

SDN Experiment 2

计算机75 姚杰 2174311698

Introduction

In this experiment, a customized 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 because 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 encountered* part for more information) After we settle the problem, we can use the information to build a multigraph.

```
Terminal - test@sdnexp: ~/sdn/exp2
{
  'src': {'hw_addr': '4a:b6:d4:64:f4:a3', 'name': 'u's23-eth3', 'port_no': '00000003', 'dpid': '0000000000000017', 'dst': {'hw_addr': 'b6:60:aa:91:23:17', 'name': 'u's12-eth3', 'port_no': '00000003', 'dpid': '000000000000000c'}},
  'src': {'hw_addr': 'da:ef:e1:8b:32:29', 'name': 'u's9-eth2', 'port_no': '00000002', 'dpid': '0000000000000009', 'dst': {'hw_addr': 'f6:3a:65:97:8b:ff', 'name': 'u's22-eth2', 'port_no': '00000002', 'dpid': '0000000000000016'}},
  'src': {'hw_addr': '16:23:78:0e:f3:9d', 'name': 'u's12-eth2', 'port_no': '00000002', 'dpid': '000000000000000c', 'dst': {'hw_addr': '82:83:89:72:1b:df', 'name': 'u's18-eth3', 'port_no': '00000003', 'dpid': '0000000000000012'}},
  'src': {'hw_addr': 'ba:83:83:6f:69:2b', 'name': 'u's3-eth2', 'port_no': '00000002', 'dpid': '0000000000000003', 'dst': {'hw_addr': '16:c4:9b:db:d4:7c', 'name': 'u's2-eth2', 'port_no': '00000002', 'dpid': '0000000000000002'}},
  'src': {'hw_addr': '8a:a4:09:19:09:60', 'name': 'u's24-eth3', 'port_no': '00000003', 'dpid': '0000000000000018', 'dst': {'hw_addr': '02:1b:aa:4d:33:e2', 'name': 'u's25-eth4', 'port_no': '00000004', 'dpid': '0000000000000019'}},
  'src': {'hw_addr': 'c2:c9:f4:bf:e2:93', 'name': 'u's14-eth3', 'port_no': '00000003', 'dpid': '000000000000000e', 'dst': {'hw_addr': '82:7e:d2:0f:e3:03', 'name': 'u's21-eth3', 'port_no': '00000003', 'dpid': '0000000000000015'}},
  'src': {'hw_addr': 'ea:6d:34:6c:69:7c', 'name': 'u's7-eth2', 'port_no': '00000002', 'dpid': '0000000000000007', 'dst': {'hw_addr': '5a:49:be:74:76:69', 'name': 'u's25-eth3', 'port_no': '00000003', 'dpid': '0000000000000019'}},
  'src': {'hw_addr': '0e:da:eb:86:32:0f', 'name': 'u's16-eth2', 'port_no': '00000002', 'dpid': '0000000000000010', 'dst': {'hw_addr': '82:f4:30:b8:f7:78', 'name': 'u's7-eth3', 'port_no': '00000003', 'dpid': '0000000000000007'}},
  'src': {'hw_addr': '0e:d9:88:b0:e7:4e', 'name': 'u's16-eth3', 'port_no': '00000003', 'dpid': '0000000000000010', 'dst': {'hw_addr': '6e:27:45:a6:df:07', 'name': 'u's9-eth3', 'port_no': '00000003', 'dpid': '0000000000000002'}},
  'src': {'hw_addr': '6e:27:45:a6:df:07', 'name': 'u's9-eth3', 'port_no': '00000003', 'dpid': '0000000000000009', 'dst': {'hw_addr': '8e:d9:88:b0:e7:4e', 'name': 'u's16-eth3', 'port_no': '00000003', 'dpid': '0000000000000010'}},
  'src': {'hw_addr': '6a:76:1f:f7:3d:c3', 'name': 'u's17-eth3', 'port_no': '00000003', 'dpid': '0000000000000011', 'dst': {'hw_addr': '16:8e:15:6a:f5:7f', 'name': 'u's16-eth4', 'port_no': '00000004', 'dpid': '0000000000000010'}},
  'src': {'hw_addr': '32:08:00:c4:61:4f', 'name': 'u's21-eth2', 'port_no': '00000002', 'dpid': '0000000000000015', 'dst': {'hw_addr': '0e:aa:7d:0b:fb:e4', 'name': 'u's13-eth3', 'port_no': '00000003', 'dpid': '0000000000000004'}},
  'src': {'hw_addr': 'fa:28:fa:91:4a:45', 'name': 'u's13-eth4', 'port_no': '00000004', 'dpid': '000000000000000d', 'dst': {'hw_addr': '5a:c9:c9:78:62:a2', 'name': 'u's15-eth2', 'port_no': '00000002', 'dpid': '000000000000000f'}},
  'src': {'hw_addr': 'da:de:bf:3b:69:b3', 'name': 'u's20-eth3', 'port_no': '00000003', 'dpid': '0000000000000014', 'dst': {'hw_addr': '5a:d6:77:30:b4:ee', 'name': 'u's19-eth3', 'port_no': '00000003', 'dpid': '0000000000000010'}},
  'src': {'hw_addr': 'ca:76:28:e7:4b:de', 'name': 'u's24-eth2', 'port_no': '00000002', 'dpid': '0000000000000018', 'dst': {'hw_addr': '72:96:df:77:b1:13', 'name': 'u's11-eth3', 'port_no': '00000003', 'dpid': '000000000000000b'}},
  'src': {'hw_addr': 'f6:3a:65:97:8b:ff', 'name': 'u's22-eth2', 'port_no': '00000002', 'dpid': '0000000000000016', 'dst': {'hw_addr': 'da:ef:e1:8b:32:29', 'name': 'u's9-eth2', 'port_no': '00000002', 'dpid': '0000000000000009'}},
  'src': {'hw_addr': '16:8e:15:6a:f5:7f', 'name': 'u's16-eth4', 'port_no': '00000004', 'dpid': '0000000000000010', 'dst': {'hw_addr': '6a:76:1f:f7:3d:c3', 'name': 'u's17-eth3', 'port_no': '00000003', 'dpid': '0000000000000011'}},
  'src': {'hw_addr': 'de:83:cd:b5:af:b4', 'name': 'u's5-eth2', 'port_no': '00000002', 'dpid': '0000000000000005', 'dst': {'hw_addr': 'a6:df:67:9f:5d:33', 'name': 'u's2-eth3', 'port_no': '00000003', 'dpid': '0000000000000002'}},
  'src': {'hw_addr': 'ae:8c:93:9b:b0:d6', 'name': 'u's10-eth3', 'port_no': '00000003', 'dpid': '000000000000000a', 'dst': {'hw_addr': '32:13:0f:67:69:a0', 'name': 'u's14-eth2', 'port_no': '00000002', 'dpid': '000000000000000e'}},
  'src': {'hw_addr': '42:b0:94:ff:f6:3f', 'name': 'u's22-eth3', 'port_no': '00000003', 'dpid': '0000000000000016', 'dst': {'hw_addr': '3e:cb:3e:1d:f2:6b', 'name': 'u's15-eth3', 'port_no': '00000003', 'dpid': '000000000000000f'}},
  'src': {'hw_addr': '4a:1d:43:9c:26:43', 'name': 'u's4-eth4', 'port_no': '00000004', 'dpid': '0000000000000004', 'dst': {'hw_addr': '3a:7b:87:a7:23:78', 'name': 'u's6-eth2', 'port_no': '00000002', 'dpid': '0000000000000006'}},
  'src': {'hw_addr': '32:13:0f:67:69:a0', 'name': 'u's14-eth2', 'port_no': '00000002', 'dpid': '000000000000000e', 'dst': {'hw_addr': 'ae:8c:93:9b:b0:d6', 'name': 'u's10-eth3', 'port_no': '00000003', 'dpid': '000000000000000a'}},
  'src': {'hw_addr': '52:e3:6c:ac:e4:el', 'name': 'u's1-eth3', 'port_no': '00000003', 'dpid': '0000000000000001', 'dst': {'hw_addr': 'd6:34:d1:42:1b:d7', 'name': 'u's23-eth2', 'port_no': '00000002', 'dpid': '0000000000000017'}}
}
```

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 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.

```
1 if src_mac not in self.graph:
2     self.graph.add_node(src_mac)
3     #dpid refers to the switch's dpid connected directly to the host
4     self.graph.add_edge(src_mac, dpid)
5     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)

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

```

8         out =
datapath.ofproto_parser.OFPacketOut(datapath=datapath,buffer_id=datapath.ofproto.OFP_NO_BUF
FER,in_port=in_port,actions=[], data=None)
9         datapath.send_msg(out)
10        return
11    else:
12        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 dictionary to save this.

The final result shall be:

```

*** Adding links:
(s1, ILLINOIS) (10.00Mbit 34ms delay) (10.00Mbit 34ms delay) (s1, s23) (10.00Mbit 50ms delay) (10.00Mbit 50ms delay) (s1, s25) (s2, MITRE) (10.00Mbit 13ms delay) (10.00Mbit 13ms delay) (s2, s3) (10.00Mbit 14ms delay) (10.00Mbit 14ms delay) (s2, s5) (s3, CARNEGIE) (10.00Mbit 15ms delay) (10.00Mbit 15ms delay) (s3, s4) (s4, CASE) (10.00Mbit 17ms delay) (10.00Mbit 17ms delay) (s4, s6) (10.00Mbit 12ms delay) (10.00Mbit 12ms delay) (s4, s11) (s5, ETAC) (10.00Mbit 10ms delay) (10.00Mbit 10ms delay) (s5, s8) (s6, AFGWC) (s7, BBN) (10.00Mbit 17ms delay) (10.00Mbit 17ms delay) (s7, s16) (10.00Mbit 18ms delay) (10.00Mbit 18ms delay) (s7, s25) (s8, NBS) (10.00Mbit 13ms delay) (10.00Mbit 13ms delay) (s8, s17) (s9, Tinker) (10.00Mbit 19ms delay) (10.00Mbit 19ms delay) (s9, s16) (10.00Mbit 14ms delay) (10.00Mbit 14ms delay) (s9, s22) (s10, AMES) (10.00Mbit 15ms delay) (10.00Mbit 15ms delay) (s10, s14) (10.00Mbit 14ms delay) (10.00Mbit 14ms delay) (s10, s18) (s11, RADC) (10.00Mbit 17ms delay) (10.00Mbit 17ms delay) (s11, s24) (s12, McClellan) (10.00Mbit 40ms delay) (10.00Mbit 40ms delay) (s12, s18) (10.00Mbit 44ms delay) (10.00Mbit 44ms delay) (s12, s23) (s13, RAND) (10.00Mbit 15ms delay) (10.00Mbit 15ms delay) (s13, s15) (10.00Mbit 15ms delay) (10.00Mbit 15ms delay) (s13, s20) (10.00Mbit 18ms delay) (10.00Mbit 18ms delay) (s13, s22) (10.00Mbit 15ms delay) (10.00Mbit 15ms delay) (s15, s22) (s16, BBN15) (10.00Mbit 12ms delay) (10.00Mbit 12ms delay) (s16, s19) (s19, UCSB) (10.00Mbit 48ms delay) (10.00Mbit 48ms delay) (s19, s20) (s20, UCLA) (s21, Stanford) (s22, SDC) (10.00Mbit 13ms delay) (10.00Mbit 13ms delay) (s24, s25) (s25, MIT)
*** Configuring hosts
AFGWC AMES AMES13 BBN BBN15 CARNEGIE CASE ETAC HARVARD ILLINOIS Lincoln MIT MITRE McClellan NBS RADC RAND SDC
*** Starting controller
c0
*** Starting 25 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19 s20 s21 s22 s23 s24 s25 ... (10.00Mbit 50ms delay) (10.00Mbit 15ms delay) (10.00Mbit 15ms delay) (10.00Mbit 12ms delay) (10.00Mbit 17ms delay) (10.00Mbit 14ms delay) (10.00Mbit 10ms delay) (10.00Mbit 13ms delay) (10.00Mbit 14ms delay) (10.00Mbit 19ms delay) (10.00Mbit 40ms delay) (10.00Mbit 44ms delay) (10.00Mbit 15ms delay) (10.00Mbit 18ms delay) (10.00Mbit 15ms delay) (10.00Mbit 17ms delay) (10.00Mbit 19ms delay) (10.00Mbit 12ms delay) (10.00Mbit 13ms delay) (10.00Mbit 12ms delay) (10.00Mbit 48ms delay) (10.00Mbit 15ms delay) (10.00Mbit 48ms delay) (10.00Mbit 18ms delay) (10.00Mbit 19ms delay) (10.00Mbit 44ms delay) (10.00Mbit 16ms delay) (10.00Mbit 17ms delay) (10.00Mbit 13ms delay) (10.00Mbit 50ms delay)
*** Starting CLI:
mininet> UCLA ping MIT
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data:
From 10.0.0.12: icmp_seq=1 Destination Host Unreachable
From 10.0.0.12: icmp_seq=2 Destination Host Unreachable
From 10.0.0.12: icmp_seq=3 Destination Host Unreachable
From 10.0.0.12: icmp_seq=4 Destination Host Unreachable
From 10.0.0.12: icmp_seq=5 Destination Host Unreachable
From 10.0.0.12: icmp_seq=6 Destination Host Unreachable
From 10.0.0.12: icmp_seq=7 Destination Host Unreachable
64 bytes from 10.0.0.12: icmp_seq=8 ttl=64 time=1034 ms
64 bytes from 10.0.0.12: icmp_seq=9 ttl=64 time=525 ms
64 bytes from 10.0.0.12: icmp_seq=10 ttl=64 time=526 ms
64 bytes from 10.0.0.12: icmp_seq=11 ttl=64 time=525 ms
64 bytes from 10.0.0.12: icmp_seq=12 ttl=64 time=527 ms
64 bytes from 10.0.0.12: icmp_seq=13 ttl=64 time=525 ms
64 bytes from 10.0.0.12: icmp_seq=14 ttl=64 time=524 ms
64 bytes from 10.0.0.12: icmp_seq=15 ttl=64 time=523 ms
64 bytes from 10.0.0.12: icmp_seq=16 ttl=64 time=526 ms
64 bytes from 10.0.0.12: icmp_seq=17 ttl=64 time=525 ms
64 bytes from 10.0.0.12: icmp_seq=18 ttl=64 time=524 ms
64 bytes from 10.0.0.12: icmp_seq=19 ttl=64 time=523 ms
64 bytes from 10.0.0.12: icmp_seq=20 ttl=64 time=523 ms
64 bytes from 10.0.0.12: icmp_seq=21 ttl=64 time=523 ms

```

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 unnecessary procedure.

Weighted DiGraph

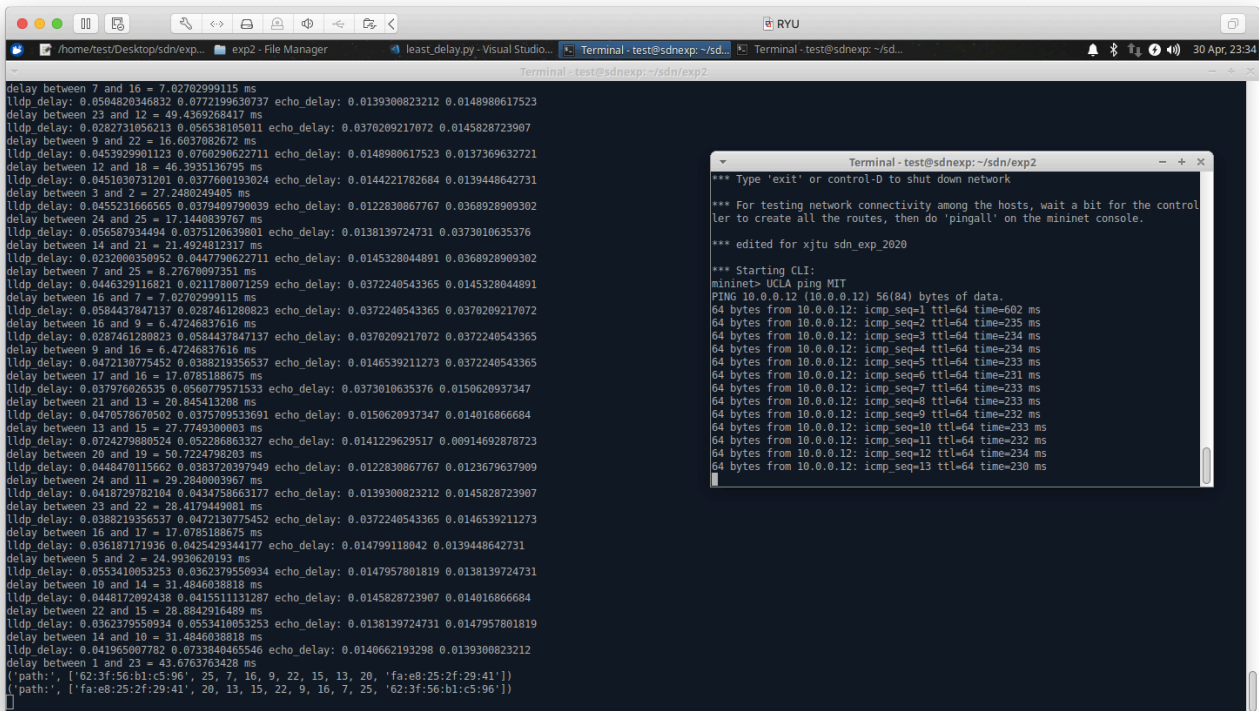
The DiGraph is a little different from the previous one. See the code below:

```

1 switch_list = get_all_switch(self.topology_api_app, None)
2 switches = [switch.dp.id for switch in switch_list]
3 self.graph.add_nodes_from(switches)
4
5 # add edges
6 links = get_all_link(self.topology_api_app, None)
7 for link in links:
8     self.graph.add_edge(link.src.dpid, link.dst.dpid, port=link.src.port_no)
9     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.

```

1 if eth.ethertype == ether_types.ETH_TYPE_LLDP:
2     # ignore lldp packet
3     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.

```
1 if dst in self.mac_to_port[dpid]:
2     out_port = self.mac_to_port[dpid][dst]
3     #self.logger.info(out_port)
4 else:
5     if dst_mac in self.graph:
6         if dst_mac not in self.paths[src_mac]:
7             try:
8                 path = nx.shortest_path(self.graph, src_mac, dst_mac, weight=None)
9                 self.paths[src_mac][dst_mac] = path
10                print('path:', path)
11            except:
12                return
13        path = self.paths[src_mac][dst_mac]
14        next_hop = path[path.index(dpid)+1]
15        out_port = self.graph[dpid][next_hop]['port']
16        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 described above. You shall modify it or refer to my codes.

Source Code

Here are the codes of the customized controller.

Fewest_hops.py

```
1 from ryu.base import app_manager
2 from ryu.ofproto import ofproto_v1_3
3 from ryu.controller.handler import set_ev_cls
4 from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
5 from ryu.controller import ofp_event
6 from ryu.lib.packet import packet
7 from ryu.lib.packet import ethernet
8 from ryu.lib.packet import arp
9 from ryu.lib import hub
10 from ryu.topology.api import get_all_host, get_all_link, get_all_switch
11 from ryu.lib.packet import ether_types
12 import networkx as nx
13
14 ETHERNET = ethernet.ethernet.__name__
15 ETHERNET_MULTICAST = "ff:ff:ff:ff:ff:ff"
16 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.OFPacketOut(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.

```



```

126         out =
parser.OFPPacketOut(datapath=datapath,buffer_id=ofproto.OFP_NO_BUFFER,in_port=in_port,
actions=actions,data=msg.data)
127         datapath.send_msg(out)
128
129     def _get_topology(self):
130         hub.sleep(10)
131         self.logger.info('\n\n\n')
132         #hosts = get_all_host(self)
133         switches = get_all_switch(self)
134         links = get_all_link(self)
135
136         self.logger.info('switches:')
137         for switch in switches:
138             self.logger.info(switch.to_dict())
139             #self.logger.info(switch)
140             self.graph.add_node(switch.dp.id)
141
142         self.logger.info('links:')
143         for link in links:
144             self.logger.info(link.to_dict())
145             #self.logger.info('src='+str(link.src.dpid)+', dst='+str(link.dst.dpid))
146             self.graph.add_edge(link.src.dpid, link.dst.dpid, port=link.src.port_no)
147             self.graph.add_edge(link.dst.dpid, link.src.dpid, port=link.dst.port_no)
148

```

Least_delay.py

```

1 from ryu.base import app_manager
2 from ryu.ofproto import ofproto_v1_3
3 from ryu.controller.handler import set_ev_cls
4 from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
5 from ryu.controller import ofp_event
6 from ryu.lib.packet import packet
7 from ryu.lib.packet import ethernet
8 from ryu.lib.packet import arp
9 from ryu.lib import hub
10 from ryu.topology.api import get_all_host, get_all_link, get_all_switch
11 from ryu.lib.packet import ether_types
12 import networkx as nx
13 import time
14
15 ETHERNET = ethernet.ethernet.__name__
16 ETHERNET_MULTICAST = "ff:ff:ff:ff:ff:ff"
17 ARP = arp.arp.__name__
18
19 class NetworkAwareness(app_manager.RyuApp):
20     OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
21
22     def __init__(self, *args, **kwargs):
23         super(NetworkAwareness, self).__init__(*args, **kwargs)
24         self.dpid_mac_port = {}
25         self.topo_thread = hub.spawn(self._get_topology)
26         self.mac_to_port={}
27         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 graph
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_BUFFER,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.OFPAActionOutput(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')
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)

```