一、实验目的

（1）学习利用 os_ken.topology.api 发现网络拓扑。
（2）学习利用 LLDP 和 Echo 数据包测量链路时延。
（3）学习计算基于跳数和基于时延的最短路由。
（4）学习设计能够容忍链路故障的路由策略。
（5）分析网络集中式控制与分布式控制的差异，思考 SDN 的得与失。

二、实验任务

必做题：最小时延路径

跳数最少的路由不一定是最快的路由，链路时延也会对路由的快慢产生重要影响。请实时地利用 LLDP 和 Echo 数据包测量各链路的时延，在网络拓扑的基础上构建一个有权图，然后基于此图计算最小时延路径。具体任务是，找出一条从 SDC 到 MIT 时延最短的路径，输出经过的路线及总的时延，利用 Ping 包的 RTT 验证你的结果。

选做题：容忍链路故障

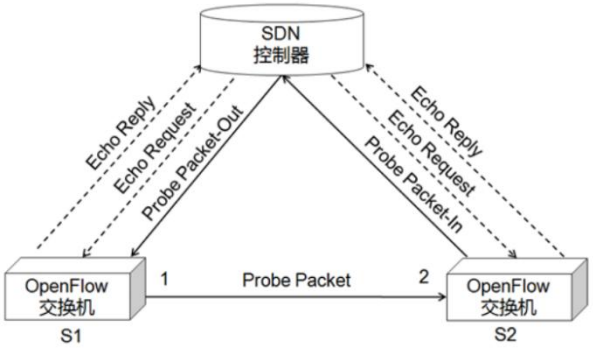
1970 年的网络硬件发展尚不成熟，通信链路和交换机端口发生故障的概率较高。请设计 OSKen app ，在任务一的基础上实现容忍链路故障的路由选择：每当链路出现故障时，重新选择当前可用路径中时延最低的路径；当链路故障恢复后，也重新选择新的时延最低的路径。

请在实验报告里附上你计算的（1）最小时延路径（2）最小时延路径的 RTT（3）链路故障/恢复后发生的路由转移。

三、实验过程

3.1 最小时延路径

测量链路时延的思路可参考下图。



此部分实验核心逻辑是，当控制器接收到携带 ipv4 报文的 Packet_In 消息时，调用 networkx 计算最短路，也可以自行实现如迪杰斯特拉算法，然后把相应的路由下发到沿途交换机。

控制器将带有时间戳的 LLDP 报文下发给 S1 ，S1 转发给 S2，S2 上传回控制器，根据收到时间和发送时间即可计算出控制器经 S1 到 S2 再返回控制器的时延，记为 lldp_delay_s12。

反之，控制器经 S2 到 S1 再返回控制器的时延，记为 lldp_delay_s21 。交换机收到控制器发来的 Echo 报文后会立即回复控制器，可以利用 Echo Request/Reply 报文求出控制器到 S1、S2 的往返时延，记为 echo_delay_s1 ，echo_delay_s2。则 S1 到 S2 的时延 delay = (lldp_delay_s12 + lldp_delay_s21 - echo_delay_s1 - echo_delay_s2) / 2。

3.1.1 修改 OSKen 与进行代码实现

为完成相关实验操作，需要对 OSKen 做出以下修改。

(1) os_ken/topology/switches.py 的 PortData/__init__()

PortData 记录交换机的端口信息，我们需要增加 self.delay 属性记录上述的 lldp_delayself.timestamp 为 LLDP 包在发送时被打上的时间戳，具体发送的逻辑查看源码。

```
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
        self.sent = 0
        self.delay = 0
```

(2) os_ken/topology/switches.py 的 Switches/lldp_packet_in_handler()

lldp_packet_in_handler()处理接收到的 LLDP 包，在这里用收到 LLDP 报文的时间戳减去发送时的时间戳即为 lldp_delay ，由于 LLDP 报文被设计为经一跳后转给控制器，我们可将 lldp_delay 存入发送 LLDP 包对应的交换机端口。

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def lldp_packet_in_handler(self, ev):
    # add receive timestamp
    recv_timestamp = time.time()
    if not self.link_discovery:
        return
    msg = ev.msg
    try:
        src_dpid, src_port_no = LLDPPacket.lldp_parse(msg.data)
```

```
        except LLDPPacket.LLDPUnknownFormat:
            # This handler can receive all the packets which can be
            # not-LLDP packet. Ignore it silently
            return
        # calc the delay of lldp packet
        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
...
```

（3）对 network_awareness.py 进行修改加入相应功能

①导入 struct 模块

Struct 模块用于将数据打包成二进制字符串以及从二进制字符串中解包数据。在代码中，它主要用于处理 EventOFPEchoReply 事件时，将发送时间以二进制形式存储在 OFPEchoRequest 消息的数据部分，并在收到 OFPEchoReply 时解包获取发送时间，从而计算回声请求的延迟。

②加入类属性初始化

加入的 self.switches 是一个空字典，用于存储交换机的相关信息。在 packet_in_hander 方法中，当首次接收到数据包时，会获取交换机信息并存储在这个字典中，以便后续使用。 self.lldp_delay 字典用于存储链路层发现协议数据包的延迟信息。键是一个元组 (src_dpid, dpid) ，表示源交换机的 DPID 和目标交换机的 DPID，值是相应的延迟。self.echo_delay 用于存储回声请求的延迟信息。键是交换机的 DPID，值是该交换机的回声请求延迟。

③加入 packet_in_hander 方法

该方法用于处理 EventOFPPacketIn 事件，即当交换机向控制器发送数据包时触发。首先，尝试解析接收到的数据包是否为 LLDP 数据包。如果是，则获取源交换机的 DPID 和端口号。若 self.switches 为空，则获取交换机信息。遍历交换机的端口信息，若找到匹配的源交换机和端口号，则将对应的 LLDP 延迟信息存储在 self.lldp_delay 字典中。

④加入 echo_hander 方法

该方法用于处理 EventOFPEchoReply 事件，即当交换机响应控制器的回声请求时触发。记录当前时间作为接收时间 recv_time 。从 OFPEchoReply 消息的数据部分解包获取发送时间 send_time 。计算回声请求的延迟，并将其存储在 self.echo_delay 字典中，键为交换机的 DPID。

⑤加入 echo_send_requests 方法

该方法用于向指定的交换机发送回声请求（Echo Request）。记录当前时间作为发送时间 send_time ，并使用 struct.pack 将其打包成二进制字符串。创建 OFPEchoRequest 消息，并将打包后的发送时间作为数据部分。通过交换机的数据路径发送该消息，并暂停 0.2 秒。

⑥ calculate_link_delay 方法

该方法用于计算两个交换机之间的链路延迟。从 self.lldp_delay 和

self.echo_delay 字典中获取相应的延迟信息，若键不存在则返回默认值 0。根据公式 (lldp_delay_s12 + lldp_delay_s21 - echo_delay_s1 - echo_delay_s2) / 2 计算延迟。确保计算结果为非负数，并返回该值。

（4）对 shortest_path.py 进行修改加入相应功能

①导入 ether_types 模块

ether_types 模块包含以太网帧类型的常量定义，在代码中用于判断以太网帧的类型，如判断是否为链路层发现协议帧或 IPv6 帧。

② 完成 handle_arp 方法的具体实现

首先过滤特定类型的以太网帧，当接收到的以太网帧类型为 LLDP 或 IPv6 时，直接返回，不进行后续处理。当接收到的是广播 ARP 数据包时，检查是否已经记录过该数据包的源 MAC、目的 IP 和接收端口的组合。如果已经记录过且接收端口不同，则认为是重复数据包，直接返回，避免广播环路。同时记录源 MAC 地址和接收端口的映射关系，用于后续的转发决策。若已经学习到目的 MAC 地址对应的端口，则发送流表项到交换机，同时通过 OFPPacketOut 消息将数据包发送到指定端口。

③完善路径延迟计算

在 handle_ipv4 方法中，增加路径延迟的计算功能。通过遍历路径上的每一段链路，从 self.network_awareness.lldp_delay 字典中获取链路延迟，并累加得到总延迟。最后将延迟信息以毫秒为单位记录到日志中。

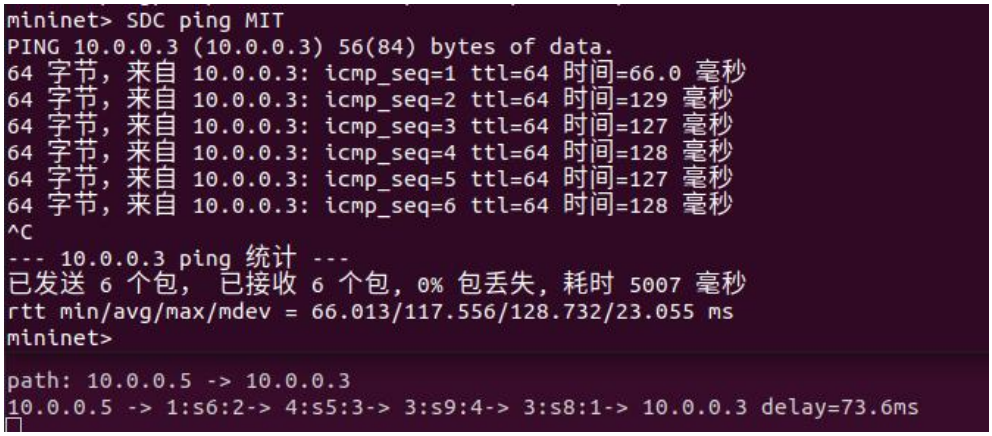
3.1.2 运行测试

执行 sudo python3 topo_1970.py 开启实验拓扑，后运行实验代码，观察输出结果。其中，观察到拓扑连接如下所示。

```
# add edges between switches
self.addLink( s1 , s9, bw=10, delay='10ms')
self.addLink( s2 , s3, bw=10, delay='11ms')
self.addLink( s2 , s4, bw=10, delay='13ms')
self.addLink( s3 , s4, bw=10, delay='14ms')
self.addLink( s4 , s5, bw=10, delay='15ms')
self.addLink( s5 , s9, bw=10, delay='29ms')
self.addLink( s5 , s6, bw=10, delay='17ms')
self.addLink( s6 , s7, bw=10, delay='10ms')
self.addLink( s7 , s8, bw=10, delay='62ms')
self.addLink( s8 , s9, bw=10, delay='17ms')
```

```
# ... and now hosts
h1 = self.addHost( 'HARVARD' )
h2 = self.addHost( 'SRI' )
h3 = self.addHost( 'UCSB' )
h4 = self.addHost( 'UCLA' )
h5 = self.addHost( 'RAND' )
h6 = self.addHost( 'SDC' )
h7 = self.addHost( 'UTAH' )
h8 = self.addHost( 'MIT' )
h9 = self.addHost( 'BBN' )
```

首先进行 SDC ping MIT，计算 RTT 理论值应为 (17 + 29 + 17) * 2 = 126ms



经过验证，发现所测得数值与理论值误差较小，验证了设计的可行性。

3.2 容忍链路故障

3.2.1 处理链路故障原理

当链路状态发生变化时，与之关联的端口状态也会改变，此时会产生 EventOFPPortStatus 事件。通过将该事件与特定的处理函数进行绑定，就能获取端口状态改变的相关信息，进而执行相应的处理操作。这一机制可用于实时感知网络端口状态变化情况，以便及时做出响应。

同时，控制器会采取操作，删除网络拓扑中所有交换机上除默认流表之外的流表项。后续交换机收到数据包，就只能匹配默认流表项，进而向控制器发送 packet_in 消息。控制器接收到消息后，会重新计算最小时延路径，并将新的流表下发至交换机，以适应链路状态变化后的网络情况，保障网络数据传输的高效性。

3.2.2 实现过程

首先对 network_awareness.py 再进行略微修改，即加入如下内容。

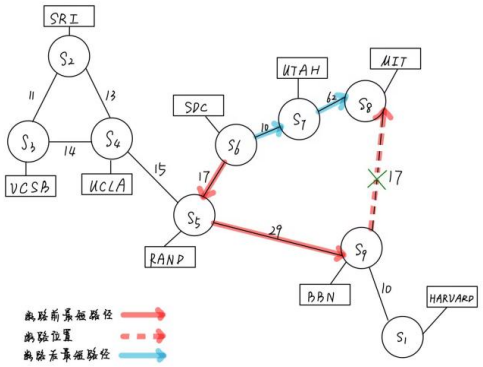
①加入 port_status_handler 方法

该方法用于处理 EventOFPPortStatus 事件，即当交换机端口状态发生变化时触发。当端口状态为添加或修改时，更新交换机的端口信息，清空拓扑图 self.topo_map ，并删除所有与端口相关的流表项。当端口状态为删除时，从交换机的端口信息中移除该端口。最后，向观察者发送端口状态变化的事件。

② 加入 delete_flow 方法

该方法用于删除指定交换机上与特定端口相关的流表项。创建一个匹配规则，匹配指定端口的输入。创建一个 OFPFlowMod 消息，命令为删除流表项。通过交换机的数据路径发送该消息，并记录日志信息。若出现异常，记录错误信息。

3.2.3 实验结果



各节点连接线段单位统一为 ms，且由于美观性，对距离进行拉伸，以数值为准，线段长度仅供参考。

本部分仍以 SDC ping MIT 为例进行测试，首先记录 SDC ping MIT 结果如下所示。

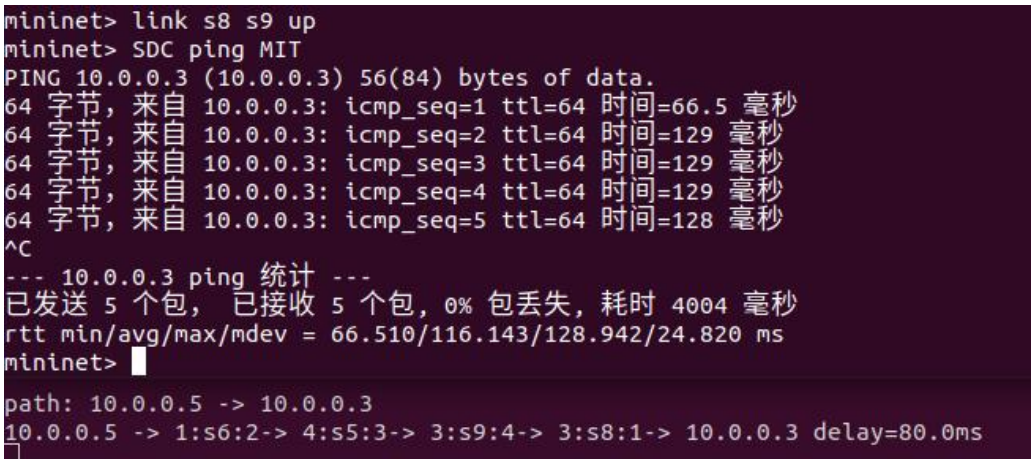
```
mininet> SDC ping MIT
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 字节，来自 10.0.0.3: icmp_seq=1 ttl=64 时间=66.0 毫秒
64 字节，来自 10.0.0.3: icmp_seq=2 ttl=64 时间=128 毫秒
64 字节，来自 10.0.0.3: icmp_seq=3 ttl=64 时间=128 毫秒
64 字节，来自 10.0.0.3: icmp_seq=4 ttl=64 时间=128 毫秒
64 字节，来自 10.0.0.3: icmp_seq=5 ttl=64 时间=127 毫秒
^C
--- 10.0.0.3 ping 统计 ---
已发送 5 个包， 已接收 5 个包，0% 包丢失，耗时 4006 毫秒
rtt min/avg/max/mdev = 65.970/115.209/127.781/24.620 ms
mininet>
path: 10.0.0.5 -> 10.0.0.3
10.0.0.5 -> 1:s6:2-> 4:s5:3-> 3:s9:4-> 3:s8:1-> 10.0.0.3 delay=79.6ms
```

为测试对链路故障的容忍，执行 link s8 s9 down 命令后再次 SDC ping MIT，观察到结果如下所示。

```
mininet> link s8 s9 down
mininet> SDC ping MIT
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 字节，来自 10.0.0.3: icmp_seq=1 ttl=64 时间=74.5 毫秒
64 字节，来自 10.0.0.3: icmp_seq=2 ttl=64 时间=146 毫秒
64 字节，来自 10.0.0.3: icmp_seq=3 ttl=64 时间=145 毫秒
64 字节，来自 10.0.0.3: icmp_seq=4 ttl=64 时间=146 毫秒
64 字节，来自 10.0.0.3: icmp_seq=5 ttl=64 时间=146 毫秒
^C
--- 10.0.0.3 ping 统计 ---
已发送 6 个包， 已接收 5 个包，16.6667% 包丢失，耗时 5009 毫秒
rtt min/avg/max/mdev = 74.547/131.369/146.218/28.415 ms

path: 10.0.0.5 -> 10.0.0.3
10.0.0.5 -> 1:s6:3-> 2:s7:3-> 2:s8:1-> 10.0.0.3 delay=83.1ms
```

再次开启链路，执行 link s8 s9 up 命令，SDC ping MIT 结果如下所示。



根据上述实验结果，观察到 SDC ping MIT 最短路径的时延为(17 + 29 + 17) = 63ms，RTT = 63 * 2 = 126 ms。若 s8 和 s9 链路断开，此时最短路径的时延(10 + 62) = 72ms，72 * 2 = 144ms。若 s9 和 s8 链路重新恢复正常，则最短路径的时延 delay 以及 RTT 也恢复成最初的值。

在上面的示例中，开始时 RTT 约等于 130ms，在 s9 和 s8 之间的链路断开后，RTT ≈ 146 ms，且路径由原来的 10.0.0.5->s6->s5->s9->s8->10.0.0.3 变为 10.0.0.5->s6->s7->s8->10.0.0.3 。而当链路恢复后连接后，RTT 又变为约 128 ms，路径重新变回 10.0.0.5->s6->s5->s9->s8->10.0.0.3。

实验结果与理论分析相近，可见实现了对链路故障的容忍。从上图中可以发现，当链路故障时，最小时延链路相应发生了改变，而当故障恢复后，最小时延链路也得到了恢复。

四、源代码

```
sudo python3 topo_1970.py
uv run osken-manager shortest_forward.py --observe-links
```

shortest_forward.py

```
import eventlet
eventlet.monkey_patch()
from os_ken.base import app_manager
from os_ken.controller import ofp_event
from os_ken.controller.handler import CONFIG_DISPATCHER,
MAIN_DISPATCHER, DEAD_DISPATCHER, HANDSHAKE_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, arp, ipv4
from os_ken.topology import event
import sys
```

```
import networkx as nx
from os_ken.lib.packet import ether_types
from network_awareness import NetworkAwareness

ETHERNET = ethernet.ethernet.__name__
ETHERNET_MULTICAST = "ff:ff:ff:ff:ff:ff"
ARP = arp.arp.__name__

class LeastHops(app_manager.OSKenApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {
        'network_awareness': NetworkAwareness
    }

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.network_awareness = kwargs['network_awareness']
        self.weight = 'delay'
        self.mac_to_port = {}
        self.sw = {}
        self.path = None

    def add_flow(self, datapath, priority, match, actions, idle_timeout=0,
hard_timeout=0):
        try:
            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)
        except Exception as e:
            self.logger.error(f"Failed to add flow: {e}")

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        try:
            msg = ev.msg
```

```
            dp = msg.datapath
            ofp = dp.ofproto
            parser = dp.ofproto_parser

            dpid = dp.id
            in_port = msg.match['in_port']

            pkt = packet.Packet(msg.data)
            eth_pkt = pkt.get_protocol(ethernet.ethernet)
            arp_pkt = pkt.get_protocol(arp.arp)
            ipv4_pkt = pkt.get_protocol(ipv4.ipv4)

            pkt_type = eth_pkt.ethertype

            # layer 2 self-learning
            dst_mac = eth_pkt.dst
            src_mac = eth_pkt.src

            if isinstance(arp_pkt, arp.arp):
                self.handle_arp(msg,  in_port,  dst_mac,  src_mac,  pkt,
pkt_type)

            if isinstance(ipv4_pkt, ipv4.ipv4):
                self.handle_ipv4(msg,    ipv4_pkt.src,    ipv4_pkt.dst,
pkt_type)
        except Exception as e:
            self.logger.error(f"Error in packet_in_handler: {e}")

    def handle_arp(self, msg, in_port, dst, src, pkt, pkt_type):
        try:
            dp = msg.datapath
            ofp = dp.ofproto
            parser = dp.ofproto_parser

            # the identity of switch
            dpid = dp.id
            self.mac_to_port.setdefault(dpid, {})

            eth_pkt = pkt.get_protocol(ethernet.ethernet)
            if    eth_pkt.ethertype    in    [ether_types.ETH_TYPE_LLDP,
ether_types.ETH_TYPE_IPV6]:
                return

            # get the mac
```

```
            dst = eth_pkt.dst
            src = eth_pkt.src

            # get protocols
            header_list = {p.protocol_name: p for p in pkt.protocols if not
isinstance(p, str)}
            if dst == ETHERNET_MULTICAST and ARP in header_list:
                arp_packet = header_list[ARP]
                dst_ip = arp_packet.dst_ip
                if (dpid, src, dst_ip) in self.sw:
                    if self.sw[(dpid, src, dst_ip)] != in_port:
                        return
                else:
                    self.sw[(dpid, src, dst_ip)] = in_port

            # self-learning
            # Record the mapping relationship between src_mac to in_port
            self.mac_to_port[dpid][src] = in_port

            out_port = self.mac_to_port[dpid].get(dst, ofp.OFPP_FLOOD)

            actions = [parser.OFPActionOutput(out_port)]

            if out_port != ofp.OFPP_FLOOD:
                match = parser.OFPMatch(in_port=in_port, eth_src=src,
eth_dst=dst)
                self.add_flow(dp, 1, match, actions, idle_timeout=5,
hard_timeout=5)

            out = parser.OFPPacketOut(
                datapath=dp, buffer_id=msg.buffer_id, in_port=in_port,
                actions=actions, data=msg.data)
            dp.send_msg(out)
        except Exception as e:
            self.logger.error(f"Error in handle_arp: {e}")

    def handle_ipv4(self, msg, src_ip, dst_ip, pkt_type):
        try:
            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)

            # calc path delay
            delay = 0
            for i in range(len(dpid_path) - 1):
                src_dpid = dpid_path[i]
                dst_dpid = dpid_path[i + 1]
                delay += self.network_awareness.lldp_delay.get((src_dpid,
dst_dpid), 0)
            self.logger.info("delay= %.5fms", delay * 1000)
            self.logger.info("time= %.5fms", delay * 2000)

            # send flow mod
            for node in port_path:
                in_port, dpid, out_port = node
                self.send_flow_mod(parser, dpid, pkt_type, src_ip, dst_ip,
in_port, out_port)
                self.send_flow_mod(parser, dpid, pkt_type, dst_ip, src_ip,
out_port, in_port)

            # send packet_out
            _, dpid, out_port = port_path[-1]
            dp = self.network_awareness.switch_info[dpid]
            actions = [parser.OFPActionOutput(out_port)]
            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)
        except Exception as e:
            self.logger.error(f"Error in handle_ipv4: {e}")

    def send_flow_mod(self, parser, dpid, pkt_type, src_ip, dst_ip,
in_port, out_port):
```

```
        try:
            dp = self.network_awareness.switch_info[dpid]
            match = parser.OFPMatch(
                in_port=in_port,   eth_type=pkt_type,   ipv4_src=src_ip,
ipv4_dst=dst_ip)
            actions = [parser.OFPActionOutput(out_port)]
            self.add_flow(dp, 1, match, actions, 10, 30)
        except Exception as e:
            self.logger.error(f"Error in send_flow_mod: {e}")

    def show_path(self, src, dst, port_path):
        path_str = f'path: {src} -> {dst}'
        self.logger.info(path_str)
        path = src + ' -> '
        for node in port_path:
            path += '{}:s{}:{}'.format(*node) + ' -> '
        path += dst
        self.logger.info(path)
```

network_awareness.py

```
import eventlet
eventlet.monkey_patch()

from os_ken.base import app_manager
from os_ken.controller import ofp_event
from     os_ken.controller.handler     import     CONFIG_DISPATCHER,
MAIN_DISPATCHER, DEAD_DISPATCHER, HANDSHAKE_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, arp, ipv4
from os_ken.topology import event
import sys
import networkx as nx
from os_ken.lib.packet import ether_types
from network_awareness import NetworkAwareness

ETHERNET = ethernet.ethernet.__name__
ETHERNET_MULTICAST = "ff:ff:ff:ff:ff:ff"
ARP = arp.arp.__name__


class LeastHops(app_manager.OSKenApp):
```

```
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {
        'network_awareness': NetworkAwareness
    }

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.network_awareness = kwargs['network_awareness']
        self.weight = 'hop'
        self.mac_to_port = {}
        self.sw = {}
        self.path = None

    def add_flow(self, datapath, priority, match, actions, idle_timeout=0,
hard_timeout=0):
        try:
            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)
        except Exception as e:
            self.logger.error(f"Failed to add flow: {e}")

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        try:
            msg = ev.msg
            dp = msg.datapath
            ofp = dp.ofproto
            parser = dp.ofproto_parser

            dpid = dp.id
            in_port = msg.match['in_port']

            pkt = packet.Packet(msg.data)
            eth_pkt = pkt.get_protocol(ethernet.ethernet)
```

```
            arp_pkt = pkt.get_protocol(arp.arp)
            ipv4_pkt = pkt.get_protocol(ipv4.ipv4)

            pkt_type = eth_pkt.ethertype

            # layer 2 self-learning
            dst_mac = eth_pkt.dst
            src_mac = eth_pkt.src

            if isinstance(arp_pkt, arp.arp):
                self.handle_arp(msg,  in_port,  dst_mac,  src_mac,  pkt,
pkt_type)

            if isinstance(ipv4_pkt, ipv4.ipv4):
                self.handle_ipv4(msg,    ipv4_pkt.src,    ipv4_pkt.dst,
pkt_type)
        except Exception as e:
            self.logger.error(f"Error in packet_in_handler: {e}")

    def handle_arp(self, msg, in_port, dst, src, pkt, pkt_type):
        try:
            dp = msg.datapath
            ofp = dp.ofproto
            parser = dp.ofproto_parser

            # the identity of switch
            dpid = dp.id
            self.mac_to_port.setdefault(dpid, {})

            eth_pkt = pkt.get_protocol(ethernet.ethernet)
            if    eth_pkt.ethertype    in    [ether_types.ETH_TYPE_LLDP,
ether_types.ETH_TYPE_IPV6]:
                return

            # get the mac
            dst = eth_pkt.dst
            src = eth_pkt.src

            # get protocols
            header_list = {p.protocol_name: p for p in pkt.protocols if not
isinstance(p, str)}
            if dst == ETHERNET_MULTICAST and ARP in header_list:
                arp_packet = header_list[ARP]
                dst_ip = arp_packet.dst_ip
```

```
            if (dpid, src, dst_ip) in self.sw:
                if self.sw[(dpid, src, dst_ip)] != in_port:
                    return
            else:
                self.sw[(dpid, src, dst_ip)] = in_port

        # self-learning
        # Record the mapping relationship between src_mac to in_port
        self.mac_to_port[dpid][src] = in_port

        out_port = self.mac_to_port[dpid].get(dst, ofp.OFPP_FLOOD)

        actions = [parser.OFPActionOutput(out_port)]

        if out_port != ofp.OFPP_FLOOD:
            match = parser.OFPMatch(in_port=in_port, eth_src=src,
eth_dst=dst)
            self.add_flow(dp, 1, match, actions, idle_timeout=5,
hard_timeout=5)

        out = parser.OFPPacketOut(
            datapath=dp, buffer_id=msg.buffer_id, in_port=in_port,
            actions=actions, data=msg.data)
        dp.send_msg(out)
    except Exception as e:
        self.logger.error(f"Error in handle_arp: {e}")

def handle_ipv4(self, msg, src_ip, dst_ip, pkt_type):
    try:
        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)

            # calc path delay
            delay = 0
            for i in range(len(dpid_path) - 1):
                src_dpid = dpid_path[i]
                dst_dpid = dpid_path[i + 1]
                delay += self.network_awareness.lldp_delay.get((src_dpid,
dst_dpid), 0)
            self.logger.info("delay= %.5fms", delay * 1000)
            self.logger.info("time= %.5fms", delay * 2000)

            # send flow mod
            for node in port_path:
                in_port, dpid, out_port = node
                self.send_flow_mod(parser, dpid, pkt_type, src_ip, dst_ip,
in_port, out_port)
                self.send_flow_mod(parser, dpid, pkt_type, dst_ip, src_ip,
out_port, in_port)

            # send packet_out
            _, dpid, out_port = port_path[-1]
            dp = self.network_awareness.switch_info[dpid]
            actions = [parser.OFPActionOutput(out_port)]
            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)
        except Exception as e:
            self.logger.error(f"Error in handle_ipv4: {e}")

    def send_flow_mod(self, parser, dpid, pkt_type, src_ip, dst_ip,
in_port, out_port):
        try:
            dp = self.network_awareness.switch_info[dpid]
            match = parser.OFPMatch(
                in_port=in_port,   eth_type=pkt_type,   ipv4_src=src_ip,
ipv4_dst=dst_ip)
            actions = [parser.OFPActionOutput(out_port)]
            self.add_flow(dp, 1, match, actions, 10, 30)
        except Exception as e:
            self.logger.error(f"Error in send_flow_mod: {e}")
```

```
def show_path(self, src, dst, port_path):
    path_str = f'path: {src} -> {dst}'
    self.logger.info(path_str)
    path = src + ' -> '
    for node in port_path:
        path += '{}:s{}:{}'.format(*node) + ' -> '
    path += dst
    self.logger.info(path)
```