SDN Experiment 2

计算机75 姚杰 2174311698

Introduction

In this experiment, a customed topology of network has been given to us, as known as ARPAnet, and thus we will tackle with the following questions by using RYU controller and mininet:

1. Finding a link route of fewest hops.
2. Finding a link route of least delay, and proving it by PING package.

Environment

Operating System: Linux version 4.15.0-20-generic
RYU Controller: 4.30 version
Mininet: 2.3.0d4 version

Content

This time we still use RYU as the remote controller. In this report I will not elucidate every process explicitly, for basic process such as constructing the controller has been depicted in the previous report, which you shall take as a reference.

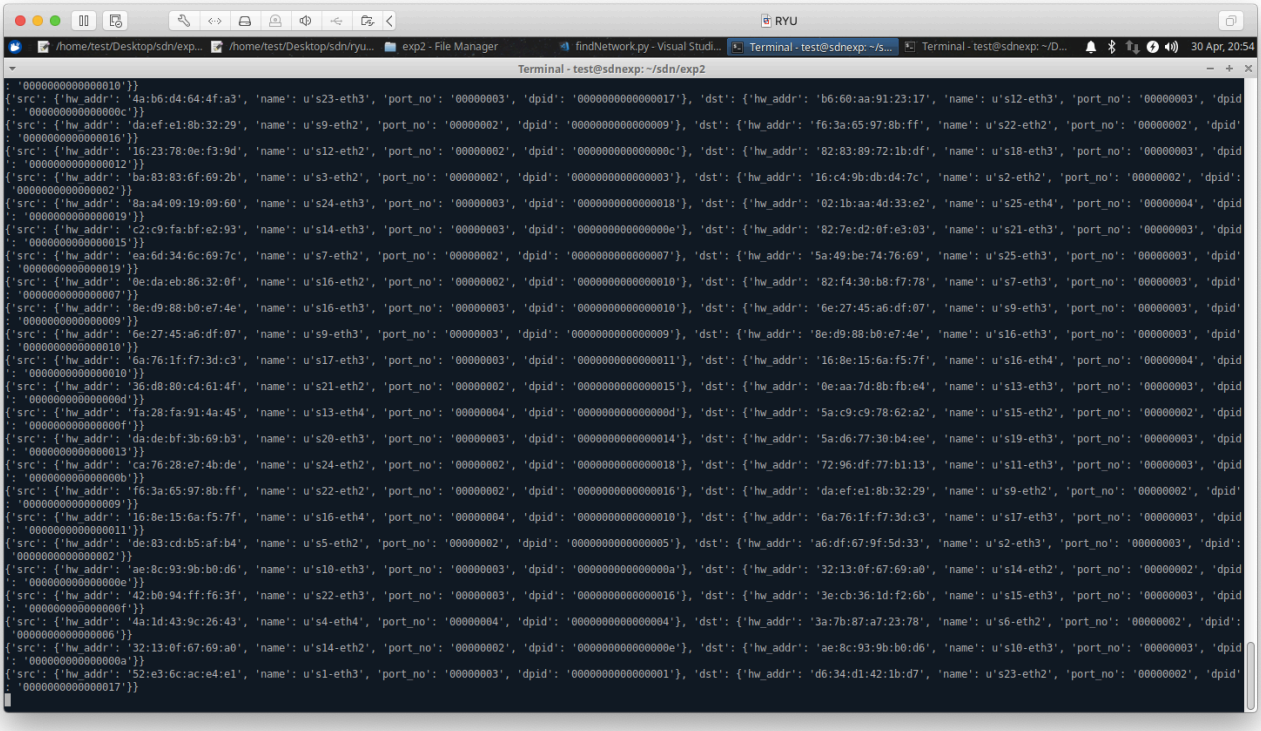
Finding a route of fewest hops

Basic understanding

By the hand of RYU's api, we are able to obtain the global topology structure of any given network. According to the api of Networkx, we need to obtain the information of all switches, hosts and links in the network, and thus create a DiGraph. Since the DiGraph has been created, we can calculate the shortest path easily becaust Networkx has provided us with fruitful api.

Building a DiGraph of the topology

However, before we have obtained the topology information of the network, we need to understand how it works. (See *Questions I have encoutered* part for more information) After we settle the problem, we can use the information to build a multigrah.



The result has shown above. We shall see that the controller is not able to obtain the information of the hosts. That happens because the hosts are silent, which means if they don't offer any packages they cannot be detected. Furthermore, the api: *get_all_host*, *get_all_switch* and *get_all_link* are using LLDP protocol, therefore we need to allow packages of this protocol to get through or it will cause the wrong the result.(See *Questions I have encoutered* part for more information).

After we have obtained the topology, we can create a DIGraph by using the nodes in the topology. Since the nodes of hosts are undetectable, we simply using switches and links instead. With regard to nodes of the hosts, we can add new rules in the package_in handler: when a new host which is not in the topology is sending packages to the switch, add it into the controller.

```
if src_mac not in self.graph:
    self.graph.add_node(src_mac)
    #dpid refers to the switch's dpid connected directly to the host
    self.graph.add_edge(src_mac, dpid)
    self.graph.add_edge(dpid, src_mac, port=in_port)
```

Flooding

Since the network topology has loops, simply using ARP protocol will not work. Instruction PDF has two ways of solving it. We use the latter method: using (dpid, mac, dstination mac) as key to record every port's value. So next when receiving the same key but different value, drop it(because it must be a package causing by loops)

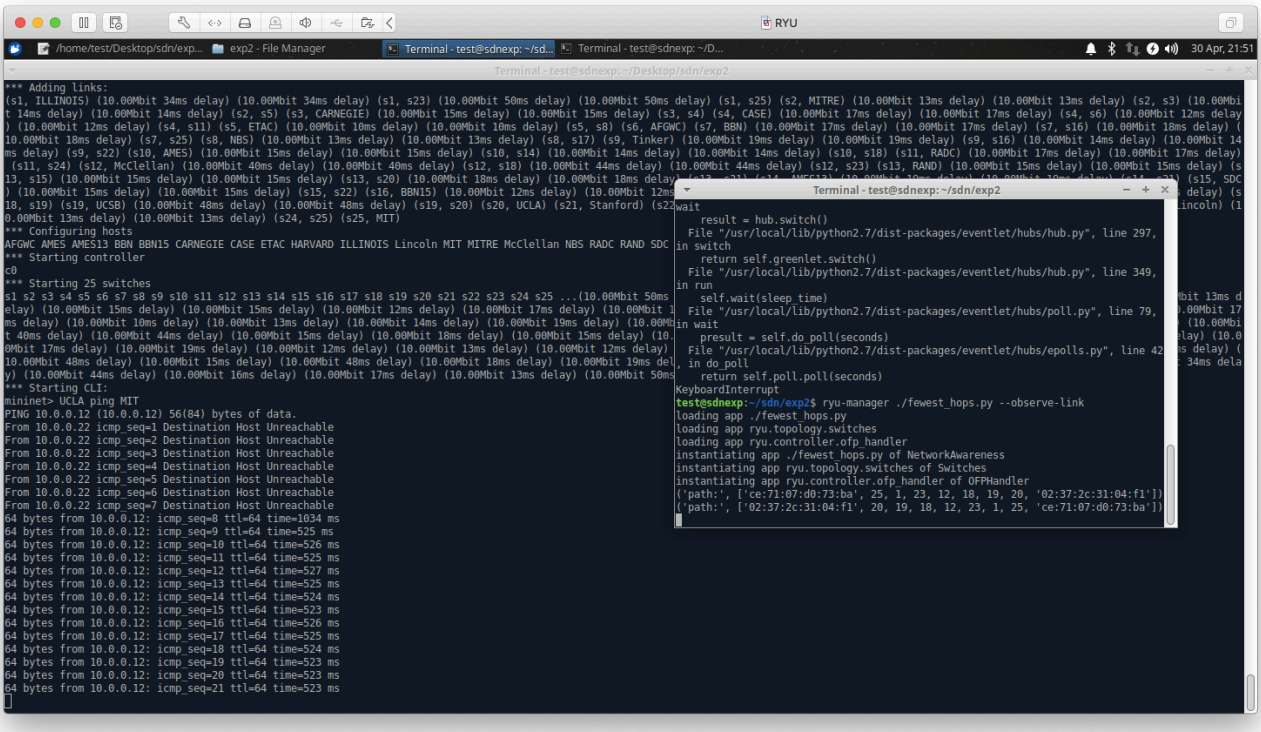
```
if ETHERNET in header_list:
    eth_dst = header_list[ETHERNET].dst
    eth_src = header_list[ETHERNET].src
    if eth_dst == ETHERNET_MULTICAST and ARP in header_list:
        arp_dst_ip = header_list[ARP].dst_ip
        if (datapath.id, eth_src, arp_dst_ip) in self.sw:  # Break the loop
            if self.sw[(datapath.id, eth_src, arp_dst_ip)] != in_port:
```

```
                out = datapath.ofproto_parser.OFPPacketOut(datapath=datapath,buffer_id=datapath.ofproto.OFP_NO_BUFFER,in_port=in_port,actions=[], data=None)
                datapath.send_msg(out)
                return
            else:
                self.sw[(datapath.id, eth_src, arp_dst_ip)] = in_port
```

Route of fewest hops

It is quite easy to find the path of the shortest hops by *shortest_path()*. However, we cannot call this function everytime the package is sent to a new switch, which means when the destination and source have decided, the path is certain. To achieve this, we can create a global two-dimensional dictonary to save this.

The final result shall be:



Finding a route of least delay

Basic understanding

In this section, there shall be two methods to calculate the delay between given points. First, we use

$$\_delay=(lldp\_delay\_s12 + lldp\_delay\_s21 - echo\_delay\_s1 - echo\_delay\_s2)/2\_$$

to calculate the delay on the link. Second, the delay has been provided on the network, and thus we regard the topology as a weighted graph. So the delay can be calculated by Networkx. Since the second method is quite similar with the previous experiment, we now skip uneccesary procedure.

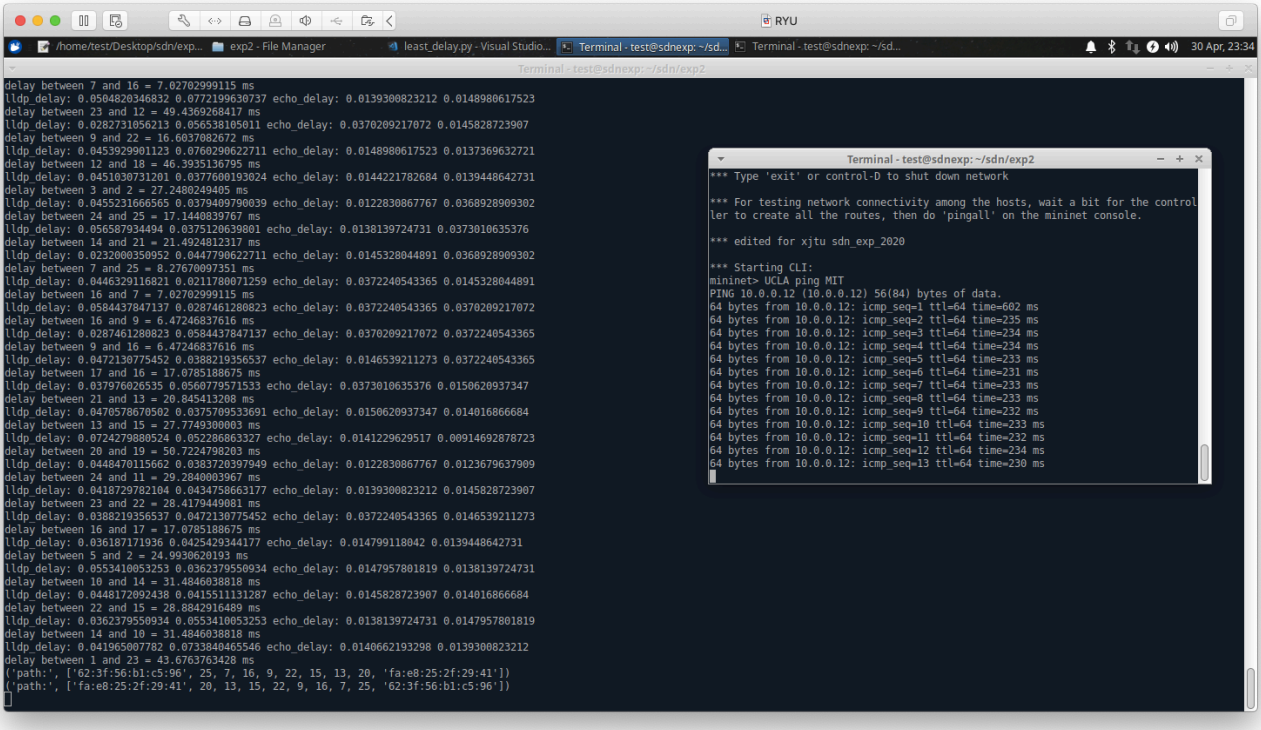
Weighted DiGraph

The DiGraph is a little different from the previous one. See the code below:

```python
switch_list = get_all_switch(self.topology_api_app, None)
switches = [switch.dp.id for switch in switch_list]
self.graph.add_nodes_from(switches)

# add edges
links = get_all_link(self.topology_api_app, None)
for link in links:
    self.graph.add_edge(link.src.dpid, link.dst.dpid, port=link.src.port_no)
    self.graph.add_edge(link.dst.dpid, link.src.dpid, port=link.dst.port_no)
```

Below is the result:



Questions I have encountered

1. Get_all_link api cannot work correctly

RYU controller uses LLDP protocol to detect the topology. Therefore, the switch shall let this kind of package pass and do nothing. So, you **MUST** add this to ignore any LLDP protocol packages.

```python
if eth.ethertype == ether_types.ETH_TYPE_LLDP:
        # ignore lldp packet
        return
```

2. Flow-tables adding failure

This is because before the loops have been eliminated, the tables have not been correctly configured. In order to solve this problem, we calculate the shortest path, and assign the out_port the correct port dpid instead using

self-learning. If the out_port has been correctly assigned, the flow-table will be distributed in a correct way.

```
if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
            #self.logger.info(out_port)
        else:
            if dst_mac in self.graph:
                if dst_mac not in self.paths[src_mac]:
                    try:
                        path = nx.shortest_path(self.graph, src_mac, dst_mac, weight=None)
                        self.paths[src_mac][dst_mac] = path
                        print('path:', path)
                    except:
                        return
                path = self.paths[src_mac][dst_mac]
                next_hop = path[path.index(dpid)+1]
                out_port = self.graph[dpid][next_hop]['port']
                self.mac_to_port[dpid][dst_mac] = out_port
```

Reference

walkthrough of Mininet
RYU docs
Li Cheng's Blog

*Attention: The codes on the 3rd website is **no longer working** in the mininet whose environment is descripted above. You shall modify it or refer to my codes.*

Source Code

Here are the codes of the customed controller.

Fewest_hops.py

```
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 ethernet
from ryu.lib.packet import arp
from ryu.lib import hub
from ryu.topology.api import get_all_host, get_all_link, get_all_switch
from ryu.lib.packet import ether_types
import networkx as nx

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

```
17
18 class NetworkAwareness(app_manager.RyuApp):
19     OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
20
21     def __init__(self, *args, **kwargs):
22         super(NetworkAwareness, self).__init__(*args, **kwargs)
23         self.dpid_mac_port = {}
24         self.topo_thread = hub.spawn(self._get_topology)
25         self.mac_to_port={}
26         self.sw={}
27         self.graph = nx.DiGraph()
28
29     def add_flow(self, datapath, priority, match, actions):
30         dp = datapath
31         ofp = dp.ofproto
32         parser = dp.ofproto_parser
33         inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
34         mod = parser.OFPFlowMod(datapath=dp, priority=priority, match=match,
instructions=inst)
35         dp.send_msg(mod)
36
37     @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
38     def switch_features_handler(self, ev):
39         msg = ev.msg
40         dp = msg.datapath
41         ofp = dp.ofproto
42         parser = dp.ofproto_parser
43         match = parser.OFPMatch()
44         actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
45         self.add_flow(dp, 0, match, actions)
46
47     @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
48     def _packet_in_handler(self, ev):
49         msg = ev.msg
50         datapath = msg.datapath
51         ofproto = datapath.ofproto
52         parser = datapath.ofproto_parser
53
54         # get Datapath ID to identify OpenFlow switches.
55         dpid = datapath.id
56         self.mac_to_port.setdefault(dpid, {})
57
58         # analyse the received packets using the packet library.
59         pkt = packet.Packet(msg.data)
60         eth = pkt.get_protocols(ethernet.ethernet)[0]
61
62         if eth.ethertype == ether_types.ETH_TYPE_LLDP:
63             # ignore lldp packet
64             return
65         if eth.ethertype == ether_types.ETH_TYPE_IPV6:
66             return
67
68         eth_pkt = pkt.get_protocol(ethernet.ethernet)
69         dst = eth_pkt.dst
70         src = eth_pkt.src
71         # add new host into the grpah
```

```
72          if src_mac not in self.graph:
73              self.graph.add_node(src_mac)
74              self.graph.add_edge(src_mac, dpid)
75              self.graph.add_edge(dpid, src_mac, port=in_port)
76
77          # get the received port number from packet_in message.
78          in_port = msg.match['in_port']
79          header_list = dict((p.protocol_name, p)for p in pkt.protocols if type(p) != str)
80          # self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)
81
82          # if the destination mac address is already learned,
83          # decide which port to output the packet, otherwise FLOOD.
84          if ETHERNET in header_list:
85              eth_dst = header_list[ETHERNET].dst
86              eth_src = header_list[ETHERNET].src
87          if eth_dst == ETHERNET_MULTICAST and ARP in header_list:
88              arp_dst_ip = header_list[ARP].dst_ip
89              if (datapath.id, eth_src, arp_dst_ip) in self.sw:  # Break the loop
90                  if self.sw[(datapath.id, eth_src, arp_dst_ip)] != in_port:
91                      out =
    datapath.ofproto_parser.OFPPacketOut(datapath=datapath,buffer_id=datapath.ofproto.OFP_NO_BUF
    FER,in_port=in_port,actions=[], data=None)
92                      datapath.send_msg(out)
93                      return
94              else:
95                  self.sw[(datapath.id, eth_src, arp_dst_ip)] = in_port
96          # learn a mac address to avoid FLOOD next time.
97          #self.mac_to_port[dpid][src] = in_port
98
99          if dst in self.mac_to_port[dpid]:
100             out_port = self.mac_to_port[dpid][dst]
101             #self.logger.info(out_port)
102         else:
103             if dst_mac in self.graph:
104                 if dst_mac not in self.paths[src_mac]:
105                     try:
106                         path = nx.shortest_path(self.graph, src_mac, dst_mac, weight=None)
107                         self.paths[src_mac][dst_mac] = path
108                         print('path:', path)
109                     except:
110                         return
111                 path = self.paths[src_mac][dst_mac]
112                 next_hop = path[path.index(dpid)+1]
113                 out_port = self.graph[dpid][next_hop]['port']
114                 self.mac_to_port[dpid][dst_mac] = out_port
115
116         # construct action list.
117         actions = [parser.OFPActionOutput(out_port)]
118
119         # install a flow to avoid packet_in next time.
120
121         if out_port != ofproto.OFPP_FLOOD:
122             match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
123             self.add_flow(datapath, 1, match, actions)
124
125         # construct packet_out message and send it.
```

```
        out = parser.OFPPacketOut(datapath=datapath,buffer_id=ofproto.OFP_NO_BUFFER,in_port=in_port, actions=actions,data=msg.data)
        datapath.send_msg(out)

    def _get_topology(self):
        hub.sleep(10)
        self.logger.info('\n\n\n')
        #hosts = get_all_host(self)
        switches = get_all_switch(self)
        links = get_all_link(self)

        self.logger.info('switches:')
        for switch in switches:
            self.logger.info(switch.to_dict())
            #self.logger.info(switch)
            self.graph.add_node(switch.dp.id)

        self.logger.info('links:')
        for link in links:
            self.logger.info(link.to_dict())
            #self.logger.info('src='+str(link.src.dpid)+', dst='+str(link.dst.dpid))
            self.graph.add_edge(link.src.dpid, link.dst.dpid, port=link.src.port_no)
            self.graph.add_edge(link.dst.dpid, link.src.dpid, port=link.dst.port_no)

```

Least_delay.py

```
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 ethernet
from ryu.lib.packet import arp
from ryu.lib import hub
from ryu.topology.api import get_all_host, get_all_link, get_all_switch
from ryu.lib.packet import ether_types
import networkx as nx
import time

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

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.topo_thread = hub.spawn(self._get_topology)
        self.mac_to_port={}
        self.sw={}
```

```
28          self.graph = nx.DiGraph()
29          self.paths = {}
30          self.lldp_delay = {}
31          self.echo_delay = {}
32
33      def add_flow(self, datapath, priority, match, actions):
34          dp = datapath
35          ofp = dp.ofproto
36          parser = dp.ofproto_parser
37          inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
38          mod = parser.OFPFlowMod(datapath=dp, priority=priority, match=match,
    instructions=inst)
39          dp.send_msg(mod)
40
41      @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
42      def switch_features_handler(self, ev):
43          msg = ev.msg
44          dp = msg.datapath
45          ofp = dp.ofproto
46          parser = dp.ofproto_parser
47          match = parser.OFPMatch()
48          actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
49          self.add_flow(dp, 0, match, actions)
50
51      @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
52      def _packet_in_handler(self, ev):
53          msg = ev.msg
54          datapath = msg.datapath
55          ofproto = datapath.ofproto
56          parser = datapath.ofproto_parser
57
58          # get Datapath ID to identify OpenFlow switches.
59          dpid = datapath.id
60          self.mac_to_port.setdefault(dpid, {})
61
62          # analyse the received packets using the packet library.
63          pkt = packet.Packet(msg.data)
64          eth = pkt.get_protocols(ethernet.ethernet)[0]
65
66          if eth.ethertype == ether_types.ETH_TYPE_LLDP:
67              # ignore lldp packet
68              return
69          if eth.ethertype == ether_types.ETH_TYPE_IPV6:
70              return
71
72          eth_pkt = pkt.get_protocol(ethernet.ethernet)
73          dst = eth_pkt.dst
74          src = eth_pkt.src
75          # add new host into the grpah
76          if src_mac not in self.graph:
77              self.graph.add_node(src_mac)
78              self.graph.add_edge(src_mac, dpid, weight=0)
79              self.graph.add_edge(dpid, src_mac, weight=0,port=in_port)
80
81          # get the received port number from packet_in message.
82          in_port = msg.match['in_port']
```

```
83          header_list = dict((p.protocol_name, p)for p in pkt.protocols if type(p) != str)
84          # self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)
85
86          # if the destination mac address is already learned,
87          # decide which port to output the packet, otherwise FLOOD.
88          if ETHERNET in header_list:
89              eth_dst = header_list[ETHERNET].dst
90              eth_src = header_list[ETHERNET].src
91          if eth_dst == ETHERNET_MULTICAST and ARP in header_list:
92              arp_dst_ip = header_list[ARP].dst_ip
93              if (datapath.id, eth_src, arp_dst_ip) in self.sw:  # Break the loop
94                  if self.sw[(datapath.id, eth_src, arp_dst_ip)] != in_port:
95                      out =
datapath.ofproto_parser.OFPPacketOut(datapath=datapath,buffer_id=datapath.ofproto.OFP_NO_BUF
FER,in_port=in_port,actions=[], data=None)
96                      datapath.send_msg(out)
97                      return
98              else:
99                  self.sw[(datapath.id, eth_src, arp_dst_ip)] = in_port
100         # learn a mac address to avoid FLOOD next time.
101         #self.mac_to_port[dpid][src] = in_port
102
103         if dst in self.mac_to_port[dpid]:
104             out_port = self.mac_to_port[dpid][dst]
105             #self.logger.info(out_port)
106         else:
107             if dst_mac in self.graph:
108                 if dst_mac not in self.paths[src_mac]:
109                     try:
110                         path = nx.shortest_path(self.graph, src_mac, dst_mac, weight=None)
111                         self.paths[src_mac][dst_mac] = path
112                         print('path:', path)
113                     except:
114                         return
115                 path = self.paths[src_mac][dst_mac]
116                 next_hop = path[path.index(dpid)+1]
117                 out_port = self.graph[dpid][next_hop]['port']
118                 self.mac_to_port[dpid][dst_mac] = out_port
119
120         # construct action list.
121         actions = [parser.OFPActionOutput(out_port)]
122
123         # install a flow to avoid packet_in next time.
124
125         if out_port != ofproto.OFPP_FLOOD:
126             match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
127             self.add_flow(datapath, 1, match, actions)
128
129         # construct packet_out message and send it.
130         out =
parser.OFPPacketOut(datapath=datapath,buffer_id=ofproto.OFP_NO_BUFFER,in_port=in_port,
actions=actions,data=msg.data)
131         datapath.send_msg(out)
132
133     def _get_topology(self):
134         hub.sleep(10)
```

```
135         self.logger.info('\n\n\n')
136         #hosts = get_all_host(self)
137         switches = get_all_switch(self)
138         links = get_all_link(self)
139 
140         self.logger.info('switches:')
141         for switch in switches:
142             self.logger.info(switch.to_dict())
143             #self.logger.info(switch)
144             self.graph.add_node(switch.dp.id)
145 
146         self.logger.info('links:')
147         for link in links:
148             try:
149                 lldp_delay1 = self.lldp_delay[(link.src.dpid, link.dst.dpid)]
150                 lldp_delay2 = self.lldp_delay[(link.dst.dpid, link.src.dpid)]
151                 echo_delay1 = self.echo_delay[link.src.dpid]
152                 echo_delay2 = self.echo_delay[link.dst.dpid]
153                 delay = (lldp_delay1 + lldp_delay2 - echo_delay1 - echo_delay2) / 2
154                 w = max(delay, 0)
155             except:
156                 w = float('inf')
157             self.logger.info('lldp_delay: %s %s echo_delay: %s %s', lldp_delay1,
    lldp_delay2, echo_delay1, echo_delay2)
158             self.logger.info('delay between %s and %s = %s ms', link.src.dpid,
    link.dst.dpid, delay * 1000)
159             self.graph.add_edge(link.src.dpid, link.dst.dpid, weight=w,
    port=link.src.port_no)
160             self.graph.add_edge(link.dst.dpid, link.src.dpid, weight=w,
    port=link.dst.port_no)
```