# Software Define Network Lab1

**实验准备：**

–Install Mininet using Virtual Box

–Try Mininet CLI to create and interact with a network

–Try ovs-vsctl to interact with Open vSwitch (OVS)

–Use wireshark to capture packets in Mininet

**实验要求：**

–Construct a fattree (k=4) topology using Mininet Python API

–Make sure all hosts are reachable

–If not reachable, try to solve it

–If reachable, show the path between two hosts

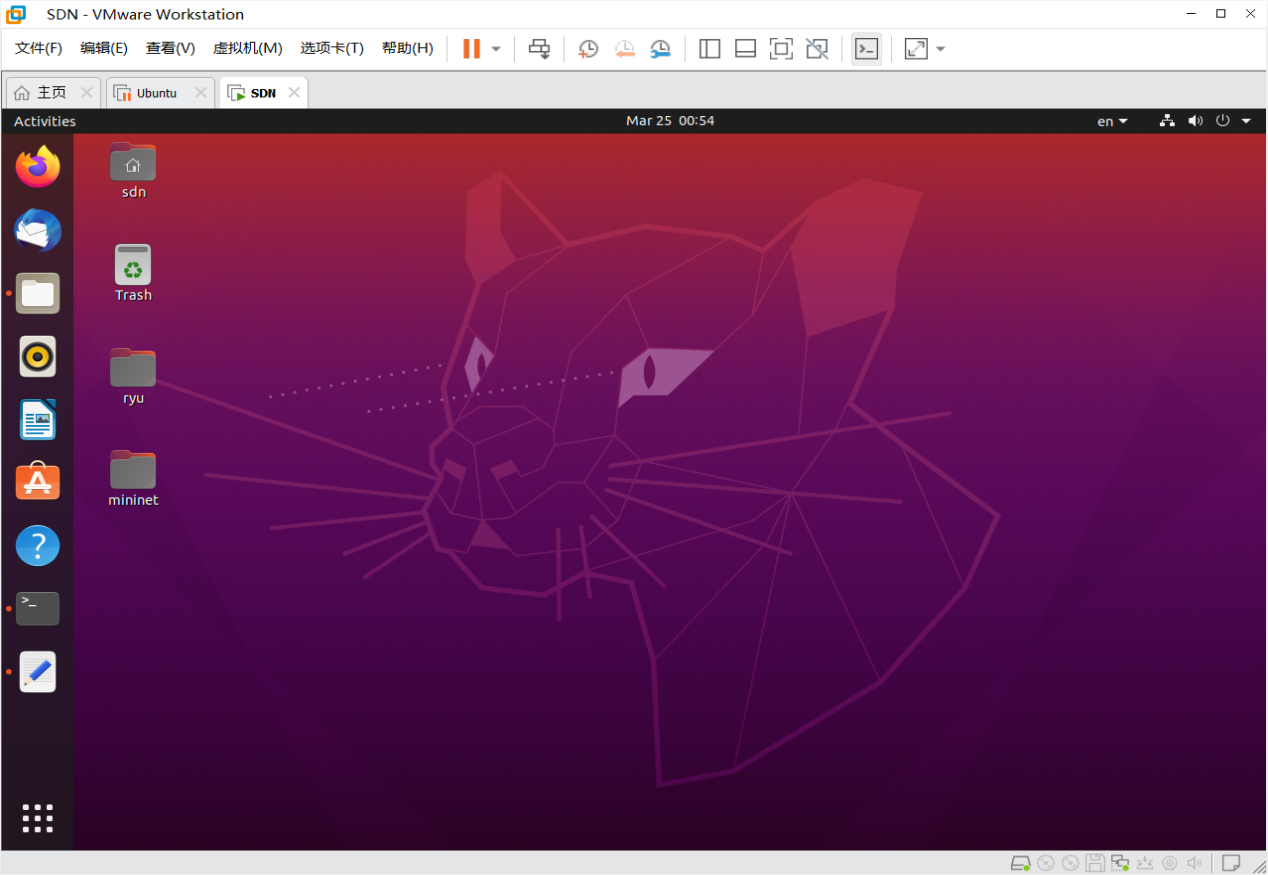
–Don’t use controller! (use ‘--controller=none’ option)

**实验环境：**

Windows 10, VMware Workstation Pro, Ubuntu

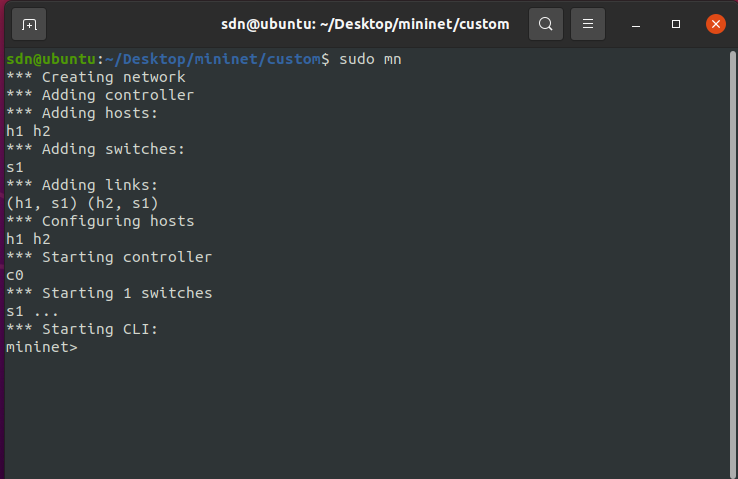
**实验过程：**

1. 安装VMware镜像，进入Ubuntu系统

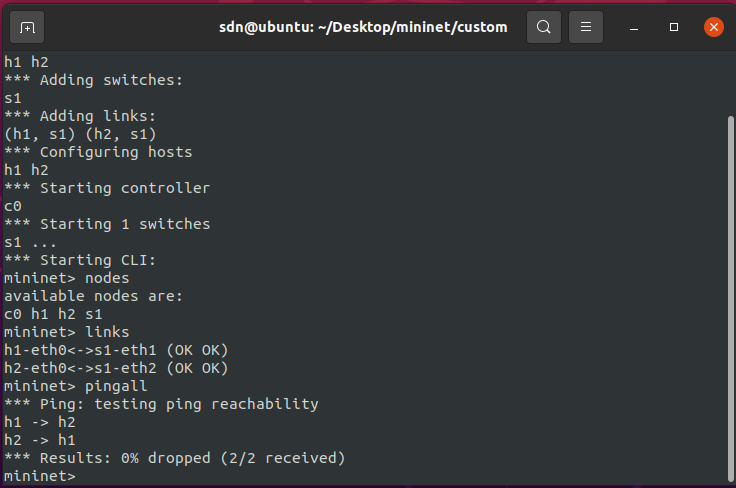


2. 创建默认Mininet拓扑，测试命令行

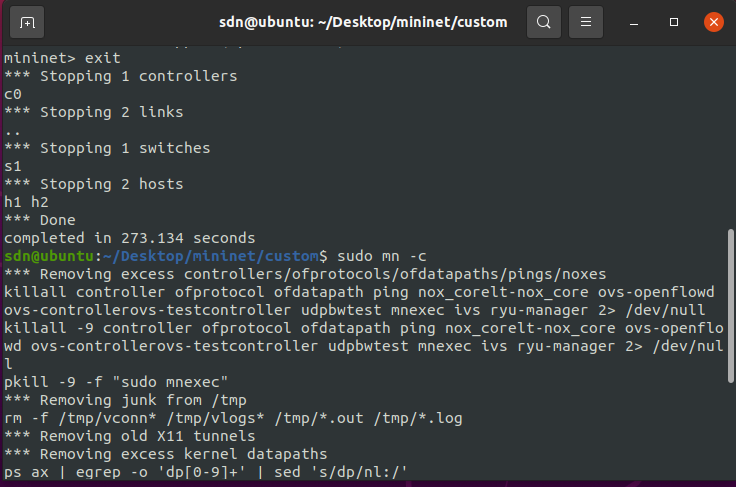
在Ubuntu系统中，打开命令行终端，输入sudo mn，创建默认拓扑。



输入Mininet命令行的各种常用指令，进行测试。输入nodes查看网络节点，输入links查看网络连接情况，输入pingall测试各节点之间的可达性。



输入exit退出Mininet命令行，输入sudo mn -c删除默认拓扑。



3. 使用Mininet的Python API，创建k=4的FatTree拓扑

首先我们要弄清楚FatTree的拓扑结构。Fat-Tree是以交换机为中心的拓扑。支持在横向拓展的同时拓展路径数目；且所有交换机均为相同端口数量的普通设备，降低了网络建设成本。

Fat-Tree结构共分为三层：核心层、汇聚层、接入层。一个k元的Fat-Tree可以归纳为5个特征：

1. 每台交换机都有k个端口；

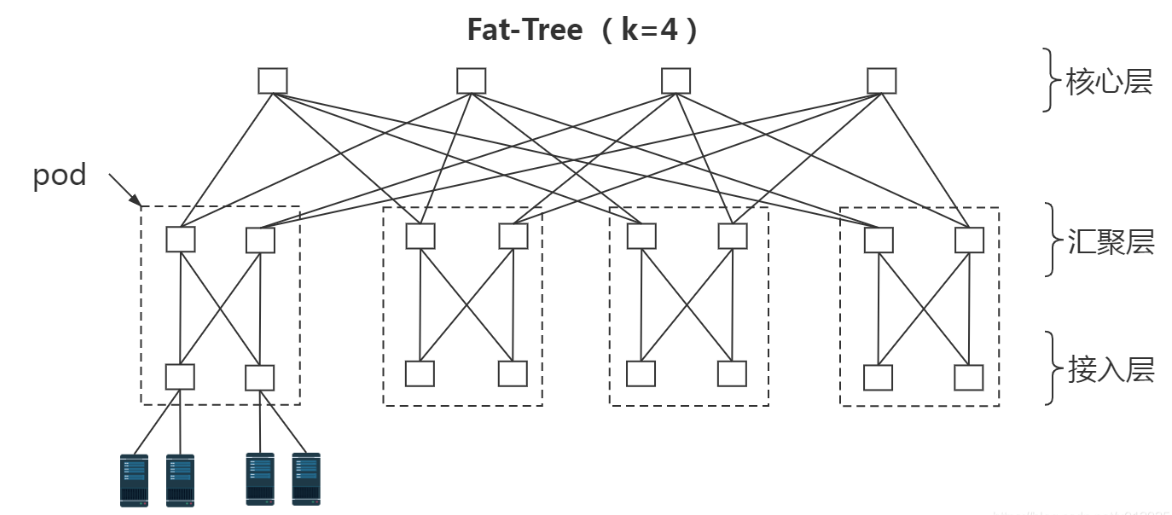
2. 核心层为顶层，一共有(k/2)^2个交换机；

3. 一共有k个pod，每个pod有k台交换机组成。其中汇聚层和接入层各占k/2台交换机；

4. 接入层每个交换机可以容纳k/2台服务器，因此，k元Fat-Tree一共有k个pod，每个pod容纳k\*k/4个服务器，所有pod共能容纳k\*k\*k/4台服务器；

5. 任意两个pod之间存在k条路径。

k = 4的FatTree拓扑结构示意图如下：



模仿实验指导书中的参考样例，我们构建k = 4的FatTree拓扑，代码如下：

from mininet.topo import Topo

from mininet.net import Mininet

from mininet.cli import CLI

from mininet.log import setLogLevel

class FatTree(Topo):

    def build(self):

        "Create custom topo."

        # Add hosts and switches

        # Add hosts in pod1

        Host11 = self.addHost('h11')

        Host12 = self.addHost('h12')

        Host13 = self.addHost('h13')

        Host14 = self.addHost('h14')

        # Add hosts in pod2

        Host21 = self.addHost('h21')

        Host22 = self.addHost('h22')

        Host23 = self.addHost('h23')

        Host24 = self.addHost('h24')

        # Add hosts in pod3

        Host31 = self.addHost('h31')

        Host32 = self.addHost('h32')

        Host33 = self.addHost('h33')

        Host34 = self.addHost('h34')

        # Add hosts in pod4

        Host41 = self.addHost('h41')

        Host42 = self.addHost('h42')

        Host43 = self.addHost('h43')

        Host44 = self.addHost('h44')

        # Add Access Layer Switchs

        AcsSwitch12 = self.addSwitch('AcS12')

        AcsSwitch21 = self.addSwitch('AcS21')

        AcsSwitch22 = self.addSwitch('AcS22')

        AcsSwitch31 = self.addSwitch('AcS31')

        AcsSwitch11 = self.addSwitch('AcS11')

        AcsSwitch32 = self.addSwitch('AcS32')

        AcsSwitch41 = self.addSwitch('AcS41')

        AcsSwitch42 = self.addSwitch('AcS42')

        # Add Distribution Layer Switchs

        DstSwitch11 = self.addSwitch('DstS11')

        DstSwitch12 = self.addSwitch('DstS12')

        DstSwitch21 = self.addSwitch('DstS21')

        DstSwitch22 = self.addSwitch('DstS22')

        DstSwitch31 = self.addSwitch('DstS31')

        DstSwitch32 = self.addSwitch('DstS32')

        DstSwitch41 = self.addSwitch('DstS41')

        DstSwitch42 = self.addSwitch('DstS42')

        # Add Core Layer Switchs

        CoreSwitch1 = self.addSwitch('CoreS1')

        CoreSwitch2 = self.addSwitch('CoreS2')

        CoreSwitch3 = self.addSwitch('CoreS3')

        CoreSwitch4 = self.addSwitch('CoreS4')

        # Add links

        # Add links between Core Layer and Distribution Layer

        self.addLink(CoreSwitch1, DstSwitch11)

        self.addLink(CoreSwitch1, DstSwitch21)

        self.addLink(CoreSwitch1, DstSwitch31)

        self.addLink(CoreSwitch1, DstSwitch41)

        self.addLink(CoreSwitch2, DstSwitch11)

        self.addLink(CoreSwitch2, DstSwitch21)

        self.addLink(CoreSwitch2, DstSwitch31)

        self.addLink(CoreSwitch2, DstSwitch41)

        self.addLink(CoreSwitch3, DstSwitch12)

        self.addLink(CoreSwitch3, DstSwitch22)

        self.addLink(CoreSwitch3, DstSwitch32)

        self.addLink(CoreSwitch3, DstSwitch42)

        self.addLink(CoreSwitch4, DstSwitch12)

        self.addLink(CoreSwitch4, DstSwitch22)

        self.addLink(CoreSwitch4, DstSwitch32)

        self.addLink(CoreSwitch4, DstSwitch42)

        # Add links between Distribution Layer and Access Layer

        self.addLink(DstSwitch11, AcsSwitch11)

        self.addLink(DstSwitch11, AcsSwitch12)

        self.addLink(DstSwitch12, AcsSwitch11)

        self.addLink(DstSwitch12, AcsSwitch12)

        self.addLink(DstSwitch21, AcsSwitch21)

        self.addLink(DstSwitch21, AcsSwitch22)

        self.addLink(DstSwitch22, AcsSwitch21)

        self.addLink(DstSwitch22, AcsSwitch22)

        self.addLink(DstSwitch31, AcsSwitch31)

        self.addLink(DstSwitch31, AcsSwitch32)

        self.addLink(DstSwitch32, AcsSwitch31)

        self.addLink(DstSwitch32, AcsSwitch32)

        self.addLink(DstSwitch41, AcsSwitch41)

        self.addLink(DstSwitch41, AcsSwitch42)

        self.addLink(DstSwitch42, AcsSwitch41)

        self.addLink(DstSwitch42, AcsSwitch42)

        # Add links between Access Layer and host

        self.addLink(AcsSwitch11, Host11)

        self.addLink(AcsSwitch11, Host12)

        self.addLink(AcsSwitch12, Host13)

        self.addLink(AcsSwitch12, Host14)

        self.addLink(AcsSwitch21, Host21)

        self.addLink(AcsSwitch21, Host22)

        self.addLink(AcsSwitch22, Host23)

        self.addLink(AcsSwitch22, Host24)

        self.addLink(AcsSwitch31, Host31)

        self.addLink(AcsSwitch31, Host32)

        self.addLink(AcsSwitch32, Host33)

        self.addLink(AcsSwitch32, Host34)

        self.addLink(AcsSwitch41, Host41)

        self.addLink(AcsSwitch41, Host42)

        self.addLink(AcsSwitch42, Host43)

        self.addLink(AcsSwitch42, Host44)

def run():

    topo = FatTree()

    net = Mininet(topo)

    net.start()

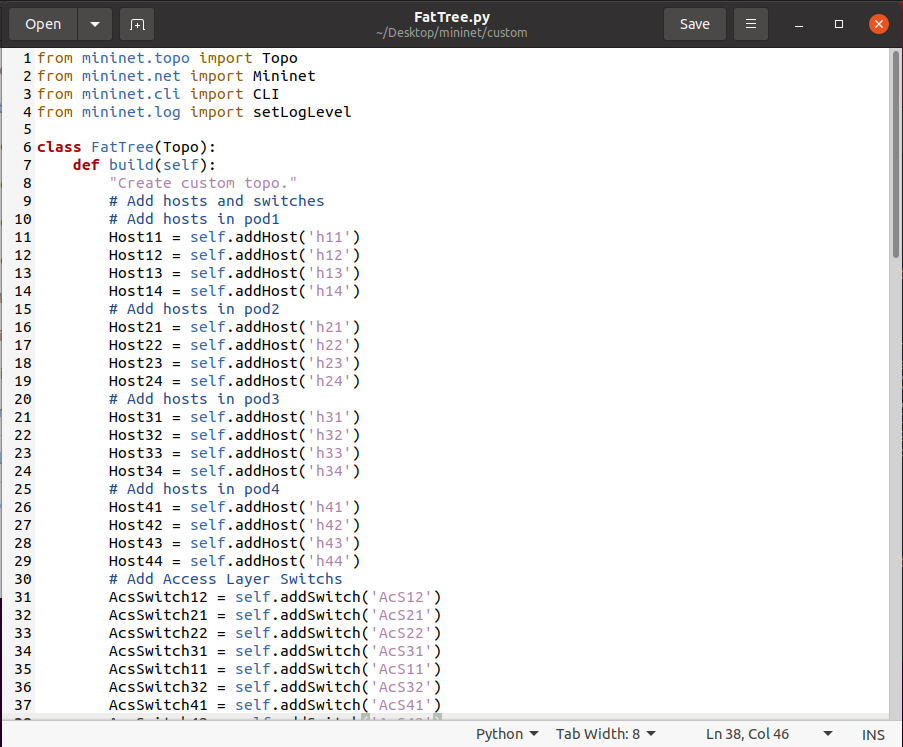
    CLI(net)

    net.stop()

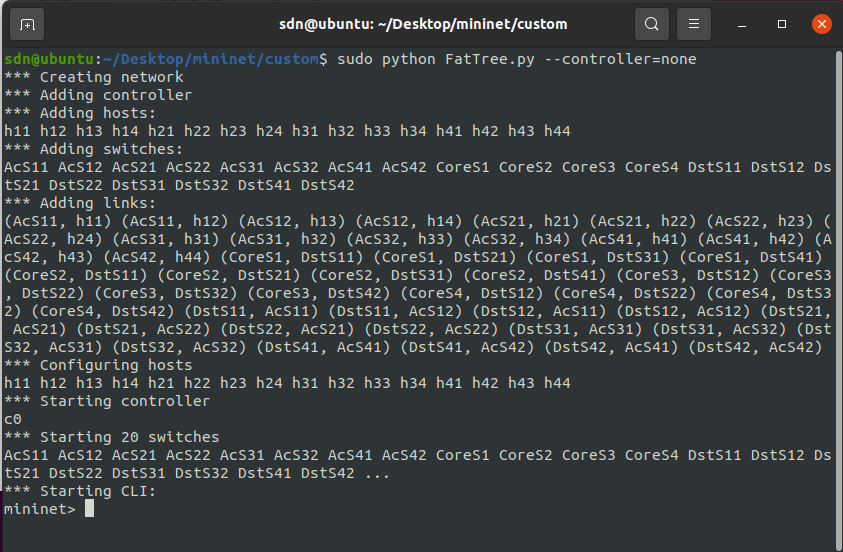
if \_\_name\_\_ == '\_\_main\_\_':

    setLogLevel('info')

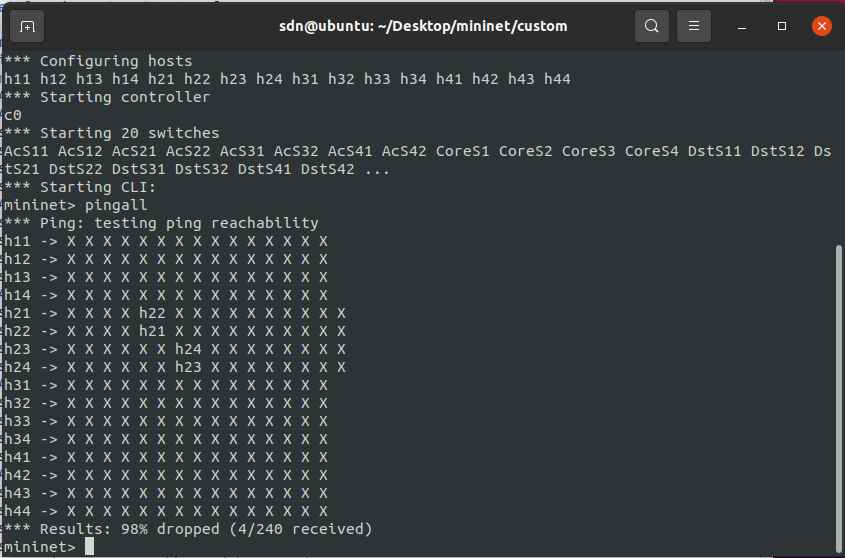
    run()



创建的FatTree拓扑结构文件命名为FatTree.py，在mininet/custom/目录下，打开命令行终端，运行sudo python FatTree.py --controller=none

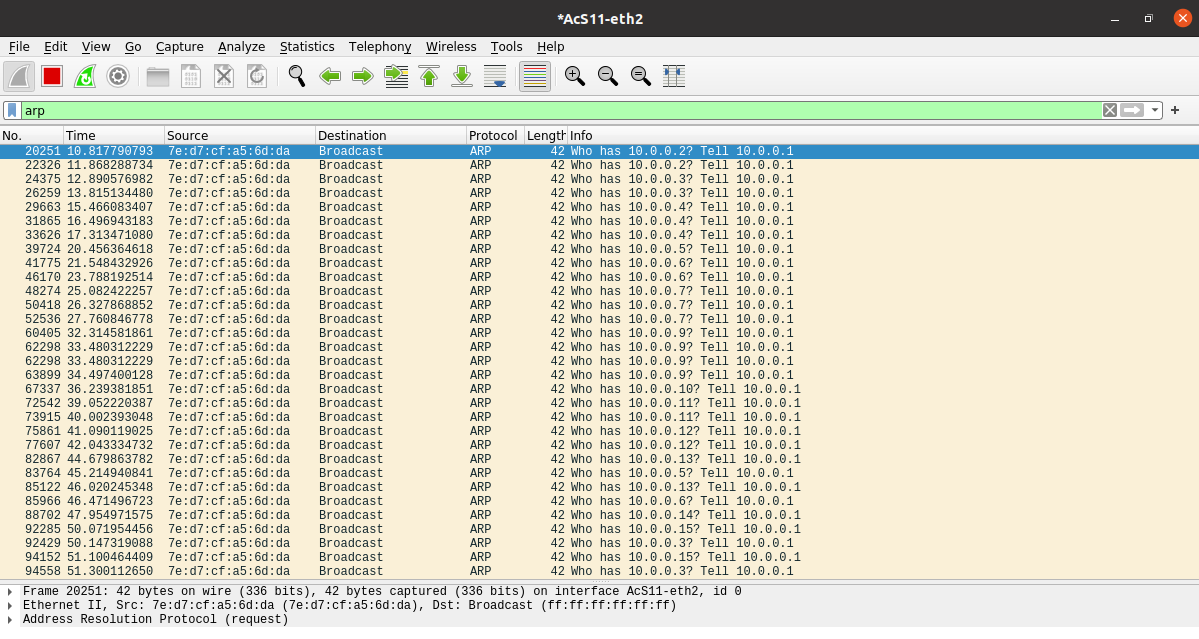


创建FatTree拓扑结构后，使用pingall指令测试各节点之间的连通性。



可以发现，pingall之后所有主机无响应，无法连通。

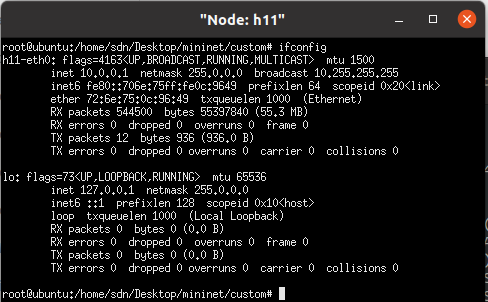
在命令行中输入sudo wireshark，使用Wireshark抓包，我们选择Acs11交换机的Ethernet2端口，进行抓包，分析情况。



情况分析：

可以看到，问题出现在IP地址为10.0.0.1的h11主机发出的ARP请求报文得不到应答。ARP请求报文得不到应答，数据链路层无法确认转发方向，导致网络节点不可达。

在Mininet命令行中输入xterm h11，为h11主机开启一个终端，在终端内输入ifconfig，查看h11主机的网络信息。



可以看到，h11主机的IP地址确实为10.0.0.1，发出的ARP请求报文得不到响应。

问题分析：

在网上查找相关资料后得到解答：k = 4和k = 2情况最大的不同，应该是拓扑中出现了环路。这时，用来探测网络连接、更新路由等信息的数据包可能形成广播风暴，让网络处于极度拥塞的状态下，所以无法ping通。

解决方法：

在网上查找相关资料后，我们得到解决方法为为网络开启生成树协议（STP），这样可以明确转发路径，进而避免数据包在一个环上来回转发。

我们需要修改代码，具体为from mininet.node import OVSBridge，同时设置参数switchOpts = {'cls': OVSBridge,'stp': 1}，在addSwitch()函数中增加参数，代码为Switch = self.addSwitch('Name', \*\*switchOpts)，为交换机开启生成树协议。此外，在net.start()函数后增加net.waitConnected()函数，等待各网络节点之间相互连接完成。

修改后的代码如下：

from mininet.topo import Topo

from mininet.net import Mininet

from mininet.cli import CLI

from mininet.log import setLogLevel

from mininet.node import OVSBridge

class FatTree(Topo):

    def build(self):

        switchOpts = {'cls': OVSBridge,'stp': 1}

        "Create custom topo."

        # Add hosts and switches

        # Add hosts in pod1

        Host11 = self.addHost('h11')

        Host12 = self.addHost('h12')

        Host13 = self.addHost('h13')

        Host14 = self.addHost('h14')

        # Add hosts in pod2

        Host21 = self.addHost('h21')

        Host22 = self.addHost('h22')

        Host23 = self.addHost('h23')

        Host24 = self.addHost('h24')

        # Add hosts in pod3

        Host31 = self.addHost('h31')

        Host32 = self.addHost('h32')

        Host33 = self.addHost('h33')

        Host34 = self.addHost('h34')

        # Add hosts in pod4

        Host41 = self.addHost('h41')

        Host42 = self.addHost('h42')

        Host43 = self.addHost('h43')

        Host44 = self.addHost('h44')

        # Add Access Layer Switchs

        AcsSwitch12 = self.addSwitch('AcS12', \*\*switchOpts)

        AcsSwitch21 = self.addSwitch('AcS21', \*\*switchOpts)

        AcsSwitch22 = self.addSwitch('AcS22', \*\*switchOpts)

        AcsSwitch31 = self.addSwitch('AcS31', \*\*switchOpts)

        AcsSwitch11 = self.addSwitch('AcS11', \*\*switchOpts)

        AcsSwitch32 = self.addSwitch('AcS32', \*\*switchOpts)

        AcsSwitch41 = self.addSwitch('AcS41', \*\*switchOpts)

        AcsSwitch42 = self.addSwitch('AcS42', \*\*switchOpts)

        # Add Distribution Layer Switchs

        DstSwitch11 = self.addSwitch('DstS11', \*\*switchOpts)

        DstSwitch12 = self.addSwitch('DstS12', \*\*switchOpts)

        DstSwitch21 = self.addSwitch('DstS21', \*\*switchOpts)

        DstSwitch22 = self.addSwitch('DstS22', \*\*switchOpts)

        DstSwitch31 = self.addSwitch('DstS31', \*\*switchOpts)

        DstSwitch32 = self.addSwitch('DstS32', \*\*switchOpts)

        DstSwitch41 = self.addSwitch('DstS41', \*\*switchOpts)

        DstSwitch42 = self.addSwitch('DstS42', \*\*switchOpts)

        # Add Core Layer Switchs

        CoreSwitch1 = self.addSwitch('CoreS1', \*\*switchOpts)

        CoreSwitch2 = self.addSwitch('CoreS2', \*\*switchOpts)

        CoreSwitch3 = self.addSwitch('CoreS3', \*\*switchOpts)

        CoreSwitch4 = self.addSwitch('CoreS4', \*\*switchOpts)

        # Add links

        # Add links between Core Layer and Distribution Layer

        self.addLink(CoreSwitch1, DstSwitch11)

        self.addLink(CoreSwitch1, DstSwitch21)

        self.addLink(CoreSwitch1, DstSwitch31)

        self.addLink(CoreSwitch1, DstSwitch41)

        self.addLink(CoreSwitch2, DstSwitch11)

        self.addLink(CoreSwitch2, DstSwitch21)

        self.addLink(CoreSwitch2, DstSwitch31)

        self.addLink(CoreSwitch2, DstSwitch41)

        self.addLink(CoreSwitch3, DstSwitch12)

        self.addLink(CoreSwitch3, DstSwitch22)

        self.addLink(CoreSwitch3, DstSwitch32)

        self.addLink(CoreSwitch3, DstSwitch42)

        self.addLink(CoreSwitch4, DstSwitch12)

        self.addLink(CoreSwitch4, DstSwitch22)

        self.addLink(CoreSwitch4, DstSwitch32)

        self.addLink(CoreSwitch4, DstSwitch42)

        # Add links between Distribution Layer and Access Layer

        self.addLink(DstSwitch11, AcsSwitch11)

        self.addLink(DstSwitch11, AcsSwitch12)

        self.addLink(DstSwitch12, AcsSwitch11)

        self.addLink(DstSwitch12, AcsSwitch12)

        self.addLink(DstSwitch21, AcsSwitch21)

        self.addLink(DstSwitch21, AcsSwitch22)

        self.addLink(DstSwitch22, AcsSwitch21)

        self.addLink(DstSwitch22, AcsSwitch22)

        self.addLink(DstSwitch31, AcsSwitch31)

        self.addLink(DstSwitch31, AcsSwitch32)

        self.addLink(DstSwitch32, AcsSwitch31)

        self.addLink(DstSwitch32, AcsSwitch32)

        self.addLink(DstSwitch41, AcsSwitch41)

        self.addLink(DstSwitch41, AcsSwitch42)

        self.addLink(DstSwitch42, AcsSwitch41)

        self.addLink(DstSwitch42, AcsSwitch42)

        # Add links between Access Layer and host

        self.addLink(AcsSwitch11, Host11)

        self.addLink(AcsSwitch11, Host12)

        self.addLink(AcsSwitch12, Host13)

        self.addLink(AcsSwitch12, Host14)

        self.addLink(AcsSwitch21, Host21)

        self.addLink(AcsSwitch21, Host22)

        self.addLink(AcsSwitch22, Host23)

        self.addLink(AcsSwitch22, Host24)

        self.addLink(AcsSwitch31, Host31)

        self.addLink(AcsSwitch31, Host32)

        self.addLink(AcsSwitch32, Host33)

        self.addLink(AcsSwitch32, Host34)

        self.addLink(AcsSwitch41, Host41)

        self.addLink(AcsSwitch41, Host42)

        self.addLink(AcsSwitch42, Host43)

        self.addLink(AcsSwitch42, Host44)

def run():

    topo = FatTree()

    net = Mininet(topo)

    net.start()

    net.waitConnected()

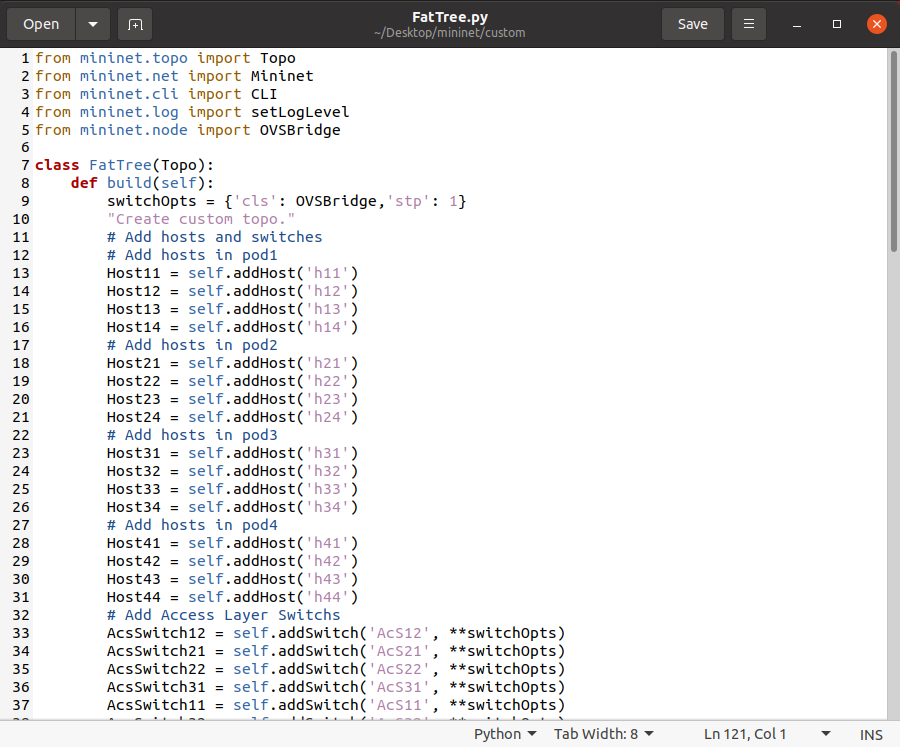
    CLI(net)

    net.stop()

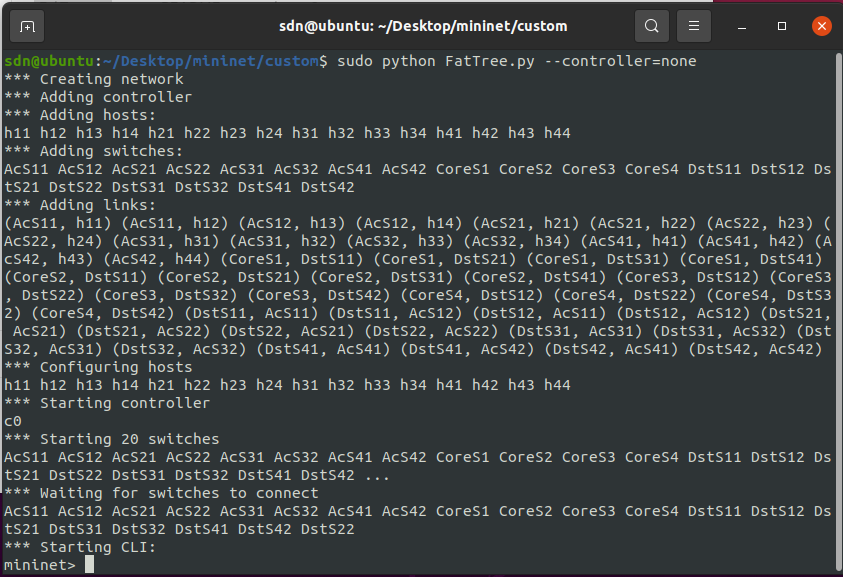
if \_\_name\_\_ == '\_\_main\_\_':

    setLogLevel('info')

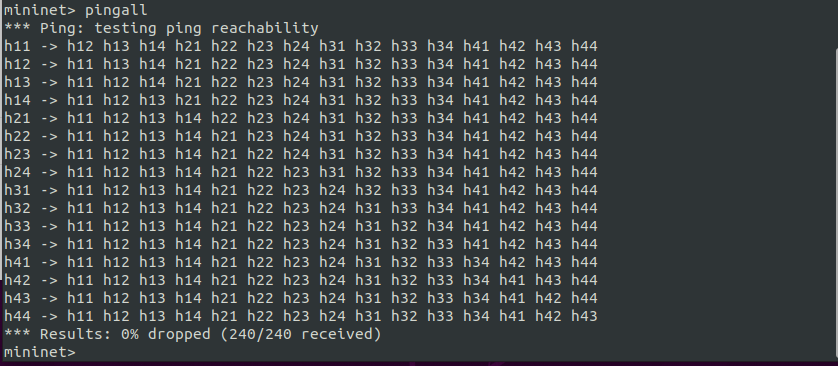
run()



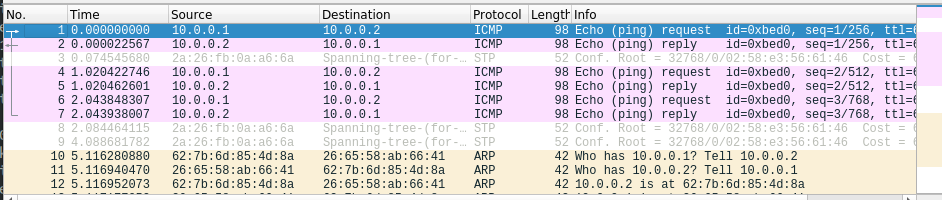
修改后的FatTree拓扑结构文件仍命名为FatTree.py，在mininet/custom/目录下，打开命令行终端，运行sudo python FatTree.py --controller=none

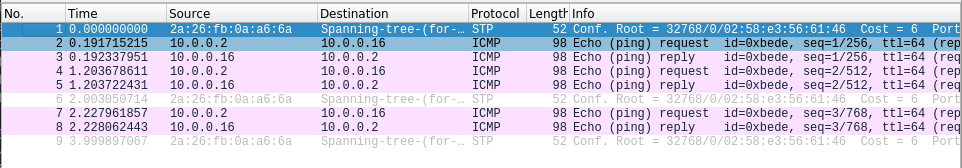


使用pingall，测试各节点之间的连通性。



可以看到，这次所有节点之间都能相互ping通，网络连通性没有问题。

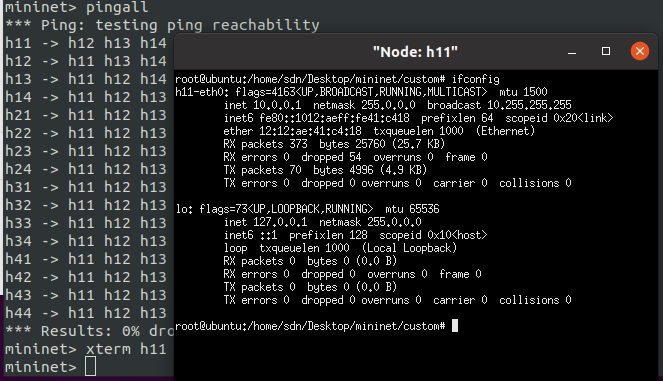




4. 利用MAC表，分析数据包转发路径

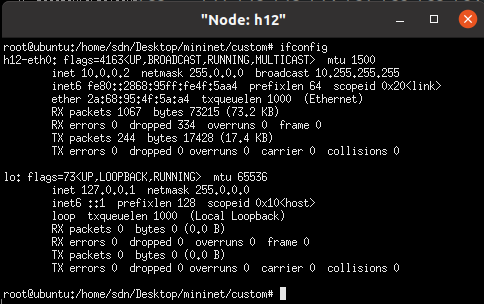
我们首先进行一个简单的路径分析，分析与同一个连接层(Access Layer)的交换机AcS11连接的两台主机h11与h12的数据包转发路径。

在Mininet命令行中输入xterm h11，打开h11主机的终端，在终端内输入ifconfig，查看h11主机的网络信息。



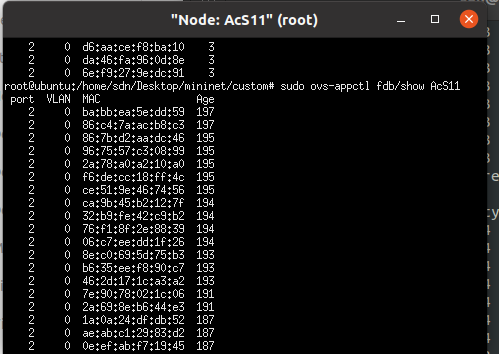
可以看到，h11主机的MAC地址为12:12:AE:41:C4:18。

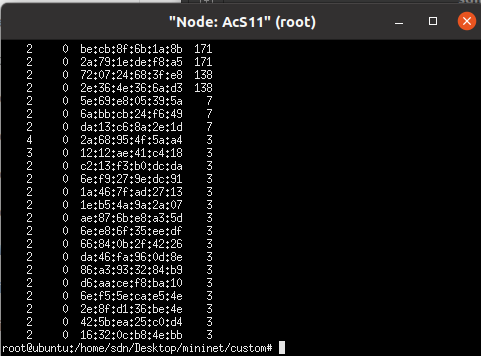
同理，在Mininet命令行中输入xterm h12，打开h12主机的终端，在终端内输入ifconfig，查看h12主机的网络信息。



可以看到，h12主机的MAC地址为2A:68:95:4F:5A:A4。

在Mininet命令行中输入xterm AcS11，打开交换机AcS11终端，在终端内输入sudo ovs-appctl fdb/show AcS11，查看AcS11的MAC表。



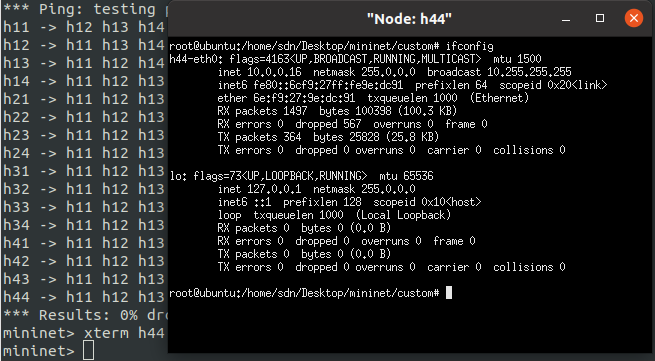


可以看到，h11主机的MAC地址出现在了MAC表中，连接Eth3端口，h12主机的MAC地址也出现在了MAC表中，连接Eth4端口。则我们可以根据信息，得到h11与h12主机之间的数据转发路径为：

h11 -> AcS11 Eth3 -> AcS12 Eth4 -> h12

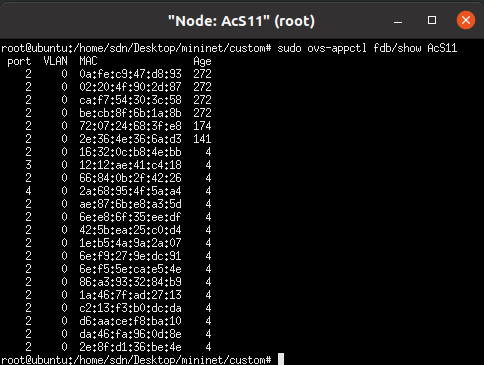
我们再进行一个更复杂的路径分析，分析h11主机与h44主机之间的数据转发路径。同时我们要偶然间在Mininet命令行中进行pingall操作，防止各节点的MAC表过期。

在Mininet命令行中输入xterm h44，打开h44主机的终端，在终端内输入ifconfig，查看h44主机的网络信息。



可以看到，h44主机的MAC地址为6E:F9:27:9E:DC:91。

在Mininet命令行中输入xterm AcS11，打开交换机AcS11终端，在终端内输入sudo ovs-appctl fdb/show AcS11，查看AcS11的MAC表。



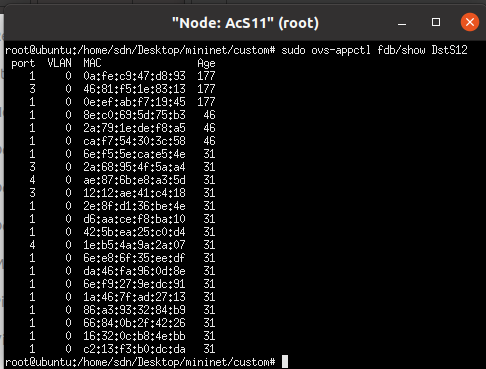
可以看到，h11主机通过Eth3端口与交换机AcS11相连，而h44主机通过Eth2端口与交换机AcS11相连。

在xterm命令行终端中输入ifconfig，查看AcS11的Eth2端口的MAC地址。



可以看到，AcS11的Eth2端口的MAC地址为46:81:F5:1E:83:13。

在xterm命令行终端内输入sudo ovs-appctl fdb/show DstS12，查看DstS12的MAC表。



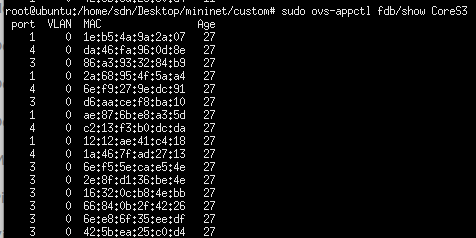
可以看到，AcS11的Eth2端口与DstS12的Eth3端口相连，而h44通过Eth1端口与DstS12相连。

在xterm命令行终端中输入ifconfig，查看DstS12的Eth1端口的MAC地址。



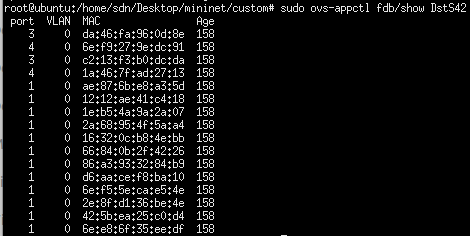
可以看到，DstS12的Eth1端口的MAC地址为96:22:B2:3D:75:3C

在xterm命令行终端内输入sudo ovs-appctl fdb/show CoreS3，查看CoreS3的MAC表。



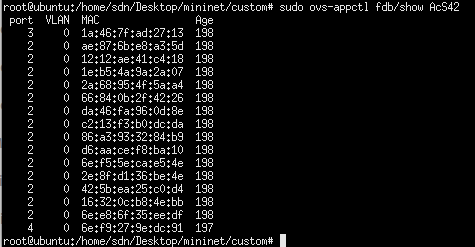
可以看到，h11主机通过Eth1端口与CoreS3相连，而h44主机通过Eth4端口与CoreS3相连。

在xterm命令行终端内输入sudo ovs-appctl fdb/show DstS42，查看DstS42的MAC表。



可以看到，h11主机通过Eth1端口与DstS42相连，而h44主机通过Eth4端口与DstS42相连。

在xterm命令行终端内输入sudo ovs-appctl fdb/show AcS42，查看AcS42的MAC表。

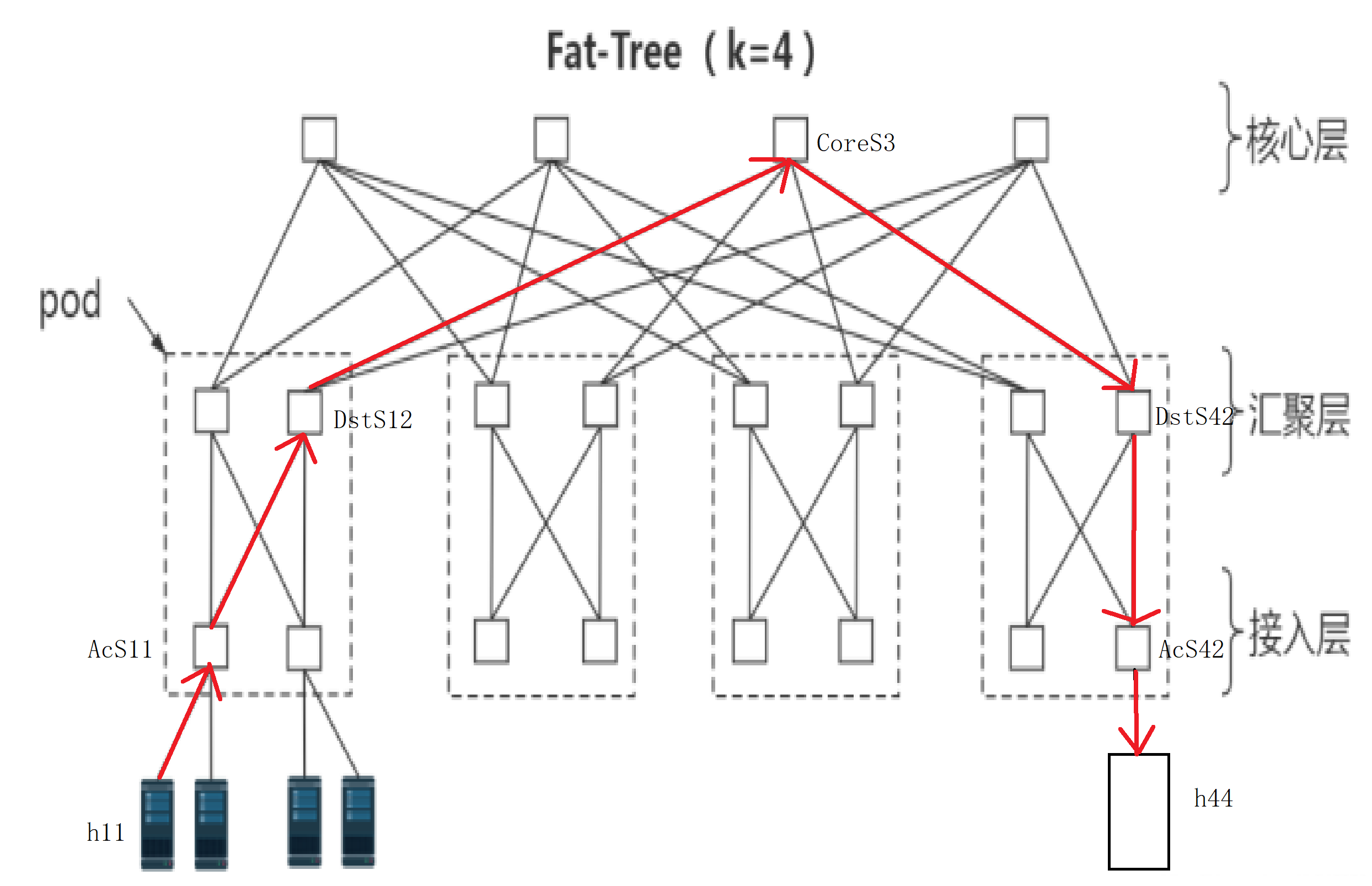


可以看到，h11主机通过Eth2端口与AcS42相连，而h44主机通过Eth4端口与AcS42相连。

综上所述，我们总结得到h11与h44主机之间的数据转发路径：

h11 -> AcS11 Eth3 -> DstS12 Eth3 -> CoreS3 Eth1 -> CoreS3 Eth4 -> DstS42 Eth4 -> AcS42 Eth4 -> h44

在示意图上表示如下：



**实验结果**

1. 熟悉了mininet命令行的基本操作，学会了在mininet命令行中查看当前网络的拓扑结构，利用pingall命令测试网络连通性，使用xterm命令在某个特定网络节点上打开终端。

2. 了解了FatTree拓扑结构的形状与特点，学会了使用Mininet Python API构建拓扑网络。

3. 了解了STP协议在网络中的作用，学会了为交换机开启STP协议。

4. 学会了在特定网络设备上启动Wireshark进行抓包，分析报文信息。

5. 学会了利用MAC表与ovs-appctl fdb/show以及ifconfig指令，分析各网络节点之间的通信情况与数据转发路径。