

Implementing Software Defined Load Balancer and Firewall

Shreya Rajkumar*

*(ECE, Georgia Institute of Technology, United States
Email: shreyarajkumar2@gmail.com)

Abstract:

Software-defined networking (SDN) is an architecture that aims to make networks fast and flexible. SDN's goal is to improve network control by enabling service providers as well as enterprises to respond quickly to changing business needs. In SDN, the administrator can shape traffic from a centralized control console without having to modify any of the individual switches belonging to the network. The SDN controller which is centralized directs the switches to deliver network services wherever they are needed, irrespective of the specific connections between a server and devices. This methodology is a shift from traditional network architecture, in which individual network devices make traffic decisions based on their configured routing tables. In this paper, I built and tested an SDN load balancer and firewall module using the Floodlight controller.

Keywords —Software-Defined Networking, Load Balancers, Firewalls, Networks

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 the cloud through VMs and containers, a remotely controlled software component is easier to handle than having hardware components that are difficult to maintain when the data center is thousands of miles away from the users.

2) Scalability: To scale an 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 the 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.

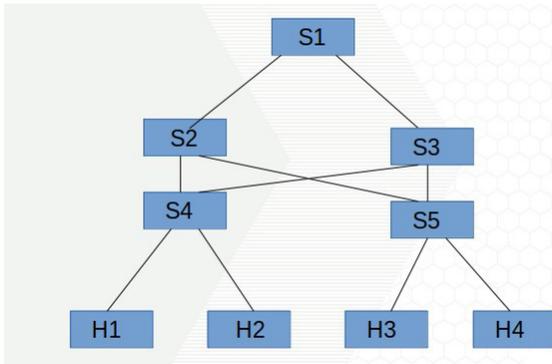
5) This eliminates the need to re-architect applications when migrating to the cloud or between clouds. super convenient and any new updates/fixes can be rolled out seamlessly to all the controllers without much hassle and risk.

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

A Load balancer and a Firewall were built using SDN. The floodlight controller was leveraged, and python modules were written to interact with this controller. The network was simulated using Mininet as seen in Fig 1, and the results were analyzed using Wireshark. Section III details the methodology. Section IV lists related work carried out in the SDN sphere, and section V concludes the work with thoughts on future work.

Fig.1 Mininet Topology



III. METHODOLOGY

To start with the implementation of the Load Balancer, I first set up Mininet and Floodlight on the systems. A realistic virtual network is created by Mininet, 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 research on the various SDN controllers available and weighing their pros and cons, I decided to choose the Floodlight controller. The Floodlight Open SDN Controller is an enterprise-class, Apache-licensed, Java-based OpenFlow Controller [5]. The main reasons for choosing this controller were because of the clear documentation as well as the presence of REST APIs to simplify application interfaces to the product.

A. MininetTopology

I sought to simulate my 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 were pushed by the flood light controller.

B. LoadBalancer

After a good understanding of the working of the entire setup, I 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, I implemented a static rule-based load balancer which works as follows:

- 1) The module uses Floodlight REST APIs to push static loadbalancingrulesinformationtoFloodlight.
- 2) The rule contains a static (hardcoded) destination IP address and the link to forward the traffic on. This way I was able to achieve two different packets with different destination IP being forwarded to different interfacesaspertherulespushedbyus.
- 3) Once the rules were loaded into the controller, I verified the correct functioning of the module through packet tracing usingWireshark.

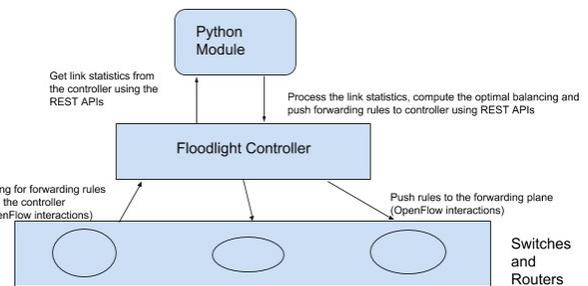
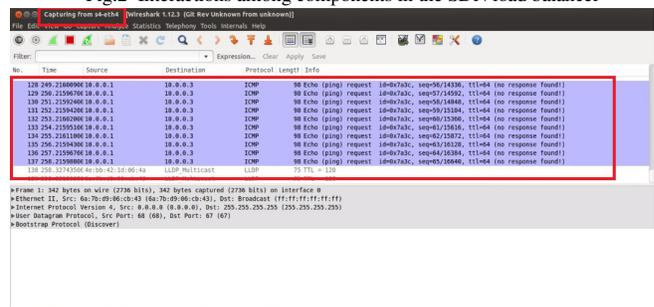


Fig.2 Interactions among components in the SDN load balancer



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=0.096 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 Packets sent from source to destination.

```
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 allowing as 1, the flow

is enabled from that source to the 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

I primarily surveyed literature regarding load balancing algorithms and firewalls. The several types of load balancing algorithms can broadly be classified into static and dynamic. A few are described below:

1) **ROUND-ROBIN**: In this, each server receives the request from clients in a circular manner. The requests are allocated to various live servers on a round-robin base.[1]

2) **WEIGHTED ROUND-ROBIN**: In this, each server receives the request from the client based on criteria that are fixed by the site administrator. In other words, 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 a 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 requests. This policy has large overheads.[1]

4) **HASH-BASED**. The algorithm works by first calculating the hash value of traffic flow using the IP address of source and destination, port number of source and destination, and URL. The request is then forwarded to the server with the highest hash value. If any other request comes with the same hash value, it will be forwarded to the same server.[2]

5) **GLOBAL FIRST FIT(GFF)**: After receiving a new flow request, the scheduler searches linearly all the available paths 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 in-coming request into mice or elephant flows. The serverselection 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 environments. The exact experiments and screenshots are explained in the Methodology part of the paper.

The possible enhancements and scope of improvements in both modules are discussed further in Future Work.

VI. FUTURE WORK

My current load balancing algorithm is based on transmission rate (bits per second through links). This might be a very limiting heuristic, as we would want certain traffic to flow via specific paths irrespective of load on the 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 in priority and the content of the traffic. A beneficial feature for the end-user of the module would be flexibility in selecting an algorithm.

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

ACKNOWLEDGMENT

I would like to thank Georgia Institute of Technology for giving me the opportunity to research on this topic.

REFERENCES

- [1] Sabiya, Japinder Singh, "Weighted Round-Robin Load Balancing Using Software Defined Networking" IJARCSSE, vol. 6, no. 6 (2016).
- [2] P. Kumari, D. Thakur, "Load Balancing in Software Defined Network",
- [3] IJCSE, vol. 5, no. 12 (2017).
- [4] Gaurav, "Server and Network Load Balancing in SDN Content Delivery Datacenter Network", Masters Thesis presented to Ryerson University (2010).
- [5] Amandeep Kaur, Vikramjit Singh, "Building L2-L4 Firewall using Soft-ware Defined Networking", IJIACS, vol. 6, no. 6(2017).