EXPERIMENTAL ANALYSIS OF SOFTWARE-DEFINED NETWORKING CONTROLLER SCALABILITY BY USING MININET EMULATION

Danijel Čabarkapa[1], Petar Pavlović[1], Dejan Rančić[2]
[1]Academy of Vocational Studies Šabac, Department of Medical and Business-Technological Studies, Šabac, Serbia, {d.cabarkapa@gmail.com, petar.pavlovic@vmpts.edu.rs}
[2]University of Niš, Faculty of Electronic Engineering, Serbia, dejan.rancic@elfak.ni.ac.rs

Abstract: *Software-Defined Networking (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 dedicated controller can schedule the data in the network effectively primarily through OpenFlow protocol. The impact of network topology on SDN controller performance can be very significant. This paper is a contribution towards the performance evaluation of scalability of the open-source RYU SDN controller by implementing multiple network topology scenarios experimented on the Mininet emulation software. RYU SDN controller is tested by analyzing throughput of the controller and checked its performance in diversified networking scenarios by specifically increasing the number of network nodes (switches and hosts). Further, the authors have provided an extensive idea of how to create an experimental SDN testbed along with analysis of obtained results keeping the networking throughput performance and scalability as the focal point.*

Keywords: *software defined networking, open flow switch, mininet, software switching, ryu controller*

1. INTRODUCTION

Software-Defined Networking (SDN) is an emerging computer network technology to mitigate the current challenges of traditional TCP/IP networks, which suffer from inflexibility in management and higher maintenance cost. The new SDN concept provides a global view of the network and facilitates centralized management through the programmable control plane, making the network more flexible to deploy different functions [1]. The SDN introduces a separation of data plane and control plane. While the data plane resides in the network elements, i.e., switches, the control plane is logically centralized and implemented as an SDN controller software running on one or more servers. In order to allow an exchange of information between these two planes, a communication protocol for this Southbound API [2] is required. Nowadays widely used OpenFlow protocol is such an open-source communication protocol, which enables vendor-independence on both planes [3]. The basic fundamental characteristic of SDN is the separation of the forwarding and control planes. Forwarding functionality, including the logic and tables for choosing how to deal with incoming packets, based on characteristics such as MAC address, IP address, and VLAN ID, reside in the forwarding plane [4]. The fundamental activities performed by the forwarding plane can be described by how it dispenses with arriving packets. The control plane of SDN containing one or multiple controllers is responsible for monitoring network states and determining network management strategies, such as the forward strategies of user flows. Benefited from the simplified design of data forwarding devices and the direct control over network behavior at controllers, SDN is capable of facilitating flexible network management.

While the SDN architecture appears to solve problems within the traditional network architecture, it also comes with some major challenges. In this paper, we highlight one of these challenges, which is related to the controller scalability feature. Topology discovery and controller scalability are crucial for successful production deployments of OpenFlow-based SDN networks. To evaluate SDN controller scalability, several solutions have been introduced over the past years. Separating the forwarding plane from the control plane and taking it on a remote system will generate questions on its capabilities of scaling on various network topology scenarios. Due to the intensive evolution and growth of SDN controllers in the market, this paper aims to present an extensive study on the performance and scalability of an open-source Python-based RYU controller available in the existing literature. RYU controller is tested by analyzing the throughput and checked its performance in diversified networking scenarios by specifically increasing the number of network nodes (switches and hosts). This paper will serve as a guideline to the SDN community to benchmark different SDN controllers.

The rest of this paper is organized as follows. Section 2 introduces a brief background about SDN architecture, OpenFlow, and RYU controller features. Section 3 presents details about the simulation environment, research methodology and metrics. Section 4 provides the obtained experimental results and evaluation of scalability statistics followed by conclusions, future works and references.

2. SDN BACKGROUND

2.1 SDN layered architecture

The architecture of the SDN can be divided into three planes: application plane, control plane, and data plane. SDN architecture concept of computer networks is based on the application of technology separating the control plane from the data plane. The relationships between SDN planes can be seen in Fig. 1 (left), which shows a basic overview of a typical SDN layered architecture.

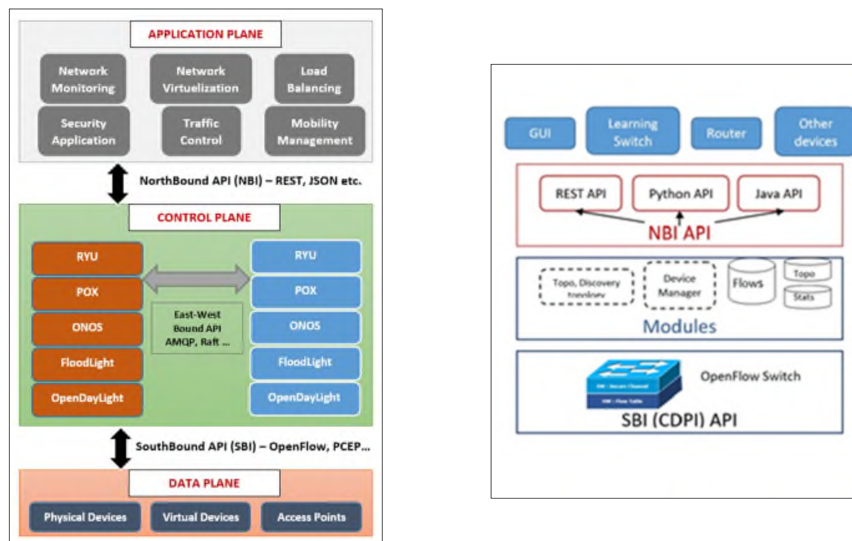

Figure 1: SDN referenced architecture (left) and SDN controller architecture (right)

The application or management plane interacts with the control plane to program the whole network and enforce different policies. Through the programmable platform provided by the control layer, the application plane can access and control switching devices at the infrastructure layer. Examples of SDN applications could include network monitoring, access and traffic control, mobility management, load balancing, and network virtualization. The interaction among application and control layers is done through interfaces that work as communication protocols. Application plane logic can be implemented directly through applications on top of controllers, which communicate through NorthBridge Application Programming Interface (NBI API) such as REST/RESTful, JSON, etc. [5].

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. We usually have routing functionality in mind when talking about the control plane, but in reality, the control plane protocols perform numerous other functions including: interface state management, connectivity management, topology or reachability information exchange, adjacent device discovery, and service provisioning [6]. The control layer bridges the application plane and the data plane, via its two interfaces. For downward interacting with the data plane (i.e., the SouthBound interface SBI), it specifies functions for controllers to access functions provided by switching devices. The functions may include reporting network status and importing packet forwarding rules. As previously mentioned, for upward interacting with the application plane the control plane uses NBI APIs. Since multiple controllers will exist for large network domains, an east-west API interface among the controllers will also be needed to share network information and coordinate their decision-making processes.

The SDN network comprises interconnected forwarding devices, which represent the data plane. Switches only perform actions depending on the controller. The interface that they use to communicate with the controller is called the SBI API. The main function of the SDN switching device is packet forwarding. Specifically, upon receiving a packet, the switch first identifies the forwarding rule that matches with the packet and then forwards the packet to the next-hop node. Compared to packet forwarding in traditional TCP/IP networks, SDN packet forwarding can also be based on other parameters, for example, TCP/UDP port, VLAN tag, and ingress switch port [7]. The principal role of the data plane is to keep current information in the forwarding table so that the data plane can independently handle as high a percentage of the network traffic as possible.

2.2 SDN Controller and OpenFlow protocol

The controller is the most important component in the SDN architecture, where the complexity resides. We have noted that the controller maintains a view of the entire network, implements policy decisions, controls all the SDN devices that comprise the network infrastructure, and provides NBI API for applications. Controllers often come with their own set of common application modules, such as a learning switch, a router, a basic firewall, and a simple load balancer. These are SDN applications, but they are often bundled with the controller. Fig. 1 (right) depicts the modules that provide the core functionality of the controller, both NBI and SBI API, and a few applications that might use the controller.

The SDN controllers have two operational modes, reactive and proactive. In the reactive mode, packets of each new flow coming to the switch are forwarded to the controller to decide how to manage the flow. This approach takes considerable time while installing rules. The amount of latency can be affected by the resources of a controller, their performance, and the controller-switch distance. In the proactive mode, rules are already installed in the switches and therefore the numbers of packets that send to the controller are reduced. In this approach, the performance becomes better and therefore the scalability [8]. One of the key controller functions is to translate application requirements into packet forwarding rules. This function dictates a communication protocol (programming language) between the application layer and the control layer. Strategies to design high-level languages for SDN controllers take at least two formats. One strategy is to utilize existing high-level languages, such as C++, Java, and Python, for application development. The other strategy adopts a clean-state design to propose new high-level languages with special features to achieve efficient network behavior.

Further, 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 specific software tools.

Various open-source and commercial controllers are being used for deploying SDN network architecture - POX, RYU, FloodLight, ONOS, ODL, OpenDayLight, etc. [9]. SDN controller software is specific, and 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 [10]. 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.

OpenFlow is a protocol that describes the communication between OpenFlow switches and a controller. OpenFlow modifies the SDN network in the way that data plane elements become simple devices that forward packets according to rules given by the controller. The main components of a controller-based SDN network are OpenFlow switches. Each OpenFlow switch dynamically maintains a flow table, which consists of flow rules (entries) that determine the handling of packets. Each entry in the flow table has three parts, “header” for matching received packets, “action” to define what to do for the matched packets, and “statistics” of the matched traffic flow [11]. The OpenFlow protocol offers suitable flow table manipulation services for a controller to insert, delete, modify, and lookup the flow table entries through a secure TCP channel remotely. The OpenFlow specification is continuously evolving with new features in every new release. With more and more vendors releasing their OpenFlow-enabled products and solutions, more software projects are being developed based on OpenFlow specifications.

2.3 RYU SDN controller

Ryu controller is an open-source and component-based SDN framework fully implemented in Python. Source code can be found on GitHub, provided and supported by Open Ryu community [12]. The architecture of a RYU controller is shown in Fig. 2. RYU SDN controller has three layers. The top layer consists of network logic applications known as the application layer. The middle layer consists of network services known as the control layer, and the bottom layer consists of physical and virtual devices know as an infrastructure layer.

Ryu uses OpenFlow protocol to associate with the switches to modify how the network will manage traffic flows and allows an event-driven programming principle. Ryu provides software components with well-defined APIs that make it simple to create control applications and SDN network management. In respect of support for SBI protocols, RYU is working with various protocols for managing network infrastructure, such as Open vSwitch Database Management Protocol (OVSDB), Netconf (RFC 6241), OF-config, XFlow etc. [13, 14]. The purpose of these protocols is to gathered network intelligence by using a controller core, performed analytics and statistics, and synchronized the new network rules. The controller uses NBI APIs such as REST/RESTful, REST/RPC user-defined APIs and provides a set of specific components such as OpenStack/Quantum virtualization, Firewall, OFREST, etc. for SDN applications. RYU is distributed with multiple applications such as a simple L2 switch, router, tenant isolation, firewall, GRE tunneling, VLAN networking, etc. Ryu applications are single-threaded entities, which implement various functionalities.

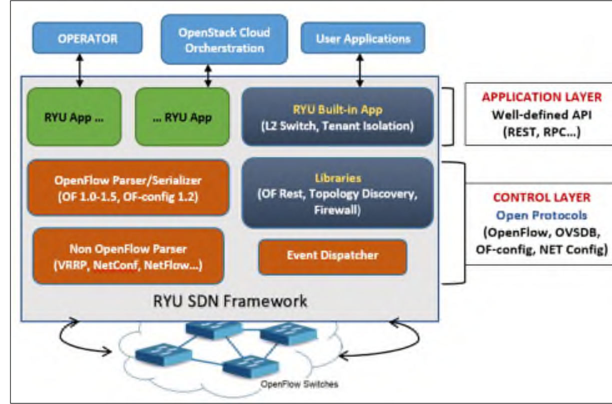

Figure 2: RYU Controller Architecture

The module called *ryu.controller.ofp_event* exports event classes that describe receptions of messages from connected OF switches. To preserve the order of events, each RYU application has a receive FIFO queue for events. The thread's main loop pops out events from the receive queue and calls the appropriate event handler. Hence, the event handler is called within the context of the event-processing thread, which works in a blocking fashion, i.e., when an event handler is given control, no further events for the RYU application will be processed until control is returned.

3. SIMULATION SETUP AND METRIC

The simulation hardware and software specifications are shown in Table 1. Controller scalability analysis were performed in an environment of a Windows 10 Operating System (OS). The VirtualBox 6.1.22 hypervisor is used to instantiate a Virtual Machine (VM) which is allocated 16 GB of RAM, and run on Xubuntu 20.04 VM contains Mininet with predefined Ryu controller software. Secure Virtual Machine Mode option must be enabled in BIOS setup due to supporting AMD-V virtualization mode.

Table 1: Simulation hardware and software specifications

		PC	Virtual Machine
Hardware	Processor	AMD Ryzen 5 3600, 3.6 GHz (6-core)	1 CPU, 6-core
	RAM	16 GB DDR4, 3200 MHz	8 GB DDR4
Software	OS	Windows 10, ver. 21H1	Xubuntu, 20.04.1 (64-bit)
	VirtualBox	-	6.1.22
	Mininet	-	2.3.0d6
	Ryu Controller	-	4.3.2
	Python	-	3.8.5

Simulation is being done on the software known as the Mininet emulation tool. Mininet is a Python-based and open-source network simulator for prototyping a large OpenFlow network. By default, Mininet creates the virtual switch in OpenFlow mode and supports various controllers including OpenFlow reference controller and OVS controller [15]. A virtual network is created according to specified links, hosts, switching devices and controllers. A Command Line Interface (CLI) is provided to interact with the virtual network, for example, checking connectivity between hosts, checking network flows, devices status monitoring etc. Mininet comes with built-in controller classes to support various controllers which include the OpenFlow reference controller. Also, Mininet contains a number of default topologies such as minimal, single, reversed, linear and tree, and provides an easy way to experiment with various custom network topologies. It is possible to develop and test code on Mininet, and OpenFlow controller or switch can simply implement into a real networking environment.
In this paper, we have considered Fat-Tree-Based custom topologies for simulation, each one with the various number of OpenFlow switches and hosts. Fat-Tree-Based topology was chosen for simplicity and to ensure that all switches are well-exercised and because it is the most popular in data centers, as shown in Figure 3. This topology has three switch layers including edge, aggregation, and core switches. Edge switches are directly connected to hosts in a way that N ports of the switch are connected to $N/2$ hosts and the rest are connected to higher level switches called the aggregation switches. In the same manner, the aggregation switches are connected to core switches [16]. All the OpenFlow switches and hosts are connected with each other in a hierarchical form. In this topology, there are several paths between every two hosts which increases fault tolerability.

Python 3.8.5 version is used to write the custom topology code. Mininet CLI is used to implement and test the topology. Due to the topology complexity, we used a Mininet high-level API. We use *Topo* as the base class that provides the ability to create parameterized network topology. We use methods *self.addSwitch()* and *self.addHost()* to import switches and hosts into topology and connect them. Method *self.addLink(node1, node2,**link_options)* is used for adding a bidirectional link that contains host and switch names and the number of options such as link bandwidth or delay. There are two classes available such as CPU Limited Host and TC Link that can be used for performance limiting and isolation. There number of ways that these classes may be used, but a simple way is to specify them as the default host and link classes to Mininet(), and then to apply the appropriate parameters in the topology.

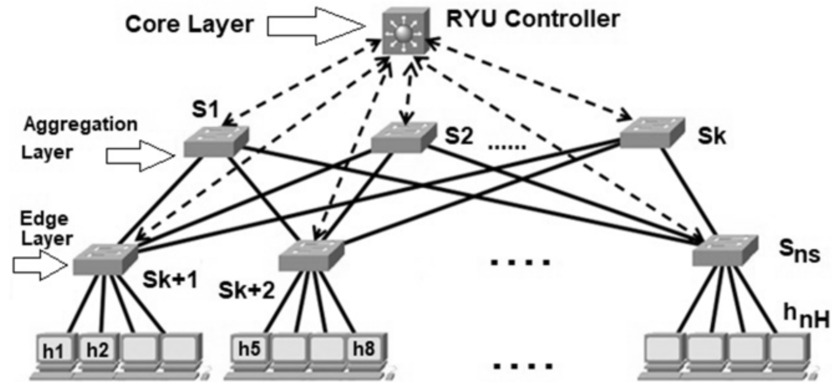

Figure 3: Fat-Tree-Based simulation network topology

To evaluate the statistics related to the scalability performance of Ryu controller, custom Fat-Tree topology is implemented with four scenarios having a difference in the number of OpenFlow switches and hosts connected to each edge-layer peripheral switch, as shown in Table 2. For each scenario, the total number of links in the topology is also shown.

Table 2: Network topology scenario table for simulation

Scenario	Number of switches	Number of hosts	Number of links
Scenario 1	6	8	16
Scenario 2	15	50	100
Scenario 3	30	200	400
Scenario 4	45	450	900

By Mininet all the hosts in topology have been assigned a unique IP address from 10.0.0.0/24 address range and a unique MAC address. The IP/MAC addresses are 10.0.0.1/00:00:00:00:00:01 for host *h1*, 10.0.0.2/00:00:00:00:00:02 for host *h2*, etc. To make OpenFlow switches connect to the RYU controller, we have used 127.0.0.1 virtual loopback IP address and 5566 port number.

The main objective of this paper is to evaluate the throughput parameter for TCP traffic in predefined SDN topology. The controller in SDN topology uses RYU handlers and decorators for sending OpenFlow messages between the nodes. We used the iPerf networking tool to test network throughput between the end hosts [17]. A typical iPerf output contains a timestamped report of the amount of data transferred through the network. In our throughput simulation test, the RYU controller is evaluated for the maximum amount of data it can process in a second between two end hosts which act as server and client. Making a TCP client-to-server connection, iPerf has executed in 50 sec. on the client host, and data have been collected every 1 sec. on the server host. TCP packets are sent from the client to the server host for a default 85.3 KB window size.

4. EXPERIMENTS AND EVALUATION

This section provides the results obtained during the experimentation and has four graphs that represent results for each scenario listed in Table 2. To evaluate the performance of Ryu controller in an experiment concerning scalability, throughput is the best matching parameter that will suffice our aim of the experiment. Consequently, in this section, we have limited the study to controller throughput only.

Graph of Fig. 4 (left) shows the results obtained by performing TCP data transmission between client and server for scenario 1. 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 is observed from the graph that the average throughput stays at 25.36 Gbps. The graph also

shows that the throughput variations are quite uniform within 50 sec of simulation, and are in the range between 25 and 26 Gbps. Most of the time simulation was running well, except for the intensive drop that occurs at 30 sec. of simulation.
Similarly, the controller scalability is disputable if we look at Fig. 4 (right) which demonstrates the TCP throughput graph of scenario 2, with 15 switches and 50 hosts. It is slightly worst as far as stability is concerned in comparison to scenario 1. As in the previous case, the simulation has executed in 50 sec. on the *h50* host, and data have been collected on the server *h1*. It is calculated from the graph that the average throughput stays at 24.36 Gbps, which is approximately 4% less than in the previous scenario. Significant throughput drops occur at 6 sec., 31 sec. and 38 sec. of simulation, leading to degraded performance of the simulation run.

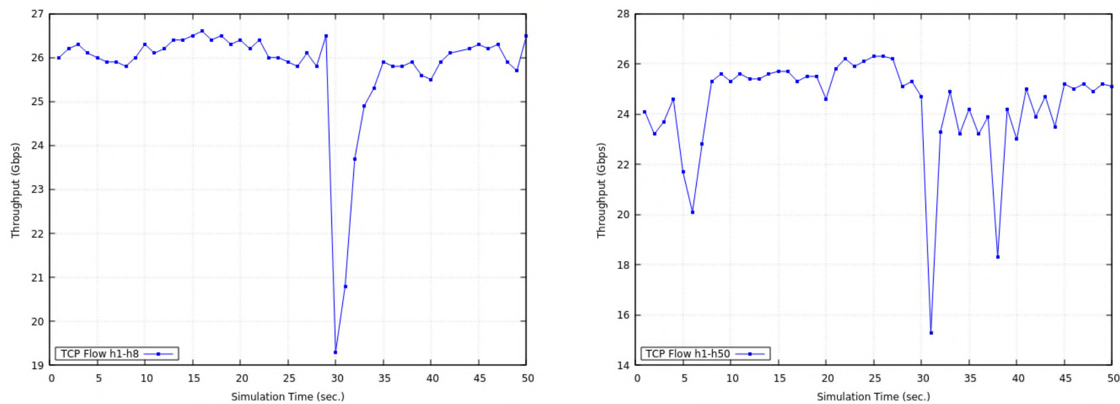

Figure 4: Throughput for topology scenario 1 (left) and scenario 2 (right)

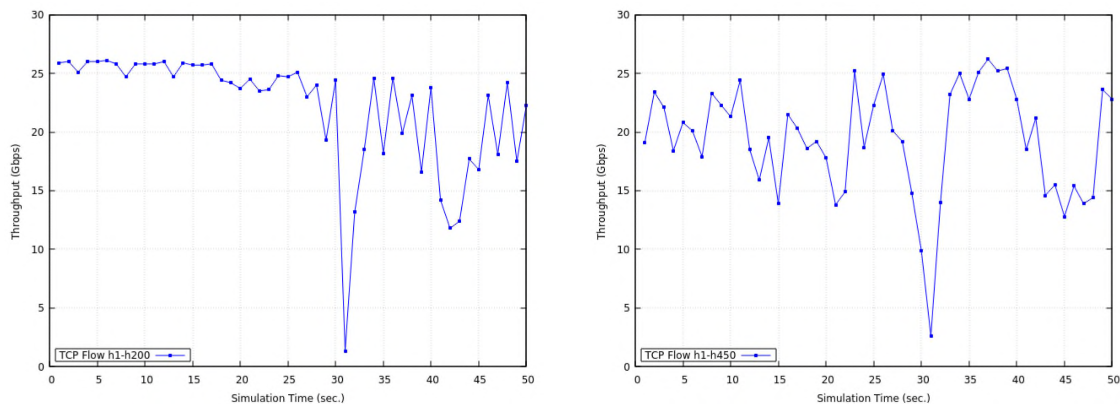

Figure 5: Throughput for topology scenario 3 (left) and scenario 4 (right)

Fig. 5 (left) shows the throughput of topology scenario 3, with 30 switches and 200 hosts. During the first half of the simulation, *h1-h200* TCP flow was running well but by the end of simulation multiple and intensive dropping instances were observed frequently leading to degraded throughput in the second half of the simulation run. As in the previous two scenarios, the simulation has executed in 50 sec. on the *h200* host, and data have been collected on the server *h1*. It was observed that the average throughput for this scenario was dropping and stays at 22.16 Gbps, which is 12.6% less than scenario 1 average throughput.
Fig. 5 (right) presents TCP throughput results for scenario 4 with 45 switches and 450 hosts. As can be seen from the graph, there are intensive throughput variations during all the simulation time. The average throughput for scenario 4 is 19.34 Gbps, which is 23.7% less than scenario 1 average throughput. A large average throughput drop of the controller indicates its overload in a situation where exist SDN network with a significantly larger number of switches and hosts.
The concern with a proposed SDN topology with a large number of switches and hosts (scenarios 3 and 4) is that this requires the RYU controller to send *packet_out* messages to every host and processing packet_in messages coming from every host for each link in both directions. With these graphs shown from Fig. 4 and Fig. 5, it was observed that RYU is a very much resource demanding controller which uses CPU and RAM utilization to optimum and thus results in a degraded performance in presence of an increasing number of switches and hosts. We limited the study to 45 switches and 450 hosts because even AMD Ryzen 5 3.6 GHz processor with 16GB RAM was proved not enough during the execution of experimentation. We decided to do the same simulation with 60 switches and 800 hosts, but Mininet always crashes. We tried to repeat this simulation three more times to confirm whether this phenomenon is accidental or not, and we always got the same results.

From the obtained scenarios results, the conclusion is that the continuous polling of data causes overload on the RYU controller. This is because having a large number of OpenFlow switches causes conflict at the data plane which demands high processing power.
Another important factor is to observe the hop count number of the path between network nodes. Again, a test was performed at a scale with 800 hosts and 60 switches. This was scaled up to 20.000 paths with intensive degradation in controller performance. The RYU controller was not able to handle significantly large *packet_in* rates. This type of data shows that the RYU controller is more suitable for delay-sensitive applications, as well as for less complex SDN networks with a smaller number of OpenFlow switches and hosts.

5. CONCLUSIONS AND FUTURE WORKS

While the SDN architecture appears to solve problems within the traditional TCP/IP network architecture, it also comes with some major challenges. In this paper, we highlight one of these challenges, which is related to controller scalability. With this paper, the authors have attempted to address the scalability features of the RYU controller by implementing various topology scenarios in a simulation environment. The authors have provided a clear procedure of how to create an experimental testbed along with analysis of obtained statistical results, keeping the throughput performance as the central focus.
The evaluation focused on the TCP flows throughput between end hosts and *packet-in/packet-out* messages that are generated when a controller receives packets from hosts. As it was insufficiently unclear how the SDN topology affects the controller scalability, four different topology scenarios were studied. These topologies vary with respect to their link density as well as with the number of switches and hosts present in the network (maximum number of switches, hosts and links are 60, 450 and 900 respectively). From the throughput measurements of the TCP flows, the throughput values decreased as the number of switches and hosts are increased in the network. For a scenario 4 topology with 45 switches and 450 hosts, the RYU controller shows 23.7% lower throughput than scenario 1 with only 6 switches and 8 hosts. The difference in throughput behavior for the RYU controller can be caused by two factors: the first one is the algorithm that a controller uses to distribute the messages between controller and hosts, and the second one is the mechanisms and libraries used for the interaction between the network and the controller. The obtained experimental results show that RYU controller exhibited better results for less complex SDN networks.
Future works will expand on experimental cases for many other research directions. We plan to keep extending this scalability research with some other SBI APIs, Java-based and multithreaded controllers, and large-domains clustering with multi-controllers.

REFERENCES

[1] V. Nguyen, A. Brunstrom, K. Grinnemo, J. Taheri, “SDN/NFV- Based Mobile Packet Core Network Architectures: A Survey,” IEEE Communications Surveys Tutorials, vol. 19, no. 3, pp. 1567-1602, 2017. doi: 10.1109/COMST.2017.2690823

[2] M. Jarschel, T. Zinner, T. Hossfeld, P. Tran-Gia, and W. Kellerer, “Interfaces, attributes, and use cases: A compass for SDN,” Communications Magazine, IEEE, vol. 52, no. 6, pp. 210-217, 2014. doi: 10.1109/MCOM.2014.6829966

[3] ONF Foundation: “OpenFlow Switch Specification”, Version 1.5.1 (Protocol version 0x06), 2015, [Online] https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[4] W. Xia, Y. Wen, C. H. Foh, D. Niyato, H. Xie, “A survey on software defined networking,” IEEE Communications Surveys & Tutorials, vol. 17, no. 1, pp. 27-51, 2015. https://doi.org/10.1007/s10922-016-9393-9

[5] W. Zhou, Li Li, Min Luo, Wu 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] N. Feamster, J. Rexford, E Zegura, “The road to SDN: An intellectual history of programmable networks”, Computer Communication Review, Vol. 44, Issue 2, pp. 87-97, 2014, https://doi.org/10.1145/2602204.2602219

[7] D. B. Rawat, S. R. Reddy, “Software-Defined Networking Architecture, Security and Energy Efficiency: A Survey”, IEEE Comm. Surveys and Tutorials, Vol. 19, Issue 1, pp. 325–346. 2017, doi:

[8] M. P. Fernandez, "Comparing OpenFlow controller paradigms scalability: Reactive and proactive", AINA IEEE 27th International Conference, pp. 1009-1016, 2013. doi: 10.1109/AINA.2013.113

[9] 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

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

[11] OpenFlow Switch Specification, Version 1.5.1 (Protocol version 0x06). Open Networking Foundation, 2015. [Online] https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf
[12] Ryu SDN Controller. Available from: https://osrg.github.io/ryu [Online] https://github.com/faucetsdn/ryu
[13] Network Configuration Protocol (NETCONF), Internet Engineering Task Force (IETF) RFC 6241, [Online] https://datatracker.ietf.org/doc/html/rfc6241
[14] OpenFlow Management and Configuration Protocol (OF-Config 1.1.1). Open Networking Foundation, 2013. [Online] https://opennetworking.org/wp-content/uploads/2013/02/of-config-1-1-1.pdf
[15] J. Yan, D. Jin, "VT-Mininet: Virtual-time-enabled Mininet for Scalable and Accurate Software-Define Network Emulation", Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, Article No. 27, pp: 1-7 https://doi.org/10.1145/2774993.2775012
[16] Li Yu, Pan Deng, "OpenFlow based load balancing for Fat-Tree networks with multipath support", Proceedings 12th IEEE International Conference on Communications (ICC13), pp.1-5, 2013
[17] iPerf 3 User Documentation [Online] https://iperf.fr/iperf-doc.php#3doc