**华中科技大学计算机学院**

**《计算机通信与网络》实验报告**

班级 本硕博2301 姓名 王家乐 学号 U202315763

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 项目 | Socket编程  （30%） | 数据可靠传输协议设计  （20%） | SDN实验  （30%） | 平时成绩(20%) | 总分 |
| 得分 |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 课程目标 | 1  （20%） | 2  （15%） | 3  （15%） | 4  （15%） | 5  （20%） | 6  （15%） | 总分 |
| 得分 |  |  |  |  |  |  |  |

教师评语：

教师签名：

给分日期：

**目 录**

[第一章 实验环境安装与使用 3](#_Toc19772)

[1.1 环境 3](#_Toc9683)

[1.2 实验任务 3](#_Toc27375)

[1.3 Mininet使用 3](#_Toc16741)

[1.4 Open vSwitch（OVS）开源多层虚拟交换机使用 4](#_Toc21844)

[1.5 Wireshark可视化 5](#_Toc31108)

[第二章 FAT TREE 搭建 6](#_Toc15148)

[2.1 环境 6](#_Toc16640)

[2.2 实验任务 6](#_Toc3791)

[2.3 FAT TREE 搭建 6](#_Toc2158)

[2.4 主机连通情况分析 8](#_Toc12184)

[2.5 其它需要说明的问题 10](#_Toc1653)

[2.6 参考资料 10](#_Toc17280)

[第三章 自学习与环路检测实验 11](#_Toc4969)

[3.1 环境 11](#_Toc27330)

[3.2 实验任务 11](#_Toc15083)

[3.3 自学习交换机 11](#_Toc1412)

[3.4 禁用端口解决环路广播 14](#_Toc31595)

[3.5 转发历史信息解决环路广播 18](#_Toc17363)

[3.6 其它需要说明的问题 20](#_Toc10931)

[第四章 链路选择与故障恢复实验 21](#_Toc14967)

[4.1 环境 21](#_Toc28656)

[4.2 实验任务 21](#_Toc19303)

[4.3 最小跳数路径选择 22](#_Toc1742)

[4.4 最小延迟路径选择 24](#_Toc26343)

[4.5 容忍链路故障 31](#_Toc23711)

[4.6 其它需要说明的问题 36](#_Toc4763)

[4.7 参考资料 36](#_Toc24495)

[心得体会与建议 37](#_Toc19977)

[5.1 心得体会 37](#_Toc23082)

[5.2 建议 37](#_Toc26727)

# 第一章 实验环境安装与使用

## **1.1 环境**

CPU：12th Gen Intel(R) Core(TM) i7-12700H

操作系统：Ubuntu 22.04

RAM大小：16GB

使用软件：VMware Workstation Pro

## **1.2 实验任务**

这一章的实验主要是完成虚拟机和 Mininet 仿真环境的搭建与启动，能够在提供好的 VMWare 虚拟机中导入镜像、登录系统并打开终端运行 Mininet；在此基础上，熟悉 mn 启动/清理、查看主机和链路、pingall 连通性测试等基本命令，学会用命令行参数或简单 Python 脚本创建小型拓扑，理解“搭一个小网络并验证是否互通”的基本流程；同时初步接触 Open vSwitch 的常用查看与配置命令，知道如何观察交换机转发表、基本理解数据转发过程，并学会在 Mininet 中调用 Wireshark 抓包，直观观察主机之间的通信，为后续更复杂的 SDN 控制与协议实验打下环境与工具使用基础。

## **1.3 Mininet使用**

使用sudo mn命令创建默认拓扑如图1.1；在minine命令行中查看网络节点、当前连接情况以及显示网络拓扑如图1.2。

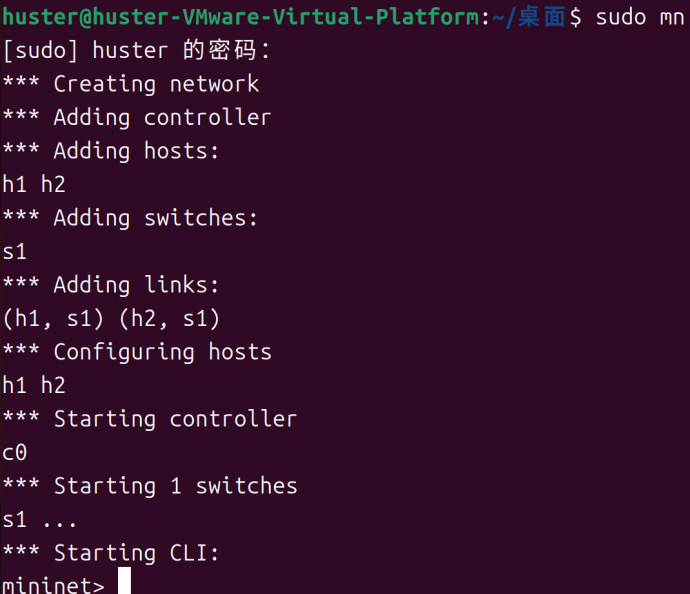
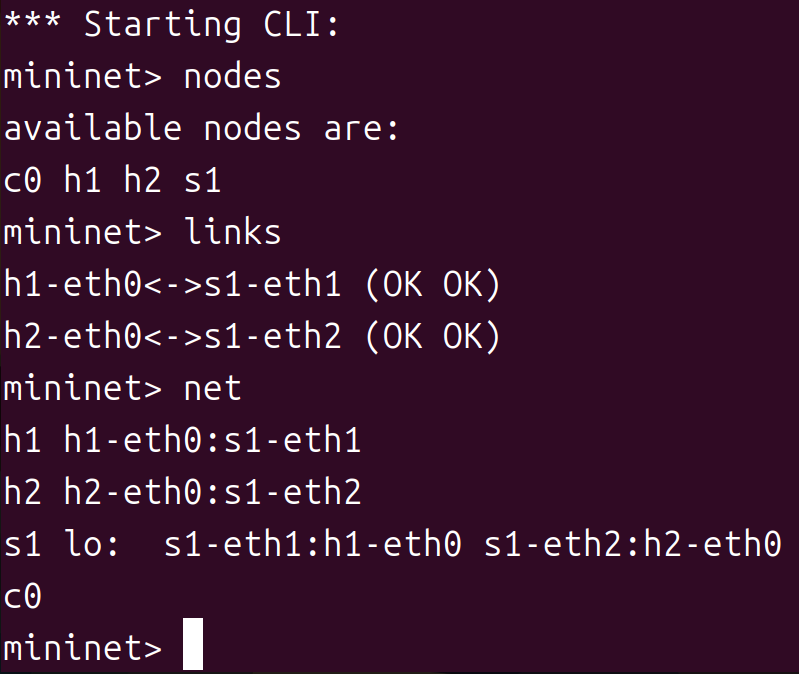
 

图1.1 创建默认拓扑 图1.2 查看节点、连接情况、拓扑

使用pingall命令和h1 ping -c3 h2命令测试网络通信如图1.3；输入exit退出mininet并使用sudo mn -c清理mininet如图1.4。

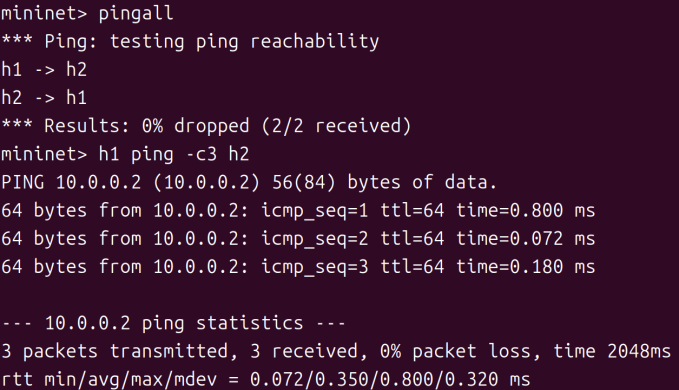
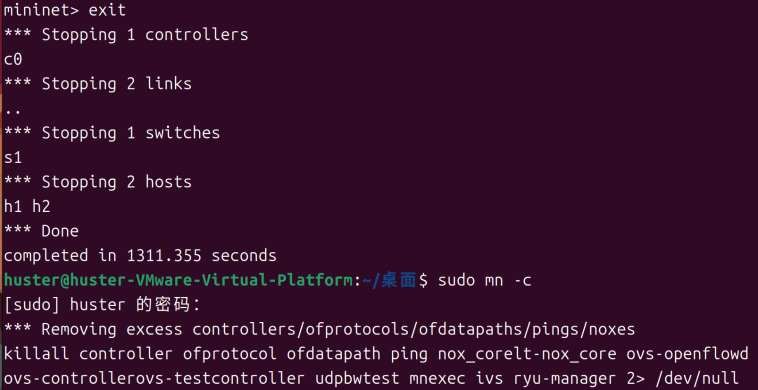
 

图1.3 测试网络通信 图1.4 退出mininet

## **1.4 Open vSwitch（OVS）开源多层虚拟交换机使用**

创建默认拓扑，打开新终端输入sudo ovs-vsctl show 命令查看交换机基本信息如图1.5。

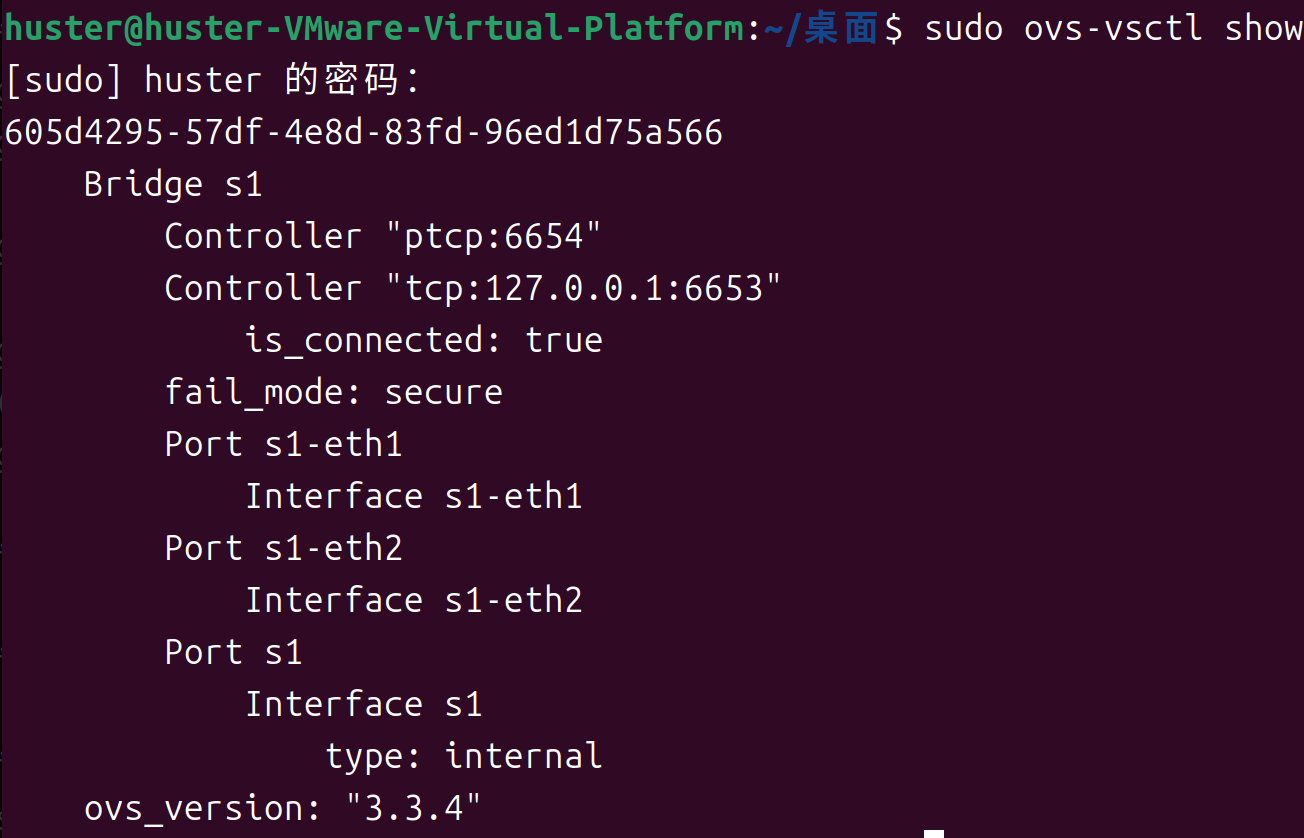


图1.5 交换机基本信息

接下来查看mac表。先使用sudo mn --mac --topo=tree,2,2 --controller=none创建一个拓扑如图1.6；对每个交换机执行sudo ovs-vsctl set-fail-mode xx standalone以及sudo ovs-vsctl set Bridge xx stp\_enable=true, 否则mac表学习不到东西，如图1.7；pingall令所有主机发送数据包，防止沉默主机现象，如图1.8；sudo ovs-appctl fdb/show xx查看mac表如图1.9。

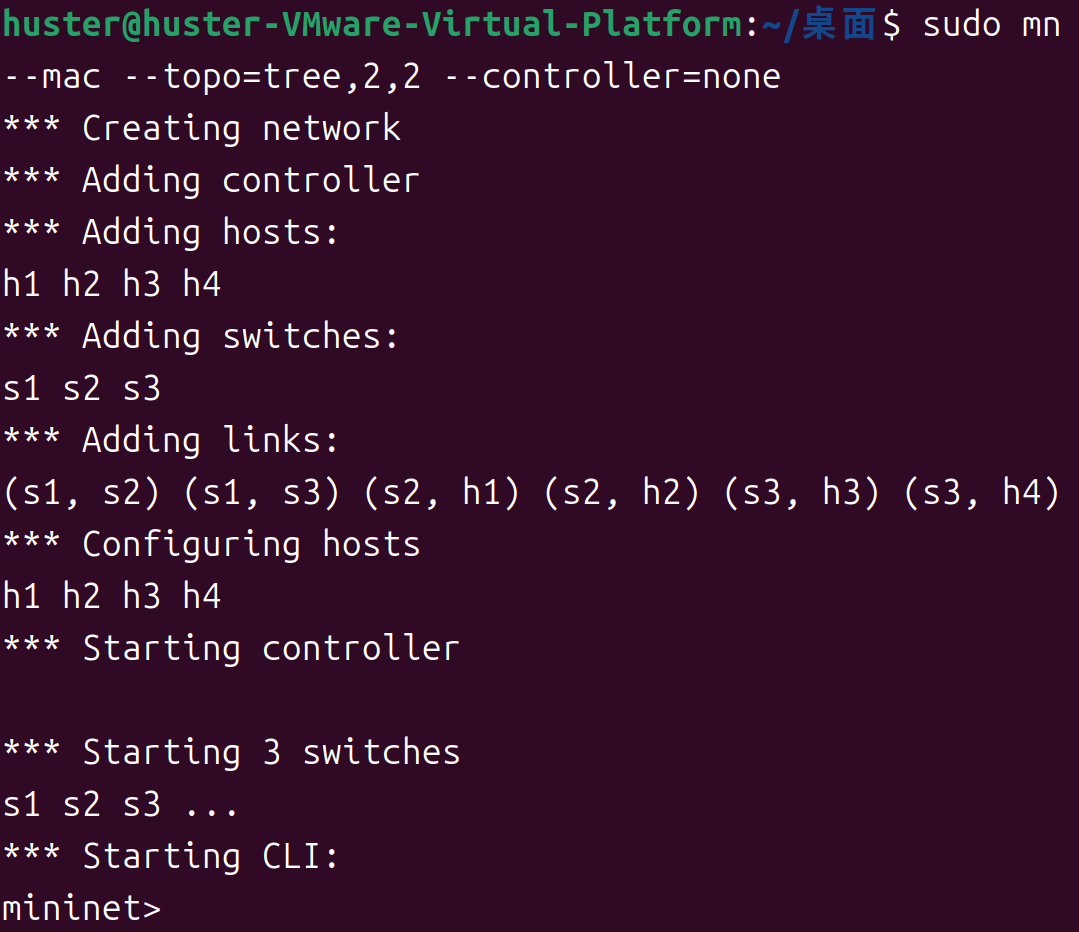


图1.6 创建一个topo

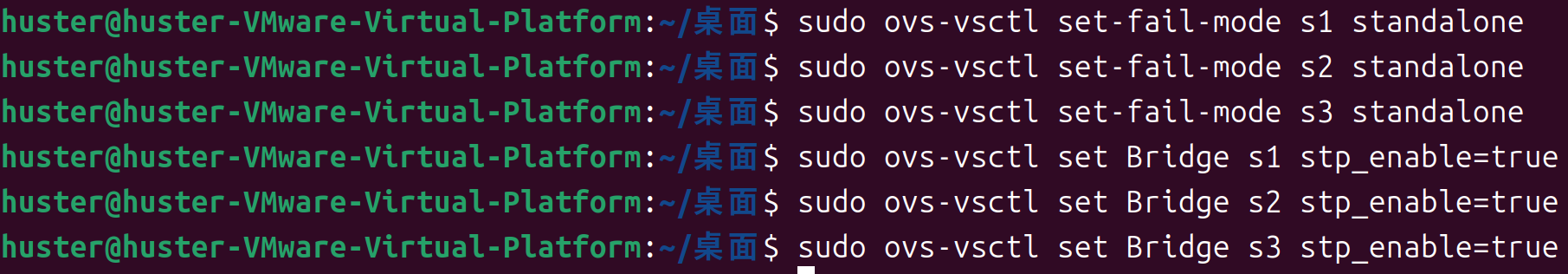


图1.7 配置交换机standalone模式和STP

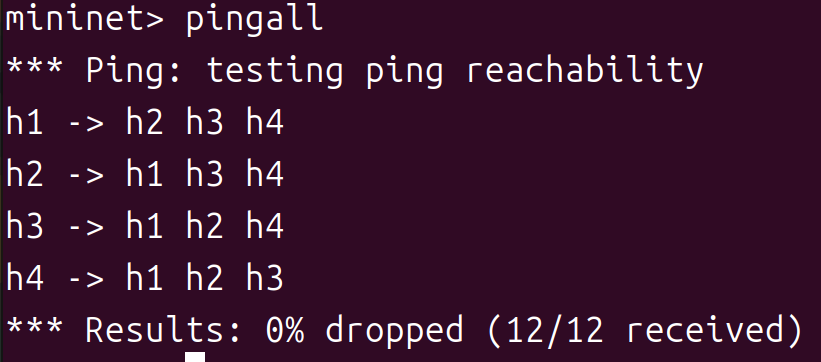


图1.8 pingall

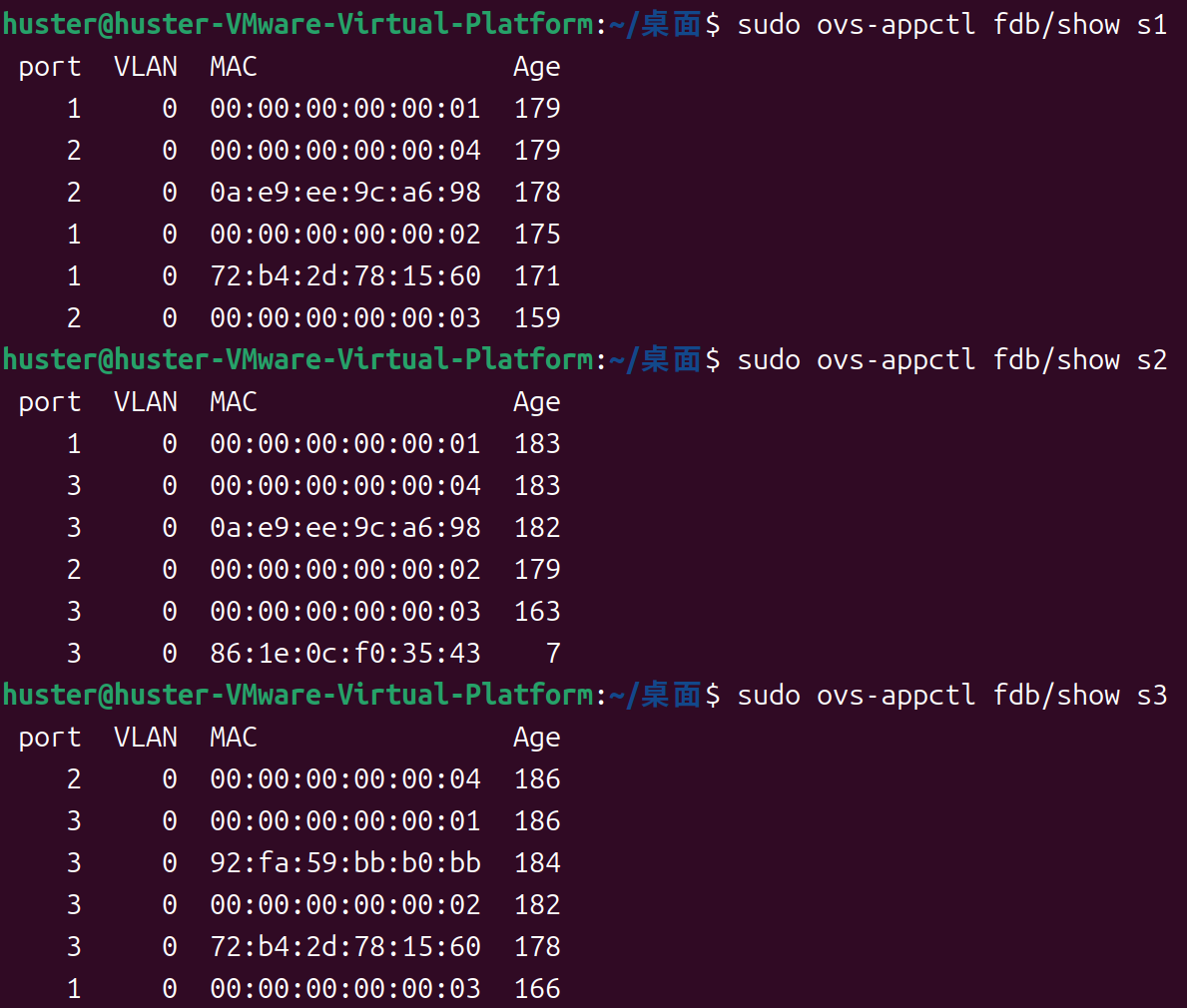


图1.9 查看mac表

## **1.5 Wireshark可视化**

创建默认topo，在mininet CLI中执行h1 wireshark，抓取h1的包如图1.10。

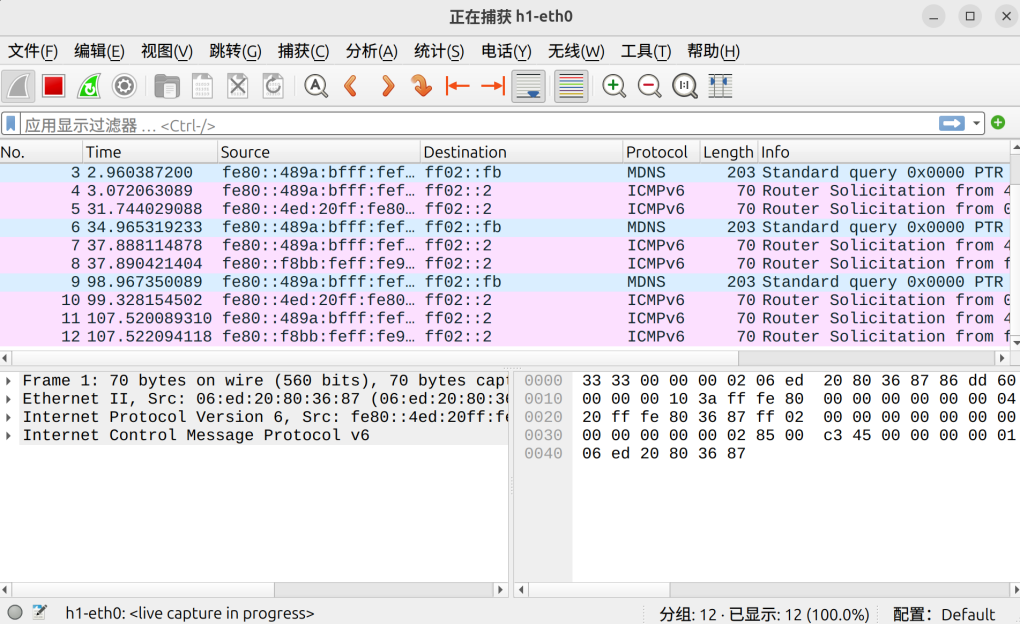


图1.10 h1抓包信息

# 第二章 FAT TREE 搭建

## **2.1 环境**

CPU：12th Gen Intel(R) Core(TM) i7-12700H

操作系统：Ubuntu 22.04

RAM大小：16GB

使用软件：VMware Workstation Pro

## **2.2 实验任务**

使用 Mininet 的Python API搭建k=4的FatTree拓扑。使用pingall查看各主机之间的连通情况，若主机之间未连通，分析原因并解决，若主机连通，分析数据包的路径。

## **2.3 FAT TREE 搭建**

图2.1所示示为一个参数K=4的FatTree：

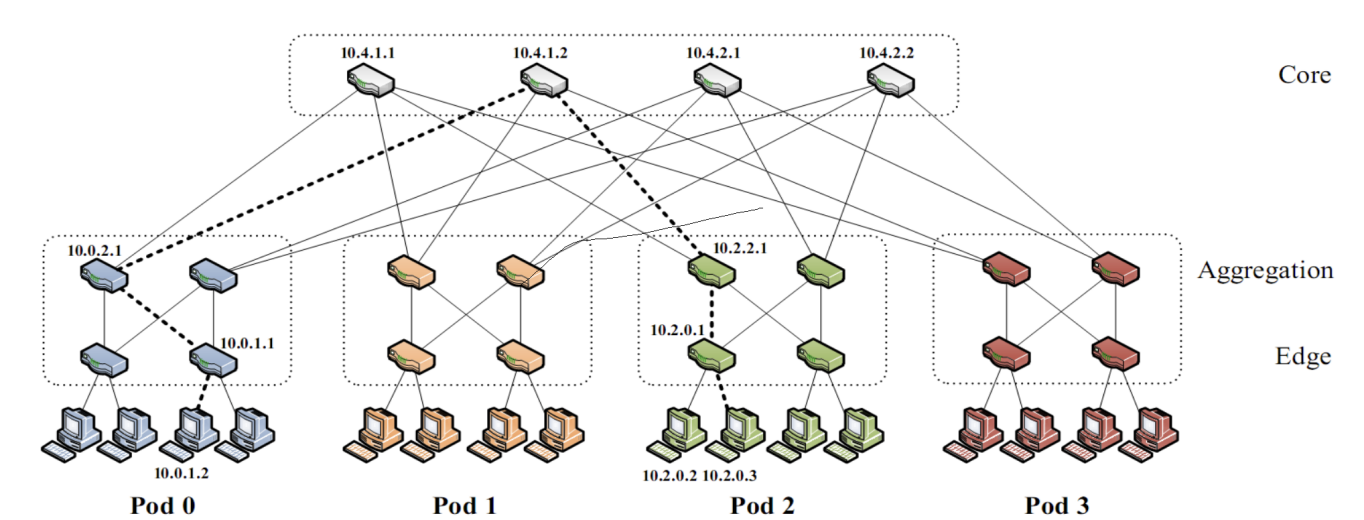


图2.1 k=4的FatTree

参数为K的FatTreeFatTree 网络架构由多个 Pod 组成，每个 Pod 包含两层交换机，每层有 k/2 个交换机。边缘层（Edge）交换机的每个 Switch 有 K 个输出端口，其中前 k/2 个端口按顺序连接到主机，而后 k/2 个端口则连接到聚合层（Aggregation）的交换机。聚合层的交换机也有 K 个输出端口，前 k/2 个端口连接到边缘层的交换机，后 k/2 个端口则连接到核心层（Core）的交换机，且每个交换机从左到右按优先级分配 k/2 个连接到核心层的 Switch。核心层由 (k/2)^2 个交换机组成，排列成二维阵列，核心交换机的端口通过特定编号顺序连接到各个 Pod，并在每个 Pod 内与聚合层交换机连接。每个 Pod 内的主机（Host）与对应的边缘层交换机相连接，完成数据的输入输出。

使用Mininet库定义了一个名为 [FatTreeTopo](vscode-file://vscode-app/c:/Users/Asus/AppData/Local/Programs/Microsoft VS Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") 的类，用于以编程方式构建一个k-ary FatTree 网络拓扑。[\_\_init\_\_](vscode-file://vscode-app/c:/Users/Asus/AppData/Local/Programs/Microsoft VS Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") 方法准备用于存储交换机和主机的列表。核心逻辑在 [build](vscode-file://vscode-app/c:/Users/Asus/AppData/Local/Programs/Microsoft VS Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ")方法中，它首先根据 [k](vscode-file://vscode-app/c:/Users/Asus/AppData/Local/Programs/Microsoft VS Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") 值创建相应数量的核心层、汇聚层和边缘层交换机，并将主机连接到边缘交换机上。随后，它调用[connect\_core\_to\_aggregation](vscode-file://vscode-app/c:/Users/Asus/AppData/Local/Programs/Microsoft VS Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ")和[connect\_aggregation\_to\_edge](vscode-file://vscode-app/c:/Users/Asus/AppData/Local/Programs/Microsoft VS Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ")这两个辅助方法，分别建立核心层与汇聚层之间、以及同一Pod内汇聚层与边缘层之间的链路，从而完成整个 FatTree拓扑的构建。实现的核心代码如下：

class **FatTreeTopo**(**Topo**):

    def **\_\_init\_\_**(self, k=4, \*\*opts):

*# 先设置k值*

        self.k = k

        self.pod = k

*# 初始化列表*

        self.core\_switches = []

        self.aggregation\_switches = []

        self.edge\_switches = []

        self.host\_list = []

*# 主机计数器*

        self.host\_counter = 1

*# 然后调用父类的初始化*

**super**(**FatTreeTopo**, self).**\_\_init\_\_**(\*\*opts)

    def **build**(self):

        """构建FatTree拓扑 - 这个方法会被Mininet自动调用"""

*# 创建核心交换机 (k/2)^2 个 - 命名为 c1, c2, c3...*

        core\_count = (self.k // 2) \*\* 2

        for i in **range**(core\_count):

            sw = self.addSwitch(f'c{i+1}')

            self.core\_switches.**append**(sw)

*# 为每个pod创建聚合交换机和边缘交换机*

        for pod in **range**(self.pod):

*# 每个pod有k/2个聚合交换机 - 命名为 a11, a12, a21, a22...*

            for agg in **range**(self.k // 2):

                agg\_sw = self.addSwitch(f'a{pod+1}{agg+1}')

                self.aggregation\_switches.**append**(agg\_sw)

*# 每个pod有k/2个边缘交换机 - 命名为 e11, e12, e21, e22...*

            for edge in **range**(self.k // 2):

                edge\_sw = self.addSwitch(f'e{pod+1}{edge+1}')

                self.edge\_switches.**append**(edge\_sw)

*# 每个边缘交换机连接k/2个主机 - 命名为 h1, h2, h3...*

                for host\_num in **range**(self.k // 2):

                    host = self.addHost(f'h{self.host\_counter}')

                    self.host\_list.**append**(host)

                    self.addLink(edge\_sw, host)

                    self.host\_counter += 1

*# 连接核心层和聚合层*

        self.**connect\_core\_to\_aggregation**()

*# 连接聚合层和边缘层（在同一个pod内）*

        self.**connect\_aggregation\_to\_edge**()

    def **connect\_core\_to\_aggregation**(self):

        """连接核心交换机和聚合交换机"""

        core\_per\_group = self.k // 2

        for core\_index, core\_sw in **enumerate**(self.core\_switches):

            core\_group = core\_index // core\_per\_group  *# 核心交换机所属的组*

            for pod in **range**(self.pod):

*# 每个核心交换机连接到每个pod中的特定聚合交换机*

                agg\_index\_in\_pod = pod \* (self.k // 2) + core\_group

                if agg\_index\_in\_pod < **len**(self.aggregation\_switches):

                    agg\_sw = self.aggregation\_switches[agg\_index\_in\_pod]

                    self.addLink(core\_sw, agg\_sw)

    def **connect\_aggregation\_to\_edge**(self):

        """连接聚合交换机和边缘交换机（在同一个pod内）"""

        switches\_per\_pod = self.k

        total\_pods = self.pod

        for pod in **range**(total\_pods):

*# 获取当前pod的所有聚合交换机*

            agg\_start = pod \* (self.k // 2)

            agg\_end = agg\_start + (self.k // 2)

            pod\_agg\_switches = self.aggregation\_switches[agg\_start:agg\_end]

*# 获取当前pod的所有边缘交换机*

            edge\_start = pod \* (self.k // 2)

            edge\_end = edge\_start + (self.k // 2)

            pod\_edge\_switches = self.edge\_switches[edge\_start:edge\_end]

*# 连接pod内的所有聚合交换机和所有边缘交换机*

            for agg\_sw in pod\_agg\_switches:

                for edge\_sw in pod\_edge\_switches:

                    self.addLink(agg\_sw, edge\_sw)

## **2.4 主机连通情况分析**

在本实验中创建FatTree时，由于没有使用控制器，如果仍保持默认的“secure/fail-secure”模式，Open vSwitch在没有流表下发的情况下会直接丢弃数据包，整个网络无法转发，因此需要把所有交换机的工作模式改成standalone，让它们像传统二层交换机一样启用自带的MAC 地址学习和基于端口的转发逻辑。另一方面，FatTree本身是一种高度冗余、存在大量环路的多根多层拓扑，如果只靠二层自学习而不做环路控制，那么广播帧和未知单播帧在环路中会被不断泛洪复制，迅速充满链路形成广播风暴，使网络陷入瘫痪。为此必须在每台交换机上启用生成树协议（STP），将部分冗余端口置为阻塞状态，只保留一条逻辑上的无环转发路径，从而既利用多交换机结构实现冗余备份。我们在创建FatTree的脚本中完成所有设置，如下：

def **run**():

    """启动FatTree拓扑"""

    topo = **FatTreeTopo**(k=4)

    net = Mininet(topo=topo, switch=OVSSwitch, controller=None, autoSetMacs=True)

    net.start()

*......*

    info('\*\*\* 配置交换机standalone模式和STP\n')

    for switch in net.switches:

*# Set fail-mode to standalone*

        switch.cmd('ovs-vsctl set-fail-mode', switch.name, 'standalone')

*# Enable Spanning Tree Protocol to prevent broadcast storms*

        switch.cmd('ovs-vsctl set Bridge', switch.name, 'stp\_enable=true')

        info(f"--- {switch.name:3}: standalone mode + STP enabled\n")

*......*

运行FatTree脚本文件并执行pingall测试，测试结果如图2.2所示。



图2.2 FatTree连通情况测试结果

使用links命令查看各端口连接情况，如下：

mininet> links

a11-eth3<->e11-eth3 (OK OK)

a11-eth4<->e12-eth3 (OK OK)

a12-eth3<->e11-eth4 (OK OK)

a12-eth4<->e12-eth4 (OK OK)

a21-eth3<->e21-eth3 (OK OK)

a21-eth4<->e22-eth3 (OK OK)

a22-eth3<->e21-eth4 (OK OK)

a22-eth4<->e22-eth4 (OK OK)

a31-eth3<->e31-eth3 (OK OK)

a31-eth4<->e32-eth3 (OK OK)

a32-eth3<->e31-eth4 (OK OK)

a32-eth4<->e32-eth4 (OK OK)

a41-eth3<->e41-eth3 (OK OK)

a41-eth4<->e42-eth3 (OK OK)

a42-eth3<->e41-eth4 (OK OK)

a42-eth4<->e42-eth4 (OK OK)

c1-eth1<->a11-eth1 (OK OK)

c1-eth2<->a21-eth1 (OK OK)

c1-eth3<->a31-eth1 (OK OK)

......

分析得到h1到h16的一条路径如图2.3：

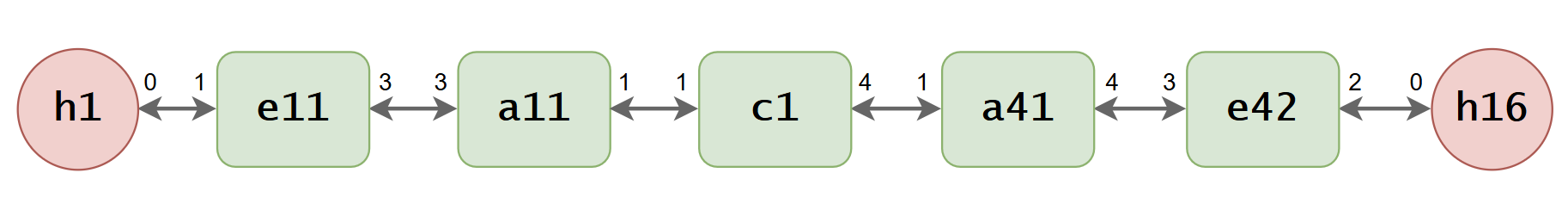


图2.3 h1到h16路径

## **2.5 其它需要说明的问题**

在本实验中，搭建FatTree拓扑的脚本通过参数k实现了一定的可扩展性，只需修改k值即可生成更大规模的FatTree，但在放大规模时需要同时考虑主机地址规划、链路数量以及物理机器性能，否则在执行pingall时可能出现超时或队列拥塞等问题；另外，本实验为了简化实现，未引入SDN控制器，而是依赖OVS的二层自学习和STP收敛，这意味着目前只能验证连通性与基本转发行为，尚无法体现FatTree在多路径负载均衡（如ECMP）等方面的优势，后续若要开展基于数据中心场景的路由或流量工程实验，可在本拓扑基础上接入OpenFlow控制器并下发精细化流表规则。

## **2.6 参考资料**

Mininet 官网：<http://mininet.org>

OVS参考网站：<http://www.openvswitch.org>  
 FatTree：<https://zhuanlan.zhihu.com/p/360386839>

# 第三章 自学习与环路检测实验

## **3.1 环境**

CPU：12th Gen Intel(R) Core(TM) i7-12700H

操作系统：Ubuntu 22.04

RAM大小：16GB

使用软件：VMware Workstation Pro

## **3.2 实验任务**

第三部分“自学习与环路检测实验”的任务是在Mininet与SDN控制器环境中，围绕交换机自学习和二层环路广播抑制展开：首先在给定拓扑上运行控制器和虚拟网络，观察普通洪泛转发带来的冗余流量与“窃听”现象，然后在self\_learning\_switch.py等程序基础上实现基于 MAC地址的自学习交换机，使交换机能够自动记录主机所在端口并下发表项，从而将盲目洪泛优化为有目的的定向转发；再引入包含环路的拓扑，通过完善loop\_breaker\_switch.py，尝试在环路链路上选择性地禁用部分端口，以“人为断环”的方式消除广播风暴；进一步在loop\_detecting\_switch.py中设计利用报文转发历史检测环路的机制，使交换机在发现相同请求在不同端口反复出现时主动停止继续转发，实现不破坏拓扑结构的自动环路抑制，最后通过主机互ping、控制器日志和抓包结果综合验证和对比不同方案在连通性与广播控制方面的效果。

## **3.3 自学习交换机**

在常规二层以太网中，自学习机制由交换机内部固定的硬件或固件实现，控制与转发功能集中于同一设备：交换机在收到数据帧时，根据源MAC地址和入端口自动更新本地MAC地址表，再在本机表项中查找目的MAC决定转发端口，表项老化策略以及与生成树协议（STP/RSTP）等配合的环路控制算法均由各交换机独立、分布式地完成，控制范围主要局限于单设备视角。相比之下，SDN 网络下的自学习交换机将控制平面与数据平面解耦，交换机本身只负责按照控制器下发的流表进行匹配和转发，当缺乏匹配表项时才通过控制通道上报数据包，由集中式控制器上的应用（如自学习模块）统一维护主机位置及 MAC→端口等映射关系，并通过流表下发实现转发路径的更新。因此，SDN自学习不仅可以在全网视角下对多台交换机进行统一调度，而且学习的“粒度”也不再局限于MAC地址，能够按MAC、IP、端口号甚至更高层特征灵活定义流规则，实现更细致的策略控制与路径优化；但同时，其性能和鲁棒性更依赖控制器的可用性和控制链路时延，初始转发时需要经历一次控制器交互，而传统交换机的自学习则在本地硬件中完成，时延极低但可编程性和全局优化能力相对受限。

自学习交换机任务的网络拓扑图如图3.1，共有四个主机和四个交换机，h1连接到s1，h2连接到s2，h3连接到h3，h4连接到s4，s1与s2、s3、s4相连，说明s1作为核心交换机，所有通信都需要经过s1，该拓扑为星型结构。

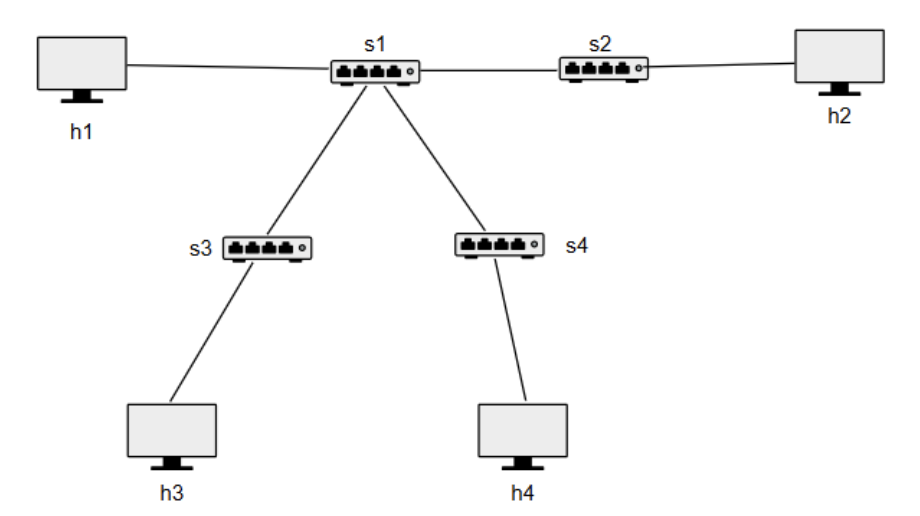


图3.1 topo\_1.py拓扑图

运行simple\_switch.py与 topo\_1.py，同时在mininet的CLI中启动wireshark对h3端口eth0 抓包，继续在mininet的CLI中，h4和h2进行通信，抓包结果如图3.2，说明控制器存在缺陷，packet\_in\_handler函数会将所有报文洪泛到交换机的所有端口，因此能在h3的eth0端口接收到10.0.0.2(h2)与10.0.0.4(h4)之间的通信。

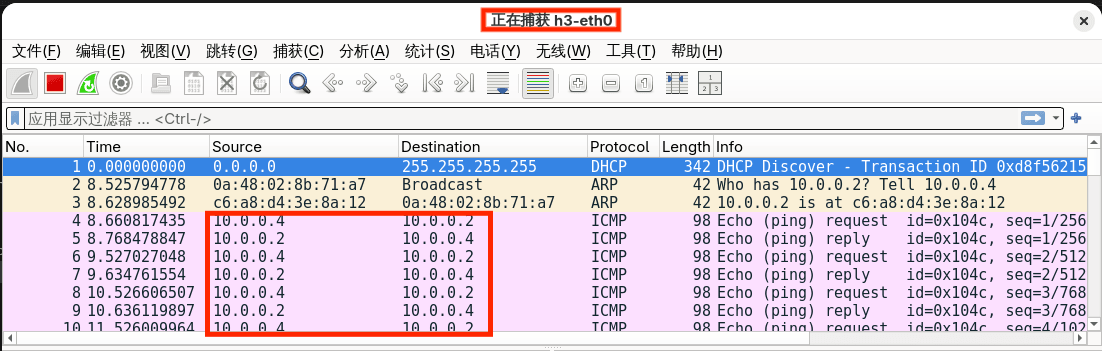


图3.2 h3抓包结果

在学习simple\_switch.py的基础上，完善self\_learning\_switch.py，通过控制平面下发流表完成交换机的自学习功能。

控制器需要维护一个mac\_to\_port映射表，用于记录每一个交换机对于mac地址的转发端口，若流表匹配，则交换机通过流表信息转发，无需控制器介入操作；若流表不匹配，交换机则会触发控制器的packet\_in\_handler函数，在packet\_in\_handler函数中，控制器先记录 (dpid,src\_mac)->in\_port映射，再查询映射表中是否存在 (dpid,dst\_mac) 的映射，若存在，向指定端口转发并下发流表，否则洪泛数据包。具体代码实现如下：

class **Switch**(**app\_manager**.**OSKenApp**):

    OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]

    def **\_\_init\_\_**(self, \*args, \*\*kwargs):

**super**(**Switch**, self).**\_\_init\_\_**(\*args, \*\*kwargs)

*# maybe you need a global data structure to save the mapping*

*# 维护 MAC 地址到端口的映射表: {dpid: {mac: port}}*

        self.mac\_to\_port = {}

*......*

    def **packet\_in\_handler**(self, ev):

*......*

*# 初始化该交换机的 MAC 表（如果还没有）*

        self.mac\_to\_port.**setdefault**(dpid, {})

*# 学习源 MAC 地址和入端口的映射*

        self.mac\_to\_port[dpid][src] = in\_port

*# 查询目的 MAC 地址是否已学习*

        if dst in self.mac\_to\_port[dpid]:

*# 映射表命中，获取输出端口*

            out\_port = self.mac\_to\_port[dpid][dst]

*# 打印五元组信息*

            self.logger.info("[Packet matched]\ndpid=%s, src=%s, in\_port=%s, dst=%s, out\_port=%s", dpid, src, in\_port, dst, out\_port)

*# 构造匹配规则和动作*

            match = parser.OFPMatch(in\_port=in\_port, eth\_dst=dst)

            actions = [parser.OFPActionOutput(out\_port)]

*# 下发流表（可以修改 hard\_timeout 参数来观察不同效果）*

            self.**add\_flow**(dp, 1, match, actions, hard\_timeout=0)

*# 转发当前数据包*

            out = parser.OFPPacketOut(

                datapath=dp,

                buffer\_id=msg.buffer\_id,

                in\_port=in\_port,

                actions=actions,

                data=msg.data

            )

            dp.send\_msg(out)

        else:

*# 映射表未命中，洪泛*

            actions = [parser.OFPActionOutput(ofp.OFPP\_FLOOD)]

            out = parser.OFPPacketOut(

                datapath=dp,

                buffer\_id=msg.buffer\_id,

                in\_port=in\_port,

                actions=actions,

                data=msg.data

            )

            dp.send\_msg(out)

再次测试并对h3抓包，h3只收到ARP广播包，如图3.3，不再收到h4和h2之间的ICMP包。控制器终端显示五元组信息：(dpid, src\_mac, in\_port, dst\_mac, out\_port)，如图3.4。

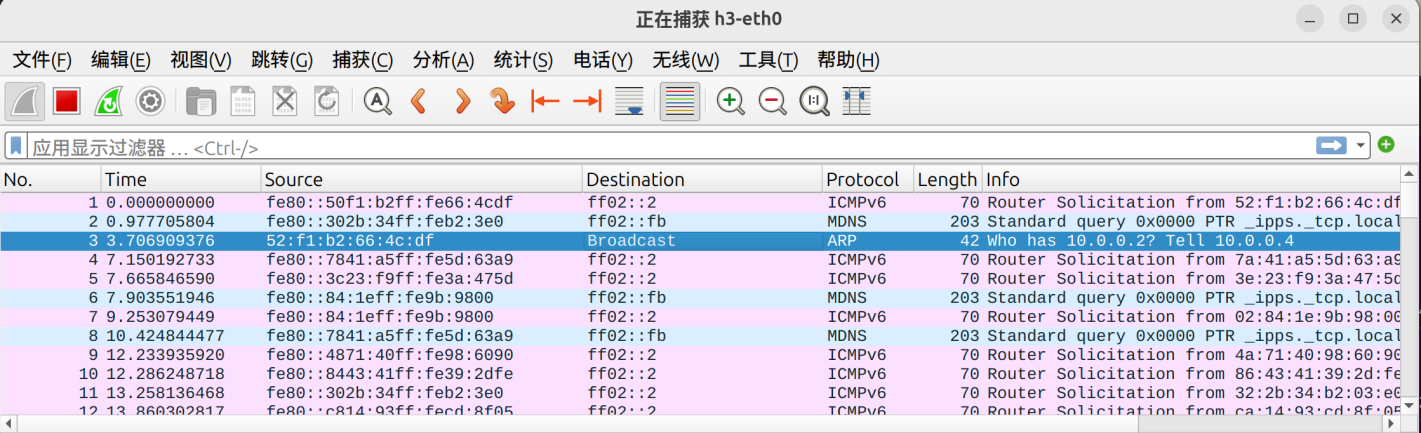


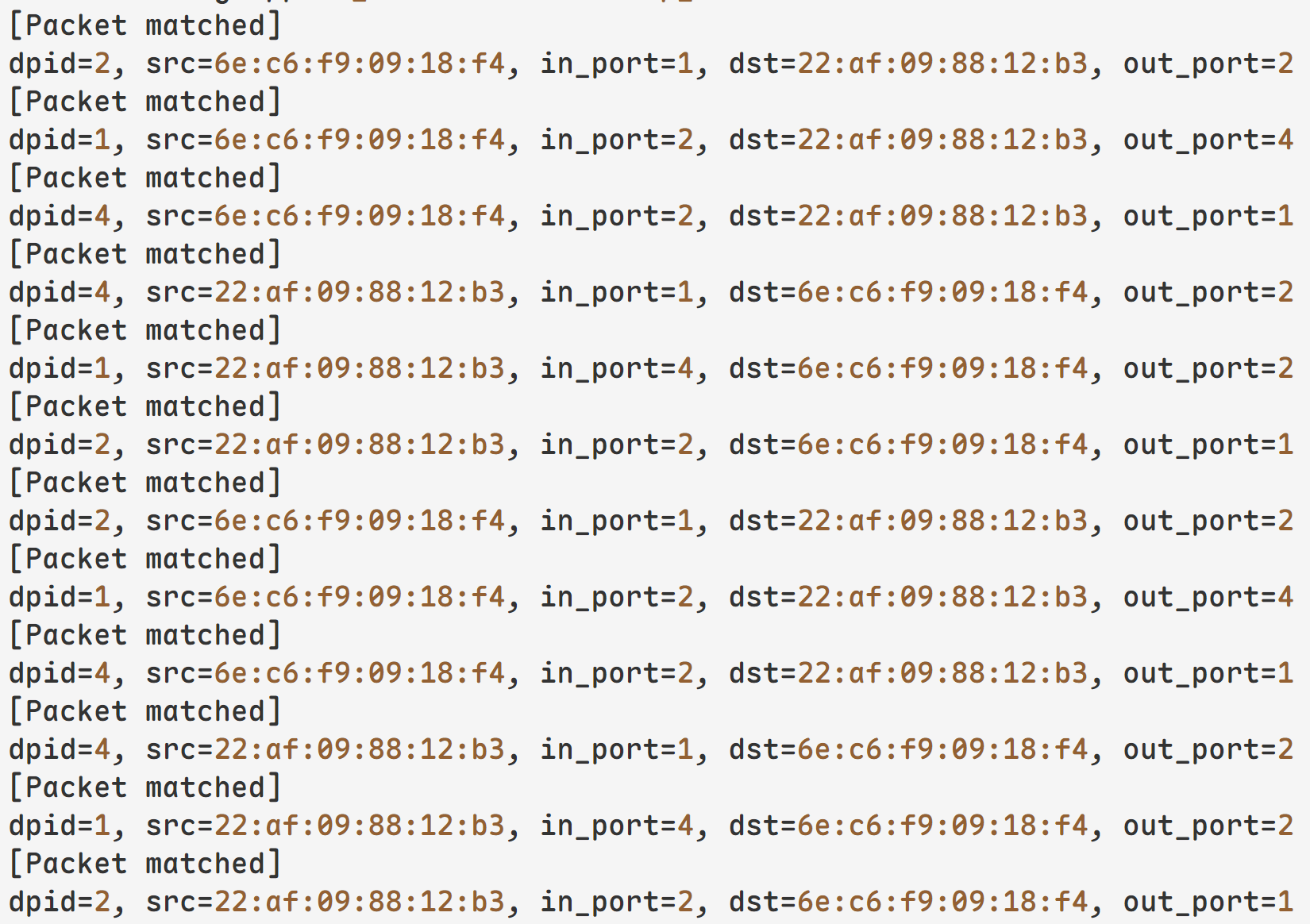
图3.3 h3抓包结果  


图3.4 控制器输出

观察可得h2的MAC：ca:14:93:cd:8f:05，h4 的MAC：52:f1:b2:66:4c:df，通信路径为:

h2 ⟺  (port\_1)s2(port\_2) ⟺  (port\_2)s1(port\_4) ⟺  (port\_2)s4(port\_1) ⟺  h4

改变hard\_timeout参数并执行通信测试。当hard\_timeout = 0时，观察到只有前几个包会触发控制器（学习阶段），这是因为流表永久有效，交换机直接转发，不经过控制器；当hard\_timeout = 5时，观察到每隔5秒，控制器会再次输出日志，这是因为流表5秒后失效，需要重新学习。

## **3.4 禁用端口解决环路广播**

topo\_2.py拓扑图如图3.5，s1、s3、s4之间形成了环路。

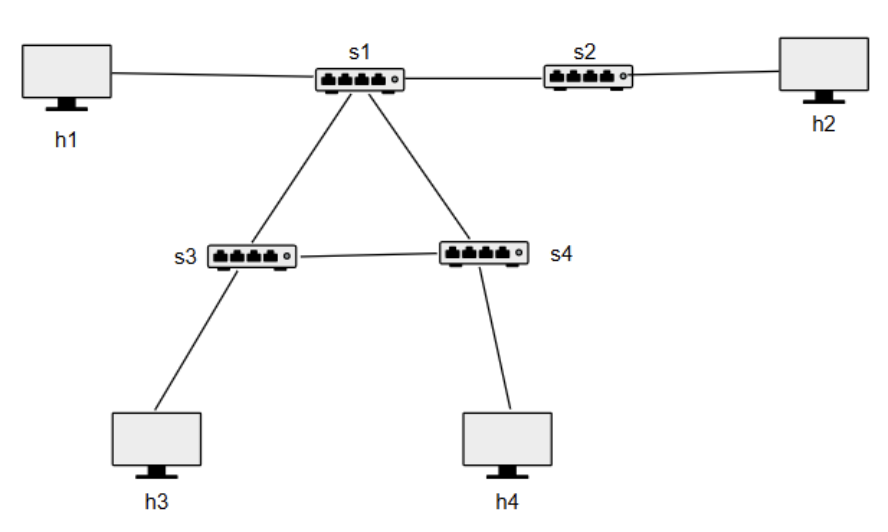


图3.5 topo\_2.py拓扑图

运行self\_learning\_switch.py和topo\_2.py ，发现h4与h2之间无法正常通信，如图3.6；观察流表n\_packets参数，发现网络内的数据包数目和流表匹配次数巨大，如图3.7；h3主机也收到了大量数据包，同时夹杂着大量的ARP Request包，如图3.8。

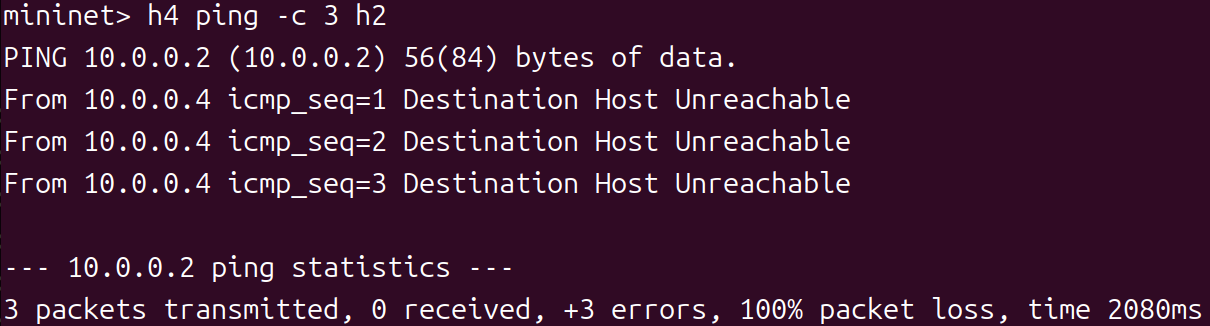


图3.6 h4与h2通信结果

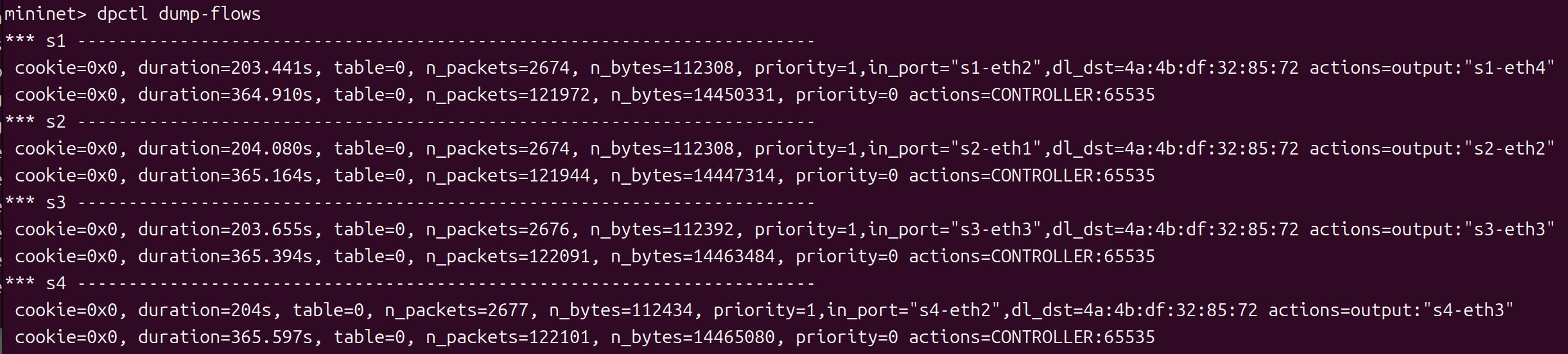


图3.7 流表信息

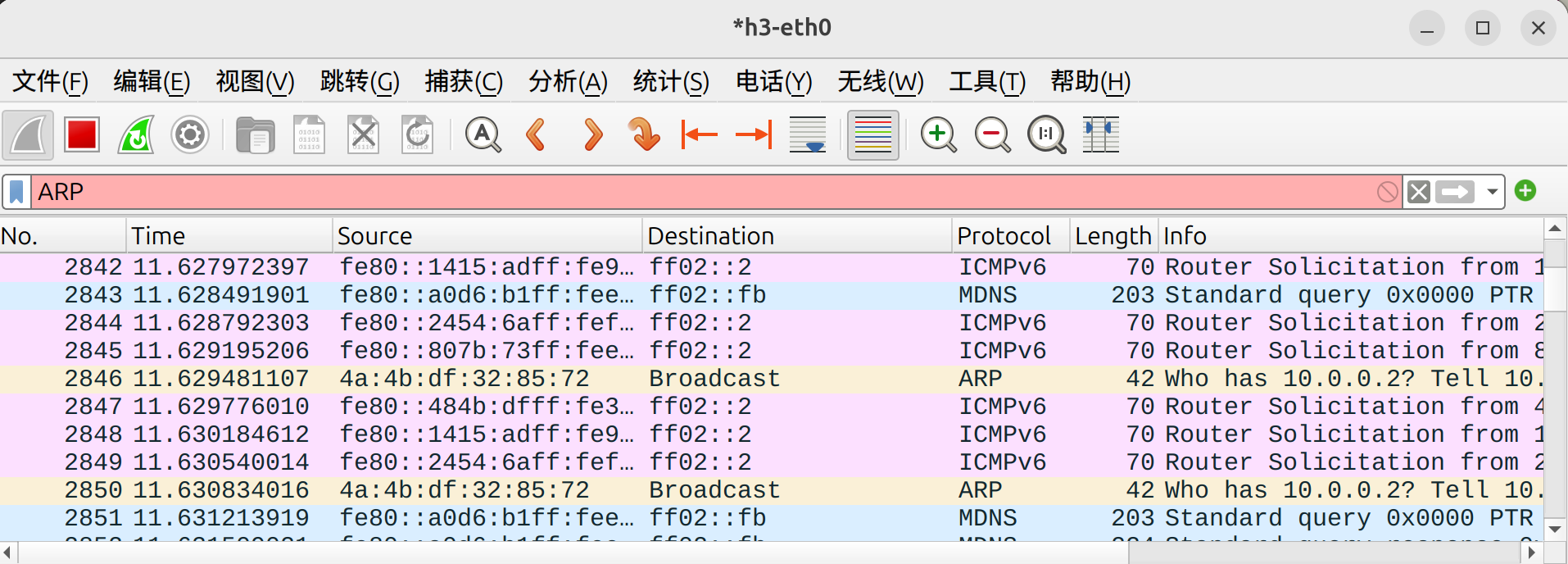


图3.8 h3抓包结果

采用禁用端口来解决环路广播，根据3.4节控制台输出结果判断，s1与s3相连的端口号为3，s1与s4相连的端口号为4。修改控制器loop\_breaker\_switch.py代码如下，禁用端口并确保学习和转发时跳过该端口。

通过静态的方式来解决网络环路问题。其核心逻辑是，当控制器第一次从指定的交换机（dpid == 1）收到数据包时，它会发送一个 OFPPortMod 消息来物理性地禁用该交换机上的一个预设端口（target\_port ），从而打破环路。为了确保正常的自学习转发，还增加了两个关键检查：一是忽略任何从被禁用端口收到的数据包；二是在查找转发出端口时，如果目标端口是被禁用的端口，则放弃单点转发，执行洪泛操作，以确保数据包能通过其他路径到达目的地。

class **Switch\_Dict**(**app\_manager**.**OSKenApp**):

    def **\_\_init\_\_**(self, \*args, \*\*kwargs):

**super**(**Switch\_Dict**, self).**\_\_init\_\_**(\*args, \*\*kwargs)

*......*

        self.flag = 0 *# only modify once*

        self.blocked = {}  *# {dpid: port\_no}*

def **packet\_in\_handler**(self, ev):

*......*

*# you need to code here to avoid broadcast loop to finish mission 2*

    if dpid == 1 and self.flag == 0:

*# 禁用端口号*

        target\_port = 3

        port = dp.ports.get(target\_port)

*# 构造 OFPPortMod 消息来禁用端口*

        port\_mod = parser.OFPPortMod(

            datapath=dp,

            port\_no=target\_port,

            hw\_addr=port.hw\_addr,

            config=ofp.OFPPC\_PORT\_DOWN,   *# 禁用端口*

            mask=ofp.OFPPC\_PORT\_DOWN,     *# 只修改 PORT\_DOWN 标志*

            advertise=0

        )

        dp.send\_msg(port\_mod)

*# 标记已修改，避免重复操作*

        self.flag = 1

        self.blocked[dpid] = target\_port

        self.logger.info("Port %s on switch %s has been disabled to break the loop", target\_port, dpid)

*# 如果这个包是从被禁用端口进来的，直接无视*

    if dpid in self.blocked and in\_port == self.blocked[dpid]:

        return

*......*

禁用s1与s3相连的端口，执行h4 ping h2。h4与h2之间通信正常，如图3.9；控制台输出如图3.10； h3只收到正常的ARP和 ICMP包，如图3.11。

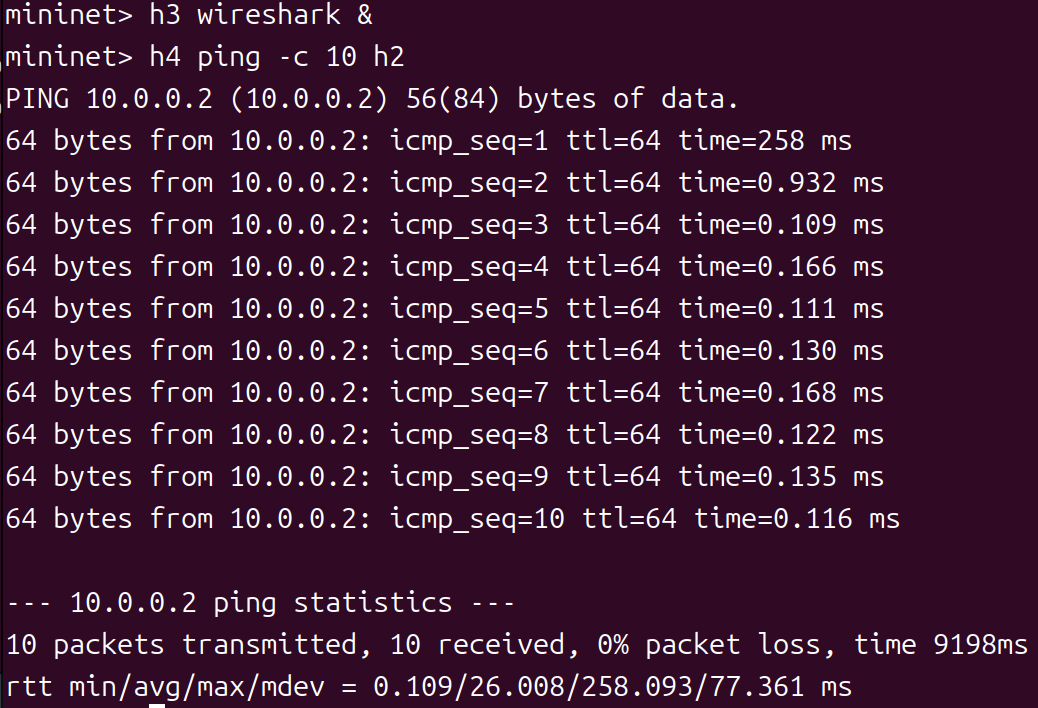


图3.9 h4与h2通信结果

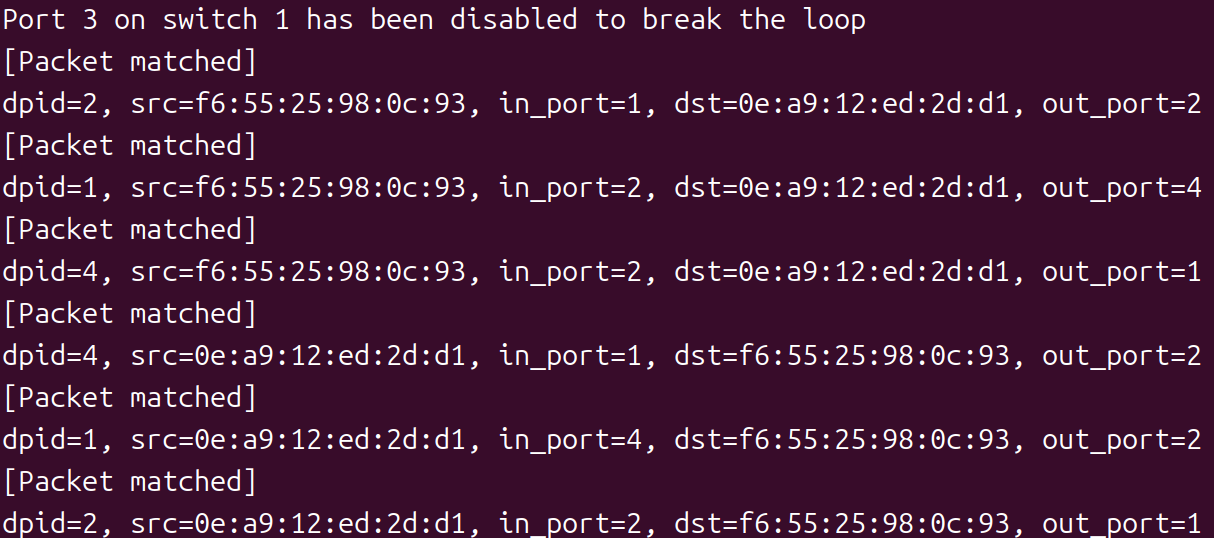


图3.10 控制台输出

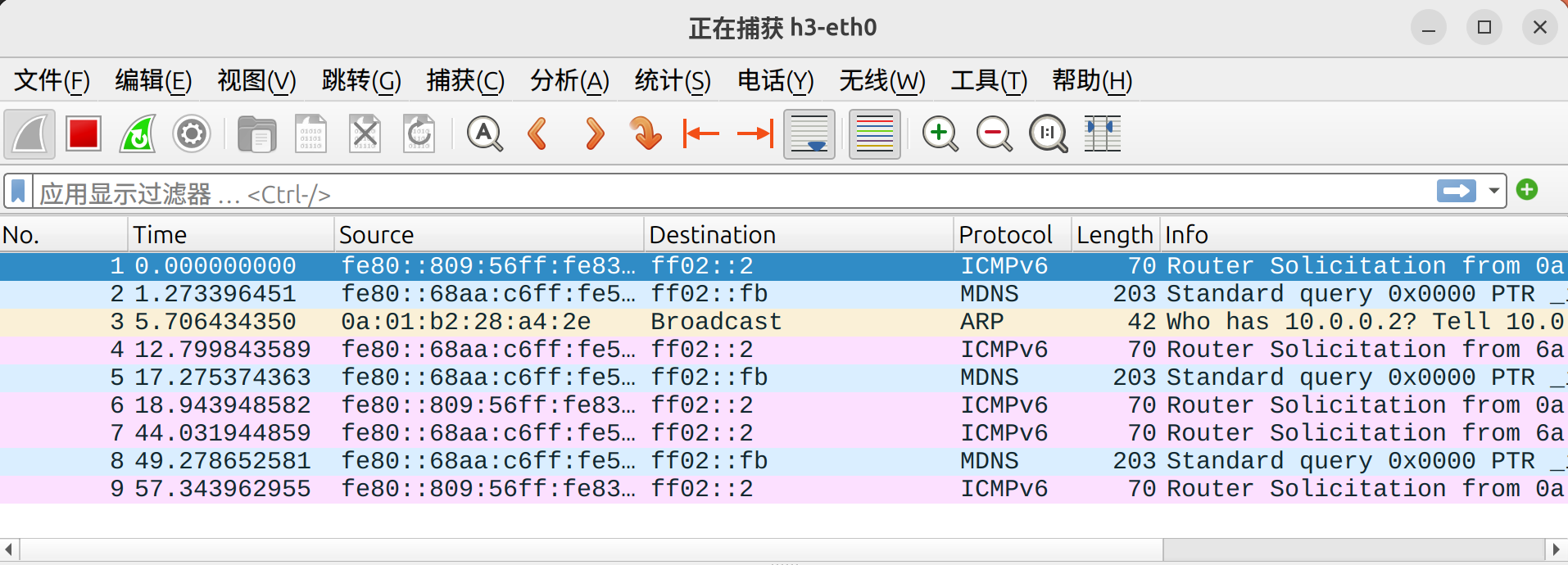


图3.11 h3抓包结果

禁用s1与s4相连的端口，执行h4 ping h2。h4与h2之间通信正常，如图3.12；控制台输出如图3.13； h3只收到正常的ARP和 ICMP包，如图3.14。

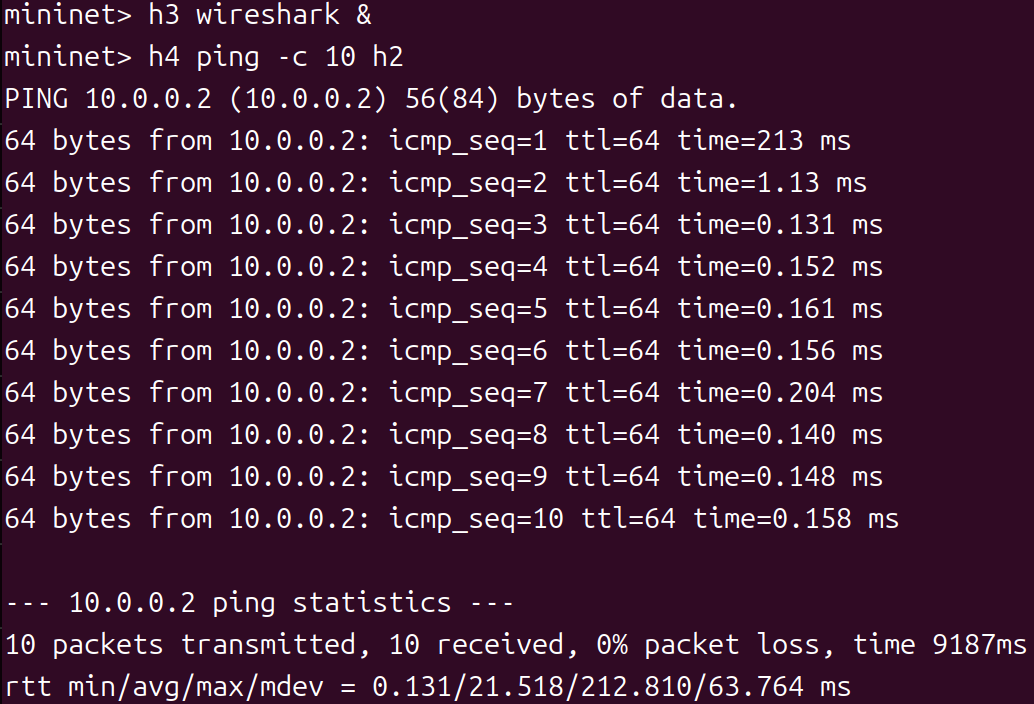


图3.12 h4与h2通信结果

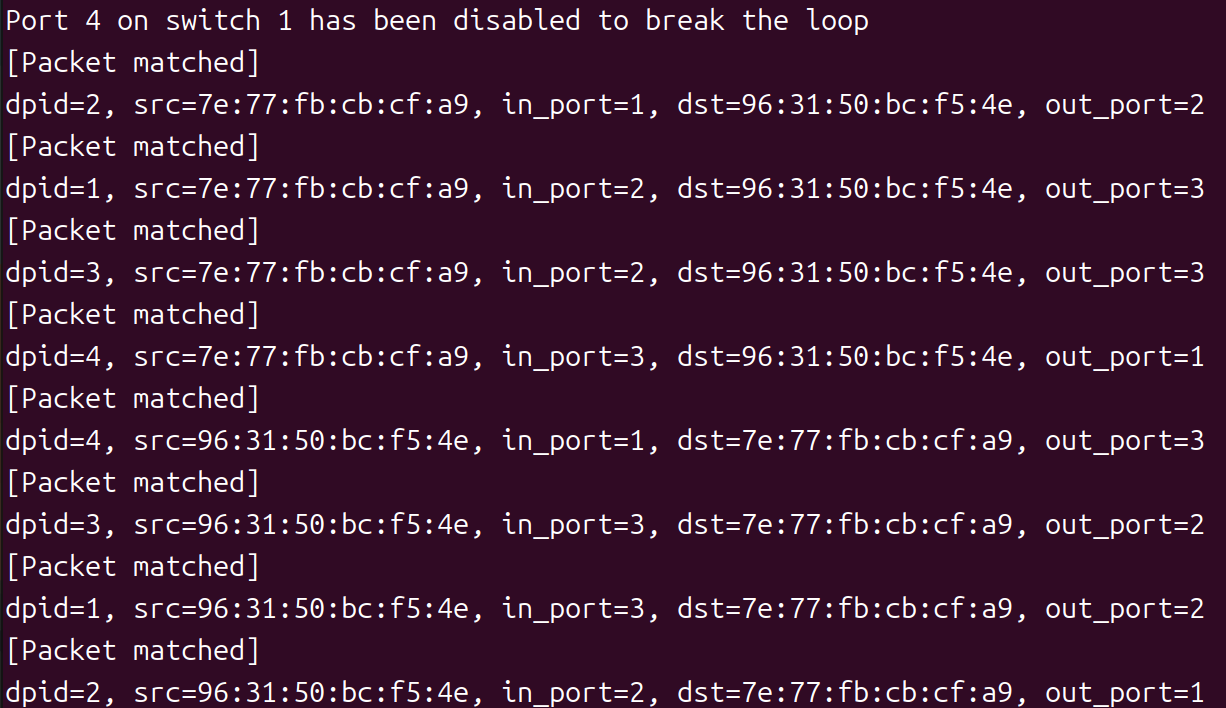


图3.13 控制台输出



图3.14 h3抓包结果

## **3.5 转发历史信息解决环路广播**

拓扑图同3.4，完善loop\_detecting\_switch.py，实现使用转发历史信息解决环路广播，具体方法如下：在控制器内维护一个映射表sw，用于记录ARP Request数据包在网络中的转发情况，具体结构为(dpid,src\_mac,dst\_ip) -> in\_port。当编号为dpid 的交换机从in\_port第一次收到src\_mac主机发出的询问dst\_ip的ARP Request数据包时，控制器根据sw 结构记录下映射。下一次该交换机收到同一(src\_mac,dst\_ip) 但in\_port不同ARP Request 数据包时(此时认为已经产生环路)，不执行任何转发操作，具体代码如下：

*# 检测并防止 ARP 广播环路*

if dst == ETHERNET\_MULTICAST and ARP in header\_list:

*# you need to code here to avoid broadcast loop to finish mission 2*

    arp\_pkt = header\_list[ARP]

    arp\_dst\_ip = arp\_pkt.dst\_ip

*# 构造映射键：(dpid, src\_mac, dst\_ip)*

    key = (dpid, src, arp\_dst\_ip)

*# 检查是否已经记录过这个 ARP Request*

    if key in self.sw:

*# 如果之前记录的端口与当前端口不同，说明产生了环路*

        if self.sw[key] != in\_port:

*# 不执行任何转发操作，直接返回*

            return

        else:

*# 第一次收到这个 ARP Request，记录下来*

            self.sw[key] = in\_port

执行h4 ping h2。h4与h2之间通信正常，如图3.15；控制台输出如图3.16。

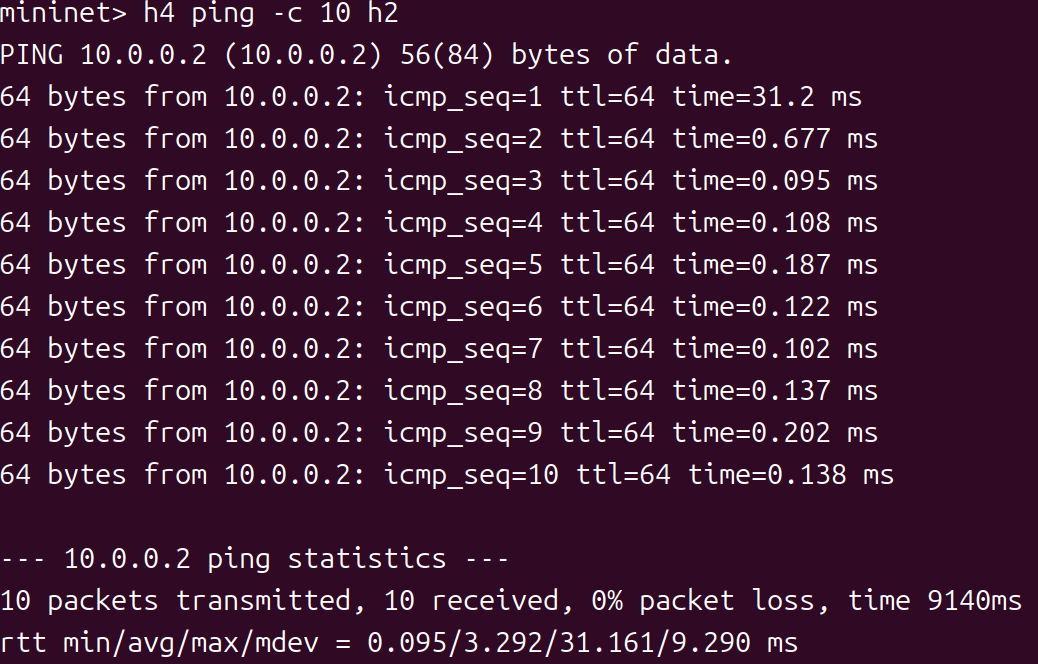


图3.15 h4与h2通信结果

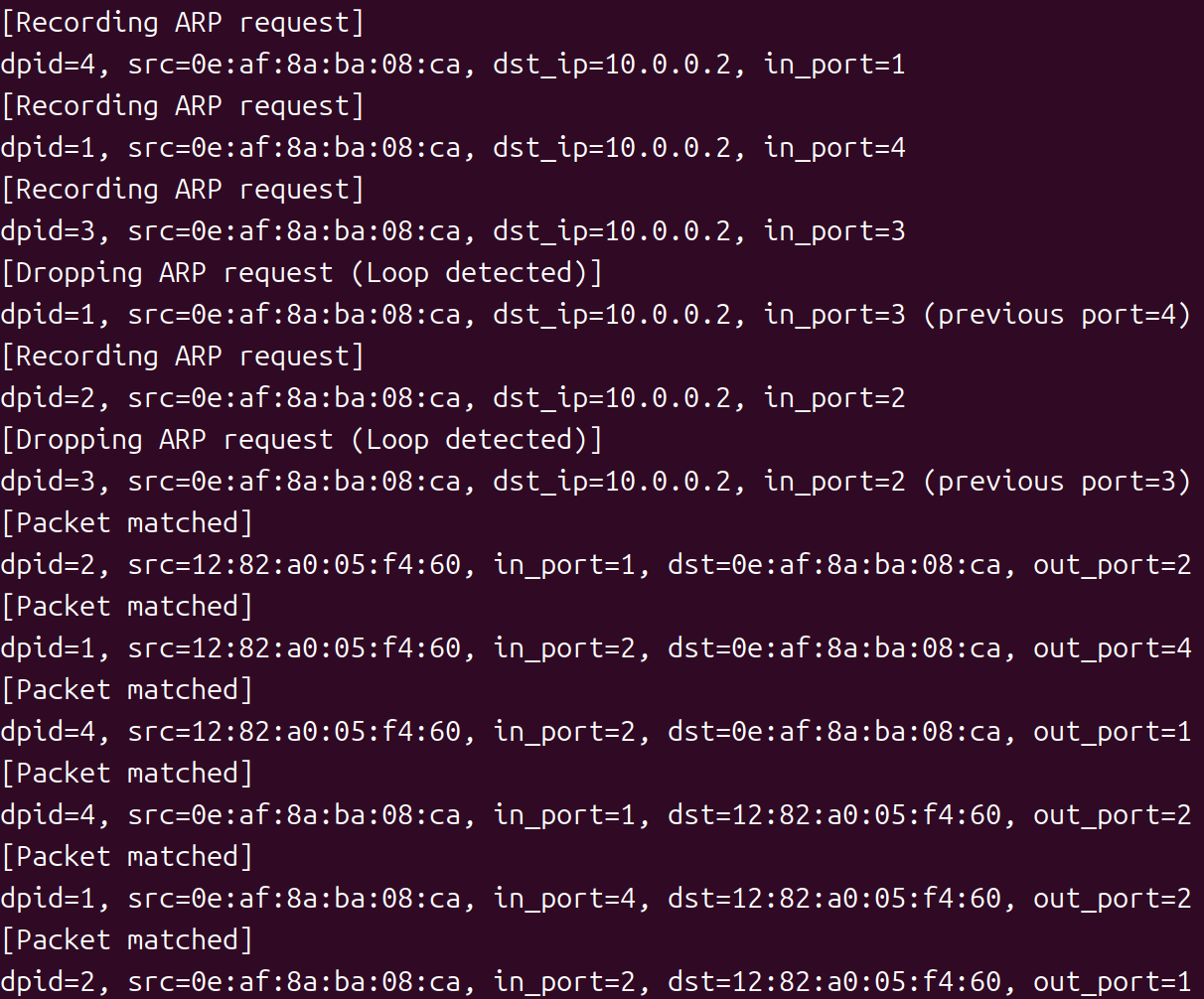


图3.16 控制台输出

控制台日志分析：

·第一次记录：s4从端口1收到h4发出的ARP Request

·第二次记录：s1从端口4收到相同的ARP Request

·第三次记录：s3从端口3收到相同的ARP Request

·环路检测：s1再次从端口3收到相同的ARP Request（说明环路）→ 丢弃！

举例说明，h4 (MAC: e6:71:b1:e3:2c:c9, IP: 10.0.0.4) 要ping h2 (IP: 10.0.0.2)：

·h4发出ARP Request：询问谁有10.0.0.2

·广播传播路径：

**h4 → s4 → s1 (端口4) → s3 (环路)**

**↓**

**s2 → h2 ✅**

·s1第一次从端口 4 收到 → 记录 (1, e6:71:b1:e3:2c:c9, 10.0.0.2) -> 4

·s1经过环路后从端口 3又收到同样的ARP → 检测到环路 → 丢弃！

## **3.6 其它需要说明的问题**

本实验中，自学习与环路检测方案虽然都能有效缓解广播风暴，但仍存在一些需要注意和改进的地方：一方面，基于MAC表的自学习在初始阶段必然经历一次洪泛过程，当网络规模增大或主机数量较多时，初始收敛期间的广播流量仍然可能对链路造成较大压力；另一方面，禁用端口的静态断环方法依赖于对拓扑结构和端口号的先验了解，一旦拓扑发生变动或交换机端口重新配置，就需要人工重新调整策略，扩展性和自适应性有限。相比之下，利用转发历史信息自动检测环路的方式具有一定的泛化能力，但其对状态表容量和匹配粒度有依赖：若键值设计过粗（例如仅使用源MAC），可能在复杂场景中出现误判或漏判；若设计过细或保存时间过长，又会增加控制器的存储和查表开销。实际工程中，往往需要将本实验中的思路与更成熟的协议机制结合起来，并对ARP广播、未知单播与多播流量分别制定更精细的管控策略，才能在保证连通性和收敛速度的同时，实现对环路和广播风暴的长期稳定抑制。

# 第四章 链路选择与故障恢复实验

## **4.1 环境**

CPU：12th Gen Intel(R) Core(TM) i7-12700H

操作系统：Ubuntu 22.04

RAM大小：16GB

使用软件：VMware Workstation Pro

Pyproject.toml：[project]

name = "lab3"

version = "0.1.0"

requires-python = ">=3.13"

dependencies = [

"mininet>=2.3.0.dev6",

"networkx>=3.4.2",

"os-ken>=3.0.1",

"tinyrpc>=1.1.7",

]

## **4.2 实验任务**

第四部分“链路选择与故障恢复实验”主要围绕在SDN环境下让控制器“聪明地选路并在链路出问题时自动恢复”为目标展开，具体包括：首先通过OS-Ken的拓扑感知模块和LLDP 链路发现机制获取交换机、主机和链路信息，在network\_awareness.py中用NetworkX构建拓扑图，并在任务一中利用networkx.shortest\_simple\_paths实现基于“最少跳数”的路径计算和流表下发；在任务二中进一步通过修改OS-Ken源码记录LLDP报文发送/接收时间与Echo往返时延，计算链路单向时延，将delay属性写入拓扑边权重，在新的控制器程序中实现基于“最小时延”的最优路径选择，打印h2到h9的最小时延路径和总时延并用ping RTT验证；在任务三中则在最小时延路径控制的基础上，监听端口状态变化事件（如EventOFPPortStatus），当通过Mininet模拟链路断开和恢复时，清空相关拓扑信息、转发表和自学习状态，强制交换机重新将报文上送控制器，从而重新根据当前链路时延计算并下发新的最优路径，实现对链路故障的容忍和自动收敛。

## **4.3 最小跳数路径选择**

网络拓扑及时延如图4.1所示，共有9个主机和9个交换机。



图4.1 网络拓扑及时延

networkx.shortest\_simple\_paths 的使用方法如下，src为源节点（IP 地址）、dst为目标节点（IP 地址）、weight为权重属性名称（'hop' 或 'delay'）、返回路径列表，按从小到大排序。

import **networkx** as **nx**

*# 创建拓扑图*

topo\_map = **nx**.**Graph**()

*# 添加边（hop=1 表示跳数权重）*

topo\_map.**add\_edge**(src, dst, hop=1, is\_host=False)

*# 计算最短路径（基于 hop 权重）*

paths = **list**(**nx**.**shortest\_simple\_paths**(topo\_map, src, dst, weight='hop'))

shortest\_path = paths[0]  *# 第一条路径即为最短路径*

解决ARP环路问题，在 least\_hops.py 的 handle\_arp( ) 方法中实现环路检测：

def **handle\_arp**(self, msg, in\_port, dst, src, pkt, pkt\_type):

    """

    使用 (dpid, src\_mac, dst\_mac) -> in\_port 的方法处理 ARP 环路

    """

    datapath = msg.datapath

    dpid = datapath.id

    parser = datapath.ofproto\_parser

    ofproto = datapath.ofproto

*# 构造唯一键*

    key = (dpid, src, dst)

*# 检测环路*

    if key in self.sw:

        if self.sw[key] != in\_port:

*# 环路！丢弃包*

            self.logger.info(

                "ARP loop detected: dpid=%s, src=%s, dst=%s, "

                "in\_port=%s (previous=%s)",

                dpid, src, dst, in\_port, self.sw[key]

            )

            return  *# 丢弃包*

    else:

*# 首次记录*

        self.sw[key] = in\_port

*# 洪泛 ARP*

    actions = [parser.OFPActionOutput(ofproto.OFPP\_FLOOD)]

    out = parser.OFPPacketOut(

        datapath=datapath,

        buffer\_id=msg.buffer\_id,

        in\_port=in\_port,

        actions=actions,

        data=msg.data

    )

    datapath.send\_msg(out)

通过OS-Ken控制器实现最小跳数路径选择。其工作流程如下：当网络中的主机首次通信时（例如主机A ping 主机B），主机A会发送ARP请求以获取主机B的MAC地址。该ARP包到达控制器后，[packet\_in\_handler](vscode-file://vscode-app/c:/Users/Asus/AppData/Local/Programs/Microsoft VS Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ")会调用[handle\_arp](vscode-file://vscode-app/c:/Users/Asus/AppData/Local/Programs/Microsoft VS Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ")方法。此方法首先通过记录(dpid, src\_mac, dst\_mac)与in\_port的映射来检测并丢弃环路中的ARP包，然后将非环路的ARP请求在网络中进行洪泛，以便主机B能收到并响应，从而完成MAC地址学习。随后，当主机A向主机B发送IP包时，该IP包同样会触发packet\_in\_handler，但这次会调用handle\_ipv4方法。此方法利用NetworkAwareness模块计算出的基于跳数（self.weight = 'hop'）的最短路径，然后为该路径上的每个交换机生成并下发精确匹配（源/目的IP、入端口）的流表规则，指导后续数据包沿此最小跳数路径转发，并最终将当前IP包从路径的最后一个交换机发送出去。

启动拓扑执行h2 ping h9，通信结果如图4.2，控制台输出如图4.3。

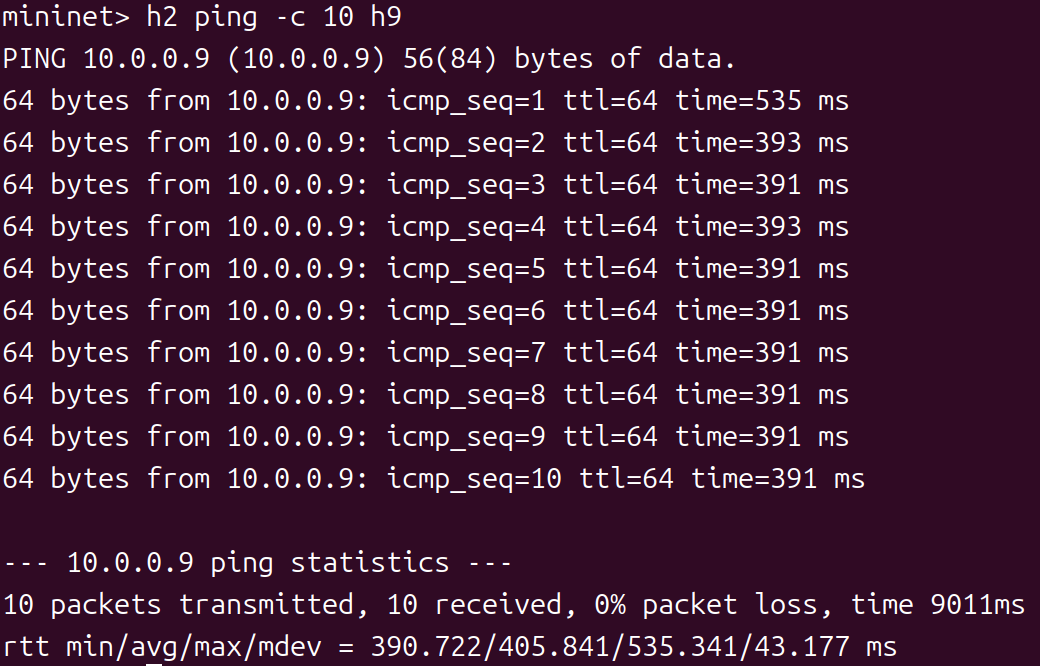


图4.2 h2与h9通信结果

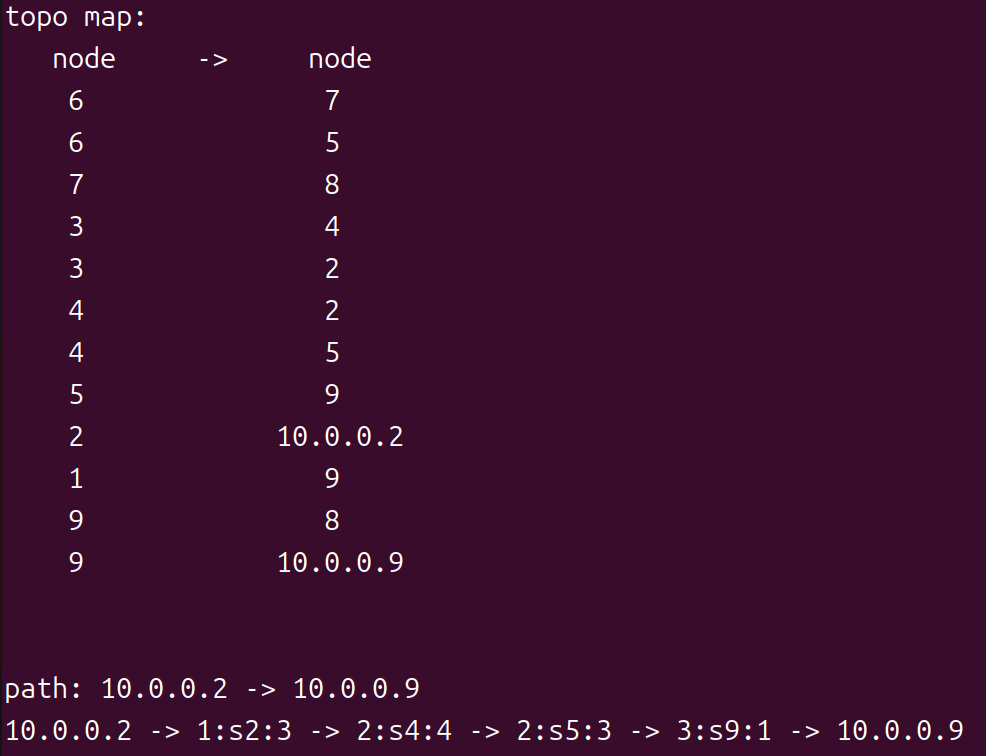


图4.3 控制台输出

可见，最小跳数路径为：h2 → s2 → s4 → s5 → s9 → h9，共4跳。

## **4.4 最小延迟路径选择**

时延测量原理如图4.4。

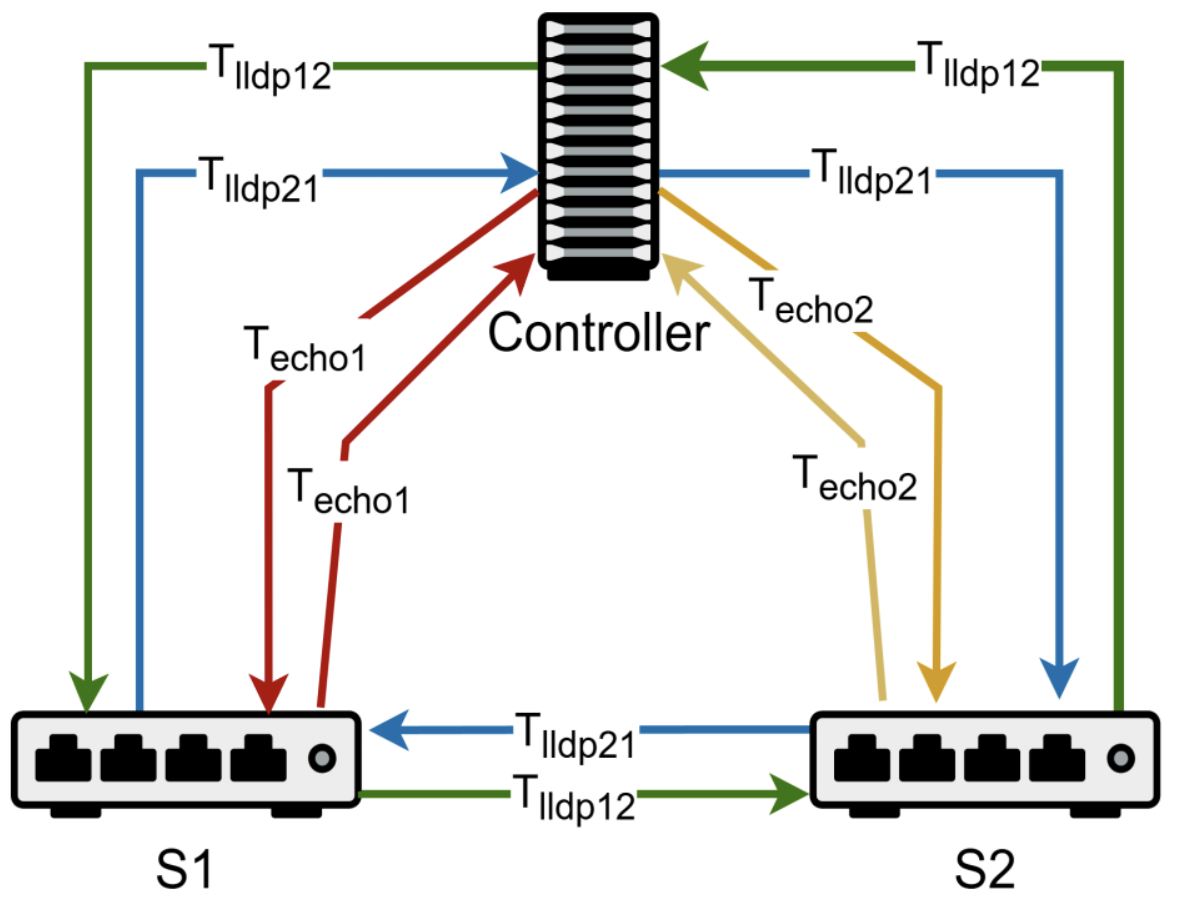


图4.4 时延测量原理

对于链路 (S1, S2)，单向时延计算为：

其中：

·Tlldp12: 控制器 → S1 → S2 → 控制器的往返时间（绿线）

·Tlldp21: 控制器 → S2 → S1 → 控制器的往返时间（蓝线）

·Techo1: 控制器 ↔ S1 的往返时间（红线）

·Techo2: 控制器 ↔ S2 的往返时间（黄线）

修改.venv/lib/python3.13/site-packages/os\_ken/topology/switches.py，在PortData中增加记录LLDP时延的变量，并在lldp\_packet\_in\_handler中计算Tlldp。

class **PortData**(**object**):

    def **\_\_init\_\_**(self, is\_down, lldp\_data):

**super**(**PortData**, self).**\_\_init\_\_**()

        self.is\_down = is\_down

        self.lldp\_data = lldp\_data

        self.timestamp = None    *# 新增：记录 LLDP 发送时间*

        self.sent = 0

        self.delay = 0           *# 新增：记录 T\_lldp*

**@set\_ev\_cls**(ofp\_event.EventOFPPacketIn, MAIN\_DISPATCHER)

def **lldp\_packet\_in\_handler**(self, ev):

*# ========== 新增开始 ==========*

    recv\_timestamp = time.time()  *# 记录接收时间*

*# ... 原有代码 ...*

*# 计算 LLDP 时延并保存到 port\_data*

    for port, port\_data in self.ports.items():

        if src\_dpid == port.dpid and src\_port\_no == port.port\_no:

            send\_timestamp = port\_data.timestamp

            if send\_timestamp:

                port\_data.delay = recv\_timestamp - send\_timestamp

*# ========== 新增结束 ==========*

修改network\_awareness.py，给NetworkAwareness类加LLDP延迟表和switches实例，并在NetworkAwareness类中添加function：packet\_in\_hander，用于处理LLDP消息。

def **\_\_init\_\_**(self, \*args, \*\*kwargs):

**super**(NetworkAwareness, self).**\_\_init\_\_**(\*args, \*\*kwargs)

*# ... 原有代码 ...*

    self.lldp\_delay\_table = {}    *# (src\_dpid, dst\_dpid) -> T\_lldp*

    self.switches = {}            *# switches 实例*

from os\_ken.base.app\_manager import lookup\_service\_brick

from os\_ken.topology.switches import LLDPPacket

**@set\_ev\_cls**(ofp\_event.EventOFPPacketIn, MAIN\_DISPATCHER)

def **packet\_in\_handler**(self, ev):

    """处理 LLDP 消息，获取 LLDP 时延"""

    msg = ev.msg

    dpid = msg.datapath.id

    try:

        src\_dpid, src\_port\_no = LLDPPacket.lldp\_parse(msg.data)

*# 获取 switches 实例（只需获取一次）*

        if not self.switches:

            self.switches = lookup\_service\_brick('switches')

*# 从 switches 中获取 LLDP 时延*

        for port in self.switches.ports.keys():

            if src\_dpid == port.dpid and src\_port\_no == port.port\_no:

*# 保存 T\_lldp*

                self.lldp\_delay\_table[(src\_dpid, dpid)] = \

                    self.switches.ports[port].delay

                break

    except:

        return

周期发送Echo，记录控制器与交换机的RTT。修改network\_awareness.py，增加数据结构echo\_RTT\_table来记录 Echo RTT、增加echo\_send\_timestamp用于记录Echo的发送时间；实现function：send\_echo\_request，构造OFPEchoRequest消息并发送，记录send\_time并存入echo\_send\_timestamp[dpid]；处理Echo回复，计算Techo，编写处理echo包的函数echo\_reply\_handler与事件EventOFPEchoReply进行绑定；周期性向每个交换机发送Echo。

def **\_\_init\_\_**(self, \*args, \*\*kwargs):

**super**(NetworkAwareness, self).**\_\_init\_\_**(\*args, \*\*kwargs)

*# ... 原有代码 ...*

    self.echo\_RTT\_table = {}       *# dpid -> T\_echo*

    self.echo\_send\_timestamp = {}  *# dpid -> send\_time*

ef send\_echo\_request(self, switch):

    """向交换机发送 Echo 请求"""

    datapath = switch.dp

    parser = datapath.ofproto\_parser

    dpid = datapath.id

*# 记录发送时间*

    send\_time = time.time()

    self.echo\_send\_timestamp[dpid] = send\_time

*# 构造 Echo 请求（data 必须是 bytes 类型）*

    data = **str**(send\_time).**encode**('utf-8')

    echo\_req = parser.OFPEchoRequest(datapath, data=data)

*# 发送*

    datapath.send\_msg(echo\_req)

**@set\_ev\_cls**(ofp\_event.EventOFPEchoReply, MAIN\_DISPATCHER)

def **handle\_echo\_reply**(self, ev):

    """处理 Echo 回复，计算 T\_echo"""

    try:

        msg = ev.msg

        datapath = msg.datapath

        dpid = datapath.id

*# 记录接收时间*

        recv\_time = time.time()

*# 获取发送时间（可选：从 data 中解码验证）*

        send\_time = self.echo\_send\_timestamp.get(dpid)

        if send\_time:

*# 计算 Echo RTT*

            self.echo\_RTT\_table[dpid] = recv\_time - send\_time

    except **Exception** as e:

        self.logger.warning(f"Failed to handle echo reply: {e}")

def **examine\_echo\_RTT**(self):

    """周期性测量 Echo RTT"""

    while True:

*# 获取所有交换机*

        switches = get\_all\_switch(self)

*# 向每个交换机发送 Echo*

        for switch in switches:

            self.send\_echo\_request(switch)

*# 睡眠（使用 hub.sleep 减少影响）*

        hub.sleep(SEND\_ECHO\_REQUEST\_INTERVAL)

*# 在 \_\_init\_\_ 中启动线程*

def **\_\_init\_\_**(self, \*args, \*\*kwargs):

*# ... 原有代码 ...*

*# 启动 Echo 测量线程*

    self.echo\_thread = hub.spawn(self.examine\_echo\_RTT)

计算链路时延并更新拓扑图权重。修改network\_awareness.py，在NetworkAwareness中添加字典link\_delay\_table，用于存储链路时延；计算链路时延；更新topo.py，修改\_get\_topology()，使其能够计算delay并添加至edge的属性中。

def **\_\_init\_\_**(self, \*args, \*\*kwargs):

**super**(NetworkAwareness, self).**\_\_init\_\_**(\*args, \*\*kwargs)

*# ... 原有代码 ...*

    self.link\_delay\_table = {}     *# (dpid1, dpid2) -> delay*

def **calculate\_link\_delay**(self, src\_dpid, dst\_dpid):

    """

    计算链路单向时延

    公式: delay = max((T\_lldp12 + T\_lldp21 - T\_echo1 - T\_echo2) / 2, 0)

    """

    try:

*# 获取 LLDP 往返时延*

        lldp\_12 = self.lldp\_delay\_table.get((src\_dpid, dst\_dpid), 0)

        lldp\_21 = self.lldp\_delay\_table.get((dst\_dpid, src\_dpid), 0)

*# 获取 Echo RTT*

        echo\_1 = self.echo\_RTT\_table.get(src\_dpid, 0)

        echo\_2 = self.echo\_RTT\_table.get(dst\_dpid, 0)

*# 计算链路时延*

        delay = **max**((lldp\_12 + lldp\_21 - echo\_1 - echo\_2) / 2, 0.0)

*# 双向写缓存，后续可回退*

        self.link\_delay\_table[(src\_dpid, dst\_dpid)] = delay

        self.link\_delay\_table[(dst\_dpid, src\_dpid)] = delay

        return delay

    except **KeyError**:

*# 链路发现和延迟计算是异步的，可能出现键不存在的情况*

        return self.link\_delay\_table.get((src\_dpid, dst\_dpid), **float**('inf'))

*# topo.py*

for link in links:

*# ... 原有代码 ...*

    delay = self.calculate\_link\_delay(link.src.dpid, link.dst.dpid)

*# 添加边到拓扑图（包含 delay 属性）*

    self.topo\_map.add\_edge(

        link.src.dpid, link.dst.dpid,

        hop=1,

        delay=delay,  *# 新增 delay 属性*

        is\_host=False

    )

实现最小时延路径控制器：ShortestDelay。修改shortest\_delay.py的weight为delay，在handle\_ipv4中计算路径时延，handle\_arp中实现ARP环路检测（同least\_hops.py）。

class **ShortestDelay**(**app\_manager**.**OSKenApp**):

    def **\_\_init\_\_**(self, \*args, \*\*kwargs):

**super**(**ShortestDelay**, self).**\_\_init\_\_**(\*args, \*\*kwargs)

        self.network\_awareness = kwargs['network\_awareness']

        self.weight = 'delay'  *# ⚠️ 改为 'delay'*

        self.mac\_to\_port = {}

        self.sw = {}

        self.path = None

def **handle\_ipv4**(self, msg, src\_ip, dst\_ip, pkt\_type):

*# ... 原有代码 ...*

*# 构建链路时延字典*

    link\_delay\_dict = {}

    path\_delay = 0.0

    for i in **range**(1, **len**(dpid\_path) - 1):

        src = dpid\_path[i]

        dst = dpid\_path[i + 1]

*# 从拓扑图中获取时延*

        if self.network\_awareness.topo\_map.has\_edge(src, dst):

            delay = self.network\_awareness.topo\_map[src][dst].get('delay', 0)

            link\_delay\_dict[f"s{src}->s{dst}"] = delay \* 1000  *# 转换为 ms*

            path\_delay += delay

*# 计算 Path RTT（往返时间 = 2 \* 单向时延）*

    path\_RTT = path\_delay \* 2

*# 输出结果*

    self.logger.info('link delay dict: %s', link\_delay\_dict)

    self.logger.info("path delay = %.5fms", path\_delay \* 1000)

    self.logger.info("path RTT = %.5fms", path\_RTT \* 1000)

*# ... 下发流表 ...*

在实现上，首先修改了os\_ken 的底层代码以记录LLDP包的收发时间戳，从而计算出 T\_lldp。接着，在 network\_awareness.py 模块中，启动一个周期性任务，不断向所有交换机发送 Echo 请求并处理其回复，以此来测量并更新各个 T\_echo 值。然后，该模块会使用上述公式整合 T\_lldp 和 T\_echo 数据，计算出所有链路的 delay，并将其作为权重更新到全局网络拓扑图中。最后，一个名为 shortest\_delay.py 的控制器应用被实现，它在进行路径选择时，将拓扑图的权重明确设置为 delay。当需要为数据包（如从 h2 到 h9）计算路径时，该应用会基于链路的延迟权重在拓扑图上执行最短路径算法，找出累积延迟最小的路径，并据此下发流表规则到沿途的交换机，从而引导后续流量沿此时延最优的路径传输。

启动拓扑执行h2 ping h9，通信结果如图4.5，最小时延及路径如图4.6，控制器发现的网络拓扑如图4.7，各交换机链路的单向时延如图4.8。

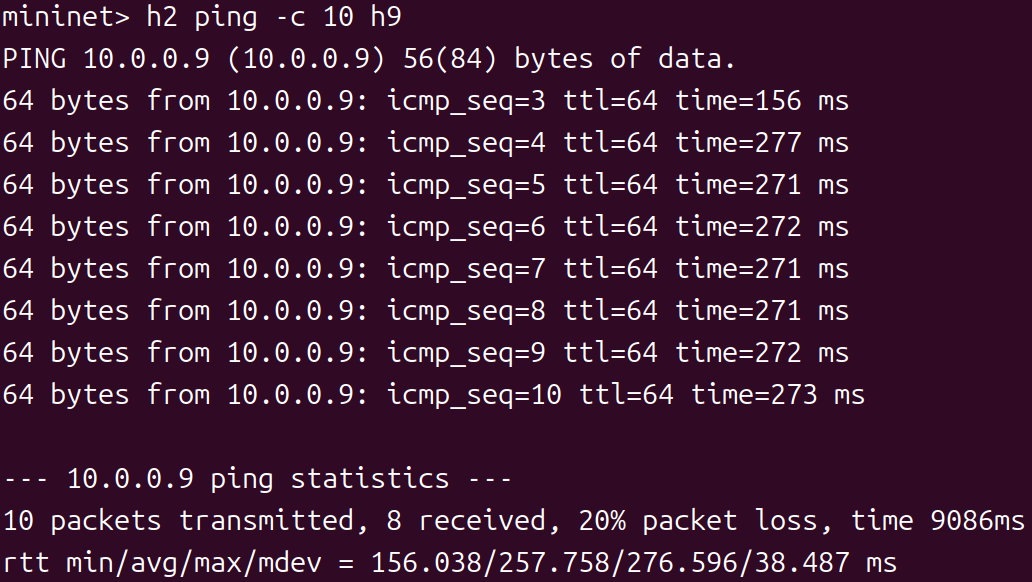


图4.5 h2与h9通信结果

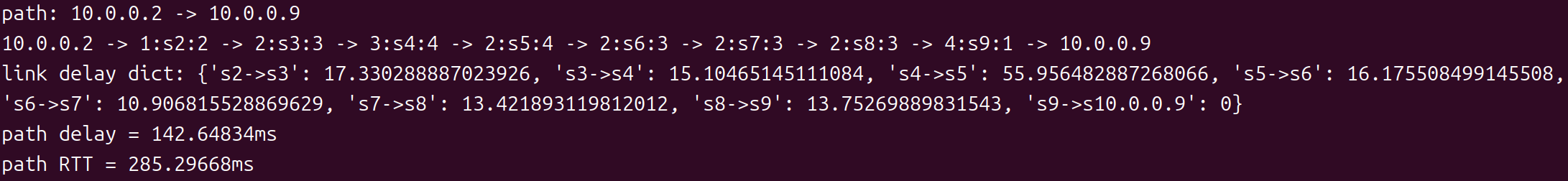


图4.6 最小时延及路径

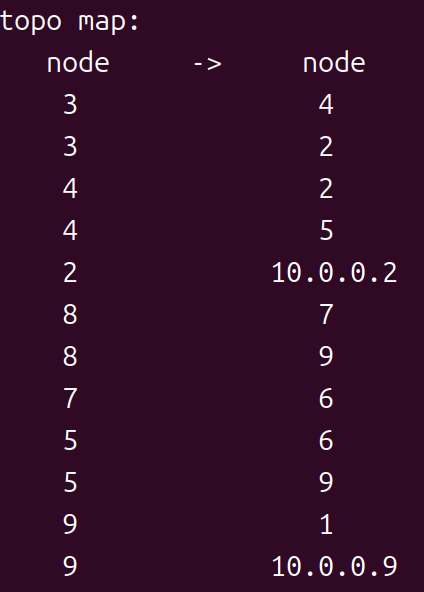


图4.7 控制器发现的网络拓扑

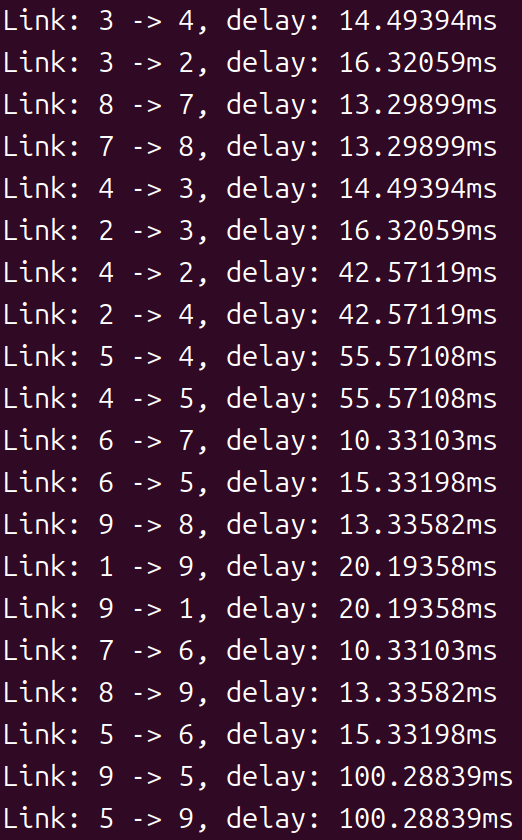


图4.8 各交换机的单向时延

可见，最小时延路径为：h2 → s2 → s3 → s4 → s5 → s6 → s7 → s8 → s9 → h9。

## **4.5 容忍链路故障**

故障恢复原理：链路状态发生变化时，OpenFlow会触发EventOFPPortStatus事件。

**@set\_ev\_cls**(ofp\_event.EventOFPPortStatus, MAIN\_DISPATCHER)

def **port\_status\_handler**(self, ev):

    msg = ev.msg

    reason = msg.reason

*# reason 类型：*

*# - OFPPR\_ADD: 端口新增*

*# - OFPPR\_MODIFY: 端口修改（link up/down）*

*# - OFPPR\_DELETE: 端口删除*

故障恢复流程：

·步骤 1: 链路故障 (link s6 s7 down)

→ 触发 EventOFPPortStatus

→ reason = OFPPR\_MODIFY

·步骤 2: 清理旧数据

→ 清空拓扑图 (topo\_map.clear())

→ 删除所有流表

→ 清空 sw、mac\_to\_port

·步骤 3: 重新发现拓扑

→ LLDP 自动重新发现

→ 构建新的拓扑图

·步骤 4: 自动路径切换

→ 交换机无流表 → PacketIn

→ 控制器重新计算最优路径

→ 下发新流表

在shortest\_path.py中实现链路故障检测与恢复逻辑。当链路状态发生变化（如端口新增、修改或删除）时，控制器会捕获 [EventOFPPortStatus](vscode-file://vscode-app/c:/Users/Asus/AppData/Local/Programs/Microsoft VS Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") 事件并触发 [port\_status\_handler](vscode-file://vscode-app/c:/Users/Asus/AppData/Local/Programs/Microsoft VS Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") 函数。该函数首先清空所有缓存的状态信息，包括网络拓扑图和 MAC 地址学习表，然后调用 [delete\_all\_flow](vscode-file://vscode-app/c:/Users/Asus/AppData/Local/Programs/Microsoft VS Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") 函数来删除所有交换机上的全部流表项。[delete\_all\_flow](vscode-file://vscode-app/c:/Users/Asus/AppData/Local/Programs/Microsoft VS Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") 在清除每个交换机的流表后，会立即为其重新安装一个 table-miss 流表项，该规则将所有未知数据包都发送给控制器。由于拓扑和流表均已清空，后续的数据包（如 ping）会触发 Packet-In 事件上报至控制器，控制器利用此时已由 LLDP 重新发现并构建的新拓扑图，计算出新的最优路径，并下发相应的流表规则，从而自动完成路径的切换与恢复。

*# 1) 确保有一个装 table-miss 的工具函数*

def **install\_table\_miss**(self, datapath):

    ofp = datapath.ofproto

    parser = datapath.ofproto\_parser

*# priority 0, match any -> send to controller*

    match = parser.OFPMatch()

    actions = [parser.OFPActionOutput(ofp.OFPP\_CONTROLLER, ofp.OFPCML\_NO\_BUFFER)]

    inst = [parser.OFPInstructionActions(ofp.OFPIT\_APPLY\_ACTIONS, actions)]

    mod = parser.OFPFlowMod(

        datapath=datapath,

        priority=0,

        table\_id=ofp.OFPTT\_ALL if **hasattr**(ofp, "OFPTT\_ALL") else 0,

        command=ofp.OFPFC\_ADD,

        match=match,

        instructions=inst,

    )

    datapath.send\_msg(mod)

*# 2) 在交换机首次连接时装一次 table-miss*

**@set\_ev\_cls**(ofp\_event.EventOFPSwitchFeatures, CONFIG\_DISPATCHER)

def **switch\_features\_handler**(self, ev):

    datapath = ev.msg.datapath

    self.install\_table\_miss(datapath)

*# 3) 删“与端口相关”的流（in\_port 与 out\_port 两类），并确保删所有表；最后发 barrier*

def **delete\_flow**(self, datapath, port\_no):

    ofp = datapath.ofproto

    parser = datapath.ofproto\_parser

    try:

*# a) 删除匹配 in\_port=port\_no 的流（所有表）*

        match\_in = parser.OFPMatch(in\_port=port\_no)

        mod\_in = parser.OFPFlowMod(

            datapath=datapath,

            table\_id=ofp.OFPTT\_ALL,

            command=ofp.OFPFC\_DELETE,

            out\_port=ofp.OFPP\_ANY,

            out\_group=ofp.OFPG\_ANY,

            match=match\_in

        )

        datapath.send\_msg(mod\_in)

*# b) 删除“动作输出到 port\_no”的流：指定 out\_port=port\_no，match 置空*

        match\_any = parser.OFPMatch()

        mod\_out = parser.OFPFlowMod(

            datapath=datapath,

            table\_id=ofp.OFPTT\_ALL,

            command=ofp.OFPFC\_DELETE,

            out\_port=port\_no,

            out\_group=ofp.OFPG\_ANY,

            match=match\_any

        )

        datapath.send\_msg(mod\_out)

*# c) 发 barrier，让删除顺序生效*

        datapath.send\_msg(parser.OFPBarrierRequest(datapath))

        self.logger.info("Cleared flows touching s%s port %s", datapath.id, port\_no)

    except **Exception** as e:

            self.logger.error("Failed to delete flows on s%s p%s: %s",

                                datapath.id, port\_no, e)

*# 4) 遍历所有交换机所有端口删流；删完立刻“重装 table-miss”*

def **delete\_all\_flow**(self):

    switches = get\_all\_switch(self)  *# 关键：传 self*

    for sw in switches:

        dp = sw.dp

        for p in sw.ports:

            port\_no = p.port\_no

            if port\_no == dp.ofproto.OFPP\_LOCAL:

                continue

            self.delete\_flow(dp, port\_no)

*# 删完该交换机上的流，马上重装 table-miss，保证后续 packet\_in 不丢*

        self.install\_table\_miss(dp)

*# 5) 端口状态事件：清缓存 + 删流（会重装 table-miss）*

**@set\_ev\_cls**(ofp\_event.EventOFPPortStatus, MAIN\_DISPATCHER)

def **port\_status\_handler**(self, ev):

    msg = ev.msg

    dp = msg.datapath

    ofp = dp.ofproto

    reason = msg.reason

*# 记录一下*

    try:

        pno = msg.desc.port\_no

        self.logger.info("[PortStatus] s%s p%s reason=%s state=0x%x",

                            dp.id, pno, reason, msg.desc.state)

    except **Exception**:

        self.logger.info("[PortStatus] s%s reason=%s", dp.id, reason)

    if reason in (ofp.OFPPR\_ADD, ofp.OFPPR\_MODIFY, ofp.OFPPR\_DELETE):

*# 1) 清空拓扑与学习表*

        if **hasattr**(self, "network\_awareness"):

            na = self.network\_awareness

            if **hasattr**(na, "topo\_map"):

                na.topo\_map.clear()

        if **hasattr**(self, "sw"): self.sw.clear()

        if **hasattr**(self, "mac\_to\_port"): self.mac\_to\_port.clear()

*# 2) 删所有交换机相关流（函数里会为每台交换机重装 table-miss）*

        self.delete\_all\_flow()

启动网络拓扑，先执行h2 ping h9，初始状态下RTT≈270ms，通信结果如图4.9，最小时延及路径如图4.10。

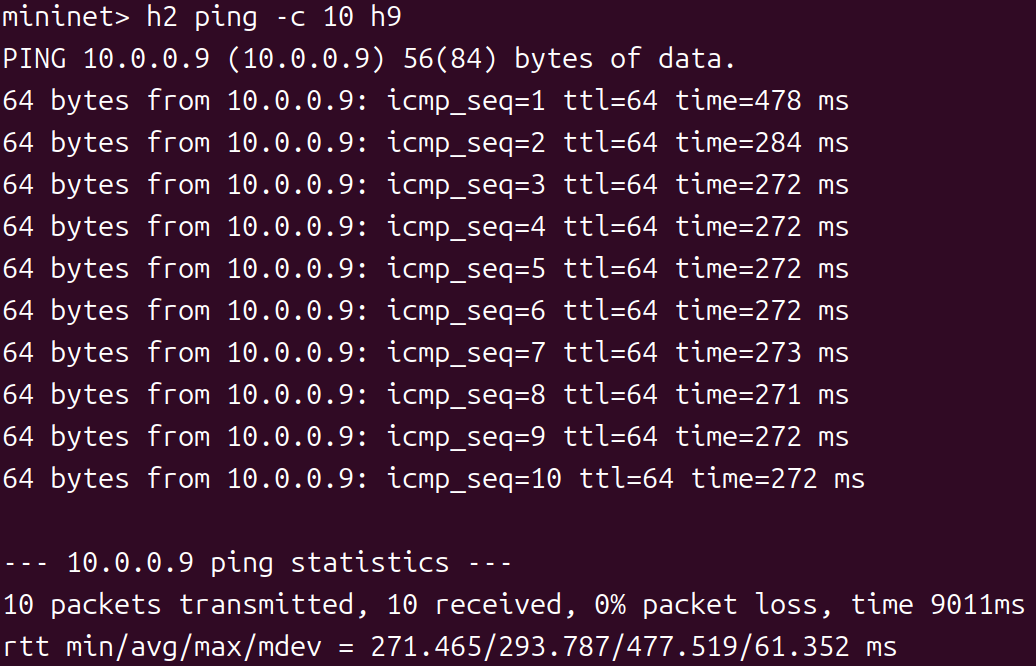


图4.9 h2与h9通信结果

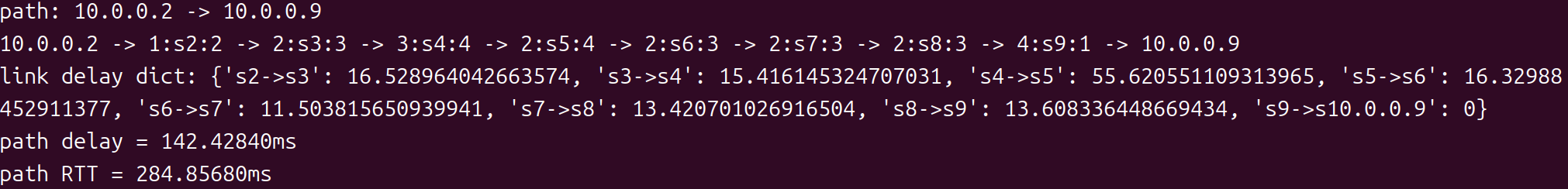


图4.10 最小时延及路径

接着执行link s6 s7 down模拟链路故障，再执行h2 ping h9，RTT≈370ms，通信结果如图4.11，最小时延及路径如图4.12。

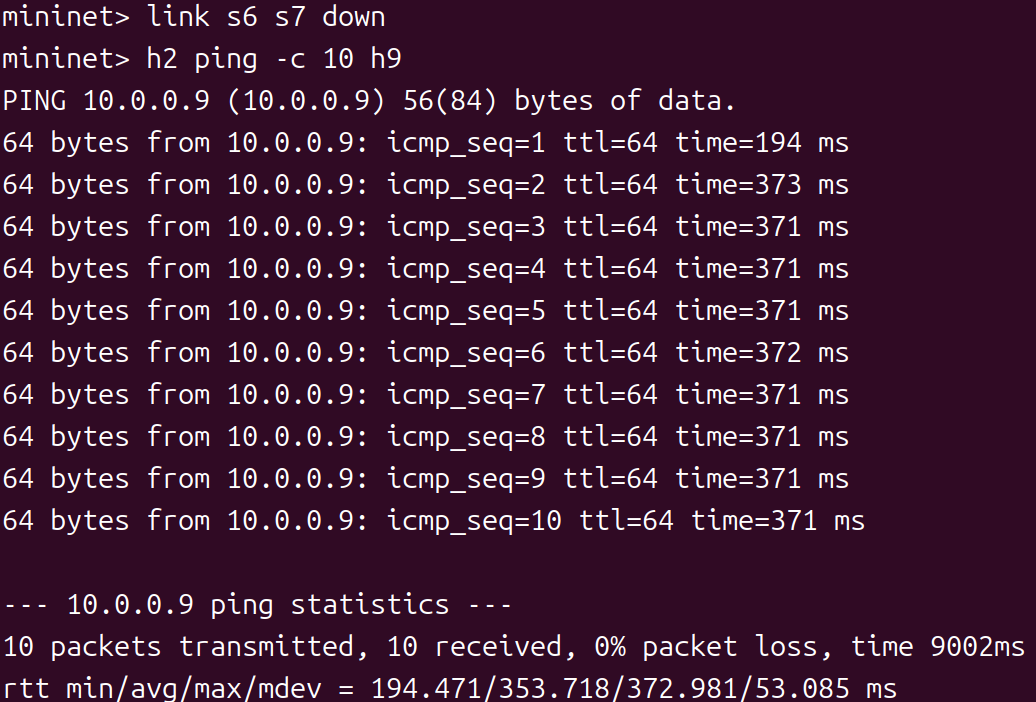


图4.11 h2与h9通信结果

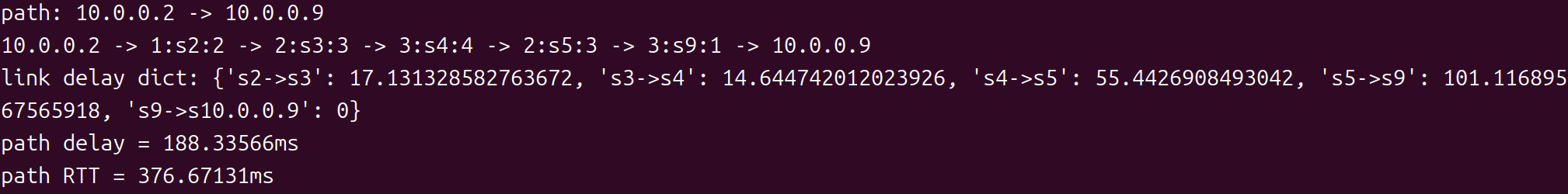


图4.12 最小时延及路径

接着执行link s6 s7 up模拟链路恢复，再执行h2 ping h9，RTT≈270ms，通信结果如图4.13，最小时延及路径如图4.14。

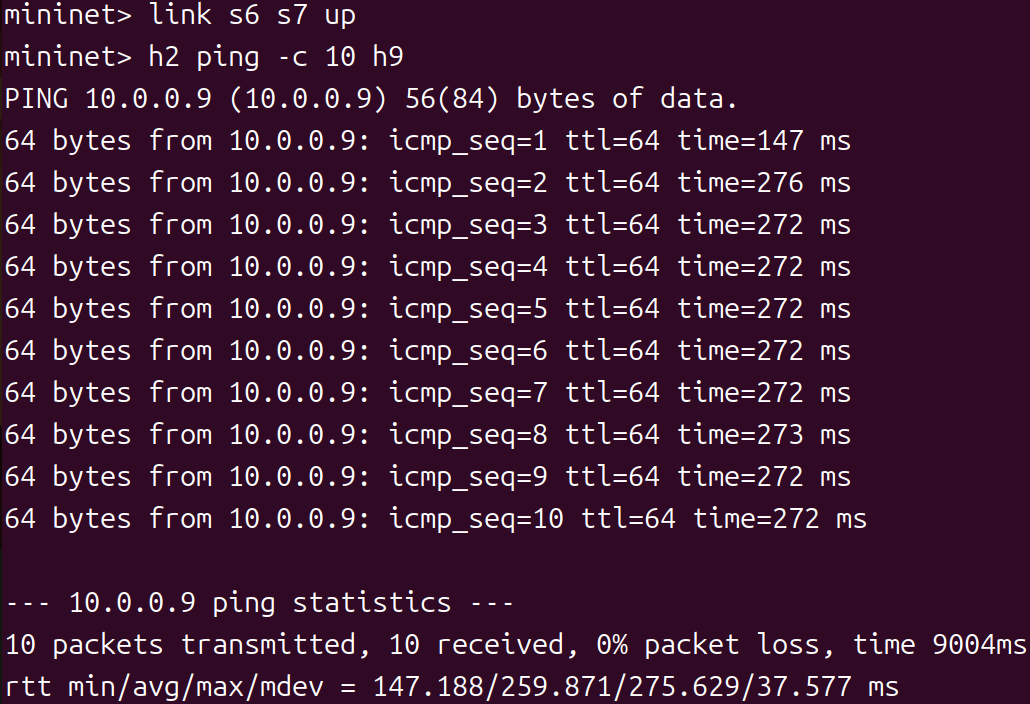


图4.13 h2与h9通信结果

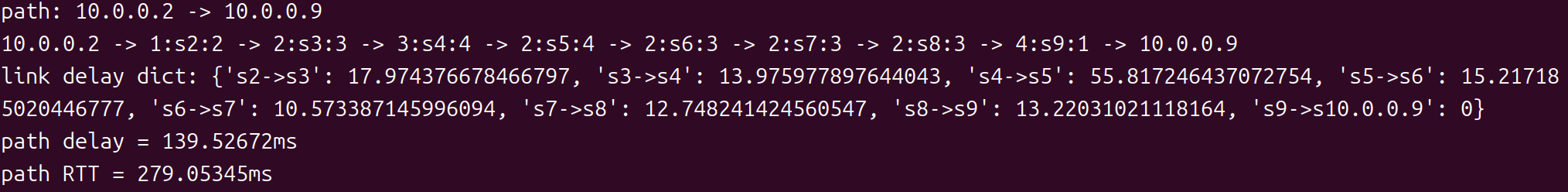


图4.14 最小时延及路径

·初始路径: h2 → s2 → s3 → s4 → s5 → s6 → s7 → s8 → s9 → h9

·故障路径: h2 → s2 → s3 → s4 → s5 → s9 → h9

·恢复路径: h2 → s2 → s3 → s4 → s5 → s6 → s7 → s8 → s9 → h9

清空 topo\_map、sw 和 mac\_to\_port 这类对象是因为它们代表了网络的动态状态和拓扑结构。当链路发生故障时，整个网络的拓扑图 (topo\_map) 已经发生了根本性改变，旧的拓扑信息是过时且错误的。基于这个旧拓扑学习到的主机位置信息 (mac\_to\_port) 和交换机状态 (sw) 也随之失效。如果不清空这些缓存，控制器会基于错误的信息计算路径，导致流量被发送到已断开的链路上，造成通信中断。因此，必须强制控制器通过 LLDP 等机制重新发现和构建一个全新的、准确的网络拓扑视图，并新学习主机位置。

相比之下，lldp\_delay\_table 这类对象记录的是链路的固有属性，例如单个链路的延迟。链路的延迟是其物理特性，并不会因为网络中其他地方的另一条链路发生故障而改变。当一条链路故障后，它会从新的拓扑图中被移除，其对应的延迟值在路径计算时自然就不会被使用。而其他仍然存在的链路，它们的延迟值是稳定且有效的，保留这些值可以避免不必要的重复测量，使得控制器在构建新路径时能立即使用这些已知的、准确的延迟数据。因此，记录链路固有属性的对象不需要在拓扑变化时清空。

## **4.6 其它需要说明的问题**

最小跳数与最小时延路径表面上只是“权重不同的最短路”，但实际实现过程中可以发现两者对控制器和底层库的依赖程度并不一样。最小跳数路径完全依赖 NetworkX 对无权图的计算，只要拓扑发现是正确的，路径就基本稳定；而最小时延路径则高度依赖 LLDP 与 Echo 报文的精确时间戳测量，实验中所有交换机、控制器都跑在同一台虚拟机上，操作系统调度延迟、Python 解释器开销、甚至虚拟机本身的负载都会引入噪声，因此控制台打印出来的链路单向时延更多是“相对大小”可信，绝对数值并不能简单等同于真实物理链路时延。

本实验为了便于观察行为，直接在 OS-Ken 的 switches.py 等源码中增加了记录时间戳和计算 LLDP 时延的字段，这种做法虽然直观，但会带来维护性问题。更工程化的做法应该是在控制器应用层通过已有的拓扑事件和 PacketIn 回调来做扩展。

在故障恢复部分，实验中采用的是“激进重置”的方式：端口状态变化时清空 topo\_map、sw、mac\_to\_port，并删除所有流表再重新安装 table-miss 规则，这能保证链路变化后不会继续沿用任何旧状态，从而强制走一轮新的 PacketIn 与路径计算，逻辑上比较“保险”。但这种做法的代价是故障瞬间会导致全网所有流量短暂中断，在更大规模拓扑下将带来明显的收敛抖动。实际网络往往只针对受影响的交换机和相关端口进行“局部收敛”，或者通过版本号、标签等方式增量更新路径。

## **4.7 参考资料**

[shortest\_simple\_paths — NetworkX 3.5 documentation](https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.simple_paths.shortest_simple_paths.html)

[内置类型 — Python 3.13.9 文档](https://docs.python.org/zh-cn/3.13/library/stdtypes.html" \l "mapping-types-dict)

[OpenFlow v1.3 Messages and Structures — os-ken 3.2.0.dev9 documentation](https://docs.openstack.org/os-ken/latest/ofproto_v1_3_ref.html" \l "echo-request)

# 心得体会与建议

## **5.1 心得体会**

SDN的一系列实验，让我对“网络是怎么跑起来的”这件事有了更加立体和具象的认识。第一章环境搭建让我熟悉了Mininet、OVS、Wireshark 等工具链，也第一次站在“上帝视角”观察一个小网络中所有主机、交换机和链路的状态，为后续SDN实验打下了基础。

在 FAT Tree 搭建实验中，我体会到参数化构造大规模拓扑的便利，同时也踩到了不少坑：比如默认secure模式下没有控制器会直接丢包、不开 STP 就会在 fat-tree 这种高度冗余拓扑里引发广播风暴等。这一过程让我对“二层自学习 + 生成树算法”在传统网络中的作用有了直观感受，也更加理解为什么数据中心场景需要通过 SDN 来做更精细的路径与冗余管理。

第三章自学习与环路检测实验可以说是我真正理解控制器的起点。从 simple\_switch.py 的洪泛行为，到自己维护 mac\_to\_port 映射，再到基于转发历史信息构造环路检测键值 (dpid, src, dst\_ip)，每一步调试的过程都离不开控制器日志和 Wireshark 抓包。这个实验让我第一次意识到：SDN 的难点不只是“写出算法”，更重要的是在实际报文和状态中验证自己的逻辑是否真的生效。

在第四章链路选择与故障恢复实验中，我对控制平面与数据平面分离的理解又加深了一层。修改 OS-Ken 源码、插入时间戳、计算 LLDP/Echo 时延的过程中，我看到了SDN在工程实现上的复杂度——需要同时处理异步事件、时间精度、状态一致性等一系列问题。链路 down/up 之后，观察控制器自动清空状态、重新发现拓扑并收敛到新路径的过程，让我对“网络具备自愈能力”有了更直观的认识，也对传统路由协议中的收敛问题产生了更多兴趣。

这门课程的实验让抽象的协议、拓扑和控制理论都落地到了可观测、可调试的系统行为上。我不仅加深了对课本知识的理解，也初步掌握了使用 Mininet、OS-Ken、NetworkX 等工具搭建和调试 SDN 系统的基本能力，为今后继续学习网络编程、网络虚拟化和数据中心网络打下了良好基础。

## **5.2 建议**

从整体体验上看，本课程的实验体系完整且循序渐进，但也有一些可以进一步优化的地方。首先是在实验环境方面，虽然课程已经提供了预装好 Mininet 和 OS-Ken 的虚拟机镜像，但在实际使用过程中，不同同学的硬件环境、虚拟机配置会带来性能差异，影响到时延测量和 LLDP/Echo 的收敛效果。建议今后可以在实验指导书中加入一小节“环境自检”，例如通过一个简单脚本自动检查 Python 版本、依赖库版本以及 Mininet/OS-Ken 是否正常工作，并给出推荐的虚拟机 CPU/内存配置范围，这样可以减少同学们在环境问题上花费的时间。

其次，在涉及修改第三方库源码的实验（如本次对 os\_ken.topology.switches 的改动）中，建议老师可以在课上或指导书里增加一些背景说明：例如为什么要选择在该层进行修改、如果不改源码是否有其他替代方案、以及在真实工程中更推荐的做法是什么。对于初学者来说，直接进入 .venv 目录改源文件虽然能完成实验，但容易给人一种“库随便改就行”的错觉，如果能在教学中多强调接口抽象和模块边界的设计理念，会更有利于培养工程思维。

另外，在链路选择与故障恢复部分，实验内容已经比较贴近实际 SDN 控制器的场景，如果时间允许，我会希望课程可以增加一些“可选进阶任务”，例如让同学们尝试在最小时延权重之外加入简单的“链路负载”因子，或者要求对多对主机同时通信的情形进行观测和分析。通过这样的小扩展，大家可以更直观地感受到单一指标选路的局限性，以及流量工程中多目标权衡的复杂性，这对有意继续在网络领域深入学习的同学会是很好的启发。

从学习节奏上看，这几次 SDN 实验的代码量和知识跨度都不小，而课上对 OS-Ken 事件模型、OpenFlow 报文类型的讲解相对有限。建议今后可以在课堂或实验前预留一点时间，专门讲解常见的事件流程（如 PacketIn、FlowMod、PortStatus）和典型调试方法（如通过日志和 Wireshark 协同分析），并配合一两个精简的示例控制器，让同学们更快跨过“能看懂、敢动手”的门槛。整体而言，这门课的实验设计已经非常有价值，如果在工程实践层面再多一些引导，相信会对学生的收获和兴趣提升有更大的帮助。