Implementing Software Defined Load Balancer and Firewall

Aarthi Vishwanathan, Anirudh Sarma, Archit Bansal, Havish Chennamaraj and Shreya Rajkumar
College of Computing
Georgia Institute of Technology

***Abstract*—Software-defined networking (SDN) is an architecture that aims to make networks agile and flexible. The goal of SDN is to improve network control by enabling enterprises and service providers to respond quickly to changing business requirements. In a software-defined network, a network engineer or administrator can shape traffic from a centralized control console without having to touch individual switches in the network. The centralized SDN controller directs the switches to deliver network services wherever they're needed, regardless of the specific connections between a server and devices. This process is a move away from traditional network architecture, in which individual network devices make traffic decisions based on their configured routing tables. In this project, we attempt to build and test an SDN load balancer and firewall module using the Floodlight controller.**

***Index Terms*—load balancers, firewalls, Floodlight, OpenFlow.**

I. Introduction

SDN has been around for a while now and shows a lot of potential in making things easier when it comes to dynamic things in the network. Firewall and load balancers, are by nature dynamic and often change frequently in any network setup. Considering this, Load balancers and Firewalls seem to be an exact fit for the kind of innovations SDN can bring in and can be most beneficial in.

Some of the points making a strong case for SDN Load balancers and Firewalls are :

1) **Cloud-Native Applications:** With the world moving on to cloud through VMs and containers, a remotely controlled software component is easier to handle than having hardware components which are difficult to maintain when the data center is thousands miles away from the users.
2) **Scalability:** To scale a SDN component can be as easy as adding a new controller to the system which requires minimum overhead and is equivalent to deploying another software component on cloud.
3) **Flexibility:** In software-defined load balancing and firewalls, administrators can deploy custom application services on a per-application basis, instead of fitting multiple applications on a single monolithic hardware appliance to save hardware costs.
4) **Hybrid cloud applications:** Software load balancers and firewalls provide a consistent application delivery architecture across different cloud environments. This eliminates the need to re-architect applications when migrating to the cloud or between clouds.
5) **Maintenance:** Maintenance of a SDN component is super convenient and any new updates/fixes can be rolled out seamlessly to all the controllers without much hassle and risk.
6) **Redundancy/Resilience:** If a server running the load balancer/firewall is brought down, other deployments can be quickly enabled to pick up the slack and prevent service disruptions.

II. Overview

In our project, we build two modules - A Load balancer and a Firewall. We leverage the the floodlight controller and wrote python modules to interact with this controller. We simulated our network using mininet, and analyzed our results using wireshark. Section III details our methodology. Section IV lists out related work carried out in the SDN sphere, and section V concludes our work with thoughts on future work.

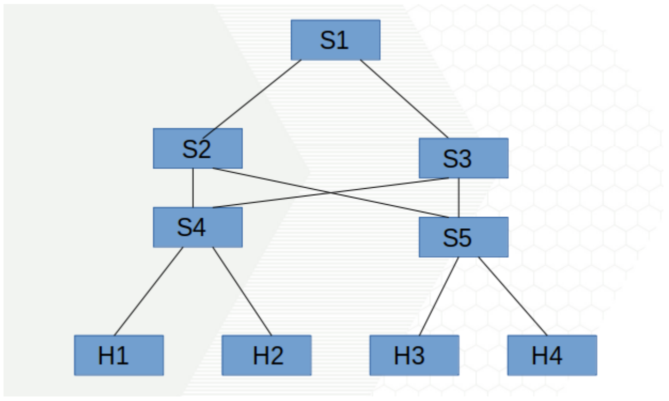


Fig. 1. Mininet topology

III. Methodology

To start with the implementation of the Load Balancer, we first setup Mininet and Floodlight on our systems. Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine[6]. For any SDN based study, the most important thing is the choice of the controller. After thorough discussion on the various SDN controllers available and weighing their pros and cons, we decided to choose the Floodlight controller. The Floodlight Open SDN Controller is an enterprise-class, Apache-licensed, Java-based OpenFlow Controller[5]. Two of the main reasons for this choice include: The fact that the Floodlight website had several coding examples along with directions on how to build the product. This was helpful to members who were new to this domain. REST APIs are included to simplify application interfaces to the product.

A. *Mininet Topology*

We sought to simulate our project in a rudimentary 3 tier topology which is displayed in Fig. 1. H1-H4 are leaf nodes, while S1-S5 are switches, which will route packets according to the policies that our pushed by the flood light controller.

B. *Load Balancer*

After a good understanding of the working of the entire setup, we decided to implement a very simple load balancing module to start with. The flow diagram for the same is displayed in Fig. 2. The python module will interact with floodlight, which in turn will push rules to the desired switches. For this, we implemented a static rule based load balancer which works as follows:

1) The module uses Floodlight REST APIs to push static load balancing rules information to Floodlight.
2) The rule contains a static (hard coded) destination ip address and the link to forward the traffic on. This way we were able to achieve two different packets with different destination ip being forwarded to different interfaces as per the rules pushed by us.
3) Once the rules were loaded into the controller, we verified the correct functioning of the module through packet tracing using Wireshark.

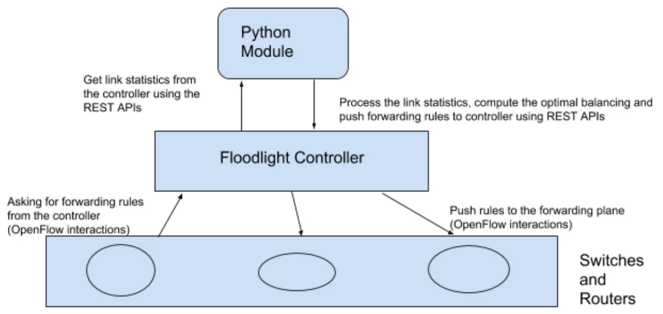

Python Module
Get link statistics from the controller using the REST APIs
Process the link statistics, compute the optimal balancing and push forwarding rules to controller using REST APIs
Floodlight Controller
Asking for forwarding rules from the controller (OpenFlow interactions)
Push rules to the forwarding plane (OpenFlow interactions)
Switches and Routers

Fig. 2. Interactions among components in the SDN loadbalancer

We then proceeded to implement a dynamic load balancing and firewall module. The basic idea is to select a path based on the least load from client towards the server, irrespective of the type of flow. The load balancer module implements the following algorithm:

1) Identify all paths from the source to the destination in the form of a list of links.
2) Fetch the transmission rate(tx bits per sec) for all the links in all the paths.
3) For every path, the cost is defined as the sum of transmission rate across all the links in the path.
4) The cost for every identified path is calculated.
5) The path with the minimum cost is chosen and the corresponding flow is pushed to the controller.

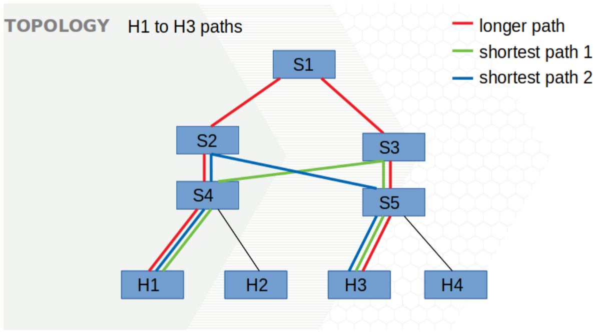
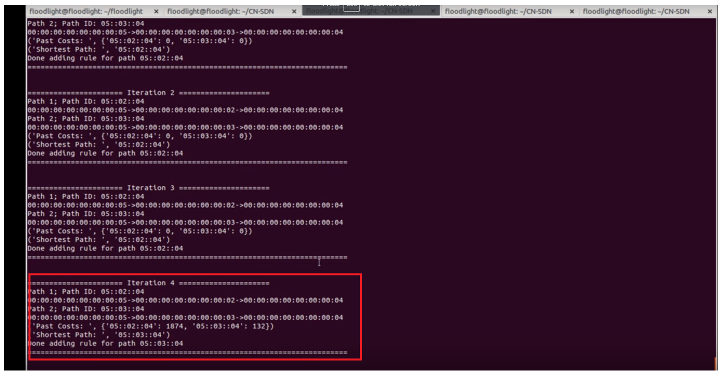
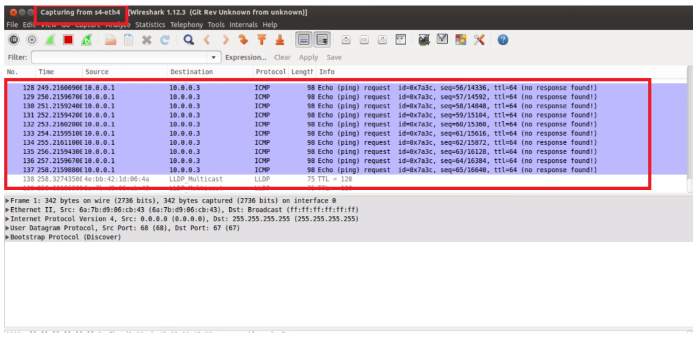


Fig. 3. Shortest and longest paths in the topology from H1 to H4


floodlight@floodlight: ~/floodlight
floodlight@floodlight: ~/CN-SDN
floodlight@floodlight: ~/CN-SDN
floodlight@floodlight: ~/CN-SDN
floodlight@floodlight: ~/CN-SDN

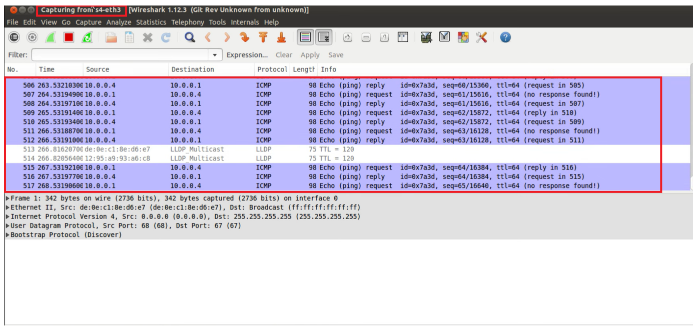
```
Path 2; Path ID: 05::03::04
00:00:00:00:00:00:00:05->00:00:00:00:00:00:00:03->00:00:00:00:00:00:00:04
('Past Costs: ', {'05::02::04': 0, '05::03::04': 0})
('Shortest Path: ', '05::02::04')
Done adding rule for path 05::02::04
==========================================================================


===================== Iteration 2 ====================
Path 1; Path ID: 05::02::04
00:00:00:00:00:00:00:05->00:00:00:00:00:00:00:02->00:00:00:00:00:00:00:04
Path 2; Path ID: 05::03::04
00:00:00:00:00:00:00:05->00:00:00:00:00:00:00:03->00:00:00:00:00:00:00:04
('Past Costs: ', {'05::02::04': 0, '05::03::04': 0})
('Shortest Path: ', '05::02::04')
Done adding rule for path 05::02::04
==========================================================================


===================== Iteration 3 ====================
Path 1; Path ID: 05::02::04
00:00:00:00:00:00:00:05->00:00:00:00:00:00:00:02->00:00:00:00:00:00:00:04
Path 2; Path ID: 05::03::04
00:00:00:00:00:00:00:05->00:00:00:00:00:00:00:03->00:00:00:00:00:00:00:04
('Past Costs: ', {'05::02::04': 0, '05::03::04': 0})
('Shortest Path: ', '05::02::04')
Done adding rule for path 05::02::04
==========================================================================

===================== Iteration 4 ====================
Path 1; Path ID: 05::02::04
00:00:00:00:00:00:00:05->00:00:00:00:00:00:00:02->00:00:00:00:00:00:00:04
Path 2; Path ID: 05::03::04
00:00:00:00:00:00:00:05->00:00:00:00:00:00:00:03->00:00:00:00:00:00:00:04
('Past Costs: ', {'05::02::04': 1874, '05::03::04': 132})
('Shortest Path: ', '05::03::04')
Done adding rule for path 05::03::04
==========================================================================
```

Fig. 4. Identification of all the paths and the costs by the load balancer


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression...
Clear
Apply
Save

```
Capturing from s4-eth4  [Wireshark 1.12.3 (Git Rev Unknown from unknown)]

No.   Time          Source             Destination    Protocol Length Info
  128 249.21600900  10.0.0.1           10.0.0.3       ICMP      98 Echo (ping) request  id=0x7a3c, seq=56/14336, ttl=64 (no response found!)
  129 250.21596700  10.0.0.1           10.0.0.3       ICMP      98 Echo (ping) request  id=0x7a3c, seq=57/14592, ttl=64 (no response found!)
  130 251.21592400  10.0.0.1           10.0.0.3       ICMP      98 Echo (ping) request  id=0x7a3c, seq=58/14848, ttl=64 (no response found!)
  131 252.21594200  10.0.0.1           10.0.0.3       ICMP      98 Echo (ping) request  id=0x7a3c, seq=59/15104, ttl=64 (no response found!)
  132 253.21602000  10.0.0.1           10.0.0.3       ICMP      98 Echo (ping) request  id=0x7a3c, seq=60/15360, ttl=64 (no response found!)
  133 254.21595100  10.0.0.1           10.0.0.3       ICMP      98 Echo (ping) request  id=0x7a3c, seq=61/15616, ttl=64 (no response found!)
  134 255.21611000  10.0.0.1           10.0.0.3       ICMP      98 Echo (ping) request  id=0x7a3c, seq=62/15872, ttl=64 (no response found!)
  135 256.21594300  10.0.0.1           10.0.0.3       ICMP      98 Echo (ping) request  id=0x7a3c, seq=63/16128, ttl=64 (no response found!)
  136 257.21596700  10.0.0.1           10.0.0.3       ICMP      98 Echo (ping) request  id=0x7a3c, seq=64/16384, ttl=64 (no response found!)
  137 258.21598800  10.0.0.1           10.0.0.3       ICMP      98 Echo (ping) request  id=0x7a3c, seq=65/16640, ttl=64 (no response found!)
  138 258.32743500  4e:bb:42:1d:06:4a  LLDP_Multicast LLDP      75 TTL = 120

▶Frame 1: 342 bytes on wire (2736 bits), 342 bytes captured (2736 bits) on interface 0
▶Ethernet II, Src: 6a:7b:d9:06:cb:43 (6a:7b:d9:06:cb:43), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▶Internet Protocol Version 4, Src: 0.0.0.0 (0.0.0.0), Dst: 255.255.255.255 (255.255.255.255)
▶User Datagram Protocol, Src Port: 68 (68), Dst Port: 67 (67)
▶Bootstrap Protocol (Discover)
```

Fig. 5. Wireshark showing traffic from H1 to H3 taking blue route


File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression...
Clear
Apply
Save

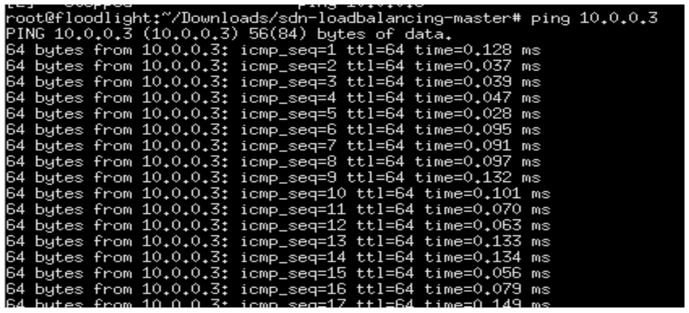
```
Capturing from s4-eth3  [Wireshark 1.12.3 (Git Rev Unknown from unknown)]

No.   Time          Source             Destination    Protocol Length Info
  506 263.53210300  10.0.0.4           10.0.0.1       ICMP      98 Echo (ping) reply    id=0x7a3d, seq=60/15360, ttl=64 (request in 505)
  507 264.53194900  10.0.0.1           10.0.0.4       ICMP      98 Echo (ping) request  id=0x7a3d, seq=61/15616, ttl=64 (no response found!)
  508 264.53197100  10.0.0.4           10.0.0.1       ICMP      98 Echo (ping) reply    id=0x7a3d, seq=61/15616, ttl=64 (request in 507)
  509 265.53191400  10.0.0.1           10.0.0.4       ICMP      98 Echo (ping) request  id=0x7a3d, seq=62/15872, ttl=64 (reply in 510)
  510 265.53193400  10.0.0.4           10.0.0.1       ICMP      98 Echo (ping) reply    id=0x7a3d, seq=62/15872, ttl=64 (request in 509)
  511 266.53188700  10.0.0.1           10.0.0.4       ICMP      98 Echo (ping) request  id=0x7a3d, seq=63/16128, ttl=64 (no response found!)
  512 266.53191000  10.0.0.4           10.0.0.1       ICMP      98 Echo (ping) reply    id=0x7a3d, seq=63/16128, ttl=64 (request in 511)
  513 266.81620700  de:0e:c1:8e:d6:e7  LLDP_Multicast LLDP      75 TTL = 120
  514 266.82056400  12:95:a9:93:a6:c8  LLDP_Multicast LLDP      75 TTL = 120
  515 267.53192100  10.0.0.1           10.0.0.4       ICMP      98 Echo (ping) request  id=0x7a3d, seq=64/16384, ttl=64 (reply in 516)
  516 267.53197700  10.0.0.4           10.0.0.1       ICMP      98 Echo (ping) reply    id=0x7a3d, seq=64/16384, ttl=64 (request in 515)
  517 268.53190600  10.0.0.1           10.0.0.4       ICMP      98 Echo (ping) request  id=0x7a3d, seq=65/16640, ttl=64 (no response found!)
▶Frame 1: 342 bytes on wire (2736 bits), 342 bytes captured (2736 bits) on interface 0
▶Ethernet II, Src: de:0e:c1:8e:d6:e7 (de:0e:c1:8e:d6:e7), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▶Internet Protocol Version 4, Src: 0.0.0.0 (0.0.0.0), Dst: 255.255.255.255 (255.255.255.255)
▶User Datagram Protocol, Src Port: 68 (68), Dst Port: 67 (67)
▶Bootstrap Protocol (Discover)
```

Fig. 6. Wireshark showing traffic from H1 to H4 taking green route

Once we had the model ready, we then simulated ping traffic across all nodes. We then explicitly tried to ping H1-H3, and H1-H4. We then observed the traffic flow on wireshark to notice that before loadbalancing, both flows opted for H1-S4-S2-S5, but soon after a few iterations of the load balancing module for balancing load between H1 and H4, we found the flow had switched to H1-S4-S3-S5 path.
The figures 4, 5 and 6 illustrates the identification of the paths and the costs, and wireshark confirming balanced load between flows H1 to H3 and H1 to H4 respectively.

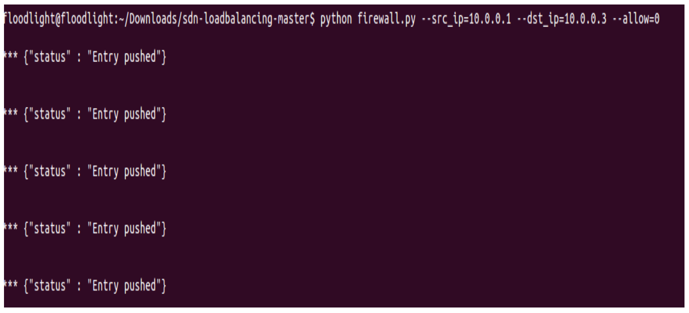
C. Firewall

The Firewall module is simpler in approach. The rules are applied as per user specification and pervades the entire network. The module takes in user parameter takes in IP, port and operation, and then flushes the rule to the entire network via the controller. If a packet arrives at a switch which matches the rule flushed, appropriate action is taken on the packet.
Firstly, the Floodlight firewall is disabled. The user can then specify the source and destination IP address and whether he would like to enable or disable a particular flow on the go. The IP addresses mentioned will be added to the flow and the corresponding action will be taken (allow or deny).
Figures 7, 8, 9 and 10 depict the working of the firewall.



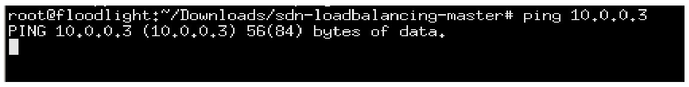
```
root@floodlight:~/Downloads/sdn-loadbalancing-master# ping 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=0.128 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.037 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.039 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=0.047 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=0.028 ms
64 bytes from 10.0.0.3: icmp_seq=6 ttl=64 time=0.095 ms
64 bytes from 10.0.0.3: icmp_seq=7 ttl=64 time=0.091 ms
64 bytes from 10.0.0.3: icmp_seq=8 ttl=64 time=0.097 ms
64 bytes from 10.0.0.3: icmp_seq=9 ttl=64 time=0.132 ms
64 bytes from 10.0.0.3: icmp_seq=10 ttl=64 time=0.101 ms
64 bytes from 10.0.0.3: icmp_seq=11 ttl=64 time=0.070 ms
64 bytes from 10.0.0.3: icmp_seq=12 ttl=64 time=0.063 ms
64 bytes from 10.0.0.3: icmp_seq=13 ttl=64 time=0.133 ms
64 bytes from 10.0.0.3: icmp_seq=14 ttl=64 time=0.134 ms
64 bytes from 10.0.0.3: icmp_seq=15 ttl=64 time=0.056 ms
64 bytes from 10.0.0.3: icmp_seq=16 ttl=64 time=0.079 ms
64 bytes from 10.0.0.3: icmp_seq=17 ttl=64 time=0.149 ms
```

Fig. 7. Packets sent from source to destination



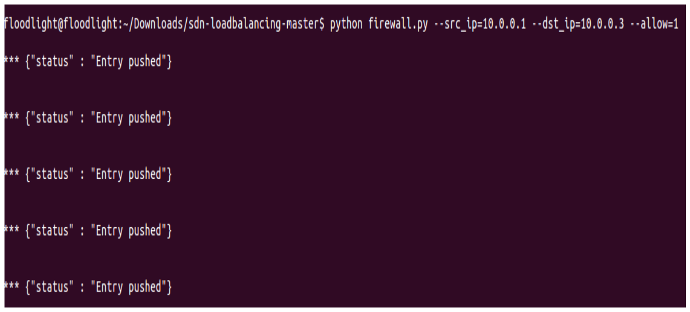
```
floodlight@floodlight:~/Downloads/sdn-loadbalancing-master$ python firewall.py --src_ip=10.0.0.1 --dst_ip=10.0.0.3 --allow=0
*** {"status" : "Entry pushed"}

*** {"status" : "Entry pushed"}

*** {"status" : "Entry pushed"}

*** {"status" : "Entry pushed"}

*** {"status" : "Entry pushed"}
```

Fig. 8. User disables flow by setting allow=0



```
root@floodlight:~/Downloads/sdn-loadbalancing-master# ping 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
```

Fig. 9. No packets sent from source to destination



```
floodlight@floodlight:~/Downloads/sdn-loadbalancing-master$ python firewall.py --src_ip=10.0.0.1 --dst_ip=10.0.0.3 --allow=1
*** {"status" : "Entry pushed"}

*** {"status" : "Entry pushed"}

*** {"status" : "Entry pushed"}

*** {"status" : "Entry pushed"}

*** {"status" : "Entry pushed"}
```

Fig. 10. User enables flow by setting allow=1

When the user mentions allow as 1, the flow is enabled from that source to destination. Similarly, when the user mentions allow as 0, the flow is disabled from that source to destination and the switches along that path are blocked.

IV. Related Work

We primarily surveyed literature regarding load balancing algorithms and firewalls. We drew inferences from the papers as to a possible implementation of a few in our project. Among the several types of load balancing algorithms, they can be broadly classified into static and dynamic. A few are described below:

1) ROUND-ROBIN: In this, each servers receive the request from clients in circular manner. The requests are allocated to various live servers on round robin base.[1]
2) WEIGHTED ROUND-ROBIN: In this, each server receive the request from the client based on criteria that are fixed by the site administrator. In other world A Static weight is assigned to each server in Weighted Round Robin (WRR) policy. We usually specify weights in proportion to actual capacities. So, for example, if Server 1's capacity is 5 times more than Server 2's, then we can assign a weight of 5 to server 1 and weight of 1 to server 2.[1]
3) RANDOM STRATEGY: From a list of live servers, the Load Balancer will randomly choose a server for sending request. This policy has large overheads.[1]
4) HASH BASED. The algorithm works by first calculating the hash value of traffic flow using IP address of source and destination, port number of source and destination and URL. The request is then forwarded to the server with highest hash value. If any other request comes with same hash value, it will be forwarded to same server.[2]
5) GLOBAL FIRST FIT(GFF): After receiving a new flow request, the scheduler searches linearly all the available paths in order to find the one which can accommodate the bandwidth requirement of this new flow. The flow is greedily assigned to the first path which is fulfilling the requirement. Global First Fit does not distribute flows evenly across all paths.[3]
6) FLOW BASED LOAD BALANCING:Classify the incoming request into mice or elephant flows. The server

selection module will select the server with the smallest elephant flow counter value or mice flow counter value depending on whether the new flow is elephant or mice, respectively. Collect port statistics and flow statistics of switches. The statistics help in the selection of Server/Path based on a weighted heuristic.[3]

Firewalls have been implemented using simple rules based on MAC, IP addresses, protocol etc[4].

V. Results

The load balancer module and firewall were successfully implemented and tested with varying topologies and environment. The exact experiments and screenshots are explained in Methodology part of the paper.
The possible enhancements and scope of improvements in both the modules are dicussed further in Future Work.

VI. Future Work

Our current load balancing algorithm is based on transmission rate(bits per second through links). This might be a very limiting heurestic, as we would want certain traffic to flow via specific paths irrespective of load on path. Future work on the same would see implementations of the algorithms that are based on other link characteristics such as rate of loss, RTT etc. Policies can also factor priority and the content of the traffic. A beneficial feature for the end-user of the module would be flexebility in selecting an algorithm.

The present firewall operates on user input, and is limited to IP an port. This implementation can be extended to take actions based on packet contents, or to specify access lists that depends on the source of originating traffic.

References

[1] Sabiya, Japinder Singh, "Weighted Round-Robin Load Balancing Using Software Defined Networking" *IJARCSSE*, vol. 6, no. 6, June 2016.

[2] P. Kumari, D. Thakur, "Load Balancing in Software Defined Network", *IJCSE*, vol. 5,no. 12, Dec 2017.

[3] Gaurav, "Server and Network Load Balancing in SDN Content Delivery Datacenter Network", *Masters Thesis presented to Ryerson University*, June 2010.

[4] Amandeep Kaur, Vikramjit Singh, "Building L2-L4 Firewall using Software Defined Networking", *IJIACS*, vol. 6,no. 6, June 2017.

[5] http://www.projectfloodlight.org/floodlight/

[6] http://mininet.org/