## 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） ；

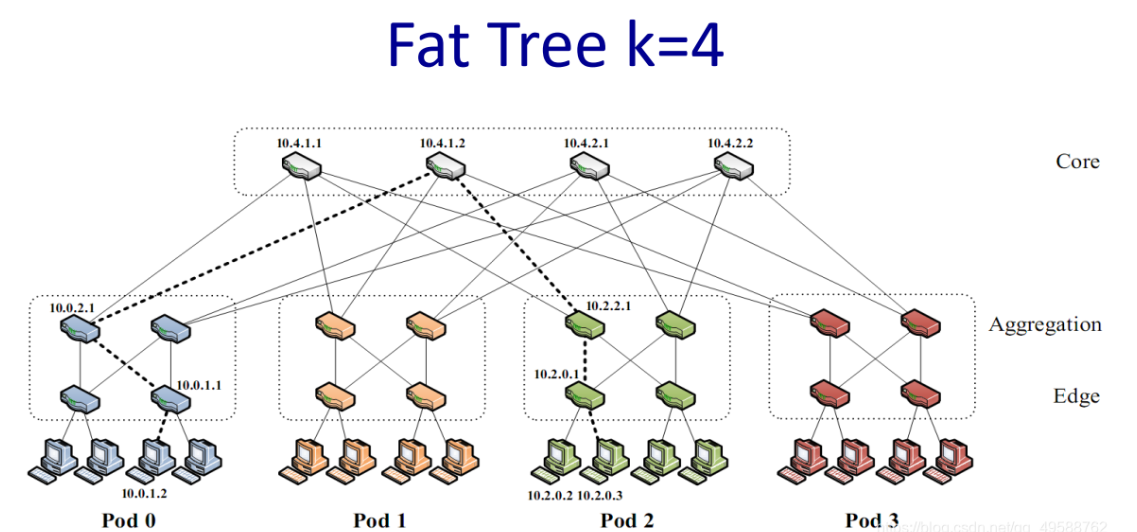
（5）完成实验报告并提交到思源学堂，要求不能使用控制器。

## 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 k³/4 台服务器。任意两个 pod 之间存在 k 条路径。



（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。

## 实验过程

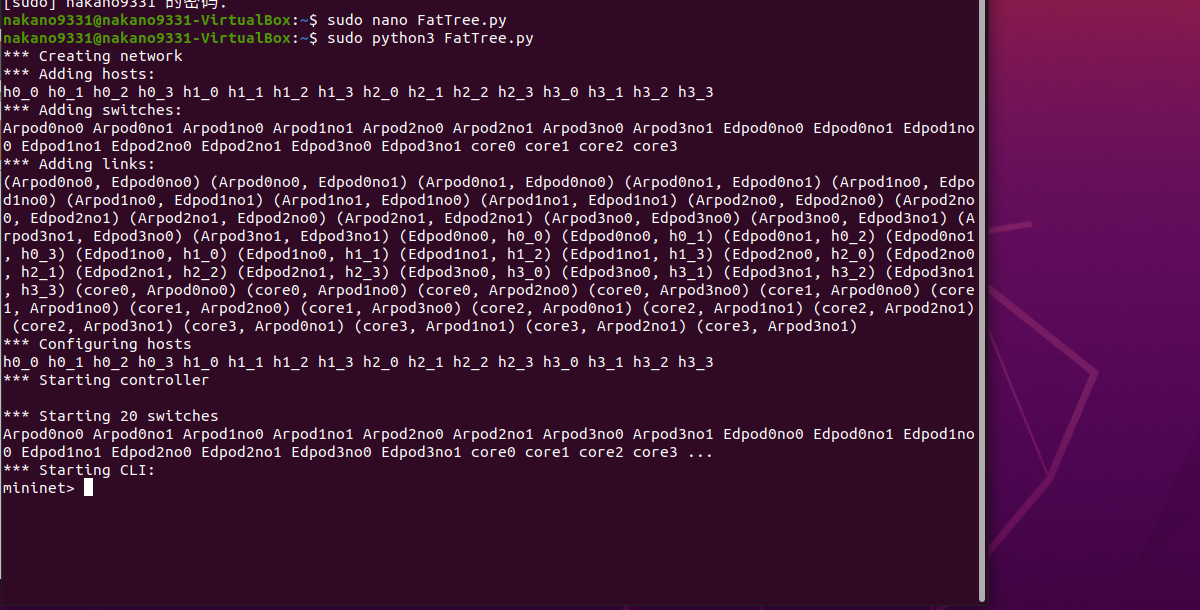
1. 编写FatTree.py使用Mininet的Python API搭建k=4的fat tree拓扑。如下图所示，

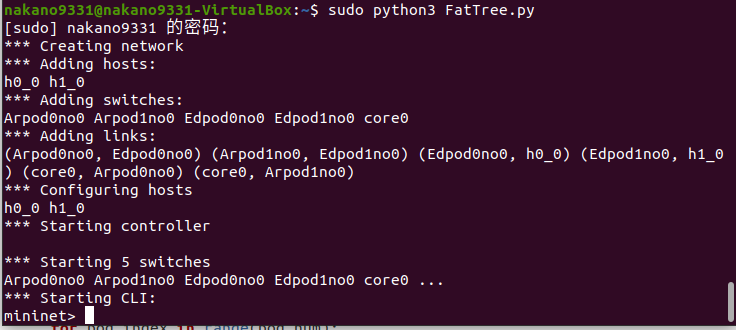
网络拓扑被成功建立。

其中核心交换机设置core\_switch\_num = 4，根据公式(k/2)2=4(k/2)^2 = 4，解得k=4。

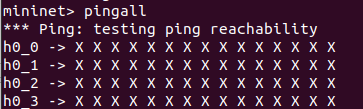
而pod = 4，每个pod的交换机数量pod\_switch\_ae\_num = 2则每个pod的汇聚和边缘交换机数量符合k/2=2k/2 = 2。每个pod的主机数量为pod\_host\_num = 4有每个 pod 的主机数量。

例如，要让k=2，则将参数调换为2，执行结果如下所示。

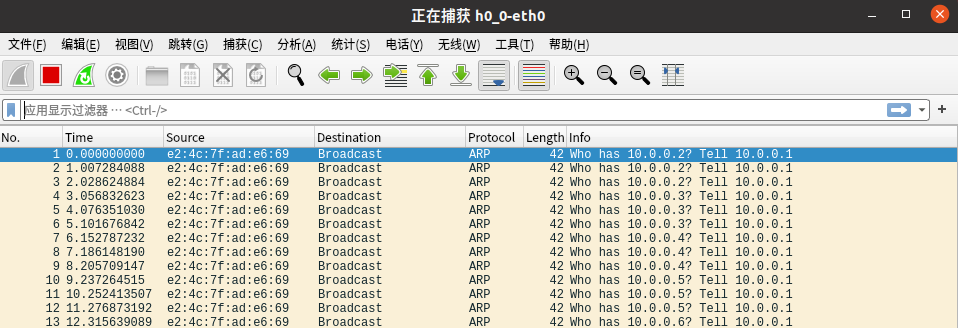




1. 使用 pingall 查看各主机之间的连通情况，发现主机之间无法互通，如下图所示。



1. 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。



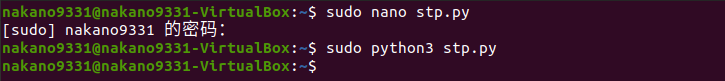
由于FatTree结构中存在环路，导致广播包在网络中不断循环转发，进而引发广播风暴，网络中的流量被广播包占据，其他正常的数据流量无法继续转发。

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。

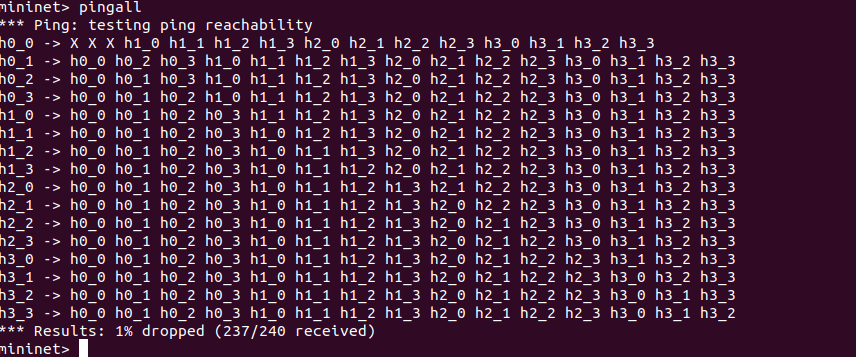
但是网络中有多个交换机，则必须对每个交换机单独执行上述命令，才能启用STP协议。这对于交换机数量较多的场景来说，逐个配置较为复杂。因此编写一个脚本来自动为每个交换机启用STP协议。

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。

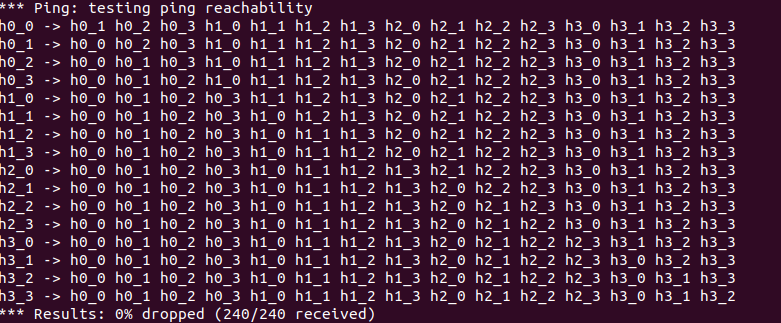


sudo ovs-vsctl del-fail-mode xx命令用于删除交换机的失败模式设置，这样交换机才能正常学习MAC地址表。通过运行stp.py脚本，所有交换机都能批量开启STP协议。

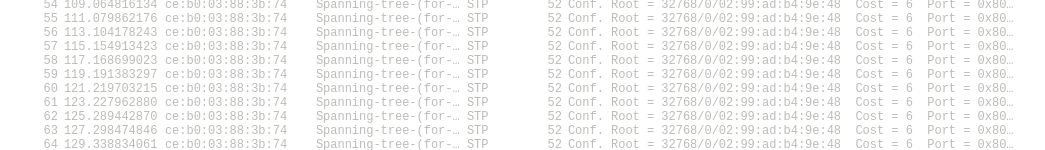
（5）在所有交换机启用STP后，重新使用pingall命令进行网络连通性测试。此时，主机之间应该可以正常ping通。



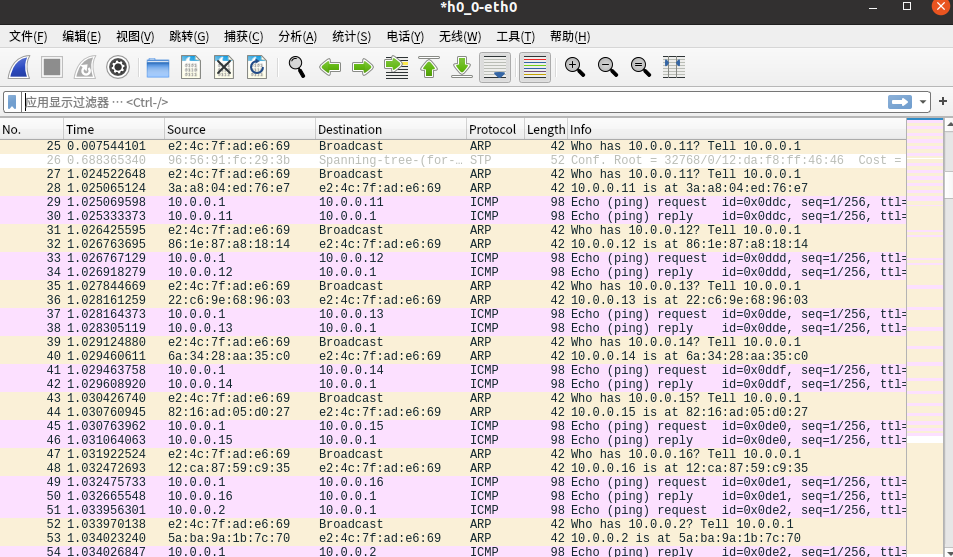
在启用STP后，主机间出现短暂的不可达情况。这是因为交换机需要时间通过STP学习网络拓扑结构。STP会等待网络稳定并生成一个无环的拓扑，这个过程可能需要一些时间。通过这些操作，可以有效地在大规模网络中启用 STP 协议，避免因环路导致的广播风暴和网络拥堵。



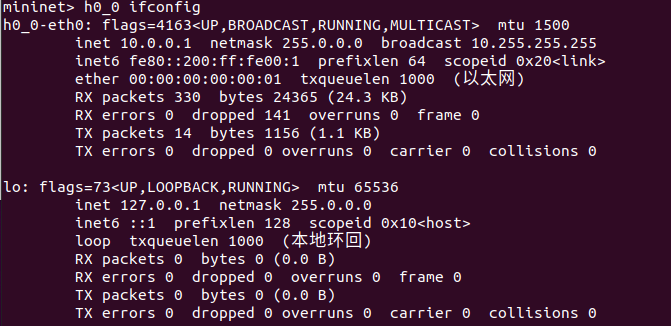
在生成树协议进行传播的过程中开启wireshark进行抓包观察，结果如下图所示。交换机会定期发送STP包以确定生成树。

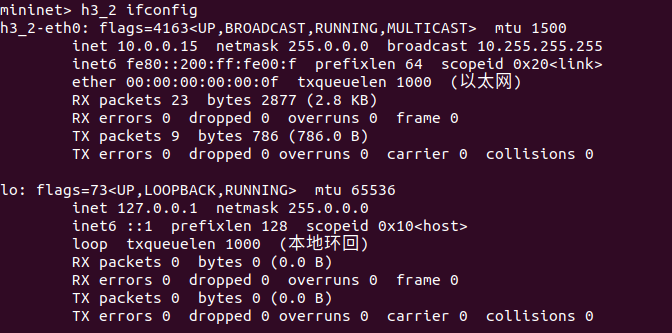


当网络连接稳定后，主机向目标主机发送报文时，通常会先发送一个ARP查询包来获取目标主机的MAC地址，随后收到目标主机的ARP回复包。此时，主机会根据获得的MAC地址发送ICMP包进行正常的网络通信。



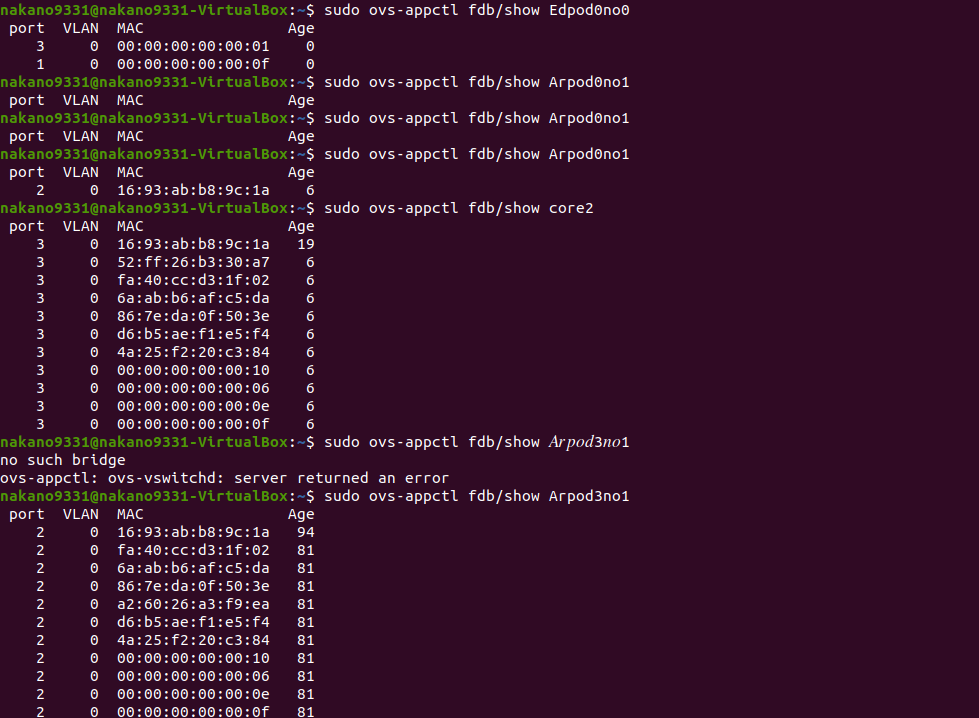
（6）数据包的路径分析，以分析从主机h0\_0到主机h3\_3的数据包路径为例。首先，使用 ifconfig 命令查询主机h0\_0和h3\_2的 MAC 地址。由下图可得，h0\_0的MAC地址为00:00:00:00:00:01，h3\_2的MAC地址为00:00:00:00:00:0f。

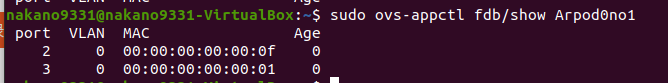


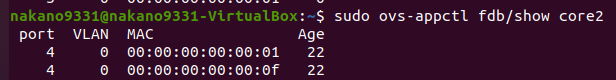


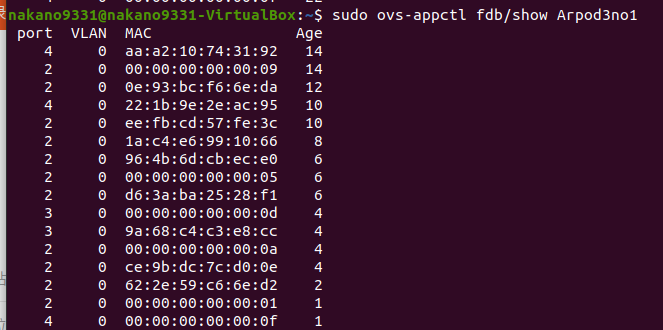
为了分析数据包的转发路径，我们需要查看交换机的MAC表。这些表项会告诉我们交换机如何将数据包转发到不同的端口。可以通过以下命令sudo ovs-appctl fdb/show <switchname>查看交换机的 MAC 表。其中，<switchname>是交换机的名称。

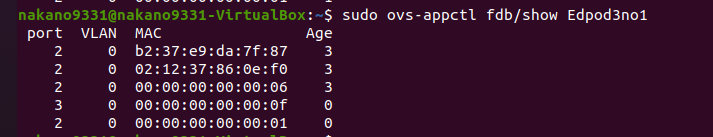
执行命令h0\_0 ping h3\_2后追踪数据包，得到如下结果。











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。

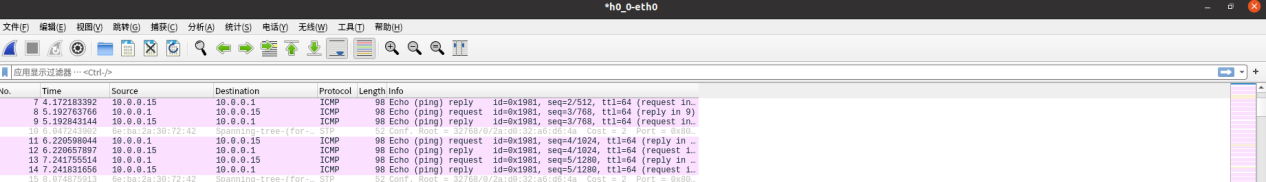
交换机Arpod0no1的MAC表显示了MAC地址 00:00:00:00:00:0f对应的端口是2。因此，数据包会从交换机Arpod0no1的端口2转发。

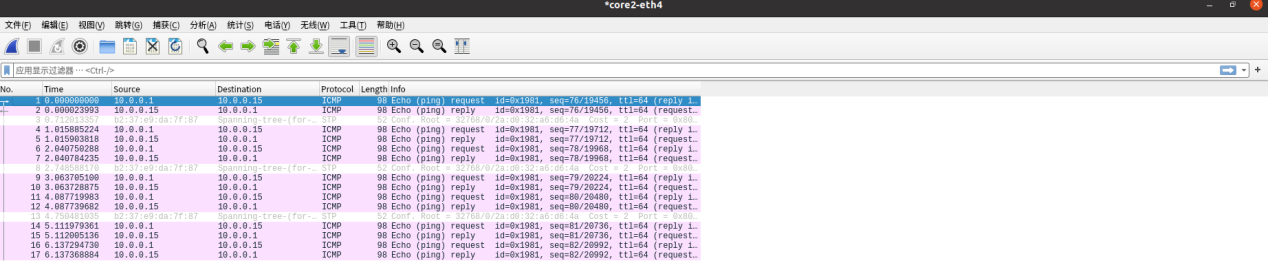
交换机core2的MAC表显示了MAC地址00:00:00:00:00:0f对应的端口是4。数据包会通过交换机core2的端口4转发。

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。

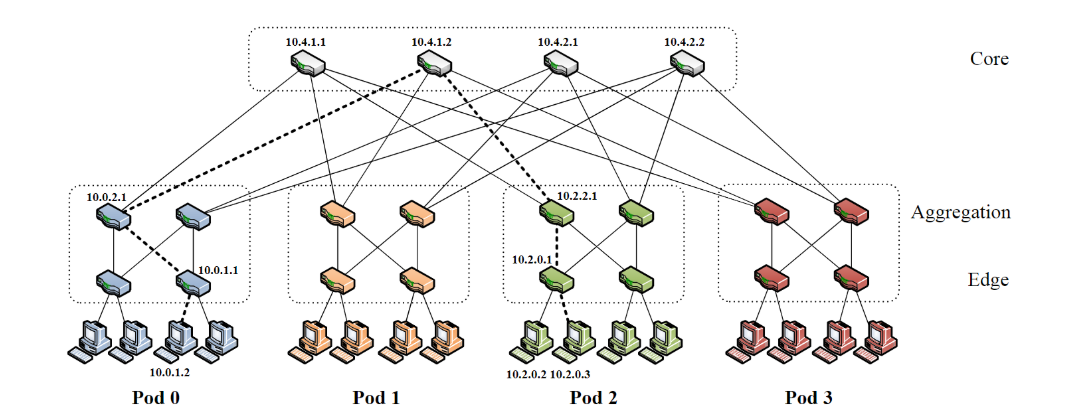
交换机 Edpod3no1的MAC表显示了MAC地址00:00:00:00:00:0f对应的端口是3。数据包会从交换机的端口3转发到目标主机h3\_3。

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。





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。

## 实验总结

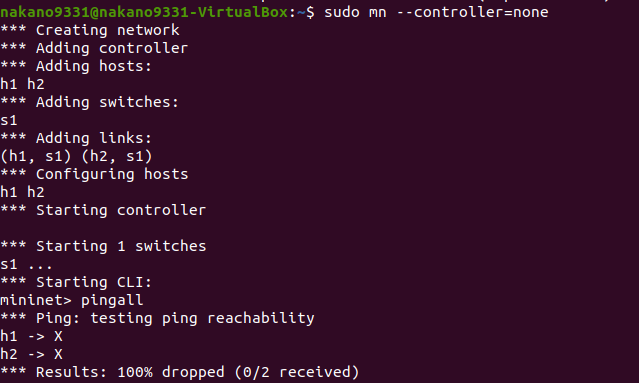
本次实验通过使用Mininet的Python API搭建了一个k=4的fat-tree拓扑，并测试了各主机之间的连通性。使用pingall命令测试所有主机之间的连通性。初次运行时，发现某些主机未能成功ping通，原因是网络中存在环路或者交换机配置问题。在抓包分析中，使用Wireshark对数据包进行了捕获。抓包结果显示主机发送了大量的ARP查询包，但始终未收到回复。这表明网络中存在广播风暴，导致数据包的转发无法正常进行。进一步分析发现，FatTree拓扑结构存在环路，导致交换机转发的广播包不断被复制，最终占用网络带宽，影响正常数据流的转发。

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，没有环路缺仍无法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。

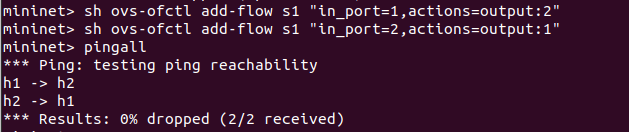


为了验证猜，尝试手动给交换机下发流表规则。例如执行下面两条指令：

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。



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。

## 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，帮助我更好地理解了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。

## 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()