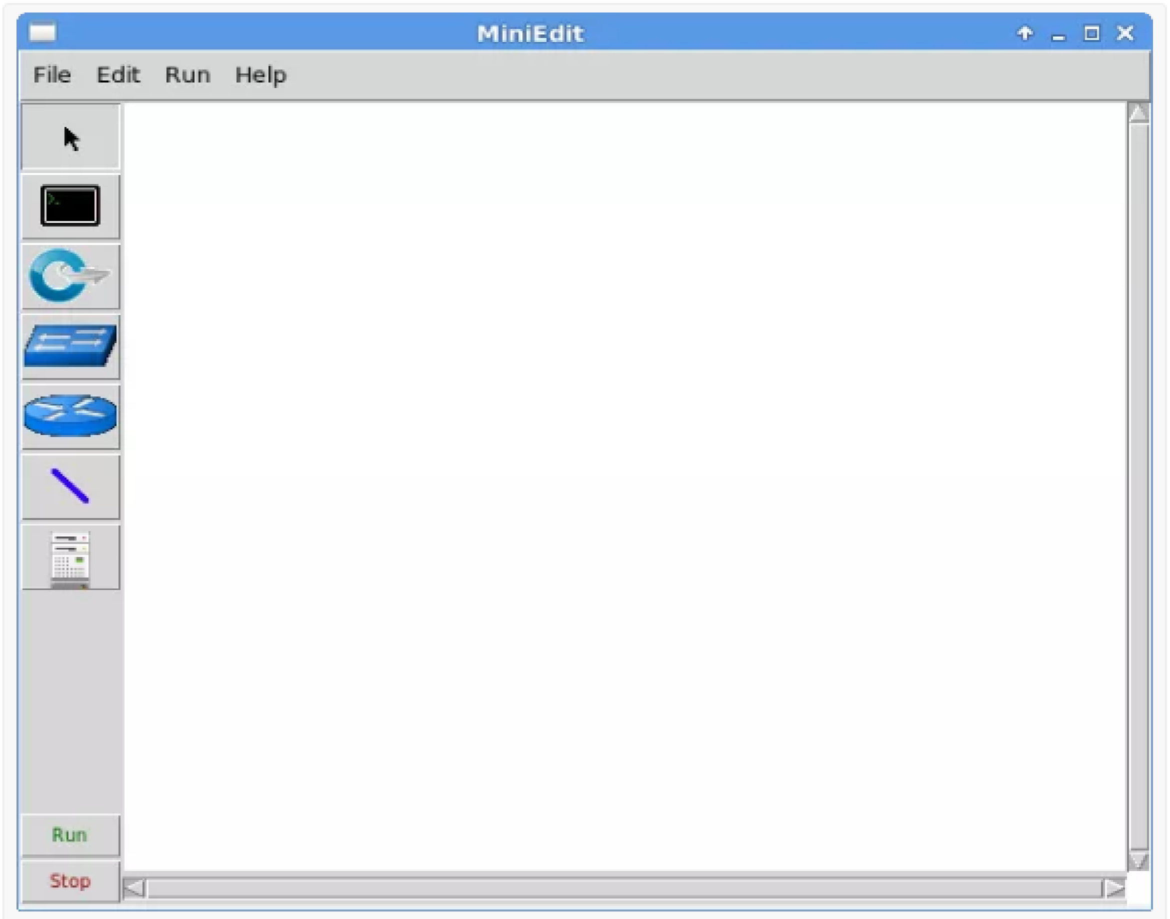
5. close the receiver by pressing Ctrl+C
6. close the log server by pressing Ctrl+C
7. decode the receiver log file on the log host:

```
$ ./ITGDec recv_log_file
```

```
----------------------------------------------------------
Flow number: 3
From 10.0.0.4:34775
To    10.0.0.3:10003
----------------------------------------------------------
Total time              =     10.016555 s
Total packets           =          6098
Minimum delay           =   3633.409810 s
Maximum delay           =   3634.259565 s
Average delay           =   3633.507249 s
Average jitter          =      0.002100 s
Delay standard deviation =      0.156419 s
Bytes received          =       3122176
Average bitrate         =   2493.612624 Kbit/s
Average packet rate     =    608.792145 pkt/s
Packets dropped         =         22216 (78.00 %)
----------------------------------------------------------
----------------------------------------------------------
Flow number: 1
From 10.0.0.4:34773
To    10.0.0.3:10001
----------------------------------------------------------
Total time              =      9.638360 s
Total packets           =          2269
Minimum delay           =   3633.402899 s
Maximum delay           =   3634.260524 s
Average delay           =   3633.461365 s
Average jitter          =      0.003114 s
Delay standard deviation =      0.135945 s
Bytes received          =       1161728
Average bitrate         =    964.253670 Kbit/s
Average packet rate     =    235.413494 pkt/s
Packets dropped         =          4274 (65.00 %)
----------------------------------------------------------
----------------------------------------------------------
Flow number: 2
From 10.0.0.4:34774
To    10.0.0.3:10002
----------------------------------------------------------
Total time              =     10.000351 s
Total packets           =          3136
Minimum delay           =   3633.407982 s
Maximum delay           =   3634.455203 s
Average delay           =   3633.514464 s
Average jitter          =      0.002740 s
Delay standard deviation =      0.221725 s
Bytes received          =       1605632
Average bitrate         =   1284.460515 Kbit/s
Average packet rate      =    313.588993 pkt/s
Packets dropped          =         16864 (84.00 %)
----------------------------------------------------------

****************  TOTAL RESULTS   ******************
Number of flows          =            3
Total time               =     10.038005 s
Total packets            =         11503
Minimum delay            =   3633.402899 s
Maximum delay            =   3634.455203 s
Average delay            =   3633.500165 s
Average jitter           =      0.003291 s
Delay standard deviation =      0.417552 s
Bytes received           =       5889536
Average bitrate          =   4693.790051 Kbit/s
Average packet rate      =   1145.944837 pkt/s
Packets dropped          =         43354 (79.00 %)
Error lines              =             0
```

4.5 Miniedit

The miniedit is a simple GUI editor included in mininet. It helps to understand how mininet can be extended and how the topology looks like. MiniEdit has a simple user interface that presents a canvas with a row of tool icons on the left side of the window, and a menu bar along the top of the window. The user interface of miniedit GUI looks as in the image below:



MiniEdit canvas

Chapter 5

Evaluation of Results

To evaluate the system, we created single controller topology and multiple controller topologies in mininet. We created a topology with three OpenFlow switches, 6 hosts and remote controllers using miniedit. We are using a script to flood the capacity of the switches. After flooding the capacity, we analyzing the behavior of the switches with respect to the controller. For filling the capacity, we sending a different number of packets from hosts to different hosts simultaneously to understand the network behavior under the load.

5.1 Single controller topology evaluation

For the single controller, we configured the controller in miniedit with port no and IP address of the machine. Then we run the topology on the mininet folder. The port no. and *IP* address of the controller was set at 6633 and 10.0.0.49 (IP address of the host machine) as shown in figure 5.1. The IP addresses for the 6 hosts machine are set 10.0.0.1, 10.0.0.2, 10.0.0.3, 10.0.0.4, 10.0.0.5, and 10.0.0.6 respectively. The POX controller is invoked using command "./pox.py forwarding.l2_pairs openflow.of_01 --port=6633" in one machine to establish the connection between the switch and the controller. After this, the mininet topology is started using miniedit which shows the switch, hosts and controller connection as shown in figure 5.3.

So, to evaluate the performance, we fill the capacity with a different number of packets. We flood switches by simultaneously sending different packets. Calculate the delay, average delay, and load jitter by observed the flow of the packets. We observe the switch controller communication to later compare it with the other topology. Once we start the traffic bridge between the hosts and client with the help of xterm and D-ITG tool we see how the controller responds and switch handles the traffic.

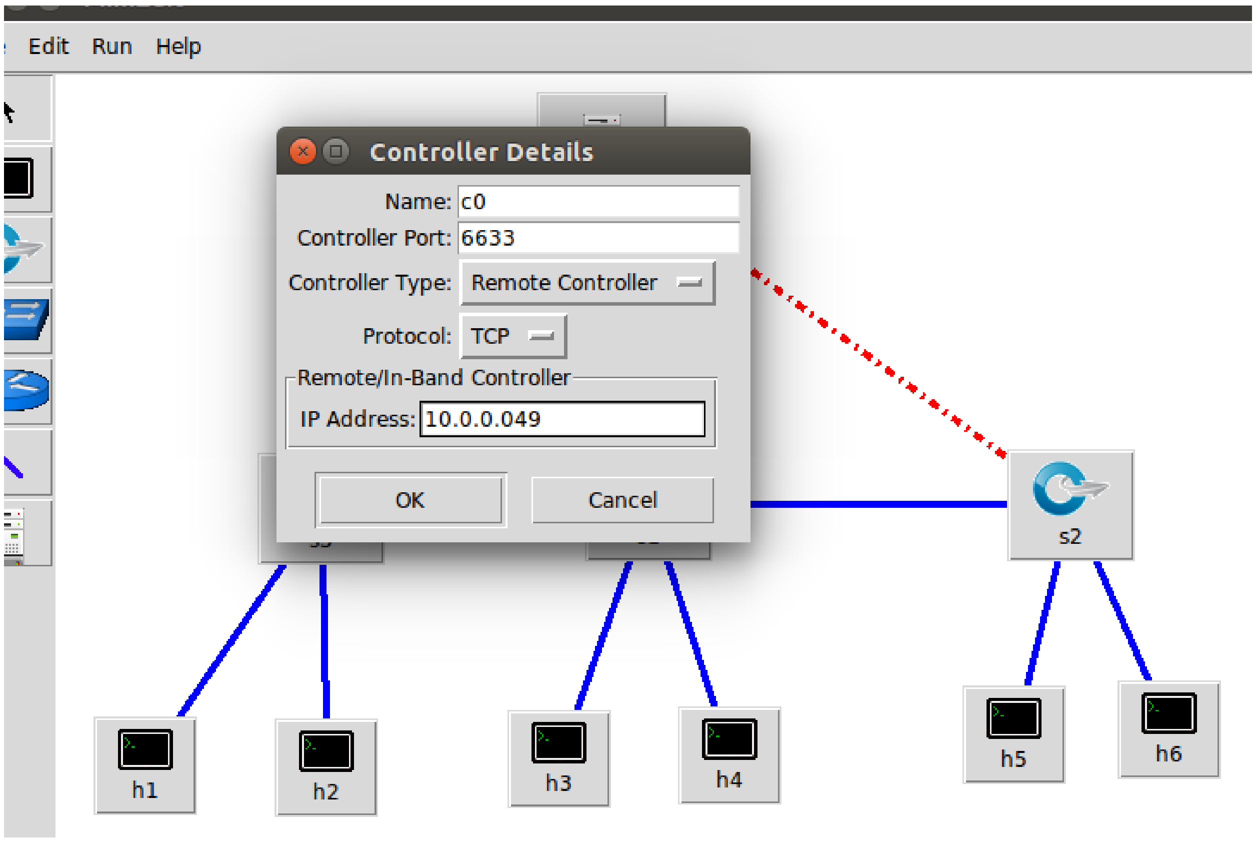
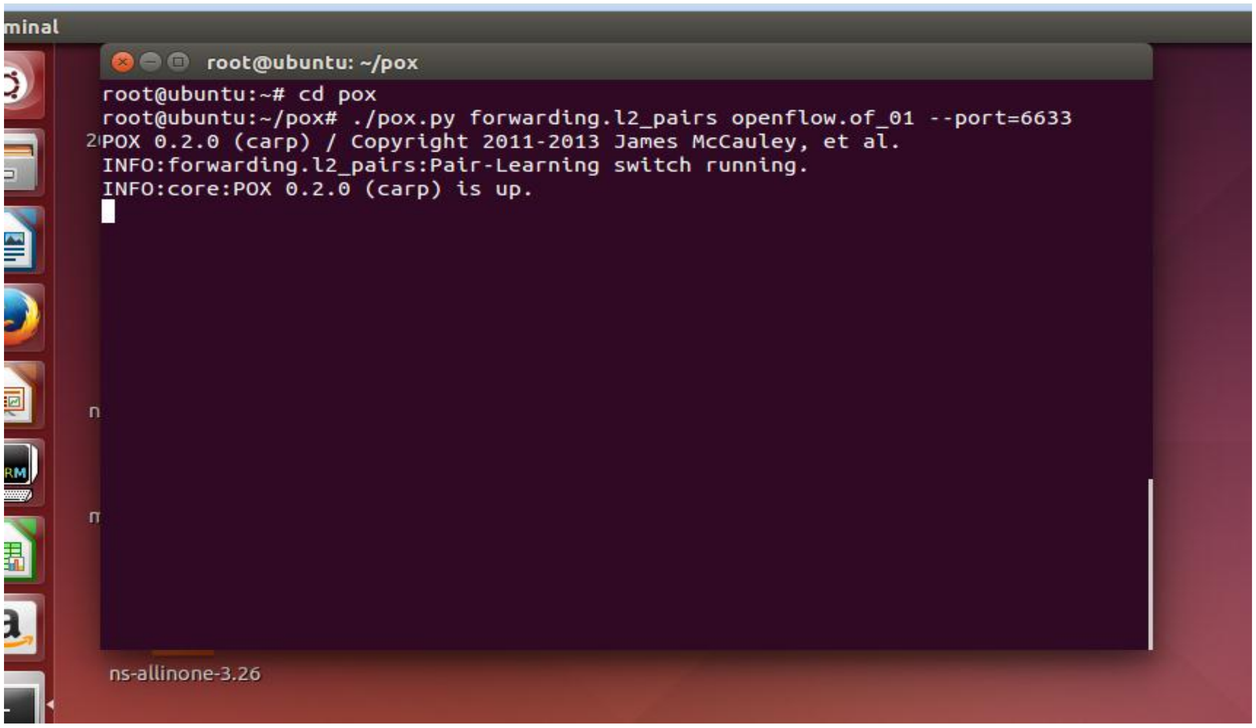
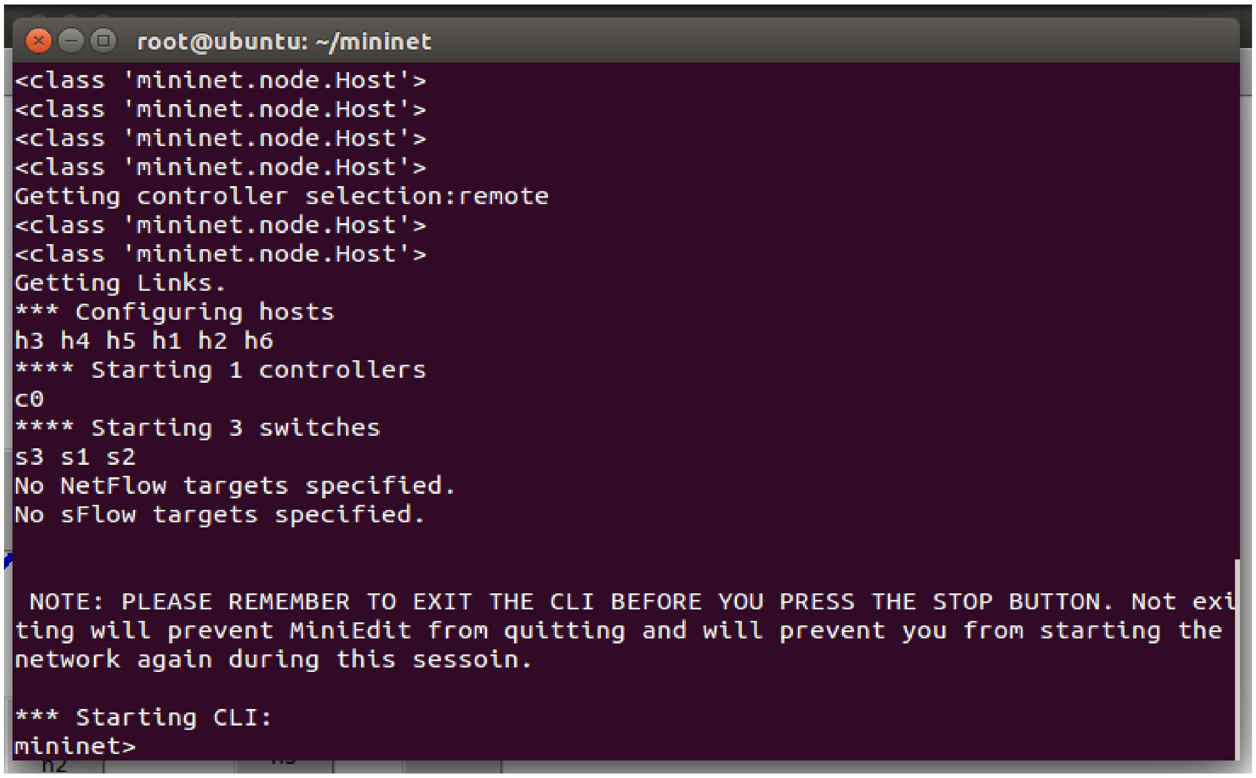

Controller Details
Name: c0
Controller Port: 6633
Controller Type: Remote Controller
Protocol: TCP
Remote/In-Band Controller
IP Address: 10.0.0.049

Fig 5.1 Controller configuration in Single topology



```
root@ubuntu:~# cd pox
root@ubuntu:~/pox# ./pox.py forwarding.l2_pairs openflow.of_01 --port=6633
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:forwarding.l2_pairs:Pair-Learning switch running.
INFO:core:POX 0.2.0 (carp) is up.
```

Fig 5.2 POX Controller with port no 6633 invoked



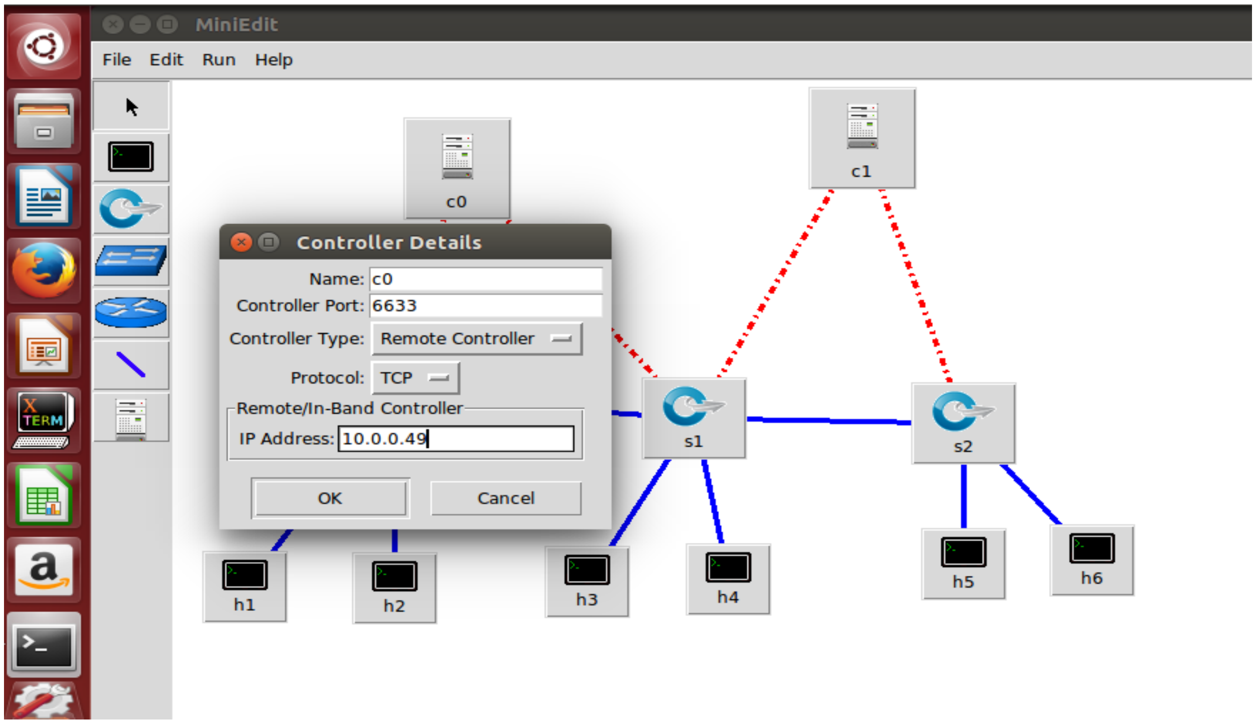
```
<class 'mininet.node.Host'>
<class 'mininet.node.Host'>
<class 'mininet.node.Host'>
<class 'mininet.node.Host'>
Getting controller selection:remote
<class 'mininet.node.Host'>
<class 'mininet.node.Host'>
Getting Links.
*** Configuring hosts
h3 h4 h5 h1 h2 h6
**** Starting 1 controllers
c0
**** Starting 3 switches
s3 s1 s2
No NetFlow targets specified.
No sFlow targets specified.


 NOTE: PLEASE REMEMBER TO EXIT THE CLI BEFORE YOU PRESS THE STOP BUTTON. Not exi
ting will prevent MiniEdit from quitting and will prevent you from starting the
network again during this sessoin.

*** Starting CLI:
mininet>
```

Fig 5.3 Mininet CLI

5.2 Multiple controllers' topology evaluation

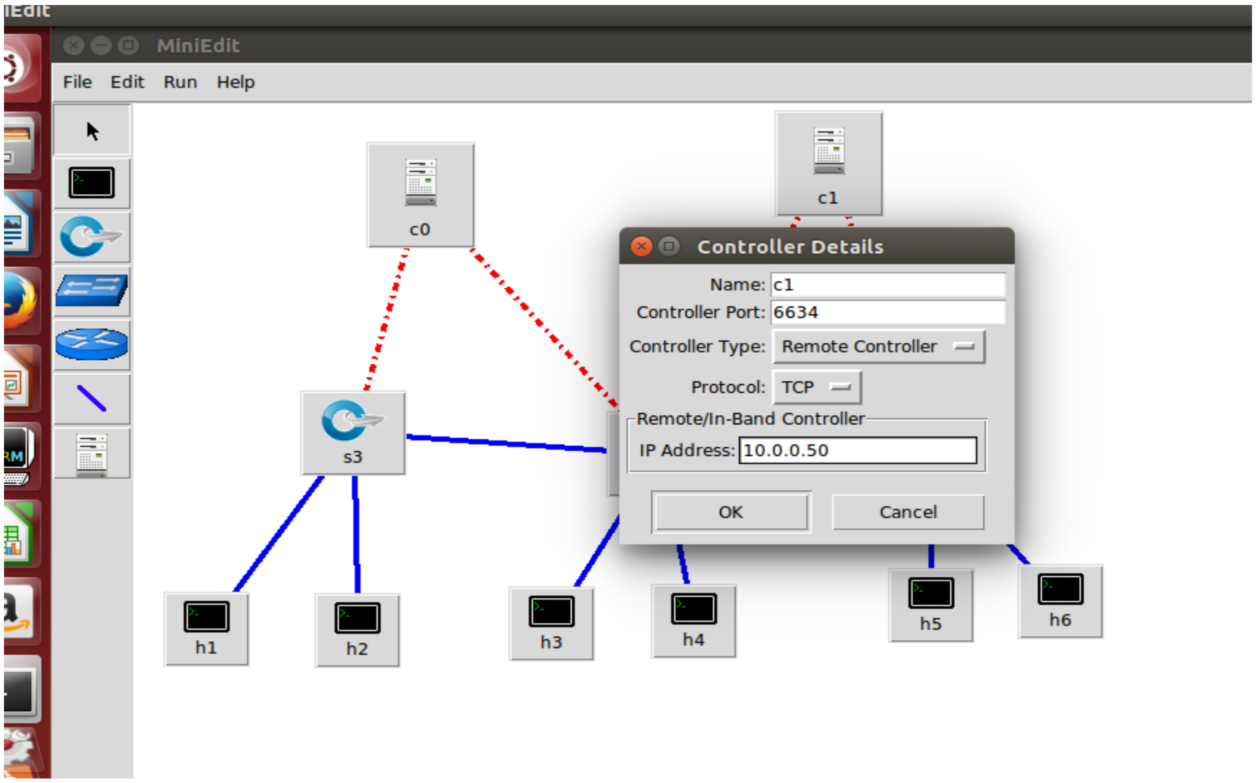
For multiple controllers, we configured the controllers in miniedit with port no and IP address of the machine. Then we run the topology on the mininet folder. The port no. and *IP* address for the first controller is set at 6634 and 10.0.0.49 (IP address of the one machine) and another controller is set at 6634 and 10.0.0.50 as shown in figure 5.4 and 5.5 respectively. The IP addresses for the 6 hosts machine are set 10.0.0.1, 10.0.0.2, 10.0.0.3, 10.0.0.4, 10.0.0.5, and 10.0.0.6 respectively.

After configuration, we fill the capacity with a different number of packets. We flood switches by simultaneously sending different packets. Calculate the delay, average delay, and load jitter and observed the flow of the packets. We observe the switch controller communication to later compare it with the other topology. Now the switches in this topology communicate with two controllers to balance the load. The load in the multicontroller topology is balanced in a better way. Then we observe packets loss ratio, total delay, and load jitter on the xterm window to evaluate.



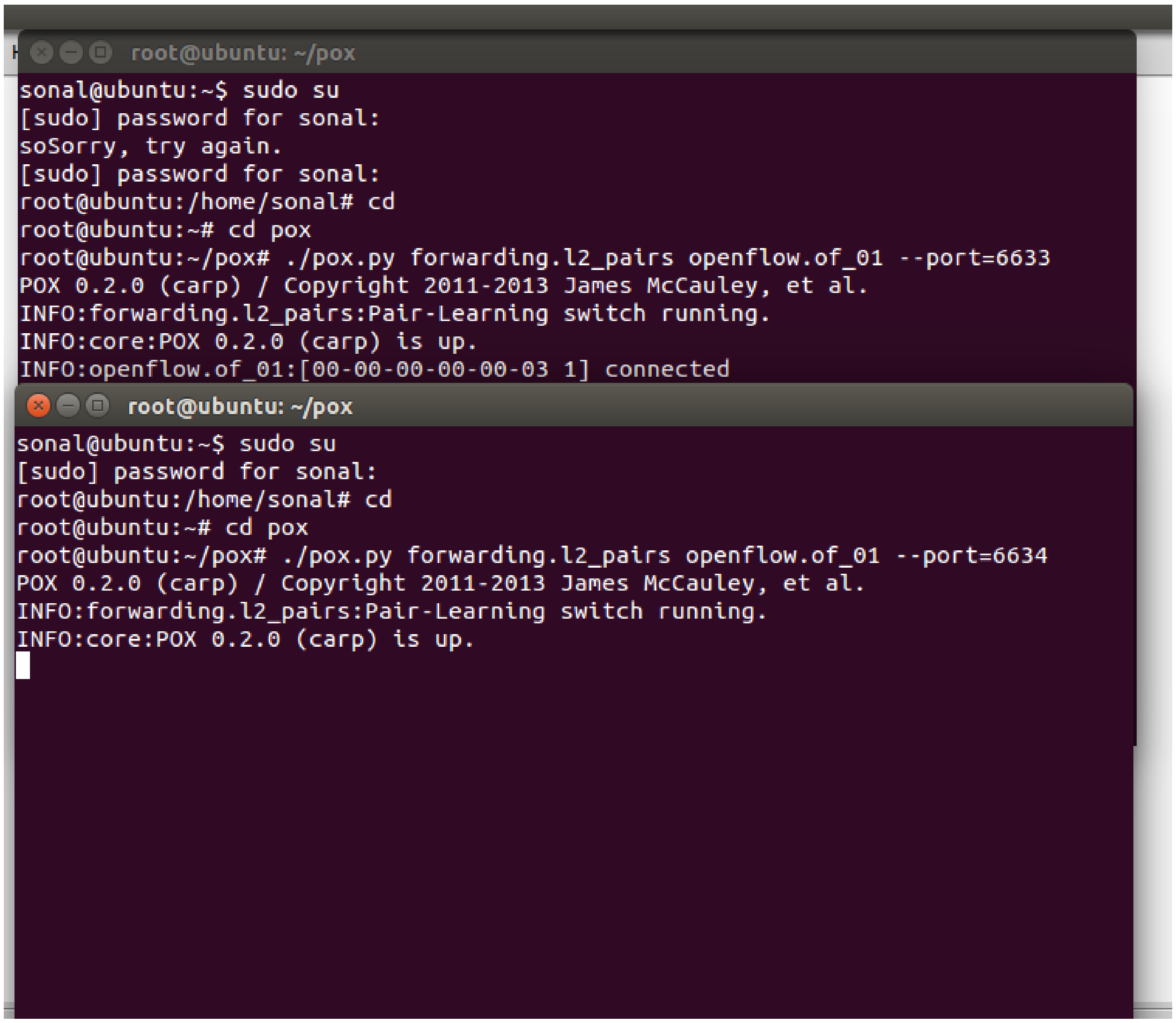
Controller Details
Name: c0
Controller Port: 6633
Controller Type: Remote Controller
Protocol: TCP
Remote/In-Band Controller
IP Address: 10.0.0.49

Fig 5.4 First Controller c0 configuration



Controller Details
Name: c1
Controller Port: 6634
Controller Type: Remote Controller
Protocol: TCP
Remote/In-Band Controller
IP Address: 10.0.0.50

Fig 5.5 Second controller c1 configuration



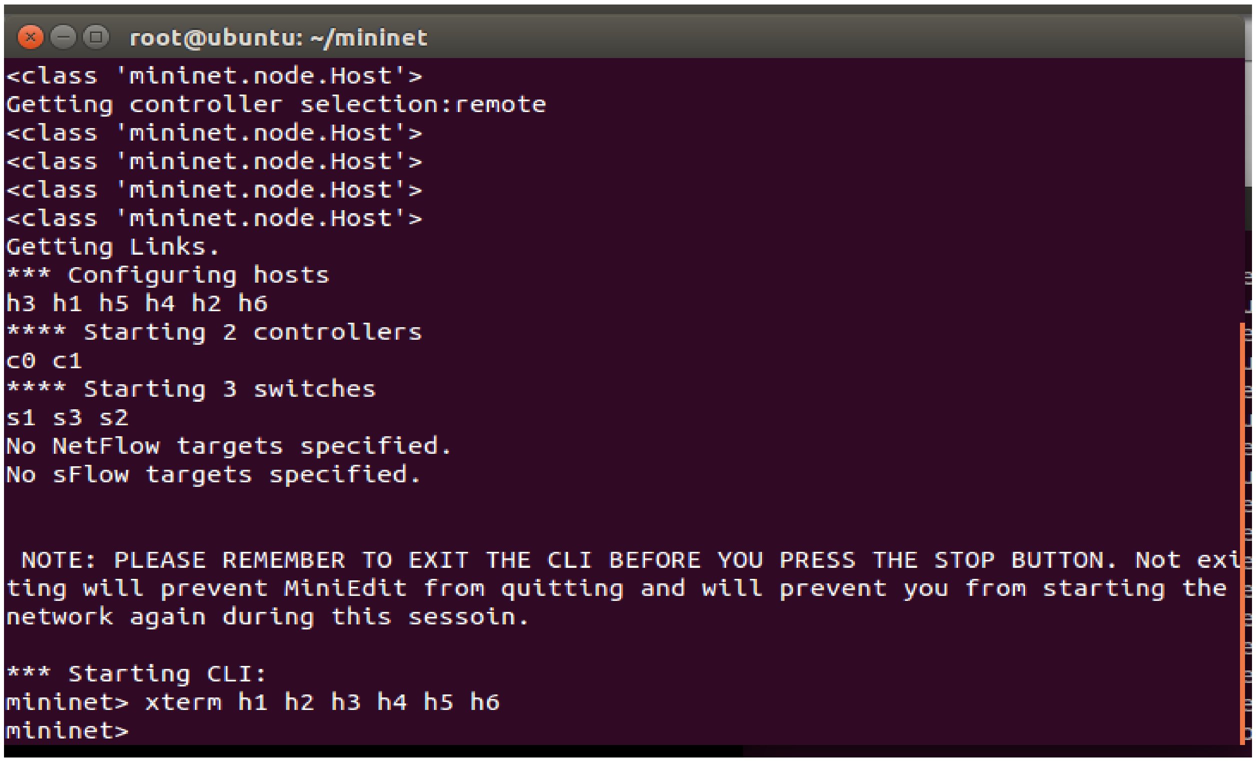
```
root@ubuntu: ~/pox
sonal@ubuntu:~$ sudo su
[sudo] password for sonal:
soSorry, try again.
[sudo] password for sonal:
root@ubuntu:/home/sonal# cd
root@ubuntu:~# cd pox
root@ubuntu:~/pox# ./pox.py forwarding.l2_pairs openflow.of_01 --port=6633
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:forwarding.l2_pairs:Pair-Learning switch running.
INFO:core:POX 0.2.0 (carp) is up.
INFO:openflow.of_01:[00-00-00-00-00-03 1] connected
```

```
root@ubuntu: ~/pox
sonal@ubuntu:~$ sudo su
[sudo] password for sonal:
root@ubuntu:/home/sonal# cd
root@ubuntu:~# cd pox
root@ubuntu:~/pox# ./pox.py forwarding.l2_pairs openflow.of_01 --port=6634
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:forwarding.l2_pairs:Pair-Learning switch running.
INFO:core:POX 0.2.0 (carp) is up.
```

Fig 5.6 Two POX controllers invoked for multi-controller architecture

5.3 Traffic Generation

We use D-ITG to generate simultaneous traffic. We send multiple packets from h1 to all the other hosts i.e. from h2 to h6 to check how different topology reacts to manage the traffic. We modify the emulator with a different number of packets from h1 to each other hosts. We send 1000, 2000, 3000, 4000, and 5000 packets parallelly to all the hosts in the network. We repeat each experiment multiple times to efficiently collect the results. Then captured average values of packet loss, load jitter, delay and throughput for single controller topology and multiple controller topology. We send packets from h1 to h6 using the script in distributed traffic generator tool. We first xterm for which we have downloaded the xming[25] server. On the xterm window to send and receive packets on the host machines. D-ITG gives a summary of the packets received packets lost, average delay and load jitter which makes it easy to compare the different architectures.



```
<class 'mininet.node.Host'>
Getting controller selection:remote
<class 'mininet.node.Host'>
<class 'mininet.node.Host'>
<class 'mininet.node.Host'>
<class 'mininet.node.Host'>
Getting Links.
*** Configuring hosts
h3 h1 h5 h4 h2 h6
**** Starting 2 controllers
c0 c1
**** Starting 3 switches
s1 s3 s2
No NetFlow targets specified.
No sFlow targets specified.


 NOTE: PLEASE REMEMBER TO EXIT THE CLI BEFORE YOU PRESS THE STOP BUTTON. Not exi
ting will prevent MiniEdit from quitting and will prevent you from starting the
network again during this sessoin.

*** Starting CLI:
mininet> xterm h1 h2 h3 h4 h5 h6
mininet>
```

Fig 5.7 Xterm from Mininet

Chapter 6 Results

After setting up the topologies and conducting the experiments we calculating the important parameters affecting the network traffic. We calculated the average delay to emphasize the behavior and differentiate between single and two controller topology network performance. The reason to choose parameter shown in the following section is that they are important to analyze the network performance. The delay in the most crucial factor while handling the incoming traffic in any networking scenario as we cannot afford too much delay. If the delay is beyond the tolerance limit it will make the network inefficient and the whole system will be jeopardized. By the following graphs, we understand that the with multiple controller architectures we can keep all these parameters in check making the system architecture more efficient and reliable. Another parameter we calculated is the average jitter. One of the important factors for networking infrastructure is to be able to handle the load in an effective manner. As the load in any network can be unpredictable and can any time reach the maximum. So, we calculated the average jitter to highlight the reaction of the topology. Per the results and graphs, we understand that the multicontroller architecture is more stable with the increase in the load. We also observed the bitrate

The values reported in the results are computed as follows:

the Delay is computed as the difference between rxTime (the reception timestamp) and txTime(the transmission timestamp) of each packet.

The jitter has been calculated using below formula where S_i and R_i respectively correspond to txTime and rxTime.

$D_i = (R_i - S_i) - (R_{i-1} - S_{i-1})$

$D_i = (R_i - R_{i-1}) - (S_i - S_{i-1})$

$\text{AvgJitter} = \frac{\sum_i^n |D|}{n}$

6.1 Single Controller Evaluation Results

We flooding the capacity of the switches with different flows. We sent multiple packets using D-ITG tool to different hosts, observed the flows, and plotted the graph for the different parameters affecting the network performance. We analyzed the output of the D_ITG tool and plotted a graph for different values.

6.1.1 Average Delay

Below the graph is of average delay for the value collected for different packets for a single controller. *X-axis* represents the number of packets for each experiment and *y-axis* represents the delay calculated for different packets in seconds

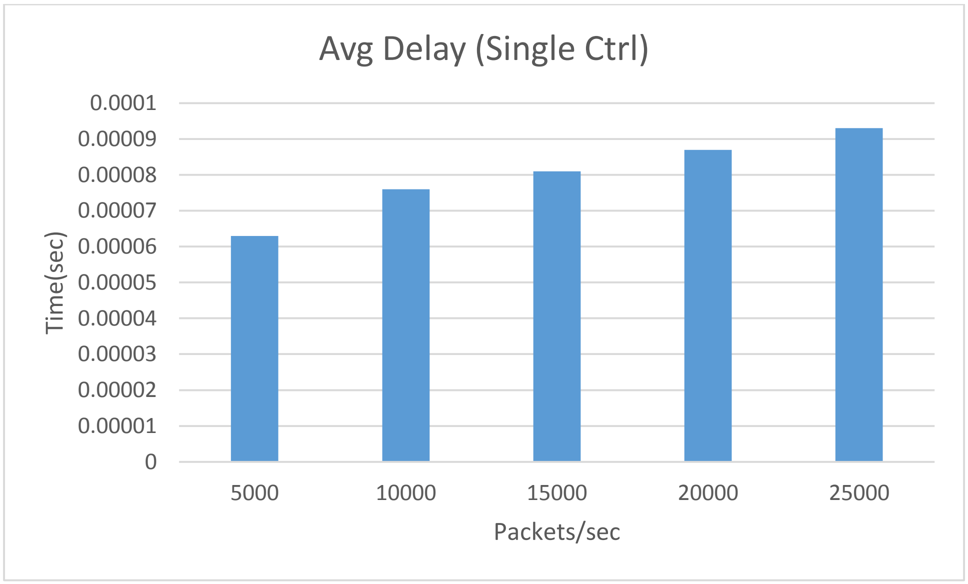


Fig 6.1 Plot for Average Delay in Single Controller Architecture

6.1.2 Maximum Delay

Below the graph is a maximum delay for the value collected for different packets for a single controller.

X-axis represents the number of packets for each experiment and *y-axis* represents the delay calculated for different packets in seconds

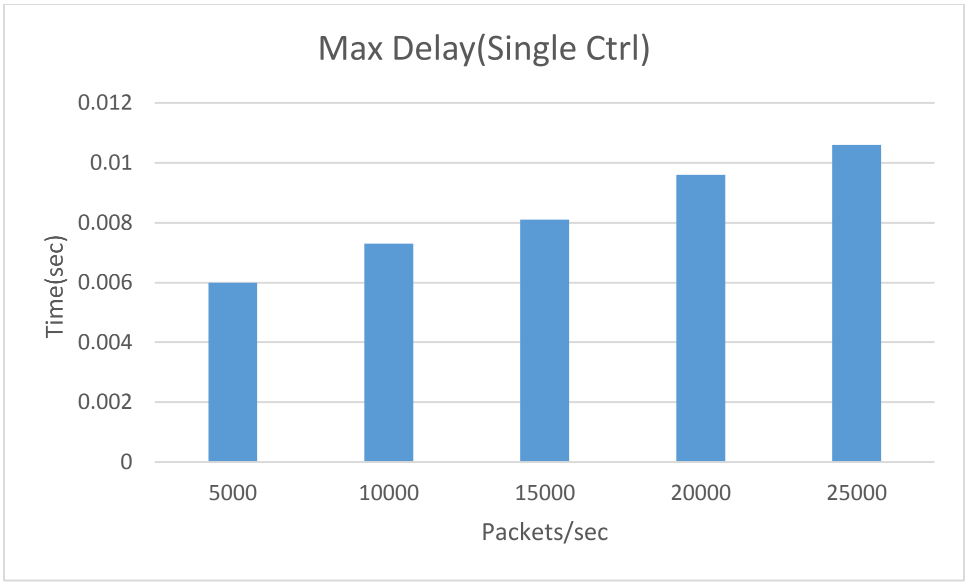


Fig 6.2 Plot for Maximum Delay in Single Controller Architecture

6.1.3 Load Jitter

Below is the graph of load jitter for the value collected for different packets for a single controller. *X-axis* represents the number of packets for each experiment and *y-axis* represents the average jitter calculated for different packets in seconds

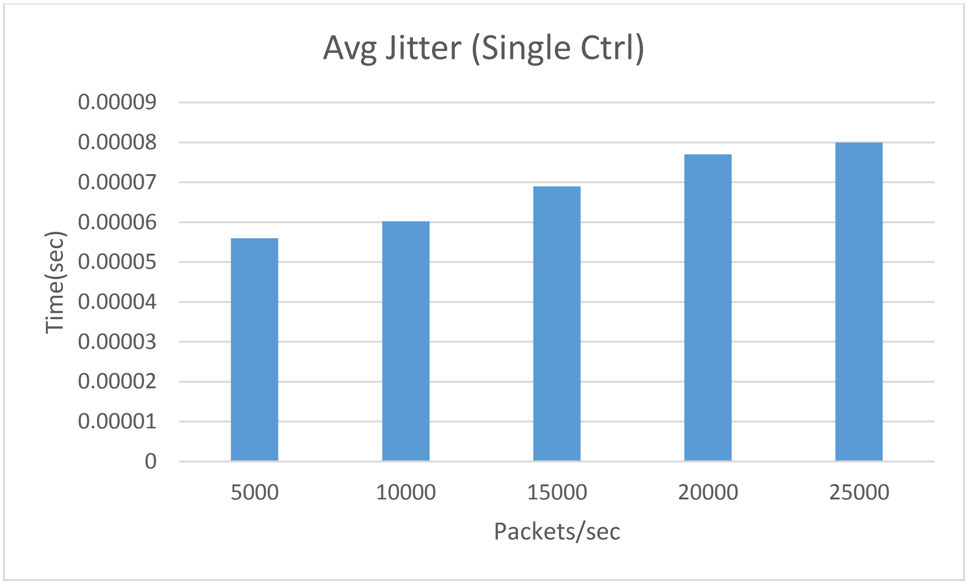


Fig 6.3 Plot for Average Jitter for Single Controller architecture

6.1.4 Average Bitrate

Below is the graph of average bitrate for the value collected for different packets for a single controller. *X-axis* represents the number of packets for each experiment and *y-axis* represents the average bitrate calculated for different packets in bite.

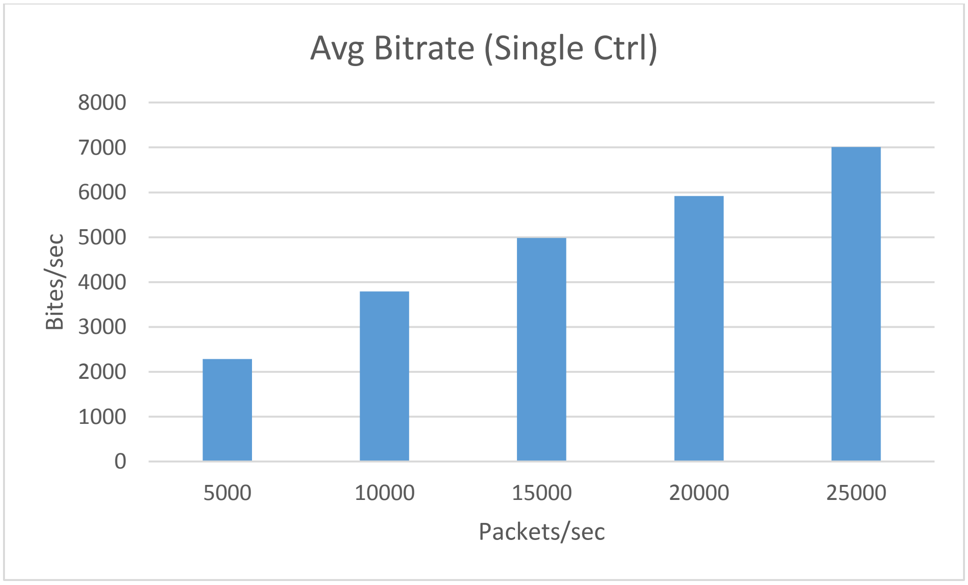


Fig 6.4 Plot for Average bit rate for Single Controller

6.1.5 Average Loss-Burst size

Below is the graph of average bitrate for the value collected for different packets for a single controller. *X-axis* represents the number of packets for each experiment and *y-axis* represents the number of packets lost. The average lost burst ratio is the average of the packet loss rate

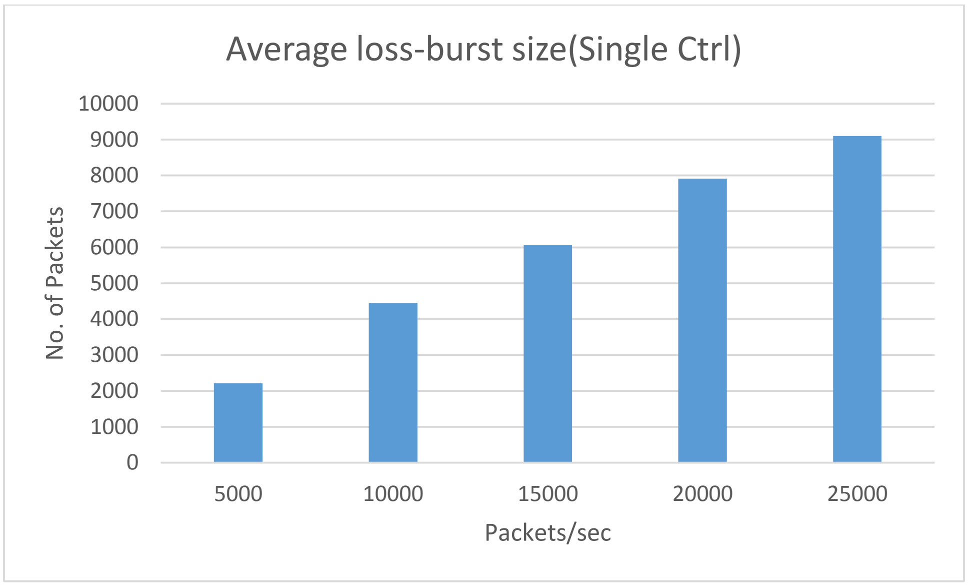


Fig 6.5 Plot for Average loss burst size for single controller

6.2 Multiple Controller Evaluation Results

Same as previous, we flooding the capacity of the switches with different flows. We sent multiple packets using D-ITG tool to different hosts, observed the flows and plotted the graph for the different parameters affecting the network performance. We analyzed the output of the D_ITG tool and plotted a graph for different values.

6.2.1 Average Delay

Below is the graph of average delay for the value collected for different packets for two controllers. *X-axis* represents the number of packets for each experiment and *y-axis* represents the delay calculated for different packets in seconds.

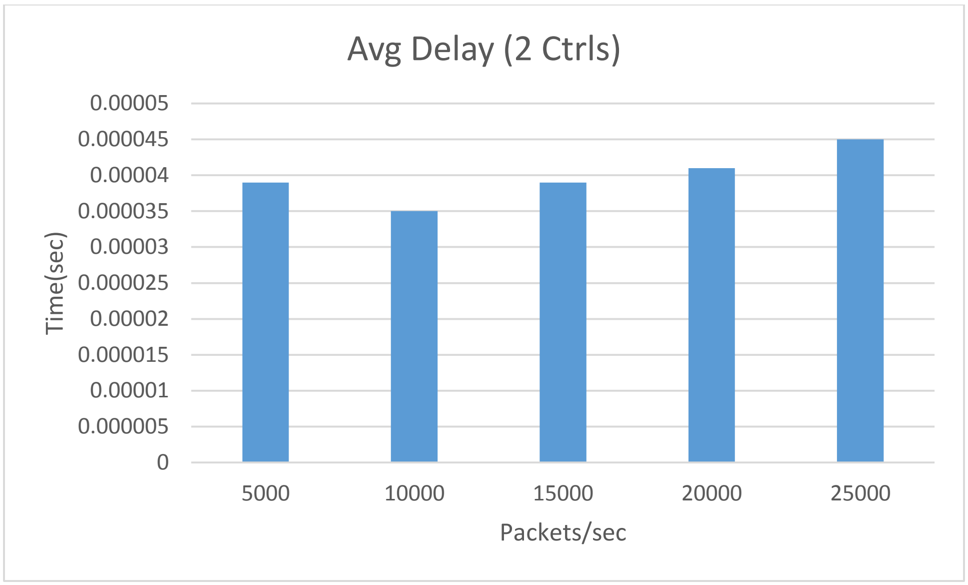


Fig 6.6 Plot for Average bit rate for Two Controller

6.2.2 Maximum Delay

Below is the graph of a maximum delay for the value collected for different packets for two controllers. *X-axis* represents the number of packets for each experiment and *y-axis* represents the delay calculated for different packets in seconds.

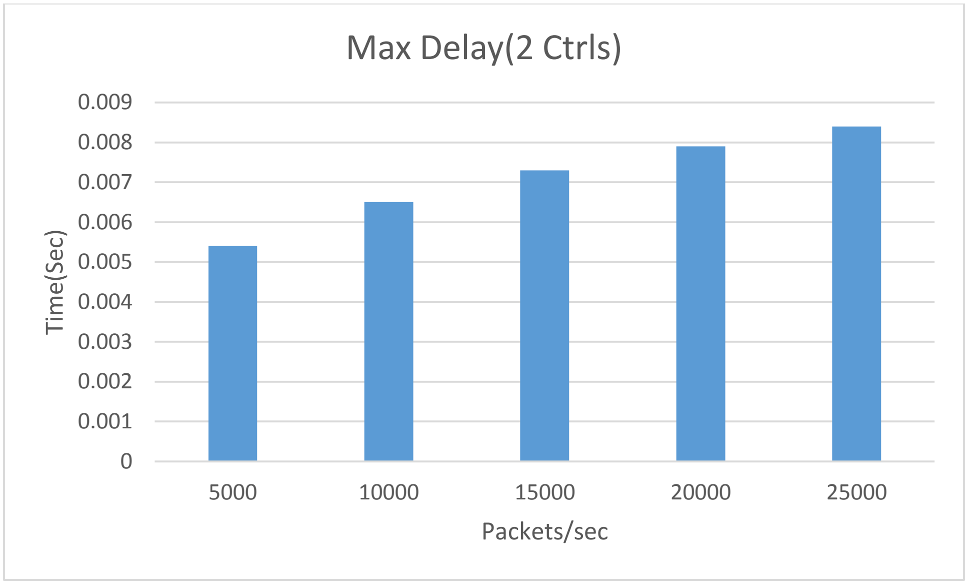


Fig 6.7 Plot for Average Delay in Two Controller Architecture

6.2.3 Load Jitter

Below is the graph of the load jitter for the value collected for different packets for two controllers.

X-axis represents the number of packets for each experiment and *y-axis* represents the average jitter calculated for different packets in seconds.

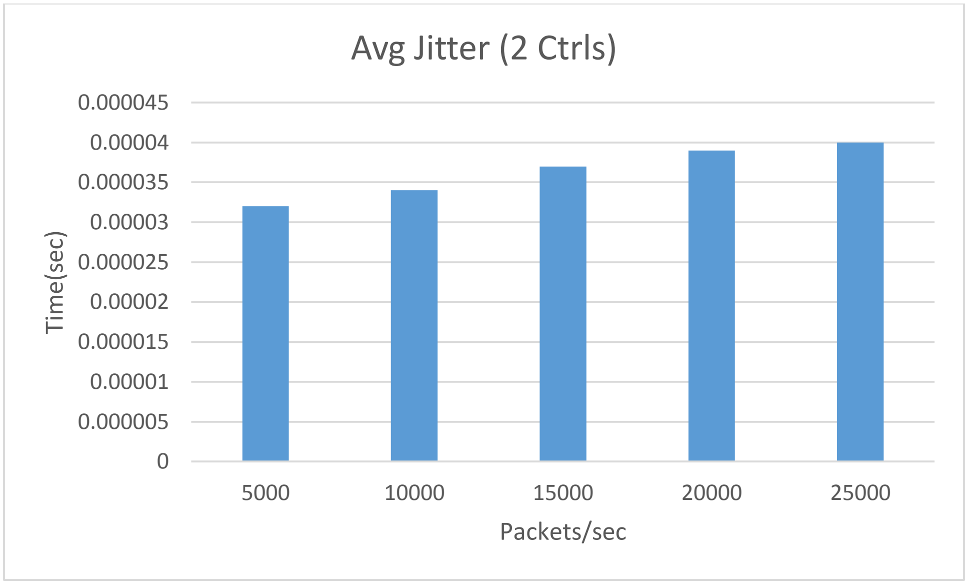


Fig 6.8 Plot for Maximum Delay in Two Controller Architecture

6.2.4 Average Bitrate

Below is the graph of average bitrate for the value collected for different packets for two controllers. *X-axis* represents the number of packets for each experiment and *y-axis* represents the average bitrate calculated for different packets in bite.

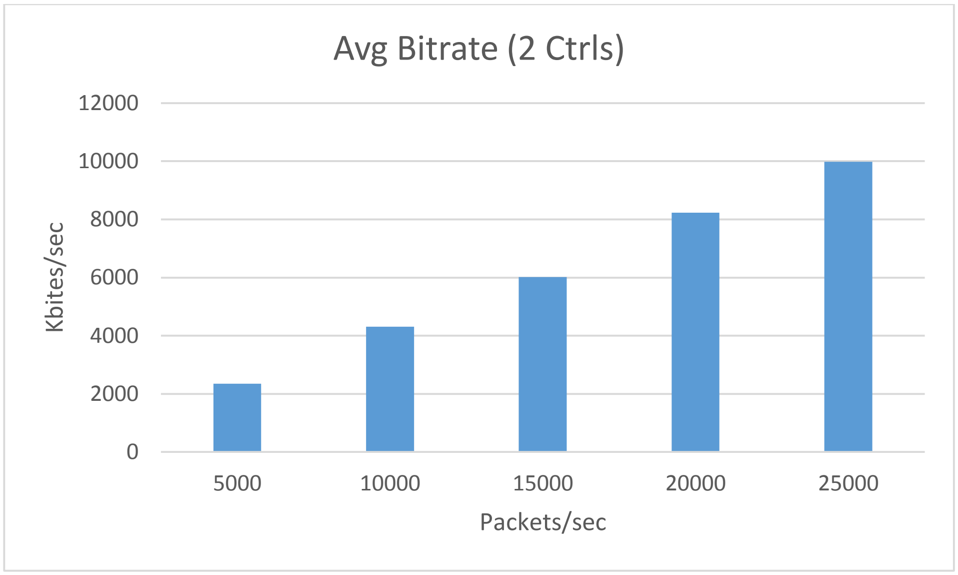


Fig 6.9 Plot for Average Bitrate in Two Controller Architecture

6.1.5 Average Loss-Burst size

Below the graph is of average bitrate for the value collected for different packets for two controllers. *X-axis* represents the number of packets for each experiment and *y-axis* represents the number of packets lost. The average lost burst ratio is the average of the packet loss rate.

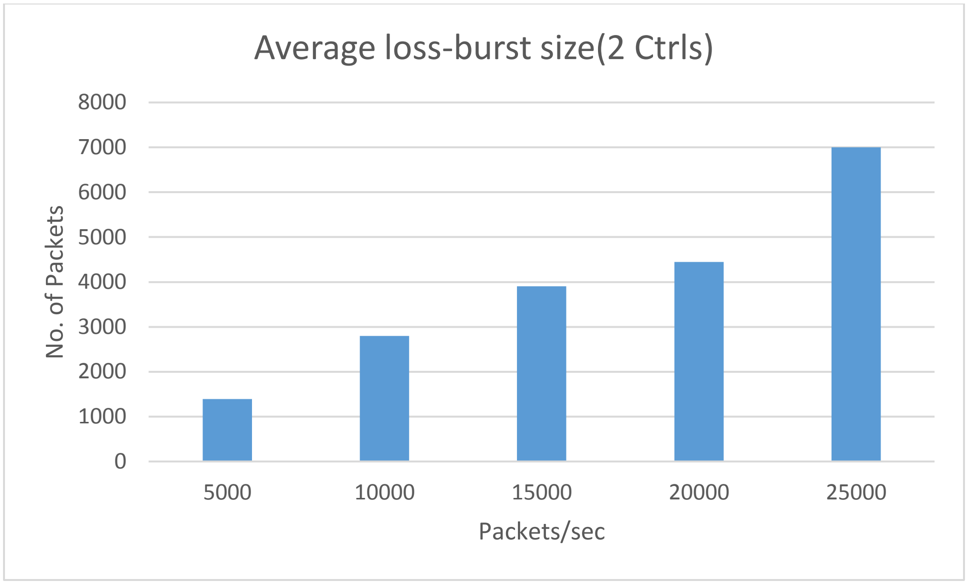


Fig 6.10 Plot for Average Jitter in Two Controller Architecture

6.3 Comparison of Single Controller v/s Multiple Controller Topology

Using the experimental result of each topology we compared the performance of both the topologies. We calculated and compared the results and plotted a graph to understand which topology is better. After analyzing the results on the graph for throughput, delay, jitter, and bitrate we could understand the multicontroller topology can handle the load in a better way.

We found that throughput for the multicontroller topology is better than the single controller topology. As the switches have two controllers to communicate and push the rules the load divided between the two controllers. Through the graphs, we can understand that as the network grows an SDN architecture with multiple controllers is always better over SDN architecture with a single controller.

6.3.1 Average Delay Graph

The delay parameter is important in networking. The more the delay the less reliable the environment So we observed the delay by sending multiple packets simultaneously. The calculation of average delay explains that with the increase in the packets flow the delay increases and it's much more than can be expected. While in the two-controller architecture, there is not much change in the delay in the results.

Packets	Avg Delay (Single Ctrl)	Avg Delay (2 Ctrls)	Avg Delay(3 ctrls)
5000	0.000063	0.000035	0.000017
10000	0.000076	0.000037	0.0000195
15000	0.000081	0.000039	0.000021
20000	0.000087	0.000041	0.000024
25000	0.000093	0.000045	0.0000275

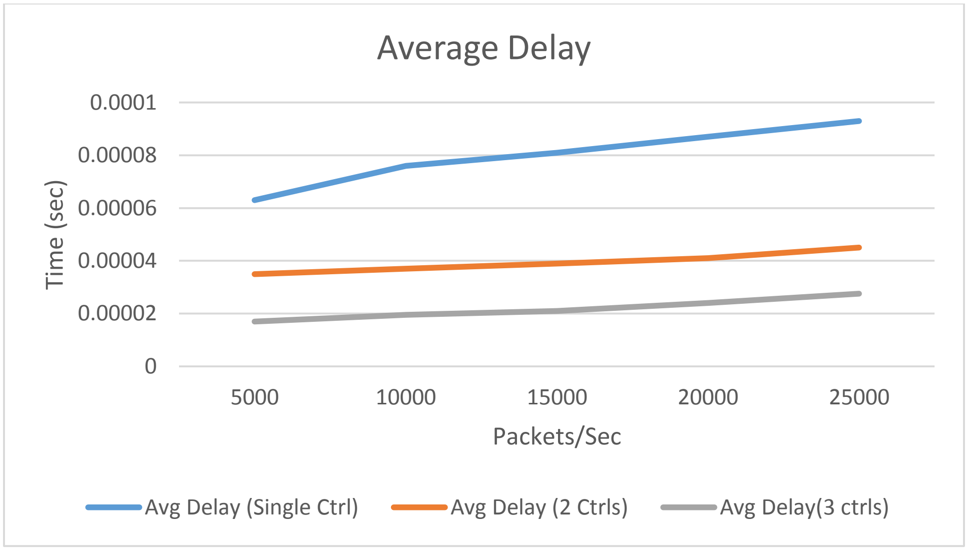


Fig 6.11 Average Delay Single v/s multiple

6.3.2 Maximum Delay Graph

The maximum delay for the single controller topology is way too high with the increase in the load.

Packets	MaxDelay(Single Ctrl)	Max Delay(2 Ctrls)	Max Delay(3 Ctrls)
5000	0.006	0.0054	0.00200825
10000	0.0073	0.0065	0.002857
15000	0.0081	0.0073	0.00369
20000	0.0096	0.0079	0.00445
25000	0.0106	0.0084	0.0059

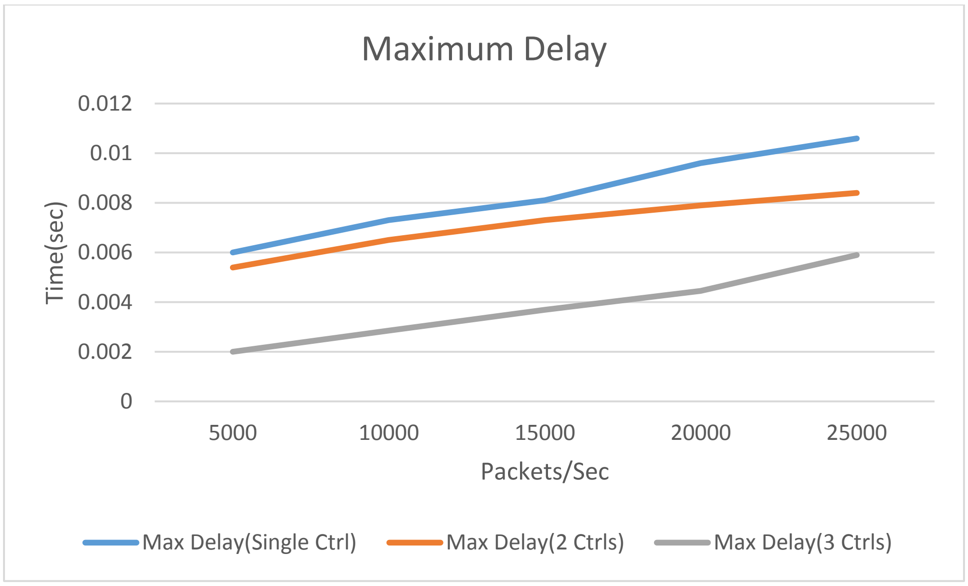


Fig 6.12 Maximum Delay Single v/s Multiple

6.3.3 Average Jitter

After running the network for several times, we calculated the average as it affects the performance. We calculated the value of average jitter and recorded for different packets which prove that multicontroller architecture ca handle the load in a better way. We can imply that the two controllers can handle the load with the minimum issue.

Packets	Avg Jitter (Single Ctrl)	Avg Jitter (2 Ctrls)	Avg Jitter (3 Ctrls)
5000	0.000056	0.000032	0.000011
10000	0.0000602	0.000034	0.0000145
15000	0.000069	0.000037	0.0000175
20000	0.000077	0.000039	0.0000205
25000	0.00008	0.00004	0.000023

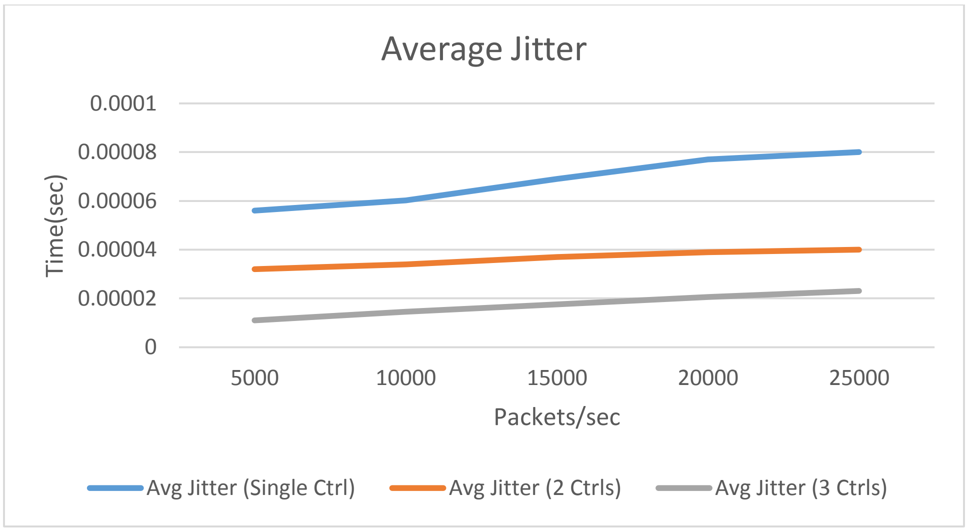


Fig 6.13 Average Jitter Single v/s Multiple

6.3.4 Average bitrate

Packets	Avg Bitrate (Single Ctrl)	Avg Bitrate (2 Ctrls)	Avg Bitrate (3 Ctrls)
5000	2100	2346	4750
10000	3793	4310	6590
15000	4987	6019	7185
20000	5917	8232	9876
25000	7010	9987	13980

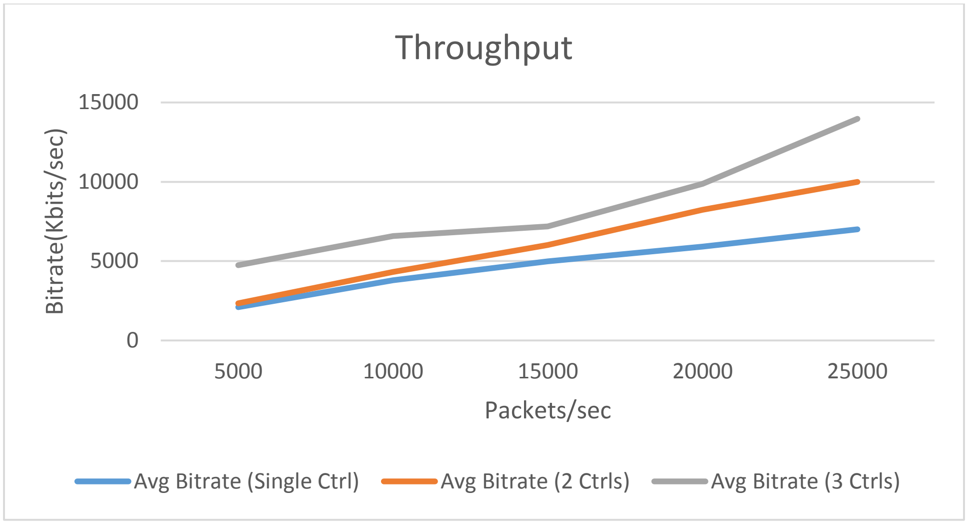


Fig 6.14 Average Bitrate Single v/s Multiple

6.3.5 Average loss burst size

Packets	Average loss-burst size(Single Ctrl)	Average loss-burst size(2 Ctrls)	Average loss-burst size(3 Ctrls)
5000	2212	1395	878
10000	4439.5	2799	1870
15000	6059	3901	2559
20000	7910	4443	3119
25000	9100	7003	4509

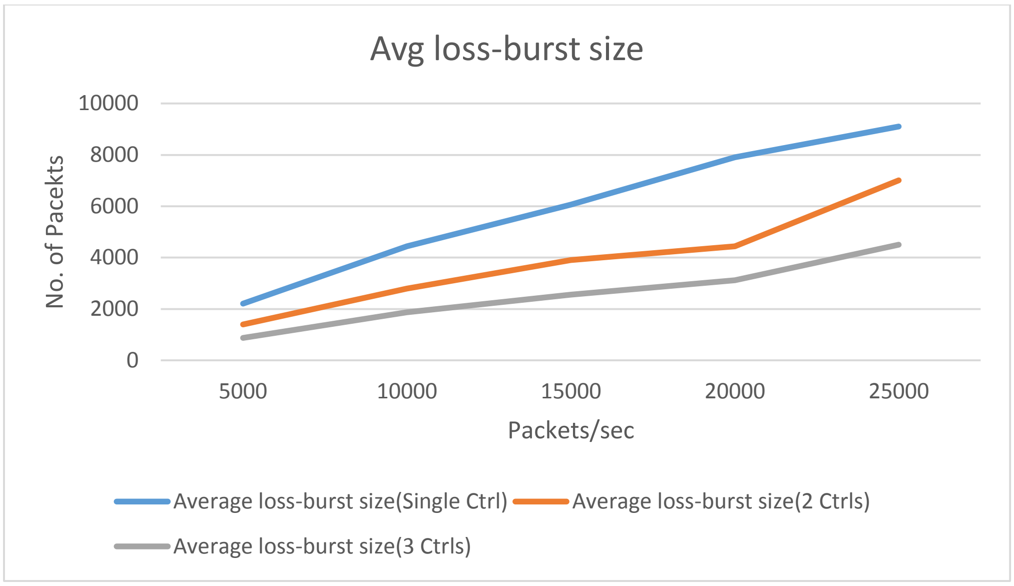


Fig 6.15 Average Loss-Burst Size Single v/s Multiple

Chapter 7 Conclusion and Future Work

7.1 Conclusion

The Software Defined Networks are widely propagated and becoming the most promising technology. As the demand is increasing it's important to take into consideration the challenges. As we observed in the thesis that as the network grows the load on the controller increases. Thus, the single controller SDN architecture becomes incompetent. By observing the evaluation results we can clearly say the multicontroller architecture better handles the network traffic as compared to the single controller as the load is divided amongst the two controllers.

With the help of the results of the experiment, we can understand the importance of having more than one controller to avoid issues presented in the hypothesis. As the delay increases the packet drop ratio increases. With the single controller architecture, the packet loss ratio is high making the system unreliable. As we move to two controller architecture the packet loss ratio is reduced and total transmission time is better thereby making it more reliable.

Also, we observed that the throughput, transmission delay, and packet loss ratio is much better for multicontroller SDN architecture. We compared the results of Single and multicontroller architecture and understood that to deal with challenges of the wider network it's intelligent to move to multicontroller SDN architecture.

7.2 Future Work

We can balance the load several ways in POX multicontroller architecture.

We can implement neural network method between controllers to automate the load balancing based on the incoming packets and requests. We can train the networks based on how the inputs are coming such as the switches and controller communication will be automated to find the best shortest path.

We can also use POX inbuilt messenger library to build a python program to apply different algorithms to establish communication between controllers in order to do the load balancing. The POX controller is python based and easy to program. We can also develop socket programming to develop communication between controllers.

Bibliography:

[1] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: a comprehensive survey," Proceedings of the IEEE, vol. 103, no. 1, pp. 14–76, 2015.

[2] W. Xia, Y. Wen, C. Heng Foh, D. Niyato, and H. Xie, "A survey on software-defined networking," IEEE Communications Surveys and Tutorials, vol. 17, no. 1, pp. 27–51, 2015.

[3] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: past, present, and future of programmable networks," IEEE Communications Surveys & Tutorials, vol. 16, no. 3, pp. 1617–1634, 2014.

[4] J. Xie, D. Guo, Z. Hu, T. Qu, and P. Lv, "Control plane of software defined networks: a survey," Computer Communications, vol. 67, pp. 1–10, 2015

[5] Open Networking Foundation (ONF), https://www.opennetworking.org/.

[6] M. Smith, M. Dvorkin, Y. Laribi, V. Pandey, P. Garg, and N. Weidenbacher, "OpFlex Control Protocol," Internet Draft, Internet Engineering Task Force, 2014, http://tools.ietf.org/html/draft-smith-opflex-00.

[7] A. Doria, J. H. Salim, R. Haas et al., Forwarding and Control Element Separation (ForCES) Protocol Specification, Internet Engineering Task Force, 2010, http://www.ietf.org/rfc/rfc5810.txt.

[8] M. Sune, V. Alvarez, T. Jungel, U. Toseef, and K. Pentikousis, "An OpenFlow implementation for network processors," Defined Netw, p. 2, 2014.

[9] Inter-SDN Controller Communication: Using Border Gateway Protocol http://www.tcs.com/resources/white_papers/Pages/InterSDN-Controller-Communication.aspx

[10] D. Erickson, "The Beacon OpenFlow controller," in Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13), pp. 13–18, ACM, New York, NY, USA, 2013.

[11] K. Phemius, M. Bouet, and J. Leguay, "DISCO: distributed multi-domain SDN controllers," in Proceedings of the IEEE Network Operations and Management Symposium (NOMS '14), pp. 1–4, Kraków, Poland, May 2014.

[12] Y. Fu, J. Bi, K. Gao, Z. Chen, J. Wu, and B. Hao, "Orion: a hybrid hierarchical control plane of software-defined networking for large-scale networks," in Proceedings of the 22nd IEEE International Conference on Network Protocols (ICNP '14), pp. 569–576, Raleigh, NC, USA, October 2014. View at Publisher.

[13] Mininet Documentation " mininet.org/sample-workflow/

[14] POX support documentation https://openflow.stanford.edu/display/ONL/POX+Wiki

[15] (rao, 2016) http://thenewstack.io/multiple-sdn-controllers/

[16] AMQP, an advanced message queuing protocol that can be found on https://www.amqp.org/.

[17] A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. Kompella, "Towards an elastic distributed SDN controller," in Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13), pp. 7–12, ACM, Hong Kong, 2013.

[18] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX '10), article 19, ACM, 2010.

[19] A. Tootoonchian and Y. Ganjali, "HyperFlow: a distributed control plane for OpenFlow," in Proceedings of the Internet Network Management Conference on Research on Enterprise Networking (INM/WREN '10), Berkeley, Calif, USA, 2010.

[20] J. Stribling, Y. Sovran, I. Zhang et al., “Flexible, wide-area storage for distributed systems with wheelfs,” in Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09), Boston, Mass, USA, April 2009.

[21]U. Krishnaswamy, P. Berde, J. Hart et al., “ONOS: an open source distributed SDN OS,” 2013, http://www.slideshare.net/umeshkrishnaswamy/open-network-operating-system.

[22] Titan Distributed Graph Database, http://thinkaurelius.github.io/titan/.

[23] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” ACM SIGOPS Operating Systems Review, vol. 44, no. 2, 2010. View at Google Scholar

[24] Tinkerpop. Blueprints, http://blueprints.tinkerpop.com/.

[25] J. Ousterhout, M. Rosenblum, S. M. Rumble et al., “The case for RAMClouds: scalable high-performance storage entirely in DRAM,” ACM SIGOPS Operating Systems Review, vol. 43, no. 4, pp. 92–105, 2010.

[26] Hazelcast Project, http://www.hazelcast.org/.

[27] K. Phemius, M. Bouet, and J. Leguay, “DISCO: distributed multi-domain SDN controllers,” in Proceedings of the IEEE Network Operations and Management Symposium (NOMS '14), pp. 1–4, Kraków, Poland, May 2014.

[28] AMQP, an advanced message queuing protocol that can be found on https://www.amqp.org/.

[29] Deepankar Gupta, Rafat Jahan, R&D Lead(SDN), Tata Consultancy Services, “Inter SDN Controller Communication (SDNi),” in Proceedings of the Opendaylight Summit 2015.

[29] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: rapid prototyping for software-defined networks,” in Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX '10), article 19, ACM, 2010.