Wireless Pers Commun (2018) 98:1679–1699
https://doi.org/10.1007/s11277-017-4939-z

Performance Analysis of SDN/OpenFlow Controllers: POX Versus Floodlight

Idris Z. Bholebawa[1] · **Upena D. Dalal**[1]

Published online: 30 August 2017

Abstract Software-Defined Networking (SDN) is an emerging network architecture that is adaptable, dynamic, cost-effective, and manageable. The SDN architecture is a form of network virtualization where the network controlling functions and forwarding functions are decoupled. A setup and configuration task of a control plane to work as an SDN controller is explained in this paper. This paper includes a brief survey of different SDN based OpenFlow-enabled controllers available in various programmable languages. This paper mainly focuses on two OpenFlow-enabled controllers, namely, POX—a Python-based controller and Floodlight—a Java-based controller. A performance comparison of both controllers is tested over different network topologies by analyzing network throughput and round-trip delay using an efficient network simulator called Mininet. A single, linear, tree and custom (user-defined) topologies are designed in Mininet by enabling external controllers. It is obtained that, a percentage improvement in round-trip time for Floodlight over POX is 11.5, 13.9, 19.6 and 14.4% for single, linear, tree and custom topology respectively. Similarly, a percentage improvement in throughput for Floodlight over POX is 5.4, 8.9, 3.8 and 4.9% for single, linear, tree and custom topology respectively.

Keywords SDN · OpenFlow · POX · Floodlight · Mininet

✉ Idris Z. Bholebawa
idris.bholebawa@gmail.com;
http://www.svnit.ac.in

Upena D. Dalal
udd@eced.svnit.ac.in;
http://www.svnit.ac.in

[1] Department of Electronics and Communication Engineering, S. V. National Institute of Technology, Surat, Gujarat 395007, India

1 Introduction

A well-developed infrastructural traditional network architecture is currently a successful contender for networking domain. However, network engineers are having limited access for configuring new network policies by transforming the high-level complex firmware into low-level configuration commands with manual effort. Due to this limitation, the Internet has become extremely difficult to evolve in terms of advance protocols and efficient performance, thus eventually restricts an evolution of physical infrastructure. This difficulty will lead to a confinement of network management and controlling capabilities and a necessary changes done manually could sometime result in an error-prone task. Researchers and network engineers refer this difficulty as 'Internet ossification' [1, 2]. Thus the current situation of the Internet is restricted to evolve to address emerging technological trends in networking.

In order to facilitate new networking evolution, a concept of programmable networks [3] has been proposed. A programmable network organizing technique endures recent prominence is Software-Defined Networks (SDN) [4]. A comprehensive survey on SDN architecture for Next Generation Networks (NGN) and its historical evolution is given in [5]. SDN is having three significant characteristics: first, it is having an ability to separate the data and control functions of core networking devices, with a well-defined Application Programming Interface (API) (generally referred as *Southbound APIs* [6]). Second, SDN provides a unified control plane function, so that multiple data plane elements can be easily controllable via a unique software program (generally referred as *Network Operating System* (*NOS*) [7]). Third, SDN allows centrally controlling capabilities, this architecture provides network engineers or network administrators a global view of a whole underlying physical network and allows to make changes globally. A well-defined open APIs (generally referred as *Northbound APIs* [6]) between control plane and applications running over the network is used in order to facilitate innovations and to enable efficient network control.

In this paper a performance of mainly two SDN controllers/NOS is analyzed. We begin by introducing SDN layered network architecture in Sect. 2 and discussing every component with a brief literature survey. In a Sect. 3, an introduction and a brief performance survey of effectively used SDN controllers are discussed. A contributed work has been carried out in Sect. 4, an individual performance of two effectively used SDN controllers/NOS over defined network topologies are analyzed with the help of an open source simulation tool, Mininet. The two controllers are: POX—a Python-based open source OpenFlow controller and the Floodlight—a Java-based open source OpenFlow controller. A performance comparison between these two SDN controllers are done by comparing throughput and round-trip time between end-user nodes. With the help of Mininet command-line interface (CLI), several topologies are proposed and performance of above mentioned OpenFlow controllers are compared and analyzed. A discussion of the results obtained after execution of proposed network topologies under the influence of POX and a Floodlight is inclusively deliberated in Sect. 5. Lastly, Sect. 6 delivers an overall conclusion on the basis of result analysis and briefly deliberate ideas related to future perspectives.

2 Software-Defined Network Architecture

SDN refers as a network architecture where data forwarding plane is controlled by a remote control plane, both planes were previously working as a single unit. With this separation, innovative ideas have been proposed and an infrastructure network architecture

gets a new direction of network evolution. A functional three layered SDN architecture and their available components are shown in Fig. 1.

The SDN architecture mainly comprises of three layers. The lowest layer is an infrastructure layer, also called a data plane, it consists of hardware networking devices which are interconnected with each other for exchanging information, and it is solely responsible for forwarding data packets regardless of network architecture. Data plane generally consists data forwarding elements, such as Ethernet switches, packet switches, routers, etc., these elements are taking active part in data packet forwarding. The middle layer is a controller layer, also called a control plane, this plane is separated from network core devices and is responsible for unified controlling action. A single SDN controller is responsible for controlling entire data flows of an underlying infrastructure network centrally. SDN controller act as an operating system for the network (NOS). A topmost layer is an application layer, also called management plane, this layer is responsible for management and data forwarding related tasks (e.g. data traffic monitoring, mobility management, routing, security, load balancer, etc.), for efficient data traffic flow. In SDN architecture, southbound APIs are used as a communication protocol between control plane (i.e. SDN controller) and data plane (i.e. switches and routers) of the network. These southbound APIs can be an open source or proprietary. Various proposals for southbound interface are practically available, such as OpenFlow [8], ForCES [9], OVSDB [10], POF [11], OpFlex [12], OpenState [13], etc. Out of these proposals, an efficient and most popular currently deployed southbound API standardizing the communication protocol

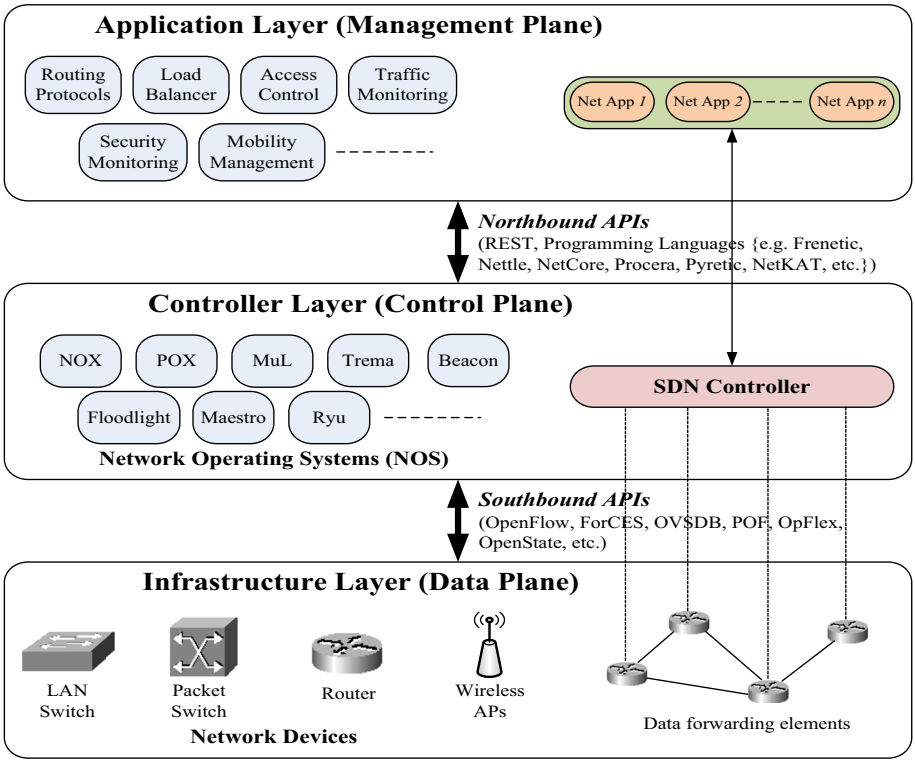

Fig. 1 SDN functional architecture and its components

between switches and software-based controller in SDN architecture is OpenFlow [14]. The northbound APIs in SDN architecture are used to communicate between the SDN controller and applications running over the network. These upper layer uses northbound interfaces such as *RESTful* APIs [15] and programming languages such as Frenetic [16], Nettle [17], NetCore [18], Procera [19], Pyretic [20], NetKAT [21], etc.

The SDN architecture currently using an OpenFlow Protocols (OFP), as discussed previously, to communicate between SDN controller (now called OpenFlow controller) and OpenFlow-enabled switches over a secure channel. A first version of OFP was version 0.2.0 released in March, 2008. In December, 2009, the most widely deployed version was released i.e. OpenFlow version 1.0.0 [22]. A currently operational version of OFP which was released in October, 2013 is OpenFlow version 1.4.0 [23]. Using these protocols an OpenFlow-enabled switch sends information to OpenFlow controller and controller controls the data flows of an underlying network. Most commercial switch vendors now include support of OpenFlow APIs as southbound interface. Many giant networking companies formed a group called Open Networking Foundation (ONF) [24], mainly for the purpose of promoting and adopting SDN through open standards development. Any data packet arrives at a switch, the switch will undergo several processes in order to forward a packet by matching flow table entries. A flow diagram of data packet processing in flow-tables of a switch is explained in [25]. Also, an optimal solution for scheduling multiple data flows are discussed in [26]. Moreover, an analytical model for switch table occupancy done by the SDN controller is nicely described in [27]. OpenFlow is an open source technology in which experimental and production traffic can share the same OpenFlow switch. The action set performed by the switch after matching the arrived data flows with the flow entries present in a flow table (done by a controller), can allow to send data packet to the switch data path and the different flow entries can be used for experimental purpose. This way experimental traffic can be tested without interfering with production traffic [8]. Hence, as long as the OpenFlow-enabled switch communicates with an OpenFlow controller, there are a variety of possibilities for company vendors to implement a data plane in diverse ways.

3 Software-Defined Network Controllers/Network Operating Systems

The SDN controllers in a software-defined network is acting as a ‘brain’ of the network. It serves as a sort of operating system for the network. It is an intermediate layer of SDN architecture as shown in Fig. 1, providing a unified network controlling scheme. It is a strategic control point which relays control information to the underlying switches or routers via southbound APIs and deals with networking applications or business application via northbound APIs. In SDN architecture, an SDN controller (also called OpenFlow controller) uses the OFP to configure underlying core network devices and choose the best path for forwarding data traffic. Since, the control plane is usually a centrally controllable software program, network traffic can be managed dynamically. Various open source SDN controller program (NOS) are currently being used for deploying architecture, these controllers are NOX [28], POX [29], Floodlight [30], MuL [31], Trema [32], Beacon [33], Maestro [34], Ryu [35], etc. A brief description of these open source OpenFlow controllers are listed in Table 1.

In a well deployed large scale production environment, an availability of SDN controller is of major concern for scalability, reliability and for other performances. Thus,

placement of SDN controller in a large scaled network should be well defined, the issues and their solutions are described in [36]. In this section, two effectively used open source OpenFlow controllers, namely POX and Floodlight, are introduced. A feature survey, an installation guidelines and overall functioning of both controllers in SDN architecture is discussed in brief. The OpenFlow network architecture is implemented using a prototype simulator and emulator called Mininet [37, 38]. A performance and the working of both OpenFlow controllers in an OpenFlow network is analyzed using the same Mininet simulator.

3.1 POX Controller

POX is an open source OpenFlow controller based on Python language. It is in general a rapid developing and prototyping network controller software. A POX controller can mostly run on Python *v*2.7, and can officially support Windows, Mac OS and Linux as described in [29]. Moreover, a POX controller is capable for converting unusual OpenFlow devices to operate as switches, routers, load balancers, firewall devices, etc., and provides better performance to SDN architecture. Some of the salient features of a POX controller described in [39], are listed as follows:

- It can provide a Pythonic OpenFlow interface.
- It is having reusable sample components for path selection, topology discovery, etc.
- It is capable to run in any operating system environment and comes preinstalled with a Mininet simulator.
- It can support the same Graphical User Interface (GUI) and virtual architecture similar to NOX.
- It can provide better performance as compared to NOX written in Python.

A POX controller code can be easily downloaded from the POX repository on '*github*' [40]. Syntax for command-line interface (CLI) as well as different options and components required for operating a POX controller in diverse ways is described in [41]. Here POX controller version 0.2.0 is available by default with a POX installation package. An activated POX controller will then ready to connect with underlying data forwarding elements remotely in a Mininet simulator. A connecting steps of a POX controller with underlying network components and its performance analysis is discussed in Sect. 4.

Table 1 List of available OpenFlow controllers

Controller name	Programming language	License provider
NOX [28]	C++	GPL
POX [29]	Python	Apache
Floodlight [30]	Java	Apache
MuL [31]	C	GPLv2
Trema [32]	C, Ruby	GPLv2
Beacon [33]	Java	GPLv2
Maestro [34]	Java	LGPL
Ryu [35]	Python	Apache

3.2 Floodlight Controller

The Floodlight controller is an open source OpenFlow controller based on Java language. It is Apache-licensed and is supported by a largest community of developers for SDN controllers which includes a number of network researchers and engineers from BigSwitch Networks [42]. The Floodlight controller is designed as heavily concurrent systems for attaining the throughput required by multiple data centers and enterprise class networks. Since, OpenFlow specifies protocols through an OpenFlow-enabled switch, a controller activated remotely can modify the behavior of core networking devices through a well-defined data forwarding instruction set. The Floodlight controller is designed to work with different switches, routers, access points, etc. that supports OpenFlow standards. The Floodlight controller could also support hybrid networks where OpenFlow-enabled switches connected through an existing traditional switches. Some of the salient features of the Floodlight controller highlighted in [43] are listed as follows:

- It offers a module loading system that makes it simple to extend and enhance.
- It is very easy to set up with minimal dependencies.
- It supports a broad range of physical and virtual OpenFlow-enabled switches.
- It can also handle a network of hybrid environment, which in turn can manage multiple segments of OpenFlow hardware switches.
- It is design to provide high-performance of a network, which is a core requirement of any commercial product from Big Switch Networks.

Floodlight is not only an OpenFlow controller, but also a collection of applications which are available on top of the controller. An internal architecture of the Floodlight controller and the applications available as Java module is shown in Fig. 2.

A detailed description of each and individual blocks shown in Fig. 2 is given in [44]. Installation of the Floodlight controller is extremely easy in Linux machines that is described in [43]. The Floodlight controller is activated on an external port having IP address of 127.0.0.1 (loopback address). Since, the Floodlight controller can support a web-based GUI interface, following URL (Uniform Resource Locator) is required to enter in an address bar of web browser:

"http://www.localhost:8080/ui/index.html"

Here, localhost can be replaced by any logical address on which controller is made active. Moreover, GUI applet (Java Application) is accessed on web browser by using *http* protocol, a port address 8080 is used to host a secondary web server, other than an authorized server administrator system running on port 80, which is also available in URL. The Floodlight controller is then completely ready to connect with OpenFlow-enabled switches forming infrastructure network architecture. Detailed steps of interfacing a controller with switches are discussed in the next section.

4 Performance Analysis of OpenFlow Controllers

In this section, a performance analysis of an OpenFlow controllers, discussed in the previous section, in an OpenFlow-enabled network topologies are scrutinized. A performance of above defined two controllers, POX—a Python-based controller and Floodlight—a Java-based controller, are analyzed by comparing their behavior in an OpenFlow-enabled network topologies. The OpenFlow-enabled network topologies are designed using Mininet

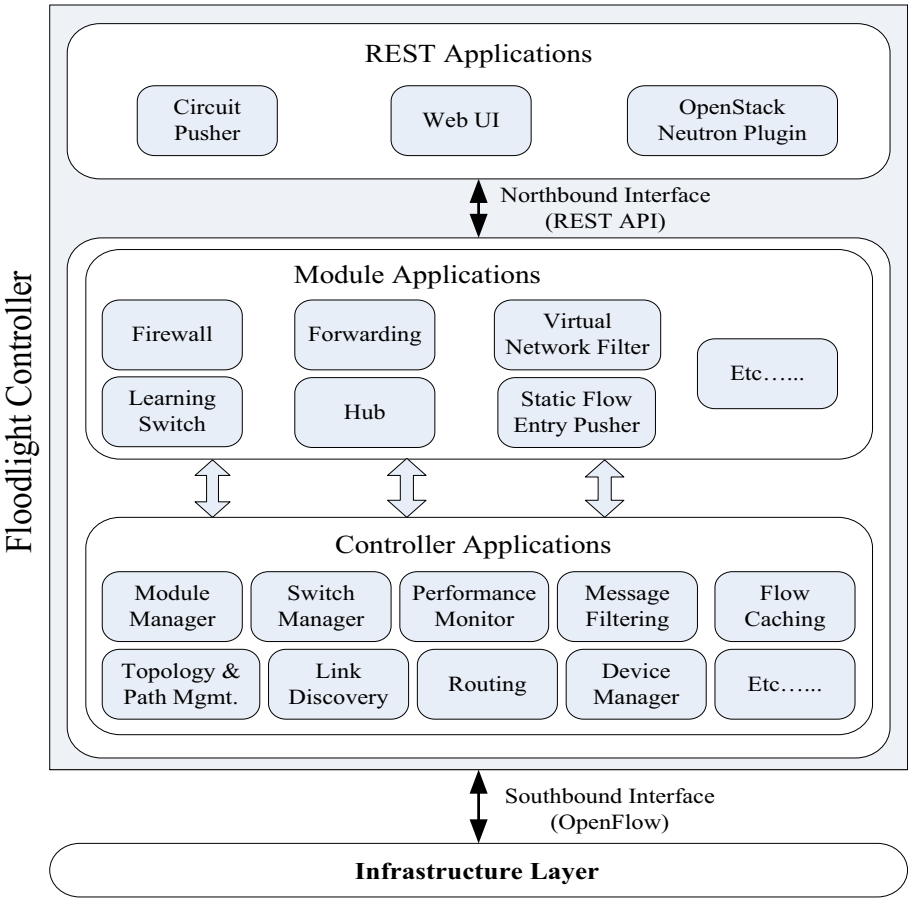

Fig. 2 Internal architecture of the Floodlight controller and its interfaces

simulation tool. Mininet provides several built-in basic network topologies like Minimal topology, Single topology, Linear topology, Tree topology, etc., which are discussed in [45]. An implementation of these basic network topologies are very simple by executing a single-line command in the Mininet console. A remotely activated OpenFlow controller (POX or Floodlight) gets connected with an underlying switch components of network topologies. Mininet also supports an implementation of user-defined custom OpenFlow network topology discussed in [45]. If, for example, an underlying network topology is complex with several intermediate nodes that can't be control via single controller, than load balancing in a network is done by inserting multiple controllers in a network [46, 47]. In this section, a design and implementation of fewer built-in Mininet basic network topologies, i.e. Single topology, Linear topology and Tree topology and a user-defined Custom network topology are proposed. Furthermore, a performance of POX and Floodlight controller over these network topologies are analyzed. All the created network topologies are having physical link parameters of 100 Mbps bandwidth with 1 ms of propagation delay.

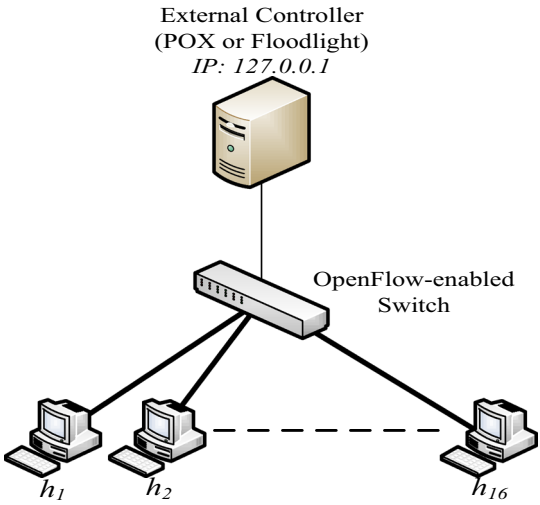

Fig. 3 Single topology having 16 hosts

4.1 Single Topology

A single topology in Mininet is having single OpenFlow-enabled switch, the defined number of hosts acquires connection with this single switch. The switch in turn gets connected with a control plane available on the top. The controllers are activated remotely using their respective commands in Mininet console as discussed in the previous section. A performance of these remotely activated external controllers are now analyzed by connecting it with an underlying single topology network. A single topology having 16 hosts is shown in Fig. 3.

A remotely activated controller (POX or Floodlight) is ready to connect with underlying network switches on an IP address of *127.0.0.1*. If a POX controller is made active, then following a single-line command will create a single topology and connect a POX controller as shown in Fig. 3.

"*$ sudo mn --topo = single,16 --controller remote,ip = 127.0.0.1 --mac*"

A status of a POX controller connecting single OpenFlow-enabled switch of Single topology is shown in Fig. 4.

If, the Floodlight controller is made active, then the same single-line Mininet command is used. Here, the Floodlight controller is available at the same remote IP address (i.e. *127.0.0.1*) instead of a POX controller. Since, the Floodlight controller provides web-based GUI interface, a generated single network topology is easily available on a web browser by entering specific URL discussed in the previous section. A snapshot of single topology in a web-based GUI applet and a status of the Floodlight controller connecting a switch of single network topology are shown in Figs. 5 and 6 respectively.

As discussed previously, a single OpenFlow-enabled switch is present in single network topology. It can be clearly seen in both controller's (POX and Floodlight) domain.

```
idris@ubuntu:~/pox$ ./pox.py samples.pretty_log forwarding.l2_learning
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
[core                  ] POX 0.2.0 (carp) is up.
[openflow.of_01        ] [00-00-00-00-00-01 1] connected
```

Fig. 4 Status of POX controller connecting a switch of single topology

4.2 Linear Topology

A linear topology in Mininet is having linear connection between switches and hosts. Each host connects with its particular switch and the switches are connected with each other linearly. All the OpenFlow-enabled switches in turn gets connected with a remote controller. A detailed explanation of linear topology and its working is explained in [45]. A linear topology having 16 hosts connected with external controllers (POX or Floodlight) is shown in Fig. 7.

Firstly, POX controller is made active on remote IP address (*127.0.0.1*). Following a single-line Mininet command will create linear topology having 16 hosts and attach a POX controller with OpenFlow-enabled switch as shown in Fig. 7.

"*$ sudo mn --topo = linear,16 --controller remote,ip = 127.0.0.1 --mac*"

A status of a POX controller connecting OpenFlow-enabled switch of linear topology is shown in Fig. 8.

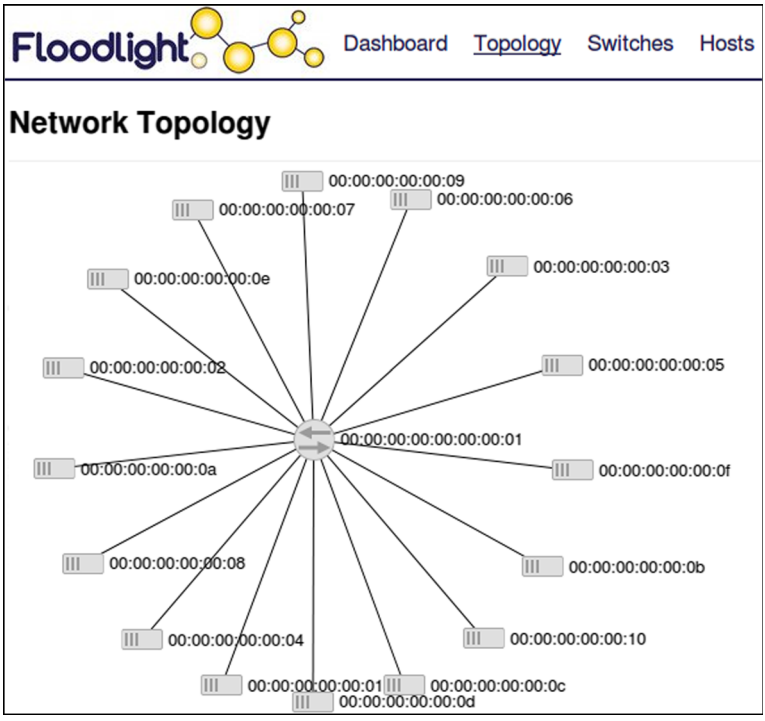

Fig. 5 Single network topology in Floodlight web-based GUI interface

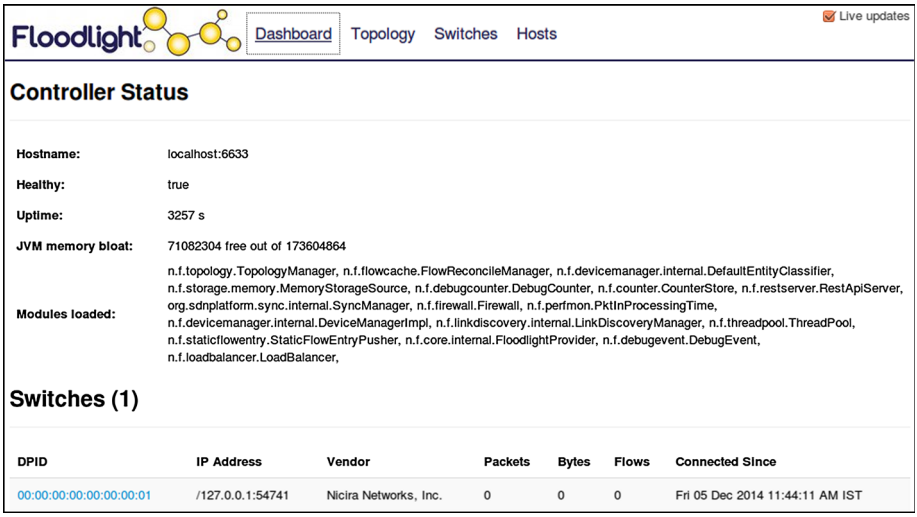

Fig. 6 Controller status in Floodlight web-based interface for single topology

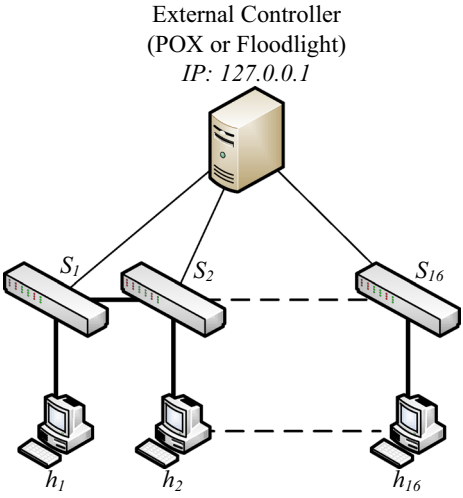

Fig. 7 Linear topology having 16 hosts

Now, the Floodlight controller is made active on the same remote IP address (*127.0.0.1*), the same single-line command as discussed above is executed to create linear topology which connects the Floodlight controller instead of a POX controller. A generated network topology in web-based Java applet is shown in Fig. 9 and a status of the Floodlight controller for connecting OpenFlow-enabled switches of linear topology in the same web-based GUI interface is shown in Fig. 10.

As discussed previously, a linear topology consists of '*n*' OpenFlow-enabled switches for connecting '*n*' hosts in a network. A controller status for both controllers is clearly portraying the architecture of linear network topology of Mininet having 16 hosts.

```
idris@ubuntu:~/pox$ ./pox.py samples.pretty_log forwarding.l2_learning
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
[core                  ] POX 0.2.0 (carp) is up.
[openflow.of_01        ] [00-00-00-00-00-10 1] connected
[openflow.of_01        ] [00-00-00-00-00-0b 4] connected
[openflow.of_01        ] [00-00-00-00-00-04 2] connected
[openflow.of_01        ] [00-00-00-00-00-02 3] connected
[openflow.of_01        ] [00-00-00-00-00-09 6] connected
[openflow.of_01        ] [00-00-00-00-00-0a 5] connected
[openflow.of_01        ] [00-00-00-00-00-01 7] connected
[openflow.of_01        ] [00-00-00-00-00-0f 8] connected
[openflow.of_01        ] [00-00-00-00-00-0e 9] connected
[openflow.of_01        ] [00-00-00-00-00-08 10] connected
[openflow.of_01        ] [00-00-00-00-00-07 13] connected
[openflow.of_01        ] [00-00-00-00-00-06 11] connected
[openflow.of_01        ] [00-00-00-00-00-05 12] connected
[openflow.of_01        ] [00-00-00-00-00-0d 14] connected
[openflow.of_01        ] [00-00-00-00-00-0c 15] connected
[openflow.of_01        ] [00-00-00-00-00-03 16] connected
```

Fig. 8 Status of POX controller connecting switches of linear topology

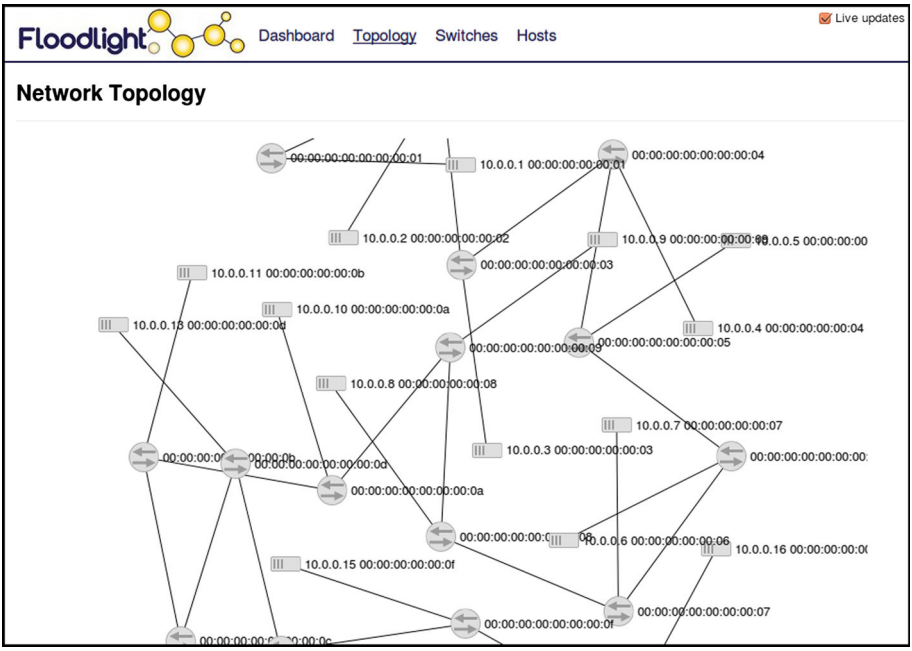

Fig. 9 Linear network topology in Floodlight web-based GUI interface

4.3 Tree Topology

In a tree network topology, all the OpenFlow-enabled switches and hosts are connected with each other in a hierarchical fashion. A detail architecture and working of tree topology is discussed in [45]. A tree topology having 16 hosts is shown in Fig. 11.

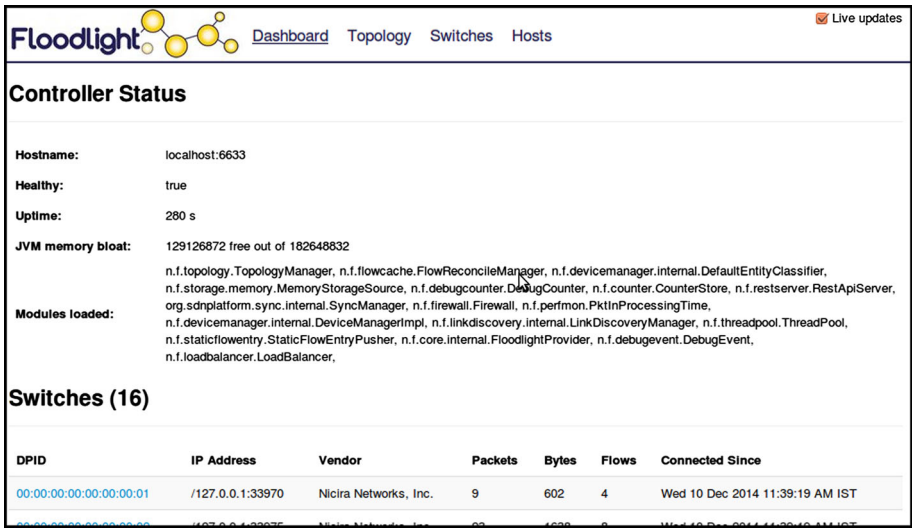

Fig. 10 Controller status in Floodlight web-based interface for linear topology

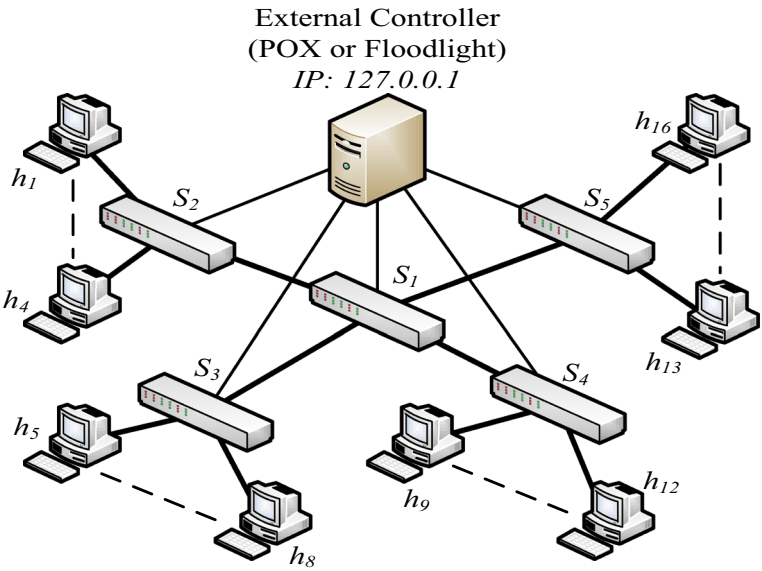

Fig. 11 Tree topology having 16 hosts

After activating a POX controller on the same remote IP address (127.0.0.1) as discussed above, a single-line Mininet command is executed for generating tree network topology which connects a POX controller on assigned IP address. A command for generating tree topology is given below:

"*$ sudo mn --topo = tree,depth = 2,fanout = 4 --controller remote,ip = 127.0.0.1 --mac*"

On execution of above command, tree topology is generated in Mininet console and a POX controller activated remotely gets connected with OpenFlow-enabled switches available in underlying network topology. A status of POX controller connecting switches is shown in Fig. 12.

Next, a Floodlight controller is made active with the same remote IP address. The same single-line Mininet command as above is required to execute for creating tree topology. The only difference is the Floodlight controller is available at remote IP address (*i.e. 127.0.0.1*) instead of a POX controller as discussed in previous case. A graphical connection of tree network topology in hierarchical fashion is shown in Fig. 13 and a status of the Floodlight controller connecting the OpenFlow-enabled switches of tree network topology in web-based interface is shown in Fig. 14.

A tree topology having 16 hosts requires 5 OpenFlow-enabled switches connected in hierarchical fashion. A status of both controllers (POX and Floodlight) connected with underlying 5 switches is clearly shown above in their respective figures (Figs. 12, 14).

4.4 Proposed (Custom) Network Topology

A custom network topology having 4 switches and 32 hosts is proposed using Python coding script. A network is designed such that each switch is having an equal number of hosts connected (i.e. 8 hosts for each switch) and switches are connected with each other. All OpenFlow-enabled switches in-turn gets connected with an external controller. A designing steps and flow diagram of a custom network topology is described in [25]. A proposed (custom) topology is shown in Fig. 15.

Again, if a POX controller is made active on remote IP address (127.0.0.1), a following single-line command will create a complete custom OpenFlow-enabled network by connecting switches with a remote controller:

"*sudo mn --custom 32hosts-4switches.py --topo mytopo --controller remote,ip = 127.0.0.1 --mac*"

A status of a POX controller connecting the switches of an underlying custom network topology is shown in Fig. 16.

If, the Floodlight controller is made active instead of a POX controller on the same IP address (i.e. 127.0.0.1), then the above same command is required to execute for creating a proposed custom network having Floodlight controller as a remote controller. A graphical connection of nodes in a custom network topology displayed on web browser is shown in Fig. 17. A status of the Floodlight controller connecting OpenFlow-enabled switches in an underlying custom network is shown in Fig. 18.

```
idris@ubuntu:~/pox$ ./pox.py samples.pretty_log forwarding.l2_learning
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
[core                  ] POX 0.2.0 (carp) is up.
[openflow.of_01        ] [00-00-00-00-00-04 1] connected
[openflow.of_01        ] [00-00-00-00-00-01 2] connected
[openflow.of_01        ] [00-00-00-00-00-02 3] connected
[openflow.of_01        ] [00-00-00-00-00-05 4] connected
[openflow.of_01        ] [00-00-00-00-00-03 5] connected
```

Fig. 12 Status of POX controller connecting switches of tree topology

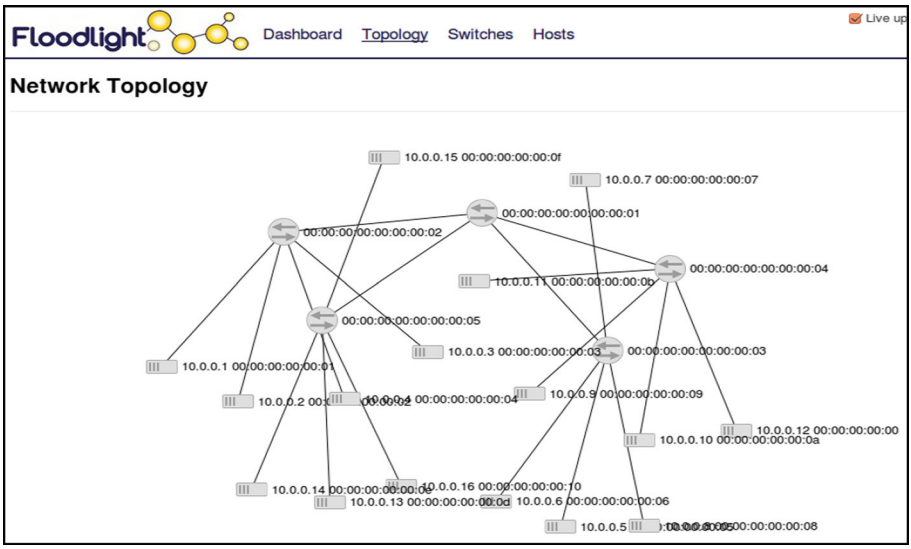

Fig. 13 Tree network topology in Floodlight web-based GUI interface

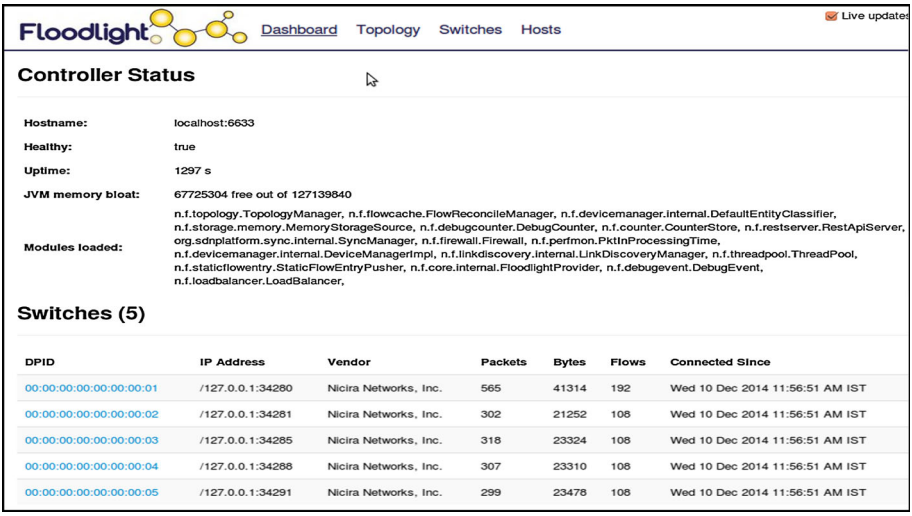

Fig. 14 Controller status in Floodlight web-based interface for tree topology

5 Result Analysis and Discussion

A performance comparison of OpenFlow-enabled controllers are done by analyzing round-trip delay between end hosts and by analyzing maximum obtained throughput between the end hosts. A round-trip time between hosts can be obtained by executing an ICMP query message using '*ping*' command (Echo request and reply message) for a connectivity test. A network overall performance is achieved from available throughput. A throughput is

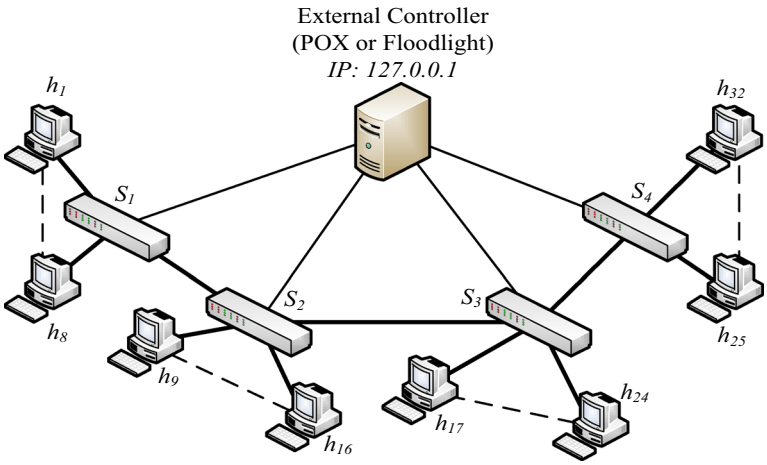

Fig. 15 Custom network topology

```
idris@ubuntu:~/pox$ ./pox.py samples.pretty_log forwarding.l2_learning
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
[core                  ] POX 0.2.0 (carp) is up.
[openflow.of_01        ] [00-00-00-00-00-04 1] connected
[openflow.of_01        ] [00-00-00-00-00-01 3] connected
[openflow.of_01        ] [00-00-00-00-00-02 2] connected
[openflow.of_01        ] [00-00-00-00-00-03 4] connected
```

Fig. 16 Status of POX controller connecting the switches of custom topology

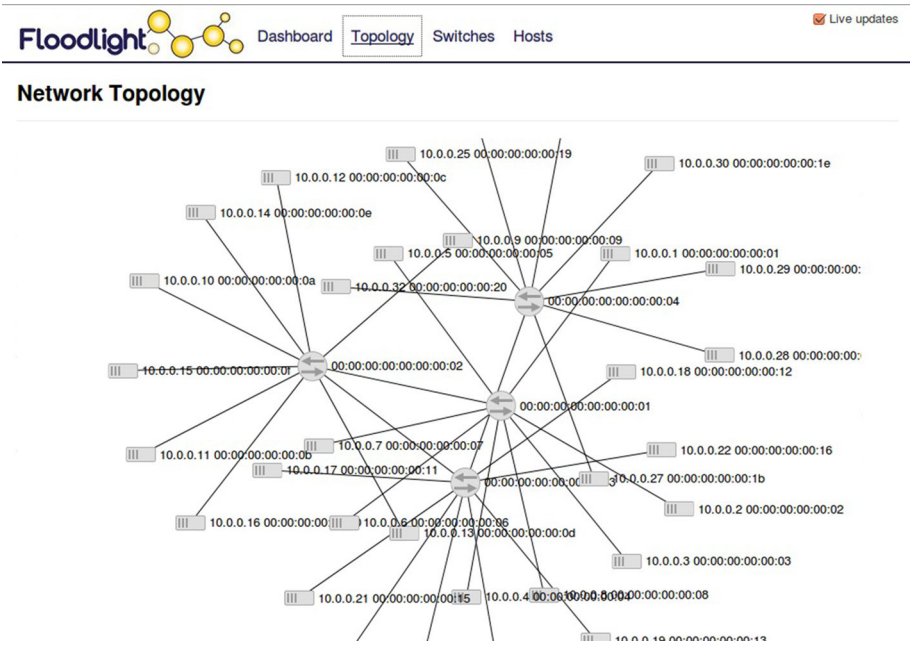

Fig. 17 Custom network topology in Floodlight web-based GUI interface

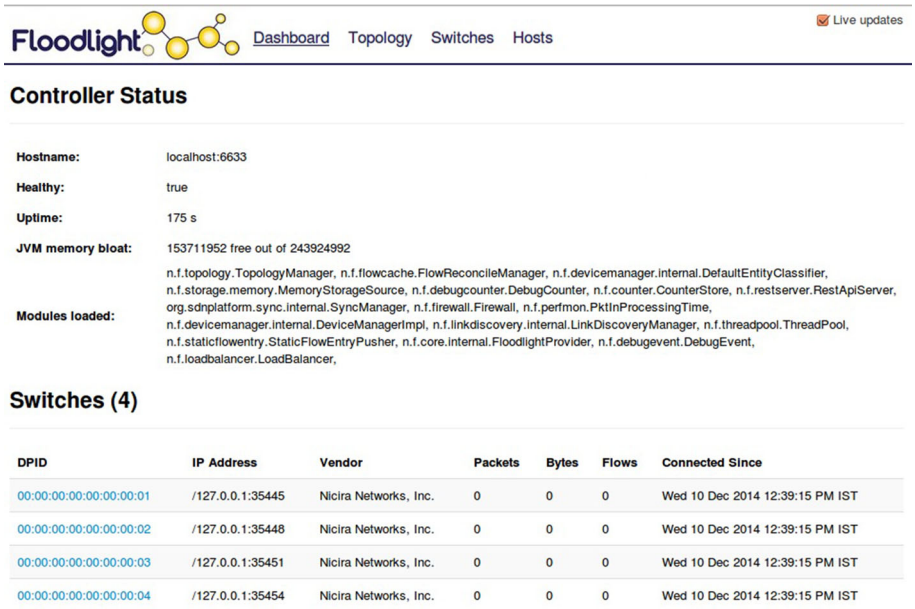

Fig. 18 Controller status of Floodlight for custom network topology

Table 2 Round-trip time (RTT) between hosts

Average round trip time (RTT) between host h_1 and h_{16}		
	POX controller (ms)	Floodlight controller (ms)
Single topology	5.3	4.69
Linear topology	43.2	37.2
Tree topology	10.73	8.63
Custom topology	12.62	10.8

Round-trip time for custom network topology is obtained between hosts h_1 and h_{32}

defined as the amount of data delivered in a given period of time. It could be obtained using the following given formula:

"*max throughput = max rcv. BW/rtt*"

A result analysis of network topologies discussed previously and performance of both controllers in the topologies are discussed below.

5.1 Round-Trip Time (*RTT*):

A performance comparison between POX and Floodlight controller in previously defined network topologies are achieved by execution of ICMP connectivity test using '*ping*' command (Echo request and reply message). A ping test is performed between end hosts h_1 and h_{16} of given network topologies for calculating round trip time (*RTT*) between hosts. A result of execution in both POX and Floodlight controller domain is tabulated in Table 2 and is shown graphically in Fig. 19.

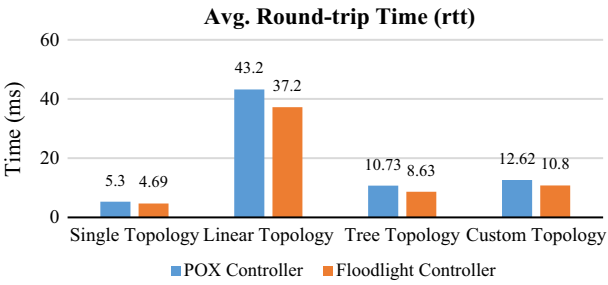

Fig. 19 Round-trip time between end hosts

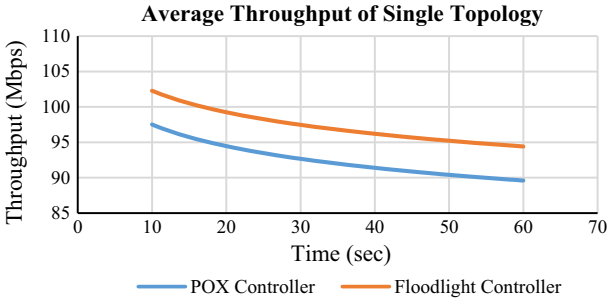

Fig. 20 Average throughput of single topology

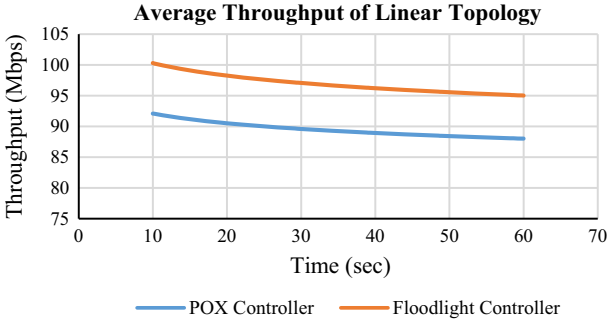

Fig. 21 Average throughput of linear topology

According to result obtained, an OpenFlow network having Floodlight controller in its control plane is faster responsive as compared to a network having POX controller. A delay between end nodes in an OpenFlow-enabled network is less with Floodlight controller as compared to POX controller. According to Table 2, the percentage improvement of round-trip time in a Floodlight controller as compared to the POX controller is 11.5, 13.9, 19.6 and 14.4% for single, linear, tree and custom topology respectively.

5.2 Throughput

An overall network performance with different control plane application is obtained by comparing network throughput. A throughput can be calculated from the above defined formula. An average throughput between end nodes in different network topologies is calculated and is shown in below figures.

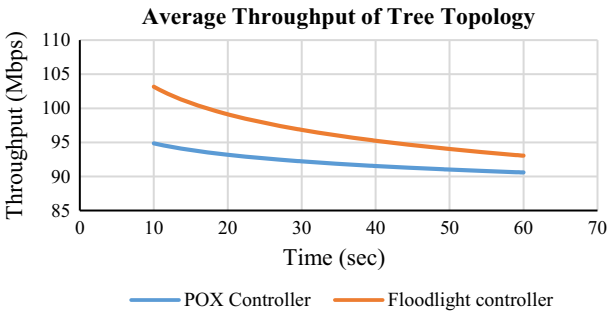

Fig. 22 Average throughput of tree topology

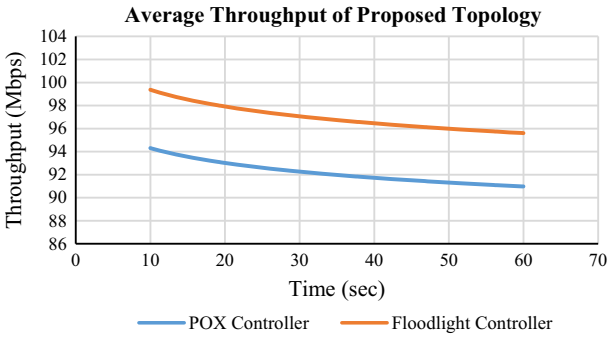

Fig. 23 Average throughput of proposed topology

An average network throughput for different network topologies, as discussed, is obtained and shown graphically in figures (Figs. 20, 21, 22, 23). An overall network performance of an OpenFlow-enabled network is better with the Floodlight controller as compared to a POX controller. A percentage improvement in throughput for Floodlight controller as compared to POX controller over single, linear, tree and custom topology is 5.4, 8.9, 3.8 and 4.9% respectively. Thus, as shown in figures, an overall network throughput obtained between end nodes is more in a network topology having Floodlight controller than having a POX controller.

6 Conclusion

The SDN based on OpenFlow-enabled networks are a promising technology for future Internet and NGN. Several control planes are available and advance innovations is still under progress for deployment of optimized NOS. Centrally controlled SDN architecture having efficient control plane application is in demand for a production environment. Many approaches are considered for an efficient performance of the SDN controller. A Python-based POX controller and a Java-based Floodlight controller is a strong contender for implementing efficient control plane application in OpenFlow-enabled networks. A performance comparison between these two controllers, POX and Floodlight, are analyzed in this paper by considering different Mininet network topologies like single, linear, tree and custom (user-defined) topology.

On the basis of performance comparison between the POX and the Floodlight controller and according to simulation results obtained, it is concluded that the Floodlight controller

provides more efficient performance as compared to the POX controller for OpenFlow-enabled network topologies. By using a Java-based Floodlight controller as a control plane application, an overall network performance is getting improved in terms of round-trip time and throughput. A percentage improvement in RTT in a Floodlight over a POX on single, linear, tree and custom topology is 11.5, 13.9, 19.6 and 14.4% respectively. That is, the Floodlight controller is faster responsive than the POX controller. Moreover, a percentage improvement in throughput of a Floodlight controller over POX controller on single, linear, tree and custom topology is 5.4, 8.9, 3.8 and 4.9% respectively. The Floodlight controller is having REST API, hence it is easier to define necessary applications according to the requirement. Unlike POX controller, the Floodlight controller can also provide a web-based interface, so that an underlying network topologies can be easily visible, configure and analyze in GUI interface. Since the Floodlight controller is based on Java scripting, it is somewhat difficult to learn and design, whereas a POX controller is based on Python scripting is easy to program and implement. If only designing related matter is concern than a POX controller is a better choice and if performance related matter is concern than the Floodlight controller is a better choice. However, network engineers requires better performance of the network with reduced complexity.

References

1. Hakiri, A., Gokhale, A., Berthou, P., & Gayraud, T. (2014). Software-defined networking: Challenges and research opportunities for future internet. *Computer Networks, 75*(A), 453–471. doi:10.1016/j.comnet.2014.10.015.
2. Turner, J., & McKeown, N. (2007). Can Overlay hosting services make IP ossification irrelevant? In *Proceedings PRESTO: Workshop on programmable routers for the extensible service of tomorrow*, Princeton, NJ.
3. Astuto, B. N., Mendonca, M., Nguyen, X. N., Obraczka, K., & Turletti, T. (2014). A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys and Tutorials, 16*(3), 1617–1634.
4. Software-Defined Networking (SDN) Definition—Open networking foundation, open networking foundation (online). https://www.opennetworking.org/sdn-resources/sdndefinition.
5. Kreutz, D., Ramos, F. M. V., Verissimo, P., Rothenberg, C. E., Azodolmolky, S., & Uhlig, S. (2015). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE, 103*(1), 14–76.
6. SDN Resources, SDxCentral, (2015). (online) https://www.sdxcentral.com/resources/sdn/.
7. Lara, A., Kolasani, A., & Ramamurthy, B. (2014). Network innovation using openflow: A survey. *IEEE Communication Surveys and Tutorials, 16*(1), 493–512.
8. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., et al. (2008). OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review, 38*(2), 69–74.
9. Doria, A., Salim, J. H., Haas, R., Khosravi, H., Wang, W., Dong, L., Gopal, R., & Halpern, J. (2010). Forwarding and control element separation (ForCES) protocol specification. Internet Engineering Task Force (online). Available: http://www.ietf.org/rfc/rfc5810.txt.
10. Pfaff, B., & Davie, B. (2013). The Open vSwitch database management protocol, RFC 7047 (Informational), internet engineering task force (online). Available: http://www.ietf.org/rfc/rfc7047.txt.
11. Song, H. (2013). Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on hot topics in software defined networking*. New York, NY, USA: ACM, pp. 127–132.
12. Smith, M., Dvorkin, M., Laribi, Y., Pandey, V., Garg, P., & Weidenbacher, N. (2014). OpFlex control protocol, internet draft, internet engineering task force (online). http://tools.ietf.org/html/draft-smith-opflex-00.
13. Bianchi, G., Bonola, M., Capone, A., & Cascone, C. (2014). OpenState: Programming platform-independent stateful OpenFlow applications inside the switch. *SIGCOMM Computer Communication Review, 44*(2), 44–51.

14. OpenFlow Current Deployment, OpenFlow. (2011). (online). http://archive.openflow.org/wp/current-deployments/.
15. Richardson, L., & Ruby, S. (2008). *RESTful web services*. Sebastopol: O'Reilly Media, Inc.
16. Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., & Walker, D. (2011). Frenetic: A network programming language. *SIGPLAN Notices, 46*(9), 279–291.
17. Voellmy, A., & Hudak, P. (2011). Nettle: Taking the sting out of programming network routers. In *Proceedings of the 13th international conference on practical aspects of declarative languages*, ser. PADL'11. Berlin, Heidelberg: Springer, pp. 235–249.
18. Monsanto, C., Foster, N., Harrison, R., & Walker, D. (2012). A compiler and run-time system for network programming languages. *SIGPLAN Notices, 47*(1), 217–230.
19. Voellmy, A., Kim, H., & Feamster, N. (2012). Procera: A language for highlevel reactive network control. In *Proceedings of the first workshop on hot topics in software defined networks*, ser. HotSDN'12. New York, NY, USA: ACM, pp. 43–48.
20. Monsanto, C., Reich, J., Foster, N., Rexford, J., & Walker, D. (2013). Composing software-defined networks. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, pp. 1–14.
21. Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., et al. (2014). NetKAT: Semantic foundations for networks. *SIGPLAN Notices, 49*(1), 113–126.
22. OpenFlow Switch Specification, Version 1.0.0 (Wire Protocol 0x01), December 2009 (online). http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf.
23. OpenFlow Switch Specification, Version 1.4.0 (Wire Protocol 0x05), October 2013 (online). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.
24. Home—Open Networking Foundation, 2013 (online). https://www.opennetworking.org/index.php?lang=en.
25. Bholebawa, I. Z., Jha, R. K., & Dalal, U. D. (2016). Performance analysis of proposed openflow-based network architecture using mininet. *Wireless Personal Communication, 86*(2), 943–958. doi:10.1007/s11277-015-2963-4.
26. Liu, Y., Li, Y., Wang, Y., & Yuan, J. (2015). Optimal scheduling for multi-flow update in software-defined networks. *Journal of Network and Computer Applications, 54*(C), 11–19.
27. Metter, C., Seufert, M., Wamser, F., Zinner T., & Tran-Gia, P. (2017). Analytic model for SDN controller traffic and switch table occupancy. In *IEEE 12th international conference on network and service management (CNSM)*, pp. 109–117. doi: 10.1109/CNSM.2016.7818406.
28. Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., & Shenker, S. (2008). NOX: Towards an operating system for networks. *ACM SIGCOMM Computer Communication Review, 38*(3), 105–110.
29. McCauley, M. (2009). POX (online). http://www.noxrepo.org/.
30. Floodlight is a Java-based OpenFlow controller. (2012). (online). http://floodlight.openflowhub.org/.
31. Saikia, D. (2013). MuL OpenFlow controller (online). http://sourceforge.net/projects/mul/.
32. Takamiya, Y., & Karanatsios, N. (2012). Trema OpenFlow controller framework (online). https://github.com/trema/trema.
33. Erickson, D. (2013). The Beacon OpenFlow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, ser. HotSDN'13. New York, NY, USA: ACM, pp. 13–18.
34. Cai, Z., Cox, A. L. & Ng, T. S. E. (2011). *Maestro: A system for scalable openflow control*, Rice University, Tech. Rep.
35. Nippon Telegraph and Telephone Corporation. (2012). Ryu network operating system (online). http://osrg.github.com/ryu/.
36. Naning, H. S., Munadi R., & Effendy, M. Z. (2017). SDN controller placement design: For large scale production network. In *IEEE Asia Pacific conference on wireless and mobile (APWiMob)*, pp. 74–79. doi:10.1109/APWiMob.2016.7811452.
37. Lantz, B., Heller B., & McKeown, N. (2010). A network in a laptop: Rapid prototyping for software-defined networks. In *Hotnets-IX proceedings of the 9th ACM SIGCOMM workshop on hot topics in networks*, New York, NY, USA.
38. Team, T. M. (2012). Mininet: An instant virtual network on your laptop (or Other PC) (online). http://www.mininet.org.
39. McCauley, M. (2009). About POX | NOXRepo (online). http://www.noxrepo.org/pox/about-pox/.
40. McCauley, M. (2013). Noxrepo/pox. Github (online). https://github.com/noxrepo/pox.
41. McCauley, M. (2014). POX Wiki—Open Netwroking Lab—confluence (online). https://openflow.stanford.edu/display/ONL/POX+Wiki.

42. Big Switch Networks, Inc. | The Leader in Open Software Defined Networking. (2015). Big switch networks (online). http://www.bigswitch.com/.
43. Floodlight OpenFlow Controller—Project Floodlight. (2015). Big switch network (online). http://www.projectfloodlight.org/floodlight/.
44. Architecture—Floodlight Controller—Project Floodlight. (2012). Big Switch Network (online). https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Architecture.
45. Jha, R. K., Kharga, P., Bholebawa, I. Z., Satyarthi, S., Gupta, A., & Kumari, S. (2014). OpenFlow technology: A journey of simulation tools. *International Journal of Computer Network and Information Security (IJCNIS), 6*(11), 49–55.
46. Blial, O., Mamoun, M. B., & Benaini, R. (2016). An overview on SDN architectures with multiple controllers. *Journal of Computer Networks and Communications*. doi:10.1155/2016/9396525.
47. Ma, Y. W., Chen, J. L., Tsai, Y. H., Cheng, K. H., & Hung, W. C. (2016). Load-balancing multiple controllers mechanism for software-defined networking. *Wireless Personal Communications*. doi:10.1007/s11277-016-3790-y.

Idris Z. Bholebawa received his B.E. in ECE from VNSGU, India in 2009. He received his M.Tech. from S. V. National Institute of Technology, India in 2011. His specialization is in Communication and Networking. He is currently a full time research scholar in department of Electronics and Communication Engineering at S. V. National Institute of Technology under the supervision on Dr. (Mrs.) U. D. Dalal. His research area includes Software Defined Networking, an OpenFlow Technology. Especially he is focusing on Optimization issues in networking and efficient performance analysis using OpenFlow.

Upena D. Dalal presently working as an Associate Professor in Electronics Engineering Department of S. V. National Institute of Technology, Surat, INDIA. She has 22 years of academic experience. She has completed her B.E. (Electronics) from SVRCET, India in 1991 and obtained M.E. (Electronics and Communications) from DDIT, India, with Gold Medal. She is also awarded with 5th N. V. Gadadhar memorial Award by IETE. She has published many conference and journal papers at National and International level. She has guided many UG and PG projects, dissertations and seminars in the area of advance communication systems. She has completed her Ph.D. in 2009 and supervising 15 research scholars presently. Her book on "Wireless Communication" is published by Oxford University Press in July 2009. One more book edited by her and Dr. Y. P. Kosta titled "WiMAX New Developments" is published by Inteh, Vienna, Austria. She is honored by "Rashtriya Gaurav Award" by India International Friendship Society.