Performance Analysis of Ryu-POX Controller in Different Tree-Based SDN Topologies

Danijel CABARKAPA, Dejan RANCIC
University of Nis, Faculty of Electronic Engineering, A. Medvedeva 14, Nis, Serbia
d.cabarkapa@elfak.rs

***Abstract*—Next generation networking architecture is required to be reliable, scalable, flexible, secure and has other advanced features. Traditional TCPIP networks are complex and cannot meet the requirements for high-quality network services. Software Defined Network (SDN) is an important technology that enables a completely new approach in how we develop and manage networks. SDN divides the data plane and control plane and promotes logical centralization of network control so that the controller can schedule the data in the network effectively through OpenFlow protocol. In this paper, we simulate the two SDN controllers of Ryu and POX, and compare their latency and throughput performance under Simple-Tree-Based (STB) and Fat-Tree-Based (FTB) network topologies. An SDN networking model has been designed using a Mininet emulator, and the code for custom STB/FTB topology is executed in Python script. Simulation outcomes indicate that in latency mode Ryu controller exhibited better results than POX controller, making it more suitable for small-scale SDN deployments. From the throughput simulation, POX controller displayed better results than Ryu, showing that it is able to respond to requests more promptly under complex FTB traffic loads, but with more hardware resources utilization.**

***Index Terms*—network topology, next generation networking, tree data structures, software defined networking, soft switching.**

I. INTRODUCTION

In contemporary distributed TCPIP architecture, network devices should communicate with each other through various communication protocols to negotiate the precise network behavior based on the configuration of every individual device. Network devices are "closed components" and neither flexible nor programmable by any means. According to basic TCP-IP networking principles, forwarding control and management planes all are tied up together in most of the network devices. The combination of these three planes in the same device chassis makes it quite complex and resistant to manage [1].

The concept of "programmable networks" has been proposed as a motivation to increase network evolution. The fundamental idea has evolved into what today is called Software Defined Network (SDN). SDN is having the ability to separate the data and control functions of core networking devices and consolidates all the control in a single node called the network controller. SDN architecture allows centrally controlling capabilities and provides a global view of a whole underlying physical network, and allows to dramatically simplifying network management [2]. This centralized entity provides programmable control of the whole network and enables real-time control of all the underlying devices. The control layer is comprised of the logically centralized "network intelligence" software, manages flow control, and has a global view of the network. Controller is the core part of the SDN network and it gives the instructions to the switches and other network devices in the physical layer through Southbound Interface (SBI). SBI helps the controller to provision physical and virtual network devices intelligently. OpenFlow is the most commonly used standardized SBI Application Programming Interface (API) and follows the basic SDN principle of separation between the control and data planes [3]. In present-day SDN networks, Ethernet switches are replacing by OpenFlow switches due to their less flexibility. Each OpenFlow switch dynamically maintains a flow table, which consists of flow rules that determine the handling of network packets. Communication between the control and the application layer is established by using Northbound Interface (NBI) API [4]. SDN devices use numerous approaches to controller-based networking that should provide the desired abstractions, from centralized control and management plane interactions to decoupling approaches and proprietary device APIs.

Although the fundamental function of a SDN controller is flow management, several different metrics can be used for its performance analysis. Two of the most important questions frequently asked are how fast can a SDN controller respond to requests sent by the OpenFlow switch (latency), and how many sent and received flow messages can a controller handle per second (throughput).

In this paper we make a performance analysis of Ryu and POX controller, considering latency and throughput as the key metrics. Both Ryu and POX controllers are runnable on Python. The reason to select these two controllers is the availability of controller source code and implementation. We have compared controllers using Mininet emulator, along with a detailed analysis of their performance in different custom network topologies, such as Simple-Tree-Based (STB) and more complex Fat-Tree-Based (FTB) topology. We have also extended the latency comparison to traditional IP network topology, where the network topology is modified so that there is no central controller device.

The rest of this paper is organized as follows. Section II presents the SDN layered architecture and gives an overview of OpenFlow, Ryu and POX controller features. Research efforts and related literature are detailed in Section III. Section IV presents the simulation environment, research methodology and metrics. Section V shows the results of SDN controllers analysis. Concluding remarks and future research directions are given in Section VI.

Digital Object Identifier 10.4316/AECE.2021.03004

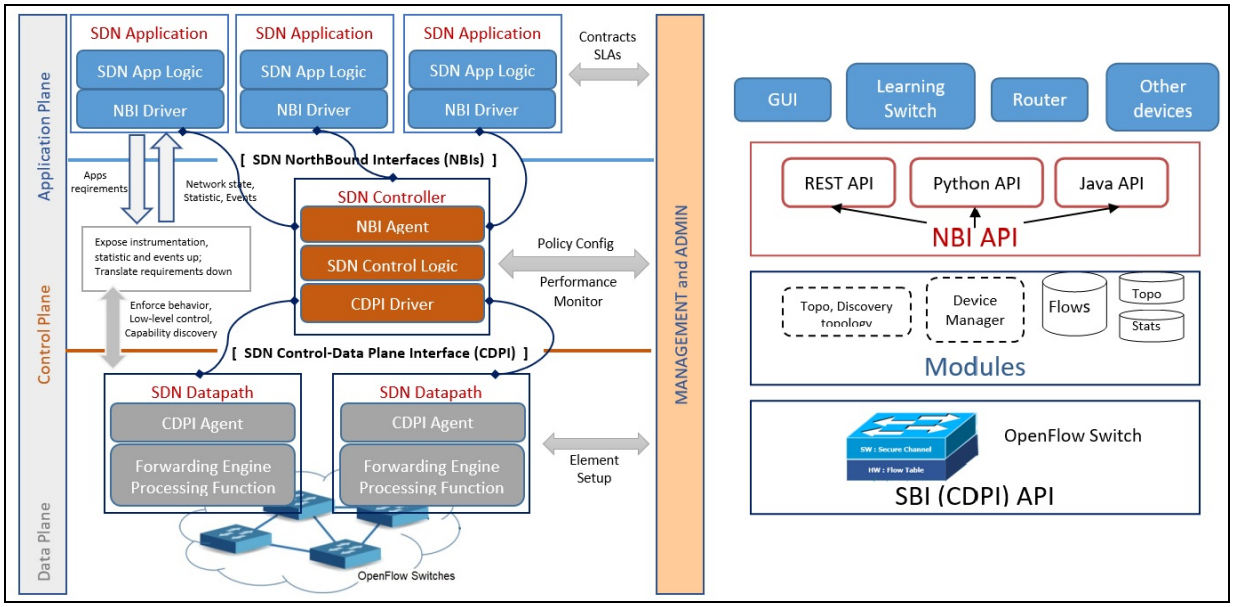

Figure 1. Overview of a typical layered SDN architecture (left) and SDN controller architecture (right)

II. SDN Background

At first, this section provides an overview of the layered SDN architecture and OpenFlow switch. Furthermore, this section presents the generic architecture of SDN controllers and gives an overview of Ryu and POX controllers.

A. SDN Layered Architecture

The architecture of the SDN network can be divided into three planes: data plane, control plane, and application plane. SDN separates the control and data plane of a traditional network device. The control plane is implemented through one or more logically centralized controllers. Control functionality is removed from network devices, that will afterward become simple packet forwarding network nodes. The relationships between SDN modules can be seen in Fig. 1, which shows a basic overview of a typical SDN architecture.

SDN Application consists of one Application Logic and one or more NBI Drivers. The SDN is programmable through software applications that interact with the underlying data plane devices. Higher-level logic can be implemented directly through these applications on top of controllers, which communicate through NBI Agents (REST, JSON, etc.) [5]. The SDN network comprises interconnected forwarding devices, which represent the data plane. The SDN Datapath is a logical network device that comprises a Control-to-Data-Plane Interface Agent (CDPI) and a set of one or more traffic forwarding engines and traffic processing functions. One or more Datapath may be contained in a single physical network device. The CDPI defined as an interface between controller and Datapath, provides event notification, statistic reporting, capabilities advertisement, and high-level control of all forwarding operations. CDPI interface is generally implemented using the OpenFlow protocol.

OpenFlow is the most widely accepted and deployed SBI standard for SDN and represents a protocol that is used for the communication between the controller and forwarding devices to install the data processing rules. OpenFlow modifies the SDN network in the sense that data plane elements become simple devices that forward packets according to rules given by the controller. The main components of a controller-based OpenFlow network are OpenFlow switches. Each OpenFlow switch dynamically maintains a flow table, which consists of flow rules (entries) that determine the handling of packets. Flow entries consist of pattern fields (that matches on bits in the packet header), a list of actions (drop, flood, forward, modify or send the packet to the controller), a set of counters (to track the packets), and a priority field. Further, 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.

B. SDN Centralized Network Control

Various open-source SDN controllers are currently being used for deploying network architecture, and these controllers are Pox, Ryu, FloodLight, ONOS, ODL, OpenDayLight, etc. [6]. Fig. 1 (right) shows the architecture of the SDN controller. The figure depicts the modules that provide the core functionality of the controller, both NBI and SBI, and a few applications that might use the controller. The SBI API is used to interface with the SDN devices. This API is OpenFlow in the case of a commercial Open SDN controller or some alternative in other SDN solutions. This means that in some product offerings both OpenFlow and alternatives coexist on the same controller. OF-Config and Open vSwitch Database Management Protocol (OVSDB) are both open protocols for the SBI, as described in [7-8]. There is a lack of a standard for the NBI, which has been implemented in several different forms. For example, the FloodLight and OpenDayLight controllers both use a Java and REST/RESTful API for applications running on separate machines, Ryu and POX use Python API, etc. [9]. Controllers coordinate network control activities by establishing communication channels between each other using an east-west API, mainly for scalability and resiliency.

The core functions of the controller are topology and device discovery and tracking, flow management, device management, and statistics tracking. These are all implemented by a set of modules internal to the controller. All the controller functions are implemented via changeable modules, and the feature set of the controller may be

adjusted to specific requirements of SDN networks. The controller tracks the topology by learning of the existence of OpenFlow switches and other SDN devices and tracking the connectivity between them. Network traffic analysis can be performed in real-time using machine learning algorithms, databases, and other software tools.

Mininet is freely available open-source software that emulates OpenFlow devices and SDN controllers. With Mininet is possible to create scalable (up to hundreds of nodes) networks by using Linux processes in network namespaces. Mininet API and orchestration modules are written in Python, with core emulation performed by compiled C code. Also, Mininet provides an easy way to experiment with various network topologies. It is possible to develop and test code on Mininet, and OpenFlow controller or switch can implement into a real system with no changes for performance evaluation and deployment [9].

Ryu controller is an open-source and component-based SDN framework implemented entirely in Python. Ryu uses OpenFlow protocol to associate with the switches to modify how the network will manage traffic flows and allows an event-driven programming paradigm in which the flow of the program is determined by events. The module called *ryu.controller.ofp_event* exports event classes that describe receptions of messages from connected switches. Ryu provides software components with well-defined APIs that make it simple to create control applications and SDN network management. In addition, Ryu supports various protocols for managing network infrastructure, such as OpenFlow, Netconf (RFC 6241), OF-config, etc. [10]. The purpose of these protocols is to gathered network intelligence by using a controller, performed analytics, and synchronized the new network rules. The controller uses NBI APIs such as REST/RESTful, REST/RPC user-defined APIs, etc. Ryu provides a set of specific components such as OpenStack/Quantum virtualization, Firewall, OFREST, etc. for SDN applications [11].

POX is a component-based open-source controller implemented in Python. Additionally, a POX controller is capable of converting particular OpenFlow devices to operate as switches, firewalls, load balancers, etc. POX controller can have direct access and manipulation capability to the forwarding devices in presence of the OpenFlow protocol. POX relies on a specific model in which the whole SDN network devices, as well as activities, are recognized as separate components that can be isolated and utilized every time and place the need is. The location of POX is between network components on one side and the applications on the other side. Furthermore, POX is responsible for achieving any type of communication between applications and SDN devices. POX allows user to develop their components according to their need. The full POX documentation is available on GitHub at [12].

III. Related Works

The very first SDN controller was made by Nicira Networks and is named NOX, which was developed along with the first version of the OpenFlow protocol [13]. Then, the POX controller is developed as a revised version of NOX with Python support. FloodLight controller [14] was developed with OpenStack plug-in which helps it in controlling a large number of network devices and resources. In paper [15], five controllers (Ryu, Pox, Trema, Floodlight, and OpenDayLight) are compared, and authors collect properties of each controller under specific evaluation: CDPI/REST API support, programming language, modularity, virtualization, etc. In [16], Floodlight and OpenDayLight are examined as far as delay and loss in various topologies and traffic loads. The results show that OpenDayLight has the best latency results under low traffic loads and for tree topologies. Floodlight exhibits the best packet loss results under high traffic volumes for tree topologies. In [17], authors have shown the performance comparison performed between the NOX, POX, Trema and Floodlight in reactive and proactive mode. The results showed that the best performance is achieved when the controller is operating in the proactive mode because forwarding rules are installed on the switch. Research in [18] gives a more extensive investigation of open-source controllers Ryu, POX, ONOS and ODL. Authors were limited to parameters such as throughput and latency using Cbench tool. In [19] authors use a qualitative comparison of different SDN controllers, along with a quantitative analysis of their performance in different network scenarios. Precisely, authors categorize and classify 34 controllers based on their capabilities and properties and discuss in-depth capabilities of benchmarking tools used for SDN controllers. In [20], the authors have presented a basic list of tests that should be conducted to evaluate the performance of a controller. Authors in [21] presented a framework named HCprobe to compare seven controllers: Ryu, Beacon, Maestro, MUL, FloodLight, NOX and POX. To evaluate the efficiency of these controllers, the authors performed additional measurements like reliability, scalability and security along with throughput and latency. The results show that FloodLight, Beacon and MUL obtained minimum latency, while Beacon performed good results in the throughput test. In [22] authors use a comparison of performance metrics such as service delay, utilized bandwidth and received packets using network monitoring tools like IPERF and D-ITG to analyze the functionality of the POX controller for the SDN environment. The results of this research were the recommendation of using POX controller for the interaction with OpenFlow switches.

IV. Performance Analysis

A. Simulation Environment

The simulation hardware and software specifications are shown in Table I. Controller performance analysis and network topology creation were performed in an environment of a Windows 10 (ver. 20H2). The hypervisor used is Oracle VirtualBox 6.1.16, and it is used to instantiate two Virtual Machines (VM), each for one type of controller. Secure Virtual Machine Mode option must be enabled in BIOS setup due to supporting AMD-V virtualization mode. Each VM is allocated 1 CPU and 6 GB of RAM. All the simulations run on a Xubuntu Server 20.04.1. Each VM contains Mininet with implemented Ryu or POX controller. Each controller ran on an individual VM and connected to predefined STB or FTB topology. The listening port number assigned to Ryu and POX controller are 6653 and 6633.

TABLE I. SIMULATION HARDWARE AND SOFTWARE SPECIFICATIONS

		PC	VM
Hardware	Processor	AMD Ryzen 5 3600, 3.6 GHz (6-core)	1 CPU, 6-core
	RAM	16 GB DDR4	6 GB DDR4
Software	OS	Windows 10 (64-bit), ver. 20H2	Xubuntu 20.04.1 (64-bit)
	VirtualBox	-	6.1.16, r140961
	Mininet	-	2.3.0d6
	Ryu	-	4.3.2
	Pox	-	0.5.0
	Python	3.8.5	3.8.5

Python 3 is used as a scripting language to write the custom network STB/FTB topology, instead of accepting of built-in Mininet topology model. The elements which are used to forward traffic from one host to another are switches that support the OpenFlow 1.3 protocol. After controller initialization, Mininet loads a Python script to instantiate the custom topology. Each OpenFlow switch is assigned a unique port for keeping track of network traffic. The Mininet will automatically assign MAC addresses that match the host's names. This will increase stability and consistency during the simulation and simplify the programming logic in Ryu or POX controller.

All the nodes have assigned a unique IP address from 10.0.0.0/24 address range, and unique MAC address. The IP/MAC addresses are (10.0.0.1/00:00:00:00:00:01) for node *h1*, (10.0.0.2/00:00:00:00:00:02) for node *h2*, etc. To make switches connect to Ryu or POX controller, we have used 127.0.0.1 virtual loopback IP address.

B. Custom STB/FTB Network Topologies

In this paper, we have considered two custom topologies for simulation, one simple with a relatively small number of network nodes, and the second with a more complex topology. In a tree network topology, all the OpenFlow switches and hosts are connected with each other in a hierarchical form, as discussed in [23]. Tree-based topology was chosen for simplicity and to ensure that all switches are well-exercised.

At first, we implemented a Simple Tree-Based (STB) topology over 4 switches (*s1-s4*), 8 hosts (*h1-h8*), and one Ryu or POX controller, as shown in Fig. 2

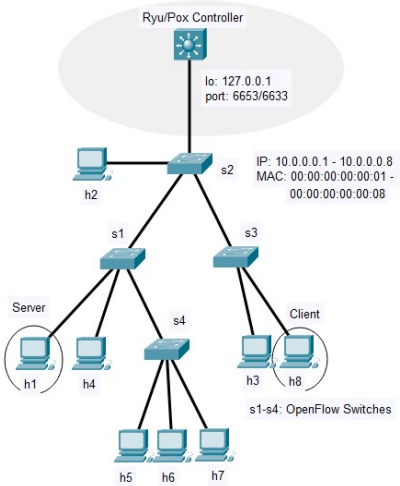

Figure 2. Custom Simple-Tree-Based topology

```
1:  from mininet.net import Mininet
2:  from mininet.node import Controller, RemoteController
3:  from mininet.cli import CLI
4:  from mininet.log import setLogLevel, info
6:  ryu_ip = '127.0.0.1'
7:  ryu_port = 6653
8:  def customTree():
9:      net=Mininet (topo=None, build=False)
10:     info ('Adding Ryu controller\n')
11:     net.addController ('c0', controller=RemoteController,
                ip=ryu_ip, port=ryu_port)
12:     info ('Adding hosts\n')
13:     h1, h2, h3, h4, h5, h6, h7, h8=[net.addHost(h) for h in ('h1', 'h2',
                'h3', 'h4', 'h5', 'h6', 'h7', 'h8')]
14:     info ('Adding switches\n')
15:     s1, s2, s3, s4=[net.addSwitch(s) for s in ('s1', 's2', 's3', 's4')]
16:     info ('Adding switch links\n')
17:     for sa, sb in [ (s1, s2), (s2, s3), (s1, s4)]:
            net.addLink( sa, sb,)
18:     for h, s in [ (h1, s1), (h2, s2), (h3, s3), (h4, s1), (h5, s3), (h6,
            s4),  (h7, s4), (h8, s4) ]: net.addLink( h, s )
19:     info ('*** Starting Mininet network ***\n')
20:     net.start()
21:     info ('*** Running Mininet CLI ***\n')
22:     CLI(net)
23:     info ('*** Stop Mininet network ***')
24:     net.stop()
25:     if __name__ == '__main__':
26:     setLogLevel('info')
27:     customTree()
28: exit(0)
```

Figure 3. Python script for STB topology with Ryu controller

Fig. 3 shows a Python script for creating custom STB topology. Mininet (line 1) is the main class to create a network. Ryu controller IP address and port number are defined in lines 6 and 7. Methods *addController()*, *addLink()*, *addHost()* and *addSwitch()* add a controller, link, host and switch to a topology and returns their names (lines 11-18). Since the low number of network nodes in STB, we use a simple for loop to add all the nodes to the network and connect them to each other. Host *h1* is denoted as a server, and *h8* as a client. We organize all the STB source code in a single file *stb-topology-dc.py*

As a second custom network, we implemented Fat-Tree-Based (FTB) topology over the 32 hosts (*h1-h32*), 12 switches (*s1-s12*) and one Ryu/POX controller. Each switch has four ports to connect with hosts. Typical tree-network topologies consist of either two or three level trees of switches or routers [23]. In this paper, all FTB switches are interconnected to each other, forming three-level architecture: core, aggregation and edge switches. The graphical representation of FTB topology is shown in Fig. 4. The *K* variable stands for the number of ports in each OpenFlow switch and the Fig. 4 is an example of a *K=4* fat-tree topology, where each edge switch connected to hosts and aggregation layer switches via 4 ports.

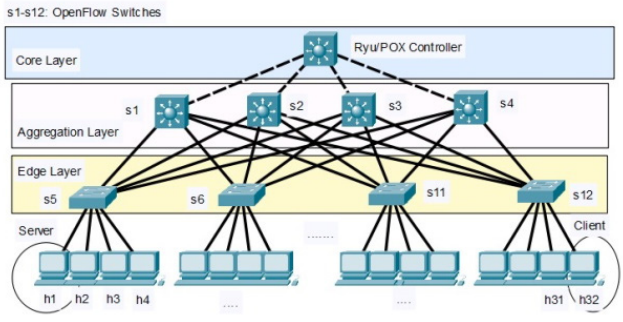

Figure 4. Custom Fat-Tree-Based topology (K=4)

```
1: from mininet.topo import Topo
2: class FatTree (Topo):
3: def __init__(self, half_ports = 2, **opts):
4:     Topo.__init__(self, **opts)
5:     agrsw = []
6:     hnum = 0
7:     snum =0
8:     for i in range (half_ports):
9:         snum += 1
10:        agrsw.append (self.addSwitch ('s%s' % snum))
11:    for i in range (half_ports*2):
12:        snum += 1
13:        sw=self.addSwitch  ('s%s' % snum))
14:        for j in range (half_ports):
15:            self.addLink(sw, agrsw[j])
16:        for j in range (half_ports):
17:            hnum += 1
18:            host = self.addHost('h%s' % hnum)
19:            self.addLink(sw, host)
20: topos={' ftb-topology-dc': FatTree}
```

Figure 5. Python script for FTB topology

Using Python scripting, this FTB topology was created which comprises 3 layers, as shown in Fig. 5. Due to the complexity of FTB, we used a high-level API with topology template abstraction. *Topo* is the base class for Mininet topologies, which provides the ability to create reusable and parametrized topology templates (line 1). Line 2 represents class definition. There are multiple ways that this class may be used, but one simple way is to specify the class as the default host and link classes and constructors to Mininet. In the script, *__init__*is the reserved method and represents the constructor for a class (line 3). The *self* variable represents the instance of the class and binds the attributes with the given arguments. Methods *self.addSwitch(), self.addLink()* and *self.addHost()* import switches and hosts into topology and connect them (lines 10-19). We organize the FTB topology source code as a *ftb-topology-dc.py*

Due to a huge number of links in the FTB topology, "broadcast storms" are frequent and intensive. This undesirable network traffic circling endlessly in the network, due to the destination address in an unknown network. Spanning-Tree Protocol (STP) is a link management protocol that provides path redundancy while preventing undesirable loops in the network [24]. STP ensures only one stable path exists between any two nodes. Both Ryu and POX have a built-in Python script for which the STP function is achieved using OpenFlow, and we used *ryu.app.simple_switch_stp_13.py* script for the Ryu controller and *openflow.spanning_tree* for the POX controller.

C. Methodology and Metrics

After all experimental setup was prepared, we began the controller performance simulation. To perform the simulation of the Ryu controller, it is followed step-by-step procedure:

(Step 1) The first step is to run the Ryu controller using the STP protocol. We provide Ryu terminal command:

$ryu-manager ryu.app.simple_switch_stp_13.py

(Step 2) The next step is to run the custom STB topology script with the following command:

$sudo ~/mininet/custom/stb-topology-dc.py

This command starts the Mininet CLI prompt, performs the initialization procedure of hosts and switches and establishes links between them. In the second VM, FTB topology initialization involves running a *ftb-topology-dc.py* script.

(Step 3) In this step, we define client and server nodes. As shown in Fig. 2 and Fig. 4, for STB *h1* denotes server and *h8* client, and for FTB *h1* denotes server and *h32* client. We have used *xterm* command for the server and client setup.

(Step 4) For the performance analysis, we need to generate the traffic between client and server and log the events using the IPERF networking tool. On the server-side we enter the command: *$iperf -s -p 6633 -i 1 > stb-ryu-result*

At the client-side, we need to provide server IP address, controller port number, and time of simulation (50 sec.) by the following command: *$iperf -c 10.0.0.1 -p 6633 -t 50*

(Step 5) The next step is parsing the log data file for obtaining specific simulation results. For parsing, we have used *-grep*, *-awk* and *-tr* commands

(Step 6) We have used the Gnuplot tool to plot the graphs of obtained results.

On the other VM, an identical procedure was performed for the POX controller.

Latency metrics: This group of metrics deals with the time between packets sent to the controller and the response received at the OpenFlow switch. The latency between the controller and OpenFlow switch is of vital importance since it impacts the performance of the entire network. Also, this metric helps to understand the controller response time under an event like network path failure. Round-Trip-Time (RTT) evaluation is an important parameter because it identifies the communication delay between the controller and the switch. If the controller and switches are physically far apart, the increased RTT will contribute to increased latency. Similarly, the time complexity of packet processing at the controller affects the overall performance. A latency performance comparison between POX and Ryu controller is achieved by execution of ICMP connectivity test using *ping* (Echo request and reply message) command. A *ping* test is performed between end hosts *h1-h8* (for STB) and *h1-h32* (for FTB).

Throughput metrics: A SDN network performance is achieved from available throughput. Throughput is generally defined as a rate for processing flow requests by the controller. From the testing point of view, it is the number of *packet_in* messages sent and the corresponding *packet_out* messages received per second. Throughput measures the rate from OpenFlow switch to controller and back to switch and it is a major factor in determining the overall number of controllers required to handle traffic load on a network.

We used the IPERF networking tool to test network performance between two hosts, server and client. A typical IPERF output contains a timestamped report of the amount of data transferred through the network.

V. Research Objectives and Simulation Results

The goal of this analysis is to attempt to measure the latency and throughput features of the Ryu and POX controller by implementing two different Mininet SDN topologies in a simulation environment. Our interest is to evaluate the situation where the number of OpenFlow switches and hosts changes from small to significantly large when we can prove that the number of SDN nodes cusses notable performance decrease, for all kinds of topologies.

First, we observe the latency against ICMP packet size

and the number of switches and hosts in topologies. For the latency comparison, we selected simulation results proposed in [25], where custom traditional IP-based network topology consists of 32 hosts and four OpenFlow switches, and which is designed using NS2 network simulator. In this traditional IP network, each switch is an independent device and there is no central switch that acts as a SDN controller. An interesting observation is the Ryu controller provides lower latency than POX and switches in the traditional IP network.

Considering SDN controller throughput, single-threaded controllers Ryu and POX show different results, especially in FTB topology where there are a larger number of OpenFlow switches and hosts, and where network traffic is more intensive. We concluded that throughput parameters can be limited by the hardware resources and capabilities of the controller itself.

Latency results: First, let's analyze RTT latency performance between the proposed OpenFlow STB/FTB network and traditional IP network. We observe the latency against ICMP packet size, from 100B to 1400B. In each performance test, we send 10 ping packets with the following sizes: 100, 500, 1000, and 1400 Bytes. At first, we send *ping* command from host *h1* to host *h8* (for STB) and then we repeat *ping* from host *h1* to host *h32* (for FTB). These *ping* commands are cycled 10 times for each test. Then we calculated the average values of measured RTTs of the first packet of the flow per test. In an IP network, a *ping* test is performed between the farthest nodes, as well. These average values of the measured RTTs for all three topologies are tabulated in Table II and shown graphically in Fig. 6.

TABLE II. AVERAGE RTT FOR THE FIRST PACKET OF THE FLOW BETWEEN FARTHEST NODES IN STB, FTB AND TRADITIONAL IP NETWORK

ICMP Packet Size (B)	**STB topology**		**FTB topology**		**Traditional IP Network**
	Ryu RTT(ms)	POX RTT(ms)	Ryu RTT(ms)	POX RTT(ms)	RTT (ms)
100	1.86	10.7	10.8	26.1	7.2
500	1.92	12.6	15.0	28.4	10.1
1000	6.70	22.8	22.5	33.1	34.4
1400	9.05	29.9	26.3	35.5	57.8

It is clearly visible that the propagation delay between nodes in STB, FTB, and IP networks is very different. From Fig. 5, the average RTT for the first packet of the flow is the minimum for Ryu controller and 100 Byte ICMP packet size in STB topology (RTT=1.86 ms). For STB-POX and FTB-Ryu topologies, RTT is almost same for the all ICM packet size. In the FTB topology, the average RTT is the minimum for the Ryu controller. Also, the difference between average RTT for POX controller in the FTB is greater among all other RTTs, except for the IP network. It is related to complex FTB topology and STP protocol, which is running to prevent broadcast loops.

In this simulation environment, it is observed that the latency in the IP network is higher than that in FTB topology, especially for larger ICMP packets. The reason for this is that the larger ICM packet requires a longer processing time on a larger number of standalone switches/routers in the traditional IP network. The highest RTT value (RTT=57.8 ms) was measured for a complex IP network consisting of 32 hosts and 4 switches, and for the ICMP packet size of 1400B. We noticed that ICMP packet length impacts the performance of the controller.

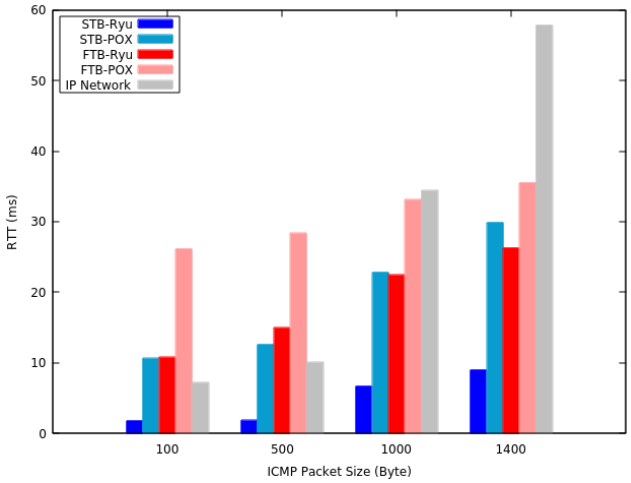

Figure 6. Average RTT for the first packet of the flow in STB/FTB topology and the traditional IP network

From the obtained results, it is obviously shown that a FTB topology is taking more time for transmission of the packet to its destination node. The initial process of establishing network flow consumes time that introduces latency in the network. When the first packet sent by *h1* arrives at the OpenFlow switch, this switch does not know how to route it, encapsulates it, and forwards all the contents of the incoming packet to the controller, being responsible for managing the installation of the flow tables in each switch. On the other hand, the Ryu controller has the least RTTs, mostly because of the less complex algorithms involved in the controller. However, the consequence of a less complex algorithm is usually a smaller number of controller capabilities.

The latency results are considered to be higher in a real SDN network, where in our simulation the communications between the switches and the controller are done through the loopback interface 127.0.0.1, where there is no delay while packets traversing the virtual ports, as compared to the real hardware ports. Note that the latency metric is intrinsically the inverse of the throughput metric.

Throughput results: Let's explain a throughput simulation, where the Ryu/POX controller is evaluated for the maximum amount of data it can process in a second between two SDN nodes. Fig. 6 shows the results obtained by performing transmission between client and server in the simple STB topology. To measure controller throughput, the IPERF test has been executed in 50 sec. on the client *h8*, and data have been collected every 1 sec. on the server *h1*.

It should be noted that the latency metric, as directly reported previously, is the inverse of the throughput metric. It is calculated from the graph that the average throughput stays at 28.66 Gbps for Ryu, and 27.08 Gbps for the POX controller. According to the observation, the average throughput in the POX controller is 5.51% less than the Ryu controller. Fig. 7. also shows that the throughput variations moderately fluctuated within the duration of the simulation. Most of the time simulation was running well, but there are few instances of excessive variations in the throughput, primarily for the POX controller.

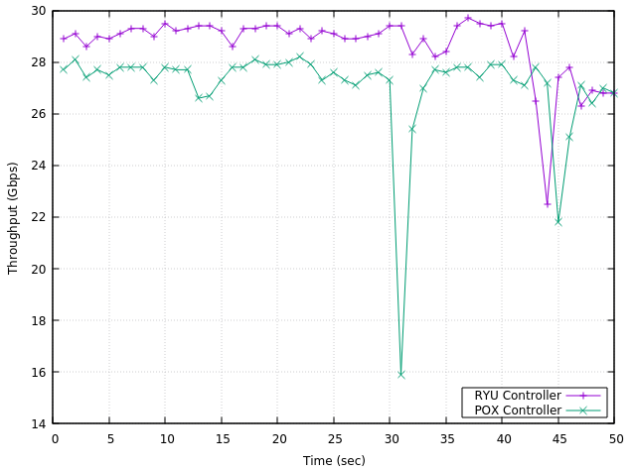

Figure 7. Throughput in the STB topology with Ryu and POX controller

Dropping instances were observed frequently by the end of simulation time, leading to degraded performance of the simulation run. Additionally, for the POX controller, there was a large drop in throughput that occurred at 31 sec. of simulation, when it drops to a value of 15.9 Gbps.

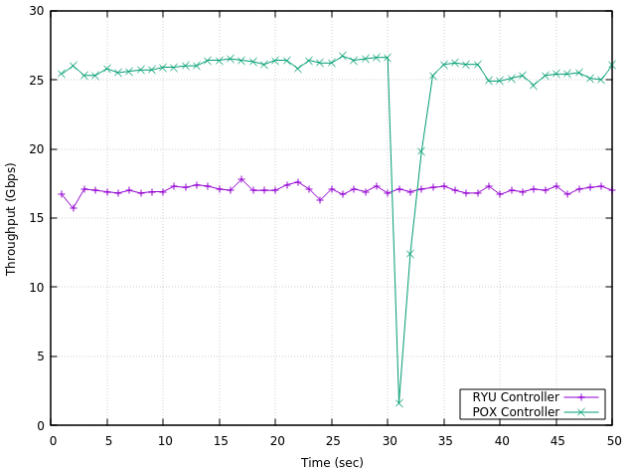

Figure 8. Throughput in the FTB topology with Ryu and POX controller

Graph on Fig. 8, shows the results obtained by performing transmission between client and server in the FTB topology. As in the previous case, the simulation has executed in 50 sec. on the *h32*, and data have been collected on the server *h1*. It is calculated from the graph that the average throughput stays at 17.02 Gbps for Ryu, and 24.97 Gbps for the POX controller. In this case, the average throughput in Ryu is 31.83% less than the POX controller. Within the duration of the simulation, the throughput variations are scientifically more uniform than previous STB case. There is only one dropping instance of throughput for the POX controller. This large drop now occurs again at 31 sec. of simulation, and the value of throughput was only 1.63 Gbps.

For a more complex FTB topology, the Ryu controller shows 31.83% lower throughput than the POX. This result shows that Ryu lacks the appropriate STP algorithm implementation and will result in the packet broadcast storm problem when controlling a network with loops. When Ryu controls a STB network without a loop, throughput results are more uniform. This means that the Ryu controller can be more suitable for small and simple OpenFlow networks where there are a small number of switches. From the statistical analysis results, the continuous polling of data causes overhead on the controller. This is because having a large number of OpenFlow switches causes conflict at the data layer which demands high processing power.

Because we cannot conclude with certainty why the Mininet emulator exhibited such large throughput drops, we decided to do the same simulations three more times to confirm whether this phenomenon is accidental or not, and the Mininet emulator always generates similar results. It appears that the hardware resource utilization, especially in FTB topology is too high in comparison to the scale of nodes support provided by the POX controller. In the benchmarking results of [26], the authors speculated that the POX controller might have a memory leak, which could be the reason for its dropping performance in the throughput test.

VI. Conclusions and Future Work

Software Defined Networking improves the existing network design by introducing advanced control in a centralized way. The SDN controller is the fundamental element used for all operations of data plane management. The controller tracks the network topology by learning of the existence of SDN switches and end-user devices and tracking the connectivity between them. Further, the controller abstracts the details of the controller-to-device protocol, so that the applications above can communicate with those SDN devices without knowing the differences and features of those devices. Besides the responsibilities that the SDN controller possesses there are many challenges as well.

This paper presents both a feature-based comparison and performance analysis of the most commonly used controller implementations Ryu and POX. We compare their throughput and latency performance under Simple-Tree-Based, Fat-Tree-Based and traditional IP network topologies. We showed that the performance of a Ryu/POX controller depends on many different factors: controller hardware and control algorithm configuration, underlying network infrastructure, the number of OpenFlow switches, the number of hosts, the number of threads, etc. Overall, in most of the simulation tests, Ryu and POX controllers show higher throughput and lower latency when OpenFlow protocol is enabled, than results obtained in the traditional IP network.

The obtained experimental results show that POX displayed better throughput results, showing that it is able to respond to requests more promptly under complex traffic SDN loads, where the number of OpenFlow switches is significantly large. However, in a latency viewpoint, Ryu exhibited better results, making it more suitable for delay-sensitive SDN applications, as well as for less complex SDN networks.

Future work can take place in several directions. Because Python controllers (Ryu and POX) do not support multi-threading, one direction for future work can be performance analysis for the multi-threaded controllers such as FloodLight, ONOS, and ODL, and evaluation of difference in throughput scalability between these controllers.

REFERENCES

[1] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, "Are we ready for SDN? Implementation challenges for software-defined networks," IEEE Communications Magazine, vol. 51, issue 7, pp. 36-43, 2013. doi:10.1109/MCOM.2013.6553676

[2] N. Feamster, J. Rexford, E. Zegura, "The Road to SDN: an intellectual history of programmable networks," ACM SIGCOMM Computer Communication Review, vol. 44, no.2, April 2014. doi:10.1145/2602204.2602219

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, "OpenFlow: enabling innovation in campus networks," ACM SIGCOMM Comp. Communication Review, vol. 38, no. 2, pp. 69-74, 2008. doi:10.1145/1355734.1355746

[4] T. Zhang, F. Hu, "Controller architecture and performance in software-defined networks," in Network Innovation through OpenFlow and SDN, CRC Press, 1st edition, 2014. doi:https://doi.org/10.1201/b16521

[5] W. Zhou, L. Li, M. Luo, W. Chou, "REST API design patterns for SDN northbound API", 28th International Conference on Advanced Information Networking and Applications Workshops (WAINA), 2014. doi:10.1109/WAINA.2014.153

[6] A. Lara, A. Kolasani, B. Ramamurthy, "Network innovation using OpenFlow: A survey," IEEE Comm. Surveys Tutorials, vol. 16, issue 1, pp. 493-512, 2014. doi:10.1109/SURV.2013.081313.00105

[7] R. Narisetty et al., "OpenFlow Configuration Protocol: Implementation for the of Management Plane," 2013, Second GENI Research and Educational Experiment Workshop, 2013, pp. 66-67. doi:10.1109/GREE.2013.21

[8] The Open vSwitch Database Management Protocol: IETF-RFC 7047, ISSN: 2070-1721, 2013, https://datatracker.ietf.org/doc/html/rfc7047

[9] R. Barrett, A. Facey, W. Nxumalo, J. Rogers, P. Watcher, M. St-Hilaire, "Dynamic traffic diversion in SDN: testbed vs mininet," International Conference on Computing, Networking and Communications (ICNC), 2017. doi:10.1109/ICCNC.2017.7876121

[10] R. C. Meena, M. Bundele and M. Nawal, "RYU SDN controller testbed for performance testing of source address validation techniques," 3rd International Conference on Emerging Technologies in Computer Engineering: Machine Learning and Internet of Things (ICETCE), 2020, pp. 1-6. doi:10.1109/ICETCE48199.2020.9091748

[11] Md. T. Islam, N. Islam, Md. Al Refat, "Node to Node Performance Evaluation through RYU SDN Controller," Wireless Personal Communications, issue 1/2020, pp. 550-570, 2020.doi:10.1007/s11277-020-07060-4

[12] POX Github Documentation, https://noxrepo.github.io/pox-doc/html/

[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, S. Shenker, "NOX: Towards an operating system for networks," ACM SIGCOMM Computer Communication Review, 2008. doi:10.1145/1384609.1384625

[14] I. Z. Bholebawa, U. D. Dalal, "Performance analysis of SDN/OpenFlow controllers: POX Versus Floodlight," Wireless Pers. Communic. 98**,** 1679-1699, 2018. doi:10.1007/s11277-017-4939-z

[15] R. Khondoker, A. Zaalouk, R. Marx, K. Bayarou, "Feature-based comparison and selection of Software Defined Networking (SDN) controllers," 2014 World Congress on Computer Applications and Inf. Systems (WCCAIS), 2014. doi:10.1109/WCCAIS.2014.6916572

[16] S. Rowshanrad, V. Abdi and M. Keshtgari, "Performance evaluation of SDN controllers: Floodlight and OpenDayLight" IIUM Engineering Journal, vol. 17, no. 2, pp. 47-57, 2016. doi:10.31436/IIUMEJ.V17I2.615

[17] M. P. Fernandez, "Comparing OpenFlow controller paradigms scalability: reactive and proactive," IEEE 27th International Conference on Advanced Information Networking and Applications (AINA), 2013. doi:10.1109/AINA.2013.113

[18] P. Bispo, D. Corujo and R. L. Aguiar, "A Qualitative and Quantitative assessment of SDN Controllers," International Young Engineers Forum (YEF-ECE), pp. 6-11, 2017. doi:10.1109/YEF-ECE.2017.7935632

[19] L. Zhu, Md M. Karim, K. Sharif, Fan Li, X. Du, M. Guizani, "SDN Controllers: Benchmarking & performance evaluation", arXiv.org, Network and Internet Architecture, 2019. https://arxiv.org/abs/1902.04491

[20] B. Vengainathan, A. Basil, M. Tassinari, V. Manral, S. Banks, "Benchmarking methodology for Software-Defined Networking (SDN) controller performance," IETF, RFC 8456, 2018, https://datatracker.ietf.org/doc/rfc8456/

[21] A. Shalimov, D. Zuikov, D. Zimarina, V. Pahskov, R. Smeliansky, "Advanced study of SDN/OpenFlow controllers," Proceedings of the Central Eastern European Software Engineering Conference CEE-SECR '13, no. 1, pp. 1-6, 2013. doi:10.1145/2556610.2556621

[22] H. M. Noman, M. N. Jasim, "POX controller and OpenFlow performance evaluation in Software Defined Networks (SDN) using mininet emulator," 3rd International Conference on Sustainable Engineering Techniques (ICSET 2020), vol. 881, 2020. doi:10.1088/1757-899X/881/1/012102

[23] C. Gomez, F. Gilabert, M. E. Gomez, P. Lopez, J. Duato, "Deterministic versus Adaptive Routing in Fat-Trees," IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1-8, 2007. doi:10.1109/IPDPS.2007.370482

[24] Po. Chi, M. Wang, J. Guo, C. Lei, "SDN migration-An efficient approach to integrate OpenFlow networks with STP-Enabled networks," 2016 International Computer Symposium (ICS), pp. 148-153, 2016. doi:10.1109/ICS.2016.0038

[25] I. Z. Bholebawa, R. K. Jha, U. D. Dalal, "performance analysis of proposed network architecture: OpenFlow vs. traditional network," International Journal of Computer Science and Information Security, Part I, ISSN 1947-5500 vol. 14, no. 3, pp. 30-39, 2016. https://sites.google.com/site/ijcsis/vol-14-no-3-mar-2016

[26] M. Jarschel, F. Lehrieder, Z. Magyari and R. Pries, "A flexible OpenFlow-controller benchmark," 2012 European Workshop on Software Defined Networking, pp. 48-53, 2012. doi:10.1109/EWSDN.2012.15