

一、实验任务

- (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（`ovs-appctl fdb/show` 查看 MAC 表）；
- (5) 完成实验报告并提交到思源学堂，要求不能使用控制器。

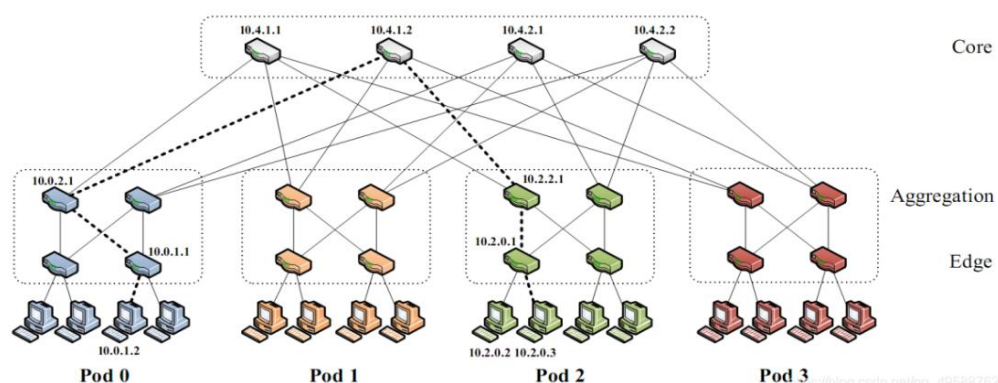
二、实验原理

(1) Fat Tree 拓扑结构

Fat Tree 是一种交换机为中心的网络拓扑，旨在支持横向扩展的同时增加网络路径数目，并且所有交换机具有相同数量的端口。这一设计降低了网络建设成本。Fat Tree 结构的特点包括核心层通过多条链路处理负载，避免网络热点的产生、通过合理的流量分配，避免 pod 内的过载问题、随着网络规模的扩大，带宽增加，可以提供高吞吐量的服务、不同 pod 之间的服务器通信通过多条并行路径，具有良好的容错能力，避免单点故障、采用商用设备替代高性能交换设备，显著降低网络设备成本等

Fat Tree 结构分为三层：核心层、汇聚层和接入层。一个 k 元的 Fat Tree 中每台交换机有 k 个端口。核心层顶层有 $(k/2)^2$ 个交换机。共有 k 个 pod，每个 pod 有 k 台交换机，其中汇聚层和接入层各有 $k/2$ 台交换机。每个接入层交换机可容纳 $k/2$ 台服务器，每个 pod 总共可容纳 $k^3/4$ 台服务器。任意两个 pod 之间存在 k 条路径。

Fat Tree $k=4$



(2) Mininet 网络仿真工具

Mininet 是斯坦福大学基于 Linux Container 架构开发的网络仿真工具。它允许用户创建包含主机、交换机、控制器和链路的虚拟网络，支持 OpenFlow 协议，适合自定义软件定义网络的实验。Mininet 的主要功能包括：网络测试平台、拓扑测试、调试功能、自定义拓扑、Python API 等。

(3) Open vSwitch (OVS)

Open vSwitch 是一个高质量、多层次的虚拟交换机软件，旨在通过编程支持大规模网络自动化，同时与多个物理机的分布式环境兼容。虽然它是虚拟交换机，但其工作原理与物理交换机类似。

OVS 连接物理网卡和多个虚拟网卡，在内部维护一个映射表，通过 MAC 地址查找并转发数据。OVS 支持访问控制功能，通过转发规则实现简单的安全行为，如允许或禁止某些流量。同时，OVS 支持多层交换，可以在虚拟环境中高效地实现数据包转发和流量控制。

OVS 的设计不仅提高了虚拟化环境中的网络性能，还通过 OpenFlow 协议为 SDN 提供了强大的支持。

三、实验过程

(1) 编写 FatTree.py 使用 Mininet 的 Python API 搭建 $k=4$ 的 fat tree 拓扑。如下图所示，网络拓扑被成功建立。

其中核心交换机设置 $\text{core_switch_num} = 4$ ，根据公式 $(k/2)^2 = 4(k/2)^2 = 4$ ，解得 $k=4$ 。而 $\text{pod} = 4$ ，每个 pod 的交换机数量 $\text{pod_switch_ae_num} = 2$ 则每个 pod 的汇聚和边缘交换机数量符合 $k/2 = 2k/2 = 2$ 。每个 pod 的主机数量为 $\text{pod_host_num} = 4$ 有每个 pod 的主机数量。

例如，要让 $k=2$ ，则将参数调换为 2，执行结果如下所示。

```
nakano9331@nakano9331-VirtualBox:~$ sudo python3 FatTree.py
*** Creating network
*** Adding hosts:
h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
*** Adding switches:
Arpod0no0 Arpod0no1 Arpod1no0 Arpod1no1 Arpod2no0 Arpod2no1 Arpod3no0 Arpod3no1 Edpod0no0 Edpod0no1 Edpod1no0 Edpod1no1 Edpod2no0 Edpod2no1 Edpod3no0 Edpod3no1 core0 core1 core2 core3
*** Adding links:
(Arpod0no0, Edpod0no0) (Arpod0no0, Edpod0no1) (Arpod0no1, Edpod0no0) (Arpod0no1, Edpod0no1) (Arpod1no0, Edpod1no0) (Arpod1no0, Edpod1no1) (Arpod1no1, Edpod1no0) (Arpod1no1, Edpod1no1) (Arpod2no0, Edpod2no0) (Arpod2no0, Edpod2no1) (Arpod2no1, Edpod2no0) (Arpod2no1, Edpod2no1) (Arpod3no0, Edpod3no0) (Arpod3no0, Edpod3no1) (Arpod3no1, Edpod3no0) (Arpod3no1, Edpod3no1) (Edpod0no0, h0_0) (Edpod0no0, h0_1) (Edpod0no1, h0_2) (Edpod0no1, h0_3) (Edpod1no0, h1_0) (Edpod1no0, h1_1) (Edpod1no1, h1_2) (Edpod1no1, h1_3) (Edpod2no0, h2_0) (Edpod2no0, h2_1) (Edpod2no1, h2_2) (Edpod2no1, h2_3) (Edpod3no0, h3_0) (Edpod3no0, h3_1) (Edpod3no1, h3_2) (Edpod3no1, h3_3) (core0, Arpod0no0) (core0, Arpod1no0) (core0, Arpod2no0) (core0, Arpod3no0) (core1, Arpod0no0) (core1, Arpod1no0) (core1, Arpod2no0) (core1, Arpod3no0) (core2, Arpod0no1) (core2, Arpod1no1) (core2, Arpod2no1) (core2, Arpod3no1) (core3, Arpod0no1) (core3, Arpod1no1) (core3, Arpod2no1) (core3, Arpod3no1)
*** Configuring hosts
h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
*** Starting controller

*** Starting 20 switches
Arpod0no0 Arpod0no1 Arpod1no0 Arpod1no1 Arpod2no0 Arpod2no1 Arpod3no0 Arpod3no1 Edpod0no0 Edpod0no1 Edpod1no0 Edpod1no1 Edpod2no0 Edpod2no1 Edpod3no0 Edpod3no1 core0 core1 core2 core3 ...
*** Starting CLI:
mininet>

nakano9331@nakano9331-VirtualBox:~$ sudo python3 FatTree.py
[sudo] nakano9331 的密码:
*** Creating network
*** Adding hosts:
h0_0 h1_0
*** Adding switches:
Arpod0no0 Arpod1no0 Edpod0no0 Edpod1no0 core0
*** Adding links:
(Arpod0no0, Edpod0no0) (Arpod1no0, Edpod1no0) (Edpod0no0, h0_0) (Edpod1no0, h1_0) (core0, Arpod0no0) (core0, Arpod1no0)
*** Configuring hosts
h0_0 h1_0
*** Starting controller

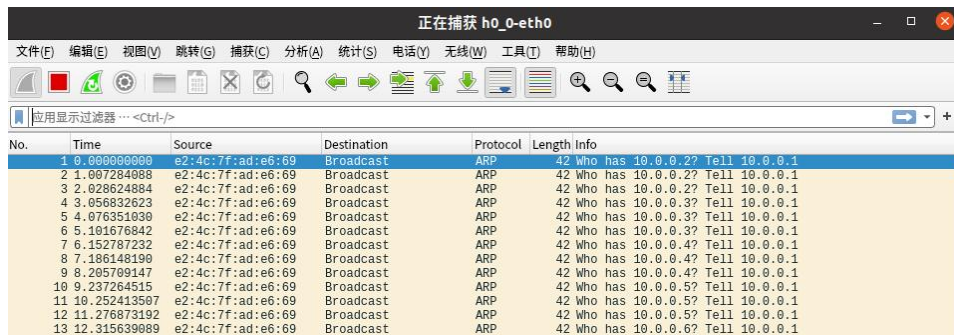
*** Starting 5 switches
Arpod0no0 Arpod1no0 Edpod0no0 Edpod1no0 core0 ...
*** Starting CLI:
mininet>
```

(2) 使用 pingall 查看各主机之间的连通情况，发现主机之间无法互通，如下图所示。

```
mininet> pingall
*** Ping: testing ping reachability
h0_0 -> X X X X X X X X X X X X X X X
h0_1 -> X X X X X X X X X X X X X X X
h0_2 -> X X X X X X X X X X X X X X X
h0_3 -> X X X X X X X X X X X X X X X
```

(3) 为进一步诊断问题，在 Mininet CLI 中执行 xterm h0_0 命令，打开终端并使用 Wireshark 进行抓包。抓包截图如下图所示，结果显示主机一直发送 ARP 查询包以获取目标

主机的 MAC 地址，但无法获得响应。



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
2	1.007284888	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
3	2.020624884	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
4	3.056832623	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
5	4.076351030	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
6	5.101676842	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
7	6.152787232	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1
8	7.186148190	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1
9	8.205709147	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1
10	9.237264515	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.5? Tell 10.0.0.1
11	10.252413507	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.5? Tell 10.0.0.1
12	11.276873192	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.5? Tell 10.0.0.1
13	12.315639089	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.6? Tell 10.0.0.1

由于 FatTree 结构中存在环路,导致广播包在网络中不断循环转发,进而引发广播风暴,网络中的流量被广播包占据,其他正常的流量无法继续转发。

广播风暴是由于以太网交换机的工作方式造成的。当交换机从某个端口收到广播包时,它会将该包复制并转发到所有其他端口,即除了接收该包的端口。这种行为会导致广播包不断循环,形成网络环路,最终导致整个网络的带宽被广播包占用,导致网络拥塞和连接失败。

(4) 通过查询相关资料,为各个端口开启生成树协议即可解决广播风暴的问题。STP 通过阻塞端口来消除网络中的环路,确保数据包不会在网络中无限循环。为了在 Open vSwitch 上启用 STP,可以使用以下命令: `sudo ovs-vsctl set bridge s1 stp_enable=true`。其中, s1 是交换机的设备名。

但是网络中有多个交换机,则必须对每个交换机单独执行上述命令,才能启用 STP 协议。这对于交换机数量较多的场景来说,逐个配置较为复杂。因此编写一个脚本来自动为每个交换机启用 STP 协议。

以下 stp.py 脚本使用 `os.system()` 函数通过系统调用执行命令来实现批量开启 STP。

```
nakano9331@nakano9331-VirtualBox:~$ sudo nano stp.py
[sudo] nakano9331 的密码:
nakano9331@nakano9331-VirtualBox:~$ sudo python3 stp.py
```

`sudo ovs-vsctl del-fail-mode xx` 命令用于删除交换机的失败模式设置,这样交换机才能正常学习 MAC 地址表。通过运行 stp.py 脚本,所有交换机都能批量开启 STP 协议。

(5) 在所有交换机启用 STP 后,重新使用 pingall 命令进行网络连通性测试。此时,主机之间应该可以正常 ping 通。

```
mininet> pingall
*** Ping: testing ping reachability
h0_0 -> X X X h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h0_1 -> h0_0 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h0_2 -> h0_0 h0_1 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h0_3 -> h0_0 h0_1 h0_2 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h1_0 -> h0_0 h0_1 h0_2 h0_3 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h1_1 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h1_2 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h1_3 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h2_0 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h2_1 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h2_2 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_3 h3_0 h3_1 h3_2 h3_3
h2_3 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h3_0 h3_1 h3_2 h3_3
h3_0 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_1 h3_2 h3_3
h3_1 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_2 h3_3
h3_2 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_3
h3_3 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2
*** Results: 1% dropped (237/240 received)
```

在启用 STP 后,主机间出现短暂的不可达情况。这是因为交换机需要时间通过 STP 学习网络拓扑结构。STP 会等待网络稳定并生成一个无环的拓扑,这个过程可能需要一些时间。通过这些操作,可以有效地在大规模网络中启用 STP 协议,避免因环路导致的广播风暴和网络拥堵。


```

** Ping: testing ping reachability
h0_0 -> h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h0_1 -> h0_0 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h0_2 -> h0_0 h0_1 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h0_3 -> h0_0 h0_1 h0_2 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h1_0 -> h0_0 h0_1 h0_2 h0_3 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h1_1 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h1_2 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h1_3 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h2_0 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h2_1 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_2 h2_3 h3_0 h3_1 h3_2 h3_3
h2_2 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_3 h3_0 h3_1 h3_2 h3_3
h2_3 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h3_0 h3_1 h3_2 h3_3
h3_0 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_1 h3_2 h3_3
h3_1 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_2 h3_3
h3_2 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_3
h3_3 -> h0_0 h0_1 h0_2 h0_3 h1_0 h1_1 h1_2 h1_3 h2_0 h2_1 h2_2 h2_3 h3_0 h3_1 h3_2
*** Results: 0% dropped (240/240 received)

```

在生成树协议进行传播的过程中开启 wireshark 进行抓包观察, 结果如下图所示。交换机会定期发送 STP 包以确定生成树。

当网络连接稳定后，主机向目标主机发送报文时，通常会先发送一个 ARP 查询包来获取目标主机的 MAC 地址，随后收到目标主机的 ARP 回复包。此时，主机会根据获得的 MAC 地址发送 ICMP 包进行正常的网络通信。

No.	Time	Source	Destination	Protocol	Length	Info
25	0.007544101	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.11? Tell 10.0.0.1
26	0.688365340	96:56:91:fc:29:3b	Spanning-tree(for-...	STP	42	Conf. Root = 32768/0/12:da:f8:ff:46:46 Cost =
27	1.024522648	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.11? Tell 10.0.0.1
28	1.025065124	3a:a8:04:ed:76:e7	e2:4c:7f:ad:e6:69	ARP	42	10.0.0.11 is at 3a:a8:04:ed:76:e7
29	1.025069598	10.0.0.1	10.0.0.11	ICMP	98	Echo (ping) request id=0x0ddc, seq=1/256, ttl=
30	1.025333373	10.0.0.11	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0ddc, seq=1/256, ttl=
31	1.026425595	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.12? Tell 10.0.0.1
32	1.026763695	86:1e:87:a8:18:14	e2:4c:7f:ad:e6:69	ARP	42	10.0.0.12 is at 86:1e:87:a8:18:14
33	1.026767129	10.0.0.1	10.0.0.12	ICMP	98	Echo (ping) request id=0x0ddd, seq=1/256, ttl=
34	1.026918279	10.0.0.12	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0ddd, seq=1/256, ttl=
35	1.027844669	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.13? Tell 10.0.0.1
36	1.028161259	22:c6:9e:68:96:03	e2:4c:7f:ad:e6:69	ARP	42	10.0.0.13 is at 22:c6:9e:68:96:03
37	1.028164373	10.0.0.1	10.0.0.13	ICMP	98	Echo (ping) request id=0x0dde, seq=1/256, ttl=
38	1.028305119	10.0.0.13	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0dde, seq=1/256, ttl=
39	1.029124880	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.14? Tell 10.0.0.1
40	1.029460611	6a:34:28:aa:35:c0	e2:4c:7f:ad:e6:69	ARP	42	10.0.0.14 is at 6a:34:28:aa:35:c0
41	1.029463758	10.0.0.1	10.0.0.14	ICMP	98	Echo (ping) request id=0x0ddf, seq=1/256, ttl=
42	1.029608920	10.0.0.14	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0ddf, seq=1/256, ttl=
43	1.030426740	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.15? Tell 10.0.0.1
44	1.030760945	82:16:ad:05:d0:27	e2:4c:7f:ad:e6:69	ARP	42	10.0.0.15 is at 82:16:ad:05:d0:27
45	1.030763962	10.0.0.1	10.0.0.15	ICMP	98	Echo (ping) request id=0x0de0, seq=1/256, ttl=
46	1.031064063	10.0.0.15	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0de0, seq=1/256, ttl=
47	1.031922524	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.16? Tell 10.0.0.1
48	1.032472693	12:ca:87:59:c9:35	e2:4c:7f:ad:e6:69	ARP	42	10.0.0.16 is at 12:ca:87:59:c9:35
49	1.032475733	10.0.0.1	10.0.0.16	ICMP	98	Echo (ping) request id=0x0de1, seq=1/256, ttl=
50	1.032665548	10.0.0.16	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0de1, seq=1/256, ttl=
51	1.033956301	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) request id=0x0de2, seq=1/256, ttl=
52	1.033970138	e2:4c:7f:ad:e6:69	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
53	1.034023240	5a:ba:9a:1b:7c:70	e2:4c:7f:ad:e6:69	ARP	42	10.0.0.2 is at 5a:ba:9a:1b:7c:70
54	1.034026847	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) reply id=0x0de2, seq=1/256, ttl=

(6) 数据包的路径分析，以分析从主机 h0_0 到主机 h3_3 的数据包路径为例。首先，使用 `ifconfig` 命令查询主机 h0_0 和 h3_2 的 MAC 地址。由下图可得，h0_0 的 MAC 地址为 00:00:00:00:00:01，h3_2 的 MAC 地址为 00:00:00:00:00:0f。

```
mininet> h0_0 ifconfig
h0_0-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 (以太网)
RX packets 330 bytes 24365 (24.3 KB)
RX errors 0 dropped 141 overruns 0 frame 0
TX packets 14 bytes 1156 (1.1 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (本地环回)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

```

mininet> h3_2 ifconfig
h3_2-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.0.15 netmask 255.0.0.0 broadcast 10.255.255.255
    inet6 fe80::200:ff:fe00:f prefixlen 64 scopeid 0x20<link>
    ether 00:00:00:00:00:0f txqueuelen 1000 (以太网)
    RX packets 23 bytes 2877 (2.8 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 9 bytes 786 (786.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (本地环回)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

为了分析数据包的转发路径，我们需要查看交换机的 MAC 表。这些表项会告诉我们交换机如何将数据包转发到不同的端口。可以通过以下命令 `sudo ovs-appctl fdb/show <switchname>` 查看交换机的 MAC 表。其中，<switchname>是交换机的名称。

执行命令 `h0_0 ping h3_2` 后追踪数据包，得到如下结果。

```

nakano9331@nakano9331-VirtualBox:~$ sudo ovs-appctl fdb/show Edpod0no0
port  VLAN  MAC                Age
  3      0  00:00:00:00:00:01    0
  1      0  00:00:00:00:00:0f    0

```

```

nakano9331@nakano9331-VirtualBox:~$ sudo ovs-appctl fdb/show Arpod0no1
port  VLAN  MAC                Age
  2      0  00:00:00:00:00:0f    0
  3      0  00:00:00:00:00:01    0

```

```

nakano9331@nakano9331-VirtualBox:~$ sudo ovs-appctl fdb/show core2
port  VLAN  MAC                Age
  4      0  00:00:00:00:00:01   22
  4      0  00:00:00:00:00:0f   22

```

```

nakano9331@nakano9331-VirtualBox:~$ sudo ovs-appctl fdb/show Arpod3no1
port  VLAN  MAC                Age
  4      0  aa:a2:10:74:31:92   14
  2      0  00:00:00:00:00:09   14
  2      0  0e:93:bc:f6:6e:da   12
  4      0  22:1b:9e:2e:ac:95   10
  2      0  ee:fb:cd:57:fe:3c   10
  2      0  1a:c4:e6:99:10:66    8
  2      0  96:4b:6d:cb:ec:e0    6
  2      0  00:00:00:00:00:05    6
  2      0  d6:3a:ba:25:28:f1    6
  3      0  00:00:00:00:00:0d    4
  3      0  9a:68:c4:c3:e8:cc    4
  2      0  00:00:00:00:00:0a    4
  2      0  ce:9b:dc:7c:d0:0e    4
  2      0  62:2e:59:c6:6e:d2    2
  2      0  00:00:00:00:00:01    1
  4      0  00:00:00:00:00:0f    1

```

```

nakano9331@nakano9331-VirtualBox:~$ sudo ovs-appctl fdb/show Edpod3no1
port  VLAN  MAC                Age
  2      0  b2:37:e9:da:7f:87    3
  2      0  02:12:37:86:0e:f0    3
  2      0  00:00:00:00:00:06    3
  3      0  00:00:00:00:00:0f    0
  2      0  00:00:00:00:00:01    0

```


根据交换机 MAC 表信息，可以分析 h0_0 到 h3_2 数据包的路径。

交换机 Edpod0no0 的 MAC 表显示了 MAC 地址 00:00:00:00:00:0f 对应的端口是 3。如果数据包的目标 MAC 地址是 00:00:00:00:00:0f，它将通过端口 3 转发。

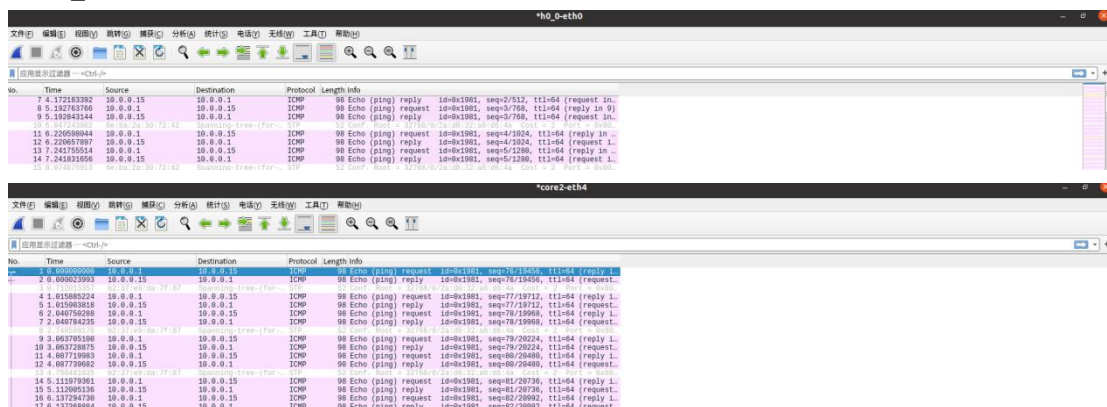
交换机 Arpod0no1 的 MAC 表显示了 MAC 地址 00:00:00:00:00:0f 对应的端口是 2。因此，数据包会从交换机 Arpod0no1 的端口 2 转发。

交换机 core2 的 MAC 表显示了 MAC 地址 00:00:00:00:00:0f 对应的端口是 4。数据包会通过交换机 core2 的端口 4 转发。

交换机 Arpod3no1 的 MAC 表显示了 MAC 地址 00:00:00:00:00:0f 对应的端口是 4。数据包会从交换机 Arpod3no1 的端口 4 转发。

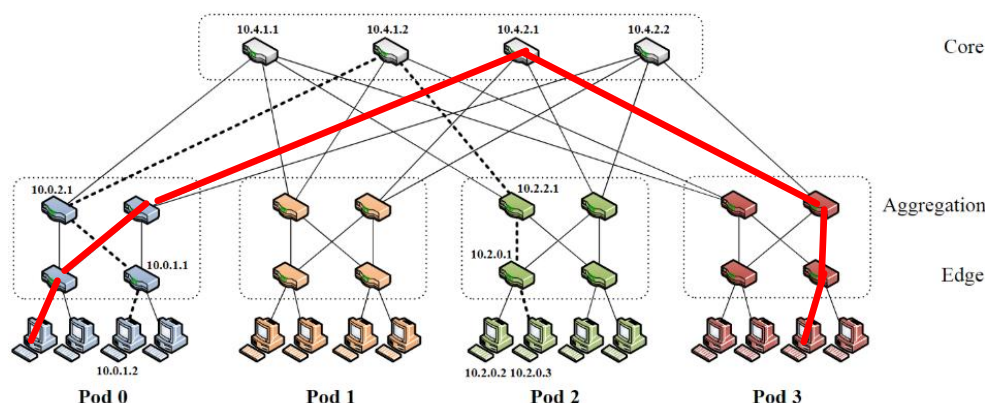
交换机 Edpod3no1 的 MAC 表显示了 MAC 地址 00:00:00:00:00:0f 对应的端口是 3。数据包会从交换机的端口 3 转发到目标主机 h3_3。

以 h0_0 终端和 core2 终端进行抓包报文为例，发现与观察 MAC 表所得结果相同。



No.	Time	Source	Destination	Protocol	Length	Info
7	1.17213392	10.0.0.1	10.0.0.1	ICMP	98	Echo (ping) request 10.0.0.1, seq=511, ttl=64 (request in)
8	1.192743760	10.0.0.1	10.0.0.1	ICMP	98	Echo (ping) request 10.0.0.1, seq=511, ttl=64 (reply in 9)
9	1.20343444	10.0.0.1	10.0.0.1	ICMP	98	Echo (ping) request 10.0.0.1, seq=511, ttl=64 (reply in 10)
10	1.21413520	10.0.0.1	10.0.0.1	ICMP	98	Echo (ping) request 10.0.0.1, seq=511, ttl=64 (reply in 11)
11	1.22483596	10.0.0.1	10.0.0.1	ICMP	98	Echo (ping) request 10.0.0.1, seq=511, ttl=64 (reply in 12)
12	1.23553672	10.0.0.1	10.0.0.1	ICMP	98	Echo (ping) request 10.0.0.1, seq=511, ttl=64 (reply in 13)
13	1.24623748	10.0.0.1	10.0.0.1	ICMP	98	Echo (ping) request 10.0.0.1, seq=511, ttl=64 (reply in 14)
14	1.25693824	10.0.0.1	10.0.0.1	ICMP	98	Echo (ping) request 10.0.0.1, seq=511, ttl=64 (reply in 15)
15	1.26763900	10.0.0.1	10.0.0.1	ICMP	98	Echo (ping) request 10.0.0.1, seq=511, ttl=64 (reply in 16)
16	1.27833976	10.0.0.1	10.0.0.1	ICMP	98	Echo (ping) request 10.0.0.1, seq=511, ttl=64 (reply in 17)
17	1.28904052	10.0.0.1	10.0.0.1	ICMP	98	Echo (ping) request 10.0.0.1, seq=511, ttl=64 (reply in 18)

因此，数据包从 h0_0 到 h3_3 的转发路径如下：



通过以上分析，可以确认数据包的完整转发路径，以及每个交换机如何根据 MAC 表转发数据包。路径为 h0_0-Edpod0no0-Arpod0no1-core2-Arpod3no1-Edpod3no1-h3_2。

四、实验总结

本次实验通过使用 Mininet 的 Python API 搭建了一个 k=4 的 fat-tree 拓扑，并测试了各主机之间的连通性。使用 pingall 命令测试所有主机之间的连通性。初次运行时，发现某些主机未能成功 ping 通，原因是网络中存在环路或者交换机配置问题。在抓包分析中，使用 Wireshark 对数据包进行了捕获。抓包结果显示主机发送了大量的 ARP 查询包，但始终未收到回复。这表明网络中存在广播风暴，导致数据包的转发无法正常进行。进一步分析发现，

FatTree 拓扑结构存在环路，导致交换机转发的广播包不断被复制，最终占用网络带宽，影响正常数据流的转发。

为了解决环路问题，启用了生成树协议。启用 STP 后，网络中的交换机能够自动消除环路并防止广播风暴。此时，再次使用 pingall 命令测试，发现主机之间的连通性恢复正常。

使用 `ovs-appctl fdb/show <switch_name>` 命令查看交换机的 MAC 表，分析数据包的转发路径。根据 MAC 表，可以看到各交换机如何根据 MAC 地址转发数据包，确保数据包能够顺利到达目标主机。

当 $k=2$ 时，中心 core 与下层的两个 pod 直连，每个 pod 中又将交换机与主机直连，不存在环路。但当执行 `sudo mn --controller=none` 时，没有环路但仍无法 ping 通，考虑是由于在 Mininet 中，没有指定控制器来管理交换机。

具体而言，Mininet 模拟的是基于 SDN 的网络环境，默认情况下，交换机依赖于控制器来接收流表规则。控制器的作用是下发转发规则给交换机，指导如何处理数据包。若无控制器，交换机不会有任何转发规则，自然无法处理流量。结果是表现为，交换机收到数据包后直接丢弃。

```
nakano9331@nakano9331-VirtualBox:~$ sudo mn --controller=none
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X
h2 -> X
*** Results: 100% dropped (0/2 received)
```

为了验证猜，尝试手动给交换机下发流表规则。例如执行下面两条指令：

```
mininet> sh ovs-ofctl add-flow s1 "in_port=1,actions=output:2"
mininet> sh ovs-ofctl add-flow s1 "in_port=2,actions=output:1"
```

使得让 s1 把来自 h1 的流量转发给 h2，反之亦然。观察执行上述命令后的显示结果。发现成功联通了两台主机，验证了猜想。

```
mininet> sh ovs-ofctl add-flow s1 "in_port=1,actions=output:2"
mininet> sh ovs-ofctl add-flow s1 "in_port=2,actions=output:1"
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

因此，在没有环路的情况下，若没有控制器，交换机就没有默认的转发规则，导致 ping 不通。解决方法是重新使用默认控制器或手动给交换机下发流表规则。

具体原因为 STP 用于防止交换机网络中的环路，通过关闭冗余链路形成一个无环的树状结构。在 STP 中，所有交换机会选举出一个根桥，后根据生成树算法来确定通往根桥的最短路径。非根桥的交换机会关闭冗余路径，导致部分路径不可用。

根桥的位置直接影响路径选择。如果根桥在 Core 层，路径可能更短；如果根桥在 Aggregation 层，路径可能被迫绕行。本实验中通过 MAC 表查看到的路径是 `h0_0 → Edpod0no0 → Arpod0no1 → core2 → Arpod3no1 → Edpod3no1 → h3_2`，即 6 跳。但如

果根桥变成了 Arpod0no0, h0_0 的流量可能先经过其他 Aggregation, 再绕到 Core 层, 增加跳数。又如果 Core 层某条链路被禁止, 流量可能通过其他 Core 交换机或多个 Aggregation, 跳数也会增加。

五、实验中遇到的问题与心得

在处理环路导致的广播风暴时, 由于网络中有多个交换机, 逐个通过编写 `stp.py` 脚本实现批量开启 STP。

在验收过程中, 当进行修改 `k=2` 时, 由于代码定义拓扑参数的方法为:

```
core_switch_num = 4 # 核心交换机数量
pod_num = 4 # pod 的数量
pod_switch_ae_num = 2 # 每个 pod 的汇聚和边缘交换机数量
pod_host_num = 4 # 每个 pod 的主机数量
```

导致修改后出现 `IndexError: list index out of range` 错误, 即代码里尝试访问列表中不存在的索引。

具体表现为文件中 `self.addLink(core_switchs[i], aggr_switchs[j][0])` 处的 `core_switchs` 或 `aggr_switchs` 列表的长度小于索引 `i` 或者 `j` 的值。即出现索引越界或列表长度不足的问题。修改方法为将所有参数全部连接到全局变量 `k` 上, 只修改 `k` 即可完成不同 `FatTree` 的生成, 具体代码在源代码处显示。

通过这次实验, 我对 Mininet 的使用和 `Fat-Tree` 拓扑的结构有了更加直观和深入的理解。实验中涉及的 ARP 查询包、广播风暴、STP 以及 MAC 表的分析, 帮助我更好地理解现代网络中的一些常见问题和协议。在面对环路和广播风暴问题时, 如何通过启用 STP 进行有效解决, 也让我认识到协议的配置和网络管理的重要性。

此外, 实验中抓包和查看 MAC 表的过程也让我对网络包的转发和交换机的工作原理有了更清晰的认识, 增强了我对网络层次结构的理解, 并且进一步提高了我分析和解决网络问题的能力。

六、源代码

FatTree.py (修改版)

```
from mininet.topo import Topo

from mininet.net import Mininet

from mininet.cli import CLI

from mininet.log import setLogLevel

# 定义全局变量 k

k = 2
```



```

class fatTree(Topo):

    def build(self):

        core_switch_num = (k // 2) ** 2

        pod_num = k

        pod_switch_ae_num = k // 2

        pod_host_num = (k // 2)

        core_switchs = []

        aggr_switchs = [[] for _ in range(pod_num)]

        edge_switchs = [[] for _ in range(pod_num)]

        hosts = [[] for _ in range(pod_num)]

        for i in range(core_switch_num):

            core_switch = self.addSwitch(f"core{i}")

            core_switchs.append(core_switch)

        for i in range(pod_num):

            for j in range(pod_switch_ae_num):

                aggr_switch = self.addSwitch(f"Arpod{i}no{j}")

                aggr_switchs[i].append(aggr_switch)

            for i in range(pod_num):

                for j in range(pod_switch_ae_num):

                    edge_switch = self.addSwitch(f"Edpod{i}no{j}")

                    edge_switchs[i].append(edge_switch)

            for i in range(pod_num):

                for j in range(pod_host_num):

                    host = self.addHost(f"h{i}_{j}") # 让 Mininet 自动处理 MAC

```

地址

```
        hosts[i].append(host)

    for core_index in range(core_switch_num):

        sub_index = core_index % (k // 2)

        for pod_index in range(pod_num):

            self.addLink(core_switchs[core_index],
aggr_switchs[pod_index][sub_index])

        for pod_index in range(pod_num):

            for aggr_index in range(pod_switch_ae_num):

                for edge_index in range(pod_switch_ae_num):

                    self.addLink(aggr_switchs[pod_index][aggr_index],
edge_switchs[pod_index][edge_index])

            for pod_index in range(pod_num):

                for edge_index in range(pod_switch_ae_num):

                    for host_index in range(pod_host_num):

                        self.addLink(edge_switchs[pod_index][edge_index],
hosts[pod_index][host_index])

def run():

    topo = fatTree()

    net = Mininet(topo, controller=None, autoSetMacs=True) # 设置
autoSetMacs 为 True, 自动生成 MAC 地址

    net.start()

    CLI(net)

    net.stop()

if __name__ == '__main__':

    setLogLevel('info')
```

```
run()
```

stp.py

```
import os
```

```
def enable_stp_for_switches():
```

```
    core_switch_num = 4
```

```
    pod_num = 4
```

```
    pod_switch_ae_num = 2
```

```
    # 启用 Core 交换机的 STP 协议
```

```
    for i in range(core_switch_num):
```

```
        switch_name = f"core{i}"
```

```
        os.system(f"sudo ovs-vsctl set Bridge {switch_name}  
stp_enable=true")
```

```
        os.system(f"sudo ovs-vsctl del-fail-mode {switch_name}")
```

```
    # 启用 Pod 交换机的 STP 协议 (Aggregation 和 Edge 交换机)
```

```
    for i in range(pod_num):
```

```
        for j in range(pod_switch_ae_num):
```

```
            aggr_switch_name = f"Arpod{i}no{j}"
```

```
            edge_switch_name = f"Edpod{i}no{j}"
```

```
            os.system(f"sudo ovs-vsctl set Bridge {aggr_switch_name}  
stp_enable=true")
```

```
            os.system(f"sudo ovs-vsctl del-fail-mode  
{aggr_switch_name}")
```

```
            os.system(f"sudo ovs-vsctl set Bridge {edge_switch_name}  
stp_enable=true")
```

```
            os.system(f"sudo ovs-vsctl del-fail-mode  
{edge_switch_name}")
```

```
if __name__ == '__main__':
```

```
    enable_stp_for_switches()
```