A Dynamic Load Balancing Algorithm for Data Plane Traffic

Lucas Soares da Silva*, Carlos Renato Storck† and Fátima de L. P. Duarte-Figueiredo*
*Pontifical Catholic University of Minas Gerais
Belo Horizonte, Minas Gerais – 31980-110
Email: lucassoares86@gmail.com, fatimafig@pucminas.br
†Federal Center for Technological Education of Minas Gerais
Contagem, Minas Gerais – 32146-054
Email: storck@cefetmg.br

***Abstract*—There is a worldwide growing demand for high bandwidth, low latency and reliability. Software Defined Network (SDN) emerged a few years ago as a new paradigm, separating the control and the data planes to improve the network management. The SDN concept applies to any type of network and is indispensable in new generations of networks such as 5G. One of the main problems in the data plane is the distribution of traffic. Quality of service (QoS) characteristics such as throughput and latency of each application can be affected by inefficient link balancing. This paper proposes an algorithm for dynamic load balancing the data plane traffic. The load balancing minimizes the effects of intense network data traffic, mitigating bottlenecks. The proposed algorithm dynamically changes the links flows as the network utilization intensifies. The algorithm finds the shortest paths and calculates the link cost choosing the best link after identifying and kind of bottlenecks. This reduces the latency and the packet loss in scenarios with network congestion. Experiments were conducted using the Mininet and the OpenDayLight controller. Through emulation results, it was observed that there was a significant improvement in jitter, packet loss, and throughput. Thus, the proposed algorithm can work in any kind of SDN environment, balancing the data plane traffic flow.**

***Index Terms*—Dynamic Load Balancing, Data Plane, SDN**

I. Introduction

The separation of the data plane and the control plane, centralizing the control and allowing network programmability, are the SDN goals. Among the SDN main benefits it can be highlighted the network management policies centralization and the resources dynamic allocation, according to the demand [1]. On SDN-based networks, the devices have flow tables that are filled by the controller, then the packets flow through the network [2]. There are two ways of operation: Reactive and Proactive. In the reactive, packets are sent from network equipment to the controller make the decisions. In the proactive way, rules are predefined on network equipment, and packets do not need to be sent to the controller. This model allows easy addition of new control policies, because the network becomes programmable and the decision-making is centralized in the controller [3].

With the emergence of SDN's, new issues related to availability also emerged. Maintaining network availability requires efficient load balancing that deals with dynamic topologies. In a dynamic environment, a load balancing algorithm that efficiently manages constant topology changes is indispensable.

In the model proposed by the SDN's, where there is centralization of network control, the load balancing algorithms of legacy networks lose their efficiency. In this new scenario, it is necessary to implement approaches to deal with load balancing and ensure the efficiency of SDN's. As new network technologies will utilize SDN, efficient load balancing is required to avoid bottlenecks.

This paper presents a load-balancing algorithm. It acts reactively in the data plane, bypassing the flows as it detects overload in the network equipment. Each possible path of the flow to be desviated is executed by algorithm proposed and checks the cost between the links, defining the best path for flow deviation. The OpenDayLight controller and the Mininet network emulator were used to evaluate the proposed algorithm [4]. The algorithm developed uses information from network equipments, through the controller, to identify bottlenecks in the data plane and to redefine the route of packets where there is a large data flow, taking into account the best path at the time of transmission. The experiments results show that the proposed algorithm promotes load balancing and consequently reduces congestion at bottleneck points in the network.

This paper is organized as follows: Section 2 presents the related work; Section 3 presents the proposed approach for dynamic load balancing in SDN; Section 4 describes the methodology adopted; Section 5 presents the results obtained; and Section 6 discusses the conclusions and propositions for future work.

II. Related Work

In the literature, load balancing proposals are found in SDN networks in the data plane [5]. Among them, part proposes to solve the problem of overloading servers and another part proposes the diversion of traffic to the shortest path, with the objective of avoiding overload and improving the data flow in the network equipment, mitigating possible bottlenecks. Previous works about the load balancing problem in SDN networks, used in this work, are summarized in Table I.

978-3-903176-23-2 © 2019 IFIP

TABLE I
LOAD BALANCING IN SDN NETWORKS

Work	Brief Description and features
Lan et al. [6]	It performs load balancing on data center networks, but does not demonstrate its operation in other network environments.
Li et al. [7]	Rules are installed by the controller with dynamically adjusted paths according to the network global view. The work does not present results of the algorithm in dynamic networks.
Belyaev et al. [8]	It provides a solution for DDoS attacks on servers, but does not ensure the the entire network balance taking into account its equipments.
Fizi and Askar [9]	It provides algorithm for the load balancing in datacenters using the POX controller in a fat-tree topology.
Koryachko et al. [10]	It provides dynamic balance proposal in SDN with QoS and with weight assignment according to the link´s flow and delay.
Liu and Xiang [11]	The main proposal approach is to find the link that be overloaded and, transfer it to the idle link on a shorter distance.
Zakia and Yedder [12]	It provides algorithm that aims to optimize the link use in data center networks. Tests in dynamic environments were not considered.

In Lan et al. it was proposed a dynamic path optimization algorithm in SDN networks that may be suitable for different network topologies. This algorithm change the flow path during transmission to perform load balancing and to solve the congestion problem in SDN based data center networks [6]. In Li et al. a fuzzy synthetic evaluation mechanism (FSEM) was presented as an SDN based load balancing solution. In this model, network traffic is allocated to the path operated by the network device where rules are installed by the controller. Paths can be dynamically adjusted by the FSEM according to the network overview [7]. For Belyaev et al. it is necessary to mitigate the effects of DDoS attacks. A two-tier solution was proposed which includes traditional server balancing and balancing between network devices. In the proposal, the network survival time in DDoS attacks is higher than the load balancing solutions already proposed in the literature for the SDN networks [8].

In [9] an algorithm for load balancing in datacenters was proposed using the POX controller, the Mininet emulator and the Fat-Tree topology. The packet loss, throughput and the size of the data received in a datacenter were verified. The results showed an average improvement of 27.6%, 87.95% and 26.23%, respectively, in the results that used the algorithm for load balancing. The proposed algorithm will be compared with that of [9].

Koryachko et al. [10] proposes a dynamic balancing approach in SDN with QoS. It has been proposed a mathematical model that assigns weight according to link´s flow and delay. The authors propose an algorithm that checks the delay of each link and, through the Dijkstra algorithm, calculates the route with the shortest execution time.

In [11] a dynamic mechanism has been proposed for load balancing in data center. The proposal was to create a static balance that proactively updates the rules in the network equipment flow table and, reactively acts on the network to balance the load and improve the link use. The main approach is to find the link that is overloaded and, transfer it to the idle link on a shorter distance. Through the simulation and the theoretical calculation, the author is able to balance the network load. This work also presents only one topology in the tests.

Zakia and Yedder [12] propose an SDN-based algorithm for load dynamic management, in order to optimize the link use in data center networks. The algorithm finds the shortest paths of each host and calculates the link cost. When occurs congestion on a particular path, it replaces the congested path to the best alternative path that has minimal link cost and less traffic flow. The algorithm performance is evaluated by the transfer rate measuring and, also the delay and loss of packets in a fat-tree topology. The results show a better performance in load balancing over time, as the algorithm executes, but the results were presented only for data center networks, not presenting tests in dynamic environments.

III. PROPOSED ALGORITHM

Load balancing in SDN networks can be applied to the data plane and control plane. It aims to minimize the effects of intense data traffic in the network, mitigating bottlenecks. One of the main problems is the efficient distribution of data plane resources.

This work presented algorithm has two modules: the overload identification module and the load balancer module. Figures 1 and 2 respectively show their flows. The dynamic verification approach was used, where the network devices are analyzed and the flow is deviated according to the demand. The algorithm monitors the entire network according to the information provided by the controller. When an overload is identified on a equipment, the flow deviation module is triggered. This module identifies the best path and deviate the flow to it.

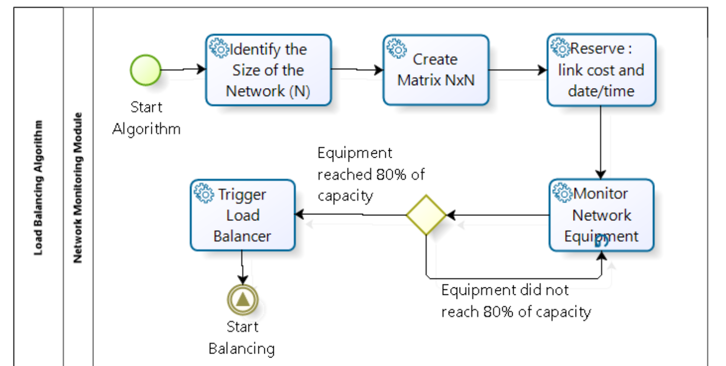


Fig. 1. Overload identification module.

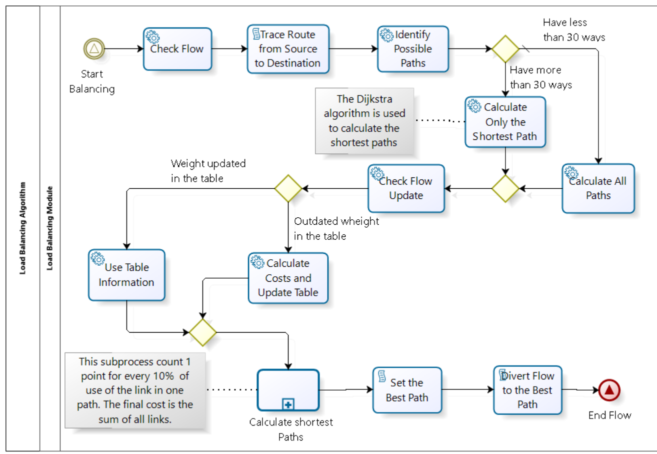


Fig. 2. Load balancer module.

The proposed algorithm works as follows: N is the size of the network, where N represents the total number of network equipment, and a N x N table is created. In each field of the table [row, column] is reserved a space to fill the cost of the link and a space to fill in the date and time at which the cost was measured. After the creation of the table, for each equipment available in the network, statistics are collected by the controller and monitored through the algorithm implemented.

The load balancer module is triggered when the use of some network equipment reaches a pre-defined percentage. The network manager can change the default value that, in this work, was 80% of the network equipment use, making the algorithm parameterizable to meet the peculiarities of each network type and size. This module searches the table for the best path and deviates the overloaded flow to this new path, reducing the overhead of the bottleneck equipment. For each traffic flow that passes through the network equipment, the balancer module identifies the best route. For each possible path, it is checked whether the flow has already been updated and whether its flow is still viable. If the path weight is outdated in the table, the algorithm calculates the cost again and updates the data. If the path weight is updated, the algorithm uses the information stored in the table.

The Algorithm 1 shows the cost calculation. For each possible path of the flow to be deviated, the Algorithm 1 is executed and checks the cost between the links, defining the best path for flow deviation. The choice of the best path does not only consider the shortest of them, because it has not always the lowest cost. To deal with this problem, a equation has been developed that assists this process. The algorithm considers the transmission capability of the network device and assigns a weight to each link according to its capacity. So the higher is the score, the higher is the link congestion.

Algorithm 1 Calculate Weight

1: **function** CALCULATEWEIGHT($path$)
2: **if** $weight.update()$ **then**
3: readTable($path$)
4: **else**
5: **for** node **do** 0 size(currentPath)
6: $tx \leftarrow bytesTransmitted$
7: $rx \leftarrow bytesReceived$
8: $transmissionRate \leftarrow tx + rx$
9: time.sleep(1)
10: $tx \leftarrow bytesTransmitted$
11: $rx \leftarrow bytesReceived$
12: $C \leftarrow capacitySwitch()$
13: $U \leftarrow tx + rx - transmissionRate$
14: $Weight \leftarrow U/C * 100$
15: **end for**
16: updateTable($path, Weight$)
17: **end if**
18: **return** $Weight$
19: **end function**

Equations (1) and (2) have been defined to ensure that the larger paths will be used only in scenarios where the shortest paths are overloaded. In this context, *P* represents the weight of the link, *U* stands for utilization and *C* represents the total capacity of the network device. The final weight of the path is the sum of all the points of your links. The Algorithm 2 presents the pseudo code that performs this calculation.

$$\lceil P \rceil \rightarrow U/C * 100 \quad (1)$$

$$\sum_{i=1}^{n} \rightarrow P_1 + P_2 + ... + P_n \quad (2)$$

Algorithm 2 Set the best path

1: **procedure** BESTPATH
2: **for** currentPath **do** 0 $nx.allSimplePaths()$
3: **for** node **do** 0 $range(len(currentPath))$
4: $path \leftarrow currentPath[node] + " :: " + currentPath[node+1]$
5: calculateWeight($path$)
6: $cost \leftarrow cost + weight$
7: **end for**
8: $tmp \leftarrow str(currentPath[len(currentPath) - 1])$
9: $linkFinalCost[tmp] \leftarrow cost$
10: $cost \leftarrow 0$
11: $weight \leftarrow 1$
12: $tmp \leftarrow null$
13: **end for**
14: $bestPath \leftarrow min(linkFinalCost)$
15: deviateFlow($bestPath$)
16: **end procedure**

After the cost calculation and the definition of the best path, the procedure presented in the Algorithm 3 receives the best path and deviates the flow.

Algorithm 3 Deviate the Flow

1: **procedure** DEVIATEFLOW($bestPath$)
2: **for** currentNode **do** 0 $range(len(bestPath))$
3: $sourceNode \leftarrow bestPath[currentNode]$
4: $destinationNode \leftarrow bestPath[currentNode + 1]$
5: $previousNode \leftarrow bestPath[currentNode - 1]$
6: $inport \leftarrow linkPorts[previousNode + " :: " + sourceNode]$
7: $outport \leftarrow linkPorts[sourceNode + " :: " + destinationNode]$
8: $xmlSrcToDst \leftarrow XMLSourceToDestination$
9: $xmlDstToSrc \leftarrow XMLDestinationToSource$
10: $URLFlow \leftarrow XMLWithBetterPath$
11: $command \leftarrow commandToFlowDeviation + XML$
12: systemCommand($command$)
13: **end for**
14: **end procedure**

The Algorithm 4 checks whether the cost is current or must be recalculated.

Algorithm 4 Cost Update

1: **function** CHECKFLOW(path)
2: $row \leftarrow path.split(" : ", 1)[0]$
3: $column \leftarrow path.split(" :: ", 1)[1]$
4: $tableTime \leftarrow table[row, column, 1]$
5: $currentTime \leftarrow datetime.now()$
6: **if** $(currentTime - tableTime) <= 30s$ **then**
7: $status \leftarrow true$
8: **else**
9: $status \leftarrow false$
10: **end if**
11: **return** $status$
12: **end function**

The time that a flow can be considered up-to-date, in this work, is thirty seconds. From that time, it was defined that the flow will be outdated and a recalculation will be necessary. This time is defined by the variable *time*. The value is parameterizable and it is up to the manager to set the value.

The Algorithm 5 shows how the table is updated when the information is not filled or if it is outdated.

Algorithm 5 Update Table

1: **procedure** UPDATETABLE($path, weigth$)
2: $row \leftarrow path.split(" : ", 1)[0]$
3: $column \leftarrow path.split(" :: ", 1)[1]$
4: $currentTime \leftarrow datetime.now()$
5: $table[row, column, 0] \leftarrow weigth$
6: $table[row, column, 1] \leftarrow currentTime$
7: **end procedure**

The balancing algorithm proposed in this work fits the complexity level O(n^2) - upper asymptotic limit - in its worst case, considering that an N x N matrix is created where N represents the quantity of equipment in the network. The calculation is performed only for the paths in which there are possibility of the flow being deviated. In this way, we can consider that, on average, the algorithm never uses the complete matrix to perform its calculations and almost always its cost will be less than O(n^2).

IV. Evaluation Methodology

A variety of controllers are available for SDN networks. Network operating systems provide controllers with an interface to a particular programming language, enabling the security feature settings, management, virtualization, and control [13]. The OpenDayLight is an open source code driver, based on Java with Python support that implements the OpenFlow protocol. It is widely used and it was this work choice.

The Mininet emulator was used for the experiments. The Mininet emulates a virtual network running on the kernel of the physical or virtualized system. The emulator is open source and also allows testing and development with SDN [14]. In the performed tests, the Mininet was run on a physical machine with Ubuntu 15.10 operating system, with 16GB of RAM and an AMD FX 8300 processor with 8 cores. It is important to note that these physical hardware resources were shared with a virtual machine running 3 Linux 14.04 LTS servers in the VirtualBox tool. For an emulation that better reflects the reality of a network environment, the controller was run on a virtual machine with the Ubuntu server 14.04 LTS operational system, to verify the physical separation between the driver and the network equipments. The driver used in this work was the OpenDayLight - Carbon SR3 (ODL). The ODL driver is a collaborative platform of open source code written in Java and has broad support for SDN networks. The ODL uses APIs REST, web interface and, provides support for large networks.

The algorithm proposed in this paper was tested in two distinct scenarios. In the first scenario, presented in Figure 3, the same network topology as the authors [9] is used. The objective of the algorithm proposed in this paper is different from the objective proposed by the authors cited, but the tests conducted show that this algorithm is equally efficient in a scenario similar to that demonstrated by the authors with the fat tree topology. The second simulated scenario, presented in Figure 3, demonstrates the mesh topology used in the performed tests. This topology was chosen because it is decentralized and provides a conducive environment for the application of new technologies, such as the use of equipment and devices that are part of the Internet of Things.

Figure 3 shows the fat-tree topology emulated, following the model presented in [9], for results comparison. The difference of the environment presented by [9] was the bandwidth (20 Mbps), because in the tests performed in this work a network with a greater capacity, of 1 Gbps, was used. It is important to note that in this topology, to perform the test with all the available paths would be impracticable, due to scalability issues, there were cases where there were more than a thousand available paths to be calculated. Therefore, in networks with more than thirty paths between one device and another, by standard, only the shortest paths were calculated to identify the best route for flow deviation.

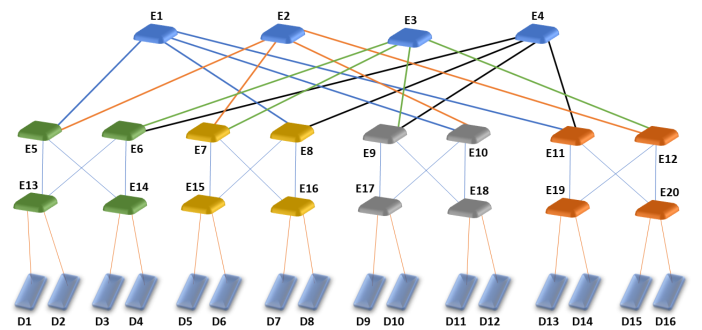


Fig. 3. Fat-tree topology.

The mesh network topology used was defined with sixteen network equipment and sixteen network devices connected to these equipment. The mesh network is a widely used topology that provides an environment where new network devices can move in and out of the network at any time. Network equipment can also change functions within the

environment, making the network dynamic and susceptible to data load unbalance at unconventional times, requiring constant monitoring. Figure 4 presents this topology.

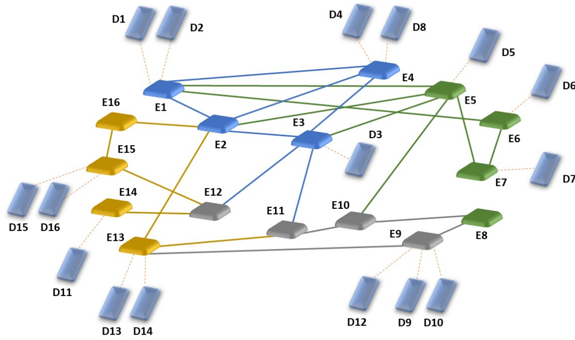


Fig. 4. Mesh topology.

In this paper, the term Network Equipment (E) was used for data plane equipments in SDN and the term Network Device (D) refers to devices connected to network equipment. The iPerf tool was used for traffic generation, which provides bandwidth verification in networks by the UDP, TCP and SCTP packets transmission. The packets capture was done with the Wireshark tool that is a multiplatform tool, which allows the following of all the packages that transit through the network and has support for several protocols. With Mininet running the network topology and the iPerf generating data traffic between the devices, the load balancer algorithm has been activated. All the paths with overflow were analyzed and, in the balancing table, each stored network equipment weight was analyzed.

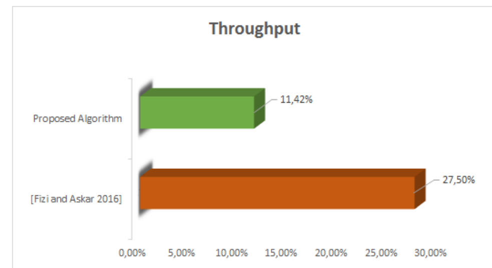
The monitoring time was 60 seconds for the performed experiments. The same experiments were also performed without using the load balancing algorithm to compare the performance gain. The experiments were repeated thirty times with a confidence interval of 95%, being the highest and the lowest results removed for each series.

V. Results

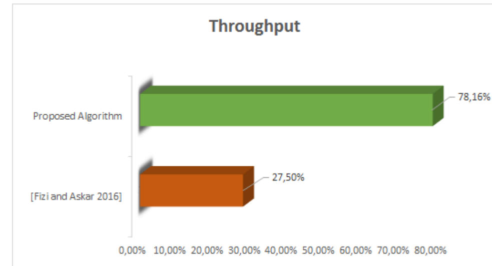
A. Fat-Tree Results

Traffic between the devices was emulated and monitored. The network workload was emulated to use 80% of available bandwidth for the proposed algorithm evaluation. The throughput, the number of lost packets and, the jitter were analyzed and compared with the results that were presented in [9]. The graphics represent the percentage of improvement on a scale from 0 to 100% with the proposed algorithm introduction. The graphics of Figures 5(a) and 5(b) present the average values of the throughput improvement in the fat-tree topology, with a transmission rate of 200 and 800 Mbps, with the proposed load balancing and, with the [9] solution.

Using the proposed algorithm, the improvement of throughput at a transmission rate between 200 Mbps devices was 11.42%. This result was below the solution proposed by the authors [9]. However, when the communication speed of



(a) Throughput at 200 Mbps

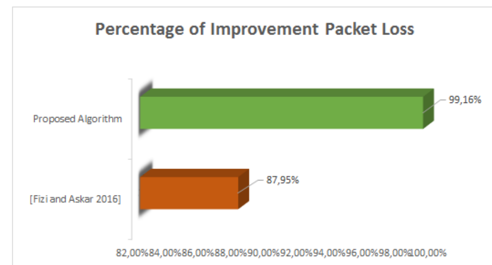


(b) Throughtput at 800 Mbps

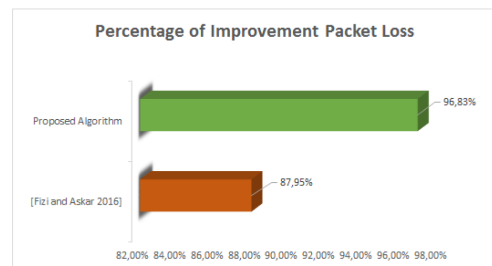
Fig. 5. Throughput improvement percentage.

the devices increases to 800 Mbps, in a more overloaded network, the throughput got a significant improvement of 78.16%. This improvement was more than the double of the value shown in [9]. When the network is overloaded, the tendency is to decrease the rate transmission. As the proposed balancing algorithm deviates to a path without overhead, the transmission rate returns to the normality, increasing the improvement percentage.

The Figures 6(a) and 6(b) present the lost packets comparison among the algorithm proposed in this paper and the one presented in [9].



(a) Packet loss at 200 Mbps

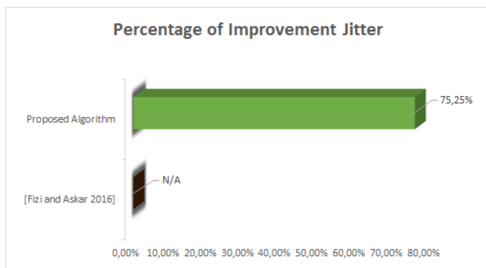


(b) Packet loss at 800 Mbps

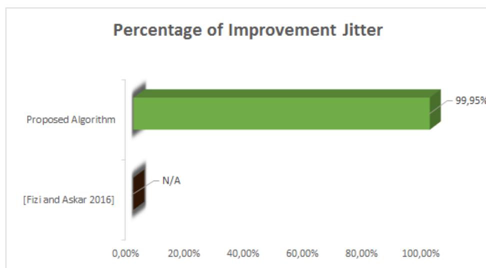
Fig. 6. Comparison of the average amount of packet loss.

In Figure 6(a), with an average transmission rate of 200 Mbps, the packet loss was reduced in about 99.16%. With the 800 Mbps transmission rate, this loss increased somewhat, but the improvement rate remained high: 96.83% as shown in the graphic of Figure 6(b).

The Figures 7(a) and 7(b) present the average value obtained from the jitter. In Figure 7(a), the results show that the jitter had a reduction of 75.25% on average. In the work presented in [9] there is no information about the jitter. In Figure 7(b), the jitter improvement percentage was 99.95% in relation to the tests performed without the load balancer algorithm. This significant improvement can be explained by the fact that in a network where data is traveling with an overhead on some of its types of equipment and, passes to use 80% of the transmission link capacity for communication between two devices, is going to have a great delay in the delivery of the package, in case of the data be traveling on an already overloaded equipment.



(a) Jitter at 200 Mbps



(b) Jitter at 800 Mbps

Fig. 7. Mean variation of Jitter.

As the load balancer algorithm has deviated the flow to an idle device, the jitter had significant improvement. The explanation for this significant improvement is the lower packet retention in lines of equipment at rush times. With the network less congested, thanks to the balancing, the fluidity is improved, which reflects in the improvement of the jitter. This result is highly beneficial to applications involving the transmission of images and videos, the streaming class applications.

B. Mesh Results

Figure 8 presents the comparison of the throughput between the algorithm proposed in this work and the solution presented by [9]. We can see that both had very close performance gains, with an average difference of 0.40% between them. In tests performed without the load balancing algorithm there was a reduction in throughput due to network equipment overload. The amount of data transferred was the same, but the lack of balance caused many packets to be lost.

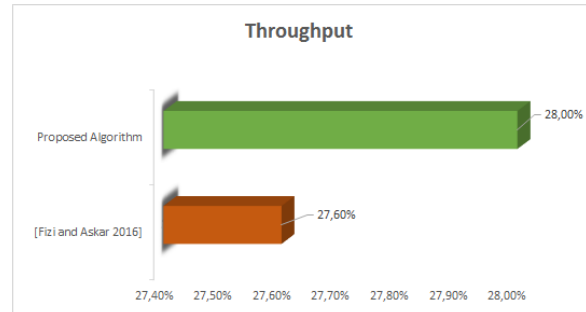


Fig. 8. Throughput improvement percentage.

Figure 9 presents the comparison between the results of the algorithm proposed in this article and the solution presented in [9]. Results were close, with an average difference of 6% between them. However, we must take into consideration that the proposed algorithm in [9] was simulated in a Fat-tree network and is geared towards data center SDN networks that are generally static environments, while the proposed algorithm focuses on a dynamic network environment.

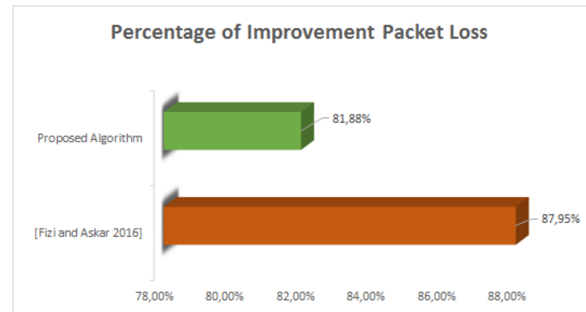


Fig. 9. Average performance gain on packet loss.

The results presented in Figure 10 show that jitter has been reduced by an average of 75.42%. This reduction shows that traffic on an overloaded and unbalanced network uses some excess network equipment while other equipment is idle.

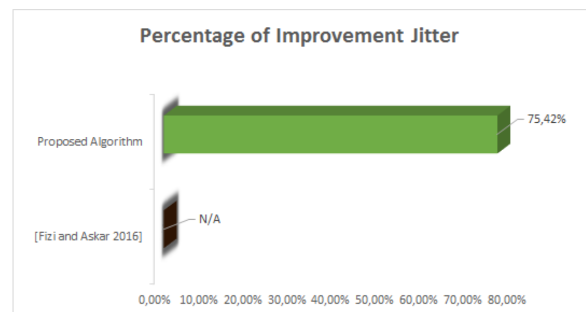


Fig. 10. Jitter improvement.

With the load balancer assigning weight to the paths, the path that goes through the overloaded network equipment will be heavier and the path that does not pass through this equipment will be chosen even if it is a longer path. As the algorithm transfers traffic to the less overloaded path, jitter consequently decreases.

VI. Conclusions

This work presented a dynamic load balancing algorithm for data plane traffic. Through Mininet and iPerf emulations, it was possible to demonstrate the use of the algorithm in SDN. The first goal of the algorithm was the link overload identification. A table of link costs is maintained by the algorithm it is and dynamic updated making possible its second goal: a selection of a less overloaded link in case of a possible network congestion.

Through the implemented algorithm, it was observed that there was a significant improvement in jitter, packet loss, and throughput for a fat-tree network topology, with high transmission rates, characterizing congestion. With this, the proposed algorithm is able to dribble congestion, making dynamic the network load balancing. The proposed algorithm can work in any kind of SDN environment. It is mainly indicated for dynamic network environments where types of equipment and devices can constantly enter and exit the environment, changing the network structure and traffic.

As future work, the use of the algorithm in a clustered environment, with other network topologies, can be investigated. Another proposal is the parallelization of the algorithm, to reduce the time of calculation of the path and the realization of tests in real environment.

Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. The authors thank Federal Center for Technological Education of Minas Gerais (CEFET-MG) and Pontifical Catholic University of Minas Gerais (PUC Minas) for the financial support.

References

[1] H. Sufiev, Y. Haddad, L. Barenboim, and J. Soler, "Dynamic SDN Controller Load Balancing," *Future Internet*, vol. 11, no. 3, 2019.

[2] O. Flauzac, C. Gonzalez, A. Hachani, and F. Nolot, "SDN Based Architecture for IoT and Improvement of the Security," in *Advanced Information Networking and Applications Workshops (WAINA), 2015 IEEE 29th International Conference on*, March 2015, pp. 688–693.

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

[4] H. Xue, K. T. Kim, and H. Y. Youn, "Dynamic Load Balancing of Software-Defined Networking Based on Genetic-Ant Colony Optimization," *Sensors*, vol. 19, no. 2, 2019.

[5] A. A. Neghabi, N. Jafari Navimipour, M. Hosseinzadeh, and A. Rezaee, "Load Balancing Mechanisms in the Software Defined Networks: A Systematic and Comprehensive Review of the Literature," *IEEE Access*, vol. 6, pp. 14 159–14 178, 2018.

[6] Y.-L. Lan, K. Wang, and Y.-H. Hsu, "Dynamic load-balanced path optimization in SDN-based data center networks," in *Communication Systems, Networks and Digital Signal Processing (CSNDSP), 2016 10th International Symposium on*, 2016, pp. 1–6.

[7] J. Li, X. Chang, Y. Ren, Z. Zhang, and G. Wang, "An effective path load balancing mechanism based on SDN," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, 2014, pp. 527–533.

[8] M. Belyaev and S. Gaivoronski, "Towards load balancing in SDN-networks during DDoS-attacks," in *Science and Technology Conference (Modern Networking Technologies)(MoNeTeC), 2014 First International*, 2014, pp. 1–6.

[9] F. S. Fizi and S. Askar, "A novel load balancing algorithm for software defined network based datacenters," in *International Conference on Broadband Communications for Next Generation Networks and Multimedia Applications (CoBCom)*, 2016, pp. 1–6.

[10] V. Koryachko, D. Perepelkin, and V. Byshov, "Approach of dynamic load balancing in software defined networks with QoS," in *Embedded Computing (MECO), 2017 6th Mediterranean Conference on*. IEEE, 2017, pp. 1–5.

[11] X. Liu and Y. Xiang, "Disturbance based dynamic load balancing routing mechanism under SDN," in *Computer Science and Network Technology (ICCSNT), 2016 5th International Conference on*. IEEE, 2016, pp. 651–656.

[12] U. Zakia and H. B. Yedder, "Dynamic load balancing in SDN-based data center networks," in *Information Technology, Electronics and Mobile Communication Conference (IEMCON), 2017 8th IEEE Annual*. IEEE, 2017, pp. 242–247.

[13] A. Lara, A. Kolasani, and B. Ramamurthy, "Network Innovation using Openflow: A Survey," *Communications Surveys Tutorials, IEEE*, vol. 16, no. 1, pp. 493–512, 2014.

[14] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6.