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 2401 姓名 刘辉宇 学号 U66666666

项目	Socket 编程（40%）	数据可靠传输协议设计（20%）	SDN 实验（20%）	平时成绩（20%）	总分
得分					

教学目标达成情况一览表

课程目标	1（20%）	2（15%）	3（10%）	4（5%）	5（15%）	6（20%）	7（15%）	总分
得分								

教师评语：

教师签名：

给分日期：

目　　录

华中科技大学计算机学院 I
《计算机通信与网络》实验报告 I
教学目标达成情况一览表 I
实验一 Socket 编程实验 1
1.1 环境 1
1.2 系统功能需求 1
1.3 系统设计 2
1.4 系统实现 5
1.5 系统测试及结果说明 6
1.6 其它需要说明的问题 9
1.7 参考文献 9
实验二 数据可靠传输协议设计实验 10
2.1 环境 10
2.2 实验要求 10
2.3 协议的设计、验证及结果分析 10
2.4 其它需要说明的问题 20
2.5 参考文献 20
实验三 SDN 网络实验 21
3.1 环境 21
3.2 实验要求 21
3.3 FAT TREE 搭建 21
3.4 自学习和环路检测实验 29
3.5 链路选择和故恢复实验 41
3.6 参考文献 51
心得体会与建议 52
4.1 心得体会 52
4.2 建议 52

实验三 SDN 网络实验

3.1 环境

操作系统：Ubuntu 22.04
实验软件：VMware Workstation

3.2 实验要求

本次实验引导进行理解 SDN 基础理论与典型架构，通过 Mininet 网络仿真平台和 Open vSwitch 工具，实现主机组网、流表管理和协议调试，强化对网络拓扑控制和交换节点操作能力。学生需系统掌握 SDN 仿真环境搭建、网络协议配置、虚拟交换机管理，以及流量分析与控制方法。

1. 围绕 Fat Tree 网络拓扑的构建与验证。首先要求学生利用 Mininet 编写实验脚本搭建多 Pod 分层结构，理解 Fat Tree 拓扑中的 Pod、交换机和主机分布；并用 pingall 等工具测试主机通信与链路连通性，对 k 参数变化下拓扑结构进行分析与性能对比，掌握主流数据中心网络的层次化特征。

2. SDN 控制器下的自学习交换机与环路检测。学生需基于 OpenFlow 协议编写控制器应用，完成自学习 L2 交换、环路断开和检测、多种流表项下发等功能。实验过程中要准确处理 PacketIn 事件、实现 MAC 表维护、分析流表匹配逻辑，同时用 Mininet 和 Wireshark 辅助抓取和分析转发行为。

3. 链路发现协议（LLDP）、最短路径转发与链路延迟测量。要求学生实现 SDN 控制器的链路发现功能，利用 LLDP 报文快速检测物理和虚拟链路状态，通过控制器应用实现链路延迟统计与最短路径计算，验证 SDN 网络中的链路动态感知与全局路由优化能力。

3.3 FAT TREE 搭建

3.3.1 实验拓扑结构

拓扑类型：FAT TREE (k=4)
主机数量：16 台（h1-h16）
交换机数量：20 台（边缘层 8 台，e00-e31;聚合层 8 台，a00-a31；核心层 4 台，c0-c3）

Pod 数量：4 个，每个 Pod 包含 2 个变元交换机和 2 个聚合交换机

主机 IP 地址分配规则：

```
10.<Pod+1>.<Edge+1>.<Host+1>
例如：
h1: 10.1.1.1 (Pod0, Edge0, Host0)
h16: 10.4.2.2 (Pod3, Edge1, Host1)
```

3.3.2 代码实现

基于 Mininet Python API 实现 k=4 的 Fat Tree 拓扑。代码采用面向对象设计，定义 FatTreeTopo 类继承自 Topo。初始化时根据 k=4 计算拓扑参数：4 个 Pod，每个 Pod 包含 2 个边缘交换机和 2 个聚合交换机，核心层共 4 个交换机。拓扑构建分为三个步骤：首先通过嵌套循环创建所有交换机，并按 Pod 组织；然后为每个边缘交换机创建 2 个主机，共 16 个；最后按照 Fat Tree 规则建立三层链接：边缘层连接主机、边缘层连接聚合层、聚合层连接核心层。核心层连接采用公式 `core_idx = (agg * (k/2) + core_group) % core_switches` 实现，确保每个聚合交换机连接到不同的核心交换机。运行函数 `run_fat_tree()`创建网络后，将所有 OVS 交换机配置为 standalone 模式，不依赖控制器。随后使用 `pingAll()` 测试连通性，并可通过 `ovs-appctl fdb/show` 查看 MAC 地址表，分析数据包转发路径，具体代码如下：

```
#!/usr/bin/env python3
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import OVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
class FatTreeTopo(Topo):
    def __init__(self, k=4):
        super(FatTreeTopo, self).__init__()
        self.k = k
        self.pod_num = k
        self.edge_switches_per_pod = k // 2
        self.agg_switches_per_pod = k // 2
        self.core_switches = (k // 2) ** 2
        self._create_switches()
        self._create_hosts()
        self._create_links()
```

```
    def _create_switches(self):
        self.edge_switches = []
        self.agg_switches = []
        self.core_switches_list = []
        # Create edge switches (2 per pod, 4 pods = 8 total)
        for pod in range(self.pod_num):
            pod_edges = []
            for edge in range(self.edge_switches_per_pod):
                switch_name = f'e{pod}{edge}'
                switch = self.addSwitch(switch_name)
                pod_edges.append(switch)
            self.edge_switches.append(pod_edges)
        # Create aggregation switches (2 per pod, 4 pods = 8 total)
        for pod in range(self.pod_num):
            pod_aggs = []
            for agg in range(self.agg_switches_per_pod):
                switch_name = f'a{pod}{agg}'
                switch = self.addSwitch(switch_name)
                pod_aggs.append(switch)
            self.agg_switches.append(pod_aggs)
        # Create core switches (4 total for k=4)
        for core_idx in range(self.core_switches):
            switch_name = f'c{core_idx}'
            switch = self.addSwitch(switch_name)
            self.core_switches_list.append(switch)
    def _create_hosts(self):
        """Create hosts: 2 hosts per edge switch"""
        self.hosts_list = []
        host_count = 0
        for pod in range(self.pod_num):
            pod_hosts = []
            for edge in range(self.edge_switches_per_pod):
                edge_hosts = []
                for host in range(self.k // 2):    # 2 hosts per edge switch for k=4
```

```
                    host_name = f'h{host_count + 1}'
                    host_ip = f'10.{pod + 1}.{edge + 1}.{host + 1}'
                    host_node = self.addHost(host_name, ip=host_ip)
                    edge_hosts.append(host_node)
                    host_count += 1
                pod_hosts.append(edge_hosts)
            self.hosts_list.append(pod_hosts)
    def _create_links(self):
        """Create all links according to Fat Tree topology rules"""
        self._create_edge_to_host_links()
        self._create_edge_to_agg_links()
        self._create_agg_to_core_links()
    def _create_edge_to_host_links(self):
        """Connect edge switches to hosts (ports 0-1 of edge switches)"""
        for pod in range(self.pod_num):
            for edge in range(self.edge_switches_per_pod):
                for host in range(self.k // 2):
                    host_node = self.hosts_list[pod][edge][host]
                    edge_switch = self.edge_switches[pod][edge]
                    self.addLink(host_node, edge_switch)
    def _create_edge_to_agg_links(self):
        """Connect edge switches to aggregation switches"""
        for pod in range(self.pod_num):
            for edge in range(self.edge_switches_per_pod):
                for agg in range(self.agg_switches_per_pod):
                    edge_switch = self.edge_switches[pod][edge]
                    agg_switch = self.agg_switches[pod][agg]
                    self.addLink(edge_switch, agg_switch)
    def _create_agg_to_core_links(self):
        for pod in range(self.pod_num):
            for agg in range(self.agg_switches_per_pod):
                # Each agg switch connects to k/2 = 2 core switches
                for core_group in range(self.k // 2):
                    core_idx = (agg * (self.k // 2) + core_group) % self.core_switches
                    agg_switch = self.agg_switches[pod][agg]
```

```
                core_switch = self.core_switches_list[core_idx]
                self.addLink(agg_switch, core_switch)

def run_fat_tree(ping_timeout=0.5):
    Args:
        ping_timeout: Timeout for each ping in pingall (default: 0.5s for speed)
                      Increase to 1.0 or 2.0 if you see timeout errors
    from mininet.clean import cleanup
    cleanup()
    # Create topology
    topo = FatTreeTopo(k=4)
    net = Mininet(topo=topo, switch=OVSSwitch, controller=None, autoSetMacs=True)
    net.start()
    for switch in net.switches:
        switch.cmd('ovs-vsctl set-fail-mode', switch.name, 'standalone')
    result = net.pingAll(timeout=ping_timeout)
    if result == 0:
        for switch in net.switches:
            try:
                output = switch.cmd(f"ovs-appctl fdb/show {switch.name}")
                info(output)
            except:
                info(f"Could not get MAC table for {switch.name}\n")
    CLI(net)
    net.stop()


if __name__ == '__main__':
    run_fat_tree(ping_timeout=0.5)    # Change to 1.0 or 2.0 if needed
```


3.3.2 测试流程和问题分析

执行 pingall 指令后，所有交换机连通失败（图 3.1）：

即使同一主机下的 h1 和 h2 也无法通信，查看 arp 条目显示所有为 incomplete（如图 3.2）

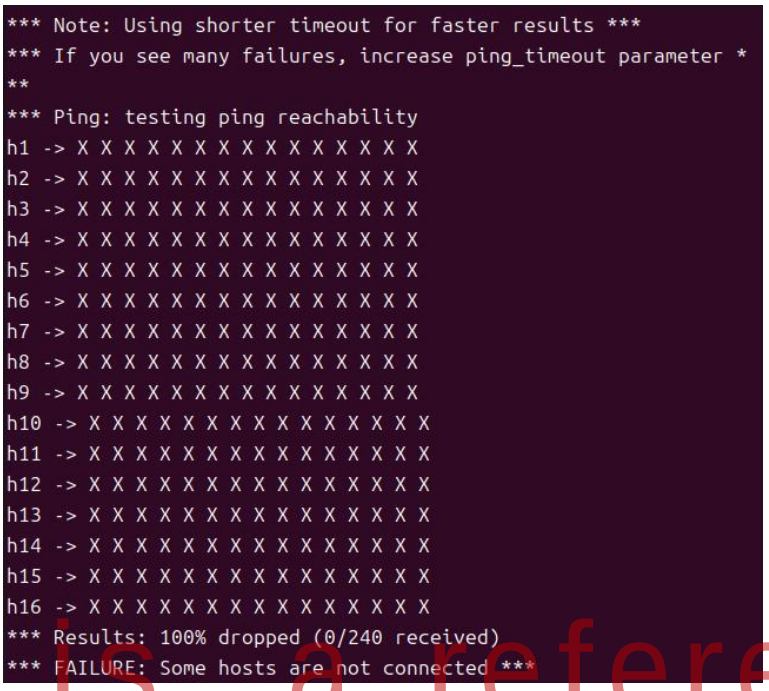

图 3.1

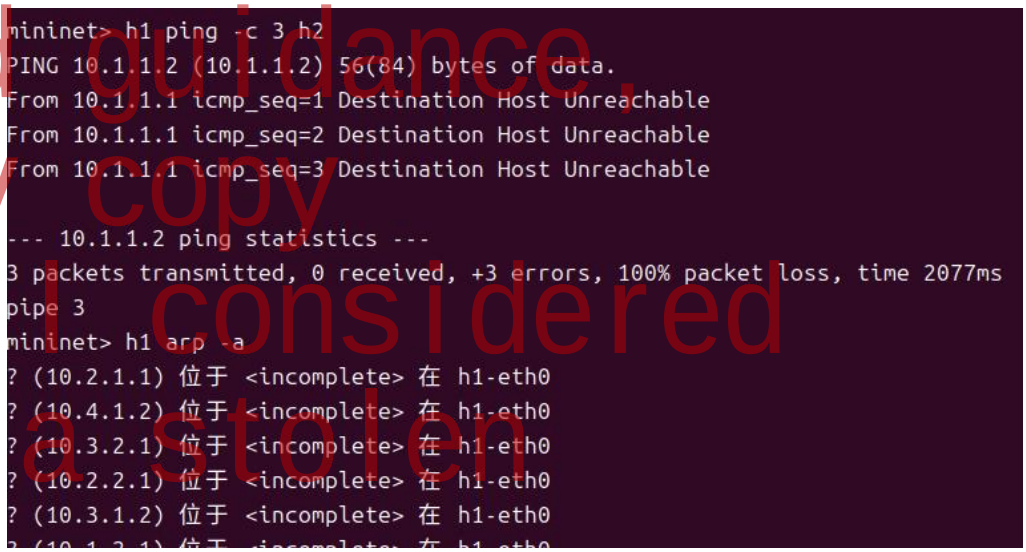

图 3.2

查看主机之间的交换机的失败原因，发现 OVS 交换机默认 fail-mode 为 secure。在 secure 模式下，五控制器的交换机不转发任何数据包。查看流表，现实有 NORMAL action，但是交换机依旧拒绝转发，诊断指令如下，结果如图 3.3:

sh ovs-vsctl get-fail-mode e00
sh ovs-ofctl dump-flows e00
h1 ping -c 3 h2

```
mininet> sh ovs-ofctl dump-flows e00     # 显示有 NORMAL action
 cookie=0x0, duration=840.970s, table=0, n_packets=66812134, n_bytes=5342083666,
 priority=0 actions=NORMAL
mininet> h1 ping -c 3 h2                 # 失败
PING 10.1.1.2 (10.1.1.2) 56(84) bytes of data.

--- 10.1.1.2 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2034ms
```

图 3.3

因此，我们要把所有交换机的 fail-mode 改成 standalone 模式，在该模式下，交换机像传统二层交换机一样工作，进行 MAC 地址自学习和转发：

```
sh ovs-vsctl set-fail-mode <交换机名> standalone
sh ovs-vsctl get-fail-mode e00 #验证
```



```
mininet> sh ovs-vsctl set-fail-mode e00 standalone
mininet> sh ovs-vsctl set-fail-mode e01 standalone
mininet> sh ovs-vsctl set-fail-mode e10 standalone
mininet> sh ovs-vsctl set-fail-mode e11 standalone
mininet> sh ovs-vsctl set-fail-mode e20 standalone
mininet> sh ovs-vsctl set-fail-mode e21 standalone
mininet> sh ovs-vsctl set-fail-mode e30 standalone
mininet> sh ovs-vsctl set-fail-mode e31 standalone
mininet> sh ovs-vsctl set-fail-mode a00 standalone
mininet> sh ovs-vsctl set-fail-mode a01 standalone
mininet> sh ovs-vsctl set-fail-mode a10 standalone
mininet> sh ovs-vsctl set-fail-mode a11 standalone
mininet> sh ovs-vsctl set-fail-mode a20 standalone
mininet> sh ovs-vsctl set-fail-mode a21 standalone
mininet> sh ovs-vsctl set-fail-mode a30 standalone
mininet> sh ovs-vsctl set-fail-mode a31 standalone
mininet> sh ovs-vsctl set-fail-mode c0 standalone
mininet> sh ovs-vsctl set-fail-mode c1 standalone
mininet> sh ovs-vsctl set-fail-mode c2 standalone
mininet> sh ovs-vsctl set-fail-mode c3 standalone
mininet> sh ovs-vsctl get-fail-mode e00
standalone
```

图 3.4

此时 h1 和 h2 仍然不能传输数据，交换机已经设置完成，我们继续查看主机状态，结果如图 3.5.我们发现 RX packets 和 dropped 数量巨大，说明存在广播风暴：

```
#查看主机状态
h1 ifconfig
# RX packets 和 dropped 数量异常巨大（数百万）
```

```
mininet> h1 ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.1.1.1  netmask 255.0.0.0  broadcast 10.255.255.255
        inet6 fe80::200:ff:fe00:1  prefixlen 64  scopeid 0x20<link>
        ether 00:00:00:00:00:01  txqueuelen 1000  (以太网)
        RX packets 36390731  bytes 2908954749 (2.9 GB)
        RX errors 0  dropped 72751761  overruns 0  frame 0
        TX packets 42  bytes 2300 (2.3 KB)
        TX errors 0  dropped 31 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (本地环回)
        RX packets 61  bytes 4672 (4.6 KB)
        RX errors 0  dropped 90  overruns 0  frame 0
        TX packets 61  bytes 4672 (4.6 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

图 3.5

信息提示出现本地环回，我们用 links 命令进行分析，如图 3.6 所示，我们以查看 Pod 0 为例子，出现环路：e00 → a00 → e01 → a01 → e00：

```
a20-eth3<->c0-eth3 (OK OK)
a20-eth4<->c1-eth3 (OK OK)
a21-eth3<->c2-eth3 (OK OK)
a21-eth4<->c3-eth3 (OK OK)
a30-eth3<->c0-eth4 (OK OK)
a30-eth4<->c1-eth4 (OK OK)
a31-eth3<->c2-eth4 (OK OK)
a31-eth4<->c3-eth4 (OK OK)
```

图 3.6

采用 STP 协议进行修复，他会选举一个根桥，计算每个转换机刀根桥的对端路径，阻塞冗余路径的端口，消除环路，保留无环的生成树拓扑：

```
sh ovs-vsctl set Bridge <交换机名> stp_enable=true
```

修改命令后，主机的通信得到恢复：

```
mininet> h1 ping -c 3 h2
PING 10.1.1.2 (10.1.1.2) 56(84) bytes of data.
64 bytes from 10.1.1.2: icmp_seq=1 ttl=64 time=16.5 ms
64 bytes from 10.1.1.2: icmp_seq=2 ttl=64 time=0.043 ms
64 bytes from 10.1.1.2: icmp_seq=3 ttl=64 time=0.058 ms

--- 10.1.1.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2055ms
rtt min/avg/max/mdev = 0.043/5.517/16.450/7.730 ms
```

图 3.7

3.4 自学习和环路检测实验

3.4.1 实验简介

本实验在 OpenFlow/OS-Ken 控制器与 Mininet 仿真环境构建的软件定义网络（SDN）中进行，要求在给定的网络拓扑基础上修改控制器代码，以实现交换机的自学习以及防治环路广播。通过本次实验，您将完成以下任务：完善 self_learning_switch.py，实现交换机自学习；完善 loop_barrier_switch.py，利用端口禁用的方式防止环路广播；完善 loop_detecting_switch.py，利用转发历史信息防止环路广播。完成上述任务后，您将获得以下收获：熟悉并掌握网络工具的基本使用；熟悉并理解 SDN 的工作机制；了解 SDN 下的自学习与传统网络下的差异并实现 SDN 下的自学习；了解环路广播的形成原因，掌握在环路广播场景中的防治方法。

3.4.2 任务一：自学习交换机

（1）代码逻辑结构

Topo_1.py 的结构如下，四台交换机和四台主机构成，具体为：h1 连接到 s1，h2 连接到 s2，h3 连接到 s3，h4 连接到 s4，且 s1 作为核心交换机，与 s2、s3、s4 分别相连，其余三台交换机只与 s1 连接。该拓扑为星型结构，所有主机间的通信都需经过中心节点 s1，这种网络布局突出核心节点的管理作用。

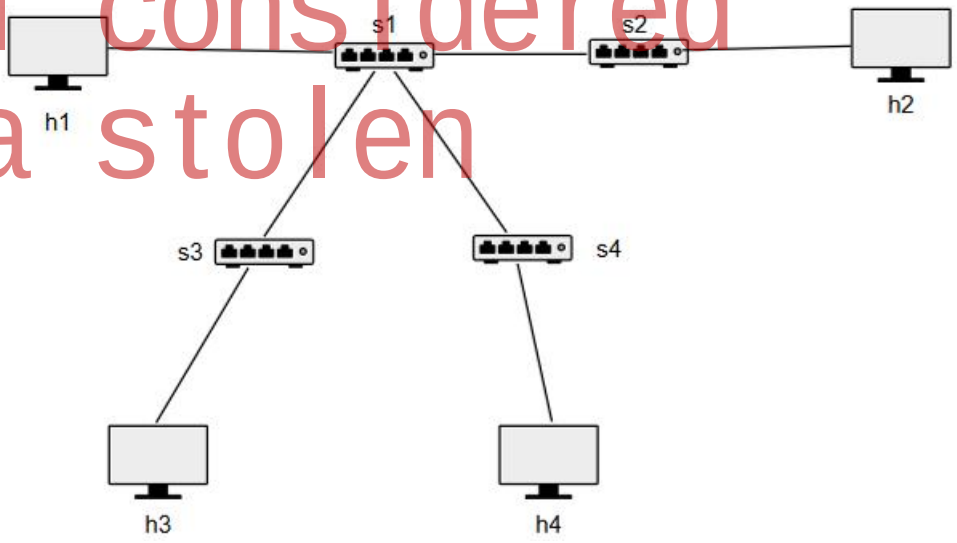

图 3.8

(2)代码实现

self_learning_switch.py 控制器维护 MAC 到端口的映射表。收到包时记录源 MAC 与入端口；若目的 MAC 已学习则下发精确流表并转发；否则洪泛。通过记录源 MAC 实现自学习，减

少控制器干预，具体如下：

```
from os_ken.base import app_manager
from os_ken.controller import ofp_event
from os_ken.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
from os_ken.controller.handler import set_ev_cls
from os_ken.ofproto import ofproto_v1_3
from os_ken.lib.packet import packet
from os_ken.lib.packet import ethernet
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 add_flow(self, datapath, priority, match, actions,idle_timeout=0,hard_timeout=0):
        dp = datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        mod = parser.OFPFlowMod(datapath=dp, priority=priority,
                                idle_timeout=idle_timeout,
                                hard_timeout=hard_timeout,
                                match=match,instructions=inst)
        dp.send_msg(mod)
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER,ofp.OFPCML_NO_BUFFER)]
        self.add_flow(dp, 0, match, actions)
```

```
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        # the identity of switch
        dpid = dp.id
        # the port that receive the packet
        in_port = msg.match['in_port']
        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        # get the mac
        dst = eth_pkt.dst
        src = eth_pkt.src
        # 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]
                       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)]
            self.add_flow(dp, 1, match, actions, hard_timeout=0)
            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)
```


（3）测试流程和问题分析

运行 self_learning_switch.py 与 topo_1.py，遵循下述步骤使用如下命令，对 h3-eth0 抓包，使用 h4 ping h2 后，发现 h3 从 ARP 后未收到过 h4 与 h2 之间通信的数据包，说明成功实现自学习交换机，如下图 3.9 所示

```
终端 1：bash cd lab2
        source .venv/bin/activate
         osken-manager self_learning_switch.py
终端 2：bash cd lab2
        sudo ./topo_1.py
在 终端 2 中，
        mininet> h3 wireshark &
        bash mininet> h4 ping -c 20 h2
```

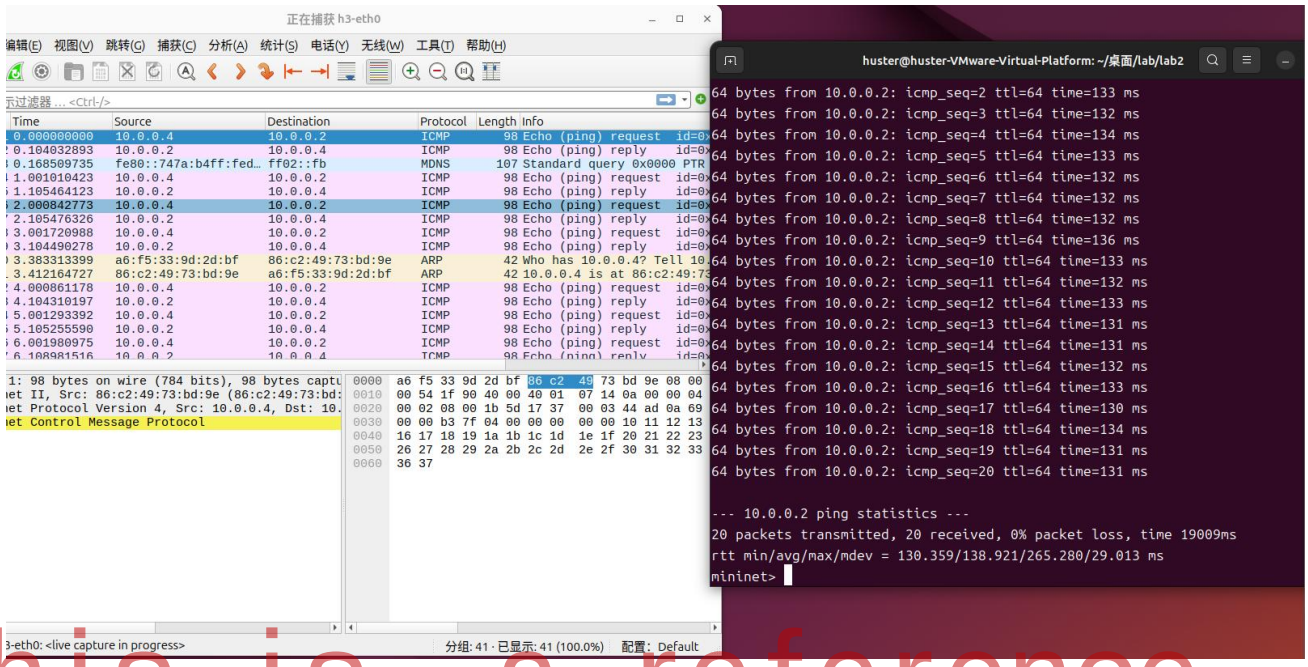

图 3.9

清除网络环境，重新运行 self_learning_switch.py 与 topo_1.py，遵循下述步骤使用如下命令，通过控制器输出如图 3.10 所示：

任意终端：sudo mn -c

终端 1：bash cd lab2

source .venv/bin/activate

osken-manager self_learning_switch.py

终端 2：bash cd lab2

sudo ./topo_1.py

在 终端 2 中，

mininet> h4 ping -c 10 h2

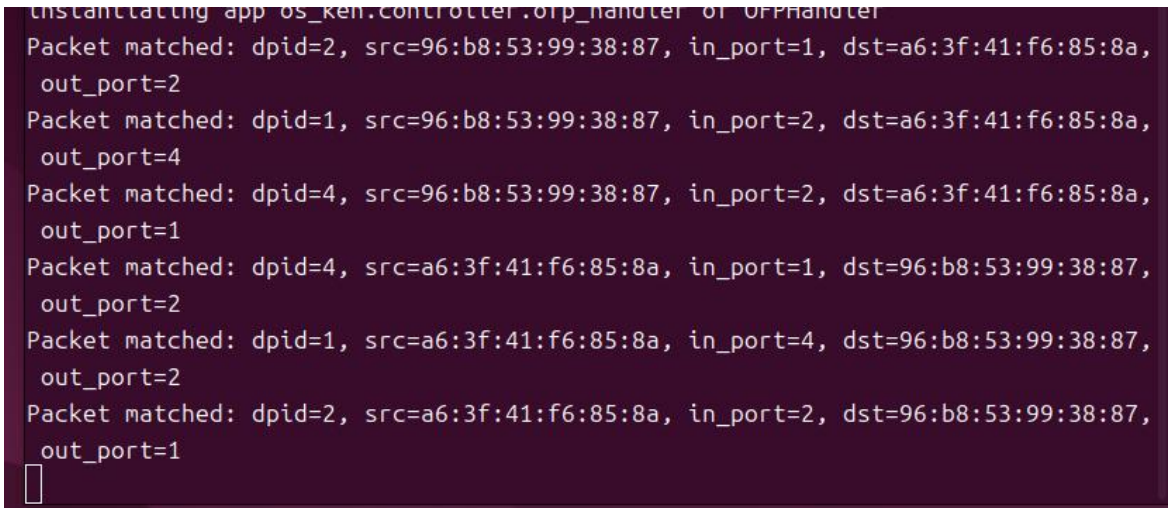

图 3.10

MAC 地址识别：h4 的 MAC 地址为 96:b8:53:99:38:87(依据：前三条去程包中 src=96:b8:53:99:38:87，表示 h4 发送 ping 请求)；h2 的 MAC 地址为 a6:3f:41:f6:85:8a（依据前三条去程包中 dst=a6:3f:41:f6:85:8a，以及后三条回程包中 src=a6:3f:41:f6:85:8a，表示 h2 作为目的和回复方）

通信路径分析：去程路径（h4→h2）：h4（MAC: 96:b8:53:99:38:87）从 s2 的端口 1 进入，从端口 2 转发到 s1；s1 从端口 2 接收，从端口 4 转发到 s4；s4 从端口 2 接收，从端口 1 转发到达 h2（MAC: a6:3f:41:f6:85:8a）。完整路径为：h4→s2(1→2)→s1(2→4)→s4(2→1)→h2，共经过 3 个交换机，4 跳。回程路径（h2→h4）：h2（MAC: a6:3f:41:f6:85:8a）从 s4 的端口 1 进入，从端口 2 转发到 s1；s1 从端口 4 接收，从端口 2 转发到 s2；s2 从端口 2 接收，从端口 1 转发到达 h4（MAC: 96:b8:53:99:38:87）。完整路径为：h2→s4(1→2)→s1(4→2)→s2(2→1)→h4，路径与去程对称，方向相反。涉及的交换机和端口：

- s2（dpid=2）：端口 1 连接 h4，端口 2 连接 s1
- s1（dpid=1）：端口 2 连接 s2，端口 4 连接 s4
- s4（dpid=4）：端口 2 连接 s1，端口 1 连接 h

清除网络环境，修改 self_learning_switch.py 中的 self.add_flow(dp, 1, match, actions, hard_timeout):声明中的 hard_timeout 参数，分别修改为 0（流表一直有效）和 5（流表 5 秒后失效），启动实验

任意终端：sudo mn -c

#修改代码：

```
self.add_flow(dp, 1, match, actions, hard_timeout=0)
self.add_flow(dp, 1, match, actions, hard_timeout=5)
```

#分别运行下面步骤：

终端 1：bash cd lab2

source .venv/bin/activate

osken-manager self_learning_switch.py

终端 2：bash cd lab2

sudo ./topo_1.py

步骤 3：在 终端 2 中，

mininet> h4 ping -c 20 h2

图 3.11 描述了 hard_timeout=0 时候的两个终端的信息，在 ping 20 个包（图中蓝色方框部分）的情况下，只在学习阶段（前几次 ping）有输出（图中黄色方框部分），之后因为流表永久有效，控制器不再输出因为流表永久有效，交换机直接根据流表转发，不经过控制器

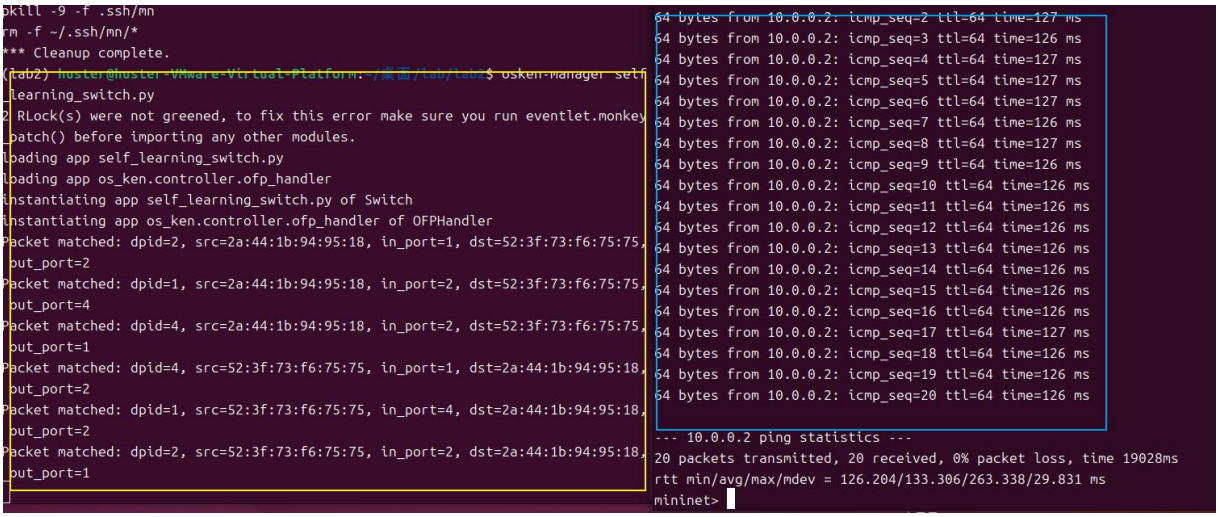

图 3.11

图 3.12 描述了 hard_timeout=5 时候的两个终端的信息，在 ping 20 个包（图中蓝色方框部分）的情况下，持续输出，每 5 秒重新输出匹配信息（图中黄色部分）.因为流表 5 秒后失效，需要重新学习,控制器持续介入转发过程

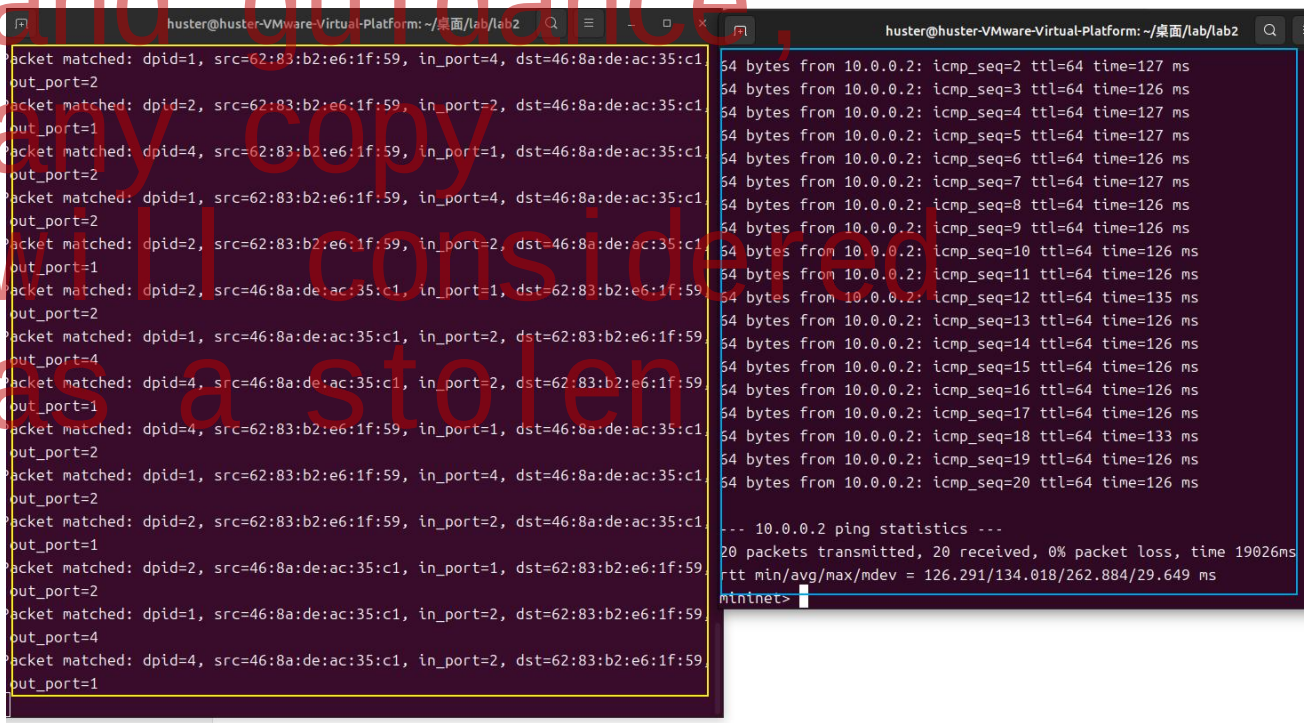

图 3.12

综上所述，hard_timeout=0（永久有效），流表一旦下发，永久保存在交换机中，交换机可以直接根据流表转发，不需要控制器参与；控制器只在首次学习时输出日志；性能好，延迟低；但流表不会自动过期，可能占用交换机资源。hard_timeout=5（5 秒后失效），流表在下发后 5 秒自动删除；5 秒后，数据包需要重新上送控制器；控制器需要重新学习并下发流表；

流表会自动清理，节省资源；需要控制器持续介入，增加延迟

3.4.2 任务二：禁用端口解决环路广播

（1）实验结构和背景：

运行 self_learning_switch.py 和 topo_2.py ，发现 h4 与 h2 之间无法正常通信

```
mininet> h4 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) 字节的数据。
来自 10.0.0.4 icmp_seq=1 目标主机不可达
来自 10.0.0.4 icmp_seq=2 目标主机不可达
来自 10.0.0.4 icmp_seq=3 目标主机不可达
来自 10.0.0.4 icmp_seq=4 目标主机不可达
来自 10.0.0.4 icmp_seq=5 目标主机不可达
来自 10.0.0.4 icmp_seq=6 目标主机不可达
^C
--- 10.0.0.2 ping 统计 ---
已发送 7 个包， 已接收 0 个包, +6 错误, 100% packet loss, time 6167ms
```

图 3.13

发现网络内的数据包数目和流表匹配次数巨大(通过观察 n_packets 参数可知)，这不符合正常通信的逻辑，对 h3 的 eth0 端口抓包,可以发现与本次通信无关的 h3 主机也收到了大量数据包，同时夹杂着大量的 ARP 数据包。为使 h4 和 h2 正常通信，通信前 h4 需在数据链路层层次广播 ARP 数据包来获取 h2 的 IP 地址与 MAC 地址的映射关系。比较 topo_1.py 与 topo_2.py 的网络结构，发现 topo_2.py 在 s3 和 s4 之间增加了一条链路，在 s1、s3、s4 之间形成了环路，广播的数据包会在环路中不断来回转发(称为环路广播)，淹没了正常通信的数据包，使得 h4 与 h2无法正常通信。另外，我们能在 h3 的抓包过程中，发现有重复的 ARP 数据包，观察 wireshark 内的 Info 字段可知它们的内容均为 ARP Request(10.0.0.4 在询问谁拥有 10.0.0.2)，这也佐证了环路广播


```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x0, duration=380.309s, table=0, n_packets=3290, n_bytes=138180, priority=1,in_port="s1-eth2",dl_d
st=fe:5c:ec:bf:66:16 actions=output:"s1-eth4"
 cookie=0x0, duration=388.975s, table=0, n_packets=668068, n_bytes=74009662, priority=0 actions=CONTROLLER
:65535
*** s2 ------------------------------------------------------------------------
 cookie=0x0, duration=380.571s, table=0, n_packets=3290, n_bytes=138180, priority=1,in_port="s2-eth1",dl_d
st=fe:5c:ec:bf:66:16 actions=output:"s2-eth2"
 cookie=0x0, duration=388.979s, table=0, n_packets=666119, n_bytes=73817384, priority=0 actions=CONTROLLER
:65535
*** s3 ------------------------------------------------------------------------
 cookie=0x0, duration=380.040s, table=0, n_packets=3288, n_bytes=138096, priority=1,in_port="s3-eth3",dl_d
st=fe:5c:ec:bf:66:16 actions=output:"s3-eth3"
 cookie=0x0, duration=388.982s, table=0, n_packets=668062, n_bytes=74016418, priority=0 actions=CONTROLLER
:65535
*** s4 ------------------------------------------------------------------------
 cookie=0x0, duration=380.202s, table=0, n_packets=3289, n_bytes=138138, priority=1,in_port="s4-eth2",dl_d
st=fe:5c:ec:bf:66:16 actions=output:"s4-eth3"
 cookie=0x0, duration=388.985s, table=0, n_packets=668039, n_bytes=74015442, priority=0 actions=CONTROLLER
:65535
```

图 3.14

（2）代码思路

loop_breaker_switch.py：控制器在交换机初始化时检测到 s1（dpid=1），主动禁用其连接 s3 或 s4 的指定端口（如端口 2），通过 OFPPortMod 消息将端口设为 DOWN，物理断开环路链路。同时保留自学习功能：维护 MAC-端口映射表，已学习则精确转发并下发流表，未学习则洪泛。通过主动禁用端口避免 ARP 广播在环路中无限循环，实现环路消除。

初始化：

设置控制器版本

创建字典 mac_to_port

创建标志 flag = 0（只执行端口禁用一次）

安装默认流表函数 add_flow(datapath, priority, match, actions, idle_timeout, hard_timeout):

构造指令 APPLY_ACTIONS(actions)

构造 FlowMod 消息，发送给交换机

当交换机与控制器建立连接 (SwitchFeatures) 时：

创建匹配条件：匹配所有

动作为输出到控制器

调用 add_flow 安装默认流表

当交换机上收到数据包 (PacketIn) 时：

取出 datapath、dpid、in_port

解析以太网头，忽略 LLDP/IPv6

得到源 MAC src、目的 MAC dst

若 dpid 为 1 且 flag 为 0：

发送 PortMod 禁用端口 target_port

flag = 1

日志记录端口已禁用

在 mac_to_port[dpid] 记录 {src -> in_port}

若 dst 已在 mac_to_port[dpid] 中：

out_port = mac_to_port[dpid][dst]

日志记录命中信息

构造匹配 (in_port, eth_dst=dst)

动作为输出到 out_port

调用 add_flow 下发匹配流

构造 PacketOut 将当前数据包转发到 out_port

否则：

动作为洪泛

构造 PacketOut 洪泛发送

Topo_2.py:该拓扑定义了一个包含环路的网络结构：s1 连接 s2、s3、s4，s3 也连接 s4，形成 s1-s3-s4-s1 的环路。四个交换机分别连接主机 h1、h2、h3、h4。使用 Mininet 的 RemoteController 连接到本地控制器（127.0.0.1:6633），链路带宽设置为 10Mbps。该拓扑用于测试环路检测和消除方案，通过禁用 s1 的某个端口可以打破环路，恢复网络正常通信。

函数 GeneratedTopo():

继承 Topo

创建交换机 s1-s4

创建主机 h1-h4

连接 s1-h1, s2-h2, s3-h3, s4-h4

连接交换机：

s1-s2, s1-s3, s1-s4, s3-s4（链路带宽 10）

topos 字典注册拓扑

函数 setupNetwork(ip):

topo = GeneratedTopo()

若 ip 为空 → ip = '127.0.0.1'

返回 Mininet(topo, RemoteController(ip,6633), 链路类型 TCLink)

函数 connectToRootNS(net, sw, ip, prefix, routes):

创建 root 节点 (不在命名空间)

用 TCLink 连接 root 与 sw，记录接口 intf

root.setIP(ip, prefix, intf)

net.start()

对 routes 中每条 route，执行 route add

函数 sshd(net, cmd='/usr/sbin/sshd', opts='-D'):

sw = net.switches[0]

调用 connectToRootNS(net, sw, '10.123.123.1', 8, ['10.0.0.0/8'])

对每个 host，执行 host.cmd(cmd + opts + '&')

dumpNodeConnections(hosts)

CLI(net)

CLI 退出后，停止每个 host 的 sshd

net.stop()

函数 start_network(net):

net.start()

dumpNodeConnections(hosts)

```
        CLI(net)
        net.stop()


    main:
        setLogLevel('info')
        调用 start_network(setupNetwork(controller_ip))
```

（3） 测试流程和问题分析

```
终端 1：启动控制器
    osken-manager self_learning_switch.py
终端 2：启动网络拓扑
    sudo ./topo_2.py
Mininet CLI 中:
# 尝试 ping（会失败）
mininet> h4 ping -c 3 h2
# 查看流表（数据包数量异常巨大）
mininet> dpctl dump-flows
mininet> h3 wireshark &
```


Ping 失败后我们查看流表，发现无法通信延迟极高，数据包流量异常巨大，启动抓包后会发现大量重复的 ARP Request 包，而控制器输出无法停止，ARP 广播包在环路中无限循环，淹没了正常通信的数据包。

```
Packet matched: dpid=1, src=fe:14:c3:6f:34:be, in_port=2, dst=52:7e:00:f6:f5:
 out_port=3
Packet matched: dpid=1, src=fe:14:c3:6f:34:be, in_port=2, dst=52:7e:00:f6:f5:
 out_port=4
Packet matched: dpid=1, src=fe:14:c3:6f:34:be, in_port=2, dst=52:7e:00:f6:f5:
 out_port=4
Packet matched: dpid=1, src=fe:14:c3:6f:34:be, in_port=2, dst=52:7e:00:f6:f5:
 out_port=4
Packet matched: dpid=1, src=fe:14:c3:6f:34:be, in_port=2, dst=52:7e:00:f6:f5:
```

图 3.15

观察 s1（dpid=1）的输出，发现 dpid=1, in_port=2, out_port=4 和 dpid=1, in_port=2, out_port=3 出现多次。s1 从 in_port=2(s2)接受数据包，转发到 out_port=3 和 out_port=4. 观察 s3 的记录 dpid=3, in_port=2 是从端口 2 接受数据包，s4 有数据 dpid=4, in_port=2 和 dpid=4, in_port=3，所以拓扑结构为 s1 从端口 2 转发到端口 3，s3 从端口 2 接受数据包，那么 s1 端口 3 链接 s3，同理，s1 端口 4 链接 s4，所以要使得通信成功，要打破换路，选择禁

用 3 端口和 4 端口，为了使得禁用端口有效，我们选择命令行禁用，最后结果如图 3.16：

```
1. 启动实验
osken-manager loop_breaker_switch.py  # 终端 1
sudo ./topo_2.py                      # 终端 2
2. 在 Mininet CLI 中手动禁用端口 3
mininet> sh ovs-ofctl mod-port s1 3 down
mininet> sh ovs-ofctl show s1  # 确认端口 3 显示 PORT_DOWN
mininet> h4 ping -c 10 h2     # 测试
3. 测试场景 2：恢复端口 3，禁用端口 4
mininet> sh ovs-ofctl mod-port s1 3 up
mininet> sh ovs-ofctl mod-port s1 4 down
mininet> sh ovs-ofctl show s1
mininet> h4 ping -c 10 h2
```



```
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
mininet> h4 ping -c 10 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=538 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.635 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.153 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.069 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.091 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.064 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.066 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.102 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.235 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.062 ms

--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9181ms
rtt min/avg/max/mdev = 0.062/53.978/538.310/161.443 ms
```

图 3.16

3.4.2 任务三：转发历史信息解决环路广播

（1） 核心思想和简单例子

环路判断逻辑：维护映射表 sw[(dpid, src_mac, dst_ip)] = in_port，记录每个 ARP 请求首次进入交换机的端口。如果同一个 ARP 请求从不同端口到达同一交换机，判定为环路并丢弃。

判断规则：key 不存在则首次收到，记录端口并转发；key 存在且端口相同则正常重复，继续处理；key 存在但端口不同则检测到环路，丢弃包。

当我们 h4 ping h2，存在环路 s1-s3-s4-s1：步骤 1，h4 发出 ARP Request "Who has 10.0.0.2?"；步骤 2，s1 从端口 3 首次收到，key = (1, 00:00:00:00:00:04, 10.0.0.2)，sw[key] 不存在，记录 sw[key] = 3，日志记录 "Recording ARP request: dpid=1, src=00:00:00:00:00:04, dst_ip=10.0.0.2, in_port=3"，然后洪泛转发；步骤 3，ARP 包在环路中传播 s1 → s3 → s4 → s1；步骤 4，s1 从端口 2 再次收到相同 ARP Request，key = (1, 00:00:00:00:00:04, 10.0.0.2)，sw[key] = 3 已存在，当前 in_port = 2，3 ≠ 2 检测到环路，日志记录 "Loop detected! Dropping ARP request: dpid=1, src=00:00:00:00:00:04, dst_ip=10.0.0.2, in_port=2 (previous port=3)"，直接丢弃包不转发，从而阻断环路，ARP Request 不再无限循环，正常通信可以建立。

（2） 指令测试与结果

运行指令后，数据直接成功发送（图 3.17）：

```
1. 启动实验
osken-manager loop_detecting_switch.py  # 终端 1
sudo ./topo_2.py                        # 终端 2
mininet> h4 ping -c 10 h2
```


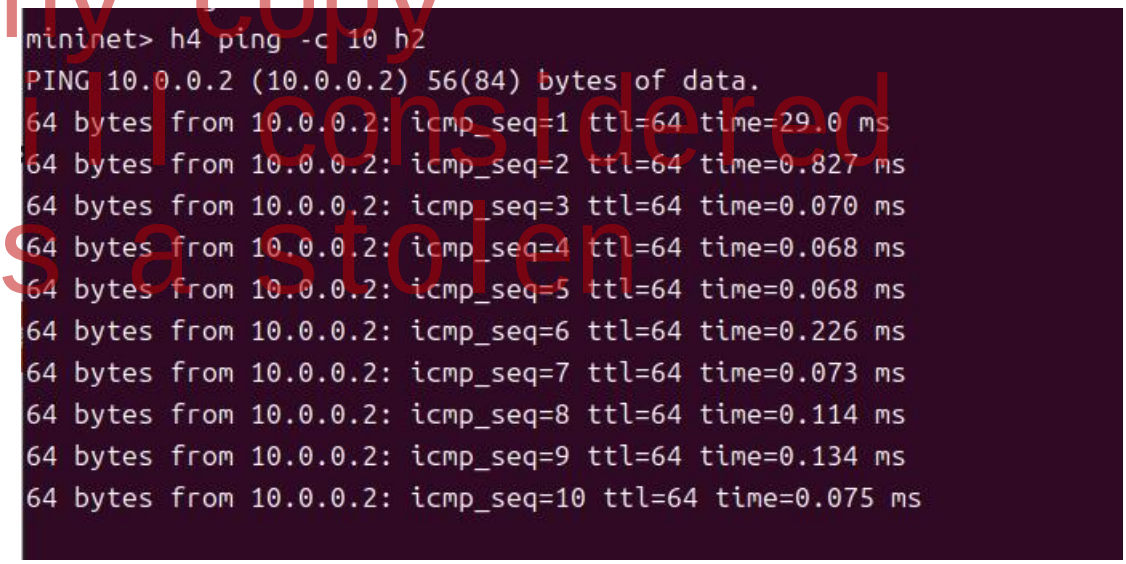

图 3.17

3.5 链路选择和故恢复实验

3.5.1 任务一：最少跳数路径

（1）任务解释和分析

`networkx.shortest_simple_paths` 是 NetworkX 中计算所有最短简单路径（无环）的生成器函数。在本实验中，通过 `network_awareness.py` 的 `shortest_path()` 方法调用，传入 `self.topo_map`（包含主机 IP 和交换机 DPID 的拓扑图）、源和目标主机 IP 地址，以及权重参数 `weight='hop'`（或 `'delay'`）。函数返回一个生成器，按路径长度（权重和）从小到大生成所有最短路径，通过 `list()` 转换为列表后取 `paths[0]` 作为最短路径。该函数的特点包括：返回简单路径（无重复节点）、支持多条等权重路径（`paths[1]`，`paths[2]` 为次短路径）、无路径时抛出异常（实验中捕获并记录日志）。在实际应用中，流程为：控制器通过 LLDP 发现网络拓扑并构建 NetworkX Graph，当主机发送数据包触发 PacketIn 事件时，调用 shortest_path()计算最短路径（如['10.0.0.2', 2, 4, 5, 9, '10.0.0.9']表示 h2→s2→s4→s5→s9→h9），然后根据路径计算端口信息并下发流表规则。

（2）实验结果

通过下面指令，得到的结果如下图 3.18

```
1. 启动环境（两个终端）
source .venv/bin/activate
2.启动实验
sudo ./topo.py    # 终端 1
uv run osken-manager least_hops.py --observe-links      # 终端 2
mininet> h2 ping h9
```

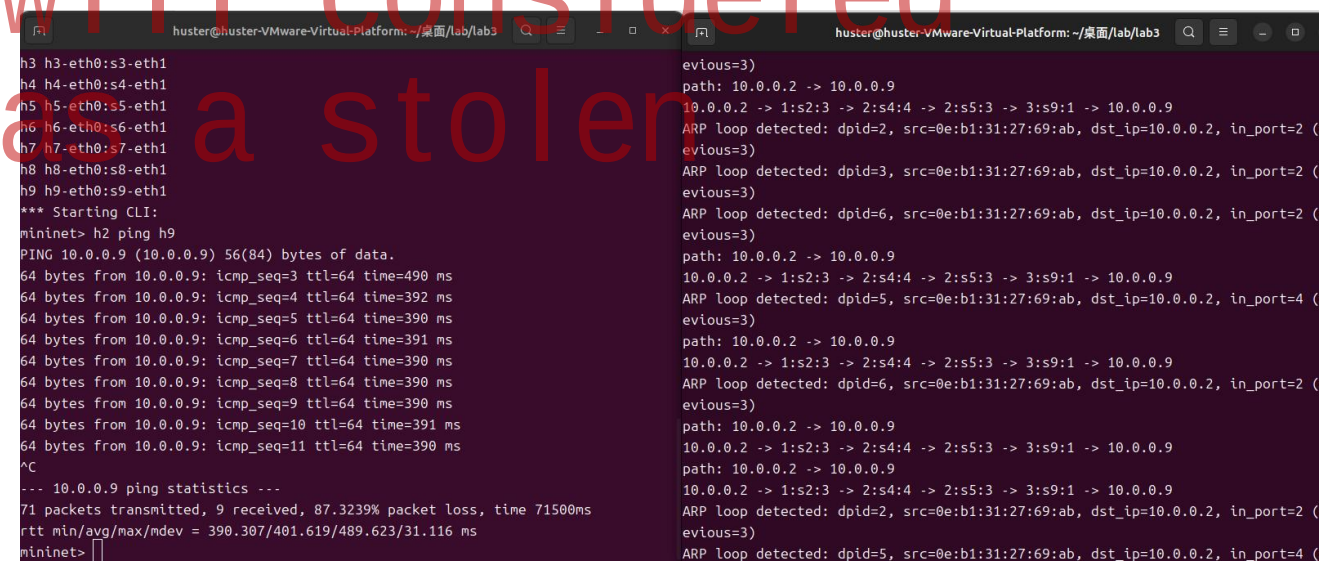

图 3.18

3.5.1 任务二：最少时延路径

（1）实现逻辑和整体结构

任务二的核心是实现基于链路时延的最短路径选择，主要分为三个部分：（1）时延测量，通过 LLDP 和 Echo 消息测量链路时延；（2）拓扑构建:将测量得到的时延作为边的权重属性添加到 NetworkX 图中；(3)路径计算:使用 NetworkX 的 `shortest_simple_paths` 函数，以 `delay` 为权重计算最小时延路径.链路时延的计算公式为：delay = max((T_lldp12 + T_lldp21 - T_echo1 - T_echo2) / 2, 0)

控制器启动后，NetworkAwareness 模块初始化，创建所需的数据结构（如 lldp_delay_table 和 echo_RTT_table），并分别启动拓扑发现线程与 Echo 测量线程。系统随后进入周期性测量阶段，通过 packet_in_handler 处理 LLDP 包以获取链路时延，通过 examine_echo_RTT 定期发送 Echo 包以测量 RTT。接着，在 _get_topology 函数中调用 calculate_link_delay 计算链路时延，并将结果添加到 topo_map 的边属性中。当主机发送数据包时，ShortestDelay 模块处理 IPv4 包，调用 shortest_path(src_ip, dst_ip, weight='delay')，计算路径中每条链路的时延及总路径时延和 RTT，最终输出结果并下发流表。

（2）遇到的问题

network_awareness.py：代码通过使用 .get(key, 0) 方法避免在异步链路发现和时延计算中因键缺失导致的 KeyError，如果键不存在则默认返回 0。随后分别获取链路两个方向的 LLDP 时延 lldp_12 和 lldp_21，以及两个交换机的 Echo 往返时延。根据公式 (T_lldp12 + T_lldp21 - T_echo1 - T_echo2) / 2 计算单向链路时延，并使用 max(delay, 0) 确保结果不为负值，从而保证计算的稳定性与有效性。

```
def calculate_link_delay(self, src_dpid, dst_dpid):
    try:
        lldp_12 = self.lldp_delay_table.get((src_dpid, dst_dpid), 0)
        lldp_21 = self.lldp_delay_table.get((dst_dpid, src_dpid), 0)
        echo_1 = self.echo_RTT_table.get(src_dpid, 0)
        echo_2 = self.echo_RTT_table.get(dst_dpid, 0)
        delay = (lldp_12 + lldp_21 - echo_1 - echo_2) / 2
        return max(delay, 0)
    except KeyError:
        return 0
```

shortest_delay.py：代码通过 range(len(dpid_path) - 1) 遍历路径中的所有边，包括主机到交换机的边。每次迭代使用 topo_map[src][dst].get('delay', 0) 获取链路的 delay 值，并根据节点类型格式化链路名称以提高可读性。随后将各链路时延累加得到总路径时延，计算往返时延为单向时延的两倍，输出包含每条链路时延、总路径时延和路径 RTT 的结果。

```
link_delay_dict = {}
path_delay = 0.0
for i in range(len(dpid_path) - 1):
    src = dpid_path[i]
    dst = dpid_path[i + 1]
    if self.network_awareness.topo_map.has_edge(src, dst):
        delay = self.network_awareness.topo_map[src][dst].get('delay', 0)
        # Format link name
        if isinstance(src, str):    # Host IP
            link_name = f"h{src.split('.')[-1]}->s{dst}"
        elif isinstance(dst, str):
            link_name = f"s{src}->h{dst.split('.')[-1]}"
        else:
            link_name = f"s{src}->s{dst}"
        link_delay_dict[link_name] = delay * 1000    # Convert to ms
        path_delay += delay
path_RTT = path_delay * 2
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)
```


（3）实现步骤

1.修改 PortData 类编辑:`.venv/lib/python3.13/site-packages/os_ken/topology/switches.py`

```
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
```

2.在 lldp_packet_in_handler 中计算 T_lldb,在同一文件中，找到 `lldp_packet_in_handler` 方法并修改：

```
@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
    # ========== 新增结束 ==========
    if not self.link_discovery:
        return
    msg = ev.msg
    try:
        src_dpid, src_port_no = LLDPPacket.lldp_parse(msg.data)
    except LLDPPacket.LLDPUnknownFormat:
        return
    # ========== 新增开始 ==========
    # 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 ==========
```

network_awareness.py,calculate_link_delay.py，shortest_delay.py 代码复杂，请看代码详情

（4）实验结果

经过下面的命令顺序，获取图 3.19 的结果

```
终端 1：启动拓扑
sudo ./topo.py
终端 2：启动控制器
uv run osken-manager shortest_delay.py --observe-links
mininet> h2 ping -c 10 h9

控制器输出示例：
Link: 2 -> 3, delay: 32.00000ms
Link: 2 -> 4, delay: 80.00000ms
Link: 6 -> 7, delay: 20.00000ms
Link: 8 -> 9, delay: 26.00000ms
path: 10.0.0.2 -> 10.0.0.9
```

10.0.0.2 -> 1:s2:3 -> 2:s4:2 -> 1:s5:3 -> 2:s6:1 -> 1:s7:2 -> 1:s8:2 -> 3:s9:1 -> 10.0.0.9

link delay dict: {'s2->s4': 80.0, 's4->s5': 110.0, 's5->s6': 30.0, 's6->s7': 20.0, 's7->s8': 24.0, 's8->s9': 26.0}

path delay = 290.00000ms

path RTT = 580.00000ms

Ping 输出对比：

64 bytes from 10.0.0.9: icmp_seq=1 ttl=64 time=270 ms ✓ 接近 path delay

64 bytes from 10.0.0.9: icmp_seq=2 ttl=64 time=268 ms

图 3.19

3.5.1 任务三：容忍链路故障

（1）实现逻辑

任务整体分三步：先由 NetworkAwareness 应用通过 LLDP 和 Echo 消息周期测量链路时延，维护拓扑图 topo_map 及 link_info、link_delay 等结构；ShortestDelay 控制器在收到 ARP/IPv4 PacketIn 时，利用 networkx.shortest_simple_paths 基于 delay 权重计算主机间最小时延路径，输出路径与时延并下发双向流表，同时保持 ARP 泛洪自学习；当链路端口状态变化触发 EventOFPPortStatus 时，控制器清空拓扑缓存、端口映射和流表并重新安装默认规则，使后续 PacketIn 触发新的路径计算，实现链路故障后的自动重路由。

（2）关键代码

network_awareness.py 中的 calculate_link_delay 把双向 LLDP 往返时间和控制器到交换机的 Echo RTT 结合起来计算链路单向延迟，并被 _get_topology 在扫描链路时调用，把 delay 写入 link_delay_table 与 topo_map。这样后续路径搜索就能直接基于实时测得的链路时延。

```
def calculate_link_delay(self, src_dpid, dst_dpid):
    try:
        lldp_12 = self.lldp_delay_table.get((src_dpid, dst_dpid), 0)
        lldp_21 = self.lldp_delay_table.get((dst_dpid, src_dpid), 0)
        echo_1 = self.echo_RTT_table.get(src_dpid, 0)
        echo_2 = self.echo_RTT_table.get(dst_dpid, 0)
```

```
            delay = (lldp_12 + lldp_21 - echo_1 - echo_2) / 2
            return max(delay, 0)
        except KeyError:
            return 0
```

shortest_delay.py，handle_ipv4 负责沿着 network_awareness.shortest_path 找到的延迟最小路径构建端口跳序列、累积每段链路的 delay，打印路径与 RTT，再下发正反方向的流表，同时即时记录拓扑和 link_info 不齐全的情况，以便链路变化时平滑过渡。

```
    def handle_ipv4(self, msg, in_port, src_ip, dst_ip, pkt_type):
        parser = msg.datapath.ofproto_parser


        dpid_path = self.network_awareness.shortest_path(src_ip, dst_ip,weight=self.weight)
        if not dpid_path:
            self.logger.warning(
                "No path found from %s to %s, topology may be re-discovering. "
                "Topo map edges: %d, link_info entries: %d",
                src_ip, dst_ip,
                len(self.network_awareness.topo_map.edges()),
                len(self.network_awareness.link_info)
            )
            return
        self.path=dpid_path
        port_path = []
        for i in range(1, len(dpid_path) - 1):
            in_key = (dpid_path[i], dpid_path[i - 1])
            out_key = (dpid_path[i], dpid_path[i + 1])
            if in_key not in self.network_awareness.link_info or out_key not in self.network_awareness.link_info:
            switch_in_port = self.network_awareness.link_info[in_key]
            switch_out_port = self.network_awareness.link_info[out_key]
            port_path.append((switch_in_port, dpid_path[i], switch_out_port))
        self.show_path(src_ip, dst_ip, port_path)
        link_delay_dict = {}
        path_delay = 0.0
        for i in range(len(dpid_path) - 1):
            src = dpid_path[i]
```

```
                dst = dpid_path[i + 1]
                if self.network_awareness.topo_map.has_edge(src, dst):
                    delay = self.network_awareness.topo_map[src][dst].get('delay', 0)
                    # Format link name
                    if isinstance(src, str):    # Host IP
                        link_name = f"h{src.split('.')[-1]}->s{dst}"
                    elif isinstance(dst, str):    # Host IP
                        link_name = f"s{src}->h{dst.split('.')[-1]}"
                    else:    # Switch to switch
                        link_name = f"s{src}->s{dst}"
                    link_delay_dict[link_name] = delay * 1000    # Convert to ms
                    path_delay += delay
            path_RTT = path_delay * 2
```


（3）实验结果

通过下面指令，进入实验验证，得到结果图 3.20，3.21，3.22，3.23 所示，实验通过

```
两个终端：
    source .venv/bin/activate
第一个终端：
    uv run osken-manager shortest_delay.py --observe-links
第二个终端：
    sudo ./topo.py
>mininet h2 ping -c 3 h9（第一次会全 fail 触发学习）
>mininet h2 ping -c 3 h9
>mininet link s6 s7 down
>mininet h2 ping -c 3 h9
>mininet link s6 s7 up
>mininet h2 ping -c 3 h9
```

```
mininet> h2 ping -c 3 h9
PING 10.0.0.9 (10.0.0.9) 56(84) bytes of data.
64 bytes from 10.0.0.9: icmp_seq=2 ttl=64 time=273 ms
64 bytes from 10.0.0.9: icmp_seq=3 ttl=64 time=271 ms

--- 10.0.0.9 ping statistics ---
3 packets transmitted, 2 received, 33.3333% packet loss, time 2045ms
rtt min/avg/max/mdev = 271.204/272.334/273.464/1.130 ms
mininet> link s6 s7 down
mininet> h2 ping -c 3 h9
PING 10.0.0.9 (10.0.0.9) 56(84) bytes of data.
64 bytes from 10.0.0.9: icmp_seq=2 ttl=64 time=373 ms
64 bytes from 10.0.0.9: icmp_seq=3 ttl=64 time=371 ms

--- 10.0.0.9 ping statistics ---
3 packets transmitted, 2 received, 33.3333% packet loss, time 2052ms
rtt min/avg/max/mdev = 370.748/371.850/372.952/1.102 ms
mininet> link s6 s7 up
mininet> h2 ping -c 3 h9
PING 10.0.0.9 (10.0.0.9) 56(84) bytes of data.
64 bytes from 10.0.0.9: icmp_seq=2 ttl=64 time=273 ms
64 bytes from 10.0.0.9: icmp_seq=3 ttl=64 time=271 ms

--- 10.0.0.9 ping statistics ---
3 packets transmitted, 2 received, 33.3333% packet loss, time 2054ms
rtt min/avg/max/mdev = 270.581/271.788/272.996/1.207 ms
mininet>
```

图 3.20 mininet 控制台输出


```
Path found: ['10.0.0.2', 2, 3, 4, 5, 6, 7, 8, 9, '10.0.0.9']
path: 10.0.0.2 -> 10.0.0.9
10.0.0.2 -> 1:s2:2 -> 2:s3:3 -> 3:s4:4 -> 2:s5:4 -> 2:s6:3 -> 2:s7:3 -> 2:s8:3 -
> 4:s9:1 -> 10.0.0.9
link delay dict: {'h2->s2': 0, 's2->s3': 17.044544219970703, 's3->s4': 14.312624
93133545, 's4->s5': 55.34029006958008, 's5->s6': 15.337347984313965, 's6->s7': 1
0.473728179931 64, 's7->s8': 12.594819068908691, 's8->s9': 13.586282730102539, 's
9->h9': 0}
path delay = 138.68964ms
path RTT = 277.37927ms
```

图 3.21 初始状态示例

```
Path found: ['10.0.0.2', 2, 3, 4, 5, 9, '10.0.0.9']
path: 10.0.0.2 -> 10.0.0.9
10.0.0.2 -> 1:s2:2 -> 2:s3:3 -> 3:s4:4 -> 2:s5:3 -> 3:s9:1 -> 10.0.0.9
link delay dict: {'h2->s2': 0, 's2->s3': 16.283273696899414, 's3->s4': 16.161561
012268066, 's4->s5': 55.45246601104736, 's5->s9': 100.49867630004883, 's9->h9':
0}
path delay = 188.39598ms
path RTT = 376.79195ms
ARP packet: dpid=2, src_ip=10.0.0.2, dst_ip=10.0.0.9, src_mac=ba:a6:16:e2:26:17,
 dst_mac=b2:1f:3b:6b:08:ac, in_port=1
ARP packet flooded from switch 2, port 1
ARP packet: dpid=3, src_ip=10.0.0.2, dst_ip=10.0.0.9, src_mac=ba:a6:16:e2:26:17,
 dst_mac=b2:1f:3b:6b:08:ac, in_port=2
ARP packet flooded from switch 3, port 2
```

图 3.21 s6 与 s7 之间链接发生故障时

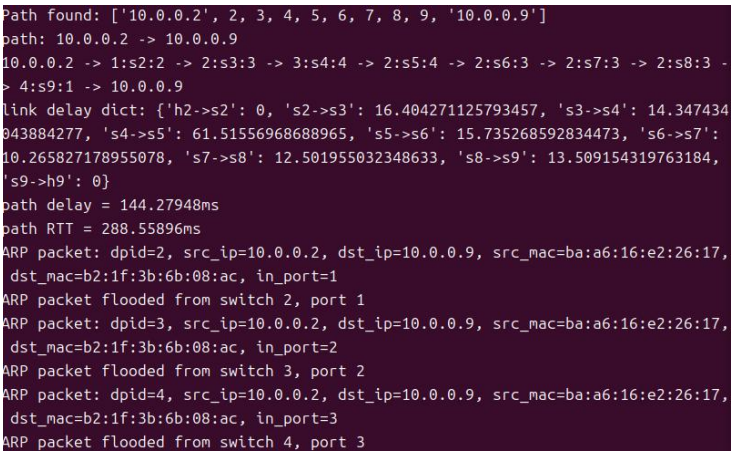

图 3.21 s6 与 s7 之间链接恢复时

（3） 遇到的问题和解决方法

1.控制器输出

```
Deleted all flows on switch 8
All flow entries deleted Port status changed on switch 7, port 2
Topology map cleared
Link info cleared
```

从日志看：事件已触发（Port status changed on switch 7, port 2）拓扑已重新发现，问题：拓扑图中没有 s6 -> s7 的链路（链路已断开），事件被多次触发（"Port status changed" 出现多次），导致重复清空；解决方法：添加了去重机制：记录每个端口状态变化的最后处理时间，如果同一端口在 2 秒内再次触发事件，则忽略，避免重复清空导致拓扑重建被打断

```
        port_key = (datapath.id, msg.desc.port_no)
        current_time = time.time()
        if port_key in self.last_port_status_time:
            time_diff = current_time - self.last_port_status_time[port_key]
            if time_diff < 2.0:
                return
        self.last_port_status_time[port_key] = current_time
```

2.控制器输出

```
All flow entries deleted All data structures cleared,
waiting for topology re-discovery...
Ignoring duplicate port status event for switch 7, port 2
Ignoring duplicate port status event for switch 7, port 2
```

问题原因，删除所有流表时，默认流表（priority=0，将数据包发送到控制器）也被删除，导致数据包无法到达控制器，因此控制器没有任何输出。解决方法：在 delete_all_flow() 中，

删除所有流表后立即重新安装默认流表，确保数据包能发送到控制器。

```
            match = parser.OFPMatch()
            actions=[parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCML_NO_BUFFER)]
            self.add_flow(datapath, 0, match, actions)
```

（4） 问题思考

解释为什么需要清空拓扑图、sw 这类对象，而不需要清空 lldp_delay_table 这类 记录 delay 的对象？

拓扑图、sw、mac_to_port 这类结构记录的是当前网络的“形状”和二层学习状态：链路断开或恢复后，旧的节点连边关系、端口映射以及 ARP/自学习缓存都可能失效，必须清空让控制器通过新的 PacketIn、LLDP 重新发现，避免沿着已断链路转发。lldp_delay_table、echo_RTT_table 等则是对链路物理特性的测量缓存；链路若还存在，它们的值仍然有效，保留可以加快恢复；如果链路真正消失，后续 LLDP/echo 不会再更新，路径搜索时也会因为拓扑缺少该边自动绕开，所以无需特意清空。

3.6 参考文献

[1] (美)James F. Kurose，(美)Keith W. Ross 著，陈鸣译. 计算机网络：自顶向下方法(原书 第 7 版). 北京：机械工业出版社,2018.5

[2] 华中科技大学计算机科学与技术学院. 基于 SDN 的软件定义实验指导手册

心得体会与建议

4.1 心得体会

这三次的实验自己动手做下来，对我的理论知识既有检验也有牢固的作用，纸上得来终究浅，期间我也查阅了很多资料，了解 Winsock 库的基本操作，探究三种流水线传输协议的区别，查询如何使用 SDN 实验中用到的软件等，这也极大锻炼了我的动手能力，每次做完一个实验都是一件令人开心的事情，感谢老师在我遇到困难的时候对我的指导！

4.2 建议

1. 强烈建议可以提供报告模版的 Latex 版本。
2. lab1：建议增加进阶拓扑案例，同时提供更多调试示例，引导学生深入理解胖树构建与排错关键点
3. lab2：可加入更复杂的环路场景，并给出详细抓包分析指导，帮助学生熟练掌握二层控制器调试思路。
4. lab3：建议扩展多故障情境训练，并辅以逐步排查指引，让学生在时延测量与路由切换中积累系统化经验。
5. lab2 中 TCP 是一个简单的版本，是在 GBN 的基础上修改，我希望以后可以增加难度，例如可以给接收方增加一个缓存的空间（因为实际生活中 TCP 一般是缓存失序的报文），这样可以模拟实际的生活的情况。