Data Center Network with Software Defined Network

Javier Benitez Fernandez, Sowmya Ravidas
Aalto University School of Science
javier.benitezfernandez@aalto.fi, sowmya.ravidas@aalto.fi

Abstract

In this report, we emulate a simple data center network using mininet emulation platform. The network is a fat tree topology consisting of hosts and openflow switches. Further, we develop our own Software Defined Networking (SDN) controller to find the shortest path forwarding between the hosts.

KEYWORDS: Mininet, SDN, OVS, Pox

1 Introduction

We use Mininet emulation platform to create a data center network. This network consists of hosts and openflow switches with single or multiple paths between them. The multiple paths enable fault tolerance and load balancing in a data center network.

We have developed our own SDN controller that discovers the path between the hosts and the switches. It also identifies the shortest path and sends the packets through them.

In section 2, we describe our Experimental Setup and Network Topology. In section 3 we explain the working of our controller and Section 4 concludes the report.

2 Experimentation Setup

We use Mininet [1], a network emulation platform to create our topology. The network topology of the Data Center network is as shown in Figure 1.

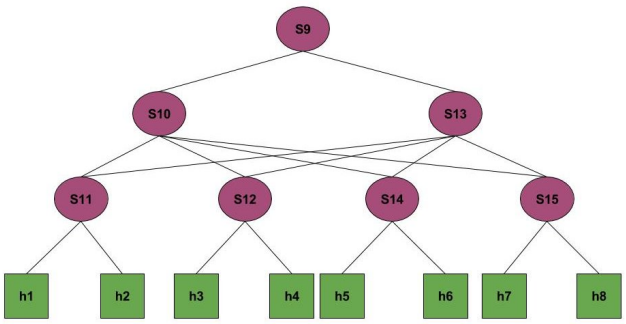

Figure 1: Topology of a Data Center Network

This is a fat tree topology of depth 3. We create the topology in mininet by creating the hosts, switches and also the links between them. For example, the connections between h1,h2 and s11 are created as shown below.

```
h1 = self.addHost( 'h1' )
h2 = self.addHost( 'h2' )
s11 = self.addSwitch( 's11' )
self.addLink(h1,s11)
self.addLink(h2,s11)
```

In our experiment, we use 8 hosts and 7 switches with 18 interconnections in total. As shown above, we create each host, switches and also specify the links between them.

We run the following command to launch mininet with the fat tree topology.

```
sudo mn --custom ~/mininet/custom/
topo-fattree.py --topo fattree
--mac --arp --switch ovsk
 --controller remote
 --ip 192.168.56.1
```

The command performs the following [4] :

1. sudo : To run as root
2. mn : It launches the mininet
3. –custom /mininet/custom/ topo-fattree.py: This creates our specified fat tree topology
4. –mac : this makes the mac address of the specified hosts (h1 to h8) to be same as their node number.
5. –arp : This installs static ARP entries in the hosts
6. –switch ovsk : In our experiment, we use Open vSwitch in kernel switch for all switches from S9 to S15
7. –controller remote : It uses a remote controller that runs outside mininet
8. –ip 192.168.56.1 : The ip for the switches to connect the remote controller

The sample output of this command is shown in Figure 2. As seen from the figure, mininet adds the controller, switches, links and then configures the hosts.

3 The Developed Controller and its Functionalities

We have our mininet up and running and the next task is to develop our controller that can traverse the fat tree topology in an efficient way. We use the POX [2] controller platform to execute our controller code. Our controller is designed

```
➜ ~  sudo mn --custom ~/mininet/custom/topo-fattree.py --topo fattree --mac --arp --s
witch ovsk --controller remote --ip 192.168.56.1
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8
*** Adding switches:
s9 s10 s11 s12 s13 s14 s15
*** Adding links:
(h1, s11) (h2, s11) (h3, s12) (h4, s12) (h5, s14) (h6, s14) (h7, s15) (h8, s15) (s10,
s9) (s11, s10) (s11, s13) (s12, s10) (s12, s13) (s13, s9) (s14, s10) (s14, s13) (s15,
s10) (s15, s13)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8
*** Starting controller
c0
*** Starting 7 switches
s9 s10 s11 s12 s13 s14 s15 ...
*** Starting CLI:
mininet>
```

Figure 2: Launching mininet

like a static controller where we statically assign IPs and ports to the switches. The controller knows this information and works with it. For example, if the dataflow path identifier is 11 i.e, Switch 11, then its port 1 and port 2 are connected to hosts with IPs 10.0.0.1 and 10.0.0.2 respectively.

```
if self.connection.dpid==11:
        self.host={
            "10.0.0.1":1,
            "10.0.0.2":2
        }
```

To solve the problem of fat tree with the traditional switches, in our controller, we have replaced flood by unicast, where we choose randomly which switch it has to forward to, either Switch 10 or Switch 13.

```
if random.randint(0,1)==0:
            port=self.switches[10]
        else:
            port=self.switches[13]
    return port
```

We have implemented an algorithm that statically selects the shortest path from one host to the other. We have divided the algorithm in two principals actors, switches level 2 and switches level 3. First, the switches in level 2 perform a search if the host is connected to them, and if not, it sends the packet to a switch of the next level. Switches level 3, checks to which switch is the host connected to and sends the packet to that switch alone.

To explain how we achieve the shortest path, consider a packet being sent from h1 to h8. The packet reaches S11 since we have statically assigned the switch port values. S11 can randomly choose whether to forward to S10 or S13. In either case the path will be the shortest path. In the Figure 3 and Figure 4, we demonstrate the cases when h1 sends packets to h8. The cost of the path h1->S11->S13->S15->h8 is 4 and the cost of the path h1->S11->S10->S15->h8 is also 4. In our algorithm we have layer two switches connected

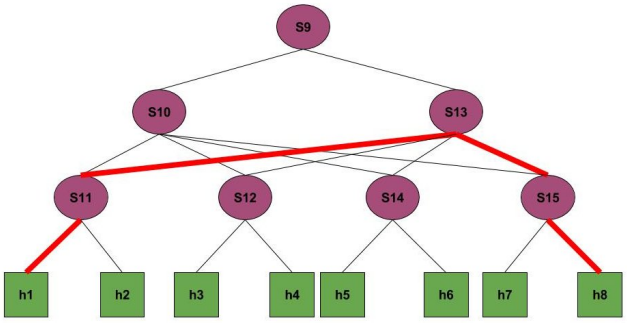

Figure 3: Shortest path from h1 to h8 via S13

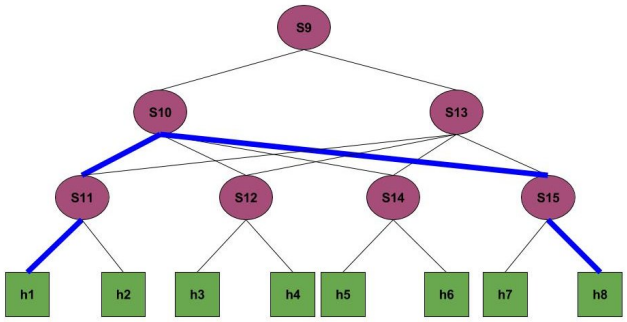

Figure 4: Shortest path from h1 to h8 via S10

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8
h2 -> h1 h3 h4 h5 h6 h7 h8
h3 -> h1 h2 h4 h5 h6 h7 h8
h4 -> h1 h2 h3 h5 h6 h7 h8
h5 -> h1 h2 h3 h4 h6 h7 h8
h6 -> h1 h2 h3 h4 h5 h7 h8
h7 -> h1 h2 h3 h4 h5 h6 h8
h8 -> h1 h2 h3 h4 h5 h6 h7
*** Results: 0% dropped (56/56 received)
mininet> h1 ping h8
PING 10.0.0.8 (10.0.0.8) 56(84) bytes of data.
64 bytes from 10.0.0.8: icmp_seq=1 ttl=64 time=12.3 ms
64 bytes from 10.0.0.8: icmp_seq=2 ttl=64 time=0.831 ms
64 bytes from 10.0.0.8: icmp_seq=3 ttl=64 time=0.129 ms
64 bytes from 10.0.0.8: icmp_seq=4 ttl=64 time=0.106 ms
64 bytes from 10.0.0.8: icmp_seq=5 ttl=64 time=0.097 ms
^C
--- 10.0.0.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 0.097/2.705/12.363/4.837 ms
mininet> h1 iperf -s &
mininet> h2 iperf -c h1
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.2 port 51334 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  23.6 GBytes  20.3 Gbits/sec
mininet>
```

Figure 5: Results of Ping and iperf test

to layer one and hence it does not flood to layer 3 as long as this connection exists. This avoids flooding, achieves shortest path and increases the performance. Moreover, it is secure as the message reaches the correct host and there is no possibility of spoofing.

We used an idle timeout of 30ms, which occurs when no packets are matched within the limit that we have set. We used a hard timeout value of 10ms, that occurs when the fixed period of time elapses irrespective of the the number of packets matched.

We measured the performance based on the ping test and iperf tests. Sample output of Ping and iperf test are as shown in the Figure 5. We can see that all hosts can reach each other and exchange packets. Ping from h1 to h8 takes an average round trip time(rtt) of 2.705ms and the minimum rtt is 0.097ms. The maximum rtt is for the first packet- 12.363ms, this is because the h1 tries to identify the switch that is connected to h8. If the connection doesn't exists, it floods the message.

The iperf test shows the bandwidth of 20.3Gb/sec for transferring 23.6Gb in a 10sec interval. This shows that the algorithm is quite efficient in handling large amounts of data and is suitable for a fat tree based data center networks as shown in Figure 1. With this simple yet efficient algorithm, we are able to achieve good performance and limited latency without compromising the security of the network.

4 Conclusion

In this report, we emulated a simple fat tree based data center network and also developed our own controller based on Pox, that manages the flow of packets within the network. The controller derives the shortest path forwarding between the hosts. Our controller works with the static assigned IPs and switch port values, basically performing a unicast forwarding. We have verified the performance using ping and iperf tests. We conclude that our algorithm avoids flooding in most cases which increases the performance and it also ensures confidentiality and integrity of the message. The performance shows that the algorithm is suitable for a fat tree based data center network.

References

[1] Mininet. WWW Introduction to Mininet: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet.

[2] Openflow networking lab. WWW Pox Wiki : https://openflow.stanford.edu/display/ONL/POX+Wiki.

[3] Sdn. WWW Slides of SDN lecture, Computer Networks- Avanced Features, Aalto University: https://mycourses.aalto.fi/pluginfile.php/101877/mod_folder/content/0/SDN.pdf?forcedownload=1.

[4] University of wisconsin-madison. WWW Assignment 4 : http://pages.cs.wisc.edu/~akella/CS838/F12/assignment4.html.

[5] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky. Advanced study of sdn/openflow controllers. In *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, page 1. ACM, 2013.