SDN Load Balancer using Dijkstra's Algorithm

Aishwarya Sabane
Electrical Engineering
San Jose State University
San Jose, CA
SJSU ID: 010873668
Class ID: 39

Shibani Tapikar
Electrical Engineering
San Jose State University
San Jose, CA
SJSU ID: 010987171
Class ID: 56

Jay Pathak
Electrical Engineering
San Jose State University
San Jose, CA
SJSU ID: 011500138
Class ID: 32

Swapnil Bahulekar
Electrical Engineering
San Jose State University
San Jose, CA
SJSU ID: 011483069
Class ID: 02

Abstract — Software Defined Networking is a platform for network administrators and network engineers to control, change and manage the inherent functionalities of networking nodes by abstracting the information from lower-layers of the topology. One amongst the most popular applications of Software Defined Networking is of load balancing on the network nodes by balancing the traffic load and optimizing best paths along with bandwidth and reducing latency.[1]
In our project, the implementation of a load balancer is carried out on the Floodlight SDN controller based on the Dijkstra's algorithm using a script written in Python. A meshed topology of OpenVSwitches is build to depict network complexity in real-life applications.[1] Various Application Programming Interfaces are used to gain information regarding the devices connected in the topology including their network and hardware addresses, link and switch latencies, and the bandwidth with the least cost path for data packets. Mininet is used to create a virtual topology of these nodes.[2] A combination of various test tools is utilized to verify the advantages of implementing the load-balancing scheme on the network topology.[2]

Keywords—Software Defined Networking; Load Balancing; Dijkstra's Algorithm; Application Programming Interface

I. MOTVIATION

Legacy devices and networking protocols have dominated the networking industry since the past few decades. Companies such as Juniper, Cisco, Arista, Akamai etc. have been researching newer avenues at overcoming the limitations of the traditional architecture of network management taking into account the changing traffic trends, boost of devices connected to the Internet, increase in the number of IoT devices, rise of the 'Big Data' era and cloud services.[2]

Software-Defined Networking is the perfect platform for enhancing the scope and capability of network management of current networking technology and implementing high-level applications to attain maximum optimization of network resources such as storage and bandwidth. Most popular applications implemented on SDN controllers have been traffic shapers, firewalls, load balancer, flow optimizers, mobile networking, WAN etc.[3]
Traffic patterns of the Internet have changed in magnitude tremendously over the past decade. This is seen due to the sudden wave of mobile devices being connected to the Internet. To provide every customer and organization with the best quality of service, it is important to optimize utilization of network resources such as processing power, storage, memory and bandwidth.[3]
It is for this reason that load balancing became a popular application of SDN. Implementing this over the SDN floodlight controller over Mininet, a practical experience of bandwidth and latency optimization was achieved in this project.[4] The development of a load balancing code helped the team members enhance their python scripting skills and gain sufficient knowledge regarding selecting the best available path for routing the IP packets using Dijkstra's algorithm.[4] Differencing in the path taken by packets before and after load balancing is clearly noticed by observing performance metrics of the network such as bandwidth, cost, latency etc. Studying and implementing different APIs helped gain all the little details regarding the network nodes.

II. Introduction

In the 21st century, staying globally connected is the key for most organizations for improving sales and increasing revenue. A high-speed connection, sufficient storage and increased bandwidth is the key factor in developing the networking technology in today's market.[5] Social networking sites and online web portals for purchasing and selling goods make use of internet servers for delivering reliable information and a secure connection to their customers.

The backbone networks of our world have all been connected making use of the BGP (exterior) and OSPF as the interior routing protocol for the various autonomous systems. Dijkstra's algorithm plays a major role here in finding the most efficient, shortest and least cost path for data delivery.[5] It is this very principle in use while creating a load balancing application on top of the SDN floodlight controller through the Northbound API and using the Openvswitches for network devices utilizing the OpenFlow protocol via the Southbound API of the controller. Creating a virtual topology over Mininet, improvement in the bandwidth, latency and cost of transportation of data have distinctly been displayed along with further scope for the project.[5]

III. Tools Used

We have used various tools to build this application. First, we selected a suitable open-source controller which separates the data plane and control plane to implement a software defined network. Next, we chose a tool to emulate a real-life complex network with multiple paths to various end-points. Furthermore, we selected a tool to verify the results. Finally, we selected a tool required to form network graphs which can implement the Dijkstra's Algorithm. Following tools are used:

A. Floodlight

Floodlight is a java-based OpenSDN controller, which works with the OpenFlow protocol to develop a SDN environment. OpenFlow defines a communication protocol using which the control plane (SDN controller) talks to the forwarding plane (switches and routers) to modify and push rules in the network.[6]

We have used the Floodlight controller due to its modular design and the ease to use REST APIs to develop an application. The floodlight website offers a variety of REST APIs, which are highly adaptable and gives access to a variety of modules used to make our load balancing application.[6]

B. Mininet

Mininet is a linux-based network emulator, which builds a virtualized software network.[7] Mininet can be used to quickly create a virtual network running actual kernel in the devices and switches and, software application code on a personal computer. We use mininet for building our topology, which emulates a real scenario. Using this tool, we are able to create different custom topologies by running python scripts and choosing the most suitable topology. Moreover, mininet boots boots up quickly and has a small footprint. This helps in its efficiency.[7]

C. Iperf

In this project we use Iperf to create a data-stream of ICMP packets to analyze the bandwidth between the two selected hosts. Beyond this, one can create TCP or UDP packets to measure the throughput and losses using Iperf network testing tool. It allows the user to optimize or test a network by setting various parameters like datagram size.[8]

Typical Iperf output contains a time-stamped report of the amount of data transferred and the throughput measured. We have used Iperf to ping the data from one host to another and in return we obtain the bandwidth with a timestamp attached to it.[8]

D. Dijkstras Algorithm and Networkx

Dijkstra's Algorithm is used to find the shortest path between two nodes. In this application, a graph is formed with help of networkx library in python. The nodes represent the vertices of this graph and the links from nodes to other nodes represent the edges of the graph. Based on this graph, Dijkstra's Algorithm is used to source node and find the shortest path to every node from this source.[9] Thus, we obtain a shortest path tree for the entire network and we can hence choose the shortest path from one segment of the topology. This is used to calculate the transmission rates, which gives the link cost. With the help of all of these decisions, the best path is obtained.

IV. Advantages

We have seen a tremendous rise in data activity since the last decade. This has put a massive load on the backend network, which is used to manage this data activity. Hence, load balancers are not only important in their application but also inevitable in today's data trends.[10] Following are few of the advantages that our application offers.

- Our balancer acts as a traffic management system. This system is based on SDN and Dijkstra's Algorithm works as the traffic cop, who shows the correct and efficient routes to data packets.[11]
- Using a SDN based load balancer we can easily orchestrate the application on to a different cluster of nodes in various network infrastructures.
- Attributes such as High Availability (HA) and faster transmission rates are easily achievable.
- Flow rules that are pushed to the devices, can be easily changed with a load balancing application running over SDN controller.
- Readily adaptable to changes in the network infrastructure.[12]

- Redundancy, Scalability and Flexibility are some of the features that our software based load balancer provides over any hardware based load balancers.

V. Implementation Details

A. *Infrastructure Details*

We have used Floodlight as our SDN controller as it provides some advantages:

- Supports broad range of virtual and physical OpenFlow switches.
- Designed to be high performance.
- Easy to setup with minimal dependencies.

B. *Creation of custom topology*

Created a custom topology in mininet such that analysis of load balancer script is possible. We wrote a python script in which we made API calls to addlink(), addHost, addswitch().

addlink(): Used to add link either to connect switch-switch or to connect switch-host.

addHost():Used for addition of hosts in network topology by stating its IP address.

addswitch() : Used to add new switches in the network Topology.

C. *Execution of topology*

Create the topology and run the mininet command for execution specifying IP address of the remote controller. In our case we are using Floodlight VM so we have provided IP address as the localhost address 127.0.0.1.Use the command:
$sudo mn --custom Topo.py –topo mytopo –controller=remote,ip=127.0.0.1,port=6653

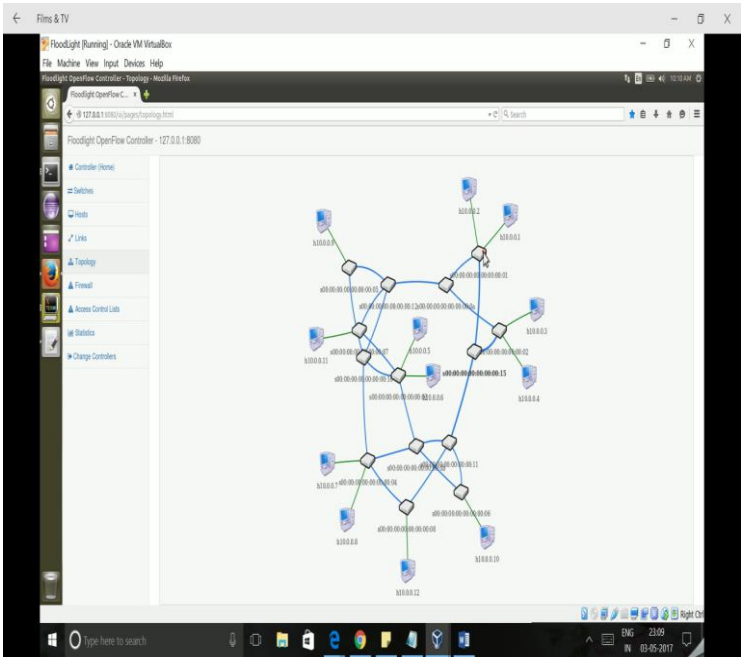

Figure 1 - Network Topology

D. *Flow of Implementation*

1. Start the Floodlight Controller and create a custom Topology in mininet with 12 hosts and 14 switches in mininet.
2. Test the connectivity by using ping command till there is 0 % drop in the packets.
3. Load balancing is to be done between two hosts h1 (10.0.0.1) and h4 (10.0.0.4)
4. Before Load Balancing: Check output ports of switches s1 and s2 to which hosts h1 and h2 are connected .Ping h1 to h4 to verify the packet flow through wireshark.
   Use the iPerf command at nodes h1 and h4 to estimate the bandwidth utilization.
5. Run the python script for load balancer.
6. Enter the hosts in which load balancing is to be done (h1 and h4)
7. Create dictionaries like Linkports {}, path{} etc., to store information obtained by accessing floodlight REST API.
8. Retrieve device information like IP address, MAC, ports, switched and store these in corresponding dictionaries.
9. Get the information of all connected switches with its links. Extract the source, destination switches and its corresponding port numbers.
10. Add an edge in the graph between source and destination nodes. Retrieve latency from switchlinks URI and store it in linkLatency dictionary with its

key as the path from source to destination. Store all links in switchlinks dictionary with its corresponding switchIDs.

11. Calculate route/path from source to destination .First the graph will give the shortest paths based on minimum hop counts from source to destination and link costs will be calculated for these paths. Store the paths in path dictionary with key as switchIDs.
12. After fetching route information, access the REST API and retrieve transmission rate and add it to the cost variable. This will give link costs for all the links for the switches in the shortest paths.
13. Find the minimum cost path from source to destination and add a flow rule based on these link costs using staticflowpusher URI.
14. After load balancing: Use iperf command and ping from h1 to h4 and observe the increase in bandwith which verifies Load balancer is working.
    Estimate the ping time from source to destination and verify that ping time decreases after running load balancer script.
    Find latency of switches in shortest paths by accessing RSET API . Retrieve duration in seconds and byte count to calculate switchlatency and find the total path Latency by adding the switch latency with the link latency retrieved from the switchlinks URI.This value will be low for the chosen minimum cost path.

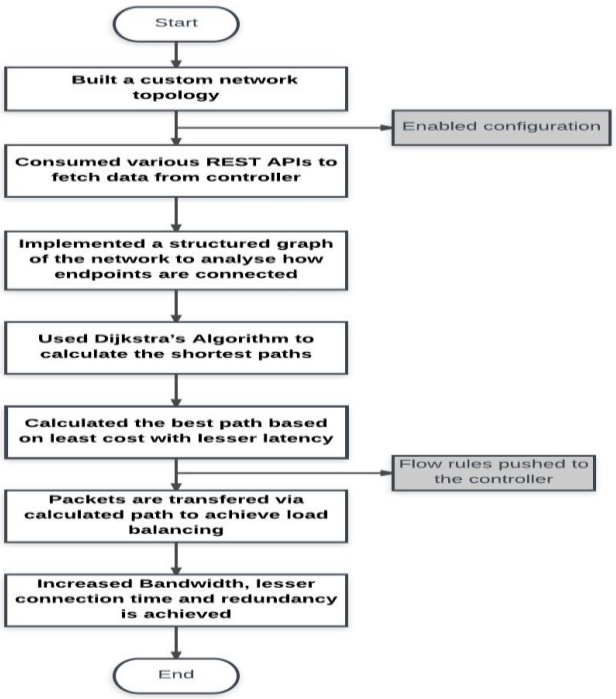

Figure 2 Flow of Implementation

VI. Results and Analysis

The Load Balancing is achieved by using Dijkstra's algorithm can be shown by using three different methods-

1. Ping statistics of hosts between which load balancing is done
2. Bandwidth of the path using iPerf
3. The best path using Wireshark

Load Balancing is done between host h1 connected to switch 1 and host 4 connected to switch 2. There are two shortest available paths between host 1 and host 4 which are 02:15:01 and 02:0a:01, that is from switch 2, switch 15, to switch 1 and from switch 2, switch a and switch 1.

The ping statistics between hosts on which load balancing is done show that the path selected has low latency. The average ping time before load balancing was higher as traffic was not distributed properly. But after balancing the load, the average ping time reduced drastically. The average ping time before Load Balancing was 10.311 ms but after Load Balancing the average time reduced to 0.103 ms. This can be seen in Figure 3 and Figure 4.

```
mininet> h4 ping -c2 h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=19.7 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.843 ms

--- 10.0.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.843/10.311/19.780/9.469 ms
```

Figure 3 Ping statistics before Load Balancing: Average time = 10.311 ms

```
mininet> h4 ping -c2 h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=6.62 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.516 ms

--- 10.0.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.516/3.572/6.629/3.057 ms
```

Figure 4 Ping statistics before Load Balancing: Average time = 3.572 ms

```
mininet> h4 ping -c2 h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.103 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.103 ms

--- 10.0.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.103/0.103/0.103/0.000 ms
```

Figure 5 Ping statistics before Load Balancing: Average time=0.103 ms

```
mininet> h4 ping -c2 h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.034 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.103 ms

--- 10.0.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.034/0.068/0.103/0.035 ms
```

Figure 6 Ping statistics before Load Balancing: Average time=0.068 ms

The second method to show that Load Balancing has been achieved is to check the path Bandwidth before and after Load Balancing. This was done using iPerf. A connection was established between hosts h1 and h4 and the Bandwidth of the path was checked. The results show that the Bandwidth of the path before Load Balancing was much less than the bandwidth of the best path selected after Load Balancing. As the load between h1 and h4 was balanced and static flow rules of the best path were pushed, the Bandwidth improved considerably. This can be seen in Figure 7 and Figure 8. The final output can be seen in Figure 9.

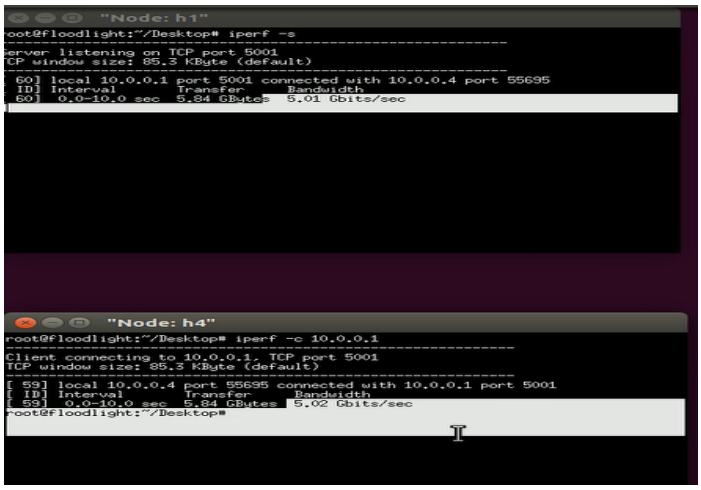

Figure 7 Bandwidth of the path before Load Balancing: 5.01 Gbits/sec

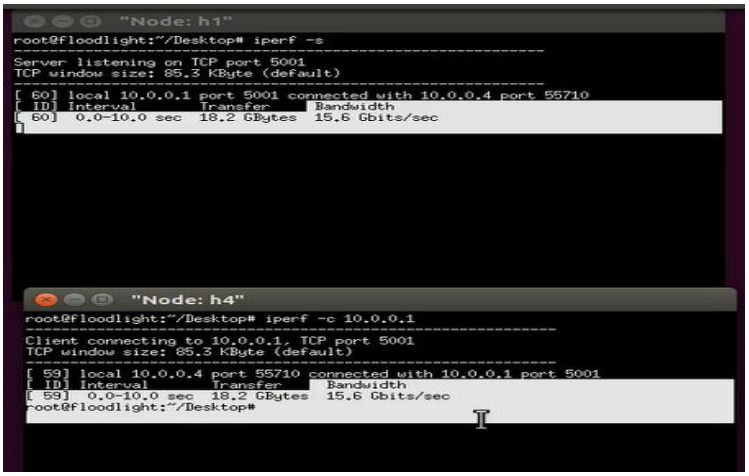

Figure 8 Bandwidth of the path after Load Balancing: 15.6 Gbits/sec

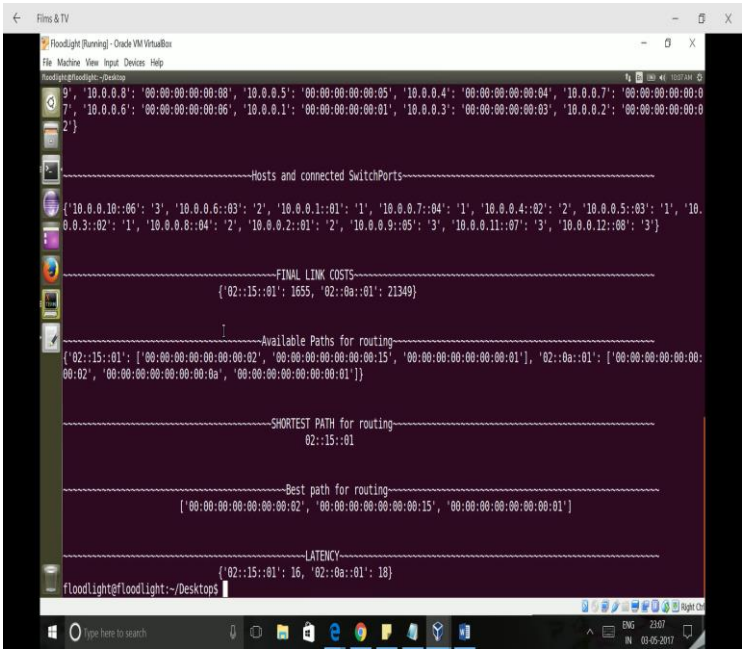

Figure 9 Final Output

The third method to show that Load Balancing has been achieved is to check the packets on a particular interface of a switch. Figure 10 shows interfaces 1 and 2 of switch 0a before implementing Load balancing. The path here is 02:0a:01.

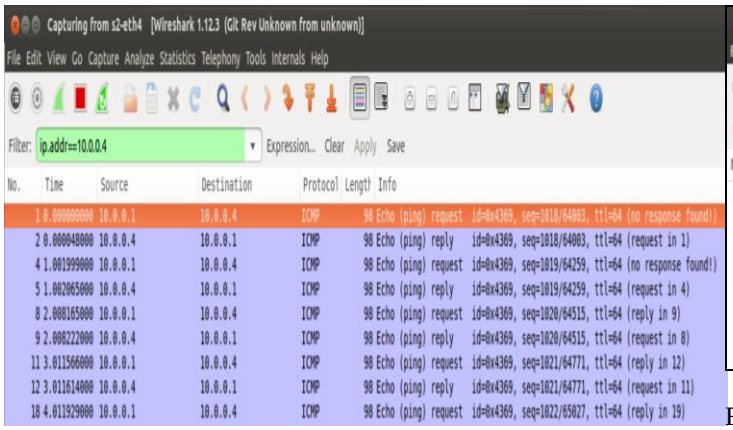

Figure 10 Switch 0a Interfaces 1 and 2 before Load Balancing

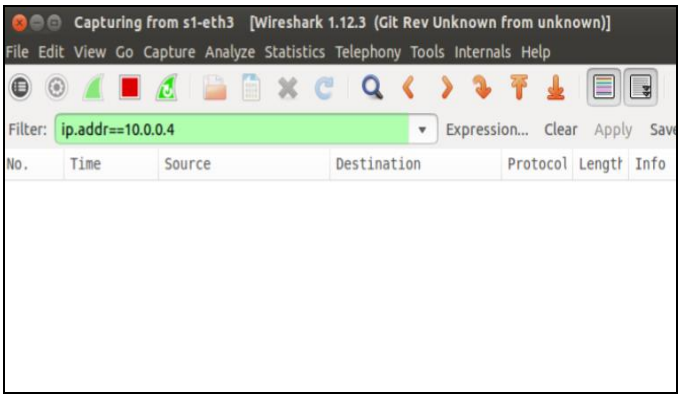

Figure 11 Switch 1 Interface 3 Before Load Balancing

When we capture packets on interface 3 of switch 1, which is connected to switch 15, before Load Balancing, we do not see any packets having destination IP address 10.0.0.4. This can be seen in Figure 11.

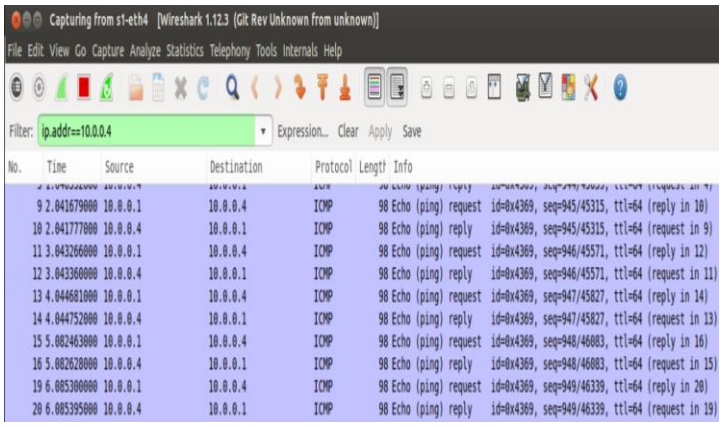

Figure 12 Switch 1 Interface 4 before Load Balancing

Figure 12 shows the packets captured on Interface 4 of switch 1, before Load Balancing. This interface is connected to switch 0a which is the destination switch.

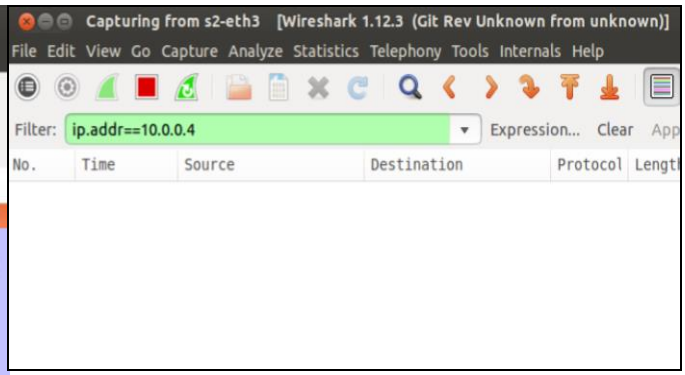

Figure 13 Switch 2 Interface 3 before Load Balancing

Figure 13 shows that no packets having destination IP address 10.0.0.4 are captured on Interface 3 of switch 2. This interface is connected to switch 15.

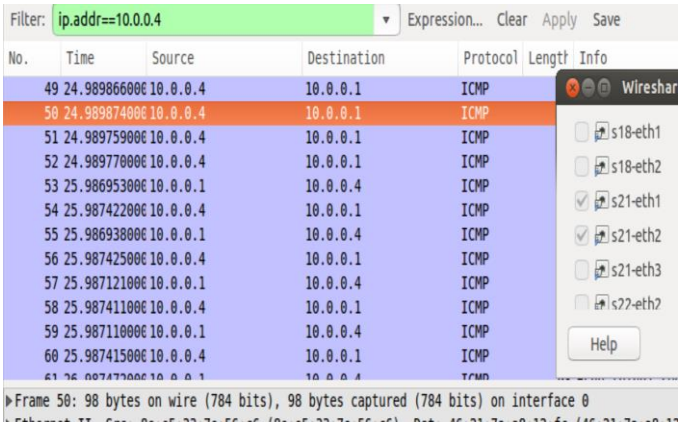

Figure 14 Switch 15 Interfaces 1 and 2 after Load Balancing

Figure 14 shows packets captured on Interfaces 1 and 2 of Switch 15 which are connected to switches 1 and 2. We can see packets having Destination IP address 10.0.0.4 here as it is the best path chosen after Load Balancing. Hence, this proves that the path 02:15:01 has been chosen as the best path.

VII. Conclusion

Load Balancing was achieved using the best shortest path between two hosts in a custom-made topology. The shortest path was chosen from all paths between the two hosts using Dijkstra's shortest path algorithm. The shortest path with the least cost was chosen as the final best path. After getting the best path, flows were pushed to the respective switches to which the hosts were connected. The latency of the path after load balancing reduced considerably which means that the load was balanced effectively. This also means that the algorithm selected the best path between the two hosts. The bandwidth of the best path increased after Load Balancing as expected. The ping statistics between the two hosts show that the traffic was managed effectively and load balancing was achieved successfully.

Thus, bandwidth optimization has been achieved and it has been ensured that the latency is low as well. The overall performance of the network has increased and the latency has decreased after load balancing.

REFERENCES

[1] Widhi Yahya, Achmad Basuki, Jehn-Ruey Jiang, "The Extended Dijkstra's-based Load Balancing for OpenFlow Network," International Journal of Electrical and Computer Engineering (IJECE), vol. 5, No. 2 pp. 289-296, April 2015.

[2] Ragalatha P, Manoj Challa, Sundeep Kumar. K, "Design and Implementation of Dynamic load balancer on openFlow enabled SDNs", IOSR Journal of Engineering, vol. 3, August 2013, pp.32-41.

[3] H. Long, Y. Shen, M. Guo, F. Tang "LABERIO: Dynamic load-balanced routing in OpenFlow-enabled networks," IEEE 27th International Conference on Advanced information Networking and Applications, 2013.

[4] Jehn-Ruey Jiang, Hsin-Wen Huang, Ji-Han Liao, Szu-Yuan Chen, "Extended Dijktra's Shortest Path Algorithm for Software Defined Networking," Poc. of the 16th Asia-Pacific Network Operations and Management Symposium(APNOMS 2014), 2014.

[5] M. Keorner, O. Kao, "Multiple service Load Balancing with OpenFlow", IEEE 13th International Conference on High performnace switching and routing, 2012.

[6] Floodlight Documentation Website, http://www.projectfloodlight.org/documentation/

[7] Mininet Website, http://mininet.org/

[8] iPerf Website, https://iperf.fr/

[9] Richard Wang, Dana Butnariu and Jennifer Rexford, "OpenFlow- Based Server Load Balancing Gone Wild", December 2010.

[10] Sandeep Sharma, Sarabjit Singh and Meenakshi Sharma, "Performance Analysis of Load Balancing Algorithms", World Academy of Science, Engineering and Technology 38 2008.

[11] Hardeep Uppal and Dane Brandon , "OpenFlow Based Load Balancing", University of Washington .

[12] Open Networking Foundation http://en.wikipedia.org/wiki/Open_Networking_Foundation

APPENDIX

*1) **CODE ANALYSIS***

1. Take input from user of the hosts on which load balancing has to be performed. For instance, if 1 and 3 are entered by the user, load balancing will be performed between hosts having IP addresses 10.0.0.1 and 10.0.0.2
2. Dictionaries storing information of the switches, MAC and IP addresses of switches, switch ports, switch links, path latency and bandwidth are created.
3. A graph which will calculate the path costs is initialized and cost is set to zero in the beginning.
4. The statistics collection is enabled by using REST API of statistics.
5. The deviceinfo function parses data into json format. It stores the IP and MAC addresses of all devices. It stores MAC addresses in deviceMAC dictionary. It also retrieves switch DPIDs and ports and stores them in switch and hostport dictionaries.
6. The SwitchLinkinfo function gets the source and destination of the host entered and the corresponding port numbers of the switches.
7. An edge is added to the graph initialized and the graph entry consists of the source and destination switch DPIDs.
8. The ports of the switches stored in the graph entry are stored in the dictionary linkports.
9. Link Latency is retrieved from the Latency URI and stored in a dictionary named linklat.
10. All the links of the switches are added to the dictionary named switchLinks.
11. The function findSwitchRoute is used to find the shortest path. The inbuilt function nx.all_shortest_paths is used to find the shortest path in the graph between host to destination. We get the shortest path based on the hop count.
12. A list named nodeList stores all the paths having the least hop count between the source and the destination.
13. The function getswitchlatency is used to retrieve the latency of switches. It retrieves the byte count and the duration from the REST URI. A dictionary named pathLatency stores the latency retrieved.
14. The function getlinklatency is used to calculate the total latency. The path latency and the switch latency is added to find out the total latency. It is stored in a dictionary named pathlat.
15. The function fetchLinkCost calculates the link costs. Bandwidth statistics are retrieved from the URI and depending on the source switch DPID and port number, the cost is computed based on the transmission rate of the links.
16. A dictionary named finalcost stores the final costs of the paths denoted by the switch DPIDs.
17. The function addflow is used to add new flows based on the shortest path calculated. The URI staticflowpusher along with the function flowRule is used to push the new rules based on the shortest path.
18. Finally, all the results like IP and MAC addresses of all devices, hosts and switch ports, Final Link Costs, Shortest Paths and the best path and Latency are printed on the screen.
19. Thus, Load Balancing has been successfully implemented using Dijkstra’s algorithm.

PEER EVALUATION

Project Members	Comments
Swapnil Bahulekar	• **Jay:** He was very punctual and finished the given tasks on time. • **Aishwarya:** Provided innovative solutions to perform tasks related to the application. • **Shibani:** Worked hard to develop the latency function and bandwidth function.
Jay Pathak	• **Aishwarya:** Highly motivated and well-spoken person, helped in understanding the application and its needs. • **Swapnil:** Implemented the networkx function and explained how it works. • **Shibani:** Properly collected the data for results and analysis of the application.
Aishwarya Sabane	• **Shibani:** Very creative and highly dedicated in her work and showed great enthusiasm in using decorators for the application. • **Jay:** Created the Python based custom topology which emulated a real-life complex network. • **Swapnil:** Learnt various tools to collect application performance results.
Shibani Tapikar	• **Swapnil:** Showed high morale and always kept the spirit of our team high. • **Aishwarya:** Used various methods of REST APIs to fetch data and modify flow rules for the application. • **Jay:** Performed the functions allocated to her in time and helped develop an efficient code.