实验指导手册（四）

1. 链路时延测量实验介绍

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。

---

2. 实验概览

1. **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。
2. **实验任务**
    - **任务一：最少跳数路径**
        - 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：最小时延路径**
        - 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：容忍链路故障**
    - 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。

3. 实验报告要求

- 报告需包含以下内容：
    - 实验目的与背景简述
    - 子任务的报告
    - 完成本实验的思考（如收获、心得、感悟、看法等）
- 要求：报告应条理清晰，逻辑完整，避免仅贴代码。

---

3. 指导与提示

- **实验建议：**
    - 不需要关注`class NetworkAwareness`中`self.switch_info`，`link_info`，`self.port_link`，`self.port_info`的作用
    - **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**清除旧拓扑**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

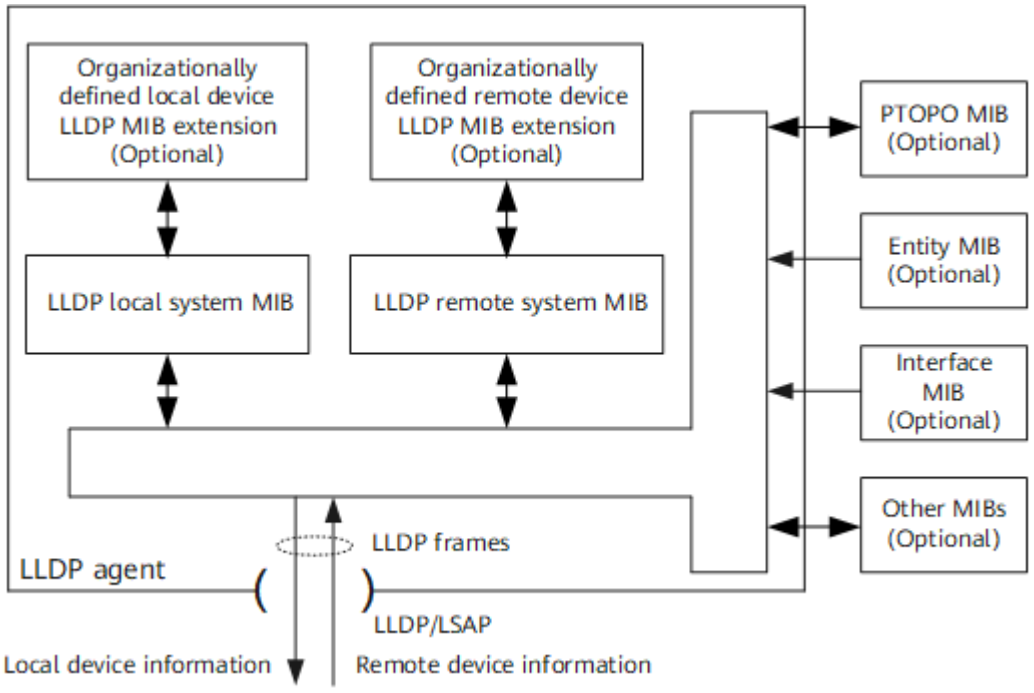
3.1 关于`LLDP`

`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：



3.2 `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。

   控制器会创建一个 PacketOut 消息，向 S1 的三个端口分别发送 LLDP 数据包。每个 LLDP 包中包含两个关键字段：

   - **Chassis ID TLV**：设置为 S1 的设备 ID（dpid）
   - **Port ID TLV**：设置为对应的端口号

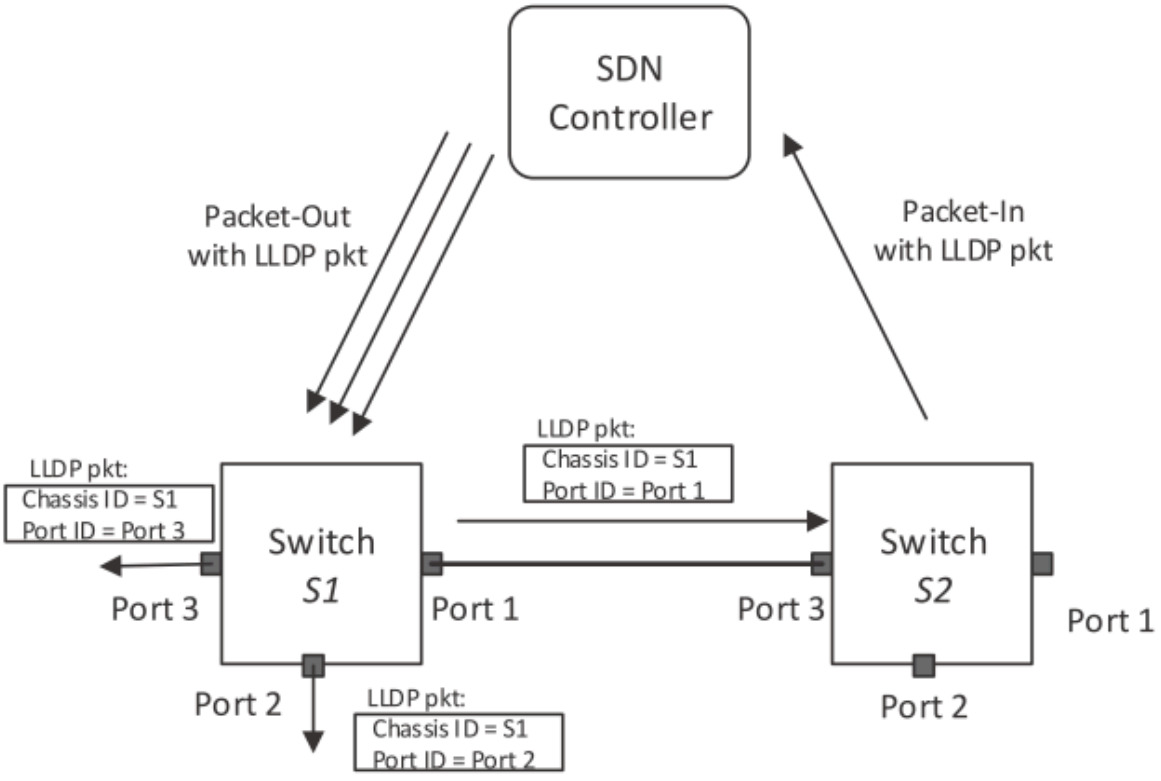
2. 控制器下发流表到 S1，指示交换机将从**控制器端口**接收到的 LLDP 数据包，发送到指定的物理端口

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. 下载实验

1. 使用`Git`下载实验文件

```
git clone https://gitee.com/forrest_a/lab3.git
cd sdn-lab3
```

2. 安装依赖 / 运行步骤(和**实验指导手册(二)**相同)。

5. 实验内容

5.1 任务一：最少跳数路径

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**

5.1.1 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。（思路见1.1）
- 能够解释 `networkx.shortest_simple_paths` 的使用方法。(阅读文档 networkx.shortest_simple_paths API)
- 参考实验指导书（二）中的实验，解决`arp`环路洪泛问题。

5.1.2 任务内容

5.1.2.1 获取网络拓扑

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：

```python
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。

     `show_network.py` 中的部分代码如下：

```python
# line = 40
def _get_topology(self):
    while True:
        self.logger.info('\n\n\n')

        # get topology
        hosts = get_all_host(self)
        switches = get_all_switch(self)
        links = get_all_link(self)

        # print
        self.logger.info('hosts:')
        for hosts in hosts:
            self.logger.info(hosts.to_dict())

        self.logger.info('switches:')
```

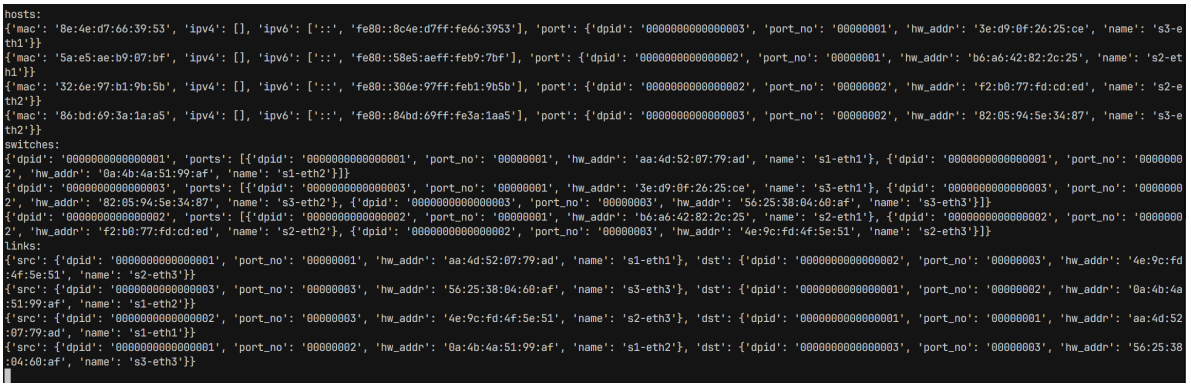
```
        for switch in switches:
            self.logger.info(switch.to_dict())

        self.logger.info('links:')
        for link in links:
            self.logger.info(link.to_dict())

        hub.sleep(2)
```

可以使用以下命令查看输出情况（注意：运行 `os_ken` 时需加上 `--observe-links` 参数）：

```
# 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
```

输出结果示例：



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：

```
# _get_topology(self) in network_awareness.py
...
# line = 88
for host in hosts:
    # take one ipv4 address as host id
    if host.ipv4:
        self.link_info[(host.port.dpid, host.ipv4[0])] = host.port.port_no
        self.topo_map.add_edge(host.ipv4[0], host.port.dpid, hop=1, delay=0, is_host=True)

...
for link in links:
```

```
    ...
    # line = 109
    self.topo_map.add_edge(link.src.dpid, link.dst.dpid, hop=1,
is_host=False)
```

这样，控制器就能维护一张完整的网络拓扑图。

5.1.2.2 计算最短路径

1. 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：

```
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')
```

2. 处理 `IPv4` 报文

在 `least_hops.py` 中，针对 IPv4 报文进行路径处理，并生成端口转发表：

```
# function in least_hops.py
def handle_ipv4(self, msg, 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:
        return
    self.path=dpid_path

    # get port path:  h1 -> in_port, s1, out_port -> h2
    port_path = []
    for i in range(1, len(dpid_path) - 1):
        in_port = self.network_awareness.link_info[(dpid_path[i],
dpid_path[i - 1])]
        out_port = self.network_awareness.link_info[(dpid_path[i],
dpid_path[i + 1])]
        port_path.append((in_port, dpid_path[i], out_port))
    self.show_path(src_ip, dst_ip, port_path)
```

```
17
18      # send flow mod
19      for node in port_path:
20          in_port, dpid, out_port = node
21          self.send_flow_mod(parser, dpid, pkt_type, src_ip, dst_ip,
    in_port, out_port)
22          self.send_flow_mod(parser, dpid, pkt_type, dst_ip, src_ip,
    out_port, in_port)
```

3. 处理`arp`环路洪泛问题

请参考实验 2，建议使用`(dpid, src_mac, dst_mac) -> in_port`的方法进行处理。

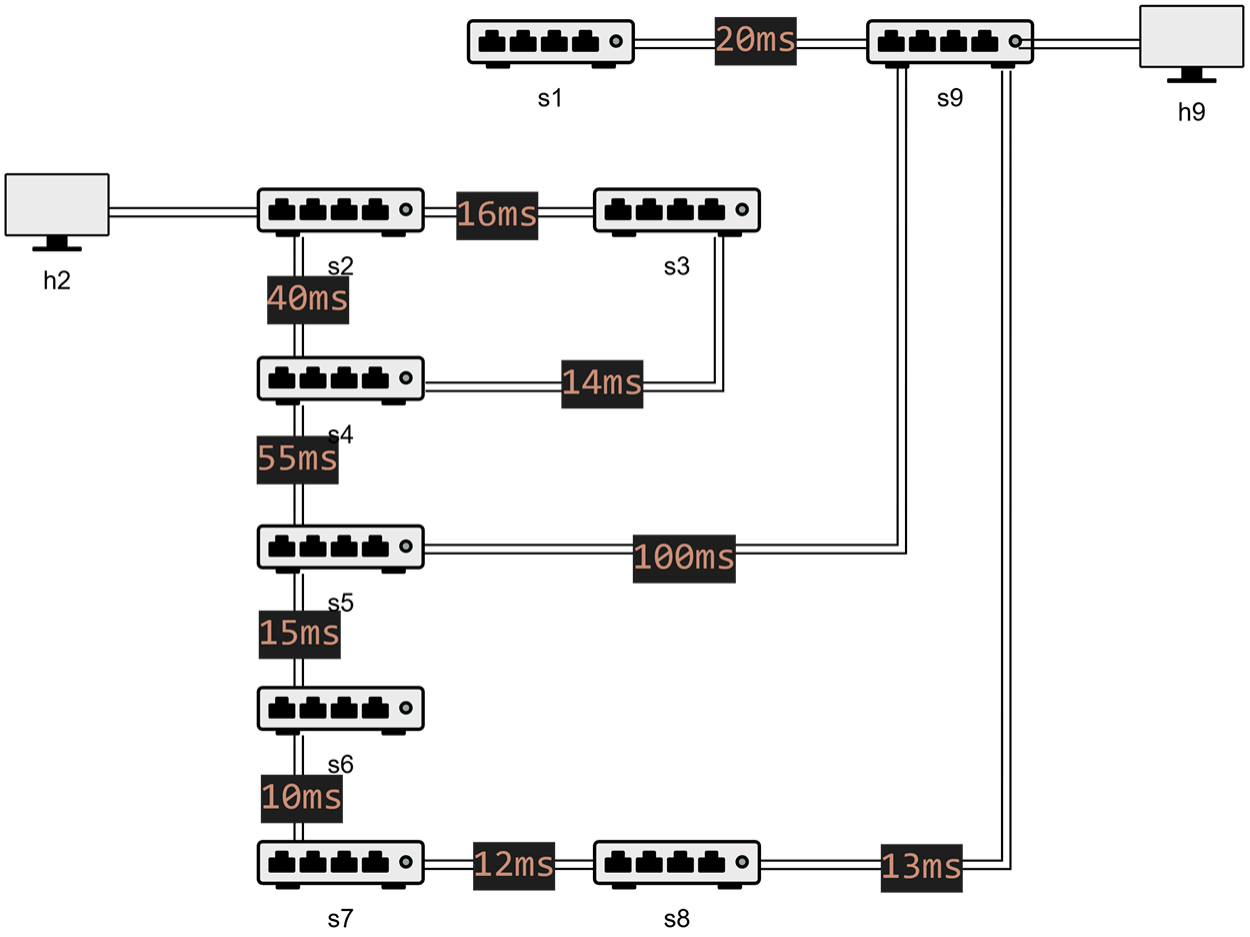
任务：

- 自学习交换机
- `arp`环路洪泛问题

```
1  def handle_arp(self, msg, in_port, dst,src, pkt,pkt_type):
2      """
3          handle arp loop here
4          use method: (dpid, src_mac, dst_mac) -> in_port
5      """
6      pass
```

5.1.2.3 运行实验

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：

```
*** 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=331 ms
64 bytes from 10.0.0.9: icmp_seq=5 ttl=64 time=330 ms
64 bytes from 10.0.0.9: icmp_seq=6 ttl=64 time=330 ms
64 bytes from 10.0.0.9: icmp_seq=7 ttl=64 time=330 ms
64 bytes from 10.0.0.9: icmp_seq=8 ttl=64 time=330 ms
^C
--- 10.0.0.9 ping statistics ---
8 packets transmitted, 6 received, 25% packet loss, time 7553ms
rtt min/avg/max/mdev = 137.687/298.195/330.618/71.781 ms
mininet>
```

```
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
```

5. **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。因此，前几次 `ping` 可能会输出：

   ```
   host not find/no path
   ```

   这是正常现象，不必担心。只要主机开始通信，控制器就能感知到主机的存在，进而建立包含主机的 `topo` 图。

5.1.3 任务一的报告要求：

- `networkx.shortest_simple_paths` 的使用方法
- 总结计算最短路径的通用思路
- 实验结果截图

5.2 任务二：最小时延路径

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。

5.2.1 任务要求：

- 理解利用 `LLDP` 和 `echo` 测量链路延时的原理（见2.1链路时延的测量原理）
- 在已有的最少跳数路径代码框架上，周期性地用 `LLDP` 和 `Echo` 测量链路时延，构建加权拓扑图
- 计算从 `h2` 到 `h9` 的最小时延路径，并下发相应的流表项
- 打印路径与总时延，并用 `Ping` 的 `RTT` 验证结果
- 阅读、补全任务二中的代码框架，并添加至对应的文件中

5.2.2 整体实现思路：

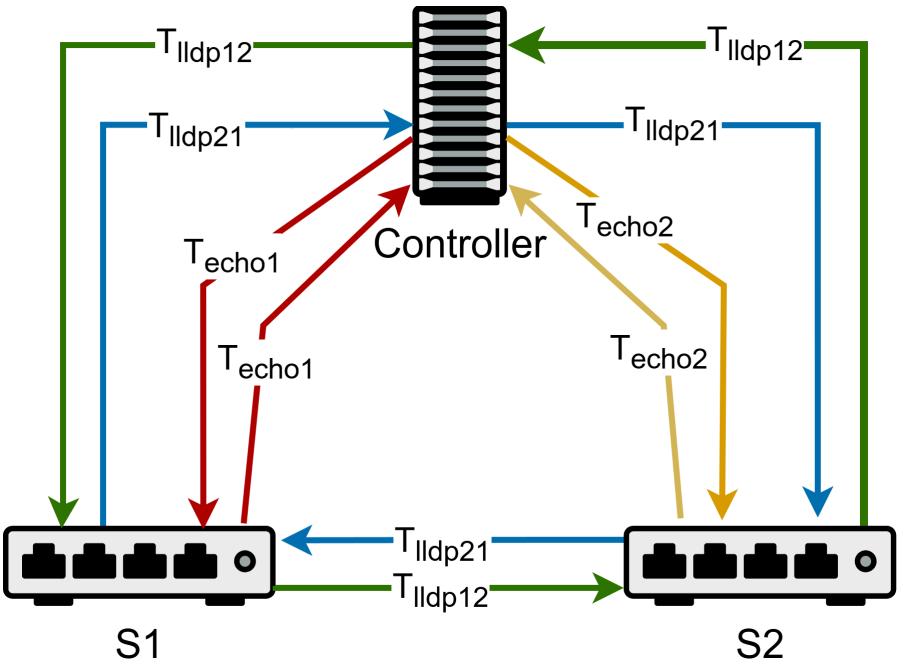
1. **获取拓扑：** 收集主机、交换机、链路信息，构建图结构。
2. **测量时延：** 周期发送 `LLDP`（带时间戳）和 `Echo`（附加发送时间），计算链路方向的 `LLDP` 往返时延与控制器到交换机的 `Echo` 时延，组合得到链路单向时延。
3. **路径计算与打印：** 基于加权图计算 `h2` → `h9` 的最小时延路径，打印每条边的时延和总时延，并以 `Ping` 的 `RTT` 验证。
4. **注意：**
    - 由于链路发现和延迟计算是异步的，所以在`calculate_link_delay`中，你需要处理`lldp_link_delay[(s1, s2)]`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

      内置类型dict — Python 3.13.8 文档

5.2.3 任务内容

5.2.3.1 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：
    - $T_{lldp_{12}}$：`Controller` → `S1` → `S2` → `Controller`，即绿线
    - $T_{lldp_{21}}$：`Controller` → `S2` → `S1` → `Controller`，即蓝线
  - `Echo` 往返：
    - T_{echo_1}：`Controller` → `S1` → `Controller`，即红线
    - T_{echo_2}：`Controller` → `S2` → `Controller`，即黄线
  - 则链路（`S1`，`S2`）的单向时延为：

    $delay = max(\frac{T_{lldp_{12}}+T_{lldp_{21}}-T_{echo_1}-T_{echo_2}}{2}, 0)$
- **合理性约束：** 若计算结果为负数，取 0。

5.2.3.2 实现

在**实现**部分给出的代码框架均需要补充完整并插入至相应的位置

1. **增强 `LLDP`：记录发送与接收时间差**

   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`中的 `PortData`，增加 `LLDP` 时延

```python
# .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}

```python
# .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

```python
# 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（即步骤1、2中被我们修改的类）。按如下的方式即可获取相应的T_{lldp}
  - **具体做法**：在`class NetworkAwareness`中添加`function`：`packet_in_hander`，用于处理`LLDP`消息。

  示例代码：

```python
# 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`

1. 在`network_awareness.py`中增加数据结构:

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

```python
# 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
```

2. 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**。有两种常用的处理方法：

   - **数值数据**（如时间戳、测量值）推荐使用`struct.pack`，`struct.unpack`。其中,即使只打包一个对象`struct.unpack`也会解压为一个`list`

     struct --- 将字节串解读为打包的二进制数据 — Python 3.13.8 文档

   - **文本数据**（如消息、标签）使用`bytes(..., encoding="utf-8")`，`data.decode('utf-8')`。或者使用`data=b''`构造`bytes`类型的空串

   代码框架：

```python
# 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]
    '''
```

   参考文档及相关代码： Echo Request

```python
def send_echo_request(self, datapath, data):
    ofp_parser = datapath.ofproto_parser    # 创建构造 OpenFlow 消息对象 的构造器

    req = ofp_parser.OFPEchoRequest(datapath, data) # 构造 OpenFlow Echo 请求消息
    datapath.send_msg(req)  # 发送消息
```

3. 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")
```

参考文档及相关代码： EventOFPSwitchFeatures

```
# Example:
@set_ev_cls(ofp_event.EventOFPEchoReply,[HANDSHAKE_DISPATCHER, CONFIG_DISPATCHER, MAIN_DISPATCHER])
def echo_reply_handler(self, ev):
    self.logger.debug('OFPEchoReply received: data=%s',
                      utils.hex_array(ev.msg.data))
```

4. 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`
  - 使用`hub.spawn`创建新线程执行`examine_echo_RTT`

- 注：这里仅进行简单实现，不考虑并发带来的数据冲突问题。

- 注意：

  协程睡眠使用`hub.sleep`，以减小对测量的影响。因为`hub.sleep`基于`gevent`实现。`gevent`中的所有协程均通过一个`OS`线程执行，使用`gevent.sleep`可以切换至下一个协程而不造成停顿。使用`time.sleep`则会将`OS`线程挂起进而阻塞该`os`线程上的所有`gevent`协程。

- 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`

1. 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
```

2. 计算链路时延

- 公式：$delay = max(\frac{T_{lldp_{12}}+T_{lldp_{21}}-T_{echo_1}-T_{echo_2}}{2}, 0)$

- 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

相关文档：[内置类型dict — Python 3.13.8 文档]

3. 更新`topo`：修改`_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

   代码框架：

   ```
   # get_topology() in network_awareness
   ...
   def _get_topology(self):
       ...
       for link in links:
           ...

           '''
           TODO:
               计算链路delay
               将delay存入link_delay_table
               使用self.logger.info打印delay消息
           '''

           # Add link to topology graph
           # TODO: 将delay添加至edge的属性中
           self.topology_graph.add_edge(
               link.src.dpid,
               link.dst.dpid,
               hop=1,
               is_host=False
           )
   ```

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。
   - **步骤：**

1. 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", )
```

代码框架：

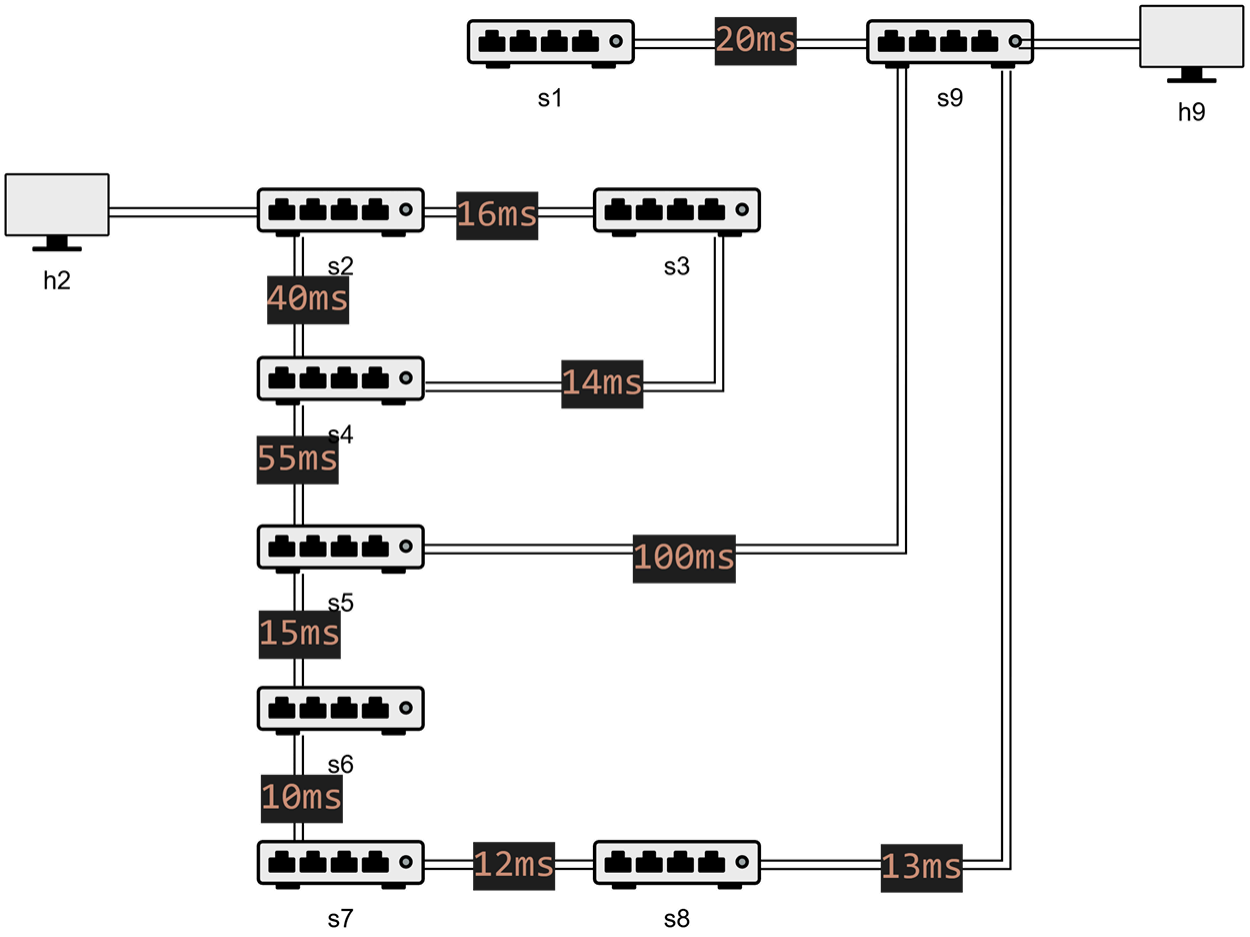
```
# handle_ipv4() in shortest_delay.py
def handle_ipv4(self, msg, src_ip, dst_ip, pkt_type):
    parser = msg.datapath.ofproto_parser

    # get shortest path
    dpid_path = self.network_awareness.shortest_path(src_ip, dst_ip,weight=self.weight)
    if not dpid_path:
        return

    self.path=dpid_path
    # get port path:  h1 -> in_port, s1, out_port -> h2
    port_path = []
    for i in range(1, len(dpid_path) - 1):
        in_port = self.network_awareness.link_info[(dpid_path[i], dpid_path[i - 1])]
        out_port = self.network_awareness.link_info[(dpid_path[i], dpid_path[i + 1])]
        port_path.append((in_port, dpid_path[i], out_port))
    self.show_path(src_ip, dst_ip, port_path)

    # calc path delay
    '''
    TODO:
        利用dpid_path(最短路)和link_delay_table计算path delay, path RTT
        输出link delay dict, path delay,  path RTT
        输出语句示例:
            self.logger.info('link delay dict: %s', )
            self.logger.info('path delay = %.5fms', )
            self.logger.info('path RTT = %.5fms', )
    '''

```

```
30      # send flow mod
31      ...
```

- 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

5.2.3.3 运行

网络图：



1. 启动 `topo`

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

2. 运行控制器

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

3. 运行 `ping` 命令

```
mininet> h2 ping h9
```

5.2.3.4 实验结果

此处给出实验结果示例。你的结果应与所给示例相似。

```
*** 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
```

5.2.4 任务二的报告要求：

- 实现逻辑及关键代码，重点描述如何减小测量误差
- 展示实验结果（类似3.2实验结果示例，`delay`的测量误差应控制在**20ms**以内）
- 实验过程中遇到的问题、解决方法

5.3 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**。

5.3.1 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

5.3.2 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有：
    - 拓扑图
    - 相关流表
    - `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

提示：

- `EventOFPPortStatus` 事件相关文档：Port Status Message
- `OFPFlowMod` 相关文档：Modify State Messages
- 实际上，如果仅删除 `s6` 与 `s7` 上的流表，`arp` 请求会被 `s5` 转发至 `s6`，导致 `s6` 建立流表，进而产生环路。

  因此，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。这里给出三种实现方式，任选其一即可。

  1. 添加流表项时设定 `hard_timeout`，待流表项生命到期时会自动销毁（不要修改 `App` 启动时添加的流表）
  2. 获取所有流表项的 `dpid`，`port` 信息，逐一使用 `delete flow` 删除
  3. 获取最短路径上的 `dpid`，`port` 信息，逐一使用 `delete flow` 删除（存在缺陷：按照顺序执行指令 `h2 ping h9`，`pingall`，`link s6 s7 down`，`h2 ping h9`，并不能恢复 `h2` 与 `h9` 之间的通信）

  因为为流表项设置生命周期比较简单，在**任务内容**部分，我们只给出方式2的代码框架。

5.3.3 任务内容

5.3.3.1 实现

1. 捕捉链路状态变化

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：

```python
# 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
```

1. 5.3.3.2 删除相关流表

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。

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

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

相关文档：

- OFPMatch
- OFPFlowMod

代码框架：

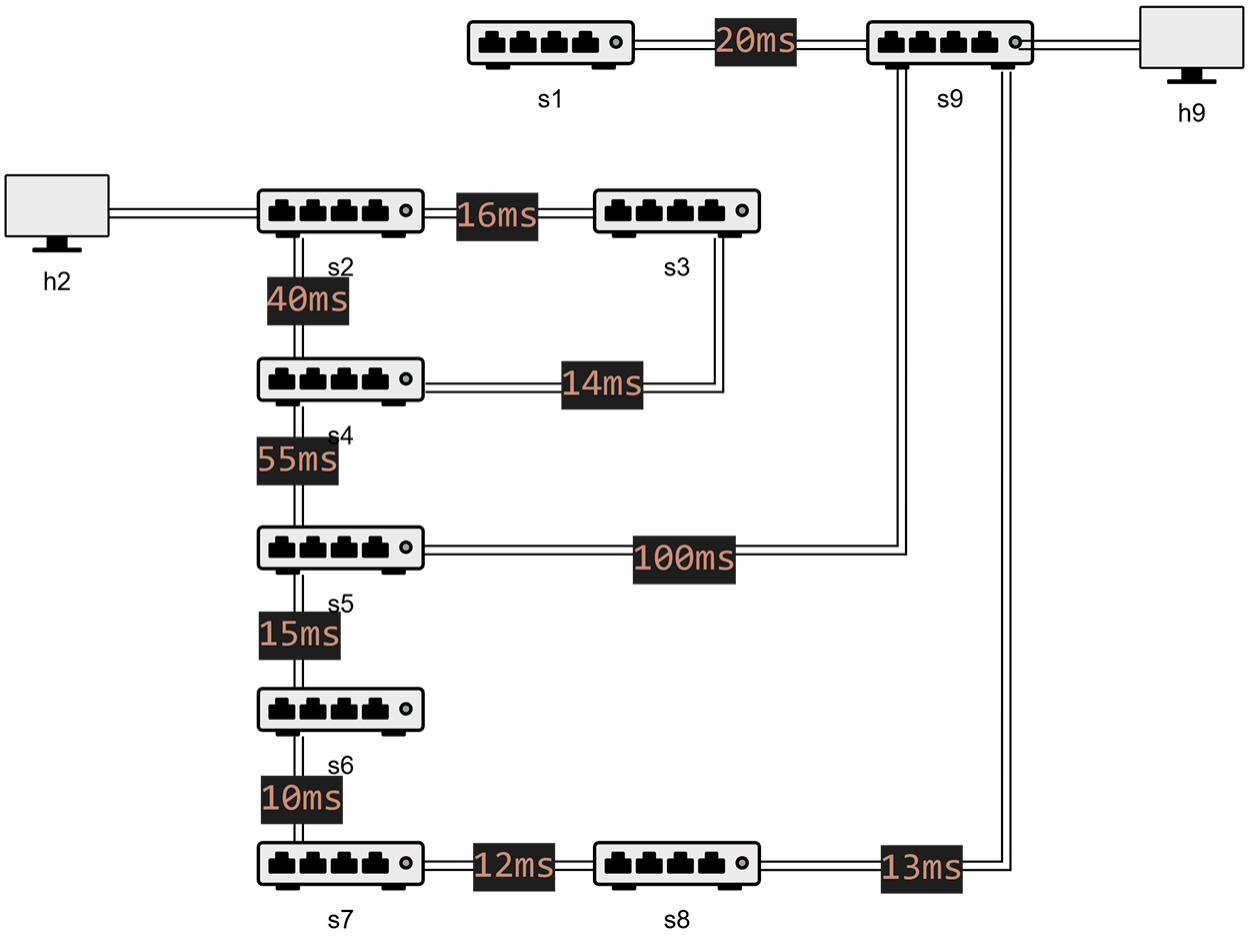
```python
# class ShortestDelay in shortest_delay2.py
from os_ken.topology.api import get_all_switch

def delete_all_flow(self):
    '''
    TODO:
        参考_get_topology() in network_awareness.py
        遍历所有switch的端口
        delete_flow
    '''
```

```python
# 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))
```

这样，当链路状态发生变化时，拓扑图和相关流表会被清除，新的数据包将触发`packet_in_handler()`，控制器会重新计算并下发新的最优路径。

5.3.3.3 运行

网络图：



1. 运行 `topo`

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

2. 运行控制器

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

3. 初始状态：`h2 ping h9`，选择**最优**路径，`RTT` ≈ **270ms**

```
mininet> h2 ping h9
```

4. 执行 `link s6 s7 down`：控制器重新选择**次优**路径，`RTT` ≈ **370ms**

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

5. 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
```

5.3.3.4 实验结果示例

你的实验结果应当与下面的示例类似。

```
*** 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>
```

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): 100.02481937408
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
```

5.3.4 任务三的报告要求：

实验报告中需要包含以下部分：

- 实现逻辑及关键代码
- 展示实验结果（类似3.3实验结果示例，`delay`偏差应当小于**20ms**）
- 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，而不需要清空`lldp_delay_table`这类记录`delay`的对象
- 指出`delete flow`与`add flow`在实现流程上的异同