



西安交通大学
XI'AN JIAOTONG UNIVERSITY

软件定义网络实验报告

Lab1 Fattree

课程名称:	软件定义网络
姓名:	曾锦程
学院:	计算机学院
专业:	计算机科学与技术
学号:	2203613040
指导老师:	张鹏

2023 年 3 月 15 日

西安交通大学实验报告

专业： 计算机科学与技术
姓名： 曾锦程
学号： 2203613040
日期： 2023 年 3 月 15 日
地点： Personal Device

课程名称： 软件定义网络 指导老师： 张鹏 成绩：
实验名称： Lab1 Fattree 实验类型： 同组学生姓名：

一、 实验目的和要求

- (1) 使用 Mininet 的 Python API 搭建 $k=4$ 的 fat tree 拓扑；
- (2) 使用 pingall 查看各主机之间的连通情况；
- (3) 若主机之间未连通，分析原因并解决（使用 wireshark 抓包分析）
- (4) 若主机连通，分析数据包的路径（提示：ovs-appctl fdb/show 查看 MAC 表）
- (5) 完成实验报告并提交到思源学堂
- (6) 要求不能使用控制

二、 实验内容和步骤

- 编写 Fattree.py，使用 python 接口来搭建 Fattree 网络
- 使用 pingall 指令，测试网络是否 ping 通
- 使用 Wireshark 抓包分析网络

三、 实验环境

虚拟机 Virtual Box, Mininet, OVS, Wireshark。

四、 实验原理分析

1. Fattree 介绍

Fattree 是一种网络拓扑结构，它被广泛应用于数据中心网络中，可以提供高带宽、低延迟、可扩展的网络连接。Fattree 拓扑结构由多个层级组成，每个层级由多个交换机组成，交换机之间通过链路连接。Fattree 拓扑结构包含三个层级：边缘层（edge layer）、聚合层（aggregation layer）和核心层（core layer）。

在 Fattree 拓扑结构中，边缘层连接着数据中心的端口，聚合层提供了高速的连接，同时在核心层通过最短路径提供最快速的转发速度。在 Fattree 拓扑结构中，每个交换机的端口数量相同，交换机的规模是固定的，因此它具有可扩展性和高度的容错性。

Fattree 拓扑结构的优点在于，它可以提供非阻塞的带宽和低延迟的网络连接，同时具有可扩展性和容错性。它适用于大型数据中心，特别是那些需要高性能和高可用性的应用程序，如云计算、大数据和机器学习等。

2. 拓扑图分析

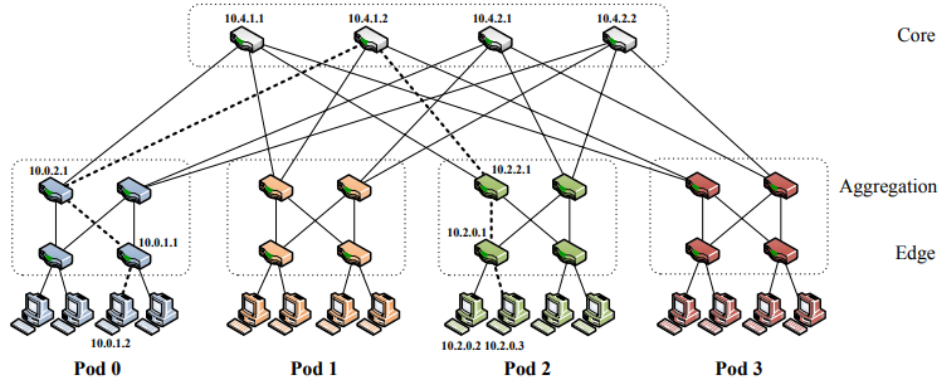


图 1: Fattree(k=4)

(1) 节点个数分析

- (i) Core Switch 的个数为: $\frac{k^2}{4}$
- (ii) Pod 数量为 k 个
- (iii) 每个 Pod 中含有 Aggregation Switch 个数为 $\frac{k}{2}$
- (iv) 每个 Pod 中含有 Edge Switch 个数为 $\frac{k}{2}$
- (v) 每个 Pod 中含有的 Host 个数为 $\frac{k^2}{4}$

(2) 路由分析

- (i) 对于 Core 层与 Aggregation 层分析如下:

对于每个 Core Switch, 需要和 $k/2$ 个 Aggregation Switch 连接, 而连接的 Aggregation 的编号在每个 Pod 中相同, 编号计算方法为:

$$AggregationNumber = CoreNumber // (k/2)$$

- (ii) 对于 Aggregation 层与 Edge 层分析如下:

对于每个 Aggregation Switch 都需要和本 Pod 中的每个 Edge Switch 连接。

- (iii) 对于 Edge 层与 Host 层分析如下:

对于每个 Pod 中的 Edge Switch 都需要负责连接本 Pod 中的 $k/2$ 个 Host, Host 的编号计算方法为:

$$HostNumber_{Begin} = EdgeNumber * (k/2)$$

$$HostNumber_{End} = EdgeNumber * (k/2) + k/2 - 1$$

3. 工具介绍

(1) Mininet

Mininet 是一个用于网络仿真的开源工具, 它能够模拟一个包括多个主机、交换机、路由器等网络设备的虚拟网络。使用 Mininet, 用户可以在自己的计算机上快速搭建一个网络环境, 进行网络应用的

测试、开发和教学等工作。

Mininet 的虚拟网络是基于 Linux 内核实现的, 并且使用 Open vSwitch 作为虚拟交换机。它提供了丰富的 API 和命令行工具, 使得用户可以方便地创建、配置和控制虚拟网络。用户可以在虚拟网络中安装各种网络应用程序, 如路由协议、拥塞控制算法、数据中心网络等, 并进行测试和评估。

Mininet 还提供了一些方便的工具和库, 如网络拓扑生成器、流量发生器、OpenFlow 控制器等, 使得用户可以快速地搭建不同类型的虚拟网络, 并进行灵活的网络测试和研究。

Mininet 的优点在于它具有开源、灵活、易用等特点, 可以在本地环境中快速构建网络实验室, 节省了实验成本和时间。因此, Mininet 被广泛应用于网络教育、网络研究和网络应用开发等领域。

(2) OVS

OVS (Open vSwitch) 是一款开源的软件交换机, 它提供了多种网络虚拟化技术和高级网络功能, 如流量控制、负载均衡、网络隔离和安全等。OVS 可以用于虚拟化环境、数据中心网络、云计算和 SDN (软件定义网络) 等场景。

OVS 的核心是一个模块化的软件交换机, 它可以在 Linux 内核中作为一个内核模块运行。OVS 还提供了一个用户空间的管理工具 (ovs-vsctl), 用于管理 OVS 交换机的配置和运行状态。OVS 支持多种数据平面, 如 Linux 内核数据平面、DPDK (Data Plane Development Kit) 数据平面和 NetFlow 数据平面等, 可以根据需求选择适合的数据平面。

OVS 支持多种网络虚拟化技术, 如 VLAN、VXLAN、GRE、Geneve 等, 可以在虚拟网络中实现多租户隔离和跨物理网络的互联。OVS 还支持 OpenFlow 协议, 可以通过 OpenFlow 控制器实现 SDN 网络控制和编程。OVS 还提供了多种高级网络功能, 如流量镜像、流量过滤、QoS (Quality of Service) 和负载均衡等, 使得网络管理更加灵活和高效。

总之, OVS 是一款功能强大、灵活可扩展的软件交换机, 它提供了多种网络虚拟化技术和高级网络功能, 可以满足不同场景下的网络需求。

五、 实验过程

1. 使用 python API 来构建 fattree

本实验要求为搭建 $k=4$ 的 fattree, 实际上做了上述对 fattree 的拓扑分析后, 可以发现很容易就做出 $k=n$ 的 fattree, 于是将代码扩充成符合 k 为任意符合条件值的 fattree。 (完整代码见附录)

```
Fattree.py x
9 class fattree(Topo):
10     def build(self,k=4):
11         #Core Switch
12         core_number = k*k/4
13         cores = []
14         for i in range(core_number):
15             core = self.addSwitch('Core S'+str(i))
16             cores.append(core)
17
18
19         #Pods
20         Pod_number = k
21         Aggregation_number = k/2
22         Edge_number = k/2
23         Host_number = k*k/4
24         Pods = []
25         for i in range(Pod_number):
26             Pod_struct = []
27
28
29         ##Aggregation
30         Pod_As = []
31         for j in range(Aggregation_number):
32             Pod_A_s = self.addSwitch('Pod'+str(i)+' A S'+str(j))
33             Pod_As.append(Pod_A_s)
34
35
36         ##Edge
37         Pod_Es = []
```

图 2: 代码片段

图 3: Fattree(k=2)

图 4: Fattree(k=4)

图 5: Fattree($k=6$)

之前没有配置，导致 pingall unreachable，对使用 xterm 指令，打开 h0，让其 ping 10.2，并在 h0 进行抓包分析。

图 6: h0 ping 10.2

3	2.048067290	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
4	3.071998606	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
5	4.096015802	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
6	5.129037033	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
7	6.144006713	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
8	7.167998797	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
9	8.192045714	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
10	9.216067023	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
11	10.239994817	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
12	11.264043396	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
13	12.287995783	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
14	13.312018846	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
15	14.336053937	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
16	15.359997232	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
17	16.384287543	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
18	17.408058562	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
19	18.432046707	86:8b:69:65:e8:6a	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1

图 7: h0 wireshark

发现在 ping 的过程中, 主机一直在广播 ARP 包, 但是没有网关的回应, 说明是 STP 协议未配置, 路由器并没有学习到拓扑结构, 所以无法应答 ARP, 加入以下代码, 让其自动给每个路由器配置 STP 协议, 再观察结果。

```

1 for i in range(core_number):
2     os.system('sudo ovs-vsctl set bridge '+'Core_s'+str(i)+' stp_enable=true')
3     os.system('sudo ovs-vsctl del-fail-mode '+'Core_s'+str(i))
4 for i in range(Pod_number):
5     for j in range(Aggregation_number):
6         os.system('sudo ovs-vsctl set bridge '+'Pod'+str(i)+'_A_s'+str(j)+'
7             stp_enable=true')
8         os.system('sudo ovs-vsctl del-fail-mode '+'Pod'+str(i)+'_A_s'+str(j))
9     for j in range(Edge_number):
10        os.system('sudo ovs-vsctl set bridge '+'Pod'+str(i)+'_E_s'+str(j)+'
11            stp_enable=true')
12        os.system('sudo ovs-vsctl del-fail-mode '+'Pod'+str(i)+'_E_s'+str(j))

```

使用命令 pingall, 发现成功 ping 通所有路径。

```

mininet> pingall
*** Ping: testing ping reachability
Pod0_h0 -> Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod0_h1 -> Pod0_h0 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod0_h2 -> Pod0_h0 Pod0_h1 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod0_h3 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod1_h0 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod1_h0 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod1_h1 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod1_h2 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod1_h3 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h2 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod2_h0 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod2_h1 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod2_h2 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod2_h3 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod3_h0 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h1 Pod3_h2 Pod3_h3
Pod3_h1 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h2 Pod3_h3
Pod3_h2 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
Pod3_h3 -> Pod0_h0 Pod0_h1 Pod0_h2 Pod0_h3 Pod1_h0 Pod1_h1 Pod1_h2 Pod1_h3 Pod2_h0 Pod2_h1 Pod2_h2 Pod2_h3 Pod3_h0 Pod3_h1 Pod3_h2 Pod3_h3
*** Results: 9% dropped (240/240 received)

```

图 8: pingall

10	8.438867015	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request	id=0x2391, seq
11	8.439284184	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply	id=0x2391, seq
12	8.440963921	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) request	id=0x2392, seq
13	8.441228628	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply	id=0x2392, seq
14	8.442777214	10.0.0.1	10.0.0.4	ICMP	98	Echo (ping) request	id=0x2393, seq
15	8.442999651	10.0.0.4	10.0.0.1	ICMP	98	Echo (ping) reply	id=0x2393, seq
16	8.444217993	10.0.0.1	10.0.0.5	ICMP	98	Echo (ping) request	id=0x2394, seq
17	8.444471544	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) reply	id=0x2394, seq
18	8.445761064	10.0.0.1	10.0.0.6	ICMP	98	Echo (ping) request	id=0x2395, seq
19	8.446098906	10.0.0.6	10.0.0.1	ICMP	98	Echo (ping) reply	id=0x2395, seq
20	8.447696196	10.0.0.1	10.0.0.7	ICMP	98	Echo (ping) request	id=0x2396, seq
21	8.447992730	10.0.0.7	10.0.0.1	ICMP	98	Echo (ping) reply	id=0x2396, seq
22	8.449420348	10.0.0.1	10.0.0.8	ICMP	98	Echo (ping) request	id=0x2397, seq
23	8.449693375	10.0.0.8	10.0.0.1	ICMP	98	Echo (ping) reply	id=0x2397, seq
24	8.451803900	10.0.0.1	10.0.0.9	ICMP	98	Echo (ping) request	id=0x2398, seq
25	8.453510594	10.0.0.9	10.0.0.1	ICMP	98	Echo (ping) reply	id=0x2398, seq
26	8.455101138	10.0.0.1	10.0.0.10	ICMP	98	Echo (ping) request	id=0x2399, seq

图 9: wireshark

同时打开主机 h0,ping 10.16, 发现 ping 通, 而且抓包分析可以看出, 主机发送 ARP 包, 成功获得

了交换机的应答!

```
root@sdnexp:~/Desktop# ping 10.16
PING 10.16 (10.0.0.16) 56(84) bytes of data.
64 bytes from 10.0.0.16: icmp_seq=1 ttl=64 time=0.675 ms
64 bytes from 10.0.0.16: icmp_seq=2 ttl=64 time=0.092 ms
64 bytes from 10.0.0.16: icmp_seq=3 ttl=64 time=0.138 ms
64 bytes from 10.0.0.16: icmp_seq=4 ttl=64 time=0.054 ms
64 bytes from 10.0.0.16: icmp_seq=5 ttl=64 time=0.082 ms
64 bytes from 10.0.0.16: icmp_seq=6 ttl=64 time=0.060 ms
64 bytes from 10.0.0.16: icmp_seq=7 ttl=64 time=0.092 ms
64 bytes from 10.0.0.16: icmp_seq=8 ttl=64 time=0.072 ms
64 bytes from 10.0.0.16: icmp_seq=9 ttl=64 time=0.056 ms
```

图 10: h0 ping 10.16

53	19.388124696	7e:4b:47:f4:80:a7	Broadcast	ARP	42	Who has 10.0.0.16? Tell 10.0.0.1
54	19.388846821	92:00:cb:8e:63:d0	7e:4b:47:f4:80:a7	ARP	42	10.0.0.16 is at 92:00:cb:8e:63:d0
55	19.388849928	10.0.0.1	10.0.0.16	ICMP	98	Echo (ping) request id=0x2a63, seq=1/256,
56	19.388994591	10.0.0.16	10.0.0.1	ICMP	98	Echo (ping) reply id=0x2a63, seq=1/256,
57	20.389069465	10.0.0.1	10.0.0.16	ICMP	98	Echo (ping) request id=0x2a63, seq=2/512,
58	20.389103466	10.0.0.16	10.0.0.1	ICMP	98	Echo (ping) reply id=0x2a63, seq=2/512,

图 11: h0 wireshark

3. 分析数据包的路径

先使用 `xx ifconfig` 指令查看主机的 mac 地址:

```
mininet> Pod0_h0 ifconfig
Pod0_h0-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 10.0.0.1 netmask 255.0.0.0 broadcast 10.255.255.255
inet6 fe80::200:ff:fe00:1 prefixlen 64 scopeid 0x20<link>
ether 00:00:00:00:00:01 txqueuelen 1000 (Ethernet)
RX packets 634 bytes 62188 (62.1 KB)
RX errors 0 dropped 262 overruns 0 frame 0
TX packets 27 bytes 2298 (2.2 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

图 12: h0 ifconfig

```
mininet> Pod3_h3 ifconfig
Pod3_h3-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 10.0.0.16 netmask 255.0.0.0 broadcast 10.255.255.255
inet6 fe80::200:ff:fe00:10 prefixlen 64 scopeid 0x20<link>
ether 00:00:00:00:00:10 txqueuelen 1000 (Ethernet)
RX packets 619 bytes 59240 (59.2 KB)
RX errors 0 dropped 264 overruns 0 frame 0
TX packets 27 bytes 2318 (2.3 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

图 13: h15 ifconfig

可以看出 h0 的 mac 地址为 00:01,h15 的 mac 地址为 00:10, 现以 h0 向 h15ping, 分析该数据包的路径, 查看所有交换机的 Mac 表 (调用 Macshow.py, 见附录)。

```

2 0 ee:cf:0b:db:b4:18 35
2 0 00:00:00:00:00:01 9
-----
Pod3_E_s1
port VLAN MAC Age
2 0 00:00:00:00:00:08 111
2 0 00:00:00:00:00:07 105
2 0 00:00:00:00:00:06 103
3 0 00:00:00:00:00:0f 99
2 0 00:00:00:00:00:0d 99
2 0 00:00:00:00:00:0b 95
2 0 00:00:00:00:00:03 93
2 0 76:44:2e:b6:de:6d 86
2 0 4e:38:d3:8d:c5:ed 86
2 0 b2:6d:80:3a:2f:13 86
2 0 7e:1c:b2:3b:45:31 86
2 0 16:ca:a0:77:98:1c 86
2 0 02:e5:c3:61:0d:b2 85

```

图 14: mac show 实例

六、实验结果分析

1. 分析数据包的路径

查阅所有交换机的 mac 表, 只要同时包含 00:01 和 00:10 两项, 说明数据包通过该交换机, 于是可以得出下表:

Hop	Hop-1	Hop-2	Hop-3	Hop-1	Hop-2	Hop-3	Hop-4
00:01->00:10	h0	Pod0-E-s0	Pod0-A-s1	Core-s3	Pod3-A-s1	Pod3-E-s1	h15

表 1: 数据包传输路径表

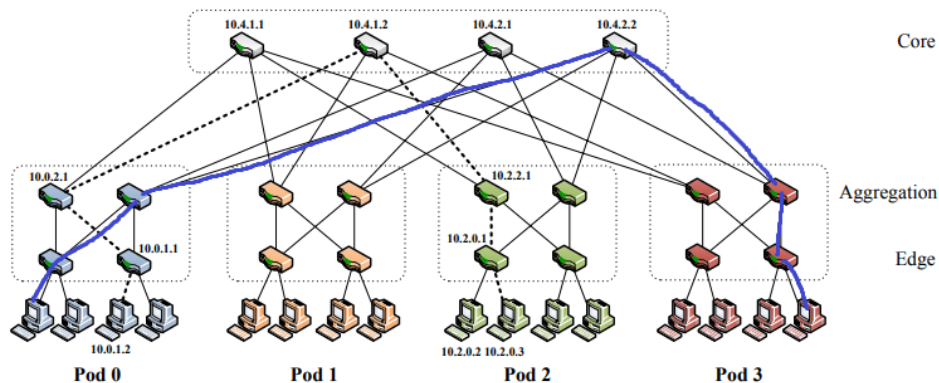


图 15: 数据包传输路径图

其中 Edge 层与主机层路径和主机编号有关, Aggregation 层与 Edge 层路径和 Edge switch 所连主机有关, Core 与 Aggregation 层路径和 Core Switch 编号有关。

七、实验感悟

- (1) 基本了解了在 linux 系统调用 mininet 的命令, 以及一些基本 shell 命令的调用

- (2) 对 Fattree 拓扑结构有了更深的理解
- (3) 学会了通过查看路由器 MAC 表来分析数据包路径的方法
- (4) 学会了 Python 面对对象构建网络拓扑的方法, 学会了如何在 python 中调用命令行, 对 python 有了更深的理解。

八、 附录

Fattree.py

```
1 from mininet.topo import Topo
2 from mininet.net import Mininet
3 from mininet.cli import CLI
4 from mininet.log import setLogLevel
5 import os
6 import commands
7
8
9 class fattree(Topo):
10     def build(self,k=4):
11         #Core Switch
12         core_number = k*k/4
13         cores = []
14         for i in range(core_number):
15             core = self.addSwitch('Core_s'+str(i))
16             cores.append(core)
17
18
19         #Pods
20         Pod_number = k
21         Aggregation_number = k/2
22         Edge_number = k/2
23         Host_number = k*k/4
24         Pods = []
25         for i in range(Pod_number):
26             Pod_struct = []
27
28
29         ##Aggregation
30         Pod_As = []
31         for j in range(Aggregation_number):
32             Pod_A_s = self.addSwitch('Pod'+str(i)+'_A_s'+str(j))
33             Pod_As.append(Pod_A_s)
34
35
36         ##Edge
37         Pod_Es = []
38         for j in range(Edge_number):
39             Pod_E_s = self.addSwitch('Pod'+str(i)+'_E_s'+str(j))
40             Pod_Es.append(Pod_E_s)
```

```

41
42
43     ##Host
44     Pod_Hs = []
45     for j in range(Host_number):
46         Pod_H = self.addHost('Pod'+str(i)+'_h'+str(j))
47         Pod_Hs.append(Pod_H)
48         Pod_struct.append(Pod_As)
49         Pod_struct.append(Pod_Es)
50         Pod_struct.append(Pod_Hs)
51         Pods.append(Pod_struct)
52
53
54     #Core Link
55     for i in range(len(cores)):
56         a_number = i//(k/2)
57         for j in range(len(Pods)):
58             self.addLink(cores[i],Pods[j][0][a_number])
59
60
61     #Pod Switch Link
62     for i in range(len(Pods)):
63         for j in range(len(Pods[0][0])):
64             for k in range(len(Pods[0][1])):
65                 self.addLink(Pods[i][0][j],Pods[i][1][k])
66
67
68     #Host Link
69     Host_number = Pod_number
70     for i in range(len(Pods)):
71         for j in range(len(Pods[0][1])):
72             h_number = j*Host_number/2
73             for k in range(Host_number/2):
74                 self.addLink(Pods[i][1][j],Pods[i][2][k+h_number])
75
76
77
78 def run(k=4):
79     topo = fattree(k)
80     net = Mininet(topo,controller=None)
81     core_number = k*k/4
82     Pod_number = k
83     Aggregation_number = k/2
84     Edge_number = k/2
85     Host_number = k
86     for i in range(core_number):
87         os.system('sudo ovs-vsctl set bridge '+'Core_s'+str(i)+' stp_enable=true')
88         os.system('sudo ovs-vsctl del-fail-mode '+'Core_s'+str(i))
89     for i in range(Pod_number):
90         for j in range(Aggregation_number):

```

```

91         os.system('sudo ovs-vsctl set bridge '+'Pod'+str(i)+'_A_s'+str(j)+'
           stp_enable=true')
92         os.system('sudo ovs-vsctl del-fail-mode '+'Pod'+str(i)+'_A_s'+str(j))
93     for j in range(Edge_number):
94         os.system('sudo ovs-vsctl set bridge '+'Pod'+str(i)+'_E_s'+str(j)+'
           stp_enable=true')
95         os.system('sudo ovs-vsctl del-fail-mode '+'Pod'+str(i)+'_E_s'+str(j))
96
97     net.start()
98     CLI(net)
99     net.stop()
100
101
102 if __name__ == '__main__':
103     k = int(input('input the k of the fattree topo:'))
104     setLogLevel('info')
105     run(k)

```

Macshow.py

```

1  import os
2  import commands
3
4  def macshow(k=4):
5      core_number = k*k/4
6      Pod_number = k
7      Aggregation_number = k/2
8      Edge_number = k/2
9      for i in range(core_number):
10         print('-----')
11         print('Core_s'+str(i))
12         os.system('sudo ovs-appctl fdb/show '+'Core_s'+str(i))
13     for i in range(Pod_number):
14         for j in range(Aggregation_number):
15             print('-----')
16             print('Pod'+str(i)+'_A_s'+str(j))
17             os.system('sudo ovs-appctl fdb/show '+'Pod'+str(i)+'_A_s'+str(j))
18         for j in range(Edge_number):
19             print('-----')
20             print('Pod'+str(i)+'_E_s'+str(j))
21             os.system('sudo ovs-appctl fdb/show '+'Pod'+str(i)+'_E_s'+str(j))
22
23
24 if __name__ == '__main__':
25     k = int(input('input the k of the fattree topo:'))
26     macshow(k)

```