Multi-path Load Balancing for SDN Data Plane

M.C. Nkosi, A.A. Lysko, S. Dlamini

CSIR Meraka Institute, Council for
Scientific and Industrial Research
P.O. Box 395
Pretoria 0001, South Africa
{mnkosi2, alysko}@csir.co.za

Abstract— **Networks have become an important feature of our day-to-day to life and therefore, user experience is an imperative goal to be achieved by network operators. Load balancing is a method of improving network performance, availability, minimizing delays and avoiding network congestion. In this paper, we study dynamic load balancing to improve network performance and reduce network response time. The load balancer is applied to OpenFlow SDN network's data plane with Opendaylight as the controller. The flexibility of the load balancer is tested by using it on two different network topologies. Results show that the load balancer can improve the overall performance of the network and reduce delay. The main contribution of this work is a load balancing mechanism for SDN centralized controller environments which can be employed at any point in time in a network, for example, before network failure or after link failure, to avoid data plane congestion and link overloading.**

Keywords—SDN, OpenFlow, Data Plane, Controller, Load balancing

I. INTRODUCTION

Software Defined Networking (SDN) is a key enabling networking paradigm for 5G communications which promises to simplify network management, reduce operational costs, reduce network resource utilization, and stimulate innovation of new services and network evolution [1]. The core function of SDN is to separate the control plane from the data plane by bringing about network control functions based on an abstract representation of the network. The network functions are implemented by removing control decisions, such as routing, from hardware devices and enabling programmable flow tables in the hardware using a standardized protocol, such as OpenFlow (OF). The network abstraction is achieved through the use of a logically centralized SDN controller which defines the behavior of the data plane. When a new flow is initiated in the data plane and there is no routing policy that exists within the flow table, the forwarding device sends the first packet of that flow to the controller. The controller then defines the relevant forwarding route for that flow. One forwarding policy is usually defined for each flow. Two or more forwarding policies maybe defined for a single flow only for backup purposes. Therefore, there are too many packets using a particular flow, that flow may experience overloading and cause delays and the overall network efficiency. Thus, the ever increasing volume of network traffic requires efficient traffic engineering and management methods to ensure availability, scalability and reliability of network.

Load balancing is a method of managing incoming traffic by distributing and sharing the load fairly among available network resources to improve network availability, to reduce latency and bandwidth utilization. In legacy networks, a load balancer is a device designed on a particular hardware [2]. The load balancers are costly and differ based on vendor. In SDN, load balancers are program codes which can easily be implemented on the SDN controller to efficiently manage network load. Most SDN controllers, such as, for example, Onos, Ryu, Pox, floodlight, come with built in static network load balancers with predefined load balancing policies such as, for example, Round Robin, and Random. Mostly, the load balancing in SDN controller is implemented using two approaches, namely, the stateless and the state-full load balancing [3]–[4]. In stateless approach, a controller does not monitor the state of the network and will only load balance traffic at a particular predefined time. This method works well when traffic demands are known in advance. In state-full approach, the controller keeps track of the state of the network and performs load balancing as required. Although state-full approach is preferred, it is expensive because the controller should maintain per-flow states for flow table [5]. This may sometimes lead to controller over loading, which may ultimately cause network unavailability and delays.

In this paper, a dynamic flow load balancer is used to define alternative flows for a flow and then traffic load traversing a flow is shared equally among alternative flows. Load balancing is performed as required to relive the controller from maintaining per-flow states table. Opendaylight is used as the centralized SDN controller. In particular, the objective of this work is to compare the performance of Opendaylight built in load balancer, which is based on round robin and random policies, with a dynamic load balancer. The dynamic load balancer uses Dijkstra algorithm to calculate shortest alternative routes for each flow and balances traffic amongst the defined alternative flows. The flexibility of the load balancer is tested by using different network two different topologies. Mininet emulation tool is used to simulate network topologies, iperf is used to generate traffic load and Wireshark is used monitor the performance of the emulated network.

The rest of the paper is organized as follows. Section two provides related work. Section three presents system setup and methodology. Section four describes results. Section five concludes this paper.

II. RELATED WORK

SDNs, by their nature, accumulate a lot of control traffic due to control plane operations and signaling events that

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©2018 IEEE

must be addressed efficiently to ensure effective and resilient networks.vWithout efficient management methods, the control traffic may overwhelm the controller, more particularly in a centralized controller network, and cause delay and loss of information [1],[6]. Work focusing on control plane operations management where controller traffic load balancing is addressed to ensure efficient SDN control plane operation, is discussed in [6]-[7]. Proper management of data plane traffic can relief controller from suffering from overloading. Existing works focus on data plane load balancing to address load balancing after network failures such as link failure, switch failure and less on dynamic load balancing helpful in avoiding network delays and traffic congestion [8]-[9].vWang et al. proposed a path load balancing mechanism which balances traffic after link failure in the data plane [8]. Adami et al, [9], introduced a class-based traffic recovery load balancing method to ensure resilient network. In addition, several recent studies have focused on dynamic load balancing in the data plane from various perspectives [4],[10]-[11].vKhan et. al, studied dynamic load balancing based on traffic volume by monitoring link usage and load balancing traffic among available links to avoid link over loading [4].vIn their method, Floodlight was used as a centralized controller. Gupta et. al studied flow statistics based load balancing using POX controller [10]. In their method, load balancing is performed to avoid server overloading by fairly sharing server connection requests among multiple servers. Mallik et. al, introduced a multi-path congestion control with load balancing to try to protect network from congestions caused by load spikes [11].

All these studies concern load balancing in the data plane using Floodlight or POX as the SDN controller. Different from the existing solutions, we investigate the performance of multipath-based load balancing in a single centralized Opendaylight controller's data plane. The multi-path dynamic load balancing method defines alternative paths for a flow and shares a flow's load among the alternatives paths.

III. SYSTEM SETUP AND METHODOLOGY

Software defined networks (SDNs) consist of a controller and a set of OpenFlow switches. Whenever a new traffic flow request enters a switch, the switch sends a routing request to the controller. After receiving the request message, the controller calculates an optimal path and configures routing tables in all the switches along the optimal path. As shown in Fig. 1, each switch consists of a group table and a set of flow tables with associated action set.

A flow table is made up of flow entries which are flow routes for each source and destination pair. Thus, a single

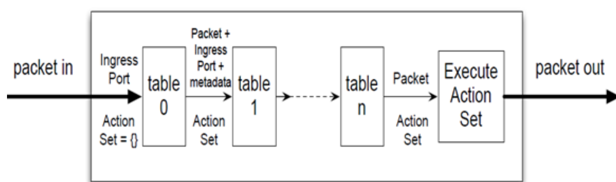

Fig. 1: typical components of an OpenFlow switch

Flow is defined for each source destination pair and all traffic for each pair will use the same route irrespective of how much congested that flow is.

In this study, two scenarios with different network topologies are considered. As depicted in Fig. 2, the first scenario is made up of a network with 7 OpenFlow-switches (OF-switches) and 8 hosts. The second scenario, as shown in Fig. 3, is a bigger network with 40 switches and 81 hosts. The two scenarios were chosen to test the performance, scalability and flexibility of the proposed load balancing method.

Nitrogen version of the open source Opendaylight SDN controller was used as the centralized controller in both the scenarios. Opendaylight is developed by Linux foundation to promote SDN. It is modular, scalable and supports a variety of southbound interfaces such as, other than OpenFlow, the broadband gateway protocol, OVSDB and many others[12]-[13]. Karaf Dlux features [14] were used to monitor the topologies, nodes and controller-switch communications.

Mininet SDN emulation tool was used to emulate the network topologies. Mininet uses a single kernel to run the emulated topologies and employs Open Virtual Switch (OvSwitch) as the default OpenFlow switch[15] . OpenFlow version 1.3 was used because it is still the most supported version in SDN hardware switches.

A source-destination pair was identified in each topology and pings were performed to generate congestion between the source and destination pair. Irrespective of how much the path link flow is congested, new incoming traffic of same source-destination pair is queued on one same data flow path. This at times, may cause delay and ultimately loss of information. To avoid this problem, multi-flow load balancing method was used.

The multi-flow load balancing method calculates alternative short paths which are pushed down into the flow table. A traffic load for a single flow was shared fairly among the alternative flows. The load balancing algorithm takes source and destination pair as an input. The algorithm extracts network topology using JSON and REST APIs and performs link, port, MAC, and IP mappings together with switch and port connections. The algorithm also extracts ports transmission rates statistics to understand the load on each port for each flow. Possible best alternative paths are chosen based on lowest flow cost. Flow cost is calculated as the sum of number of transmitted and received packets at that particular time. PC with Linux Ubuntu 18.04 with 8GB RAM and 2.7GHz processing speed was used to implement this study. IPerf was used to create TCP data streams and to measure the throughput of the data flow before load balancing and after load balancing.

In a nutshell, the following is the step by step implementation methodology for our study for each topology scenario:

- Emulate a network topology using mininet and run mininet *ping all* to ensure that nodes and links are up and running.
- Identify data flow path for a source destination pair.
- Verify that the path is indeed used for transmission for the source-destination pair using Wireshark.

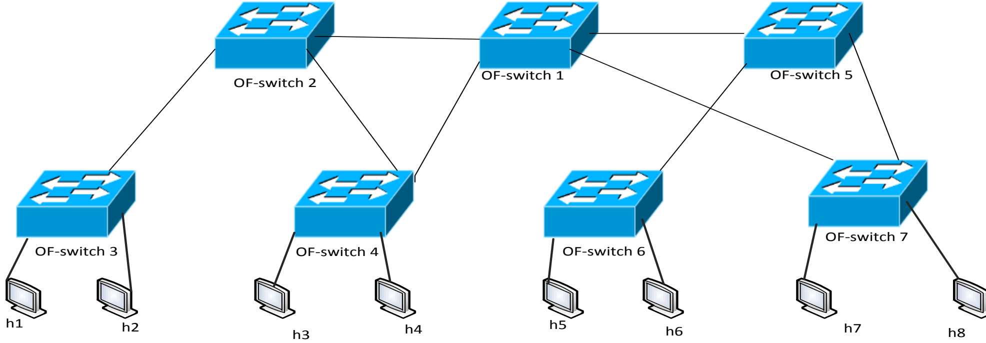

Fig. 2: network topology for scenario 1

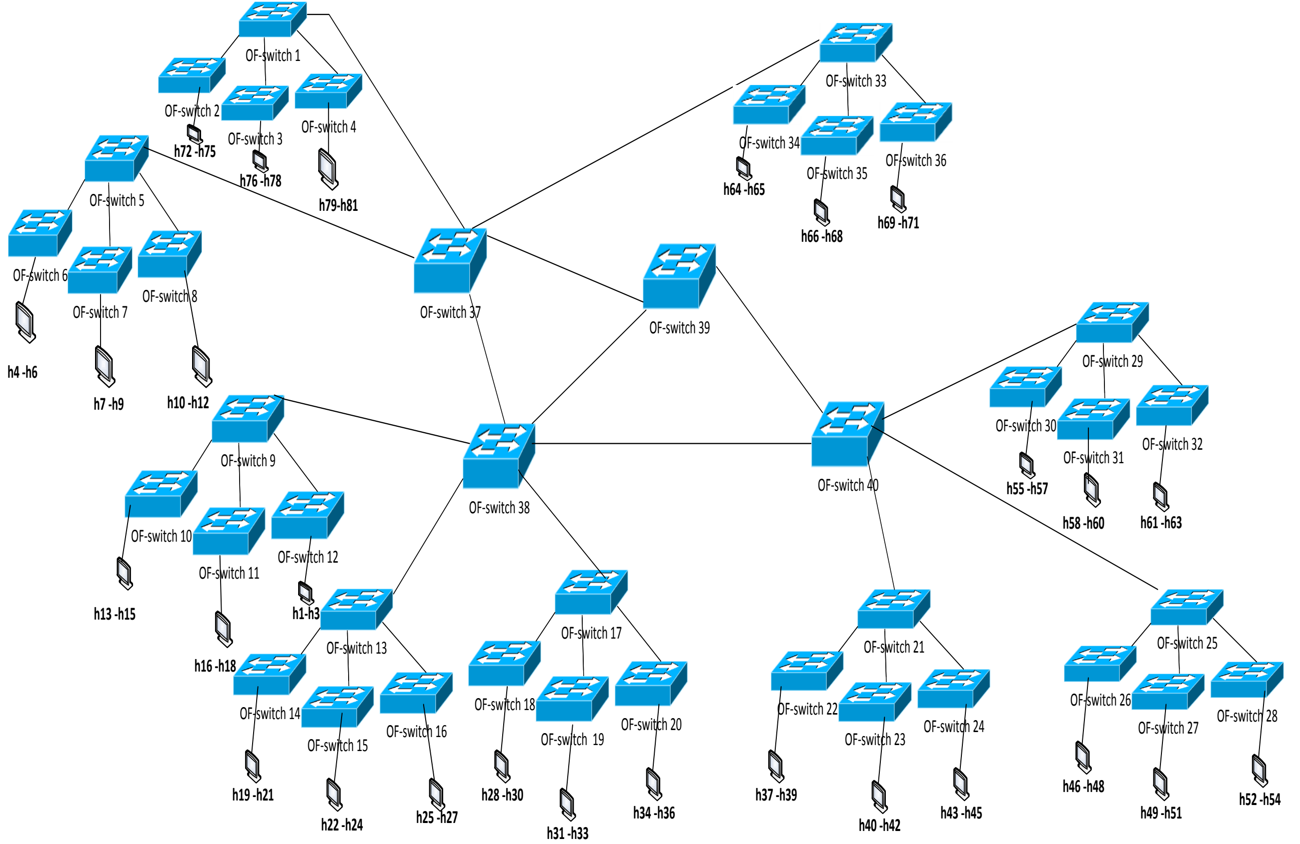

Fig. 3: network topology for scenario 2

- Create a ping on the source-destination pair to generate traffic and flow congestion.
- Perform an iperf and another ping to measure the latency and bandwidth utilization on the over loaded flow. The ping is performed for 10 packets each with packet size 10240 MB. The iperf test was performed using default TCP window frames of size 85.3 KByte with time interval of 15 sec.
- Perform load balancing on the congested source-destination pair (i.e. data path flow)
- Perform an iperf and ping again on the source-destination pair to measure the performance of the load balancing.

Results for both scenarios based on the methodology are highlighted in the next section.

IV. RESULTS AND DISCUSSIONS

A. Scenario 1

As shown in Fig. 2, scenario 1 was made of a smaller network with a total of seven switches and eight hosts. A mininet ping all was performed to ensure 100% reachability. H1-h8 was chosen as the source-destination pair. Using Dijkstra's shortest path method, it was calculated that the best path for the pair is [S3-S2-S1-S5-S7]. It was also verified using Wireshark that transmission for the pair use the flow route [S3-S2-S1-S5-S7]. Using mininet xterm, a ping was performed for h1-h8 (i.e., on xterm h1, ping 10.0.0.8 (h1's IP address), and on xterm h8, ping 10.0.0.1(h1's IP address)). After congesting the best path for the source-destination pair, that is the flow [S3-S2-S1-S5-S7], another ping and an iperf with h1 as the client and h8 as the server, were performed to measure bandwidth utilization on the flow before load balancing. The ping results are shown in Fig. 4 and the iperf results are shown in Fig 5.

Load balancing was performed for the source-destination pair. The load balancing algorithm first computes all the flow paths for h1-h8 pair. The following was defined as the paths for the pair h1-h8: [S3-S2-S4-S1-S7], [S3-S2-S1-S5-S7]. The algorithm then computes path costs for all the defined paths by using network statistics. The path cost is calculated as: Cost= Tx +Rx where Tx is number of transmitted packets and Rx is number of received packed. The costs for the defined paths were calculated as [S3-S2-S4-S1-S7:0], [S3-S2-S1-S5-S7:8]. The path with lowest cost was chosen as the shortest path flow. That is to say [S3-S2-S4-S1-S7:1] was chosen and was pushed down to flow routing table as the flow to be used. Load balancing is repeated until path cost for all paths are equal (that is all paths will have same load).

Ping and iperf tests were performed after load balancing to measure bandwidth utilization after the ping. Results are shown in Fig 4 and Fig 5 respectively.

B. *Scenario 2*

As shown in Fig. 3, scenario 2 was made of bigger network with a total of 40 switches and 81 hosts. Same process as described in section iv.A, was followed for scenario 2. The *h7-h54 source*-destination pair was used with the path flow *[S5-S37-S39-S40-S25]* as the defined default flow by Opendaylight controller. Ping and iperf results are shown in Fig. 6 and Fig. 7 respectively.

For load balancing, the following were defined as the possible path flow with associated costs:

*[S5-S37-S39-S40-S25: **4**]; [S5-S37-S38-S40-S25:**0**];*

*[S5-S37-S38-S39-S40-S25:**0**];[S5-S37-S39-S38-S40-S25:**0**].*

Path *[S5-S37-S38-S40-S25]* was chosen as the new path and load balancing was repeated until load was fairly distributed among the alternative path flows. Ping and iperf results are also shown in Fig. 6 and Fig. 7 respectively.

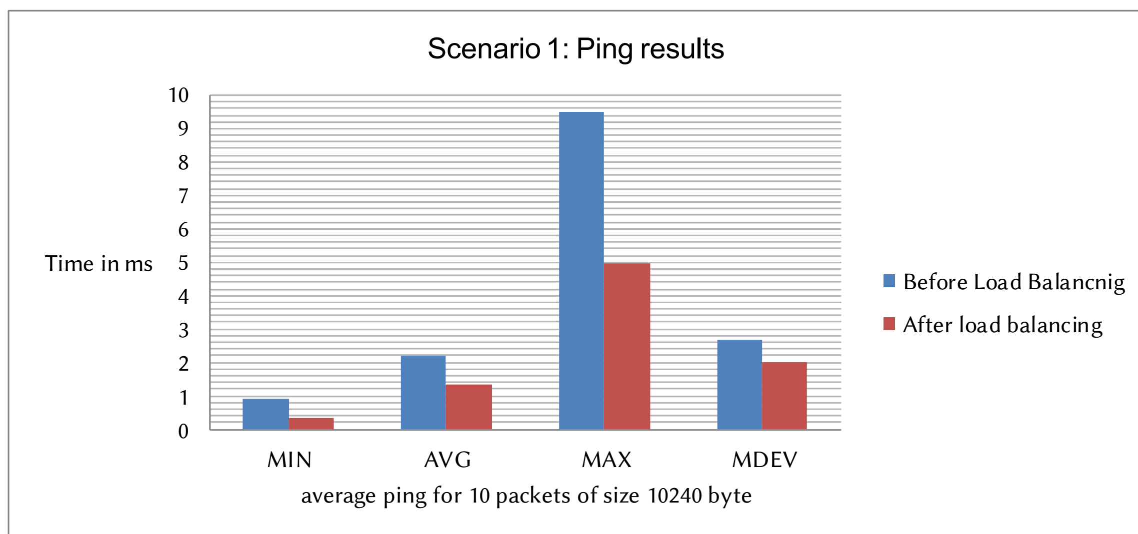

Fig. 4: Scenario 1 ping results for source-destination pair: h1-h8

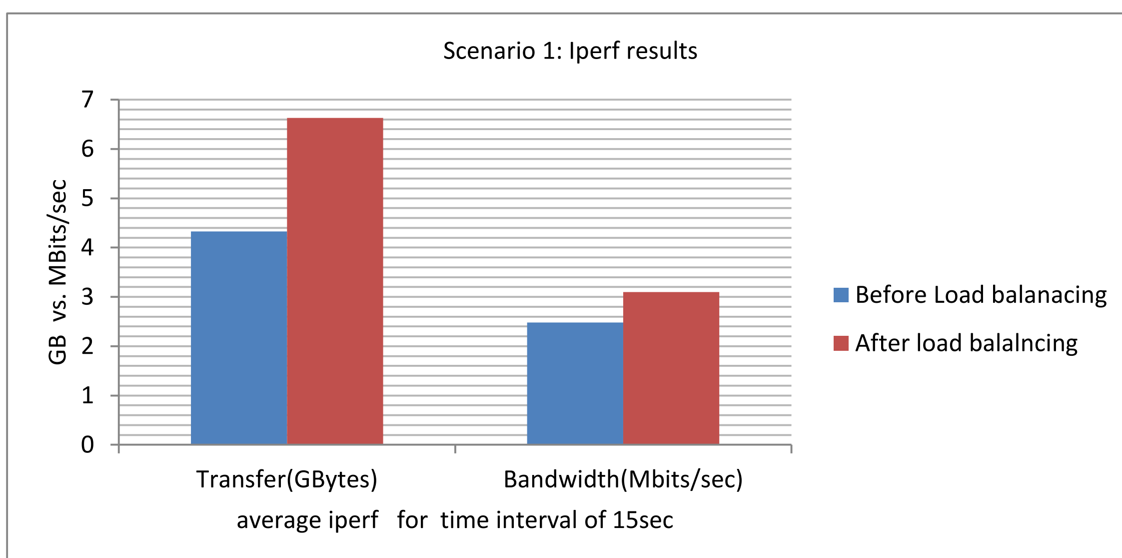

Fig. 5: Scenario 1 Iperf results for source destination pair: h1-h8

For load balancing, the following were defined as the possible path flow with associated costs:

*[S5-S37-S39-S40-S25: **4**]; [S5-S37-S38-S40-S25:**0**];*

*[S5-S37-S38-S39-S40-S25:**0**]; [S5-S37-S39-S38-S40-S25:**0**]*.Path *[S5-S37-S38-S40-S25]* was chosen as the new path and load balancing was repeated until load was fairly distributed among the alternative path flows. Ping and iperf results are also shown in Fig. 6 and Fig. 7 respectively.

C. Discussions

As it can be observed from the results of scenario 1, the maximum average ping after load balancing is 4.9m/s as compared to that of before load balancing which is 10m/s. The iperf results show an increase of about 2.2GB in data transfer after load balancing. However, the increase is not as significant as expected. This is assumed to be due to the fact that part of the alternative path routing path uses some link from the congested path. Notice that, the first links in both paths are the same: [S3-S2-S1-S5-S7] [S3-S2-S1-S5-S7]. From scenario 2, it can be observed that the increase is significant in data transfer, from about 4GB to 10 GB.

This is because, unlike, in scenario 1, there are multiple different routing paths that the load balancer can use to transfer data. Therefore, the load balancer works much quicker when there are more options in alternative paths.

From scenario 1 and 2, it can be concluded that the load balancer is flexible for both smaller networks and larger networks. It also can improve network performance and avoid overall network delay.

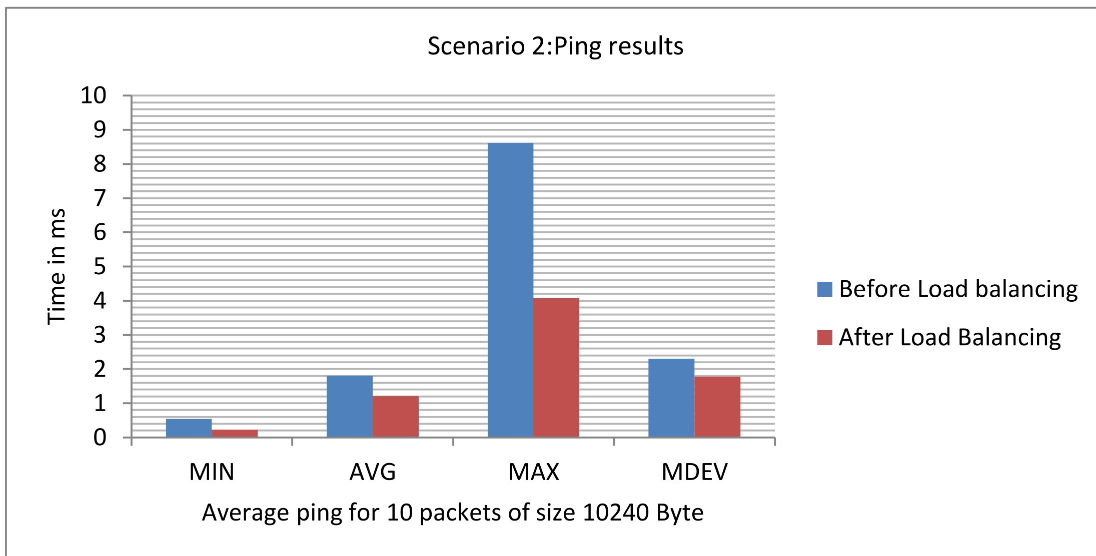

Fig. 6: Scenario 2 ping results for source-destination pair: h7-h54

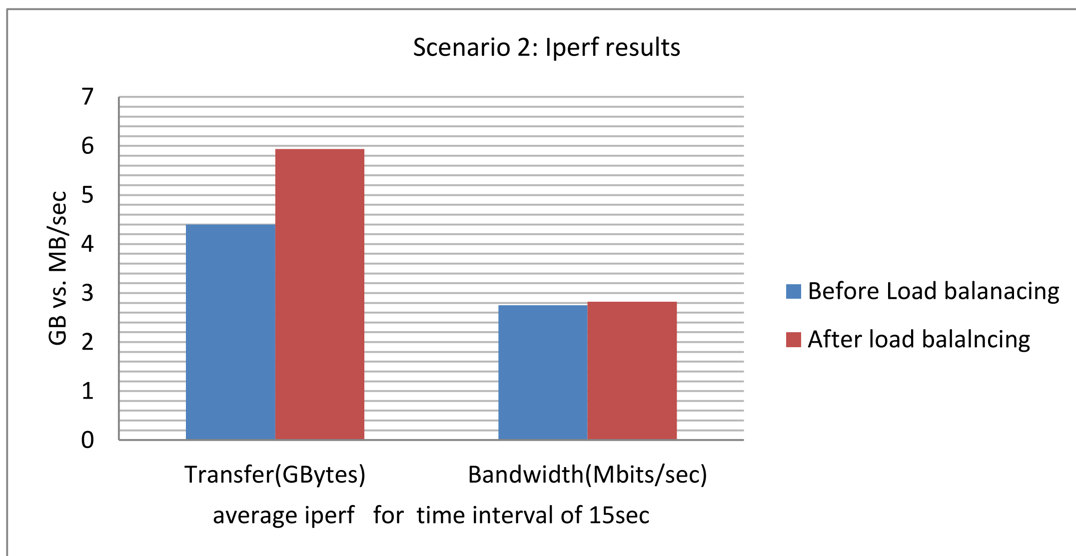

Fig. 7: Scenario 2 Iperf results for source destination pair: h7-h54

V. CONCLUSION

We have performed load balancing for both larger and smaller networks using a dynamic multi-path load balancer in an OpenFlow based SDN data plane. The load balancer determines alternative paths first and then reroute traffic equally amongst the defined paths. The network topologies were emulated using mininet emulation tool. The network uses SDN Opendaylight controller which uses built in load balancer based on round robin and random policies. The performance of the built in load balancer was compared to the multi-path load balancer method. The load balancer has improved the overall network performance in transfer rate and response time. However, it was found that for better network improvement, the data plane should have multiple alternative links so that multiple path paths can be defined for a routing path. The overall contribution of this work is a multi-path load balancing method, which, unlike other load balancing method, can be applied to a network at any state(before date plane failure, or after data plane failure), to ensure network efficiency.

REFERENCES

[1] G. Bianchi, A. Capone, M. Bonola, G. Bianchi, and M. Bonola, “Public Review for OpenState : Programming Platform-independent Stateful OpenFlow Applications Inside the Switch a c m s i g c o m m OpenState : Programming Platform-independent Stateful OpenFlow Applications Inside the Switch,” vol. 44, no. 2, pp. 44–51, 2014.

[2] W. K. Soo, T.-C. Ling, A. H. Maw, and S. T. Win, “Survey on load-balancing methods in 802.11 infrastructure mode wireless networks for improving quality of service,” *ACM Comput. Surv.*, vol. 51, no. 2, 2018

[3] S. Bhandarkar and K. A. Khan, “Load Balancing in Software-defined Network (SDN) Based on Traffic Volume,” vol. 2, no. 7, pp. 72–76, 2015

[4] “US9621642B2 - Methods of foMethods of forwarding data packets using transient tables and related load balancers.”

[5] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, “On scalability of software-defined networking,” *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 136–141, 2013.

[6] T. Kim, S. G. Choi, J. Myung, and C. G. Lim, “Load balancing on distributed datastore in opendaylight SDN controller cluster,” *2017 IEEE Conf. Netw. Softwarization Softwarization Sustain. a Hyper-Connected World en Route to 5G, NetSoft 2017*, 2017

[7] J. Li, X. Chang, Y. Ren, Z. Zhang, and G. Wang, “An effective path load balancing mechanism based on SDN,” *Proc. - 2014 IEEE 13th Int. Conf. Trust. Secur. Priv. Comput. Commun. Trust. 2014*, pp. 527–533, 2015

[8] D. Adami, S. Giordano, M. Pagano, and N. Santinelli, “Class-based traffic recovery with load balancing in software-defined networks,” *2014 IEEE Globecom Work. GC Wkshps 2014*, pp. 161–165, 2014.

[9] K. Kaur, S. Kaur, and V. Gupta, “Flow statistics based load balancing in OpenFlow,” *2016 Int. Conf. Adv. Comput. Commun. Informatics*, pp. 378–381, 2016

[10] A. Mallik and S. Hegde, “A novel proposal to effectively combine multipath data forwarding for data center networks with congestion control and load balancing using Software-Defined Networking Approach,” *2014 Int. Conf. Recent Trends Inf. Technol. ICRTIT 2014*, 2014

[11] Z. K. Khattak, M. Awais, and A. Iqbal, “Performance evaluation of OpenDaylight SDN controller,” *Proc. Int. Conf. Parallel Distrib. Syst. - ICPADS*, vol. 2015–April, pp. 671–676, 2014.

[12] P. Berde *et al.*, “Facilitation of the OpenDaylight Architecture,” *Proc. third Work. Hot Top. Softw. Defin. Netw. - HotSDN ’14*, pp. 1–6, 2014.

[13] S. Badotra, “Open Daylight as a Controller for Software Defined Networking,” vol. 8, no. 5, pp. 1105–1111, 2017.

[14] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, “Reproducible network experiments using container-based emulation,” *Proc. 8th Int. Conf. Emerg. Netw. Exp. Technol. - Conex. ’12*, p. 253, 2012.

[15] S. Badotra, “Open Daylight as a Controller for Software Defined Networking,” vol. 8, no. 5, pp. 1105–1111, 2017.