Lab 6: Load balancer

Dharshini Tharmarajan

March 2020

1 Introduction

In this lab I learned the functionality of a load balancer, and tried to implement a load balancer in python with the POX API. The load balancer used a Round-robin algorithm, to redirect requests towards a service, to three backend servers. I started by making a summary of the theory part, edited the code, and then did Tasks and Questions. Through the lab Iselin Eriksen Eng and I worked togehter, because we have been in the same lab group together. We wrote our own reports for this lab, but since we worked together, there might be a few similarities in the reports.

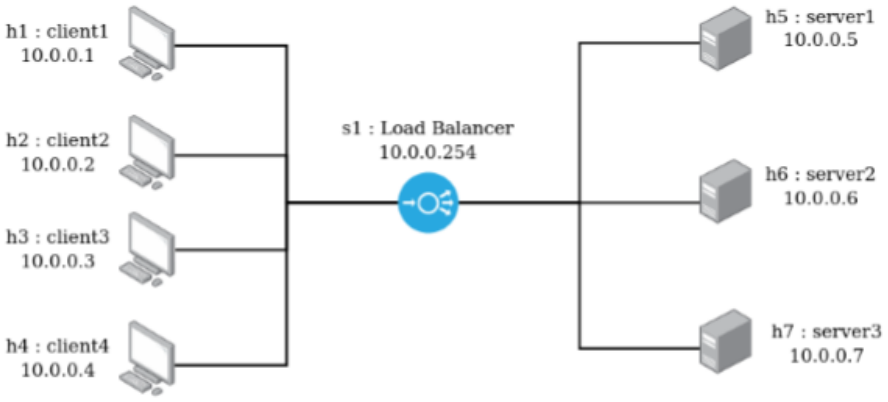


Figure 1: Topology for lab 6.

Contents

2 Theory

2.1 Load Balancer

- Modern applications are expected to handle extreme loads, which can handle a lot of requests per second.
- For a web server which runs on a local machine, a lot of power is needed to serve all these request without the response time approaching infinity.
- Load Balancer is a component which helps to spread the traffic across a cluster of servers, and is used to improve the responsiveness and availability of applications or websites.
- In web applications, the load balancer is often placed between the web servers and the Internet, and distribute requests between multiple identical web servers.
- Figure 2 shows three users who wants to access nrk.no, where their machines query a DNS server to get the site's IP address.
- In Figure 4, the 3 users send requests to the IP address, which corresponds to nrk.no. The requests gets distributed among the backend servers, by the load balancer.
- Load balancer keeps track of the status of all resources, when it distributes requests.
- If a server is unavailable to manage new requests or is not responding, the load balancer stops sending traffic to that server.

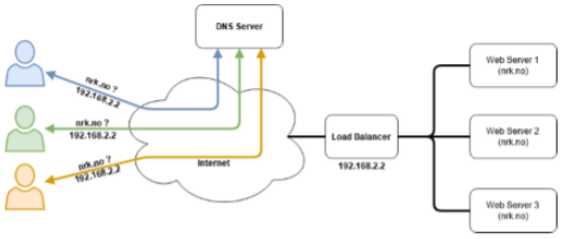


Figure 2: Machines query a DNS server to get nrk.no's IP address.

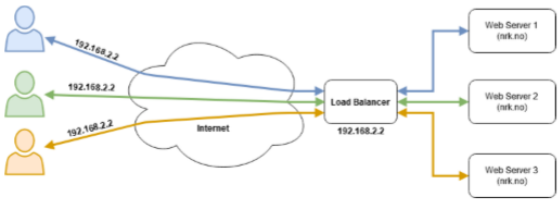


Figure 3: Requests distrubuted by the load balancer.

2.2 Round- robin scheduling

- The algorithm aims to distribute equal portions of a particular unit in a circular order, without a priority.
- Round robin scheduling cycles through a list which contains servers, and sends new request to the next server.
- When it reaches the end of the list, it starts over again from the beginning (1).

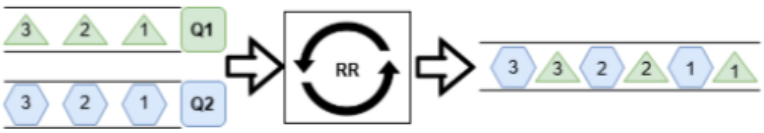


Figure 4: The algorithm distributes access to the channel fairly among each object in Q1 and Q2.

3 Walkthrough

3.1 Topology

- In this lab we have 7 hosts and 1 switch, as shown in figure 1.
  - 4 hosts as clients, 3 hosts as servers.
- There is a a BASH script provided for this lab named **run.sh**, which runs in terminal.
  - It spawn two terminal windows:
    - In one it will start Mininet with the required arguments.
    - In the other it will launch the controller.
  - Script assumes that the file SimpleLoadBalancer.py is located in */home/ubuntu/pox/ext* directory.

3.2 Method

T1:

1. Started by using wget to get the files from github:
   (a) wget *https:// raw.githubusercontent.com/simehag/TTM4180/master/lab_6 /SimpleLoadBalancer.py*

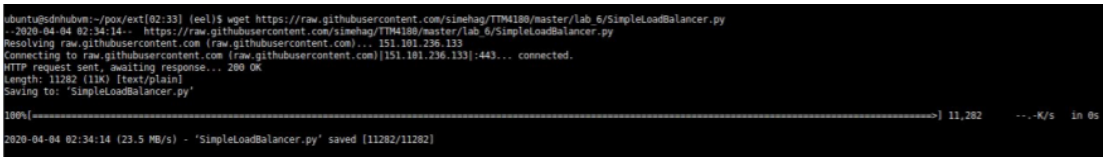


Figure 5: Downloading SimpleLoadBalancer from github

(b) wget *https:// raw.githubusercontent.com/simehag/TTM4180/master/lab_6/run.sh*

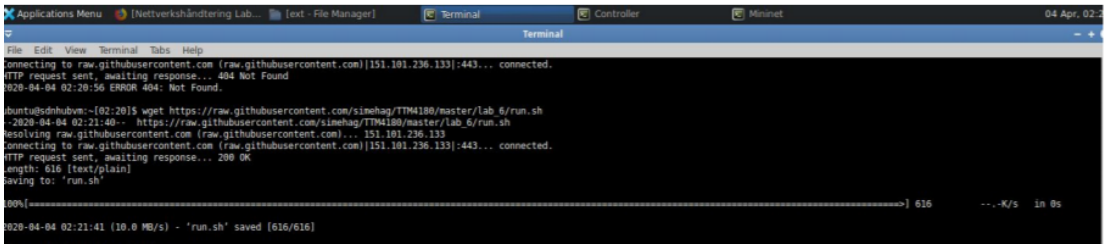


Figure 6: Downloading run.sh from github

2. Moved the SimpleLoadBalancer.py file to */home/ubuntu/pox/ext* directory. The run.sh file stayed in */home/ubuntu*.

T2:

1. Used charm command in the terminal to open Pycharm, and edited SimpleLoadBalancer.py to implement the required functionalities (see section 3.3.
2. After editing the code, ran *chmod +x run.sh* to make the run.sh script executable.

T3:

1. Used the command *sudo wireshark &* in the terminal to open wireshark. Choose the interface "any" to listen on all interfaces.
2. Ran the BASH script in the terminal with: *sh run.sh*.
3. Pinged 2 packets from each host 2 times in this order:
   - h1, h2, h3, h4, h1, h2, h4, h3.
4. Saved the pcapng as Lab6 in the pox directory.

T4:

1. Took screenshots of the controller terminal, which shows that the load balancer distribute the traffic to the backend servers according to the Round Robin algorithm. The pcapng file exists as a separate file.
2. We see that by letting the round_robin select the server, we get flow entries which corresponds to the list, by looking at the "Installed flow rule" in the controller output:
   - 10.0.0.1 $\rightarrow$ 10.0.0.5
   - 10.0.0.2 $\rightarrow$ 10.0.0.6
   - 10.0.0.3 $\rightarrow$ 10.0.0.7
   - 10.0.0.4 $\rightarrow$ 10.0.0.5

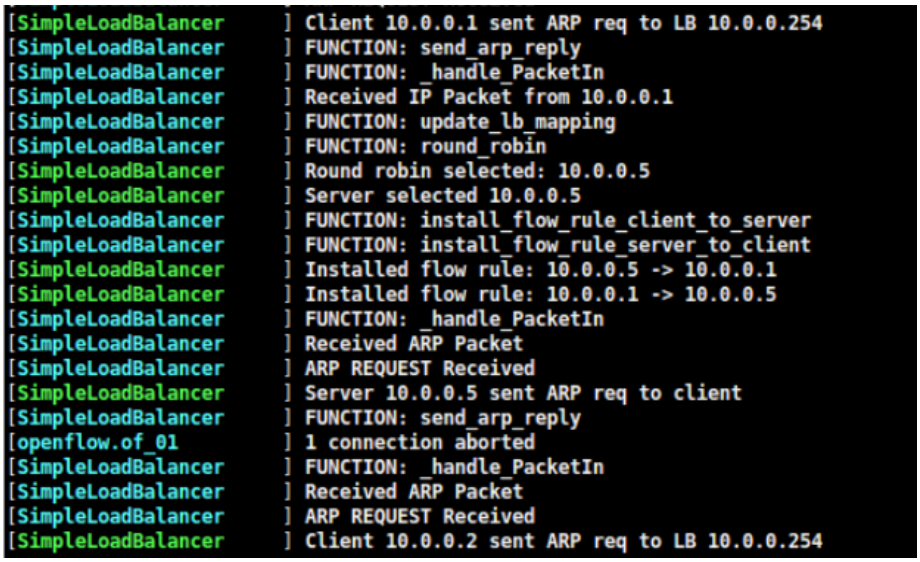


Figure 7: Round robin - H1

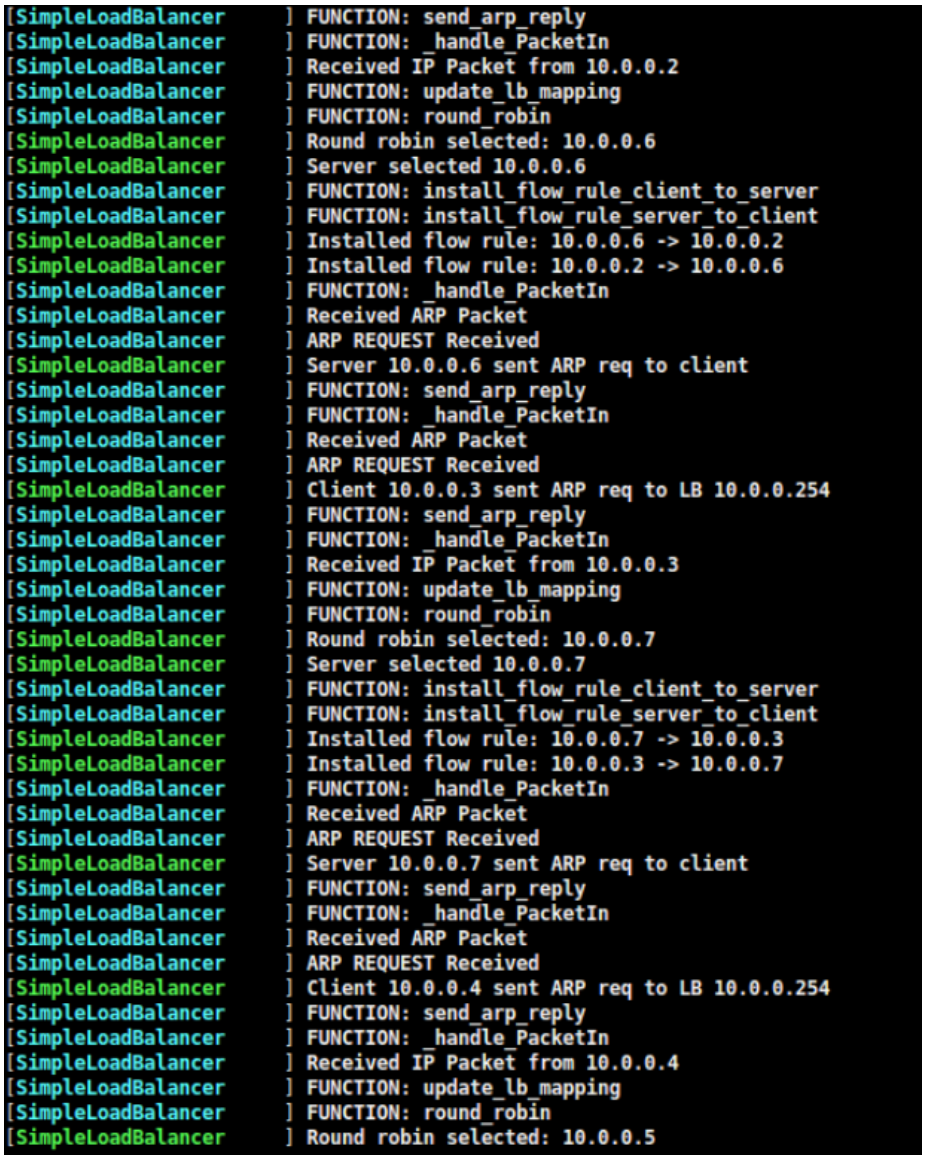


Figure 8: Round robin - H2, H3, H4

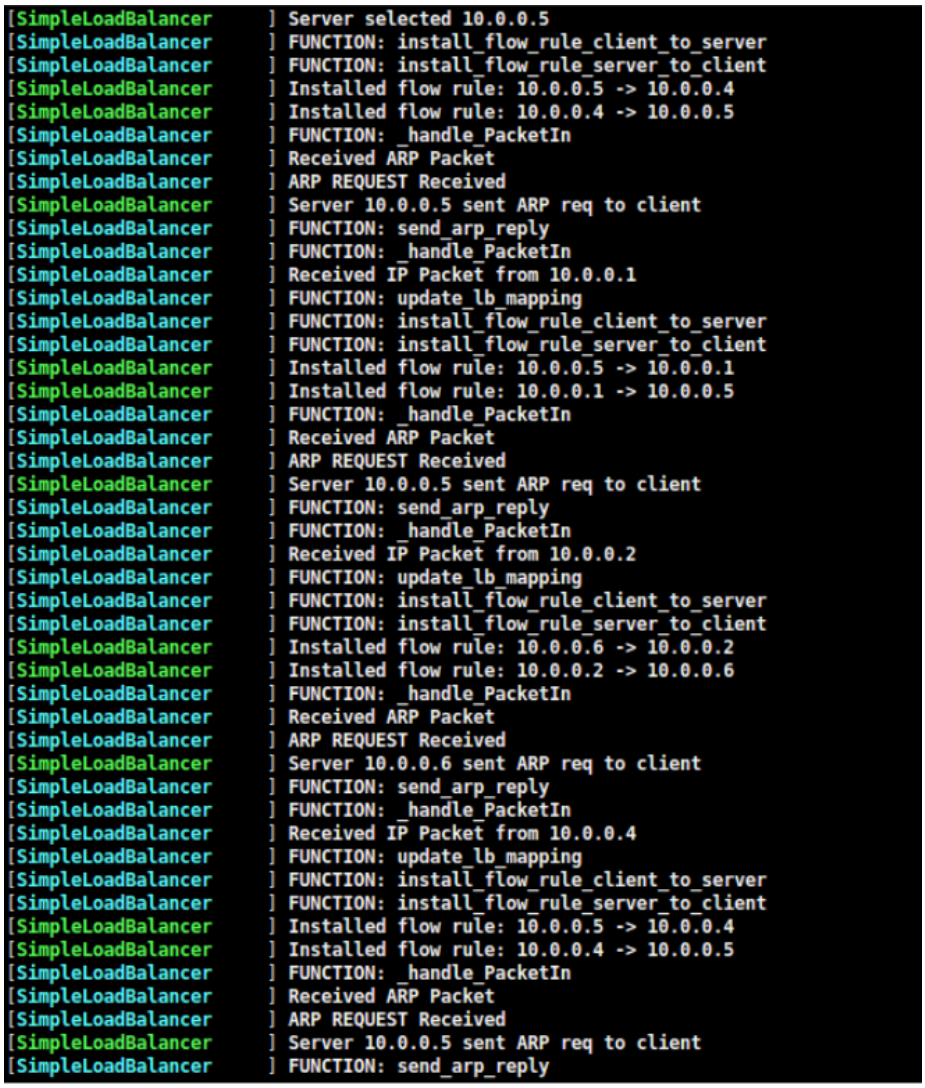


Figure 9: After pinging from H1, H2 H4 for the second time.

Figure 10: After pinging from H3 for the second time.

3.3 Editing the code - Load Balancer Functionality

The code was edited such that it corresponded with the given functionalities in the lab paper (2). The code can be found in the appendix A.1.

1. Switch asks for MAC address of all requests of the servers, by sending ARP request. This was done when the switch connected to the controller, in order to associate these MAC addresses and the corresponding switch port to the real IP addresses of the servers.
   - Funcitons: *_handle_ConnectionUp*.
2. Switch answers to ARP requests from the clients searching for MAC adresses of the service, by sending an ARP reply where the MAC address is set to fake load balancer MAC.
   - Functions: *_handle_PacketIn* and *send_arp_reply*
3. Switch answers to ARP requests from the servers searching for the MAC adresses of clients, by sending an ARP reply where the MAC address is set to fake load balancer MAC.
   - Functions: *_handle_PacketIn* and *send_arp_reply*
4. Switch redirect flows from the clients, by using the Round Robin algorithm, if the packet is not from a server and is destined for LB. The Round Robin algorithm chooses which server to redirect the packet to. A flow entry gets installed in the switch's forwarding table, that maps server $\rightarrow$ client, and client $\rightarrow$ server.
   - When the server gets redirected packets, the source mac is set to load balancer MAC and the source ip is set to client ip, by the load balancer.
   - The load balancer also set the packet destination to server mac.
   - Thus, the server thinks that the load balancer "owns" the client ip.

- Functions: *_handle_PacketIn*, *update_lb_mapping*, *install_flow_rule_server_to_client*, *install_flow_rule_client_to_server* and *round_robin*.

5. The switch directs flows from server to clients, and installs a flow rule from server to client. The MAC source is set to load balancer MAC, and the packet source ip is set to load balancer ip. By doing this clients think that they only talk with the load balancer.
    - Functions: *_handle_PacketIn* and *install_flow_rule_server_to_client*.
6. The IDLE timeout is set to 10 sec.

4 Questions

4.1 Question 1

What type of packets does the load balancer need to manage, in order to behave as mentioned in Section 1.3?

- The load balancer needs to manage ARP packet and IP packets in order to behave as mentioned in Section 1.3. In my code in A.1 I have implemented methods in *_handle_PacketIn* which knows how to handle IP packets, and ARP packets.

4.2 Question 2

Which messages are sent on the "internal" network (between s1 and h5-h7) when s1 connects to the controller?

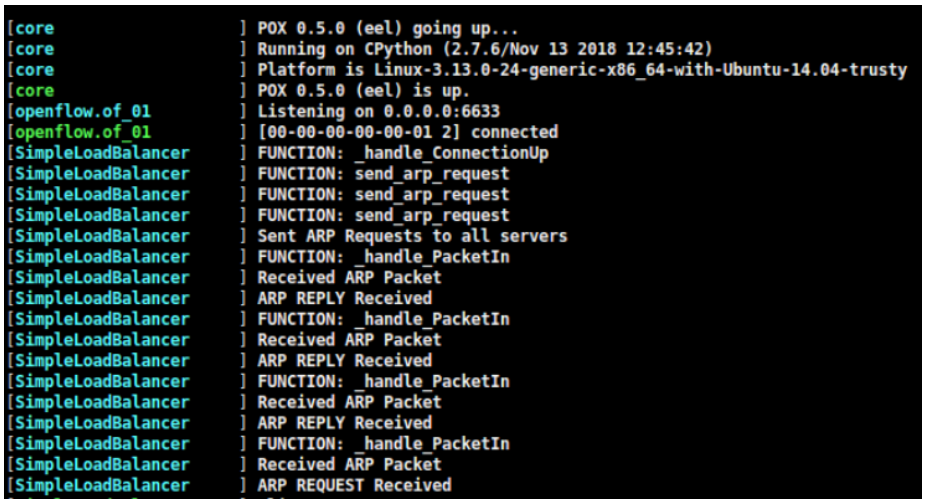


Figure 11: Controller output, when s1 connects to the controller.

- Messages that are sent on the "internal" network when s1 connects to the controller are ARP-requests and ARP-replies.
- When s1 connects to the controller, the function *_handle_ConnectionUp* is called. We have got the *server_ips* as list, which are our backend servers.
- For each *ip* in *server_ips* we send a ARP-request.
- The load balancer (switch) will send ARP-requests which is broadcasted to all the 7 hosts in the network, to find the mac address of the of *ip*.
- This request of info reach everyone but is ignored by the hosts, except the target ip machine. The host which has the following *ip*, will respond to

the switch with a ARP-reply. Server's MAC and port gets added to the *SERVERS* - dictionary as values, where Server's IP is the key.

  - Note that if the server ip is already in the *SERVERS* - dictionary, the same ip-address will not be added twice.

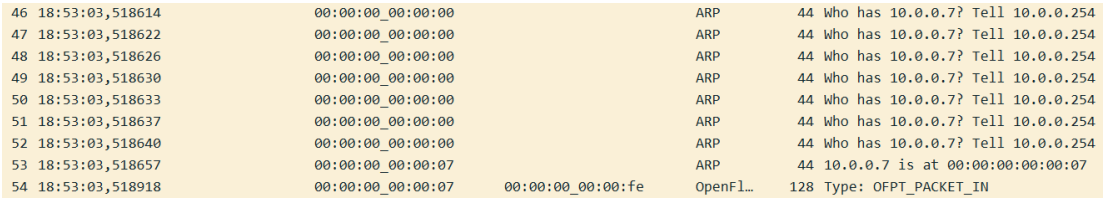
- The ARP-requests are sent 3 times in total from the switch, since we have three elements in the *server_ips* list, which are *10.0.0.5*, *10.0.0.6* and *10.0.0.7*.
- The ARP-replies are sent from *10.0.0.5*, *10.0.0.6* and *10.0.0.7* once.
- We see that in figure 12, 13 and 14, all of them have a *OFPT_PACKET_IN* message at the end. This is because in a OpenFlow SDN, when a switch receives a packet on its port, it will try to match the packet, and see if there are any matching flow entries in the flow table. If there is no match, the switch will by default send the packet to the controller for inspection (3). That's what happened here.

26	18:53:03,516498	00:00:00_00:00:00		ARP	44	Who has 10.0.0.5? Tell 10.0.0.254
27	18:53:03,516503	00:00:00_00:00:00		ARP	44	Who has 10.0.0.5? Tell 10.0.0.254
28	18:53:03,516506	00:00:00_00:00:00		ARP	44	Who has 10.0.0.5? Tell 10.0.0.254
29	18:53:03,516510	00:00:00_00:00:00		ARP	44	Who has 10.0.0.5? Tell 10.0.0.254
30	18:53:03,516514	00:00:00_00:00:00		ARP	44	Who has 10.0.0.5? Tell 10.0.0.254
31	18:53:03,516517	00:00:00_00:00:00		ARP	44	Who has 10.0.0.5? Tell 10.0.0.254
32	18:53:03,516520	00:00:00_00:00:00		ARP	44	Who has 10.0.0.5? Tell 10.0.0.254
33	18:53:03,516535	00:00:00_00:00:05		ARP	44	10.0.0.5 is at 00:00:00:00:00:05
34	18:53:03,516816	00:00:00_00:00:05	00:00:00_00:00:fe	OpenFl…	128	Type: OFPT_PACKET_IN

Figure 12: ARP-Requests and reply where the ip address the load balancer is searching for is: 10.0.0.5

36	18:53:03,517846	00:00:00_00:00:00		ARP	44	Who has 10.0.0.6? Tell 10.0.0.254
37	18:53:03,517854	00:00:00_00:00:00		ARP	44	Who has 10.0.0.6? Tell 10.0.0.254
38	18:53:03,517858	00:00:00_00:00:00		ARP	44	Who has 10.0.0.6? Tell 10.0.0.254
39	18:53:03,517862	00:00:00_00:00:00		ARP	44	Who has 10.0.0.6? Tell 10.0.0.254
40	18:53:03,517866	00:00:00_00:00:00		ARP	44	Who has 10.0.0.6? Tell 10.0.0.254
41	18:53:03,517870	00:00:00_00:00:00		ARP	44	Who has 10.0.0.6? Tell 10.0.0.254
42	18:53:03,517873	00:00:00_00:00:00		ARP	44	Who has 10.0.0.6? Tell 10.0.0.254
43	18:53:03,517895	00:00:00_00:00:06		ARP	44	10.0.0.6 is at 00:00:00:00:00:06
44	18:53:03,518183	00:00:00_00:00:06	00:00:00_00:00:fe	OpenFl…	128	Type: OFPT_PACKET_IN

Figure 13: ARP-Requests and reply where the ip address the load balancer is searching for is: 10.0.0.6



46	18:53:03,518614	00:00:00_00:00:00		ARP	44	Who has 10.0.0.7? Tell 10.0.0.254
47	18:53:03,518622	00:00:00_00:00:00		ARP	44	Who has 10.0.0.7? Tell 10.0.0.254
48	18:53:03,518626	00:00:00_00:00:00		ARP	44	Who has 10.0.0.7? Tell 10.0.0.254
49	18:53:03,518630	00:00:00_00:00:00		ARP	44	Who has 10.0.0.7? Tell 10.0.0.254
50	18:53:03,518633	00:00:00_00:00:00		ARP	44	Who has 10.0.0.7? Tell 10.0.0.254
51	18:53:03,518637	00:00:00_00:00:00		ARP	44	Who has 10.0.0.7? Tell 10.0.0.254
52	18:53:03,518640	00:00:00_00:00:00		ARP	44	Who has 10.0.0.7? Tell 10.0.0.254
53	18:53:03,518657	00:00:00_00:00:07		ARP	44	10.0.0.7 is at 00:00:00:00:00:07
54	18:53:03,518918	00:00:00_00:00:07	00:00:00_00:00:fe	OpenFl…	128	Type: OFPT_PACKET_IN

Figure 14: ARP-Requests and reply where the ip address the load balancer is searching for is: 10.0.0.7

4.3 Question 3

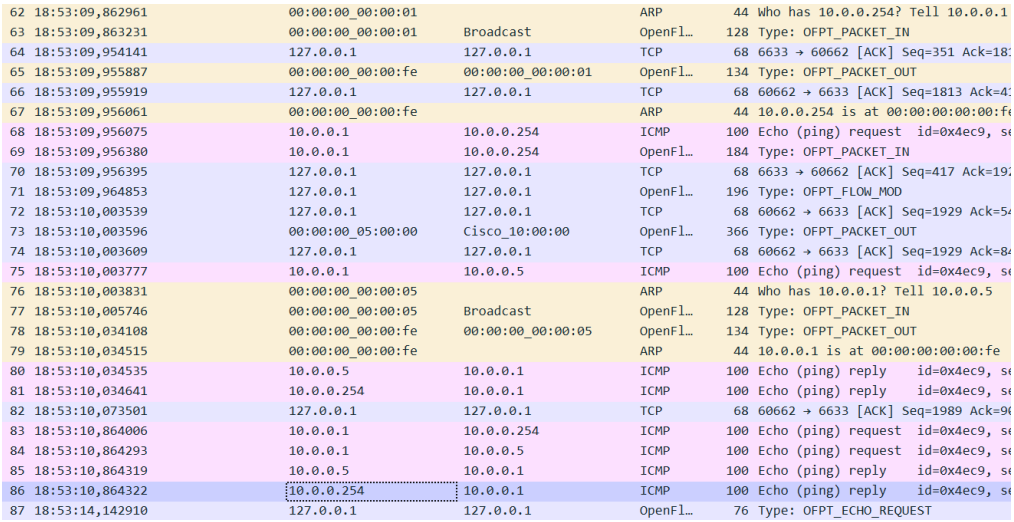
Which messages are sent when h1 pings the service at 10.0.0.254? Include traffic between h1 and switch, switch and the server, and switch and controller, in your answer.

1. **H1 → Load Balancer (Switch):**
   - **Line 62:** H1 starts with sending an ARP-request searching for the mac address of the load balancer-ip (10.0.0.254), which is supposed to be broadcasted. This message is inside a OpenFlow packet.
2. **Loadbalancer (Switch) → Controller:**
   - **Line 63:** Since the load balancer does not have any matching flow entry in its forwarding table, it will forward the packet to the controller by using the *OFPT_PACKET_IN* message and ask for instructions. The controller knows that the mac address 10.0.0.1 was searching for belongs to the load balancer, and therefore doesn't broadcast the packet.
3. **Controller → Loadbalancer (Switch)**:
   - **Line 65:** The Controller sends an *OFPT_PACKET_OUT* message, which tells the load balancer to send a packet to H1, which has a ARP-reply to H1 inside the packet.
4. **Loadbalancer (Switch) → H1**:
   - **Line 67:** H1 gets an ARP-reply packet which contains the mac address that belongs to IP-address 10.0.0.254, which is 00:00:00:00:00:fe.
5. **H1 → LoadBalancer (Switch):**
   - **Line 68:** H1 sends an Echo request to 10.0.0.254.
6. **LoadBalancer(Switch) → Controller:**
   - **Line 69:** This time load balancer got a IP packet. Since it doesnt know what to do with it, it sends it to the controller for instruction by using *OFPT_PACKET_IN*.

7. **Controller → Loadbalancer (Switch):**
    - We have defined in our code in function *_handle_PacketIn*, that when a IP packet arrives from a host, where the destination IP of the packet is set to load balancer ip, we install a flow rule between a server and client (H1). The server gets chosen by using the Round Robin algorithm. Server chosen for h1 is 10.0.0.5.
    - **Line 71:** The *OFPT_FLOW_MOD* installs a flow entry in the load balancer's table - which makes all the packets from h1 go to server 10.0.0.5, which the server would respond to.
8. **Load balancer (Switch) → Server (10.0.0.5):**
    - **Line 75:** Switch forwards the Echo request to the server.
9. **Server (10.0.0.5) → Load balancer (Switch):**
    - **Line 76:** The server wants to send a Echo reply to 10.0.0.1, but doesn't know the mac address of 10.0.0.1. Therefore it sends an ARP-request to the switch, which is supposed to be broadcasted.
10. **Load balancer (Switch) → Controller:**
    - **Line 77:** Since the load balancer doesn't know where to send the packet, it asks the controller for instructions, by forwarding the packet by using *OFPT_PACKET_IN*. The controller knows that the mac address 10.0.0.5 was searching for "belongs" to the load balancer, and therefore doesn't broadcast the packet.
11. **Controller → Loadbalancer (Switch)**:
    - **Line 78:** The Controller sends an *OFPT_PACKET_OUT* message, which tells the load balancer to send a packet to 10.0.0.5, which has a ARP-reply to the server inside the packet.
12. **Load balancer (Switch) → Server (10.0.0.5)**:
    - **Line 79:** Server gets an ARP-reply packet which contains the mac address that belongs to IP-address 10.0.0.1, which is 00:00:00_00:00:fe.
13. **Server (10.0.0.5) → Load balancer (Switch) → H1**
    - **Line 80-86:** Since the server knows the mac adress of 10.0.0.1, which is the fake mac of load balancer, it sends echo reply to load balancer, and the load balancer forwards the packet further to H1.
    - When the rest of the echo requests are sent from H1, H1 will think that the server has the mac address 00:00:00_00:00:fe and IP: 10.0.0.245.
    - When server gets a packet from H1, its mac address of the sender is set to the fake load balancer mac.

- The server will answer back echo replies to the rest of the echo requests.

14. Note here that there are some lines in wireshark that shows TCP packets. OpenFlow uses TCP as it transport protocol. The Controller's port is 6633 (4), and the switch's port is 60662.



62	18:53:09,862961	00:00:00_00:00:01		ARP	44	Who has 10.0.0.254? Tell 10.0.0.1
63	18:53:09,863231	00:00:00_00:00:01	Broadcast	OpenFl…	128	Type: OFPT_PACKET_IN
64	18:53:09,954141	127.0.0.1	127.0.0.1	TCP	68	6633 → 60662 [ACK] Seq=351 Ack=181
65	18:53:09,955887	00:00:00_00:00:fe	00:00:00_00:00:01	OpenFl…	134	Type: OFPT_PACKET_OUT
66	18:53:09,955919	127.0.0.1	127.0.0.1	TCP	68	60662 → 6633 [ACK] Seq=1813 Ack=41
67	18:53:09,956061	00:00:00_00:00:fe		ARP	44	10.0.0.254 is at 00:00:00:00:00:fe
68	18:53:09,956075	10.0.0.1	10.0.0.254	ICMP	100	Echo (ping) request id=0x4ec9, se
69	18:53:09,956380	10.0.0.1	10.0.0.254	OpenFl…	184	Type: OFPT_PACKET_IN
70	18:53:09,956395	127.0.0.1	127.0.0.1	TCP	68	6633 → 60662 [ACK] Seq=417 Ack=197
71	18:53:09,964853	127.0.0.1	127.0.0.1	OpenFl…	196	Type: OFPT_FLOW_MOD
72	18:53:10,003539	127.0.0.1	127.0.0.1	TCP	68	60662 → 6633 [ACK] Seq=1929 Ack=54
73	18:53:10,003596	00:00:00_05:00:00	Cisco_10:00:00	OpenFl…	366	Type: OFPT_PACKET_OUT
74	18:53:10,003609	127.0.0.1	127.0.0.1	TCP	68	60662 → 6633 [ACK] Seq=1929 Ack=84
75	18:53:10,003777	10.0.0.1	10.0.0.5	ICMP	100	Echo (ping) request id=0x4ec9, se
76	18:53:10,003831	00:00:00_00:00:05		ARP	44	Who has 10.0.0.1? Tell 10.0.0.5
77	18:53:10,005746	00:00:00_00:00:05	Broadcast	OpenFl…	128	Type: OFPT_PACKET_IN
78	18:53:10,034108	00:00:00_00:00:fe	00:00:00_00:00:05	OpenFl…	134	Type: OFPT_PACKET_OUT
79	18:53:10,034515	00:00:00_00:00:fe		ARP	44	10.0.0.1 is at 00:00:00:00:00:fe
80	18:53:10,034535	10.0.0.5	10.0.0.1	ICMP	100	Echo (ping) reply id=0x4ec9, se
81	18:53:10,034641	10.0.0.254	10.0.0.1	ICMP	100	Echo (ping) reply id=0x4ec9, se
82	18:53:10,073501	127.0.0.1	127.0.0.1	TCP	68	60662 → 6633 [ACK] Seq=1989 Ack=90
83	18:53:10,864006	10.0.0.1	10.0.0.254	ICMP	100	Echo (ping) request id=0x4ec9, se
84	18:53:10,864293	10.0.0.1	10.0.0.5	ICMP	100	Echo (ping) request id=0x4ec9, se
85	18:53:10,864319	10.0.0.5	10.0.0.1	ICMP	100	Echo (ping) reply id=0x4ec9, se
86	18:53:10,864322	10.0.0.254	10.0.0.1	ICMP	100	Echo (ping) reply id=0x4ec9, se
87	18:53:14,142910	127.0.0.1	127.0.0.1	OpenFl…	76	Type: OFPT_ECHO_REQUEST

Figure 15: Messages in wireshark after sending ping from h1 to 10.0.0.254.

References

[1] educative, "Load balancing - grokking the system design interview." (read 04.04.2020).

[2] N. I. of Science, T. D. of Information Security, and C. Technology, "Lab 6: Load balancer." (read 01.04.2020).

[3] R. Izard, "How to process a packet-in message," Aug 2018. (read 04.04.2020).

[4] O. N. Foundation, "Openflow switch specification." (read 04.04.2020).

A Apendix

A.1 SimpleLoadBalancer.py

```
from pox.core import core
from pox.openflow import *
import pox.openflow.libopenflow_01 as of
from pox.lib.packet.arp import arp
from pox.lib.packet.ipv4 import ipv4
from pox.lib.addresses import EthAddr, IPAddr
log = core.getLogger()
import time
import random
import pox.log.color


IDLE_TIMEOUT = 10
LOADBALANCER_MAC = EthAddr("00:00:00:00:00:FE")
ETHERNET_BROADCAST_ADDRESS=EthAddr("ff:ff:ff:ff:ff:ff")

class SimpleLoadBalancer(object):

  def __init__(self, service_ip, server_ips = []):
    core.openflow.addListeners(self)
    self.SERVERS = {} # IPAddr(SERVER_IP)]={'server_mac':EthAddr(
    SERVER_MAC),'port': PORT_TO_SERVER}
    self.CLIENTS = {}
    self.LOADBALANCER_MAP = {} # Mapping between clients and
    servers
    self.LOADBALANCER_IP = service_ip
    self.SERVER_IPS = server_ips
    self.ROBIN_COUNT = 0

  def _handle_ConnectionUp(self, event):
    self.connection = event.connection
    log.debug("FUNCTION: _handle_ConnectionUp")
    for ip in self.SERVER_IPS:
      selected_server_ip = ip
      self.send_arp_request(self.connection, selected_server_ip)
    log.debug("Sent ARP Requests to all servers")

  def round_robin(self):
    log.debug("FUNCTION: round_robin")
    a = self.SERVERS.keys()
    if self.ROBIN_COUNT == len(self.SERVER_IPS):
      self.ROBIN_COUNT = 0
    server = a[self.ROBIN_COUNT]
    self.ROBIN_COUNT += 1
    log.info("Round robin selected: %s" % server)
    return server

  def update_lb_mapping(self, client_ip):
    log.debug("FUNCTION: update_lb_mapping")
    if client_ip in self.CLIENTS.keys():
      if client_ip not in self.LOADBALANCER_MAP.keys():
        selected_server = self.round_robin()
        log.info("Server selected %s "%selected_server)
```

```
52          self.LOADBALANCER_MAP[client_ip]=selected_server
53
54
55  def send_arp_reply(self, packet, connection, outport):
56    log.debug("FUNCTION: send_arp_reply")
57
58                # Create an ARP reply
59    arp_rep= arp()
60    arp_rep.hwtype = arp_rep.HW_TYPE_ETHERNET
61    arp_rep.prototype = arp_rep.PROTO_TYPE_IP
62    arp_rep.hwlen = 6
63    arp_rep.protolen = arp_rep.protolen
64    arp_rep.opcode = arp_rep.REPLY
65
66                # Set MAC destination and source
67    arp_rep.hwdst = packet.src
68    arp_rep.hwsrc = LOADBALANCER_MAC
69
70    #Reverse the src, dest to have an answer. Set IP source and
      destination
71    arp_rep.protosrc = packet.payload.protodst
72    arp_rep.protodst = packet.payload.protosrc
73
74    # Create ethernet frame, set packet type, dst, src
75    eth = ethernet()
76    eth.type = ethernet.ARP_TYPE
77    eth.dst = packet.src
78    eth.src = LOADBALANCER_MAC
79    eth.set_payload(arp_rep)
80
81    # Create the necessary Openflow Message to make the switch send
       the ARP Reply
82    msg = of.ofp_packet_out()
83    msg.data = eth.pack()
84
85    # Append the output port which the packet should be forwarded
      to.
86    msg.actions.append(of.ofp_action_output(port = of.OFPP_IN_PORT)
      )
87    msg.in_port = outport
88    connection.send(msg)
89
90
91  def send_arp_request(self, connection, ip):
92
93    log.debug("FUNCTION: send_arp_request")
94
95    arp_req = arp()
96    arp_req.hwtype = arp_req.HW_TYPE_ETHERNET
97    arp_req.prototype = arp_req.PROTO_TYPE_IP
98    arp_req.hwlen = 6
99    arp_req.protolen = arp_req.protolen
100   arp_req.opcode = arp_req.REQUEST # Set the opcode
101
102   arp_req.protodst = ip # IP the load balancer is looking for
103   arp_req.hwsrc = LOADBALANCER_MAC # Set the MAC source of the
      ARP REQUEST
```

```
104     arp_req.hwdst = ETHERNET_BROADCAST_ADDRESS # Set the MAC
        address in such a way that the packet is marked as a Broadcast
105     arp_req.protosrc = self.LOADBALANCER_IP # Set the IP source of
        the ARP REQUEST
106
107     eth = ethernet()
108     eth.type = ethernet.ARP_TYPE
109     # eth.src =LOADBALANCER_MAC
110     eth.dst = ETHERNET_BROADCAST_ADDRESS
111     eth.set_payload(arp_req)
112
113     msg = of.ofp_packet_out()
114     msg.data = eth.pack()
115     msg.actions.append(of.ofp_action_nw_addr(of.OFPAT_SET_NW_DST,ip
        ))
116
117                 # Append an action to the message which makes the
        switch flood the packet out
118     msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
119     connection.send(msg)
120
121
122
123   def install_flow_rule_client_to_server(self,event, connection,
        outport, client_ip, server_ip):
124     log.debug("FUNCTION: install_flow_rule_client_to_server")
125     self.install_flow_rule_server_to_client(connection, event.port,
         server_ip,client_ip)
126
127                 # Create an instance of the type of Openflow packet
         you need to install flow table entries
128     msg = of.ofp_flow_mod()
129     msg.idle_timeout = IDLE_TIMEOUT
130
131     msg.match.dl_type=ethernet.IP_TYPE
132
133     # MATCH on destination and source IP
134     msg.match.nw_src = client_ip
135     msg.match.nw_dst = self.LOADBALANCER_IP
136
137     # SET dl_addr source and destination addresses
138     msg.actions.append(of.ofp_action_dl_addr.set_dst(self.SERVERS[
        server_ip].get('server_mac')))
139     msg.actions.append(of.ofp_action_dl_addr.set_src(
        LOADBALANCER_MAC))
140
141     # SET nw_addr source and destination addresses
142     msg.actions.append(of.ofp_action_nw_addr.set_src(client_ip))
143     msg.actions.append(of.ofp_action_nw_addr.set_dst(server_ip))
144
145     # Set Port to send matching packets out
146     msg.actions.append(of.ofp_action_output(port=outport))
147
148     self.connection.send(msg)
149     log.info("Installed flow rule: %s -> %s" % (client_ip,server_ip
        ))
150
```

```
151   def install_flow_rule_server_to_client(self, connection, outport,
          server_ip, client_ip):
152     log.debug("FUNCTION: install_flow_rule_server_to_client")
153
154                 # Create an instance of the type of Openflow packet
       you need to install flow table entries
155     msg = of.ofp_flow_mod()
156     msg.idle_timeout = IDLE_TIMEOUT
157
158     msg.match.dl_type=ethernet.IP_TYPE
159
160     # MATCH on destination and source IP
161     msg.match.nw_src = server_ip
162     msg.match.nw_dst = client_ip
163
164     # SET dl_addr source and destination addresses
165     msg.actions.append(of.ofp_action_dl_addr.set_dst(self.CLIENTS[
      client_ip].get('client_mac')))
166     msg.actions.append(of.ofp_action_dl_addr.set_src(
      LOADBALANCER_MAC))
167
168     # SET nw_addr source and destination addresses
169     msg.actions.append(of.ofp_action_nw_addr.set_src(self.
      LOADBALANCER_IP))
170     msg.actions.append(of.ofp_action_nw_addr.set_dst(client_ip))
171
172     # Set Port to send matching packets out
173     msg.actions.append(of.ofp_action_output(port=outport))
174     self.connection.send(msg)
175     log.info("Installed flow rule: %s -> %s" % (server_ip,client_ip
      ))
176
177   def _handle_PacketIn(self, event):
178     log.debug("FUNCTION: _handle_PacketIn")
179     packet = event.parsed
180     connection = event.connection
181     inport = event.port
182     if packet.type == packet.LLDP_TYPE or packet.type == packet.
      IPV6_TYPE:
183       log.info("Received LLDP or IPv6 Packet...")
184
185     # Handle ARP Packets
186     elif packet.type == packet.ARP_TYPE:
187       log.debug("Received ARP Packet")
188       response = packet.payload
189
190       # Handle ARP replies
191       if response.opcode == response.REPLY:
192         log.debug("ARP REPLY Received")
193         if response.protosrc not in self.SERVERS.keys():
194           # Add Servers MAC and port to SERVERS dict
195           self.SERVERS[IPAddr(response.protosrc)] = {'server_mac':
      EthAddr(packet.payload.hwsrc), 'port': inport}
196
197     # Handle ARP requests
198       elif response.opcode == response.REQUEST:
199         log.debug("ARP REQUEST Received")
```

```
200         if response.protosrc not in self.SERVERS.keys() and
    response.protosrc not in self.CLIENTS.keys():
201         #Insert client's ip  mac and port to a forwarding table
202           self.CLIENTS[response.protosrc]={'client_mac':EthAddr(
    packet.payload.hwsrc),'port':inport}
203
204         if (response.protosrc in self.CLIENTS.keys()and response.
    protodst == self.LOADBALANCER_IP):
205           log.info("Client %s sent ARP req to LB %s"%(response.
    protosrc,response.protodst))
206           # Load Balancer intercepts ARP Client -> Server
207           # Send ARP Reply to the client, include the event.
    connection object
208           self.send_arp_reply(packet, connection, inport)
209
210         elif response.protosrc in self.SERVERS.keys() and response.
    protodst in self.CLIENTS.keys():
211           log.info("Server %s sent ARP req to client"%response.
    protosrc)
212
213           # Load Balancer intercepts ARP from Client <- Server
214           # Send ARP Reply to the Server, include the event.
    connection object
215           self.send_arp_reply(packet, connection, inport)
216         else:
217           log.info("Invalid ARP request")
218
219                 # Handle IP Packets
220     elif packet.type == packet.IP_TYPE:
221       log.debug("Received IP Packet from %s" % packet.next.srcip)
222       # Handle Requests from Clients to Servers
223       # Install flow rule Client -> Server
224       # Check if the packet is destined for the LB and the source
    is not a server :
225       if (packet.next.dstip == self.LOADBALANCER_IP and packet.next
    .srcip not in self.SERVERS.keys()):
226         self.update_lb_mapping(packet.next.srcip)
227
228         # Get client IP from the packet
229         client_ip = packet.payload.srcip
230         server_ip = self.LOADBALANCER_MAP.get(packet.next.srcip)
231
232         # Get Port of Server
233         outport =  int(self.SERVERS[server_ip].get('port'))
234
235         self.install_flow_rule_client_to_server(event,connection,
    outport, client_ip,server_ip)
236
237         eth = ethernet()
238         eth.type = ethernet.IP_TYPE
239         eth.src = LOADBALANCER_MAC
240         eth.dst = self.SERVERS[server_ip].get('server_mac')
241         eth.set_payload(packet.next)
242
243         # Send the first packet (which was sent to the controller
    from the switch)
244         # to the chosen server, so there is no packetloss
```

```
245         msg= of.ofp_packet_out()
246         msg.data = eth.pack()
247         msg.in_port = inport
248 
249         # Add an action which sets the MAC source to the LB's MAC
250         msg.actions.append(of.ofp_action_dl_addr.set_src(
    LOADBALANCER_MAC))
251         # Add an action which sets the MAC destination to the
    intended destination...
252         msg.actions.append(of.ofp_action_dl_addr.set_dst(self.
    SERVERS[server_ip].get('server_mac')))
253 
254         # Add an action which sets the IP source
255         msg.actions.append((of.ofp_action_nw_addr.set_src(client_ip
    )))
256         # Add an action which sets the IP destination
257         msg.actions.append(of.ofp_action_nw_addr.set_dst(server_ip)
    )
258         # Add an action which sets the Outport
259         msg.actions.append(of.ofp_action_output(port=outport))
260 
261         connection.send(msg)
262 
263       # Handle traffic from Server to Client
264       # Install flow rule Client <- Server
265       elif packet.next.dstip in self.CLIENTS.keys():
266         log.info("Installing flow rule from Server -> Client")
267         if packet.next.srcip in self.SERVERS.keys():
268                                         # Get the source IP from
    the IP Packet
269           server_ip = packet.next.srcip
270 
271           client_ip = self.LOADBALANCER_MAP.keys()[list(self.
    LOADBALANCER_MAP.values()).index(packet.next.srcip)]
272           outport=int(self.CLIENTS[client_ip].get('port'))
273           self.install_flow_rule_server_to_client(connection,
    outport, server_ip,client_ip)
274 
275           eth = ethernet()
276           eth.type =  ethernet.IP_TYPE
277           eth.src = LOADBALANCER_MAC
278           eth.dst = self.CLIENTS[client_ip].get('client_mac')
279           eth.set_payload(packet.next)
280 
281 
282           # Send the first packet (which was sent to the controller
     from the switch)
283           # to the chosen server, so there is no packetloss
284           msg = of.ofp_packet_out()
285           msg.data = eth.pack()
286           msg.in_port = inport
287 
288           # Add an action which sets the MAC source to the LB's MAC
289           msg.actions.append(of.ofp_action_dl_addr.set_src(
    LOADBALANCER_MAC))
290           # Add an action which sets the MAC destination to the
    intended destination...
```

```
291           msg.actions.append(of.ofp_action_dl_addr.set_dst(self.
      CLIENTS[client_ip].get('client_mac')))
292
293           # Add an action which sets the IP source
294           msg.actions.append(of.ofp_action_nw_addr.set_src(self.
      LOADBALANCER_IP))
295           # Add an action which sets the IP destination
296           msg.actions.append(of.ofp_action_nw_addr.set_dst(
      client_ip))
297           # Add an action which sets the Outport
298           msg.actions.append(of.ofp_action_output(port=outport))
299
300           self.connection.send(msg)
301
302
303     else:
304       log.info("Unknown Packet type: %s" % packet.type)
305       return
306
307     return
308
309 def launch(loadbalancer, servers):
310   # Color-coding and pretty-printing the log output
311   pox.log.color.launch()
312   pox.log.launch(format="[@@@bold@@@level%(name)-23s@@@reset] " +
313                  "@@@bold%(message)s@@@normal")
314   log.info("Loading Simple Load Balancer module:\n\n
       ----------------------------------CONFIG
       ----------------------------------\n")
315   server_ips = servers.replace(","," ").split()
316   server_ips = [IPAddr(x) for x in server_ips]
317   loadbalancer_ip = IPAddr(loadbalancer)
318   log.info("Loadbalancer IP: %s" % loadbalancer_ip)
319   log.info("Backend Server IPs: %s\n\n
       ----------------------------------------\n\n" % ', '.join(str(ip
       ) for ip in server_ips))
320   core.registerNew(SimpleLoadBalancer, loadbalancer_ip, server_ips)
```

Listing 1: Edited code from T3