《计算机通信与网络》

实验指导手册

华中科技大学计算机科学与技术学院

二零二四 年 六 月

目录

第一章 实验环境安装和使用

1.1 实验要求

1. 请在第一次实验课时完成第一章实验环境安装与使用与第二章 FAT TREE 搭建

2. 完成下面实验任务 1.3-1.6，熟悉 mininet，OVS 和 wireshark 使用，实验课上请让助教检查命令行窗口

3，完成实验报告时请将指令对应截图

1.2 Mininet 介绍

Mininet 是一个强大的网络仿真工具，它可以在单台普通计算机（如笔记本电脑）上瞬间创建、交互并销毁一个包含主机、交换机、路由器和链路在内的虚拟软件定义网络。

简单来说，你可以把 Mininet 想象成一个“网络领域的虚拟机”。就像 VMware 或 VirtualBox 可以在一台机器上运行多个虚拟操作系统一样，Mininet 可以在一台机器上运行一个完整的虚拟网络。这个网络中的设备（主机、交换机）都是真实的、轻量级的进程，它们运行着真实的网络代码（如 Linux TCP/IP 协议栈、Open vSwitch 等），因此其行为与真实硬件网络高度一致。

Mininet 与软件定义网络（SDN）天生一对。它可以轻松地将虚拟交换机连接到外部的 SDN 控制器（如 OpenDaylight, ONOS, Ryu, POX 等），是学习和研究 SDN 原理、开发和测试控制器应用的事实标准工具。在 Mininet 中创建的主机运行着标准的 Linux 网络协议栈，你可以使用熟悉的命令行工具（如 ping, iperf, wget 等）进行测试。它支持真实的 Open vSwitch，可以执行复杂的流表操作，与生产环境中的 SDN 交换机行为一致。提供 Python API，允许用户通过编写 Python 脚本来自定义任何复杂的网络拓扑、配置链路参数（带宽、延迟、丢包率）并自动化测试流程。非常适合与持续集成（CI）工具集成，实现网络功能的自动化测试。

Mininet 巧妙地利用了 Linux 内核的以下特性来构建虚拟网络元素：1）进程隔离：网络命名空间。每个虚拟主机都被放置在一个独立的网络命名空间中。这意味着每台主机都有自己独立的网络接口、路由表、防火墙规则等，彼此隔离，就像不同的物理机器一样。2）虚拟交换机：Open vSwitch 或 Linux Bridge。Mininet 使用成熟的虚拟交换机（通常是 Open vSwitch）来连接各个主机。这些虚拟交换机功能强大，支持 OpenFlow 协议，可以被外部的 SDN 控制器管理。3）虚拟链路：虚拟以太网对。主机与交换机之间的连接是通过“veth pair”（虚拟以太网对）实现的。这就像一根虚拟的网线，两端分别连接到两个网络命名空间（或一个命名空间和交换机）。

1.3 虚拟机环境加载和相关资源

1. VMWare 下载

VMWare Workstation Pro/Fusion。VMWare 是闭源软件，仅可免费个人使用，下载前需要注册账号: https://support.broadcom.com/group/ecx/free-downloads。

2. 本实验提供虚拟机镜像文件（network.ovf），已配置 Mininet，导入镜像文件 network.ovf 并运行。**用户名和密码均为 huster**

1.4 Mininet 使用

1. Mininet 命令行使用：

- 右键启动命令行，选择在终端中打开

- 启动 mininet

```
# shell prompt
mn -h #  查看 mininet 命令中的各个选项
sudo mn -c #  不正确退出时清理 mininet
sudo mn #创建默认拓扑，两个主机 h1、h2 连接到同一交换机 s1
```

```
mininet@mininet-vm:~$ 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>
```

- 常用命令

```
# mininet CLI 中输入
nodes # 查看网络节点
links # 查看网络连接的情况
net # 显示当前网络拓扑
dump # 显示当前网络拓扑的详细信息
xterm h1 # 给节点 h1 打开一个终端模拟器
sh [COMMAND] # 在 mininet 命令行中执行COMMAND 命令
h1 ping -c3 h2 # 即 h1 ping h2 3 次
pingall # 即 ping all
h1 ifconfig # 查看 h1 的网络端口及配置
h1 arp # 查看 h1 的 arp 表
link s1 h1 down/up # 断开/连接 s1 和 h1 的链路
exit # 退出 mininet
```

```
mininet> nodes
available nodes are:
c0 h1 h2 s1
mininet> links
h1-eth0<->s1-eth1 (OK OK)
h2-eth0<->s1-eth2 (OK OK)
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
mininet>
```

2. 创建拓扑：

- 命令行拓扑

```
sudo mn –mac –topo=tree,m,n
```

--mac 指定 mac 地址从 1 开始递增，而不是无序的 mac，方便观察。

--topo 指定拓扑参数，可选用single 和 linear 等参数。

- 自定义拓扑（方式 1）：使用Mininet Python API 创建自定义拓扑（你需要自己创建这个文件），并通过命令行运行：

```
from mininet.topo import Topo


class MyTopo( Topo ):
    "Simple topology example."


    def build( self ):
        "Create custom topo."


        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )


        # Add links
        self.addLink( leftHost, leftSwitch )
        self.addLink( leftSwitch, rightSwitch )
        self.addLink( rightSwitch, rightHost )


topos = { 'mytopo': ( lambda: MyTopo() ) }
```

命令行运行：

```
    sudo mn --custom  拓扑文件名.py --topo mytopo --controller=none
```

- 自定义拓扑（方式 2）：使用Mininet Python API 直接创建并运行网络。

```
# sudo python topo_recommend.py
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.cli import CLI
from mininet.log import setLogLevel
```

```
class S1H2(Topo):
    def build(self):
        s1 = self.addSwitch('s1')
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        self.addLink(s1, h1)
        self.addLink(s1, h2)

def run():
    topo = S1H2()
    net = Mininet(topo)

    net.start()
    CLI(net)
    net.stop()

if __name__ == '__main__':
    setLogLevel('info') # output, info, debug
    run()
```

这种方式写法较为复杂，但简化了运行命令：

```
sudo python  拓扑文件名.py
```

3. 参考文档：

进一步学习可以参考 Mininet 官网：http://mininet.org/

1.5 Open vSwitch（OVS）开源多层虚拟交换机使用

1. 常用指令：

- 查看交换机的基本信息 以默认拓扑启动 mininet，打开新终端，输入

```
sudo ovs-vsctl show
```

```
mininet@mininet-vm:~$ sudo ovs-vsctl show
e7a21c84-4464-4b53-9d84-7ac031b48c46
    Bridge s1
        Controller "ptcp:6654"
        Controller "tcp:127.0.0.1:6653"
            is_connected: true
        fail_mode: secure
        Port s1-eth1
            Interface s1-eth1
        Port s1-eth2
            Interface s1-eth2
        Port s1
            Interface s1
                type: internal
    ovs_version: "2.13.1"
mininet@mininet-vm:~$
```

- 生成树协议

```
sudo ovs-vsctl set bridge s1 stp_enable=true #开启 STP，s1 为设备名
sudo ovs-vsctl get bridge s1 stp_enable
sudo ovs-vsctl list bridge
```

- 查看 mac 表

1）启动 mininet，注意禁用控制器，否则 mac 表可能学习不到内容

```
mininet@mininet-vm:~$ sudo mn --mac --topo=tree,2,2 --controller=none
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3
*** Adding links:
(s1, s2) (s1, s3) (s2, h1) (s2, h2) (s3, h3) (s3, h4)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller

*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet> nodes
available nodes are:
h1 h2 h3 h4 s1 s2 s3
mininet>
```

2）对每个交换机执行 sudo ovs-vsctl del-fail-mode-mode xx，否则 mac 表将仍然学习不到东西

```
mininet@mininet-vm:~$ sudo ovs-vsctl del-fail-mode s1
mininet@mininet-vm:~$ sudo ovs-vsctl del-fail-mode s2
mininet@mininet-vm:~$ sudo ovs-vsctl del-fail-mode s3
mininet@mininet-vm:~$
```

3）pingall 令所有主机发送数据包，防止沉默主机现象

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet>
```

4）sudo ovs-appctl fdb/show xx 查看 mac 表

```
mininet@mininet-vm:~$ sudo ovs-appctl fdb/show s1
 port  VLAN  MAC                Age
    1     0  00:00:00:00:00:02   91
    1     0  00:00:00:00:00:01   91
    2     0  00:00:00:00:00:04   91
    2     0  00:00:00:00:00:03   91
mininet@mininet-vm:~$ sudo ovs-appctl fdb/show s2
 port  VLAN  MAC                Age
    1     0  00:00:00:00:00:01  128
    2     0  00:00:00:00:00:02  128
    3     0  00:00:00:00:00:03  128
    3     0  00:00:00:00:00:04  128
mininet@mininet-vm:~$ sudo ovs-appctl fdb/show s3
 port  VLAN  MAC                Age
    1     0  00:00:00:00:00:03  130
    3     0  00:00:00:00:00:02  130
    3     0  00:00:00:00:00:01  130
    2     0  00:00:00:00:00:04  130
```

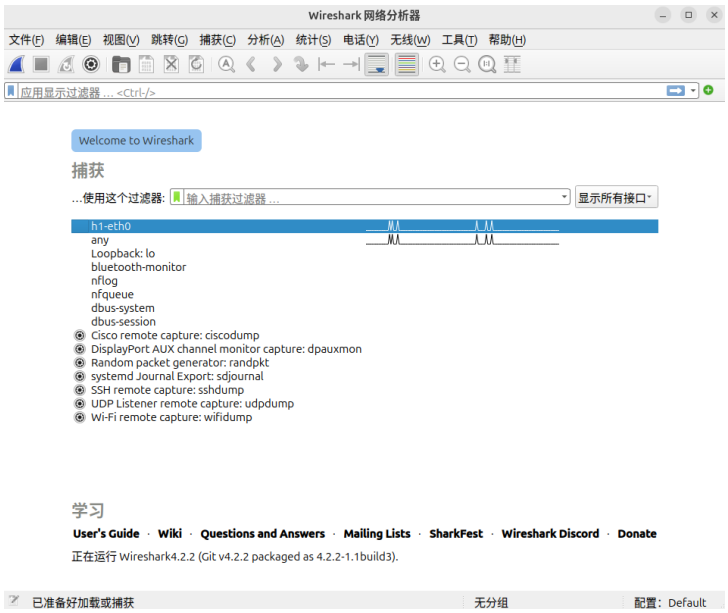
2. 参考文档：

ovs 的详细学习可参考官方网站 http://www.openvswitch.org/

1.6 wireshark 可视化

虚拟机上的 wireshark 可以直接抓取交换机的包。在 mininet CLI 中执行 主机名/交换机名 wireshark，则可以抓取主机/交换机的包，例如：

```
h1 wireshark
```



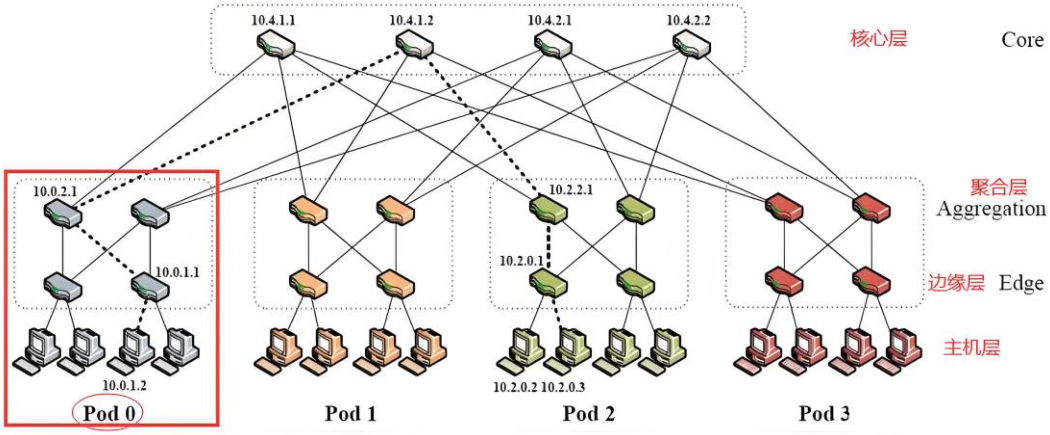
第二章 FAT TREE 搭建

2.1 实验要求：

1. 请在第一次实验课时完成本章内容
2. 本实验要求完成 fat tree 搭建，并使用 pingall 指令给出分析过程最后使得主机之间相互连通，禁止使用控制器
3. 实验后请完成实验报告，在报告中详细记录分析过程

2.2 fat tree 介绍

图示为一个参数 K=4 的 FatTree：



对于参数为 K 的 FatTree 有如下特征：

1. 网络架构中的基本组成单元 Pod
   - 有 K 个 Pod，每个 Pod 包含 2 层 Switch，每层 k/2 个 Switch。
2. 边缘层 Edge：其 Switch 有 K 个输出端口，
   - 前 k/2 个输出端口从左往右的顺序依次连接主机，
   - 后 k/2 个输出端口依次连接聚合层 Aggregation 的 Switch。
3. 聚合层 Aggregation：其 Switch 有 K 个输出端口，
   - 前 k/2 个输出端口从左往右的顺序依次连接边缘层 Edge 的 Switch，
   - 后 k/2 个输出端口依次连接核心层 Core 的 Switch，从左往右优先级每个 Switch 分配 k/2 个核心层 Switch。
4. 核心层 Core：有（k/2）^2 个 Switch，呈二维(j,i)阵列，
   - 每个 Switch 的 Port 编号连接对应 Pod，K 个 Port 从左往右顺序连接 K 个 Pod;
   - 具体到 Pod 中的 Switch 时，阵列第 j 行的 Switch 均连向聚合层的第 j 个 Switch。

5. 主机 Hosts
   - 不同 Pod 的 hosts 之间有(k/2)2 个最短路径,只使用其中一条，每条路有 5 跳（核心层有（k/2）2 个 Switch 选择）。

注意：

- 每个 Switch 交换机均相同为 K 个输入输出端口（降低网络成本）
- 支持在横向拓展的同时拓展路径数目。

2.3 实验任务

1. 使用 Mininet 的 Python API 搭建 k=4 的 fat tree 拓扑
2. 使用 pingall 查看各主机之间的连通情况
   - 若主机之间未连通，分析原因并解决（使用 wireshark 抓包分析）
   - 若主机连通，分析数据包的路径（提示：ovs-appctl fdb/show 查看 mac 表）

提示：

- wireshark 可视化分析和 mac 表查看在第一章中介绍

第三章 自学习与环路检测实验

3.1 自学习与环路检测实验基本介绍

本实验基于 OpenFLow/OS-Ken 控制器与 Mininet 仿真环境构建了软件定义网络(SDN)，你需要在已给定的网络拓扑中修改控制器代码，实现交换机自学习和防治环路广播，通过本次实验，你需要以下任务，具体要求将在后文中介绍：

- 完善 self_learning_switch.py，实现交换机自学习
- 完善 loop_breaker_switch.py，使用端口禁用防止环路广播
- 完善 loop_detecting_switch.py，使用转发历史信息防止环路广播

你将有以下收获：

- 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 下的自学习与一般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，掌握环路广播防治方法

3.2 基本操作

1. 包管理器 UV

本实验提供的镜像中已经提前下载了包管理器 UV，若无法正常使用，可以通过以下指令来重新下载：

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

UV 官网：uv

2. 安装依赖和初始化

本次实验对应文件包 lab2，依赖已经写入相关文件，同学们只需要运行 uv sync 进行同步即可，以下是常用 uv 指令供参考：

```
uv sync # 同步项目依赖到环境
# 其他常用的命令,未列出的命令可以使用 uv -h 或访问官网查看
uv init # 创建新 Python 项目
uv add # 为项目添加依赖
uv remove # 从项目移除依赖
uv lock # 为项目依赖创建锁文件
uv run # 在项目环境中运行命令(不会影响终端环境)
uv tree # 查看项目依赖树
```

3. 运行 OS-Ken App

启动 OS-Ken App

```
sudo ovs-ofctl dump-flows s1 #  在终端中打印交换机 s1 的流表
```

4. OVS 相关指令

- 查看交换机 s1 的流表

```
sudo ovs-ofctl dump-flows s1 #  在终端中打印交换机 s1 的流表
```

```
anka47@anka47:~$ sudo ovs-ofctl dump-flows s1
[sudo] password for anka47:
 cookie=0x0, duration=115.321s, table=0, n_packets=743389, n_bytes=52037230, priority=0 actions=CONTROLLER:655
35
```

- 查看所有交换机的流表

```
dpctl dump-flows #  在 mininet 终端中打印所有交换机的流表
```

```
mininet> dpctl dump-flows
*** s1 ------------------------------------------------------------------------
 cookie=0x0, duration=6.168s, table=0, n_packets=22883, n_bytes=1601810, priority=0 actions=CONTROLLER:65535
*** s2 ------------------------------------------------------------------------
 cookie=0x0, duration=6.175s, table=0, n_packets=22878, n_bytes=1601460, priority=0 actions=CONTROLLER:65535
*** s3 ------------------------------------------------------------------------
 cookie=0x0, duration=6.180s, table=0, n_packets=22883, n_bytes=1601810, priority=0 actions=CONTROLLER:65535
*** s4 ------------------------------------------------------------------------
 cookie=0x0, duration=6.186s, table=0, n_packets=22879, n_bytes=1601530, priority=0 actions=CONTROLLER:65535
mininet>
```

3.3 实验准备

1. 下载实验项目

本实验项目文件已经包含在了虚拟机桌面的 lab2 文件夹中，若有意外情况，可通过以下指令下载:

```
git clone https://github.com/cxy0629/sdn-lab2.git
```

2. 安装依赖和初始化

在 lab2 目录下

```
uv sync
```

3. 使用 python 虚拟环境运行控制器和网络拓扑

使用以下指令进入虚拟环境，执行后 Shell prompt 前会出现(lab2)的提示，表明你在'venv'虚拟环境中

```
source .venv/bin/activate
```

退出虚拟环境

```
deactivate
```

为了观察控制器和网络，你需要分别打开两个终端

- 运行控制器

```
osken-manager simple_switch.py
```

- 运行网络拓扑

```
sudo ./topo_1.py
```

注意：若出现代码无法运行，使用 chmod +x "代码文件路径"添加执行权限

3.4 简单交换机示例

```
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

class L2Switch(app_manager.OSKenApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(L2Switch, self).__init__(*args, **kwargs)

    def add_flow(self, datapath, priority, match, actions):
        dp = datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]

        mod=parser.OFPFlowMod(datapath=dp,priority=priority,match=match,instructions=i
    nst)

        dp.send_msg(mod)

  # add default flow table which sends packets to the controller
    @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_BU
```

```
            FFER)]
            self.add_flow(dp, 0, match, actions)

    # handle packet_in message
        @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

            actions = [parser.OFPActionOutput(ofp.OFPP_FLOOD)]



            out=parser.OFPPacketOut(datapath=dp,buffer_id=msg.buffer_id,in_port=msg.match[
            in_port'],actions=actions, data=msg.data)
            dp.send_msg(out)
```

该示例代码是本实验的基础代码，对应代码为 simple_switch.py，为了帮助同学们理解，对部分内容做出解释：

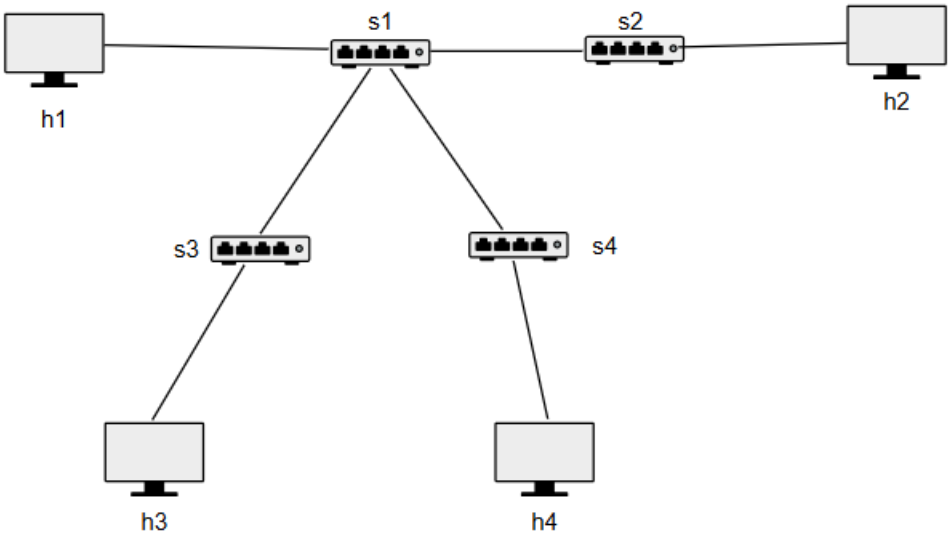
- SDN 网络内使用流表下发对交换机进行控制，对应 add_flow 函数，关键参数为交换机 id(datapath)，转发匹配原则(match)，对应转发行为(actions)
- switch_features_handler 函数用于在网络初始化交换机与控制器首次通信时下发默认流表，该流表的优先级最低，指示在无其他流表匹配情况下将数据包发送给控制器处理
- packet_in_handler 函数用于在网络初始化完成后交换机发送信息给控制器时对数据包的处理，在简单交换机示例中该处理逻辑非常简单，它对所有数据包采用洪泛，即从入端口以外的其他所有端口转发数据包，本函数也是后续实验的核心函数

3.5 任务一：自学习交换机

1. 问题描述

- 网络结构

对应 topo_1.py，如下图所示



- 简单交换机的缺陷

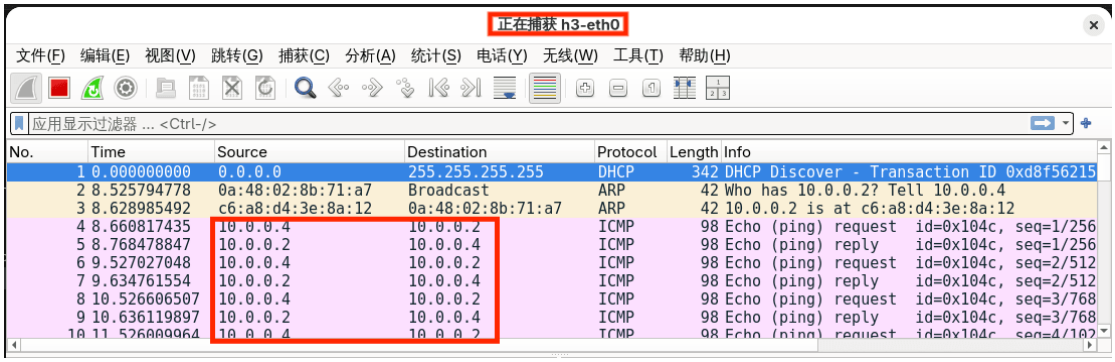
根据前文操作运行 simple_switch.py 与 topo_1.py，同时在 mininet 的 CLI 中启动 wireshark 对 h3 端口 eth0 抓包：

```
h3 wireshark &
```

继续在 mininet 的 CLI 中，h4 和 h2 进行通信：

```
h4 ping h2
```

观察 wireshark 中的抓包情况：



抓包结果显示控制器存在缺陷， 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

2. 实验要求

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，具体要求如下：

- 完善自学习逻辑：
  - 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，即 (dpid,mac)->port，可以使用嵌套字典来实现
  - 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

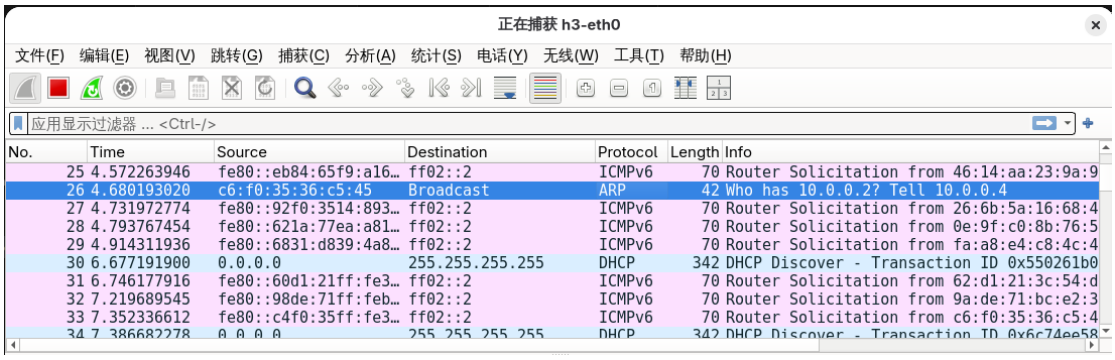
- 在 packet_in_handler 函数中，控制器先记录 (dpid,src_mac)->in_port 映射，再查询映射表中是否存在 (dpid,dst_mac) 的映射，若存在，向指定端口转发 并下发流表，否则洪泛数据包

- 观察控制器的输出并分析:
  - 在 packet_in_handler 函数内使用 self.logger.info()函数打印输出**映射表命中时**的转发信息，输出格式为 (dpid,src_mac,in_port,dst_mac,out_port) 五元组，内容将会被输出在控制器终端内
  - 使用 h4 ping h2，根据自学习逻辑，分析控制器输出内容，判断 h4 和 h2 的 mac 地址以及与它们相连交换机的端口号
  - 注意 add_flow 函数下发流表时默认硬超时参数(hard_timeout)为 0，在 packet_in_handler 函数内为流表下发指定 hard_timeout=5 ，观察控制器输出信息的规律，并比较与 hard_timeout=0 情形下的差别，思考原因

3. 结果验证

本节内容需要在报告和检查过程中展示，以确保你成功完成了实验:

- 运行 self_learning_switch.py 与 topo_1.py，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，如下图所示

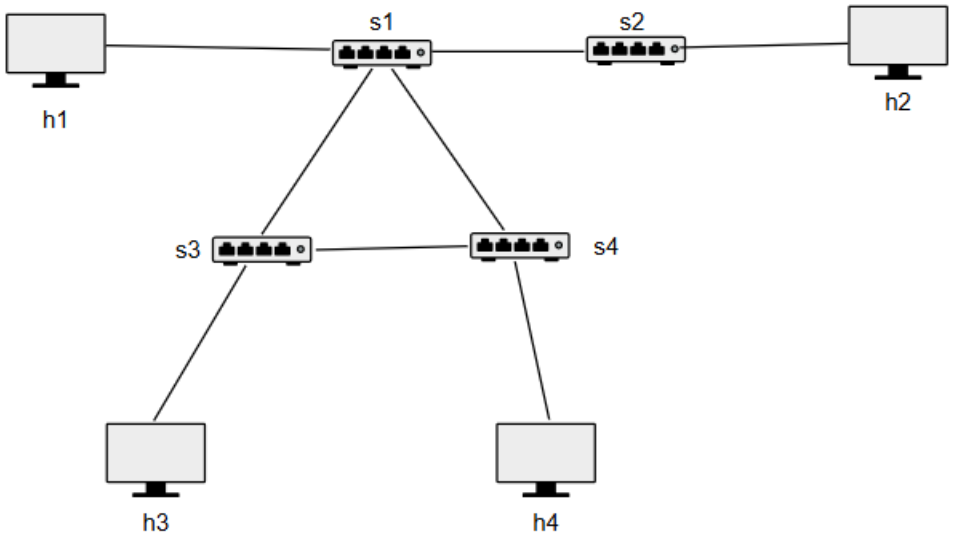


- 运行 self_learning_switch.py 与 topo_1.py，使用 h4 ping h2，通过控制器输出，指出 h4、h2 的 mac 地址和通信路径涉及的交换机以及端口号
- 分别设置 hard_timeout=5 与 hard_timeout=0，在两种场景下运行 topo_1.py 与 self_learning_switch.py，使用 h4 ping h2，指出控制器输出差异与规律并说明原因

3.6 任务二：禁用端口解决环路广播

1. 问题描述

- 对应 topo2.py，如下图所示



- 自学习交换机缺陷

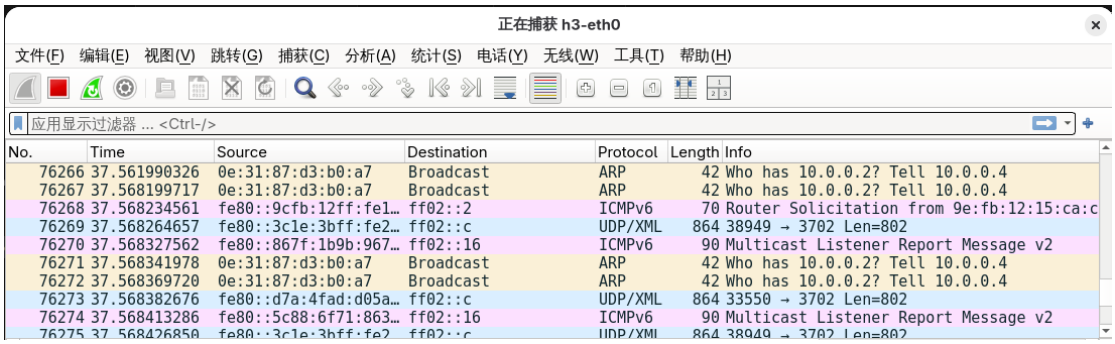
运行 self_learning_switch.py 和 topo_2.py ，发现 h4 与 h2 之间无法正常通信

```
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
```

查看流表信息

```
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
```

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，如下图所示



原因分析：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

2. 实验要求

在完成 self_learning_switch.py 的基础上，完善 loop_breaker_switch.py，实现使用禁用端口解决环路广播**(保留 3.5 节中的映射表命中时输出逻辑，同时为了方便观察控制器输出，使用默认 hard_timeout=0)**

- 实验思路：网络结构中的环路包含三条链路，分别为(s1,s3)、(s1,s4)、(s3,s4)，断开任意一条链路 的连接能够消除环路且不影响网络结构的连通性，可以通过禁用交换机端口来实现链路断连。理论上禁用三条链路的六个端口中的任意一个即可，但为了更清楚地观察禁用端口后对通信路径的影响，统一要求禁用交换机 s1 的端口
- 实验步骤：
  - 运行 self_learning_switch.py 和 topo_2.py，执行 h4 ping h2，观察控制器的输出，判断交换机 s1 与 s3、s4 相连的端口号**(可能因网络状况不同，需要多尝试几次才能确定两个端口)**
  - 完善 loop_breaker_switch.py 代码，在 packet_in_handler 函数内使用 OFFPortMod 消息修改交换机的端口配置**(注意仅需修改一次)**，消息参数如下图所示

字段名	说明
datapath	目标交换机的datapath对象（标识要配置的交换机）
port_no	要修改的端口编号（如3表示第3个端口）
hw_addr	端口的硬件MAC地址（端口的唯一标识）
config	定义端口配置的比特掩码（如OFPPC_PORT_DOWN表示禁用端口）
mask	指定哪些config字段需要更新（限制仅修改特定属性）
advertise	端口对外宣告的能力比特掩码（如OFPPF_16B_FD表示支持1Gbps全双工）

- 分别禁用 s1 与 s3 和 s1 与 s4 相连的端口，执行 h4 ping h2 ，比较控制器的输出变化，分析原因

3. 结果验证

本节内容需要在报告和检查过程中展示，以确保你成功完成了实验：

- 运行 self_learning_switch.py 和 topo_2.py，执行 h4 ping h2，观察控制器的输出，指出 s1 与 s3、s4 连接的端口号
- 运行 loop_breaker_switch.py 和 topo_2.py，确保在禁用 s1 两个不同端口的情况下，执行 h4 ping h2 能正常通信，并分别指出对应控制器输出内容的含义

3.7 任务三：转发历史信息解决环路广播

1. 问题描述

本节要解决的问题背景与 3.6 节完全相同，但 3.6 节的方法需要事先知道网络的具体端口信息，不具有一般性，且破坏了网络本身结构。本节将介绍一种更一般性的方法解决环路广播

2. 实验要求

在完成 self_learning_switch.py 的基础上，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

3. 结果验证

本节内容需要在报告和检查过程中展示，以确保你成功完成了实验：

- 运行 loop_detecting_switch.py 和 topo_2.py，确保执行 h4 ping h2 后正常通信
- 简要说明环路判断逻辑，并举一个简单例子说明

第四章 链路选择与故障恢复实验

4.1 链路选择与故障恢复实验基本介绍

在软件定义网络（SDN）中，控制器需要实时掌握网络拓扑和链路状态，才能进行高效的路由与流量调度。本实验围绕 **LLDP（链路层发现协议）** 与 **OpenFlow 控制器事件机制** 展开，重点训练学生如何：

- 通过 LLDP 报文实现链路发现，理解控制器如何感知网络拓扑
- 利用 LLDP 与 Echo 报文测量链路时延，构建带权拓扑图
- 在拓扑变化（如链路故障、恢复）时，动态更新路径选择策略
- 在 OS-Ken 控制器框架下，编写应用程序实现最少跳数路径、最小时延路径等功能

通过实验，学生将加深对 **SDN 控制与转发分离** 的理解，掌握**事件驱动编程**在网络控制器中的应用方法，并具备初步的网络测量与容错能力。

4.2 链路选择与故障恢复实验概览

1. **实验文件：**

   实验文件存放在虚拟机桌面的文件夹 lab3 中。

2. **基础准备(不需要写入实验报告）**
   - 熟悉 Mininet 的基本使用方法，能够搭建并运行简单的拓扑。
   - 掌握 OS-Ken 控制器的运行方式，理解 --observe-links 参数的作用。
3. **实验任务**
   - **任务一：时间基于最少跳数的路径选择**
     - 阅读并运行 least_hops.py，理解 least_hops.py 如何调用 network_awareness.py 获取 topo 图。
     - 能够解释 networkx.shortest_simple_paths 的使用方法。
   - **任务二：实现基于最小时延的路径选择**
     - 在新的控制器文件中实现基于链路时延的最短路径选择。
     - 正确实现 LLDP 与 Echo 报文的时延测量，并在拓扑图中维护 delay 属性。
     - 输出从 h2 到 h9 的最小时延路径及总时延，并用 ping RTT 验证。
   - **任务三：在任务二的基础上实现对链路故障的容忍**
     - 能够捕获链路断开与恢复事件，删除相关流表项。
     - 在链路故障时，控制器能重新选择新的最优路径，保证通信不中断。
4. **实验报告要求**
   - 包含实验的目的和主要内容
   - 包含子任务的报告
   - 包含完成本实验的思考（如收获、心得、感悟、看法、不足等）
   - 报告应条理清晰，逻辑完整，避免仅贴代码

4.3 提示

- **实验建议**:
  - **任务一**:
    - least_hops.py 依赖于 network_awareness.py
    - network_awareness.py 已实现获取 topo 的功能，需重点关注 topo_map.add_edge 方法（添加了哪些 属性 用于最短路径的计算）
    - networkx.shortest_simple_paths API
  - **任务二**:
    - **关于时延测量**：若出现负值，应取 0，避免影响路径计算。
    - **关于 topo 图**：不要忘记为 topo 图中的边添加相关的属性
    - **关于最短路径的计算**：不要忘记修改 networkx.shortest_simple_paths 计算所使用的属性
  - **任务三**:
    - **关于链路故障**：链路状态变化会触发 EventOFPPortStatus，需要在事件处理函数中**清除旧的信息（拓扑、转发表、sw 表等）**并**删除相关流表**。
- **代码修改和调试建议**:
  - 使用搜索引擎或 AI Chat 获取相关知识
  - 使用 vscode（或其他编辑器）的搜索功能，快速定位至目的代码处
  - 使用 ctrl+click，快速跳转至 function 和 class 的定义（需要按照对应语言的插件）
  - 使用 git 的版本管理功能记录修改，避免遗忘和混淆
  - 在代码中增加日志输出，帮助理解控制器的运行过程

4.4 LLDP 相关知识

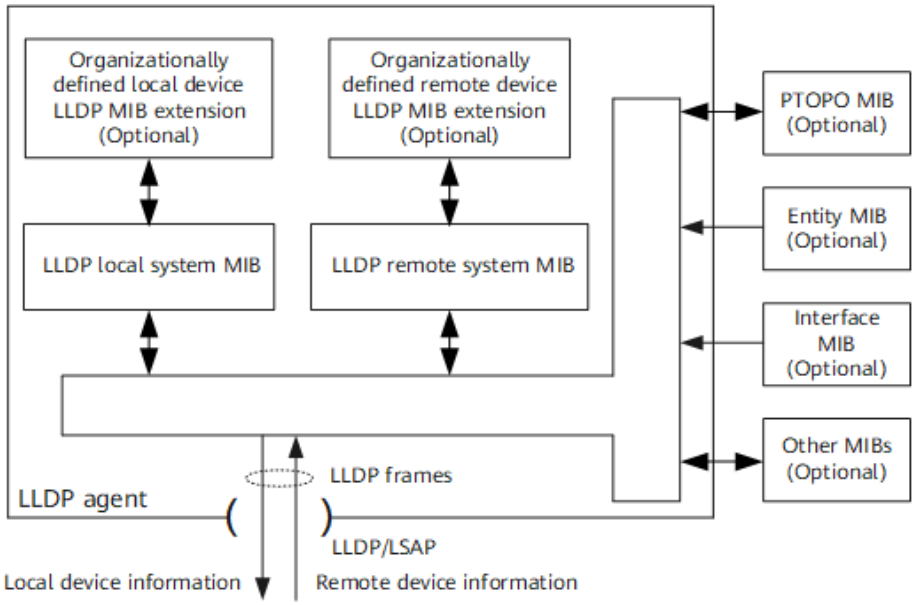
1. LLDP（链路层发现协议）

LLDP（链路层发现协议）是一种标准协议，用于在网络中自动发现邻居设备。它会定期发送包含设备信息的数据包（LLDPDU），内容包括设备名称、接口、管理地址和功能等。

这些信息以 TLV 格式封装，发送给直接连接的设备。接收方会将这些信息存入远程管理信息库（Remote MIB），而本地设备的信息则保存在本地管理信息库（Local MIB）。

LLDP 通过多个 MIB 模块（如 PTOPO MIB、Entity MIB、Interface MIB 等）维护本地信息，并同步更新远端信息，从而帮助网络管理系统了解设备连接情况和网络拓扑。

如下图所示:



2. OpenFlow 中 LLDP 处理流程（拓扑发现）

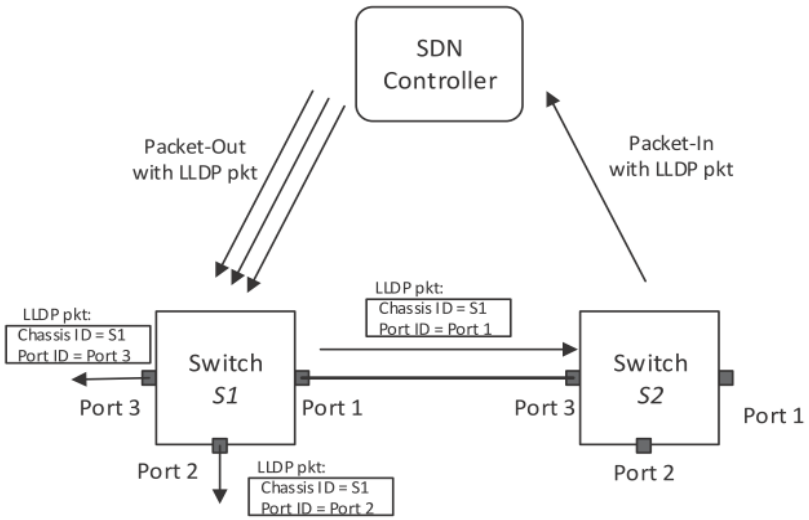
在 SDN（软件定义网络）中，网络控制器负责集中管理所有交换机，因此交换机的 LLDP 数据包发送和接收都是由控制器指令驱动完成的。

假设有两个 OpenFlow 交换机 S1 和 S2 都连接到了控制器，下面以 S1 为例，介绍控制器如何通过 LLDP 实现网络拓扑发现：

1) 控制器构造 LLDP 数据包并发送
2) 控制器下发流表到 S1，指示交换机将从**控制器端口**接收到的 LLDP 数据包，发送到指定的物理端口
3) 控制器下发流表到 S2，指示交换机将从**非控制器端口**接收到的 LLDP 数据包，打包成 PacketIn 消息发送回控制器
4) 控制器解析 LLDP 数据包

控制器收到 S2 发来的 PacketIn 消息后，解析其中的 LLDP 数据包：

- 从中提取出源交换机（S1）和源端口的信息
- 结合 PacketIn 消息中的接收端口信息，得知目的交换机（S2）和目的端口



注意：LLDP 无法发现主机（Host）

在 SDN 网络中，虽然 LLDP 能帮助控制器发现交换机之间的连接关系，但它**无法发现普通主机（如 PC 或服务器）**，原因主要有以下两点：

1) **主机不发送 LLDP 报文**。普通主机通常不会运行 LLDP 协议，因此不会主动发送 LLDP 数据包。
2) **LLDP 是单向广播，不要求回应。**交换机发送 LLDP 报文是单向的，只负责广播，不会要求接收方回应。即使主机收到了 LLDP 报文，也不会反馈任何信息给交换机或控制器。

因此，如果主机没有主动发起通信（例如发送 ARP 请求或 Ping 报文），控制器就不会收到任何与该主机相关的消息，也就无法识别它的存在。这种现象被称为 **“沉默主机现象”**，是网络拓扑发现中的一个常见挑战。

4.5 任务一：最少跳数路径

在网络路径选择中，一种常见的作法是选取最少跳数的路径，以获取较低的延迟。在 network_awareness.py 和 least_hops.py 中，我们实现了基于**最少跳数**的路径计算。整体思路可以分为三个步骤：**获取网络拓扑**、**计算最短路径**、**下发流表规则**

1. 任务要求

- 阅读相关数据结构的源码（补充部分给出了源码位置）
- 阅读并运行 least_hops.py，理解 least_hops.py 如何调用 network_awareness.py 获取 topo 图。
- 能够解释 networkx.shortest_simple_paths 的使用方法。(阅读文档 networkx.shortest_simple_paths API)
- 参考第三章中的实验，解决 arp 环路洪泛问题。

2. 任务内容

1) 获取相关信息

控制器首先需要掌握网络的拓扑结构，才能进行路径计算和流表下发。拓扑信息主要包括：主机（host）、链路（link）、交换机（switch）。**network_awareness.py 中获取网络拓扑的方式如下：**

```
def _get_topology(self):
    _hosts, _switches, _links = None, None, None
    while True:
        hosts = get_all_host(self)
        switches = get_all_switch(self)
    links = get_all_link(self)
```

补充：

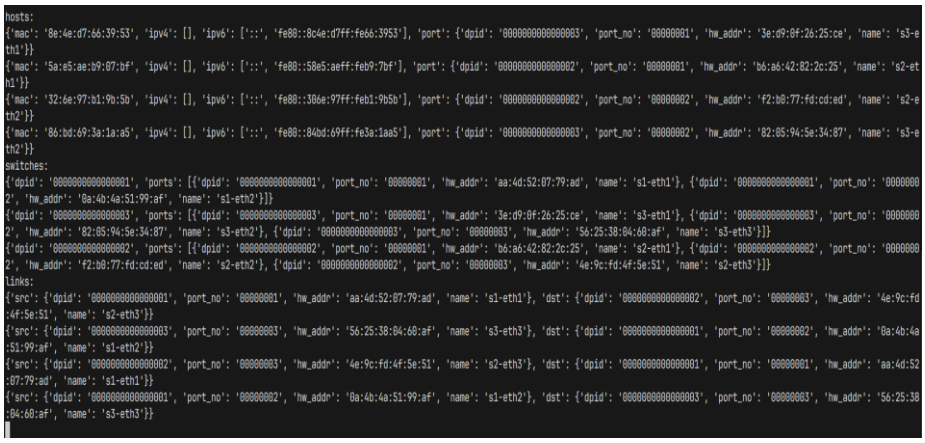
- 相关数据结构(class host, class switch, class link)见文件：.venv/lib/python3.13/site-packages/os_ken/topology/switches.py

- 示例 show_topo.py: 通过 get_all_host、get_all_link、get_all_switch 获取 host、link、switch 并打印至控制台。可以运行 show_topo.py 来进一步了解相关的数据结构。
- 运行命令:

```
# run command in Bash 1
sudo mn --topo=tree,2,2 --controller remote
```

```
# run command in Bash 2
uv run osken-manager show_topo.py --observe-links
                                    # --observe-links 表示启用链路观察功能
```

- 运行结果示例（ctrl+滚轮放大查看）



2) 建立拓扑图

在 network_awareness.py 中，控制器会将主机和链路信息加入到拓扑图中。

简言之，就是遍历获取到的 hosts 和 links，向 topo_map 中添加 hosts 与 switch 之间的边和 switch 与 switch 之间的边。

具体代码在 network_awareness.py 中的第 88 行和第 109 行附近。具体使用的函数为 self.topo_map.add_edge()。

通过这样，控制器就能维护一张完整的网络拓扑图。

3) 计算最短路径

i. 使用 Networkx 计算最短路径

在 network_awareness.py 中，调用 networkx.shortest_simple_paths 来计算最少跳数路径。具体代码如下:

```
def shortest_path(self, src, dst, weight='hop'):
try:
        paths = list(nx.shortest_simple_paths(self.topo_map, src, dst, weight=weight))
        return paths[0]
except:
```

```
self.logger.info('host not find/no path')
```

ii. 处理 IPv4 报文

具体代码见 least_hops.py 中的 handle_ipv4()

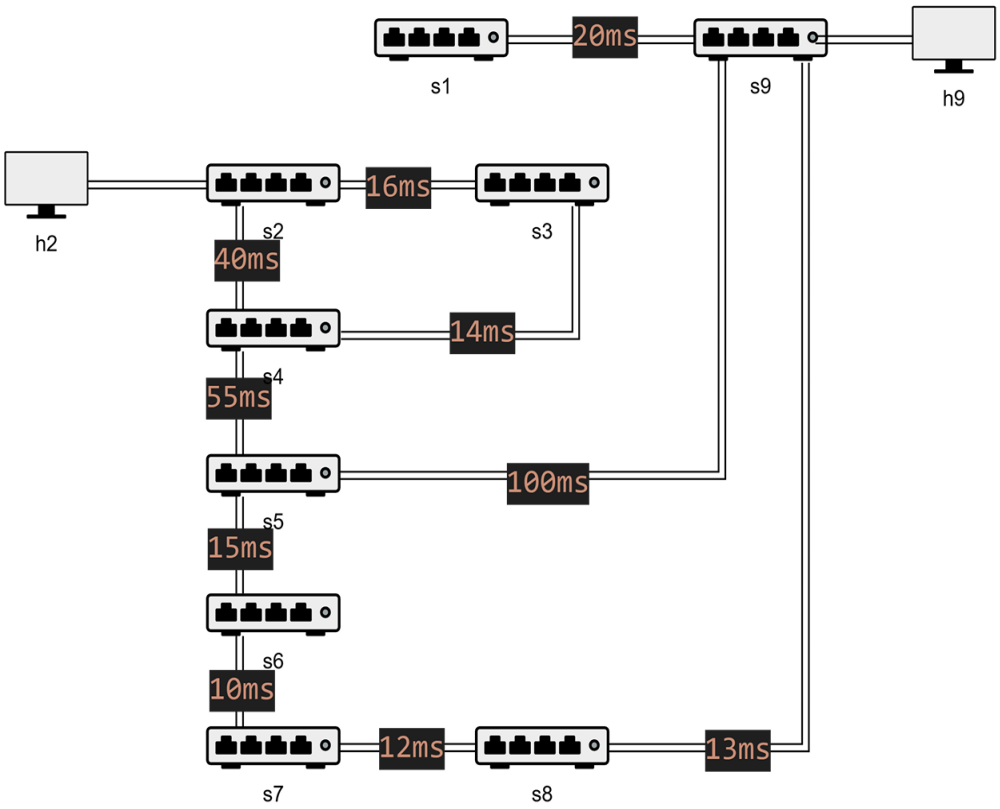
iii. 处理 arp 环路洪泛问题

本实验未处理 arp 环路洪泛问题，未实现自学习交换机。请参考第三章对 least_hops.py 中的 handle_arp()进行补全。建议使用 (dpid, src_mac, dst_mac) -> in_port 的方法进行处理。

注意，添加流表时，**务必指定该流表对应的包类型和流表的生命周期**。否则可能会导致数据包沿 arp 发现路径进行转发。

3. 运行

网络图（已在 topo.py 中实现）：



1) 启动拓扑

```
sudo ./topo.py
```

2) 运行控制器

```
uv run osken-manager least_hops.py --observe-links
```

3) 在 mininet CLI 中执行指令

```
mininet> h2 ping h9
```

4) 实验结果示例

```
mininet> h2 ping 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=200 ms
64 bytes from 10.0.0.9: icmp_seq=3 ttl=64 time=392 ms
64 bytes from 10.0.0.9: icmp_seq=4 ttl=64 time=391 ms
64 bytes from 10.0.0.9: icmp_seq=5 ttl=64 time=390 ms
64 bytes from 10.0.0.9: icmp_seq=6 ttl=64 time=390 ms
^C
--- 10.0.0.9 ping statistics ---
6 packets transmitted, 5 received, 16.6667% packet loss, time 5333ms
rtt min/avg/max/mdev = 200.127/352.695/391.931/76.286 ms
```

```
path: 10.0.0.2 -> 10.0.0.9
10.0.0.2 -> 1:s2:3 -> 2:s4:4 -> 2:s5:3 -> 3:s9:1 -> 10.0.0.9
```

实验现象补充说明：由于**沉默主机现象**，在主机未主动通信前，控制器无法感知其存在。前几次 ping 输出 host not find/no path 是正常现象。

4. 报告要求

- 阐述 networkx.shortest_simple_paths 的使用方法
- 实验结果截图

4.6 任务二：最少时延路径

在网络传输中，链路的实际时延往往比跳数更能反映路径的优劣。传统的最少跳数路由可能忽略链路质量差异，从而导致传输效率下降。在任务二中，你需要通过 LLDP 与 Echo 消息动态测量链路时延，构建加权拓扑图，并在此基础上计算从 h2 到 h9 的最小时延路径。

1. 任务要求

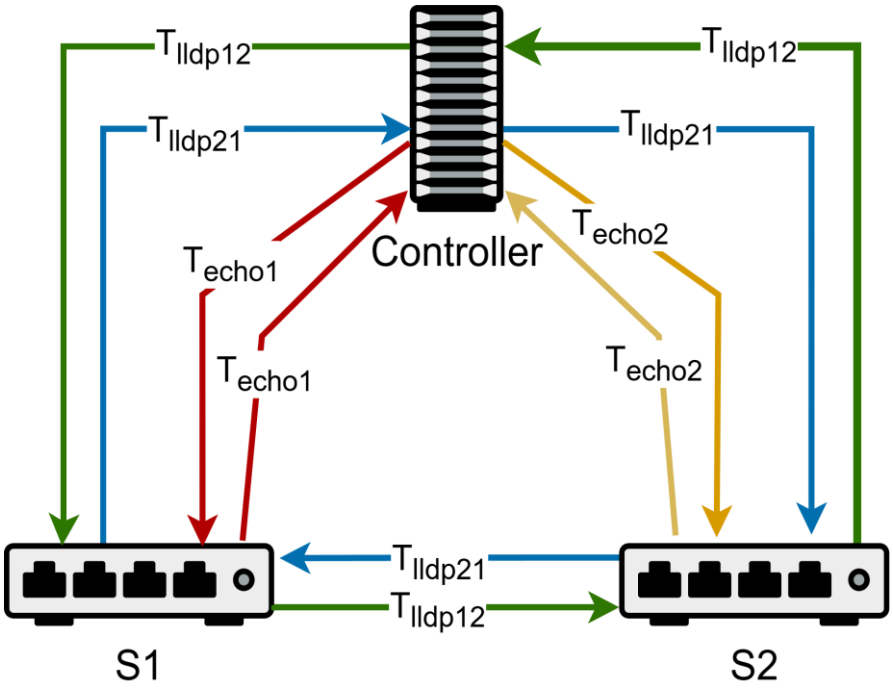
- 理解利用 LLDP 和 echo 测量链路延时的原理
- 实现最小时延路径选择
- 打印路径与总时延，并用 Ping 的 RTT 验证结果

2. 实现思路

1) **获取拓扑：** 收集主机、交换机、链路信息，构建图结构。
2) **测量时延：** 周期发送 LLDP（带时间戳）和 Echo（附加发送时间），计算链路方向的 LLDP 往返时延与控制器到交换机的 Echo 时延，组合得到链路单向时延。
3) **路径计算与打印：** 基于加权图计算 h2→h9 的最小时延路径，打印每条边的时延和总时延，并以 Ping 的 RTT 验证。
4) **注意**：由于链路发现和延迟计算是异步的，所以在 calculate_link_delay 中，你需要处理 lldp_link_delay[(s1, s2)]不存在的情况。Dict 的文档：[内置类型 dict — Python 3.13.8 文档]

3. 链路时延的测量原理

- **基本思路：** 控制器向交换机端口下发带时间戳的 LLDP，下一跳交换机将其回送控制器。控制器到各交换机的 Echo 往返时延也被周期测量。综合得到链路单向时延



- **变量定义：**
  - **LLDP 往返时间：**

    $T_{lldp_{12}}$：Controller → S1 → S2 → Controller，即绿线

    $T_{lldp_{21}}$：Controller → S2 → S1 → Controller，即蓝线
  - **ECHO 往返时间：**

    T_{echo_1}：Controller → S1 → Controller ，即红线

    T_{echo_2}：Controller → S2 → Controller ，即黄线
  - **链路$(S1,S2)$的单向延时$delay$为**

$$delay = max\left(\frac{T_{lldp_{12}} + T_{lldp_{21}} - T_{echo_1} - T_{echo_2}}{2}, 0\right), \quad delay应大于0$$

4. 实现

在这一部分，我们会给出完成实验所需代码的**框架**。请**补全**代码并插入至相应的位置。

1) 修改 os_ken 的源文件，使其支持T_{lldp}的测量

- 修改.venv/lib/python3.13/site-packages/os_ken/topology/switches.py 中 PortData，增加记录 LLDP 时延的变量

```
    # .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      # stamped at the time of sending
        self.sent = 0
        self.delay = 0             # T_lldp for this port
```

- 修改.venv/lib/python3.13/site-packages/os_ken/topology/switches.py，在 lldp_packet_in_handler 中计算 T_{lldp}

```
    # .venv/lib/python3.13/site-packages/os_ken/topology/switches.py
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def lldp_packet_in_handler(self, ev):
    # begin of addition
    recv_timestamp = time.time()  # record receive time
    # end of addition

    if not self.link_discovery:
        return

    msg = ev.msg
    try:
        src_dpid, src_port_no = LLDPPacket.lldp_parse(msg.data)
    except LLDPPacket.LLDPUnknownFormat:
        return

    # begin of addition
    # get the lldp delay, and save it into 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
    # end of addition
    ...
```

- 在 class NetworkAwareness 中添加 LLDP 延迟表和 switches 实例

```
    # network_awareness.py
def __init__(self, *args, **kwargs):
    ...
    self.lldp_delay_table = {}   # key: (src_dpid, dst_dpid) -> T_lldp
    self.switches = {}           # switches app instance
```

- 利用 lookup_service_brick 获取到正在运行的 switches 的实例（即前面被我们修改的类）。按如下的方式即可获取相应的T_{lldp}。**你需要**在 class NetworkAwareness 中添加 function： packet_in_hander，用于处理 LLDP 消息。示例代码如下：

```
    # class NetworkAwareness in network_awareness.py
from os_ken.base.app_manager import lookup_service_brick
...
```

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_hander(self, ev):
    msg = ev.msg
    dpid = msg.datapath.id
    try:
        src_dpid, src_port_no = LLDPPacket.lldp_parse(msg.data)

        if not self.switches:
            # get switches
            self.switches = lookup_service_brick('switches')

        # get lldp_delay
        for port in self.switches.ports.keys():
            if src_dpid == port.dpid and src_port_no == port.port_no:
                self.lldp_delay_table[(src_dpid, dpid)] = self.switches.ports
[port].delay
    except:
        return
```

2) 周期发送 Echo：记录控制器↔交换机的 RTT

i. 在 network_awareness.py 中增加数据结构：

- self.echo_RTT_table，用于记录 Echo RTT
- self.echo_send_timestamp，用于记录 Echo 的发送时间

```
    # class NetworkAwareness in network_awareness.py
def __init__(self, *args, **kwargs):
    ...
    self.echo_RTT_table = {}   # key: dpid -> T_echo
    self.echo_send_timestamp = {} # key: dpid -> send_time
```

ii. **在 network_awareness.py 中实现 function : send_echo_request，核心功能如下**:

- 构造 OFPEchoRequest 消息并发送
- 记录 send_time 并存入 echo_send_timestamp[dpid]

注意: 在构造 OFPEchoRequest 时, data 参数必须是一个 **bytes 类型的对象**。

代码框架：

```
    # class NetworkAwareness in network_awareness.py
def send_echo_request(self, switch):
    datapath = switch.dp
    parser = datapath.ofproto_parser
    '''
    TODO:
        构造 OFPEchoRequest 消息并发送
```

```
        记录 send_time 并存入 echo_send_timestamp[dpid]
    '''
```

参考文档：Echo Request

iii. 处理 Echo 回复，计算T_{echo}

- 编写处理 echo 包的函数 echo_reply_handler ,并添加至 network_awareness.py 中。子步骤如下：
  - 获取装有 send_time 的 msg，解析所属的交换机的 dpid 并记录 recv_time
  - 取出 data，并 decode data 获取原始数据 (可选)
  - 计算交换机 dpid 与控制器之间的 echo delay 并写入 echo_RTT_table
- 将 echo_reply_handler 与事件 EventOFPEchoReply 进行绑定

代码框架：

```
    # class NetworkAwareness in network_awareness.py
@set_ev_cls(ofp_event.EventOFPEchoReply, MAIN_DISPATCHER)
def handle_echo_reply(self, ev):
    try:
        '''
        TODO:
            获取装有 send_time 的 msg，解析所属的交换机的 dpid 并记录 recv_time
            取出 data，并 decode data 获取原始数据（可选)
            计算交换机 dpid 与控制器之间的 echo delay 并写入 echo_RTT_table
        '''
    except Exception:
        self.logger.warning("Failed to handle echo reply")
```

参考文档：EventOFPSwitchFeatures

iv. 周期性向每个交换机发送 Echo

修改 network_awareness.py，周期性向每一个 switch 发送 echo 包。**要求使用 hub.spawn** 实现。(在主线程中周期性发送 echo 会影响对 echo delay 的测量)

- 编写方法 examine_echo_RTT。
- 协程睡眠使用 hub.sleep，以减小对测量的影响。
- 使用 hub.spawn 创建新线程执行 examine_echo_RTT。

代码框架：

```
    # _get_topology() in network_awareness.py
# 调用 send_echo_request 的方式要与你实现的 send_echo_request 方式一致!
def examine_echo_RTT(self):
    while True:
```

```
        '''
        TODO:
            获取所有的 switch
            对每个 switch 的 echo RTT 进行测量
            睡眠一段时间(SEND_ECHO_REQUEST_INTERVAL)
        '''
```

3) 计算链路时延并更新 topo

i. 在 class NetworkAwareness 中添加字典 link_delay_table

```
    # NetworkAwareness.__init()__ in network_awareness.py
    def __init__(self, *args, **kwargs):
        ...
        self.link_delay_table = {}
                                        # (dpid1, dpid2) -> delay
```

ii. 计算链路时延

- **公式**: $delay = max\left(\frac{T_{lldp_{12}}+T_{lldp_{21}}-T_{echo_1}-T_{echo_2}}{2}, 0\right)$
- **代码框架**

```
    # network_awareness.py
def calculate_link_delay(self, src_dpid, dst_dpid):
    '''
    TODO:
        取出 LLDP delay 与 Echo RTT
        计算并返回 link 的 delay
    '''
```

注意：由于链路发现和延迟计算是异步的，所以在 calculate_link_delay 中，你需要处理 lldp_link_delay[(s1, s2)]不存在的情况。内置类型 dict — Python 3.13.8 文档

iii. 更新 topo：修改 network_awareness.py 中 class NetworkAwareness 的 _get_topology() ，使其能够计算 delay 并将 delay 添加至 edge 的属性中

- 在 add_edge 之前计算链路时延
- 输出 link delay info

```
    # 输出语句
self.logger.info("Link: %s -> %s, delay: %.5fms",
                 link.src.dpid, link.dst.dpid, delay*1000)
```

- 将 delay 添加至 edge 的属性中

4) 实现最小时延路径控制器：ShortestDelay

- **目标**: 计算基于 delay 权重计算最短路径（h2→h9），打印路径与总时延，并下发流表；同时输出与 Ping RTT 的对比。

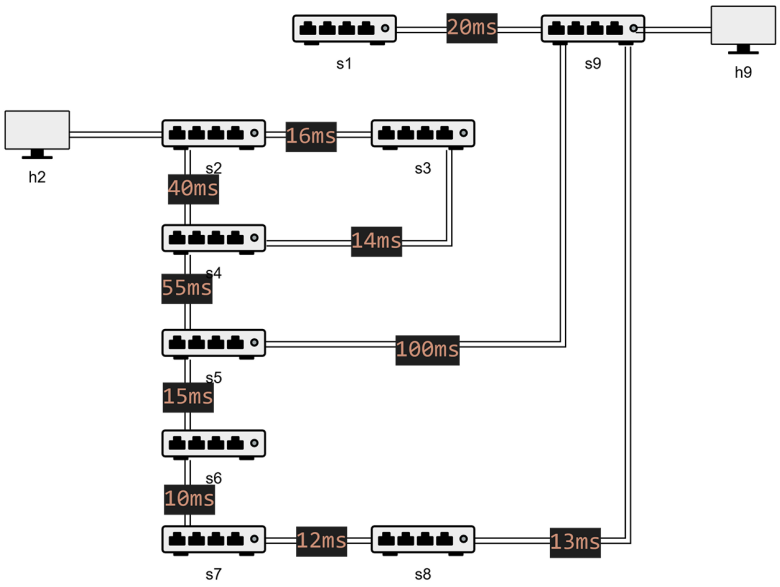
- **补充：**
  - shortest_delay.py 除类名外，其余代码均与 least_hop.py 一致(**同样未处理 arp，未实现自学习交换机)**
  - 不要忘记修改 self.weight，以切换 networkx 求解最短路径使用的属性
  - 代码框架已实现最短路径的计算，你的任务是计算 path delay, path RTT 并输出
- **步骤：**

  在 shortest_delay.py 中的 handel_ipv4()中，添加对 path delay 的计算，并输出 link delay dict, path delay 和 path RTT，格式如下：

```
self.logger.info('link delay dict: %s', )
self.logger.info("path delay= %.5fms", )
self.logger.info("path RTT = %.5fms", )
```

5. 运行

网络图：



1) 启动 topo

```
sudo ./topo.py
```

2) 运行控制器

```
uv run osken-manager shortest_delay.py --observe-links
```

3) 运行 ping 命令

```
mininet> h2 ping h9
```

4) 实验结果示例

```
*** Starting CLI:
mininet> h2 ping h9
PING 10.0.0.9 (10.0.0.9) 56(84) bytes of data.
64 bytes from 10.0.0.9: icmp_seq=3 ttl=64 time=141 ms
64 bytes from 10.0.0.9: icmp_seq=4 ttl=64 time=272 ms
64 bytes from 10.0.0.9: icmp_seq=5 ttl=64 time=271 ms
64 bytes from 10.0.0.9: icmp_seq=6 ttl=64 time=270 ms
64 bytes from 10.0.0.9: icmp_seq=7 ttl=64 time=270 ms
64 bytes from 10.0.0.9: icmp_seq=8 ttl=64 time=270 ms
^C
--- 10.0.0.9 ping statistics ---
8 packets transmitted, 6 received, 25% packet loss, time 7569ms
rtt min/avg/max/mdev = 140.829/249.097/271.868/48.421 ms
mininet>
```

```
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: {('10.0.0.2', 2): 0, (2, 3): 16.30556583404541, (3, 4): 13.789892196655273, (4, 5): 55.1152229309082, (5, 6): 15.16449451446533
2, (6, 7): 9.878396987915039, (7, 8): 12.224197387695312, (8, 9): 13.316035270690918, (9, '10.0.0.9'): 0}
path delay = 135.79381ms
path RTT = 271.58761ms
```

6. 报告要求

- **实现逻辑及关键代码**
- **展示实验结果**
- **实验过程中遇到的问题、解决方法**

4.7 任务三：容忍链路故障

在实际网络中，交换机之间的链路可能会中断或恢复。为了保证通信不中断，控制器需要在链路故障或恢复时，自动重新选择时延最低的路径。我们可以在 mininet 控制台中使用 link s6 s7 down 和 link s6 s7 up 来模拟链路故障和链路恢复。

1. 任务要求

- 学会在 mininet 中模拟链路故障与恢复
- 修改控制器代码，使其能在链路变化时自动更新路径
- 验证网络在链路故障和恢复后的自适应能力
- 阅读、补全任务三中的代码框架，并添加至对应的文件中

2. 整体实现思路

1) 捕捉链路变化，在链路变化时需要删除的对象。需要删除的对象有：拓扑图、流表、sw、mac_to_port

2) 删除流表后，交换机会因为没有对应的流表项而将数据包发送至控制器，重新走一遍寻找任务二中寻找最小延迟路径的流程。进而实现自动切换路径。

3) 提示以及相关文档：

- EventOFPPortStatus 事件相关文档：Port Status Message
- OFPFlowMod 相关文档：Modify State Messages
- 如果仅删除 s6 与 s7 上的流表，arp 请求会被 s5 转发至 s6。因为 s6 与 s7 之间的链路已经中断，s6 会将 arp 请求重新转发至 s5，进而产生环路。为简化难度，对于相关流表的删除，本实验不要求精准删除流表项。这里仅要求在链路状态发生变化时，获取所有流表项的 dpid，port 信息，并逐一相关的流表项。

3. 实现

1) 捕捉链路状态的变化

当链路状态发生变化时，相关端口的状态也会变化，从而触发

EventOFPPortStatus 事件。将该事件与自定义的处理函数进行绑定，就能捕获端口状态的变化并执行相应的逻辑处理。

请参考.venv/lib/python3.13/site-packages/os_ken/controller/ofp_handler.py 以及 EventOFPPortStatus 事件相关文档在 class NetworkAwareness 中实现相关处理。

- 代码框架:

```
# class ShortestDelay in shortest_delay2.py
@set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
def port_status_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto

    if msg.reason in [ofproto.OFPPR_ADD, ofproto.OFPPR_MODIFY]:
        # 端口新增或修改(link up 和 link down 均属于对端口状态的修改)
        datapath.ports[msg.desc.port_no] = msg.desc
        '''
        TODO:
            情况拓扑图。（调用 topo_map 使用`self.network_awareness.topo_map`）
            删除所有流表
            删除 sw
            删除 mac_to_port
        '''
    elif msg.reason == ofproto.OFPPR_DELETE:
        datapath.ports.pop(msg.desc.port_no, None)
    else:
        return
```

2) 清空相关的流表和数据结构

当链路状态发生变化时（延迟，断开，恢复），最小延迟路径可能发生变化。因此需要删除 link 对应的流表以重新规划路线。否则，在本实验指导书的方法下，交换机按照旧路线转发数据包，进而导致无法通过 Packet in 的方式进入任务二中寻找最小延迟路径的流程。

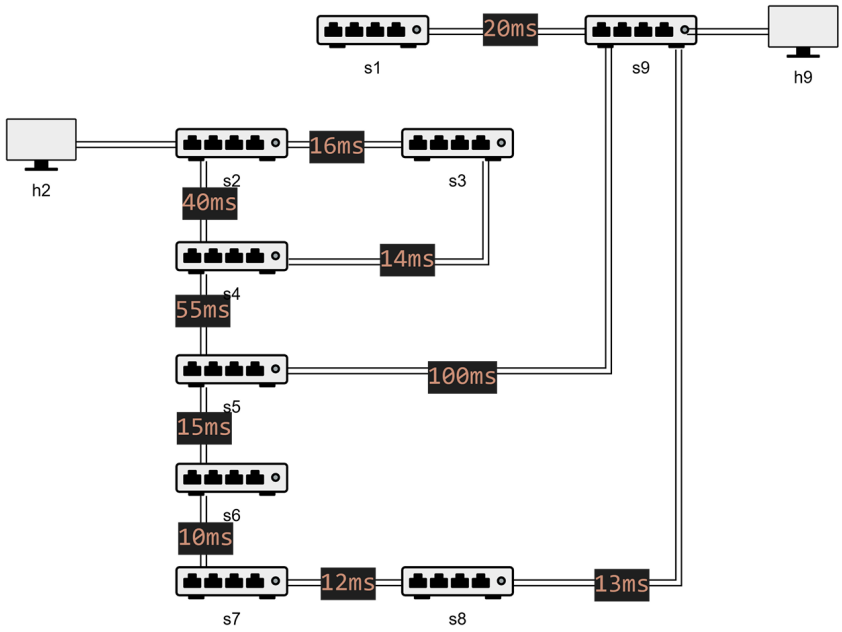
请参考 add_flow 在 class NetworkAwareness 中实现 delete_flow()函数

- **思路：**向交换机发送 OFPFC_DELETE 消息可以删除相应的流表。由于添加和删除都属于 OFPFlowMod 消息，因此可以参考 add_flow()函数实现 delete_flow()函数。
- **流程**：构造匹配字段 -> 设置 OFPFlowMod 消息 -> 发送消息到交换机
- **相关文档：**OFPMatch，OFPFlowMod
- **代码框架**：

```
# class ShortestDelay in shortest_delay2.py
from os_ken.topology.api import get_all_switch

def delete_all_flow(self):
    '''
    TODO:
        参考_get_topology() in network_awareness.py
        遍历所有 switch 的端口，然后删除删除所有流表项
    '''

# class ShortestDelay in shortest_delay2.py
def delete_flow(self, datapath, port_no):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    try:
        '''
        TODO:
            1. 构造匹配字段
            2. 设置 OFPFlowMod 消息
            3. 发送消息到交换机
            注意：你需要发送两个消息，一个用于删除 in_port 的流表，另一个用于删除 action(out_port)的流表
        '''

    except Exception as e:
            self.logger.error("Failed to delete flow entries associated with port %s on switch %s: %s", port_no, datapath.id, str(e))
```

4. 运行

网络图：



a) 运行 topo

```
sudo ./topo.py
```

b) 运行控制器

```
uv run osken-manager shortest_delay.py --observe-links
```

c) 初始状态：h2 ping h9，选择最优路径，RTT≈270ms

```
mininet> h2 ping h9
```

d) 执行 link s6 s7 down：控制器重新选择次优路径，RTT≈370ms

```
mininet> link s6 s7 down
```

e) 执行 link s6 s7 up：链路恢复，控制器再次选择最优路径，RTT≈270ms

```
mininet> link s6 s7 up
```

f) 实验结果示例

Mininet 控制台输出：

```
*** Starting CLI:
mininet> h2 ping h9
PING 10.0.0.9 (10.0.0.9) 56(84) bytes of data.
64 bytes from 10.0.0.9: icmp_seq=3 ttl=64 time=138 ms
64 bytes from 10.0.0.9: icmp_seq=4 ttl=64 time=271 ms
64 bytes from 10.0.0.9: icmp_seq=5 ttl=64 time=270 ms
64 bytes from 10.0.0.9: icmp_seq=6 ttl=64 time=270 ms
64 bytes from 10.0.0.9: icmp_seq=7 ttl=64 time=271 ms
^C
--- 10.0.0.9 ping statistics ---
7 packets transmitted, 5 received, 28.5714% packet loss, time 6349ms
rtt min/avg/max/mdev = 137.738/244.117/271.309/53.191 ms
mininet> link s6 s7 down
mininet> h2 ping h9
PING 10.0.0.9 (10.0.0.9) 56(84) bytes of data.
64 bytes from 10.0.0.9: icmp_seq=1 ttl=64 time=188 ms
64 bytes from 10.0.0.9: icmp_seq=2 ttl=64 time=371 ms
64 bytes from 10.0.0.9: icmp_seq=3 ttl=64 time=370 ms
64 bytes from 10.0.0.9: icmp_seq=4 ttl=64 time=370 ms
64 bytes from 10.0.0.9: icmp_seq=5 ttl=64 time=370 ms
^C
--- 10.0.0.9 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4085ms
rtt min/avg/max/mdev = 187.514/333.786/370.740/73.136 ms
mininet> link s6 s7 up
mininet> h2 ping h9
PING 10.0.0.9 (10.0.0.9) 56(84) bytes of data.
64 bytes from 10.0.0.9: icmp_seq=1 ttl=64 time=423 ms
64 bytes from 10.0.0.9: icmp_seq=2 ttl=64 time=271 ms
64 bytes from 10.0.0.9: icmp_seq=3 ttl=64 time=270 ms
64 bytes from 10.0.0.9: icmp_seq=4 ttl=64 time=270 ms
64 bytes from 10.0.0.9: icmp_seq=5 ttl=64 time=270 ms
^C
--- 10.0.0.9 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4289ms
rtt min/avg/max/mdev = 270.280/301.040/423.391/61.175 ms
mininet>
```

控制器输出：

初始状态结果示例

```
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: {('10.0.0.2', 2): 0, (2, 3): 15.936017036437988, (3, 4): 14.01674747467041, (4, 5): 55.03571033477783, (5, 6): 15.085220336914
062, (6, 7): 10.036230087280273, (7, 8): 11.931180953979492, (8, 9): 13.155460357666016, (9, '10.0.0.9'): 0}
path delay = 135.19657ms
path RTT = 270.39313ms
```

s6 与 s7 之间链接发生故障时，结果示例

```
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: {('10.0.0.2', 2): 0, (2, 3): 16.03078842163086, (3, 4): 13.921618461608887, (4, 5): 54.91447448730469, (5, 9): 180.02481937408
447, (9, '10.0.0.9'): 0}
path delay = 184.89170ms
path RTT = 369.78340ms
```

s6 与 s7 之间链接恢复时，结果示例：

```
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: {('10.0.0.2', 2): 0, (2, 3): 21.656513214111328, (3, 4): 14.2136812210083, (4, 5): 55.14204502105713, (5, 6): 15.1460170745849
61, (6, 7): 10.246634483337402, (7, 8): 12.320280075073242, (8, 9): 13.373017311096191, (9, '10.0.0.9'): 0}
path delay = 142.09819ms
path RTT = 284.19638ms
```

5. 报告要求

- 实现逻辑及关键代码
- 展示实验结果
- 实验过程中遇到的问题、解决方法
- 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