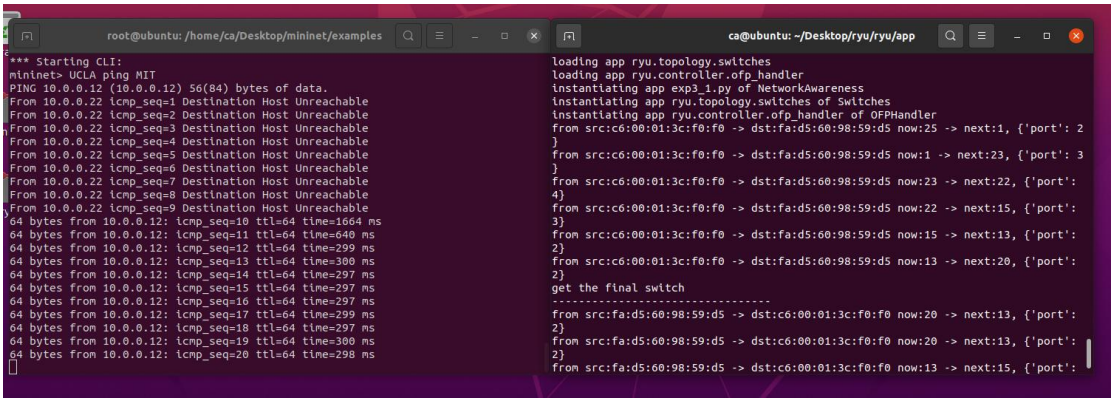
西安交通大学 SDN 第三次实验

Exp1:

在 mininet 中执行命令: python Arpanet19723.py --controller remote

在 ryu 中执行命令: ryu-manager exp3_1.py --observe-links

结果如图:



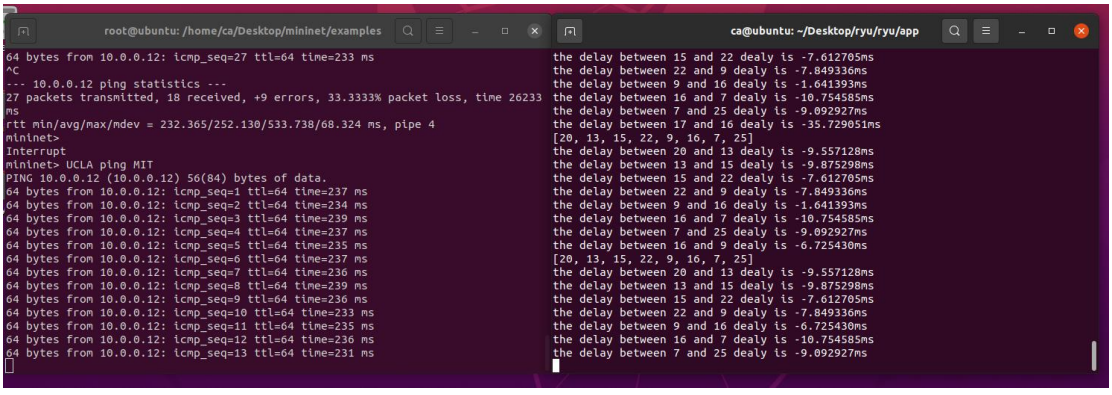
可以看到，最小跳数为 25->1->23->22->15->13->20

EXP2:

在 mininet 中执行命令: python Arpanet19723.py --controller remote

在 ryu 中执行命令: ryu-manager exp3_2.py --observe-links

结果如图:



可以看到，最小时延的路径为: 20->13->15->22->9->16->7->25

整体思路:

Exp1 中，当控制器收到 packetin 包的时候，调用 packet_in_handler()函数，如果是 lldp 包或者是 ipv6 包则 return。如果是 arp 包，则将其 ip 地址、交换机、源地址和端口相对应（学习过程）。根据该包内的 mac 地址确定其传播方式（和第一个实验相同）。然后根据所给的拓扑，依赖 networkx 构建相应的拓扑图，然后调用 api 算出最短路径。（其中每条路径的权值都是 1）

a.在初始化的时候，添加以下列表：

self.mac_table = {}

#学习 mac 地址，确定输出端口

self.arp_anti_loop = {} #用于打破环路

self.arp_table = {}

self.topo_thread = hub.spawn(self._get_topology)

self.graph = nx.DiGraph()

#创建一个图利用

networkx 自带 api 计算最短路径

b.过滤 lldp 包和 ipv6 包（ipv6 不使用 arp 协议）

ignore lldp packet

if eth_pkt.ethertype == ether_types.ETH_TYPE_LLDP:

return

if eth_pkt.ethertype == ether_types.ETH_TYPE_IPV6:return

c.获取 header list（进而获取 ip 地址）

```
header_list
=
dict((p.protocol_name,
p)for
p
in
pkt.protocols if type(p) != str)
```

d.根据 arp 包学习的过程，打破环路

```
if dst == ETHERNET_MULTICAST and ARP in header_list:
#如
果是 arp 包
arp_dst_ip = header_list[ARP].dst_ip
if (dpid, src, arp_dst_ip) in self.arp_anti_loop:
if
self.arp_anti_loop[(dpid,
src,
arp_dst_ip)] != in_port:
out = parser.OFPPacketOut(
datapath=dp,
buffer_id=ofp.OFP_NO_BUFFER,
in_port=in_port,
```

```
actions=[], data=None)

dp.send_msg(out)

return

else:

self.arp_anti_loop[(dpid, src, arp_dst_ip)]

= in_porte.获取输出端口

if self.mac_table[dpid].has_key(dst):

out_port = self.mac_table[dpid][dst]

else:

out_port = ofp.OFPP_FLOOD
```

Exp2 中, a.初始化时候，添加如下列表

```
self.mac_to_port = {}

self.network = nx.DiGraph()

self.graph = nx.DiGraph()

self.paths = {}

self.topology_api_app=self

self.echo_latency={}

self.request_latency={}

self.datapaths={}
```

self.sw_module = lookup_service_brick('switches')

self.awareness = lookup_service_brick('awareness')

self.network_aware

=

lookup_service_brick('network_aware')b.学习 mac 地址的函数

mac_learning() #与问题一相同，不在赘

述

def mac_learning(self, datapath, src, in_port):

self.mac_to_port.setdefault((datapath,datapath.id),

{})

learn a mac address to avoid FLOOD next time.

if src in self.mac_to_port[(datapath,datapath.id)]:

if

in_port

!=

self.mac_to_port[(datapath,datapath.id)][src]:

return False

else:

self.mac_to_port[(datapath,datapath.id)][src] =

in_port

return True

c.计算延时的函数

```
def get_delay(self, src, dst):

try:

fwd_delay = self.request_latency[(src,dst)]

re_delay = self.request_latency[(dst,src)]

src_latency = self.echo_latency[src]

dst_latency = self.echo_latency[dst]

delay = (fwd_delay + re_delay - src_latency -dst_latency)*(1000/2)

print('the

delay

between

%s

and

%s

dealy

is %fms'%(src,dst,delay))

return max(delay, 0)

except:

print("get delay error")

return float('inf')
```

d.获取 lldp 的延时，然后赋值给相应拓扑图中的路径，作为其权值。

```
if eth.ethertype == ether_types.ETH_TYPE_LLDP:

try:
```

```
src_dpid,
src_port_no
=
LLDPPacket.lldp_parse(msg.data)
dpid = datapath.id
if self.sw_module is None:
self.sw_module
=
lookup_service_brick('switches')
if src_dpid not in self.paths.keys():
self.paths.setdefault(src_dpid, {})
for port in self.sw_module.ports.keys():
if src_dpid == port.dpid and src_port_no
== port.port_no:
port_data
=self.sw_module.ports[port]
delay = port_data.delay
self.request_latency[(src_dpid,dpid)] = delay
# print('lldp delay between %s and %s
is %fms'%(src_dpid,dpid,delay*1000))
self.network[src_dpid][dpid]['weight']
= self.get_delay(src_dpid, dpid)
```

```
if dpid in self.network:
if
dpid
not
in
self.paths[src_dpid]:
path
=
nx.shortest_path(self.network,src_dpid,dpid)
self.paths[src_dpid][dpid]=path
path = nx.shortest_path(self.network,20,25,
weight='weight')
print(path)
total = 0
for i in range(len(path)-1):
total += self.get_delay(path[i], path[i+1])
# print("total:", total)except Exception as e:
print(e)
print("error occured")
finally:
return
```

e.确定其输出端口之后基本上与问题一相同，不在赘述，包含建立

topo 和计算最短路径（带权值）

if dst in self.mac_to_port[(datapath,datapath.id)]:

out_port

=

self.mac_to_port[(datapath,datapath.id)][dst]

else:

if self.mac_learning(datapath, src, in_port) is

False:

out_port = ofproto.OFPPC_NO_RECV

else:

out_port = ofproto.OFPP_FLOOD

源码:

Arpanet19723:

```
#!/usr/bin/python

"""
Custom topology for Mininet, generated by GraphML-Topo-to-Mininet-Network-Generator.
"""
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.node import Node
from mininet.node import CPULimitedHost
from mininet.link import TCLink
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.util import dumpNodeConnections
```

```
class GeneratedTopo( Topo ):
    "Internet Topology Zoo Specimen."

    def __init__( self, **opts ):
        "Create a topology."

        # Initialize Topology
        Topo.__init__( self, **opts )

        # add nodes, switches first...
        s1 = self.addSwitch( 's1' )
        s2 = self.addSwitch( 's2' )
        s3 = self.addSwitch( 's3' )
        s4 = self.addSwitch( 's4' )
        s5 = self.addSwitch( 's5' )
        s6 = self.addSwitch( 's6' )
        s7 = self.addSwitch( 's7' )
        s8 = self.addSwitch( 's8' )
        s9 = self.addSwitch( 's9' )
        s10 = self.addSwitch( 's10' )
        s11 = self.addSwitch( 's11' )
        s12 = self.addSwitch( 's12' )
        s13 = self.addSwitch( 's13' )
        s14 = self.addSwitch( 's14' )
        s15 = self.addSwitch( 's15' )
        s16 = self.addSwitch( 's16' )
        s17 = self.addSwitch( 's17' )
        s18 = self.addSwitch( 's18' )
        s19 = self.addSwitch( 's19' )
        s20 = self.addSwitch( 's20' )
        s21 = self.addSwitch( 's21' )
        s22 = self.addSwitch( 's22' )
        s23 = self.addSwitch( 's23' )
        s24 = self.addSwitch( 's24' )
        s25 = self.addSwitch( 's25' )

        # ... and now hosts
        h1 = self.addHost( 'ILLINOIS' )
        h2 = self.addHost( 'MITRE' )
        h3 = self.addHost( 'CARNEGIE' )
        h4 = self.addHost( 'CASE' )
        h5 = self.addHost( 'ETAC' )
        h6 = self.addHost( 'AFGWC' )
```

```
h7 = self.addHost( 'BBN' )
h8 = self.addHost( 'NBS' )
h9 = self.addHost( 'Tinker' )
h10 = self.addHost( 'AMES' )
h11 = self.addHost( 'RADC' )
h12 = self.addHost( 'McClellan' )
h13 = self.addHost( 'RAND' )
h14 = self.addHost( 'AMES13' )
h15 = self.addHost( 'SDC' )
h16 = self.addHost( 'BBN15' )
h17 = self.addHost( 'HARVARD' )
h18 = self.addHost( 'SRI' )
h19 = self.addHost( 'UCSB' )
h20 = self.addHost( 'UCLA' )
h21 = self.addHost( 'Stanford' )
h22 = self.addHost( 'USC' )
h23 = self.addHost( 'UTAH' )
h24 = self.addHost( 'Lincoln' )
h25 = self.addHost( 'MIT' )

# add edges between switch and corresponding host
self.addLink( s1 , h1 )
self.addLink( s2 , h2 )
self.addLink( s3 , h3 )
self.addLink( s4 , h4 )
self.addLink( s5 , h5 )
self.addLink( s6 , h6 )
self.addLink( s7 , h7 )
self.addLink( s8 , h8 )
self.addLink( s9 , h9 )
self.addLink( s10 , h10 )
self.addLink( s11 , h11 )
self.addLink( s12 , h12 )
self.addLink( s13 , h13 )
self.addLink( s14 , h14 )
self.addLink( s15 , h15 )
self.addLink( s16 , h16 )
self.addLink( s17 , h17 )
self.addLink( s18 , h18 )
self.addLink( s19 , h19 )
self.addLink( s20 , h20 )
self.addLink( s21 , h21 )
self.addLink( s22 , h22 )
self.addLink( s23 , h23 )
```

```
        self.addLink( s24 , h24 )
        self.addLink( s25 , h25 )

        # add edges between switches
        self.addLink( s1 , s25, bw=10, delay='50ms')
        self.addLink( s1 , s23, bw=10, delay='34ms')
        self.addLink( s2 , s3, bw=10, delay='13ms')
        self.addLink( s2 , s5, bw=10, delay='14ms')
        self.addLink( s3 , s4, bw=10, delay='15ms')
        self.addLink( s4 , s11, bw=10, delay='12ms')
        self.addLink( s4 , s6, bw=10, delay='17ms')
        self.addLink( s5 , s8, bw=10, delay='10ms')
        self.addLink( s7 , s25, bw=10, delay='18ms')
        self.addLink( s7 , s16, bw=10, delay='17ms')
        self.addLink( s8 , s17, bw=10, delay='13ms')
        self.addLink( s9 , s22, bw=10, delay='14ms')
        self.addLink( s9 , s16, bw=10, delay='19ms')
        self.addLink( s10 , s18, bw=10, delay='14ms')
        self.addLink( s10 , s14, bw=10, delay='15ms')
        self.addLink( s11 , s24, bw=10, delay='17ms')
        self.addLink( s12 , s18, bw=10, delay='40ms')
        self.addLink( s12 , s23, bw=10, delay='44ms')
        self.addLink( s13 , s20, bw=10, delay='15ms')
        self.addLink( s13 , s21, bw=10, delay='18ms')
        self.addLink( s13 , s15, bw=10, delay='15ms')
        self.addLink( s14 , s21, bw=10, delay='19ms')
        self.addLink( s15 , s22, bw=10, delay='15ms')
        self.addLink( s16 , s17, bw=10, delay='12ms')
        self.addLink( s18 , s19, bw=10, delay='44ms')
        self.addLink( s19 , s20, bw=10, delay='48ms')
        self.addLink( s22 , s23, bw=10, delay='16ms')
        self.addLink( s24 , s25, bw=10, delay='13ms')

topos = { 'generated': ( lambda: GeneratedTopo() ) }

# HERE THE CODE DEFINITION OF THE TOPOLOGY ENDS

# the following code produces an executable script working with a remote controller
# and providing ssh access to the the mininet hosts from within the ubuntu vm
controller_ip = ''

def setupNetwork(controller_ip):
    "Create network and run simple performance test"
    # check if remote controller's ip was set
```

```
            # else set it to localhost
            topo = GeneratedTopo()
            if controller_ip == '':
                #controller_ip = '10.0.2.2';
                controller_ip = '127.0.0.1';
            net = Mininet(topo=topo, controller=lambda a: RemoteController( a, ip=controller_ip,
port=6633 ), host=CPULimitedHost, link=TCLink)
            return net

        def connectToRootNS( network, switch, ip, prefixLen, routes ):
            "Connect hosts to root namespace via switch. Starts network."
            "network: Mininet() network object"
            "switch: switch to connect to root namespace"
            "ip: IP address for root namespace node"
            "prefixLen: IP address prefix length (e.g. 8, 16, 24)"
            "routes: host networks to route to"
            # Create a node in root namespace and link to switch 0
            root = Node( 'root', inNamespace=False )
            intf = TCLink( root, switch ).intf1
            root.setIP( ip, prefixLen, intf )
            # Start network that now includes link to root namespace
            network.start()
            # Add routes from root ns to hosts
            for route in routes:
                root.cmd( 'route add -net ' + route + ' dev ' + str( intf ) )

        def sshd( network, cmd='/usr/sbin/sshd', opts='-D' ):
            "Start a network, connect it to root ns, and run sshd on all hosts."
            switch = network.switches[ 0 ]  # switch to use
            ip = '10.123.123.1'  # our IP address on host network
            routes = [ '10.0.0.0/8' ]  # host networks to route to
            connectToRootNS( network, switch, ip, 8, routes )
            for host in network.hosts:
                host.cmd( cmd + ' ' + opts + '&' )

            # DEBUGGING INFO
            print("\n")
            print("Dumping host connections")
            dumpNodeConnections(network.hosts)
            print("\n")
            print("*** Hosts are running sshd at the following addresses:")
            print("\n")
            for host in network.hosts:
                print(host.name, host.IP())
```

```
            print("\n")
            print("*** Type 'exit' or control-D to shut down network")
            print("\n")
            print("*** For testing network connectivity among the hosts, wait a bit for the controller to create all the routes, then do 'pingall' on the mininet console.")
            print("\n")

            CLI( network )
            for host in network.hosts:
                host.cmd( 'kill %' + cmd )
            network.stop()

        # by zys
        def start_network(network):
            network.start()

            # DEBUGGING INFO
            print("\n")
            print("Dumping host connections")
            dumpNodeConnections(network.hosts)

            print("\n")
            for host in network.hosts:
                print(host.name, host.IP())

            print("\n")
            print("*** Type 'exit' or control-D to shut down network")
            print("\n")
            print("*** For testing network connectivity among the hosts, wait a bit for the controller to create all the routes, then do 'pingall' on the mininet console.")
            print("\n")

            print("*** edited for xjtu sdn_exp_2020")
            print("\n")

            CLI( network )
            network.stop()

        if __name__ == '__main__':
            setLogLevel('info')
            #setLogLevel('debug')
            # sshd( setupNetwork(controller_ip) )
            start_network(setupNetwork(controller_ip))
```

exp3_1:

```
from ryu.base import app_manager
from ryu.ofproto import ofproto_v1_3
from ryu.controller.handler import set_ev_cls
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
from ryu.controller import ofp_event
from ryu.lib.packet import packet
from ryu.lib.packet import arp
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.lib import hub
from ryu.topology.api import get_link, get_switch

import networkx as nx

class NetworkAwareness(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

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

        self.dpid_mac_port = {}
        self.arp_record = {}
        self.topo_thread = hub.spawn(self._get_topology)
        self.network = nx.DiGraph()

    @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

        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        dpid = dp.id
        in_port = msg.match['in_port']
        src = eth_pkt.src
        dst = eth_pkt.dst

        if eth_pkt.ethertype == ether_types.ETH_TYPE_LLDP or eth_pkt.ethertype==
ether_types.ETH_TYPE_IPV6:
            return
```

```
                header_list = dict( (p.protocol_name, p) for p in pkt.protocols if type(p) != str    )

                if src not in self.dpid_mac_port:
                    self.dpid_mac_port[src] = (dpid, in_port)

                out_port = ofp.OFPP_FLOOD
                if dst in self.dpid_mac_port:
                    if dpid == self.dpid_mac_port[dst][0]:
                        self.logger.info('get the final switch')
                        print ('---------------------------------')

                        out_port = self.dpid_mac_port[dst][1]
                    else:
                        start = self.dpid_mac_port[src][0]
                        end = self.dpid_mac_port[dst][0]

                        path = nx.shortest_path(self.network, start, end)
                        if dpid not in path:
                            return
                        else:
                            next_hop = path[path.index(dpid)+1]
                            self.logger.info('from src:%s -> dst:%s now:%s -> next:%s, %s', src, dst,
dpid, next_hop, self.network[dpid][next_hop])
                            out_port = self.network[dpid][next_hop]['port']

                else:
                    ARP = arp.arp.__name__
                    if dst == "ff:ff:ff:ff:ff:ff" and ARP in header_list:
                        arp_dst_ip = header_list[ARP].dst_ip
                        if (dpid, src, arp_dst_ip) in self.arp_record:
                            if self.arp_record[(dpid, src, arp_dst_ip)] != in_port:
                                return
                        else:
                            self.arp_record[(dpid, src, arp_dst_ip)] = in_port

                actions = [parser.OFPActionOutput(out_port)]

                if out_port != ofp.OFPP_FLOOD:
                    match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
                    self.add_flow(dp, 1, match, actions)
                data = msg.data if msg.buffer_id == ofp.OFP_NO_BUFFER else None

                out = parser.OFPPacketOut(datapath=dp, buffer_id=msg.buffer_id,in_port=in_port,
```

```
actions=actions, data=data)
                dp.send_msg(out)

        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=inst)
            dp.send_msg(mod)

        @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
        def switch_features_handler(self, ev):
            msg = ev.msg

            dp = msg.datapath
            ofp = dp.ofproto
            parser = dp.ofproto_parser
            match = parser.OFPMatch()
            actions =
[parser.OFPActionOutput(ofp.OFPP_CONTROLLER,ofp.OFPCML_NO_BUFFER)]
            self.add_flow(dp, 0, match, actions)

        def _get_topology(self):
            while True:

                switch_list = get_switch(self, None)
                switches = [switch.dp.id for switch in switch_list]
                self.network.add_nodes_from(switches)

                # get links
                links_list = get_link(self, None)
                links = [(link.src.dpid, link.dst.dpid, {'port':link.src.port_no}) for link in links_list]
                self.network.add_edges_from(links)

                # get reverse links
                links = [(link.dst.dpid, link.src.dpid, {'port':link.dst.port_no}) for link in links_list]
                self.network.add_edges_from(links)
                hub.sleep(.1)


    exp3_2:
```

```
        from ryu.base import app_manager
        from ryu.controller import ofp_event
        from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
        from ryu.controller.handler import set_ev_cls
        from ryu.ofproto import ofproto_v1_3
        from ryu.lib.packet import packet
        from ryu.lib.packet import ethernet
        from ryu.lib.packet import tcp
        from ryu.lib.packet import ether_types
        from ryu.lib.packet import arp
        import networkx as nx
        from ryu.topology.api import get_switch,get_link
        from ryu.topology import event
        from ryu.base.app_manager import lookup_service_brick
        import time
        from ryu.lib import hub
        from ryu.controller.handler import MAIN_DISPATCHER,
DEAD_DISPATCHER,CONFIG_DISPATCHER
        from ryu.topology.switches import Switches
        from ryu.topology.switches import LLDPPacket
        class ARP_PROXY_13(app_manager.RyuApp):
                OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
                def __init__(self, *args, **kwargs):
                        super(ARP_PROXY_13, self).__init__(*args, **kwargs)
                        self.mac_to_port = {}
                        self.network = nx.DiGraph()
                        self.graph = nx.DiGraph()
                        self.paths = {}
                        self.topology_api_app=self
                        self.echo_latency={}
                        self.request_latency={}
                        self.datapaths={}
                        self.sw_module = lookup_service_brick('switches')
                        self.awareness = lookup_service_brick('awareness')
                        self.network_aware = lookup_service_brick('network_aware')

                @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
                def switch_features_handler(self, ev):
                        datapath = ev.msg.datapath
                        ofproto = datapath.ofproto
                        parser = datapath.ofproto_parser
                        match = parser.OFPMatch()
                        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCML_NO_BUFFER)]
```

```
                self.add_flow(datapath, 0, match, actions)
        def add_flow(self, datapath, priority, match, actions, buffer_id=None):
                ofproto = datapath.ofproto
                parser = datapath.ofproto_parser
                inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
                if buffer_id:
                        mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
priority=priority, match=match, instructions=inst)
                else:
                        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
match=match, instructions=inst)
                datapath.send_msg(mod)
        def mac_learning(self, datapath, src, in_port):
                self.mac_to_port.setdefault((datapath,datapath.id), {})
                # learn a mac address to avoid FLOOD next time.
                if src in self.mac_to_port[(datapath,datapath.id)]:
                        if in_port != self.mac_to_port[(datapath,datapath.id)][src]:
                                return False
                else:
                        self.mac_to_port[(datapath,datapath.id)][src] = in_port
                        return True

        @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
        def _packet_in_handler(self, ev):
                msg = ev.msg
                datapath = msg.datapath
                ofproto = datapath.ofproto
                parser = datapath.ofproto_parser
                in_port = msg.match['in_port']
                pkt = packet.Packet(msg.data)
                eth = pkt.get_protocols(ethernet.ethernet)[0]
                dst = eth.dst
                src = eth.src
                dpid=datapath.id
                self.mac_learning(datapath, src, in_port)
                if eth.ethertype == ether_types.ETH_TYPE_LLDP:
                        try:
                                src_dpid, src_port_no = LLDPPacket.lldp_parse(msg.data)
                                dpid = datapath.id
                                if self.sw_module is None:
                                        self.sw_module = lookup_service_brick('switches')
                                if src_dpid not in self.paths.keys():
                                        self.paths.setdefault(src_dpid, {})
                                for port in self.sw_module.ports.keys():
```

```
                                    if src_dpid == port.dpid and src_port_no == port.port_no:
                                        port_data = self.sw_module.ports[port]
                                        delay = port_data.delay
                                        self.request_latency[(src_dpid,dpid)] = delay
                                        # print('lldp delay between %s and %s
is %fms'%(src_dpid,dpid,delay*1000))
                                        self.network[src_dpid][dpid]['weight'] =
self.get_delay(src_dpid, dpid)
                                        if dpid in self.network:
                                            if dpid not in self.paths[src_dpid]:
                                                path =
nx.shortest_path(self.network,src_dpid,dpid)
                                                self.paths[src_dpid][dpid]=path
                                path = nx.shortest_path(self.network,20,25, weight='weight')
                                print(path)
                                total = 0
                                for i in range(len(path)-1):
                                    total += self.get_delay(path[i], path[i+1])
                                        # print("total:", total)
                            except Exception as e:
                                print(e)
                                print("error occured")
                            finally:
                                return
                        if dst in self.mac_to_port[(datapath,datapath.id)]:
                            out_port = self.mac_to_port[(datapath,datapath.id)][dst]
                        else:
                            if self.mac_learning(datapath, src, in_port) is False:
                                out_port = ofproto.OFPPC_NO_RECV
                            else:
                                out_port = ofproto.OFPP_FLOOD

                        actions = [parser.OFPActionOutput(out_port)]
                        if out_port != ofproto.OFPP_FLOOD:
                            match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
                            if msg.buffer_id != ofproto.OFP_NO_BUFFER:
                                self.add_flow(datapath, 10, match, actions, msg.buffer_id)
                                return
                            else:
                                self.add_flow(datapath, 10, match, actions)
                        data = None
                        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
                            data = msg.data
                        out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
```

```
in_port=in_port, actions=actions, data=data)
                datapath.send_msg(out)

        def _send_echo_request(self):
                for datapath in self.datapaths.values():
                        parser = datapath.ofproto_parser
                        data = "%.6f" % time.time()
                        data=data.encode('utf-8')
                        echo_req = parser.OFPEchoRequest(datapath, data=data)
                        datapath.send_msg(echo_req)

        @set_ev_cls(ofp_event.EventOFPEchoReply, MAIN_DISPATCHER)
        def echo_reply_handler(self, ev):
                try:
                        latency = time.time() - eval(ev.msg.data)
                        if ev.msg.datapath.id not in self.echo_latency.keys():
                                self.echo_latency.setdefault(ev.msg.datapath.id,{})
                        self.echo_latency[ev.msg.datapath.id] = latency
                        # print('echo latency %s is %fms'%(ev.msg.datapath.id, latency*1000))
                except:
                        print("echo reply handler error")
                        return

        def get_delay(self, src, dst):
                try:
                        fwd_delay = self.request_latency[(src,dst)]
                        re_delay = self.request_latency[(dst,src)]
                        src_latency = self.echo_latency[src]
                        dst_latency = self.echo_latency[dst]
                        delay = (fwd_delay + re_delay - src_latency - dst_latency)*(1000/2)
                        print('the delay between %s and %s dealy is %fms'%(src,dst,delay))
                        return max(delay, 0)
                except:
                        print("get delay error")
                        return float('inf')

        @set_ev_cls(event.EventSwitchEnter,[CONFIG_DISPATCHER,MAIN_DISPATCHER])
        def get_topology(self,ev):
                #store nodes info into the Graph
                switch_list = get_switch(self.topology_api_app,None) #----------need to get
info,by debug
                switches = [switch.dp.id for switch in switch_list]
                self.network.add_nodes_from(switches)
```

```
                #store links info into the Graph
                link_list = get_link(self.topology_api_app,None)
                #port_no, in_port ----------need to debug, get diffirent from both
                links = [(link.src.dpid,link.dst.dpid,{'attr_dict':{'port':link.dst.port_no}}) for link in
link_list] #add edge, need src,dst,weigtht
                self.network.add_edges_from(links)
                links = [(link.dst.dpid,link.src.dpid,{'attr_dict':{'port':link.dst.port_no}}) for link in
link_list]
                self.network.add_edges_from(links)
                self._send_echo_request()

        @set_ev_cls(ofp_event.EventOFPStateChange,[MAIN_DISPATCHER,
DEAD_DISPATCHER])
        def _state_change_handler(self, ev):
                datapath = ev.datapath
                if ev.state == MAIN_DISPATCHER:
                        if not datapath.id in self.datapaths:
                                self.logger.debug('Register datapath: %016x', datapath.id)
                                self.datapaths[datapath.id] = datapath
                elif ev.state == DEAD_DISPATCHER:
                        if datapath.id in self.datapaths:
                                self.logger.debug('Unregister datapath: %016x', datapath.id)
                                del self.datapaths[datapath.id]
```