

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:

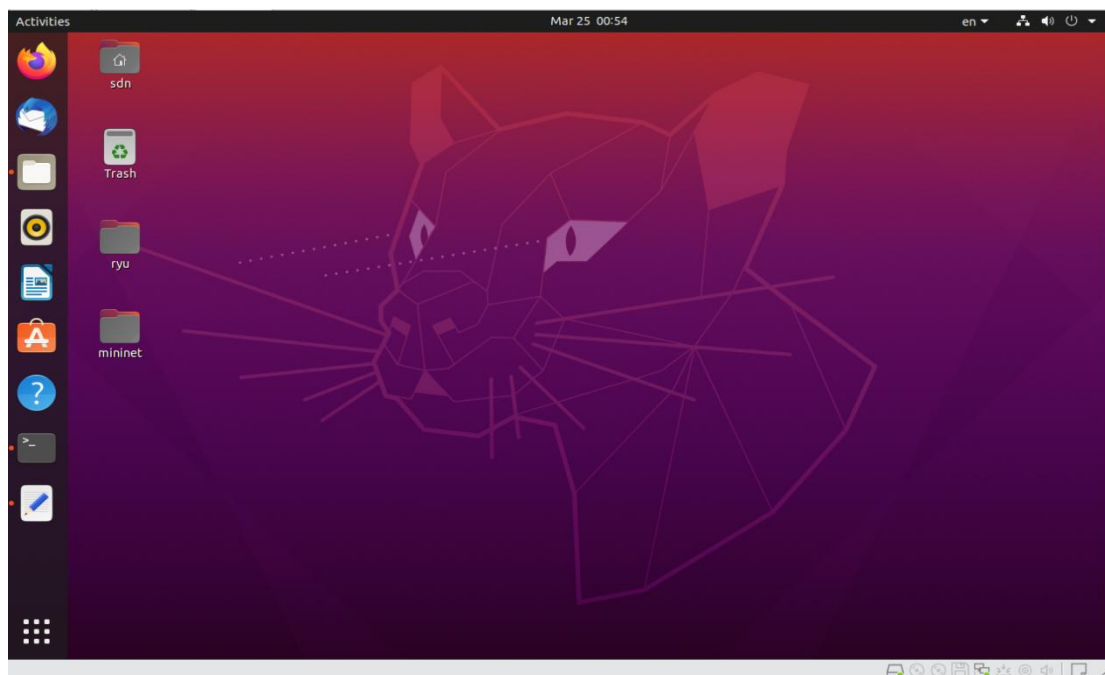
- Construct a fattree (k=4) topology using Mininet Python API
- Make sure all hosts are reachable
- If not reachable, try to solve it
- If reachable, show the path between two hosts
- Don't use controller! (use '--controller=None' option)

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。

```
sdn@ubuntu: ~/Desktop/mininet/custom
sdn@ubuntu:~/Desktop/mininet/custom$ sudo mn
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

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。

```
sdn@ubuntu: ~/Desktop/mininet/custom
mininet> nodes
available nodes are:
c0 h1 h2 s1
mininet> links
h1-eth0<->s1-eth1 (OK OK)
h2-eth0<->s1-eth2 (OK OK)
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
mininet>
```

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

```
sdn@ubuntu: ~/Desktop/mininet/custom
mininet> exit
*** Stopping 1 controllers
c0
*** Stopping 2 links
..
*** Stopping 1 switches
s1
*** Stopping 2 hosts
h1 h2
*** Done
completed in 273.134 seconds
sdn@ubuntu:~/Desktop/mininet/custom$ sudo mn -c
*** Removing excess controllers/ofprotocols/ofdatapaths/pings/noxes
killall controller ofprotocol ofdatapath ping nox_corelt-nox_core ovs-openflowd
ovs-controllerovs-testcontroller udpbwtest mnexec ivs ryu-manager 2> /dev/null
killall -9 controller ofprotocol ofdatapath ping nox_corelt-nox_core ovs-openflo
wd ovs-controllerovs-testcontroller udpbwtest mnexec ivs ryu-manager 2> /dev/nul
l
pkill -9 -f "sudo mnexec"
*** Removing junk from /tmp
rm -f /tmp/vconn* /tmp/vlogs* /tmp/*.out /tmp/*.log
*** Removing old X11 tunnels
*** Removing excess kernel datapaths
ps ax | egrep -o 'dp[0-9]+' | sed 's/dp/nl:/'
```

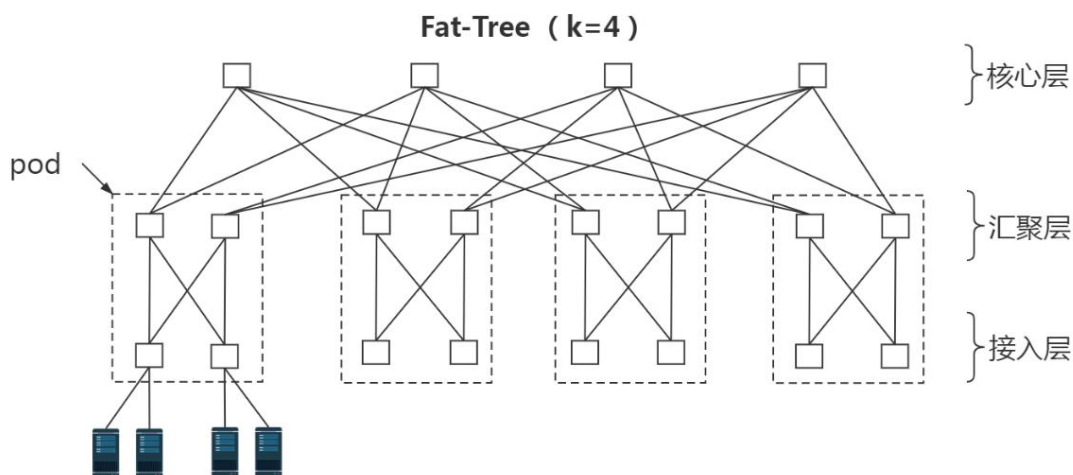
3. 使用 Mininet 的 Python API, 创建 k=4 的 FatTree 拓扑

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
        AcSwitch12 = self.addSwitch('AcS12')
        AcSwitch21 = self.addSwitch('AcS21')
        AcSwitch22 = self.addSwitch('AcS22')
        AcSwitch31 = self.addSwitch('AcS31')
        AcSwitch11 = self.addSwitch('AcS11')
        AcSwitch32 = self.addSwitch('AcS32')
        AcSwitch41 = self.addSwitch('AcS41')
        AcSwitch42 = self.addSwitch('AcS42')
        # Add Distribution Layer Switchs
        DstSwitch11 = self.addSwitch('DstS11')
        DstSwitch12 = self.addSwitch('DstS12')
        DstSwitch21 = self.addSwitch('DstS21')
```

```

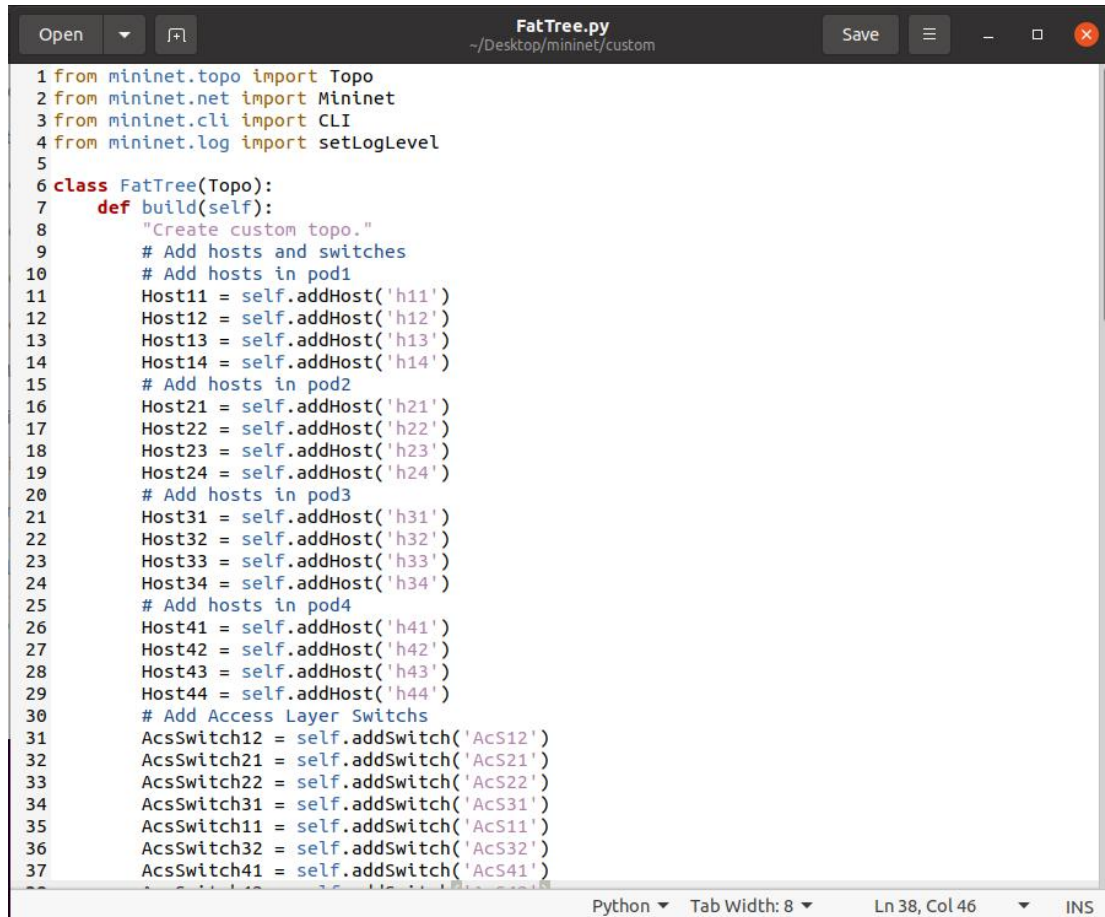
DstSwitch22 = self.addSwitch('DstS22')
DstSwitch31 = self.addSwitch('DstS31')
DstSwitch32 = self.addSwitch('DstS32')
DstSwitch41 = self.addSwitch('DstS41')
DstSwitch42 = self.addSwitch('DstS42')
# Add Core Layer Switchs
CoreSwitch1 = self.addSwitch('CoreS1')
CoreSwitch2 = self.addSwitch('CoreS2')
CoreSwitch3 = 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)
```

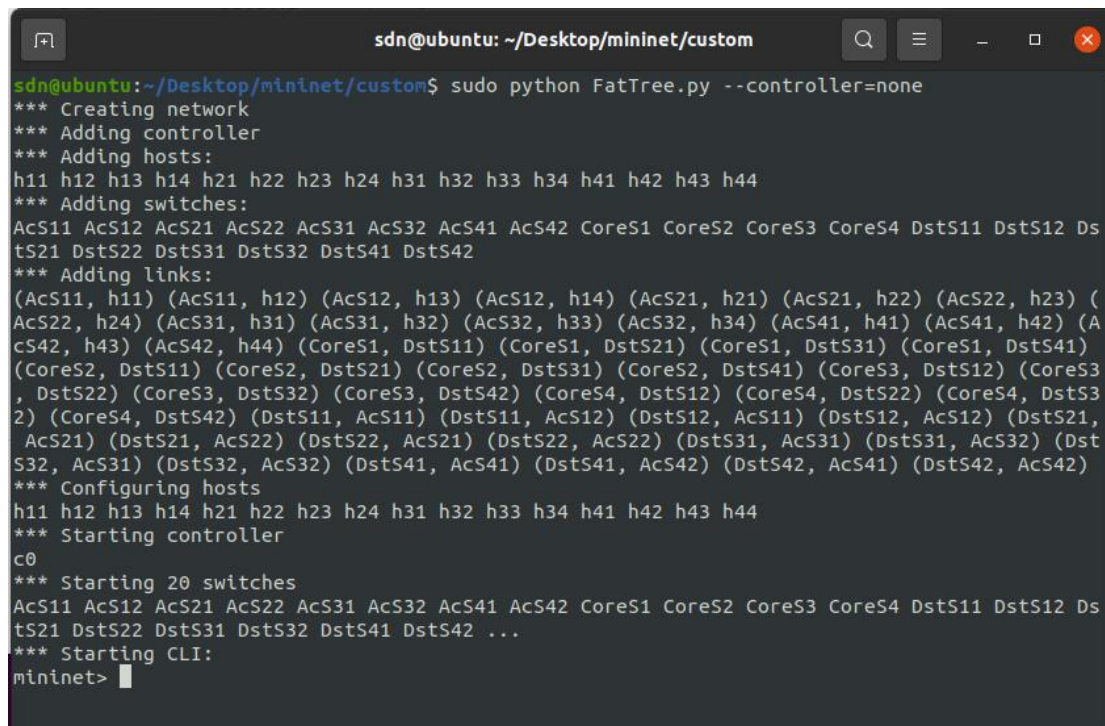
```
def run():
    topo = FatTree()
    net = Mininet(topo)
    net.start()
    CLI(net)
    net.stop()

if __name__ == '__main__':
    setLogLevel('info')
    run()
```


A screenshot of a code editor window titled 'FatTree.py' with the path '~/Desktop/mininet/custom'. The editor contains a Python script for creating a FatTree topology. The script imports classes from mininet (Topo, Mininet, CLI, setLogLevel) and defines a FatTree class with a build method. The build method adds hosts in four pods (pod1, pod2, pod3, pod4) and access layer switches (AcS11 to AcS41).

```
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
6 class FatTree(Topo):
7     def build(self):
8         "Create custom topo."
9         # Add hosts and switches
10        # Add hosts in pod1
11        Host11 = self.addHost('h11')
12        Host12 = self.addHost('h12')
13        Host13 = self.addHost('h13')
14        Host14 = self.addHost('h14')
15        # Add hosts in pod2
16        Host21 = self.addHost('h21')
17        Host22 = self.addHost('h22')
18        Host23 = self.addHost('h23')
19        Host24 = self.addHost('h24')
20        # Add hosts in pod3
21        Host31 = self.addHost('h31')
22        Host32 = self.addHost('h32')
23        Host33 = self.addHost('h33')
24        Host34 = self.addHost('h34')
25        # Add hosts in pod4
26        Host41 = self.addHost('h41')
27        Host42 = self.addHost('h42')
28        Host43 = self.addHost('h43')
29        Host44 = self.addHost('h44')
30        # Add Access Layer Switches
31        AcS11 = self.addSwitch('AcS11')
32        AcS12 = self.addSwitch('AcS12')
33        AcS21 = self.addSwitch('AcS21')
34        AcS22 = self.addSwitch('AcS22')
35        AcS31 = self.addSwitch('AcS31')
36        AcS32 = self.addSwitch('AcS32')
37        AcS41 = self.addSwitch('AcS41')
```

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`

A screenshot of a terminal window titled 'sdn@ubuntu: ~/Desktop/mininet/custom'. The terminal shows the execution of the command 'sudo python FatTree.py --controller=None'. The output displays the steps of network creation: adding controller, hosts, switches, and links, followed by starting the controller, switches, and CLI.

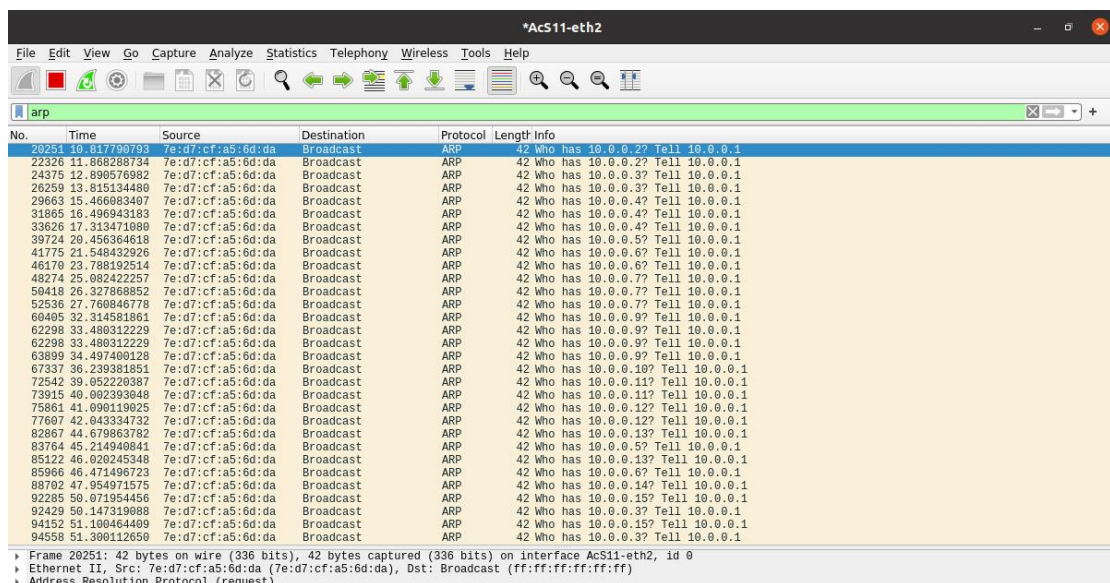
```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo python FatTree.py --controller=None
*** Creating network
*** Adding controller
*** Adding hosts:
h11 h12 h13 h14 h21 h22 h23 h24 h31 h32 h33 h34 h41 h42 h43 h44
*** Adding switches:
AcS11 AcS12 AcS21 AcS22 AcS31 AcS32 AcS41 AcS42 CoreS1 CoreS2 CoreS3 CoreS4 DstS11 DstS12 DstS21 DstS22 DstS31 DstS32 DstS41 DstS42
*** Adding links:
(AcS11, h11) (AcS11, h12) (AcS12, h13) (AcS12, h14) (AcS21, h21) (AcS21, h22) (AcS22, h23) (AcS22, h24) (AcS31, h31) (AcS31, h32) (AcS32, h33) (AcS32, h34) (AcS41, h41) (AcS41, h42) (AcS42, h43) (AcS42, h44) (CoreS1, DstS11) (CoreS1, DstS21) (CoreS1, DstS31) (CoreS1, DstS41) (CoreS2, DstS11) (CoreS2, DstS21) (CoreS2, DstS31) (CoreS2, DstS41) (CoreS3, DstS12) (CoreS3, DstS22) (CoreS3, DstS32) (CoreS3, DstS42) (CoreS4, DstS12) (CoreS4, DstS22) (CoreS4, DstS32) (CoreS4, DstS42) (DstS11, AcS11) (DstS11, AcS12) (DstS12, AcS11) (DstS12, AcS12) (DstS21, AcS21) (DstS21, AcS22) (DstS22, AcS21) (DstS22, AcS22) (DstS31, AcS31) (DstS31, AcS32) (DstS32, AcS31) (DstS32, AcS32) (DstS41, AcS41) (DstS41, AcS42) (DstS42, AcS41) (DstS42, AcS42)
*** Configuring hosts
h11 h12 h13 h14 h21 h22 h23 h24 h31 h32 h33 h34 h41 h42 h43 h44
*** Starting controller
c0
*** Starting 20 switches
AcS11 AcS12 AcS21 AcS22 AcS31 AcS32 AcS41 AcS42 CoreS1 CoreS2 CoreS3 CoreS4 DstS11 DstS12 DstS21 DstS22 DstS31 DstS32 DstS41 DstS42 ...
*** Starting CLI:
mininet>
```

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

```
sdn@ubuntu: ~/Desktop/mininet/custom
*** Configuring hosts
h11 h12 h13 h14 h21 h22 h23 h24 h31 h32 h33 h34 h41 h42 h43 h44
*** Starting controller
c0
*** Starting 20 switches
AcS11 AcS12 AcS21 AcS22 AcS31 AcS32 AcS41 AcS42 CoreS1 CoreS2 CoreS3 CoreS4 DstS11 DstS12 DstS21 DstS22 DstS31 DstS32 DstS41 DstS42 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h11 -> X X X X X X X X X X X X X X
h12 -> X X X X X X X X X X X X X X
h13 -> X X X X X X X X X X X X X X
h14 -> X X X X X X X X X X X X X X
h21 -> X X X X h22 X X X X X X X X
h22 -> X X X X h21 X X X X X X X X
h23 -> X X X X X h24 X X X X X X X
h24 -> X X X X X h23 X X X X X X X
h31 -> X X X X X X X X X X X X X X
h32 -> X X X X X X X X X X X X X X
h33 -> X X X X X X X X X X X X X X
h34 -> X X X X X X X X X X X X X X
h41 -> X X X X X X X X X X X X X X
h42 -> X X X X X X X X X X X X X X
h43 -> X X X X X X X X X X X X X X
h44 -> X X X X X X X X X X X X X X
*** Results: 98% dropped (4/240 received)
mininet>
```

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。



The screenshot shows the Wireshark interface with a packet capture on the AcS11-eth2 interface. The filter is set to 'arp'. The packet list shows a series of ARP requests from 10.0.0.1 to the broadcast address ff:ff:ff:ff:ff:ff. The packet details pane shows the structure of an ARP request.

No.	Time	Source	Destination	Protocol	Length	Info
20251	10.917790798	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
22326	11.868288734	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
24375	12.890576982	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
26259	13.815134480	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
29663	15.466083407	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1
31865	16.496943383	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1
33626	17.313471080	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1
39724	20.456364618	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.5? Tell 10.0.0.1
41775	21.548432926	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.6? Tell 10.0.0.1
46170	23.788192514	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.6? Tell 10.0.0.1
48274	25.082422257	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.7? Tell 10.0.0.1
50418	26.327868852	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.7? Tell 10.0.0.1
52536	27.760846778	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.7? Tell 10.0.0.1
60405	32.314581861	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.9? Tell 10.0.0.1
62298	33.480312229	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.9? Tell 10.0.0.1
62298	33.480312229	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.9? Tell 10.0.0.1
63899	34.497400128	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.9? Tell 10.0.0.1
67337	36.239381851	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.10? Tell 10.0.0.1
72542	39.052220387	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.11? Tell 10.0.0.1
73915	40.062393048	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.11? Tell 10.0.0.1
75861	41.090119025	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.12? Tell 10.0.0.1
77607	42.043333473	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.12? Tell 10.0.0.1
82867	44.679863782	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.13? Tell 10.0.0.1
83764	45.214940841	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.5? Tell 10.0.0.1
85122	46.020245348	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.13? Tell 10.0.0.1
85966	46.471496723	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.6? Tell 10.0.0.1
88702	47.95491575	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.14? Tell 10.0.0.1
92285	50.071954456	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.15? Tell 10.0.0.1
92429	50.147319088	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
94152	51.100464409	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.15? Tell 10.0.0.1
94558	51.300112650	7e:d7:cf:a5:6d:da	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1

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。


```
"Node: h11"
root@ubuntu:/home/sdn/Desktop/mininet/custom# ifconfig
h11-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::706e:75ff:fe0c:9649 prefixlen 64 scopeid 0x20<link>
    ether 72:6e:75:0c:96:49 txqueuelen 1000 (Ethernet)
    RX packets 544500 bytes 55397840 (55.3 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 12 bytes 936 (936.0 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 (Local Loopback)
    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

root@ubuntu:/home/sdn/Desktop/mininet/custom#
```

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)

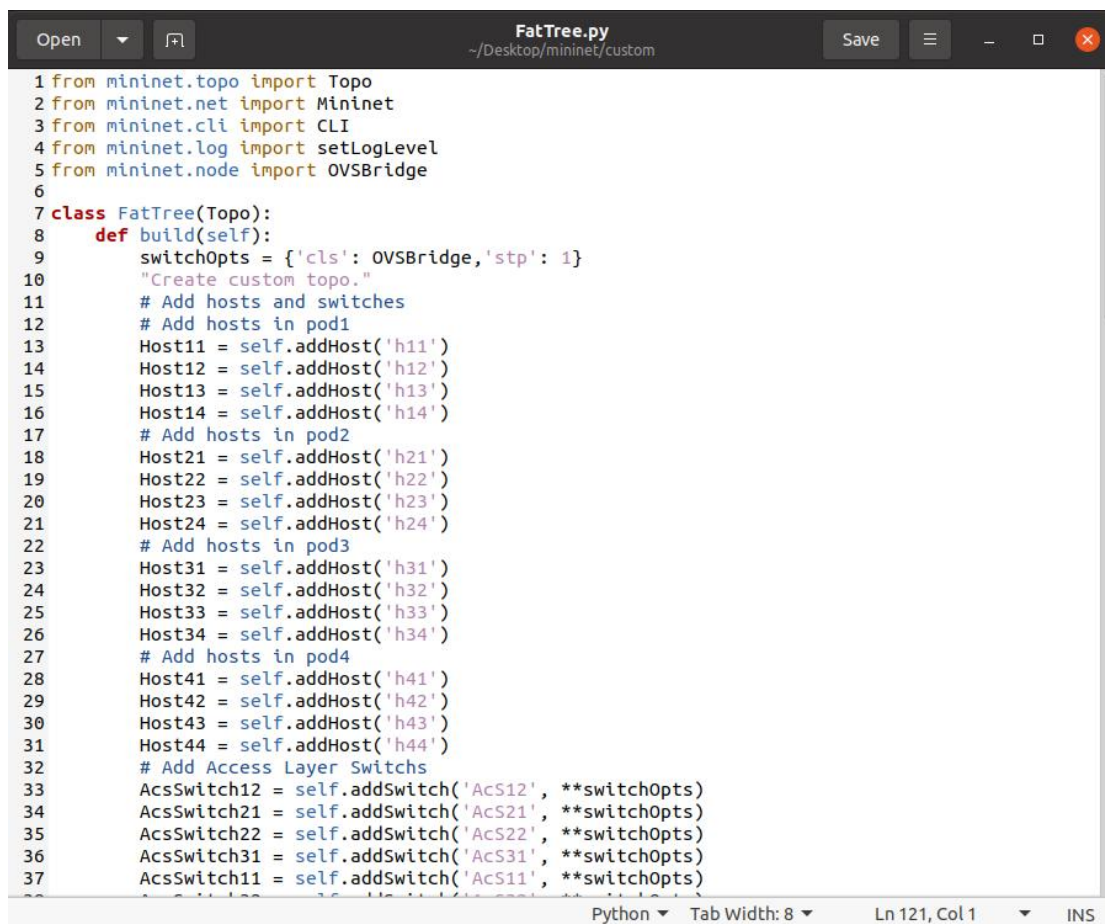
```

```

def run():
    topo = FatTree()
    net = Mininet(topo)
    net.start()
    net.waitConnected()
    CLI(net)
    net.stop()

if __name__ == '__main__':
    setLogLevel('info')
    run()

```



```

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 from mininet.node import OVSBridge
6
7 class FatTree(Topo):
8     def build(self):
9         switchOpts = {'cls': OVSBridge, 'stp': 1}
10        "Create custom topo."
11        # Add hosts and switches
12        # Add hosts in pod1
13        Host11 = self.addHost('h11')
14        Host12 = self.addHost('h12')
15        Host13 = self.addHost('h13')
16        Host14 = self.addHost('h14')
17        # Add hosts in pod2
18        Host21 = self.addHost('h21')
19        Host22 = self.addHost('h22')
20        Host23 = self.addHost('h23')
21        Host24 = self.addHost('h24')
22        # Add hosts in pod3
23        Host31 = self.addHost('h31')
24        Host32 = self.addHost('h32')
25        Host33 = self.addHost('h33')
26        Host34 = self.addHost('h34')
27        # Add hosts in pod4
28        Host41 = self.addHost('h41')
29        Host42 = self.addHost('h42')
30        Host43 = self.addHost('h43')
31        Host44 = self.addHost('h44')
32        # Add Access Layer Switchs
33        AcsSwitch12 = self.addSwitch('AcS12', **switchOpts)
34        AcsSwitch21 = self.addSwitch('AcS21', **switchOpts)
35        AcsSwitch22 = self.addSwitch('AcS22', **switchOpts)
36        AcsSwitch31 = self.addSwitch('AcS31', **switchOpts)
37        AcsSwitch11 = self.addSwitch('AcS11', **switchOpts)

```

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`


```
sdn@ubuntu: ~/Desktop/mininet/custom
sdn@ubuntu:~/Desktop/mininet/custom$ sudo python FatTree.py --controller=None
*** Creating network
*** Adding controller
*** Adding hosts:
h11 h12 h13 h14 h21 h22 h23 h24 h31 h32 h33 h34 h41 h42 h43 h44
*** Adding switches:
AcS11 AcS12 AcS21 AcS22 AcS31 AcS32 AcS41 AcS42 CoreS1 CoreS2 CoreS3 CoreS4 DstS11 DstS12 DstS21 DstS22 DstS31 DstS32 DstS41 DstS42
*** Adding links:
(AcS11, h11) (AcS11, h12) (AcS12, h13) (AcS12, h14) (AcS21, h21) (AcS21, h22) (AcS22, h23) (AcS22, h24) (AcS31, h31) (AcS31, h32) (AcS32, h33) (AcS32, h34) (AcS41, h41) (AcS41, h42) (AcS42, h43) (AcS42, h44) (CoreS1, DstS11) (CoreS1, DstS21) (CoreS1, DstS31) (CoreS1, DstS41) (CoreS2, DstS11) (CoreS2, DstS21) (CoreS2, DstS31) (CoreS2, DstS41) (CoreS3, DstS11) (CoreS3, DstS21) (CoreS3, DstS31) (CoreS3, DstS41) (CoreS4, DstS11) (CoreS4, DstS21) (CoreS4, DstS31) (CoreS4, DstS41) (DstS11, AcS11) (DstS11, AcS12) (DstS12, AcS11) (DstS12, AcS12) (DstS21, AcS21) (DstS21, AcS22) (DstS22, AcS21) (DstS22, AcS22) (DstS31, AcS31) (DstS31, AcS32) (DstS32, AcS31) (DstS32, AcS32) (DstS41, AcS41) (DstS41, AcS42) (DstS42, AcS41) (DstS42, AcS42)
*** Configuring hosts
h11 h12 h13 h14 h21 h22 h23 h24 h31 h32 h33 h34 h41 h42 h43 h44
*** Starting controller
c0
*** Starting 20 switches
AcS11 AcS12 AcS21 AcS22 AcS31 AcS32 AcS41 AcS42 CoreS1 CoreS2 CoreS3 CoreS4 DstS11 DstS12 DstS21 DstS22 DstS31 DstS32 DstS41 DstS42 ...
*** Waiting for switches to connect
AcS11 AcS12 AcS21 AcS22 AcS31 AcS32 AcS41 AcS42 CoreS1 CoreS2 CoreS3 CoreS4 DstS11 DstS12 DstS21 DstS31 DstS32 DstS41 DstS42 DstS22
*** Starting CLI:
mininet>
```

使用 pingall，测试各节点之间的连通性。

```
mininet> pingall
*** Ping: testing ping reachability
h11 -> h12 h13 h14 h21 h22 h23 h24 h31 h32 h33 h34 h41 h42 h43 h44
h12 -> h11 h13 h14 h21 h22 h23 h24 h31 h32 h33 h34 h41 h42 h43 h44
h13 -> h11 h12 h14 h21 h22 h23 h24 h31 h32 h33 h34 h41 h42 h43 h44
h14 -> h11 h12 h13 h21 h22 h23 h24 h31 h32 h33 h34 h41 h42 h43 h44
h21 -> h11 h12 h13 h14 h22 h23 h24 h31 h32 h33 h34 h41 h42 h43 h44
h22 -> h11 h12 h13 h14 h21 h23 h24 h31 h32 h33 h34 h41 h42 h43 h44
h23 -> h11 h12 h13 h14 h21 h22 h24 h31 h32 h33 h34 h41 h42 h43 h44
h24 -> h11 h12 h13 h14 h21 h22 h23 h31 h32 h33 h34 h41 h42 h43 h44
h31 -> h11 h12 h13 h14 h21 h22 h23 h24 h32 h33 h34 h41 h42 h43 h44
h32 -> h11 h12 h13 h14 h21 h22 h23 h24 h31 h33 h34 h41 h42 h43 h44
h33 -> h11 h12 h13 h14 h21 h22 h23 h24 h31 h32 h34 h41 h42 h43 h44
h34 -> h11 h12 h13 h14 h21 h22 h23 h24 h31 h32 h33 h41 h42 h43 h44
h41 -> h11 h12 h13 h14 h21 h22 h23 h24 h31 h32 h33 h34 h42 h43 h44
h42 -> h11 h12 h13 h14 h21 h22 h23 h24 h31 h32 h33 h34 h41 h43 h44
h43 -> h11 h12 h13 h14 h21 h22 h23 h24 h31 h32 h33 h34 h41 h42 h44
h44 -> h11 h12 h13 h14 h21 h22 h23 h24 h31 h32 h33 h34 h41 h42 h43
*** Results: 0% dropped (240/240 received)
mininet>
```

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。

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0xbcd0, seq=1/256, ttl=64
2	0.000022567	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0xbcd0, seq=1/256, ttl=64
3	0.074545689	2a:26:fb:0a:a6:6a	Spanning-tree-(for-...	STP	52	Conf. Root = 32768/0/02:58:e3:56:61:46 Cost = 6
4	1.020422746	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0xbcd0, seq=2/512, ttl=64
5	1.020462601	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0xbcd0, seq=2/512, ttl=64
6	2.043848307	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0xbcd0, seq=3/768, ttl=64
7	2.043938007	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0xbcd0, seq=3/768, ttl=64
8	2.084464115	2a:26:fb:0a:a6:6a	Spanning-tree-(for-...	STP	52	Conf. Root = 32768/0/02:58:e3:56:61:46 Cost = 6
9	4.088681782	2a:26:fb:0a:a6:6a	Spanning-tree-(for-...	STP	52	Conf. Root = 32768/0/02:58:e3:56:61:46 Cost = 6
10	5.116280880	62:7b:6d:85:4d:8a	26:65:58:ab:66:41	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
11	5.116940470	26:65:58:ab:66:41	62:7b:6d:85:4d:8a	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
12	5.116952073	62:7b:6d:85:4d:8a	26:65:58:ab:66:41	ARP	42	10.0.0.2 is at 62:7b:6d:85:4d:8a

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	2a:26:fb:0a:a6:6a	Spanning-tree-(for-...	STP	52	Conf. Root = 32768/0/02:58:e3:56:61:46 Cost = 6 Port
2	0.191715215	10.0.0.2	10.0.0.16	ICMP	98	Echo (ping) request id=0xbcd0, seq=1/256, ttl=64 (req
3	0.192337951	10.0.0.16	10.0.0.2	ICMP	98	Echo (ping) reply id=0xbcd0, seq=1/256, ttl=64 (req
4	1.203678611	10.0.0.2	10.0.0.16	ICMP	98	Echo (ping) request id=0xbcd0, seq=2/512, ttl=64 (req
5	1.203722431	10.0.0.16	10.0.0.2	ICMP	98	Echo (ping) reply id=0xbcd0, seq=2/512, ttl=64 (req
6	2.003050714	2a:26:fb:0a:a6:6a	Spanning-tree-(for-...	STP	52	Conf. Root = 32768/0/02:58:e3:56:61:46 Cost = 6 Port
7	2.227961857	10.0.0.2	10.0.0.16	ICMP	98	Echo (ping) request id=0xbcd0, seq=3/768, ttl=64 (req
8	2.228062443	10.0.0.16	10.0.0.2	ICMP	98	Echo (ping) reply id=0xbcd0, seq=3/768, ttl=64 (req
9	3.999897067	2a:26:fb:0a:a6:6a	Spanning-tree-(for-...	STP	52	Conf. Root = 32768/0/02:58:e3:56:61:46 Cost = 6 Port

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。

```
mininet> pingall
*** Ping: testing ping reachability
h11 -> h12 h13 h14
h12 -> h11 h13 h14
h13 -> h11 h12 h14
h14 -> h11 h12 h13
h21 -> h11 h12 h13
h22 -> h11 h12 h13
h23 -> h11 h12 h13
h24 -> h11 h12 h13
h31 -> h11 h12 h13
h32 -> h11 h12 h13
h33 -> h11 h12 h13
h34 -> h11 h12 h13
h41 -> h11 h12 h13
h42 -> h11 h12 h13
h43 -> h11 h12 h13
h44 -> h11 h12 h13
*** Results: 0% dropped
mininet> xterm h11
mininet> 
```

"Node: h11"

```
root@ubuntu:/home/sdn/Desktop/mininet/custom# ifconfig
h11-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::1012:aeff:fe41:c418 prefixlen 64 scopeid 0x20<link>
    ether 12:12:ae:41:c4:18 txqueuelen 1000 (Ethernet)
    RX packets 373 bytes 25760 (25.7 KB)
    RX errors 0 dropped 54 overruns 0 frame 0
    TX packets 70 bytes 4996 (4.9 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 (Local Loopback)
    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

root@ubuntu:/home/sdn/Desktop/mininet/custom# 
```

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。

"Node: h12"

```
root@ubuntu:/home/sdn/Desktop/mininet/custom# ifconfig
h12-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.0.2 netmask 255.0.0.0 broadcast 10.255.255.255
    inet6 fe80::2868:95ff:fe4f:5aa4 prefixlen 64 scopeid 0x20<link>
    ether 2a:68:95:4f:5a:a4 txqueuelen 1000 (Ethernet)
    RX packets 1067 bytes 73215 (73.2 KB)
    RX errors 0 dropped 334 overruns 0 frame 0
    TX packets 244 bytes 17428 (17.4 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 (Local Loopback)
    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

root@ubuntu:/home/sdn/Desktop/mininet/custom# 
```

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。

```
"Node: AcS11" (root)
2 0 d6:aa:ce:f8:ba:10 3
2 0 da:46:fa:96:0d:8e 3
2 0 6e:f9:27:9e:dc:91 3
root@ubuntu:/home/sdn/Desktop/mininet/custom# sudo ovs-appctl fdb/show AcS11
port VLAN MAC Age
2 0 ba:bb:ea:5e:dd:59 197
2 0 86:c4:7a:ac:b8:c3 197
2 0 86:7b:d2:aa:dc:46 195
2 0 96:75:57:c3:08:99 195
2 0 2a:78:a0:a2:10:a0 195
2 0 f6:de:cc:18:ff:4c 195
2 0 ce:51:9e:46:74:56 195
2 0 ca:9b:45:b2:12:7f 194
2 0 32:b9:fe:42:c9:b2 194
2 0 76:f1:8f:2e:88:39 194
2 0 06:c7:ee:dd:1f:26 194
2 0 8e:c0:69:5d:75:b3 193
2 0 b6:35:ee:f8:90:c7 193
2 0 46:2d:17:1c:a3:a2 193
2 0 7e:90:78:02:1c:06 191
2 0 2a:69:8e:b6:44:e3 191
2 0 1a:0a:24:df:db:52 187
2 0 ae:ab:c1:29:83:d2 187
2 0 0e:ef:ab:f7:19:45 187
```

```
"Node: AcS11" (root)
2 0 be:cb:8f:6b:1a:8b 171
2 0 2a:79:1e:de:f8:a5 171
2 0 72:07:24:68:3f:e8 138
2 0 2e:36:4e:36:6a:d3 138
2 0 5e:69:e8:05:39:5a 7
2 0 6a:bb:cb:24:f6:49 7
2 0 da:13:c6:8a:2e:1d 7
4 0 2a:68:95:4f:5a:a4 3
3 0 12:12:ae:41:c4:18 3
2 0 c2:13:f3:b0:dc:da 3
2 0 6e:f9:27:9e:dc:91 3
2 0 1a:46:7f:ad:27:13 3
2 0 1e:b5:4a:9a:2a:07 3
2 0 ae:87:6b:e8:a3:5d 3
2 0 6e:e8:6f:35:ee:df 3
2 0 66:84:0b:2f:42:26 3
2 0 da:46:fa:96:0d:8e 3
2 0 86:a3:93:32:84:b9 3
2 0 d6:aa:ce:f8:ba:10 3
2 0 6e:f5:5e:ca:e5:4e 3
2 0 2e:8f:d1:36:be:4e 3
2 0 42:5b:ea:25:c0:d4 3
2 0 16:32:0c:b8:4e:bb 3
root@ubuntu:/home/sdn/Desktop/mininet/custom#
```

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。

```

*** Ping: testing
h11 -> h12 h13 h14
h12 -> h11 h13 h14
h13 -> h11 h12 h14
h14 -> h11 h12 h13
h21 -> h11 h12 h13
h22 -> h11 h12 h13
h23 -> h11 h12 h13
h24 -> h11 h12 h13
h31 -> h11 h12 h13
h32 -> h11 h12 h13
h33 -> h11 h12 h13
h34 -> h11 h12 h13
h41 -> h11 h12 h13
h42 -> h11 h12 h13
h43 -> h11 h12 h13
h44 -> h11 h12 h13
*** Results: 0% dr
mininet> xterm h44
mininet>

```

"Node: h44"

```

root@ubuntu:/home/sdn/Desktop/mininet/custom# ifconfig
h44-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::6cf9:27ff:fe9e:dc91 prefixlen 64 scopeid 0x20<link>
    ether 6e:f9:27:9e:dc:91 txqueuelen 1000 (Ethernet)
    RX packets 1497 bytes 100398 (100.3 KB)
    RX errors 0 dropped 567 overruns 0 frame 0
    TX packets 364 bytes 25828 (25.8 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 (Local Loopback)
    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

root@ubuntu:/home/sdn/Desktop/mininet/custom#

```

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。

"Node: AcS11" (root)

```

root@ubuntu:/home/sdn/Desktop/mininet/custom# sudo ovs-appctl fdb/show AcS11
port  VLAN  MAC  Age
 2      0  0a:fe:c9:47:d8:93  272
 2      0  02:20:4f:90:2d:87  272
 2      0  ca:f7:54:30:3c:58  272
 2      0  be:cb:8f:6b:1a:8b  272
 2      0  72:07:24:68:3f:e8  174
 2      0  2e:36:4e:36:6a:d3  141
 2      0  16:32:0c:b8:4e:bb   4
 3      0  12:12:ae:41:c4:18   4
 2      0  66:84:0b:2f:42:26   4
 4      0  2a:68:95:4f:5a:a4   4
 2      0  ae:87:6b:e8:a3:5d   4
 2      0  6e:e8:6f:35:ee:df   4
 2      0  42:5b:ea:25:c0:d4   4
 2      0  1e:b5:4a:9a:2a:07   4
 2      0  6e:f9:27:9e:dc:91   4
 2      0  6e:f5:5e:ca:e5:4e   4
 2      0  86:a3:93:32:84:b9   4
 2      0  1a:46:7f:ad:27:13   4
 2      0  c2:13:f3:b0:dc:da   4
 2      0  d6:aa:ce:f8:ba:10   4
 2      0  da:46:fa:96:0d:8e   4
 2      0  2e:8f:d1:36:be:4e   4

root@ubuntu:/home/sdn/Desktop/mininet/custom#

```

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。

```
AcS11-eth2: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::4481:f5ff:fe1e:8313 prefixlen 64 scopeid 0x20<link>
    ether 46:81:f5:1e:83:13 txqueuelen 1000 (Ethernet)
    RX packets 4105 bytes 256424 (256.4 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1271 bytes 90944 (90.9 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

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。

```
"Node: AcS11" (root)
root@ubuntu:/home/sdn/Desktop/mininet/custom# sudo ovs-appctl fdb/show DstS12
port  VLAN  MAC                      Age
  1      0  0a:fe:c9:47:d8:93       177
  3      0  46:81:f5:1e:83:13       177
  1      0  0e:ef:ab:f7:19:45       177
  1      0  8e:c0:69:5d:75:b3        46
  1      0  2a:79:1e:de:f8:a5        46
  1      0  ca:f7:54:30:3c:58        46
  1      0  6e:f5:5e:ca:e5:4e        31
  3      0  2a:68:95:4f:5a:a4        31
  4      0  ae:87:6b:e8:a3:5d        31
  3      0  12:12:ae:41:c4:18        31
  1      0  2e:8f:d1:36:be:4e        31
  1      0  d6:aa:ce:f8:ba:10        31
  1      0  42:5b:ea:25:c0:d4        31
  4      0  1e:b5:4a:9a:2a:07        31
  1      0  6e:e8:6f:35:ee:df        31
  1      0  da:46:fa:96:0d:8e        31
  1      0  6e:f9:27:9e:dc:91        31
  1      0  1a:46:7f:ad:27:13        31
  1      0  86:a3:93:32:84:b9        31
  1      0  66:84:0b:2f:42:26        31
  1      0  16:32:0c:b8:4e:bb        31
  1      0  c2:13:f3:b0:dc:da        31
root@ubuntu:/home/sdn/Desktop/mininet/custom#
```

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。

```
DstS12-eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::9422:b2ff:fe3d:753c prefixlen 64 scopeid 0x20<link>
    ether 96:22:b2:3d:75:3c txqueuelen 1000 (Ethernet)
    RX packets 4878 bytes 311637 (311.6 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 2171 bytes 155442 (155.4 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

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。


```

root@ubuntu:/home/sdn/Desktop/mininet/custom# sudo ovs-appctl fdb/show CoreS3
port  VLAN  MAC  Age
1      0      1e:b5:4a:9a:2a:07  27
4      0      da:46:fa:96:0d:8e  27
3      0      86:a3:93:32:84:b9  27
1      0      2a:68:95:4f:5a:a4  27
4      0      6e:f9:27:9e:dc:91  27
3      0      d6:aa:ce:f8:ba:10  27
1      0      ae:87:6b:e8:a3:5d  27
4      0      c2:13:f3:b0:dc:da  27
1      0      12:12:ae:41:c4:18  27
4      0      1a:46:7f:ad:27:13  27
3      0      6e:f5:5e:ca:e5:4e  27
3      0      2e:8f:d1:36:be:4e  27
3      0      16:32:0c:b8:4e:bb  27
3      0      66:84:0b:2f:42:26  27
3      0      6e:e8:6f:35:ee:df  27
3      0      42:5b:ea:25:c0:d4  27

```

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。

```

root@ubuntu:/home/sdn/Desktop/mininet/custom# sudo ovs-appctl fdb/show DstS42
port  VLAN  MAC  Age
3      0      da:46:fa:96:0d:8e  158
4      0      6e:f9:27:9e:dc:91  158
3      0      c2:13:f3:b0:dc:da  158
4      0      1a:46:7f:ad:27:13  158
1      0      ae:87:6b:e8:a3:5d  158
1      0      12:12:ae:41:c4:18  158
1      0      1e:b5:4a:9a:2a:07  158
1      0      2a:68:95:4f:5a:a4  158
1      0      16:32:0c:b8:4e:bb  158
1      0      66:84:0b:2f:42:26  158
1      0      86:a3:93:32:84:b9  158
1      0      d6:aa:ce:f8:ba:10  158
1      0      6e:f5:5e:ca:e5:4e  158
1      0      2e:8f:d1:36:be:4e  158
1      0      42:5b:ea:25:c0:d4  158
1      0      6e:e8:6f:35:ee:df  158

```

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。

```

root@ubuntu:/home/sdn/Desktop/mininet/custom# sudo ovs-appctl fdb/show AcS42
port  VLAN  MAC  Age
3      0      1a:46:7f:ad:27:13  198
2      0      ae:87:6b:e8:a3:5d  198
2      0      12:12:ae:41:c4:18  198
2      0      1e:b5:4a:9a:2a:07  198
2      0      2a:68:95:4f:5a:a4  198
2      0      66:84:0b:2f:42:26  198
2      0      da:46:fa:96:0d:8e  198
2      0      c2:13:f3:b0:dc:da  198
2      0      86:a3:93:32:84:b9  198
2      0      d6:aa:ce:f8:ba:10  198
2      0      6e:f5:5e:ca:e5:4e  198
2      0      2e:8f:d1:36:be:4e  198
2      0      42:5b:ea:25:c0:d4  198
2      0      16:32:0c:b8:4e:bb  198
2      0      6e:e8:6f:35:ee:df  198
4      0      6e:f9:27:9e:dc:91  197
root@ubuntu:/home/sdn/Desktop/mininet/custom#

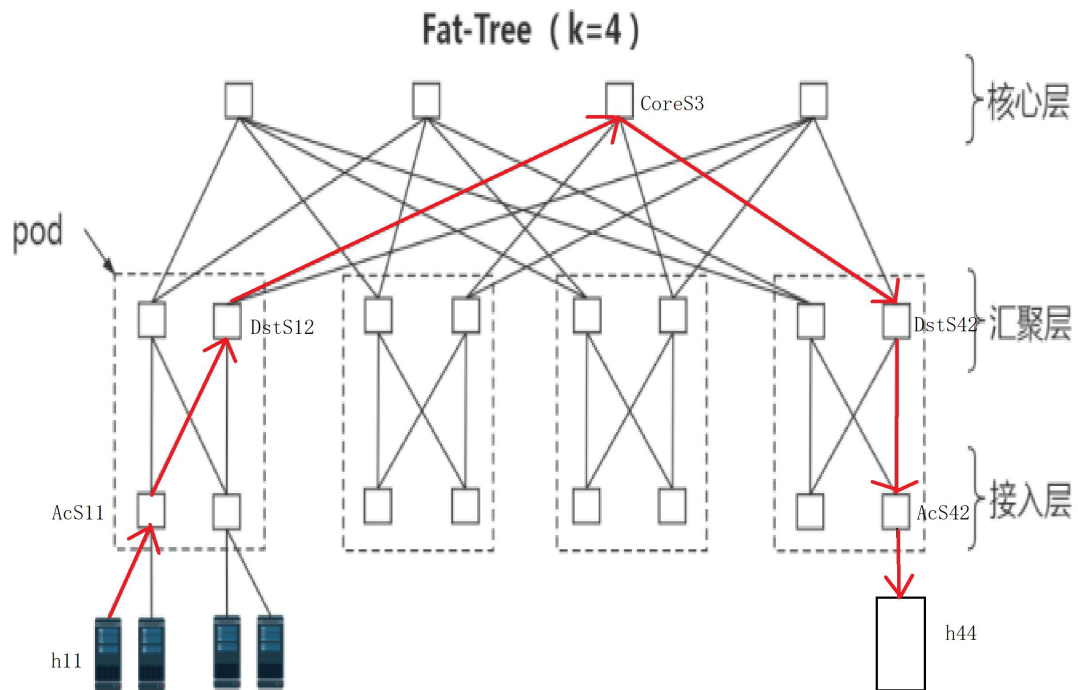
```


可以看到，h11 主机通过 Eth2 端口与 AcS11 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。