ELEC 6851 – Introduction to Telecommunication Networks

Term: Winter 2022

Project Title: Network Simulation Using Mininet

Submitted by:

Hassan Mahmood Khan (40216616)

Instructor: Chadi Assi

Date: 29th April 2022

Introduction

This project aims to simulate a Data Center Network (DCN) – which is a physical facility that is used by organizations to house critical applications and data, based on a network of computing and storage resources [1]. There exist many DCN topologies i.e., three tier, DCell, fat tree etc, each hosting a number of core components such as routers, switches, firewalls, servers, storage.

I will be implementing the Fat tree topology, proposed by M. Al-Fares et al [2], which is derived from Clos switching network; a multistage switching architecture that reduces the number of ports required in an interconnected fabric.

For simulation purposes, I will be using Mininet; a network emulator able to create virtual hosts, swictes, controllers, and links [3]. Mininet supports Software Defined Networking (SDN) and OpenFlow for custom routing.

In addition, I shall be employing the use of traffic generators for my virtual network and measuring various parameters for evaluation and analysis.

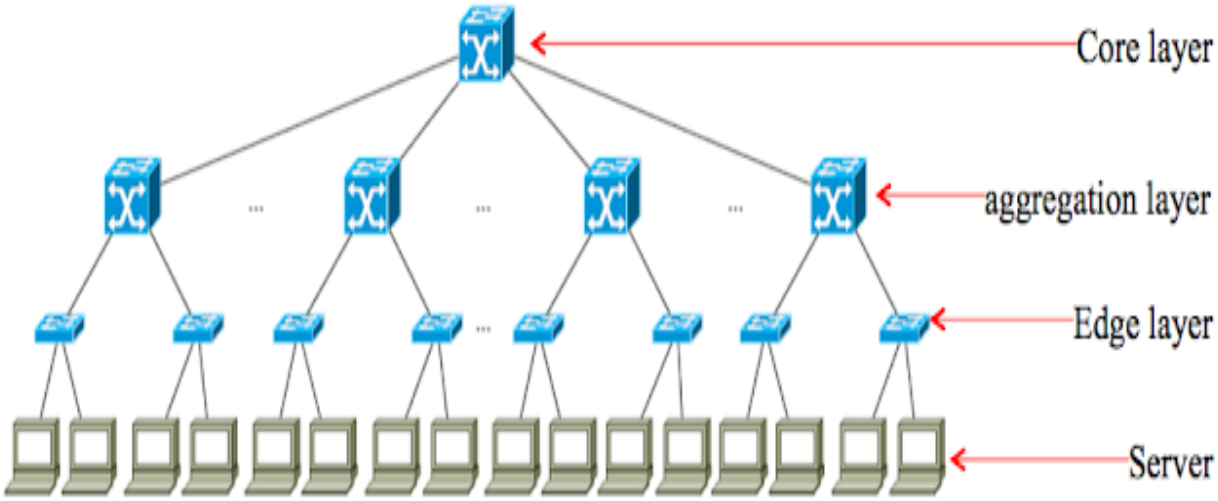
Fat Tree Network

The Fat tree topology is based on leveraging commodity Ethernet switches for interconnecting cluster nodes to achieve full bisection bandwidth or oversubscription ratio of 1:1. This means that all hosts may potentially communicate with other hosts at full bandwidth of their network interface. The distinctive feature of this topology as pointed out by M. Al Fares is its cost-effectiveness. It is able to deliver scalable bandwidth at moderate cost as opposed to other existing techiniques.

The Fat tree topology resembles the tree topology, where we have terminologies such as root, parent & child. Typical data center architectures have either a two-tier or three-tier trees of swicthes and routers. A two-tiered design has only the core and edge tiers, whilst the three-tier also has the aggregation tier.

We shall be looking at three-tier fat tree for my project. This type of topology has three layers as mentioned above i.e., core, aggregation, and edge. Switches (used to define devices that perform layer-2 swicthing) eixsting at all three levels of the architecture. For the purpose of this project, all switches have been assumed to be identical with equal number of ports.

The diagram below depicts a three-tier fat tree topology.



The fat tree topology provides a strong case for the use of small commodity switches as opposed to fewer larger ones due to the price differential.

Mininet

Mininet is a network emulator which creates a network of virtual hosts, switches, controllers and links. Mininet hosts run standard Linux network software and supports OpenFlow and Software Defined Networking (SDN) for flexible custom routing.

Mininet is been used widely for research, development, learning, testing, debuging and other purposes that can be simply implemented on a standard laptop/PC.

A few characteristics:

- Provides relatively simple network testbed
- System regressinon tests and complex topology testing are both supported
- Comes with standardized topologies but also supports arbitrary custom topologies
- It is a Command Line Interface (CLI) application and uses Python APIs for network creation and experimentation

However, there are a few limitations. Mininet-based networks cannot exceed the available bandwidth of a single server and cannot run non-Linux compatible applications.

Software Defined Networking

SDN is an approach to network architecture which allows the network to be centrally and logically controlled using using software applications. There are four core characteristics of this approach:

1. Generalized flow based forwarding as opposed to the conventional destination based addressing (e.g. Openflow)
2. Control and data plane seperation, with a remote controller interacting with local control agents to compute and distribute forwarding tables (centralized programming)
3. Control plane functions are transparent/external to the data plane switches (easier network management)
4. Programmable control applications allowing for open and flexible implementation.

OpenFlow

The OpenFlow protocol is the standard communications protocol that defines the communication between the SDN controller and network agents i.e., data plane switches, routers etc. Through this interface, the SDN controller queries switch features, configures switch parameters, add, delete, modify flow entries in the flow-table to allow network administrators to partition traffic, control flows for optimal performance, and start testing new configurations and applications [4].

Topology

I am constructing a three-layer fat tree network topology i.e., core, aggreagation, and edge layer. In order to contsruct such a topology, we must first know certain information. Each switch is identical for the purpose of this project with *k* number of ports. Thus, such a fat tree is also called k-port fat tree network topology.

From the value of k, we'll derive the number of core switches, aggregation switches, edge switches and the maximum number of hosts that can be attached [5].

There are *k* pods. A set of $(\frac{k}{2})$ number of aggregation switches and $(\frac{k}{2})$ number of edge switches are combined together and that is known as a pod.

Number of core switches = $(\frac{k}{2})^2$

Number of aggregation switches per pod = $(\frac{k}{2})$

Number of edge switches per pod = $(\frac{k}{2})$

Number of hosts connected to each pod = $(\frac{k}{2})^2$

For my project, I have chosen k=4, i.e., each switch has exactly 4 ports. We shall now obtain other parameter for our topology.

Pods = 4

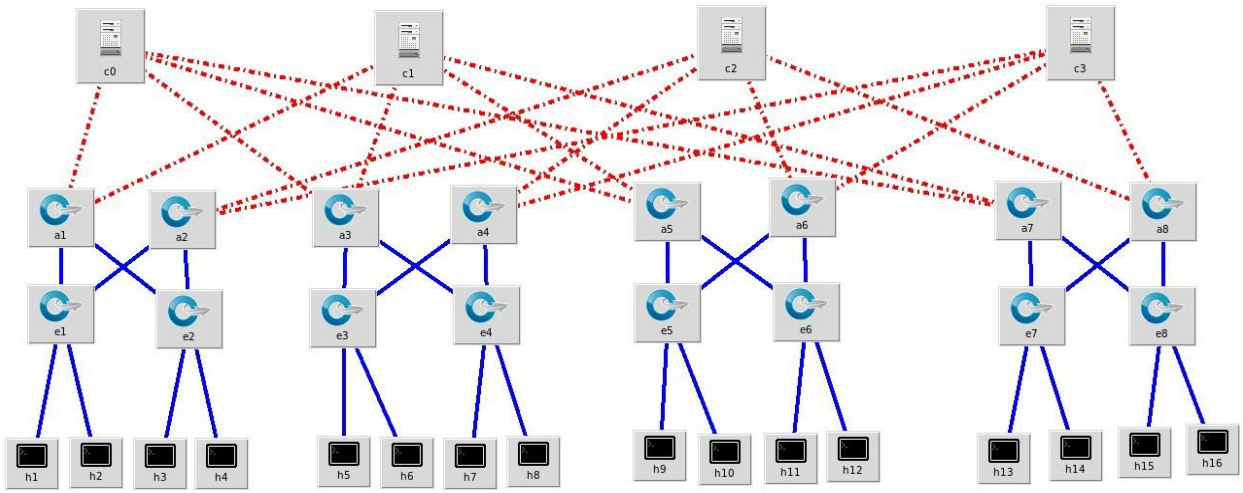
Core switches = 4

Aggregate switches per pod = 2, Total = 8

Edge switches per pod = 2, Total = 8

Hosts per pod = 4, Total = 16

Below is a snippet of the custom topology using Mininet's graphical user interface i.e., Miniedit.



Ryu Controller

For the purpose of this project, I have used a remote/external controller i.e., Ryu controller. This is due to the fact that Mininet's default controller supports upto 16 individual switches. Our simulation for k=4 has 20 switches in total.

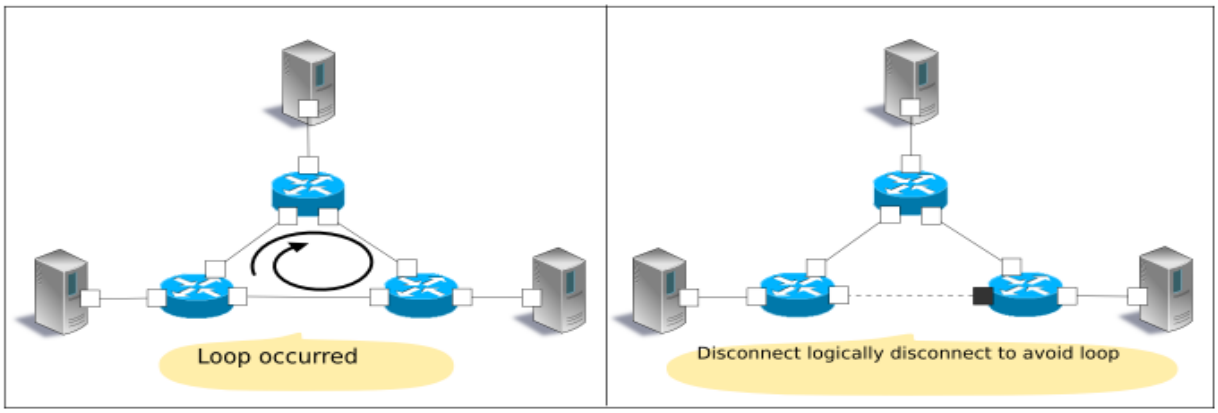
The Ryu controller, implemented in Python is an open source project maintained by open Ryu community on Github. It is compatible with SDN and is designed to increase the agility of the network. The controller runs in the background (manually started and shut down) and is connected to the custom topology on mininet. This is achieved by invoking the `RemoteController` class, which acts as a proxy for the controller.

The controller is used to implement simple learning switches or ethernet bridges. Each individual switch maintains its switch table which comprises of the MAC address of hosts, interface to reach host etc. It will flood packets on each link/interface that are missing in the switch table. In addition, they also flood broadcasts like ARP and DHCP. Since our topology presents a case of multi-path routing, this results in a loop problem and may use up all the resouces of the laptop/PC causing the system to crash.

Spanning Tree

This is solved using the Spanning Tree algorithm or Spanning Tree function. It suppresses the occurrence of broadcast streams in the network having loops, and is a means to secure network redundancy by automatically switching the path in case of a network failure.

For my project I have implemented the Spanning Tree Protocol (STP: IEEE 802.1D). It handles a network as a logical tree and by setting the ports of each switch to transfer frame or not, suppresses occurrence of broadcast streams in a network having a loop structure [6].



Simulation Process

Start the controller in the background and implement simple_switch_stp_13.py.

```
mininet@mininet-vm:~/ryu$ PYTHONPATH=. ./bin/ryu-manager ryu/app/simple_switch_stp_13.py
loading app ryu/app/simple_switch_stp_13.py
loading app ryu.controller.ofp_handler
instantiating app None of Stp
creating context stplib
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu/app/simple_switch_stp_13.py of SimpleSwitch13
```

Then start the mininet fat tree topology.

```
mininet@mininet-vm:~/mininet/custom$ sudo mn --custom ./fattree-2.py
--topo=mytopo --controller=remote,ip=127.0.0.1,port=6633 --mac --arp --link=tc
```

By default hosts start with randomly assigned MAC addresses everytime mininet is created, which makes debugging difficult. The *--mac* option sets host MAC and IP addresses to small, easy to read, unique IDs.

Since I have assumed that my nodes are static within the data center thus do not require dynamic mapping of IP to MAC addresses for ARP entries. The -- *arp* option permits that entires are not over-ridden , ensuring stability of network communication.

Furthermore, all links bandwidth has been customized to 10Mbps, thus reflecting the constraints in resorce availability and allocation in real life data center network topologies.

Displaying all nodes and links.

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=4756>
<Host h2: h2-eth0:10.0.0.2 pid=4758>
<Host h3: h3-eth0:10.0.0.3 pid=4760>
<Host h4: h4-eth0:10.0.0.4 pid=4762>
<Host h5: h5-eth0:10.0.0.5 pid=4764>
<Host h6: h6-eth0:10.0.0.6 pid=4766>
<Host h7: h7-eth0:10.0.0.7 pid=4768>
<Host h8: h8-eth0:10.0.0.8 pid=4770>
<Host h9: h9-eth0:10.0.0.9 pid=4772>
<Host h10: h10-eth0:10.0.0.10 pid=4774>
<Host h11: h11-eth0:10.0.0.11 pid=4776>
<Host h12: h12-eth0:10.0.0.12 pid=4778>
<Host h13: h13-eth0:10.0.0.13 pid=4780>
<Host h14: h14-eth0:10.0.0.14 pid=4782>
<Host h15: h15-eth0:10.0.0.15 pid=4784>
<Host h16: h16-eth0:10.0.0.16 pid=4786>
<OVSSwitch a5: lo:127.0.0.1,a5-eth1:None,a5-eth2:None,a5-eth3:None,a5-eth4:None pid=4791>
<OVSSwitch a6: lo:127.0.0.1,a6-eth1:None,a6-eth2:None,a6-eth3:None,a6-eth4:None pid=4794>
<OVSSwitch a7: lo:127.0.0.1,a7-eth1:None,a7-eth2:None,a7-eth3:None,a7-eth4:None pid=4797>
<OVSSwitch a8: lo:127.0.0.1,a8-eth1:None,a8-eth2:None,a8-eth3:None,a8-eth4:None pid=4800>
<OVSSwitch a9: lo:127.0.0.1,a9-eth1:None,a9-eth2:None,a9-eth3:None,a9-eth4:None pid=4803>
<OVSSwitch a10: lo:127.0.0.1,a10-eth1:None,a10-eth2:None,a10-eth3:None,a10-eth4:None pid=4806>
<OVSSwitch a11: lo:127.0.0.1,a11-eth1:None,a11-eth2:None,a11-eth3:None,a11-eth4:None pid=4809>
<OVSSwitch a12: lo:127.0.0.1,a12-eth1:None,a12-eth2:None,a12-eth3:None,a12-eth4:None pid=4812>
<OVSSwitch c1: lo:127.0.0.1,c1-eth1:None,c1-eth2:None,c1-eth3:None,c1-eth4:None pid=4815>
<OVSSwitch c2: lo:127.0.0.1,c2-eth1:None,c2-eth2:None,c2-eth3:None,c2-eth4:None pid=4818>
<OVSSwitch c3: lo:127.0.0.1,c3-eth1:None,c3-eth2:None,c3-eth3:None,c3-eth4:None pid=4821>
<OVSSwitch c4: lo:127.0.0.1,c4-eth1:None,c4-eth2:None,c4-eth3:None,c4-eth4:None pid=4824>
<OVSSwitch e13: lo:127.0.0.1,e13-eth1:None,e13-eth2:None,e13-eth3:None,e13-eth4:None pid=4827>
<OVSSwitch e14: lo:127.0.0.1,e14-eth1:None,e14-eth2:None,e14-eth3:None,e14-eth4:None pid=4830>
<OVSSwitch e15: lo:127.0.0.1,e15-eth1:None,e15-eth2:None,e15-eth3:None,e15-eth4:None pid=4833>
<OVSSwitch e16: lo:127.0.0.1,e16-eth1:None,e16-eth2:None,e16-eth3:None,e16-eth4:None pid=4836>
<OVSSwitch e17: lo:127.0.0.1,e17-eth1:None,e17-eth2:None,e17-eth3:None,e17-eth4:None pid=4839>
<OVSSwitch e18: lo:127.0.0.1,e18-eth1:None,e18-eth2:None,e18-eth3:None,e18-eth4:None pid=4842>
<OVSSwitch e19: lo:127.0.0.1,e19-eth1:None,e19-eth2:None,e19-eth3:None,e19-eth4:None pid=4845>
<OVSSwitch e20: lo:127.0.0.1,e20-eth1:None,e20-eth2:None,e20-eth3:None,e20-eth4:None pid=4848>
<RemoteController{'ip': '127.0.0.1', 'port': 6633} c0: 127.0.0.1:6633 pid=4750>
```

Each pair of host ping each other with pingall command.

```
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11 h12 h13 h14 h15 h16
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11 h12 h13 h14 h15 h16
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10 h11 h12 h13 h14 h15 h16
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h11 h12 h13 h14 h15 h16
h11 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h12 h13 h14 h15 h16
h12 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h13 h14 h15 h16
h13 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h14 h15 h16
h14 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h15 h16
h15 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h16
h16 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15
*** Results: 0% dropped (240/240 received)
```

Results & Analysis

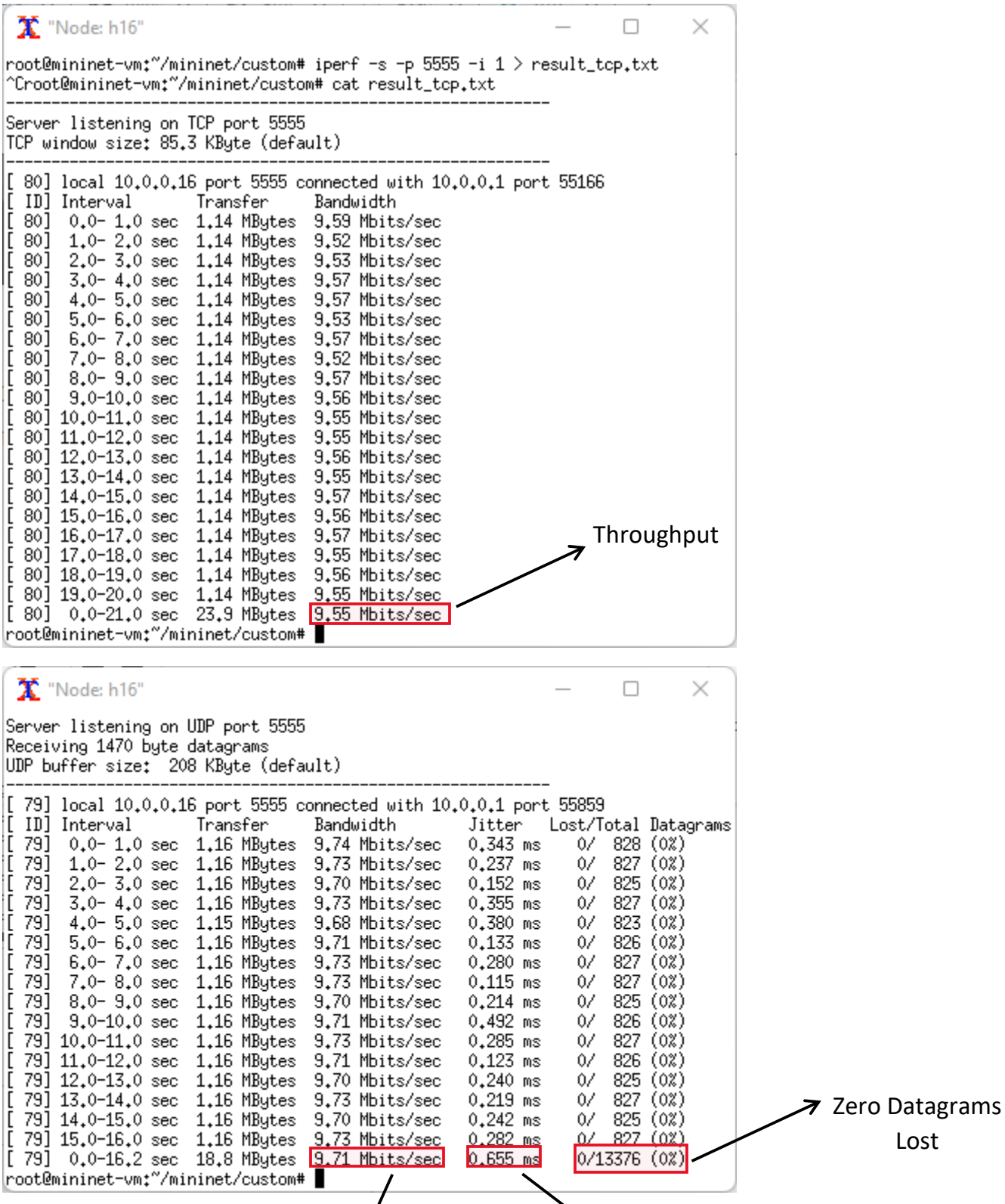
The fat tree network topology is successfully created and all links are active and working. I will now implement traffic generators for measuring performance parameters i.e., bisection bandwidth, throughput, delay etc.

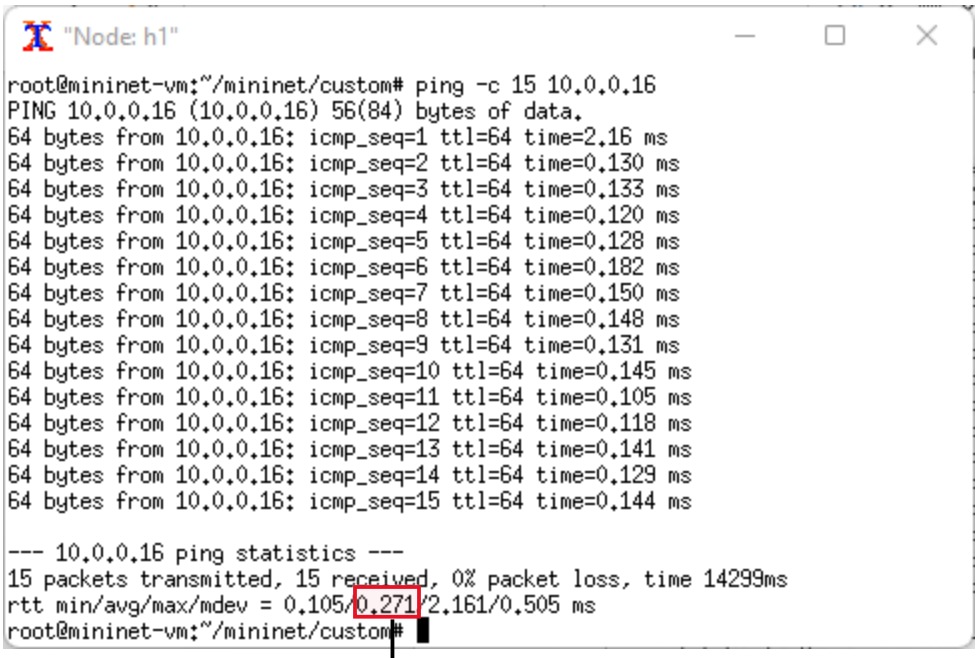
Iperf

I have employed the use of Iperf, which is a tool used for active measurement of maximum achieveable bandwidth in IP networks. It supports various parameters and protocols i.e., TCP, UDP etc. For each test it reports bandwidth, loss and other parameters.

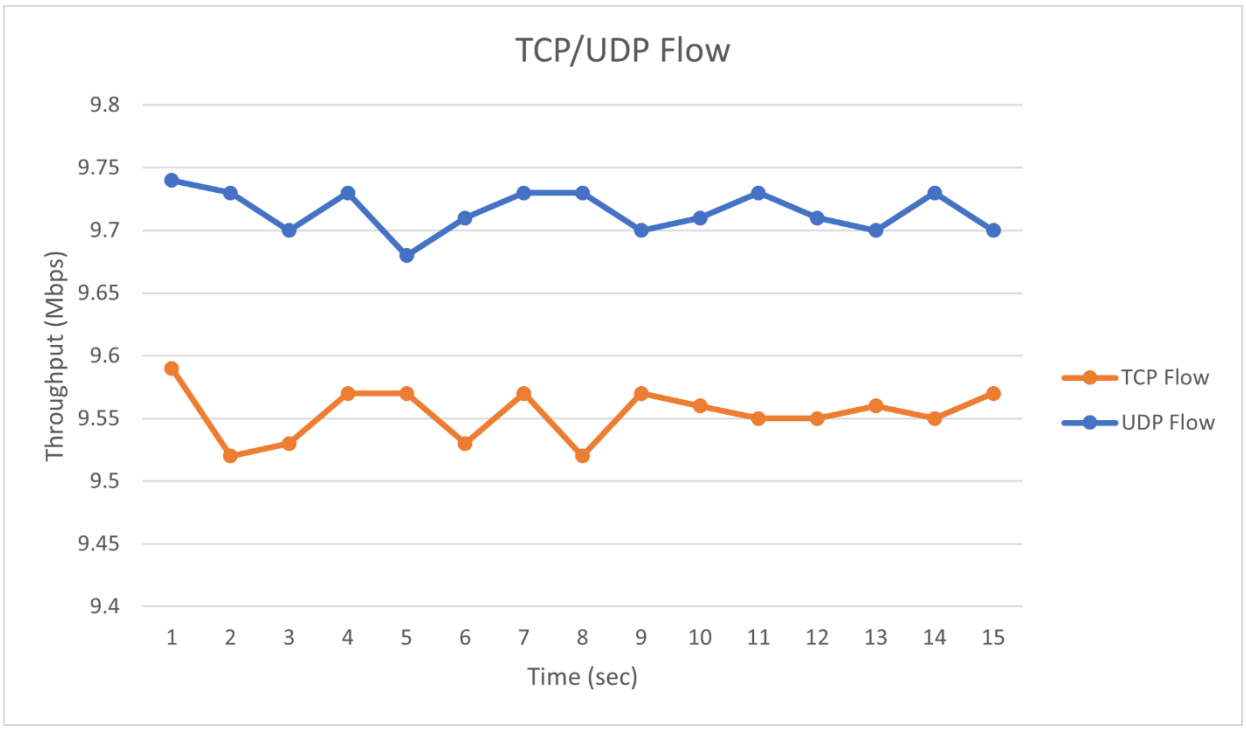
I have conducted two type of tests: TCP and UDP based tests. The following are the parameters measured:

- **Throughput**; the actual number of bits that flows through a network connection in a given period of time.
- **Rounded Trip Time (RTT) or Latency**; the amount of time it takes for a packet to make it from source to destination.
- **Jitter (latency variation) or Packet Delay Variation**; difference in packet delay.
- **Datagram Loss**
- **Bisection Bandwidth;** sum of the bandwidths of the minimal number of links that are cut when splitting the system into two parts.


"Node: h16"
root@mininet-vm:~/mininet/custom# iperf -s -p 5555 -i 1 > result_tcp.txt
^Croot@mininet-vm:~/mininet/custom# cat result_tcp.txt
Server listening on TCP port 5555
TCP window size: 85.3 KByte (default)
[80] local 10.0.0.16 port 5555 connected with 10.0.0.1 port 55166
[ID] Interval Transfer Bandwidth
[80] 0.0- 1.0 sec 1.14 MBytes 9.59 Mbits/sec
[80] 1.0- 2.0 sec 1.14 MBytes 9.52 Mbits/sec
[80] 2.0- 3.0 sec 1.14 MBytes 9.53 Mbits/sec
[80] 3.0- 4.0 sec 1.14 MBytes 9.57 Mbits/sec
[80] 4.0- 5.0 sec 1.14 MBytes 9.57 Mbits/sec
[80] 5.0- 6.0 sec 1.14 MBytes 9.53 Mbits/sec
[80] 6.0- 7.0 sec 1.14 MBytes 9.57 Mbits/sec
[80] 7.0- 8.0 sec 1.14 MBytes 9.52 Mbits/sec
[80] 8.0- 9.0 sec 1.14 MBytes 9.57 Mbits/sec
[80] 9.0-10.0 sec 1.14 MBytes 9.56 Mbits/sec
[80] 10.0-11.0 sec 1.14 MBytes 9.55 Mbits/sec
[80] 11.0-12.0 sec 1.14 MBytes 9.55 Mbits/sec
[80] 12.0-13.0 sec 1.14 MBytes 9.56 Mbits/sec
[80] 13.0-14.0 sec 1.14 MBytes 9.55 Mbits/sec
[80] 14.0-15.0 sec 1.14 MBytes 9.57 Mbits/sec
[80] 15.0-16.0 sec 1.14 MBytes 9.56 Mbits/sec
[80] 16.0-17.0 sec 1.14 MBytes 9.57 Mbits/sec
[80] 17.0-18.0 sec 1.14 MBytes 9.55 Mbits/sec
[80] 18.0-19.0 sec 1.14 MBytes 9.56 Mbits/sec
[80] 19.0-20.0 sec 1.14 MBytes 9.55 Mbits/sec
[80] 0.0-21.0 sec 23.9 MBytes 9.55 Mbits/sec
root@mininet-vm:~/mininet/custom#
Throughput
"Node: h16"
Server listening on UDP port 5555
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
[79] local 10.0.0.16 port 5555 connected with 10.0.0.1 port 55859
[ID] Interval Transfer Bandwidth Jitter Lost/Total Datagrams
[79] 0.0- 1.0 sec 1.16 MBytes 9.74 Mbits/sec 0.343 ms 0/ 828 (0%)
[79] 1.0- 2.0 sec 1.16 MBytes 9.73 Mbits/sec 0.237 ms 0/ 827 (0%)
[79] 2.0- 3.0 sec 1.16 MBytes 9.70 Mbits/sec 0.152 ms 0/ 825 (0%)
[79] 3.0- 4.0 sec 1.16 MBytes 9.73 Mbits/sec 0.355 ms 0/ 827 (0%)
[79] 4.0- 5.0 sec 1.15 MBytes 9.68 Mbits/sec 0.380 ms 0/ 823 (0%)
[79] 5.0- 6.0 sec 1.16 MBytes 9.71 Mbits/sec 0.133 ms 0/ 826 (0%)
[79] 6.0- 7.0 sec 1.16 MBytes 9.73 Mbits/sec 0.280 ms 0/ 827 (0%)
[79] 7.0- 8.0 sec 1.16 MBytes 9.73 Mbits/sec 0.115 ms 0/ 827 (0%)
[79] 8.0- 9.0 sec 1.16 MBytes 9.70 Mbits/sec 0.214 ms 0/ 825 (0%)
[79] 9.0-10.0 sec 1.16 MBytes 9.71 Mbits/sec 0.492 ms 0/ 826 (0%)
[79] 10.0-11.0 sec 1.16 MBytes 9.73 Mbits/sec 0.285 ms 0/ 827 (0%)
[79] 11.0-12.0 sec 1.16 MBytes 9.71 Mbits/sec 0.123 ms 0/ 826 (0%)
[79] 12.0-13.0 sec 1.16 MBytes 9.70 Mbits/sec 0.240 ms 0/ 825 (0%)
[79] 13.0-14.0 sec 1.16 MBytes 9.73 Mbits/sec 0.219 ms 0/ 827 (0%)
[79] 14.0-15.0 sec 1.16 MBytes 9.70 Mbits/sec 0.242 ms 0/ 825 (0%)
[79] 15.0-16.0 sec 1.16 MBytes 9.73 Mbits/sec 0.282 ms 0/ 827 (0%)
[79] 0.0-16.2 sec 18.8 MBytes 9.71 Mbits/sec 0.655 ms 0/13376 (0%)
root@mininet-vm:~/mininet/custom#
Zero Datagrams Lost
UDP Specified Bandwidth
Jitter/Packet Delay Variation



Average
Latency/RTT



We shall now look at the results obatined to draw conclusions.

Performance Metric	Result
Throughput	9.55 Mbps
Latency	0.271 ms
Jitter	0.655 ms
Datagram Loss	0%
Bisection Width	8 links for k=4
Bisection Bandwidth	8 * 10 Mbps = 80 Mbps

Based on the results obtained, we see that throughput achieved is satisfactory, given the limitations. However, due to significant latency and jitter, the throughput speed has been affected considerably. One way of addressing this and ensuring full bandwidth communication between arbitrary hosts, as described by in the paper 'A Scalable, Commodity Data Center Network Architecture' – is to employ multi-path routing techniques such as ECMP. Equal-cost multi-path routing (ECMP) is a routing strategy where packet forwarding to a single destination can occur over multiple best paths with equal routing priority. It aims to balance load flows to increase available bandwidth for usage.

Furthermore, I have modelled my topology to have fixed link bandwidth, this can cause the overall bandwidth to be limited by the bandwidth available at the root of the tree heirarchy. Single routing paths between source and destination can quickly lead to bottlenecks up and down the fat tree, limiting the overalll performance.

Often switches concentrate traffic to another subnet through a specific port even though other choices may exist with the same cost. This can cause congestion, especially if a small subclass of core switches are chosen as intermediary links between pods. Once again, a need for more fine-tuned traffic diffusion methodology is required which takes advantage of the fat tree structure.

This project enabled me to learn a great deal about the fat tree network topology, its parameters and various dynamics, how to maximize throughput both sustainably and efficiently, and get some valuable hands-on on tools such as mininet. In addition, I was able to conduct thorough literature review of the emerging techniques and discussions in this domain which has expanded my understanding of the topic area.

References

[1] "What is a data centre?" *Cisco*, 03-Sep-2021. [Online]. Available: https://www.cisco.com/c/en_ca/solutions/data-center-virtualization/what-is-a-data-center.html.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. SIGCOMM Comput. Commun. Rev. 38, 4 (October 2008), 63– 74. DOI: https://doi.org/10.1145/1402946.1402967

[3] M. P. Contributors, Mininet. [Online]. Available: http://mininet.org/.

[4] "What is openflow? definition and how it relates to Sdn." [Online]. Available: https://www.sdxcentral.com/networking/sdn/definitions/what-is-openflow/.

[5] C. Dehury, *what is Fat Tree and how to construct it in 4-steps?* 01-Jan-1970. [Online]. Available: https://blogchinmaya.blogspot.com/2017/04/what-is-fat-tree-and-how-to-construct.html.

[6] "Spanning tree," Spanning Tree - Ryubook 1.0 documentation. [Online]. Available: https://osrg.github.io/ryu-book/en/html/spanning_tree.html.

Appendix: Simulation Code

```
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.link import TCLink
from mininet.util import dumpNetConnections

class MyTopo(Topo):

    def __init__(self):
        super(MyTopo, self).__init__()

        #Number of swtiches per layer
        k = 2
        pod = k
        L1 = (pod//2)**2
        L2 = pod*pod//2
        L3 = L2

        #Creating switches
        c = [] #core layer switches
        a = [] #aggregation layer swtiches
        e = [] #edge layer switches

        #Add swicthes
        for i in range (L1):
             c_sw = self.addSwitch('c{}'.format(i+1))    #Label runs form 1 to n
             c.append(c_sw)

        for i in range (L2):
            a_sw = self.addSwitch('a{}'.format(L1+i+1))
            a.append(a_sw)


        for i in range (L3):
           e_sw = self.addSwitch('e{}'.format(L1+L2+i+1))
           e.append(e_sw)


        #Creating links between switches
        #The core layer and aggregation layer links
        for i in range(L1):
            c_sw = c[i]
            start = i%(pod//2)
```

```
            for j in range(pod):
                self.addLink(c_sw,a[start+j*(pod//2)],bw=10)

        #aggregation and edge layer links
        for i in range(L2):
            group = i//(pod//2)
            for j in range(pod//2):
                self.addLink(a[i],e[group*(pod//2)+j],bw=10)

        #Creating hosts and adding links between switches and hosts
        for i in range(L3):
            for j in range(2):
                hs = self.addHost('h{}'.format(i*2+j+1),bw=10)
                self.addLink(e[i],hs)

topos = {"mytopo":(lambda:MyTopo())}
```