

Software-Defined Networks

Lab 9 Python Load Balancer Application

University of Colorado Boulder
Department of Computer Science

Professor Levi Perigo, Ph.D.

Lab Summary

Server load-balancers (SLBs) are complex and expensive devices that perform load-balancing across servers based on several factors such as server capability, incoming requests, or round-robin fashion. SDN is a networking concept that aims to centralize networks and make network flows programmable, and NFV focusses on virtualized network functions. SDN/NFV can be used to manage networks better and reduce CapEx/OpEx. SDN-based load-balancers use SDNFV functions and applications to create flexible, programmable, and virtual load-balancing that can be deployed, managed and manipulated with ease in the industry. In this lab, students will deploy, study, and evaluate simple load-balancers, to increase understanding, and will develop their own round-robin load-balancer to application to show how a programming application can be used to control network behavior.

The objectives of this lab are to be used as guidelines, and additional exploration by the student is strongly encouraged.

Objective 1 – Implement a Stateless Round-Robin Load-balancer

1. Initialize Ryu controller with the simple_switch_13.py application. Paste the screenshot of the command used to initialize the ryu application and paste a successful ping across two hosts. [5 points]

Ryu CLI -

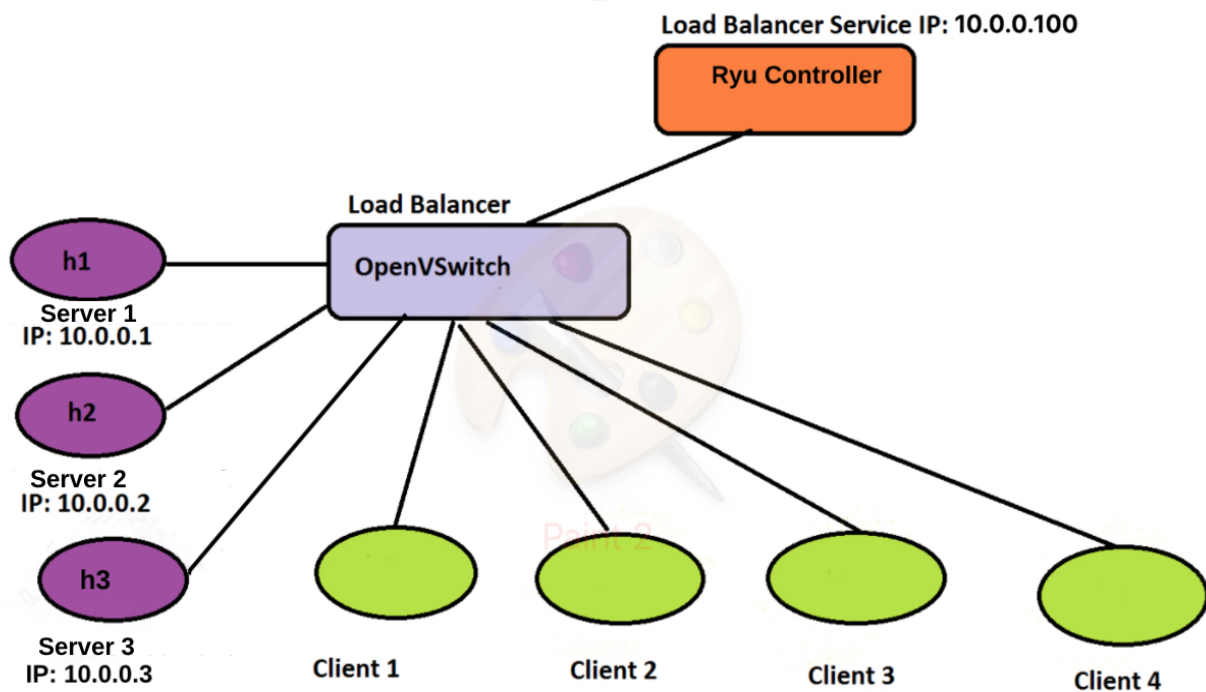
```
sdn@sdn-controllers:~/ryu/ryu/app$ sudo ryu run simple_switch_13.py
loading app simple_switch_13.py
loading app ryu.controller.ofp_handler
instantiating app simple_switch_13.py of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
```

Mininet CLI -

```
mininet@mininet-ofn:~$ sudo mn --controller=remote,ip=192.168.56.101 --mac --switch=ovsk,protocols=OpenFlow13 --topo=single,2
*** Creating network
*** Adding controller
Connecting to remote controller at 192.168.56.101:6653
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
mininet>
```

You can terminate your learning switch app now.

2. Use this link to get started with writing applications in Ryu:
https://ryu.readthedocs.io/en/latest/writing_ryu_app.html
3. Your task for this lab would be to develop a load balancer Ryu application along with the following specifications:
 - ❑ Refer to the network diagram below. That is the topology which you will have to use for this lab using Mininet.



- ❑ The Ryu controller application that you are writing can be a new Python file (a separate one) or it can be run alongside with the existing learning switch app or be inbuilt with the learning switch code. This is left to your decision. Please mention your approach in your submission. - **objective achieved**

Application - HTTP site listening on Port 8080

used python http module

- ❑ The objective of the load balancer application is to ensure that four clients (Client 1 to 4) would be able to talk to different servers (Server 1 to 3) in a round-robin fashion. . - **objective achieved**
- ❑ The main task to note is that the clients know only the IP address of the load balancer (10.0.0.100) and not the IP addresses of the individual servers. The server IPs are known to the Ryu controller app though. - . - **objective achieved**
- ❑ Use the service IP address of the load-balancer as 10.0.0.100 and the IP addresses of the servers as 10.0.0.1 (h1), 10.0.0.2 (h2), and 10.0.0.3 (h3). Client IPs are your choice. - . - **objective achieved**
- ❑ The servers can be hosting any application of your choice. - . - **objective achieved**
- ❑ Your application code should have print statements notifying the details about

the in and out packet (for example about details on, which client sent a request, which server is the packet being transmitted to, the server from which the reply is being received, how it is being sent back to the client). Not having print statements in your controller app will result in a loss of 20 points. - . - **objective achieved**

Output -

Mininet -

===== STATELESS LoadBalancer =====

h5 IP - 10.0.0.5

```
mininet> h5 curl http://10.0.0.100:8080
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
<li><a href=".bash_history">.bash_history</a></li>
<li><a href=".bash_logout">.bash_logout</a></li>
<li><a href=".bashrc">.bashrc</a></li>
<li><a href=".cache/">.cache/</a></li>
<li><a href=".config/">.config/</a></li>
<li><a href=".gitconfig">.gitconfig</a></li>
<li><a href=".mininet_history">.mininet_history</a></li>
<li><a href=".npm/">.npm/</a></li>
<li><a href=".profile">.profile</a></li>
<li><a href=".rnd">.rnd</a></li>
<li><a href=".viminfo">.viminfo</a></li>
<li><a href=".wireshark/">.wireshark/</a></li>
<li><a href=".Xauthority">.Xauthority</a></li>
<li><a href="install-mininet-vm.sh">install-mininet-vm.sh</a></li>
<li><a href="lab8.py">lab8.py</a></li>
<li><a href="lab9.mn">lab9.mn</a></li>
<li><a href="lab9.py">lab9.py</a></li>
<li><a href="loxigen/">loxigen/</a></li>
<li><a href="mininet/">mininet/</a></li>
<li><a href="node_modules/">node_modules/</a></li>
<li><a href="oflops/">oflops/</a></li>
<li><a href="oftest/">oftest/</a></li>
<li><a href="OpenDaylight-Openflow-App/">OpenDaylight-Openflow-App/</a></li>
<li><a href="openflow/">openflow/</a></li>
<li><a href="package-lock.json">package-lock.json</a></li>
<li><a href="pox/">pox/</a></li>
<li><a href="test.py">test.py</a></li>
</ul>
<hr>
</body>
</html>
```

Output -
Mininet -
ARP request handled by controller for Load balancer IP - 1

ad for first time [new TCP socket]

ARP request handled by controller for Load balancer IP - 10.0.0.100

```
packet in 0000000000000001 00:00:00:00:00:01 33:33:00:00:00:02 1
packet in 0000000000000001 00:00:00:00:00:06 33:33:00:00:00:02 6
packet in 0000000000000001 00:00:00:00:00:07 33:33:00:00:00:02 7
packet in 0000000000000001 00:00:00:00:00:02 33:33:00:00:00:02 2
----- Handling ARP Request for LoadBalancer IP-----
* Received ARP Request: Who has 10.0.0.100 ? from 10.0.0.5
* Adding flow to install MAC table in the switch:
  DPID: 0000000000000001
  Match Criteria:
    MAC Destination: 00:00:00:00:00:05
  Actions:
    Out Port: 5
* Send ARP Reply to 10.0.0.5
  Answer: 10.0.0.100 has 00:00:00:00:00:64
ethernet(dst='00:00:00:00:00:05',ethertype=2054,src='00:00:00:00:00:64'), arp(dst_ip='10.0.0.5',dst_mac='00:00:00:00:00:05',hlen=6,hwt
ype=1,opcode=2,plen=4,proto=2048,src_ip='10.0.0.100',src_mac='00:00:00:00:00:64')
```

HTTP Request being handled for first time [new TCP socket] [load balanced to h1 server]

```
----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.5
* New HTTP request received
* New Socket connection request
* Adding flow for reverse traffic from server directly to client:
  DPID: 0000000000000001
  Match Criteria:
    IPv4 Destination: 10.0.0.5
    IPv4 Source: 10.0.0.1
    Protocol: TCP
    TCP Source Port: 8080
  Actions:
    Set Field: IPv4 Source - 10.0.0.100
    Out Port: 5
* 10.0.0.5 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.1
* Request sent to backend server 10.0.0.1
packet in 0000000000000001 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000001 00:00:00:00:00:05 00:00:00:00:00:01 5
----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.5
* Server already selected for socket connection('10.0.0.5', 34760, '10.0.0.100', 8080)
* Consecutive TCP packets for same socket - SYN/ACK, FIN, mapped to the same server
* 10.0.0.5 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.1
* Request sent to backend server 10.0.0.1
----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.5
* Server already selected for socket connection('10.0.0.5', 34760, '10.0.0.100', 8080)
* Consecutive TCP packets for same socket - SYN/ACK, FIN, mapped to the same server
* 10.0.0.5 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.1
* Request sent to backend server 10.0.0.1
```

New HTTP request from same client 10.0.0.5 [load balanced to h2 server]

```
----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.5
* New HTTP request received
* New Socket connection request
* Adding flow for reverse traffic from server directly to client:
  DPID: 0000000000000001
  Match Criteria:
    IPv4 Destination: 10.0.0.5
    IPv4 Source: 10.0.0.2
    Protocol: TCP
    TCP Source Port: 8080
  Actions:
    Set Field: IPv4 Source - 10.0.0.100
    Out Port: 5
* 10.0.0.5 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.2
* Request sent to backend server 10.0.0.2
packet in 0000000000000001 00:00:00:00:00:02 ff:ff:ff:ff:ff:ff 2
packet in 0000000000000001 00:00:00:00:00:05 00:00:00:00:00:02 5

----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.5
* Server already selected for socket connection('10.0.0.5', 34762, '10.0.0.100', 8080)
* Consecutive TCP packets for same socket - SYN/ACK, FIN, mapped to the same server
* 10.0.0.5 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.2
* Request sent to backend server 10.0.0.2
```

Ryu works on an event basis. When an event is received by the controller, you can specify a custom behavior for that event. This is done by identifying the event and writing a function containing the custom behavior code. Whenever that event hits the controller, it will automatically execute the function that you have defined in your code. This is why you just need to write the function definition and not call it in anywhere within your code. These special functions can be identified easily. Just before the “def function_name(...):” line, you will see:

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)

This is an example for the Packet-In event to the controller.

Below the @ line, your function definition can start. Example: `_packet_in_handler(self, ev)`.

To create a flow entry, you will have to create three entities: match, actions, mod.

Then you are going to have to send the mod using `datapath.send_msg()` function. Reusing the

variables from above:

```
match = parser.OFPMatch(param1=value, param2=value ...)
actions = [parser.OFPActionOutput(OUTPUT_PORT)]
mod = parser.OFPFlowMod(datapath=datapath, match=match, actions=actions,
priority=INTEGER, idle_timeout=INTEGER, hard_timeout=INTEGER, cookie=INTEGER,
command=TYPE_OF_FLOW_MOD)
datapath.send_msg(mod)
```

where,

- param1, param2,.. could be in_port, ip_src, ip_dst, eth_src, eth_dst and many more parameters.
- OUTPUT_PORT is either be the integer output Port number or special values such as ofproto.OFPP_NORMAL, ofproto.OFPP_FLOOD etc.
- TYPE_OF_FLOW_MOD defines the type of flow mod. Values such as ofproto.OFPFC_ADD, ofproto.OFPFC_MODIFY, ofproto.OFPFC_DELETE are valid.
- INTEGER - 0 (or 1, 2 etc.)

To find more about OFPMatch, OFPActionOutput and OFPFlowMod, you will just have to visit this source page and search for “class <NAME>” to get an awesome idea of what to include, and examples on how to use the functions. If you are using OF v1.0 use this link.

The sequence:

- **match** helps you match the incoming packet with conditions specified in the arguments.
- **actions** sets the action for the switch (forwarding etc.)
- OpenFlow v1.3 supports something called as **instructions**, which you could look up online to perform complex action tasks.
- **mod** prepares the flow message based on the match and actions variables, and also sets the parameters like priority and timeout. command argument is essential for identifying the type of Flow message to be sent to the switches.
- **send_msg(mod)** sends out the message.

3. Think of two easy server applications that you should use to test your code. List the applications over here and provide a simple way of starting/testing them. **[10 points]**

via Python -

Steps: 1) check if python is installed, which is provided by default in Linux distro

2) `python -m http.server 8080`

via PHP -

Steps : 1) install PHP in Debian distro - `sudo apt install php libapache2-mod-php`

2) `php -S 0.0.0.0:8080`

Once the server is started we can simply do curl on the client.

[The definition of Round-robin load balancing is left to your interpretation. As long as consistent round-robin behavior is achieved, and you explain it well in your lab report, points would be awarded.]

4. What are the different types of Round-robin load balancing methods that you can come up with? How would their behavior vary with respect to client <-> server access? Write about two such implementations in your answer. **[10 points]**

Weighted Round Robin -

For a particular backend server , a weight can be assigned in the loadbalancer. This would mean, we can simply give more preference to a backend server that has more compute,storage or network resources, over other backend servers.

For example:

Server1 → Weight : 70%

Out of 10 packets, 7 packets would be routed to this backend server

Server2 → Weight 30%

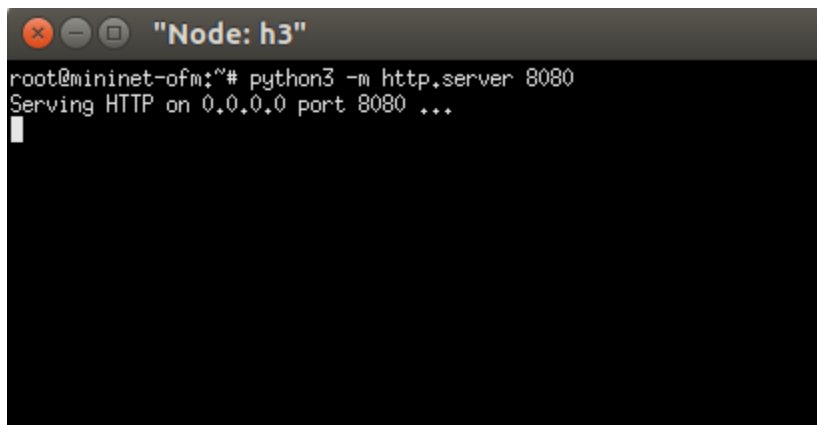
Out of 10 packets, 3 packets would be routed to this backend server

Dynamic Round Robin Load Balancing

If there are N backend servers, the HTTP requests are sent to each server in a circular fashion. We don't set weight but, this can be imagined as a weighted Round Robin Load Balancer where each backend server is 100%.

5. Use xterm to pull up individual server terminals (h1, h2, h3). What command did you use for starting your choice of application on the server? Paste the screenshots of a server running one such application. [5 points]

Application used - python http module



```
root@mininet-ofm:~# python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 ...
```

6. Using appropriate commands, make your clients talk to the load-balancer IP in a round-robin fashion and ultimately reach the server application. Paste the screenshot of the command and its corresponding output on all four hosts. [10 points]

Ryu server →

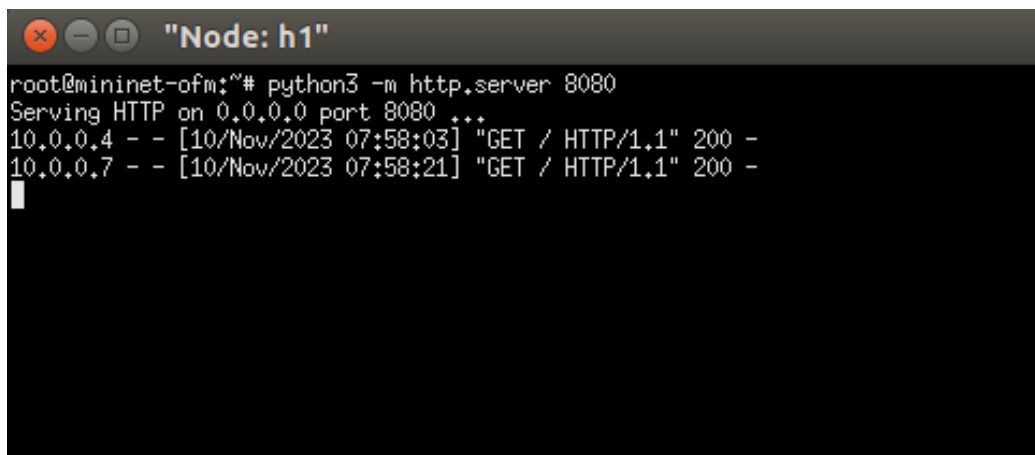
```
^Csdn@sdn-controllers:~/ryu/ryu/app$ sudo ryu run kiran_simple_switch.py
loading app kiran_simple_switch.py
loading app ryu.controller.ofp_handler
instantiating app kiran_simple_switch.py of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
packet in 00000000000000001 00:00:00:00:00:02 33:33:00:00:00:16 2
packet in 00000000000000001 00:00:00:00:00:05 33:33:00:00:00:16 5
packet in 00000000000000001 00:00:00:00:00:01 33:33:00:00:00:16 1
```

index.html → it has single line “Hello World”

```
mininet> xterm h1 h2 h3 sts. [10 points]
mininet> h4 curl http://10.0.0.100:8080
Hello World!
mininet> h5 curl http://10.0.0.100:8080
Hello World!
mininet> h6 curl http://10.0.0.100:8080
Hello World!
mininet> h7 curl http://10.0.0.100:8080
Hello World!
mininet>
```

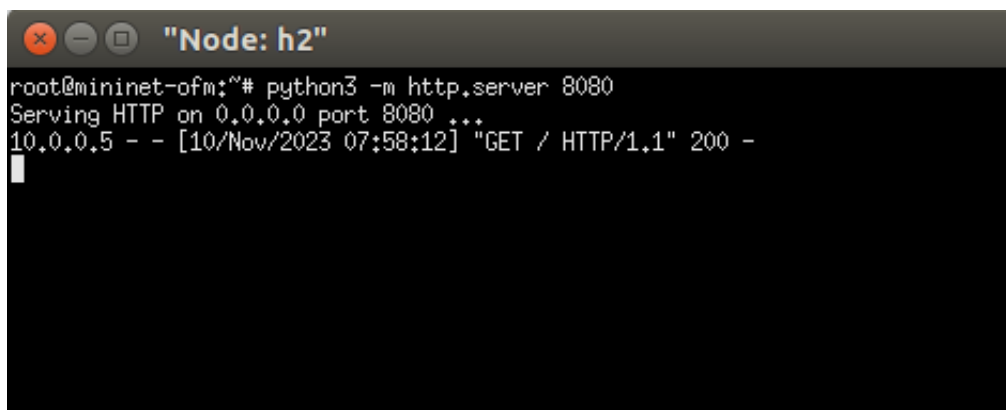
7. Paste screenshots of the terminal of corresponding servers that served the client request from the four hosts. [10 points]

H1 server CLI output - [processed request for client h4 and h7]



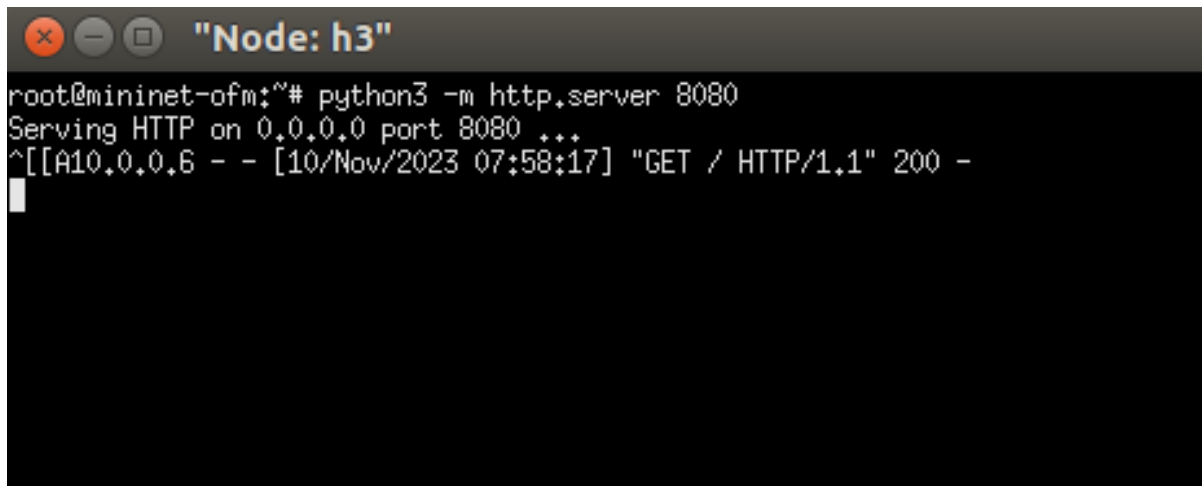
```
root@mininet-ofm:~# python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 ...
10.0.0.4 - - [10/Nov/2023 07:58:03] "GET / HTTP/1.1" 200 -
10.0.0.7 - - [10/Nov/2023 07:58:21] "GET / HTTP/1.1" 200 -
```

H2 server CLI output - [processed request for client h5]



```
root@mininet-ofm:~# python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 ...
10.0.0.5 - - [10/Nov/2023 07:58:12] "GET / HTTP/1.1" 200 -
```

H3 server CLI output - [processed request for client h6]

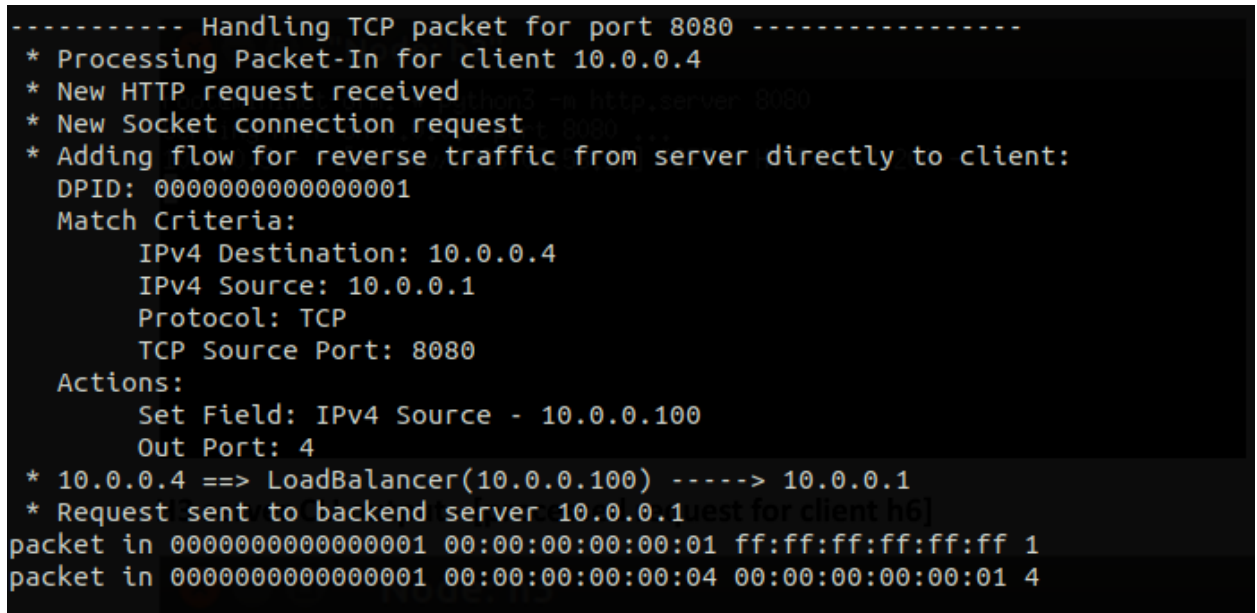


```
root@mininet-ofm:~# python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 ...
^[[A10.0.0.6 - - [10/Nov/2023 07:58:17] "GET / HTTP/1.1" 200 -
```

8. Paste the screenshot of the debug messages and the print statements seen on the Ryu controller's console identifying the traffic that is being directed to the different servers.

[10 points]

Request from h4 to load balancer



```
----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.4
* New HTTP request received
* New Socket connection request
* Adding flow for reverse traffic from server directly to client:
  DPID: 0000000000000001
  Match Criteria:
    IPv4 Destination: 10.0.0.4
    IPv4 Source: 10.0.0.1
    Protocol: TCP
    TCP Source Port: 8080
  Actions:
    Set Field: IPv4 Source - 10.0.0.100
    Out Port: 4
* 10.0.0.4 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.1
* Request sent to backend server 10.0.0.1 [test for client h6]
packet in 0000000000000001 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000001 00:00:00:00:00:04 00:00:00:00:00:01 4
```

Request from h5 to load balancer

```
----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.5 to from server directly to client
* New HTTP request received
* New Socket connection request
* Adding flow for reverse traffic from server directly to client:
  DPID: 00000000000000001
  Match Criteria:
    IPv4 Destination: 10.0.0.5
    IPv4 Source: 10.0.0.2
    Protocol: TCP
    TCP Source Port: 8080
  Actions:
    Set Field: IPv4 Source - 10.0.0.100
    Out Port: 5
* 10.0.0.5 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.2
* Request sent to backend server 10.0.0.2
packet in 00000000000000001 00:00:00:00:00:02 ff:ff:ff:ff:ff:ff 2
packet in 00000000000000001 00:00:00:00:00:05 00:00:00:00:00:02 5
```

Request from h6 to load balancer

```
----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.6 to 10.0.0.5
* New HTTP request received
* New Socket connection request
* Adding flow for reverse traffic from server directly to client:
  DPID: 00000000000000001
  Match Criteria:
    IPv4 Destination: 10.0.0.6
    IPv4 Source: 10.0.0.3
    Protocol: TCP
    TCP Source Port: 8080
  Actions:
    Set Field: IPv4 Source - 10.0.0.100
    Out Port: 6
* 10.0.0.6 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.3
* Request sent to backend server 10.0.0.3
packet in 00000000000000001 00:00:00:00:00:03 ff:ff:ff:ff:ff:ff 3
packet in 00000000000000001 00:00:00:00:00:06 00:00:00:00:00:03 6
```

Request from h7 to load balancer

```
----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.7
* New HTTP request received
* New Socket connection request
* Adding flow for reverse traffic from server directly to client:
DPID: 0000000000000001
Match Criteria:
  IPv4 Destination: 10.0.0.7
  IPv4 Source: 10.0.0.1
  Protocol: TCP
  TCP Source Port: 8080
Actions:
  Set Field: IPv4 Source - 10.0.0.100
  Out Port: 7
* 10.0.0.7 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.1
* Request sent to backend server 10.0.0.1
packet in 0000000000000001 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
packet in 0000000000000001 00:00:00:00:00:07 00:00:00:00:00:01 7
----- Handling TCP packet for port 8080 -----
```

9. Were the requests being served in a round-robin fashion? Why or why not? [5 points]

Yes, the requests are served in round robin fashion. From the above screenshots it can be observed that h7 is server by 10.0.0.1 (H1 server)

In the code, the constructor of RYU is initialized with this [serverIPAddress](#) list.

```
super(SimpleSwitch15, self).__init__(args, kwargs)
self.mac_to_port = {}
self.serverIPAddress = ["10.0.0.1", "10.0.0.2", "10.0.0.3"]
self.serverMACAddress = []
self.loadBalancerIP = "10.0.0.100"
```

The list operates like a queue.

Whenever a request comes in and invokes the event handler, the 0th location from list is popped, recorded as the 'serverIP', and pushed back at the end of the list.

```
serverIP = self.serverIPAddress.pop(0)
self.serverIPAddress.append(serverIP)
serverMAC = self.arp_table[serverIP]
```

Thus the request are sent to servers in a cyclic fashion, which mimics the behavior of round robin load balancer

10. Submit the program along with the lab document. Mention the command to execute the program. Ensure your program follows the specification listed above in box points. Each specification carries a weightage while grading this deliverable. **[90 points]**

11. Give a fancy name to your Load Balancer application and use that as your filename. **[1 point]**

filename - roundRobinLoadbalancerApp.py

12. How does SDN help improve/simplify load-balancing implementation? **[5 points]**

Load Balancing, can be fine grained controlled through SDN.

The best part is I can create custom load balancer, which loadbalance in a weird fashion -

[server1, server2, server3] ⇒ Combined weight of 60%

[server4, server5] ⇒ Combined weight of 40%

This might be a business need and it can be quickly implemented giving a granular access over the business requirements

Additionally, we can implement polymorphism where the same switch acts as load balancer for client requests from public IP. But can also act as a simple L2 switch within the organization.

13. To earn full credit please show the functioning of the code to your TA's.

Objective 3 – Stateful Load-balancer Application

This is the part where you can leverage SDN capabilities to make your network smarter. Modify your existing load-balancer in such a way that it remembers the clients based on any one state that you define. This could be done on an event basis as well. The statefulness of the load-balancer essentially means that it can force individual clients to go to specific servers based on their event/state. You can remove the learning switch component for this exercise. You will have to leverage the Layer 4 - 7 capabilities of SDN while building this stateful Load-balancer application.

Provide the code for your stateful load-balancer (this file should be different from the stateless load-balancer). Submit screenshots of working application and demonstrate it to one of your

TAs for earning full credits. [35 points]

Round Robin Load Balancing with sticky session/stateful behavior -

Rule -

1.) If the client IP is **not** in the stateTable of LoadBalancer.

If the packet is TCP SYN packet of the new socket establishment.

- Next server is picked up from the queue to relay the request to.
- Record the IP address \longleftrightarrow HTTP server IP mapping in the stateTable of the loadbalancer.
- Relay the request to the HTTP server IP popped from the queue.

1) If the client IP is in the stateTable of LoadBalancer.

- Fetch the HTTP server IP from the stateTable
- Relay the request directly to the HTTP server

Mininet CLI (observe the order of request)

```
mininet@mininet-ofm:~$ sudo mn --controller=remote,ip=192.168.56.101 --mac --switch=ovsk,protocols=OpenFlow13 --topo=single,7
\\*** Creating network
*** Adding controller
Connecting to remote controller at 192.168.56.101:6653
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1) (h5, s1) (h6, s1) (h7, s1)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> xterm h1 h2 h3
mininet> h4 curl http://10.0.0.100:8080
Hello World!
mininet> h5 curl http://10.0.0.100:8080
Hello World!
mininet> h4 curl http://10.0.0.100:8080
Hello World!
mininet> h6 curl http://10.0.0.100:8080
Hello World!
mininet> h7 curl http://10.0.0.100:8080
Hello World!
mininet>
```


Even when curl is done third time from h4 it is mapped to the same backend server which was initially used by the client

```
root@mininet-ofm:~# python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 ...
10.0.0.4 - - [10/Nov/2023 08:39:32] "GET / HTTP/1.1" 200 -
10.0.0.4 - - [10/Nov/2023 08:39:52] "GET / HTTP/1.1" 200 -
10.0.0.7 - - [10/Nov/2023 08:40:16] "GET / HTTP/1.1" 200 -
```

h1 (server logs) -

```
root@mininet-ofm:~# python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 ...
10.0.0.5 - - [10/Nov/2023 08:39:42] "GET / HTTP/1.1" 200 -
```

h2 (server logs)

```
root@mininet-ofm:~# python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 ...
10.0.0.6 - - [10/Nov/2023 08:40:05] "GET / HTTP/1.1" 200 -
```

h3 (server logs)

Server Logs -

```

----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.6
* New HTTP request received
* New Socket connection request
* Mapping state: 10.0.0.6 --> 10.0.0.3
* Adding flow for reverse traffic from server directly to client:
  DPID: 0000000000000001
  Match Criteria:
    IPv4 Destination: 10.0.0.6
    IPv4 Source: 10.0.0.3
    Protocol: TCP
    TCP Source Port: 8080
  Actions:
    Set Field: IPv4 Source - 10.0.0.100
    Out Port: 6
* 10.0.0.6 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.3
* Request sent to backend server 10.0.0.3
packet in 0000000000000001 00:00:00:00:00:03 ff:ff:ff:ff:ff:ff 3
packet in 0000000000000001 00:00:00:00:00:06 00:00:00:00:00:03 6

----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.6
* LoadBalancer state found in state table
* 10.0.0.6 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.3
* Request sent to backend server 10.0.0.3

----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.6
* LoadBalancer state found in state table
* 10.0.0.6 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.3
* Request sent to backend server 10.0.0.3

----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.6
* LoadBalancer state found in state table
* 10.0.0.6 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.3
* Request sent to backend server 10.0.0.3

----- Handling TCP packet for port 8080 -----
* Processing Packet-In for client 10.0.0.6
* LoadBalancer state found in state table
* 10.0.0.6 ==> LoadBalancer(10.0.0.100) -----> 10.0.0.3
* Request sent to backend server 10.0.0.3

```

Total Score = _____ / 196