Extended Dijkstra's based Load Balancing for OpenFlow Network

Divya Guttikonda, Rishabh Rathore, Prachi Kelkar, Bhushan Deshmukh
Email: dguttik@ncsu.edu, rrathor@ncsu.edu, pskelkar@ncsu.edu, bdeshmu@ncsu.edu

CSC/ECE 573: Internet Protocols
Fall 2016
https://sites.google.com/a/ncsu.edu/the-extended-dijkstra-s-based-load-balanci
ng-for-openflow-network/

Contents

1. Introduction

In a network where multiple clients request for resources and there are multiple servers that can respond to these requests as well as multiple paths to reach these servers, if a suboptimal path is selected, it results in less throughput and high end to end latency leading to delayed results. To solve this important issue of always selecting the optimum path and getting best performance out of the network, we propose to achieve OpenFlow based server load balancing by Extending Dijkstra's Shortest Path Algorithm[1]. Extended Dijkstra's Algorithm, considers the node cost and the server link utilization along with the link cost to find the nearest server.

The controller running the load balancer modifies the switch flow table to forward each request to the nearest server with the link load (utilization of the link between the server and the switch) lower than the threshold value. When the link utilization overshoots the threshold, then the flow table entries are dynamically modified by the controller so that the nearest server with link utilization below threshold is selected. In this way, we also prevent congestion on the servers.

By following this mechanism, end-to-end latency between clients and server will be reduced as the request will be forwarded to the nearest server. Reducing the end-to-end latency will increase the throughput of the client as more data will be transferred in less time.

1.1 Literature Survey

- **SDN**

Software Defined Networking(SDN) is an emerging networking architecture that separates control plane from data plane enabling the network control to directly programmable and abstracts the underlying infrastructure from applications and network services. Routing, forwarding and other network intelligence is centralized in the SDN controllers which maintains a global view of the network and appears as single switch to applications and policies. SDN provides dynamic, cost-effective, manageable computing of paths.

- **Openflow**

Openflow is an open standard implementing SDN functionality through various protocols.
Openflow switch has the data path portion and clear flow tables of packet forwarding paths. Openflow supported controller manages multiple devices and performs routing algorithms. Whenever a switch does not have a path to destination, it contacts controller through openflow protocol. The router then calculates the routing path for that destination and sends that path to switches.

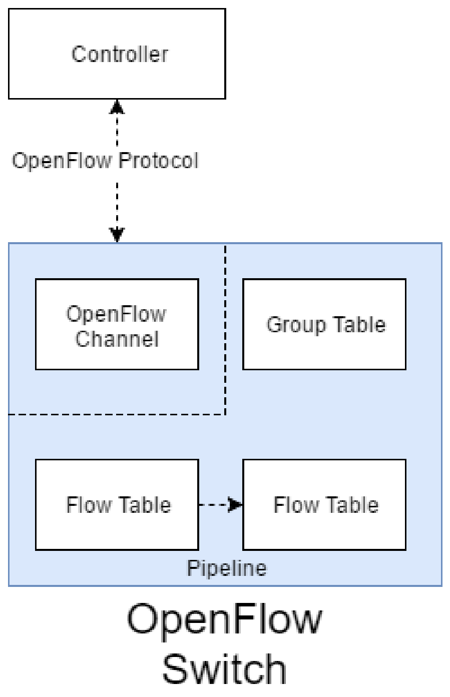


Figure 1: OpenFlow Switch

- **Ryu Controller**

Ryu Controller is an open SDN controller providing software components and APIs which help developers in creating network management and control application according to their requirement. Ryu is developed entirely in python and is available under Apache 2.0 licence. Apart from Openflow, Ryu supports NetConf and OF-config network management protocols.
RYU provides REST API (GET method) for querying information of the switches and links connected in the network. Moreover, REST API's PUT, POST and DELETE methods can be used to modify the routing information on the switches as per the rules set up on the controller.

2. System Design

2.1 Platforms for the project:

The platform we were assigned with originally was NETlabs. However, due to insufficient lab resources to create larger topologies, the project will be implemented using **GENI**[2].

OpenFlow[3] OpenFlow provides an open protocol to program the flow table in different switches and routers. It provides centralized control functions by decoupling control and data forwarding layers of routing.

Open vSwitch[4] is the open source implementation of a distributed virtual multilayer switch.

Ryu: It is an OpenFlow controller for Python based Software Defined Networking(SDN).

Ubuntu OS[6]**:** All nodes will be running on Ubuntu in Geni.

2.2 Areas for the project

Load balancing: Load balancing improves the distribution of network traffic across multiple paths and resources. Load balancing means to optimize resource use, maximize throughput, minimize end to end latency.

Routing: The process of moving a packet from the source to the destination using the best available path based on routing algorithms.

3. Major Component Decomposition

3.1 Traffic Query Unit:

The controller queries the different parameters of the links and switches such as the number of a flow's bits processed by a switch and bits passing through an link with the help of the REST (Representational State Transfer) API. REST APIs use HTTP requests to get, put, post and delete data. **ryu.app.rest_topology** provides REST APIs for retrieving the switch stats and updating the switch stats. Topology of the network is discovered from these statistics and its structure is stored in a directedgraph. The real-time data queried by TQU is passed on to the Cost Function Unit.

3.2 Cost Function Unit:

The link cost and node cost for each link and node in the topology are calculated based on real-time values supplied to it by the Traffic Query Unit. The calculations are based on the the query replies generated by TopologyDiscovery() and server_switch_monitor() function and the information is then stored on the edge on the directed graph of topology as the sum of link cost and the destination node cost.

3.3 Extended Dijkstra's Algorithm for shortest path to each server:

The proposed Extended Dijkstra's algorithm considers the node costs along with the link costs for calculation of shortest path to each server from the client. Extended Dijkstra's Algorithm calculates shortest path from client to each server and the calculated paths along with the cost are stored in path database from each host switch(switch connected to host) to each Server.

3.4 Load Balancing Engine:

All servers used in the topology are assigned the same IP alias(Virtual IP).

When a request arrives for the server IP address, the controller decides which server to route it to and the path it should take. Once a particular flow has been allocated to a server, all the packets in the flow are forwarded at line rate to that server by the datapath of one or more deployed switches.

To implement this, Load-Balancer does the following:

It chooses the appropriate server to direct requests to, based on the path database and updates the flow database to control the path taken by packets in the network, so as to minimize the response time.

Load Balancer maintains a list of servers that are running below threshold value and then select the nearest server with link utilization below threshold by referring to the cost value in path database. If all the servers are running above threshold then the server with minimum cost is selected. After selecting the server and the path, the Load Balancing engine passes the path to flow writer.

3.5 Flow Writer:

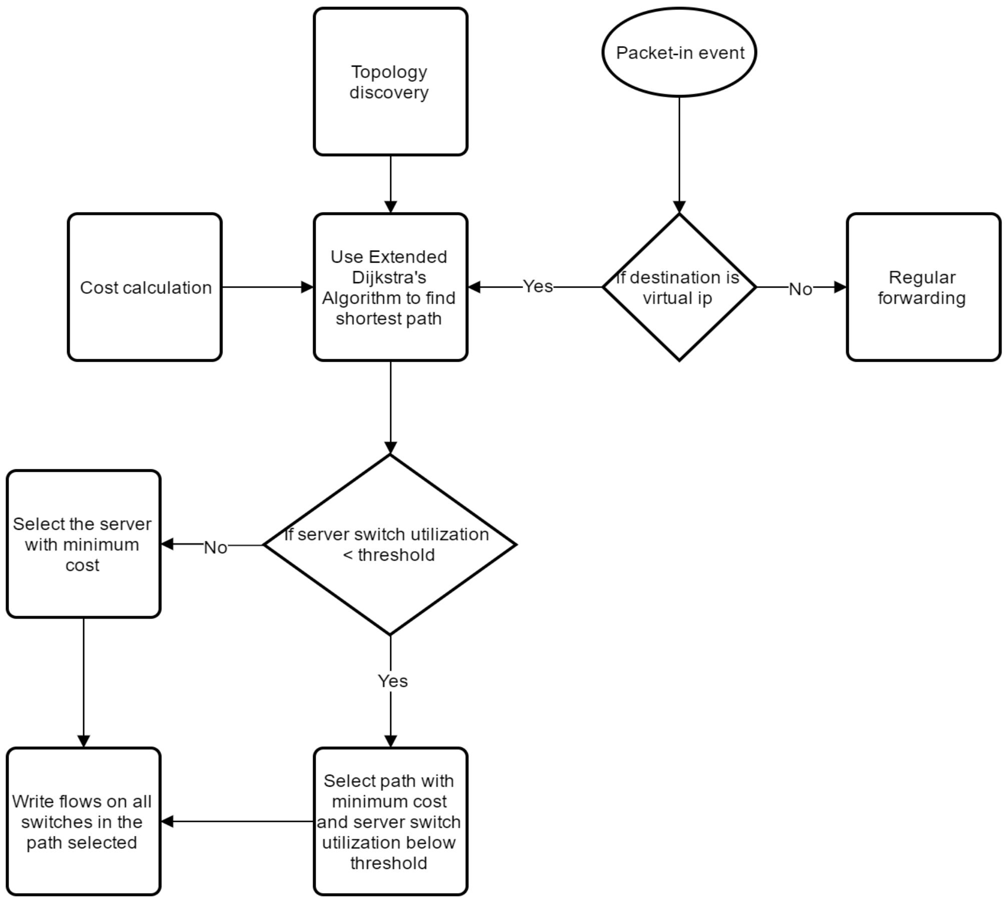
Flow writer module's purpose is to write flow entries in the switch flow table. It is called whenever there is no flow entry in the switch. It takes shortest best path as an input and rewrites flows in every switch along the path. In the ryu controller flows can be added using ofctl_rest REST application.
Flow entry consists of the source and destination hop details using following fields

- Match Field
- Priorities
- Counters
- Instructions
- Timeouts
- Cookie

We considered two server switches in our implementation, which triggers flow writer to rewrite ip addresses in them. Usually virtual ip address of server is given to hosts, so when a packet reaches server switch from host ,flow writer replaces the destination virtual ip address with real ip address of server. Conversely when a packet reaches server switch from server, it replaces the real ip address of server with virtual ip address in the flow entry.

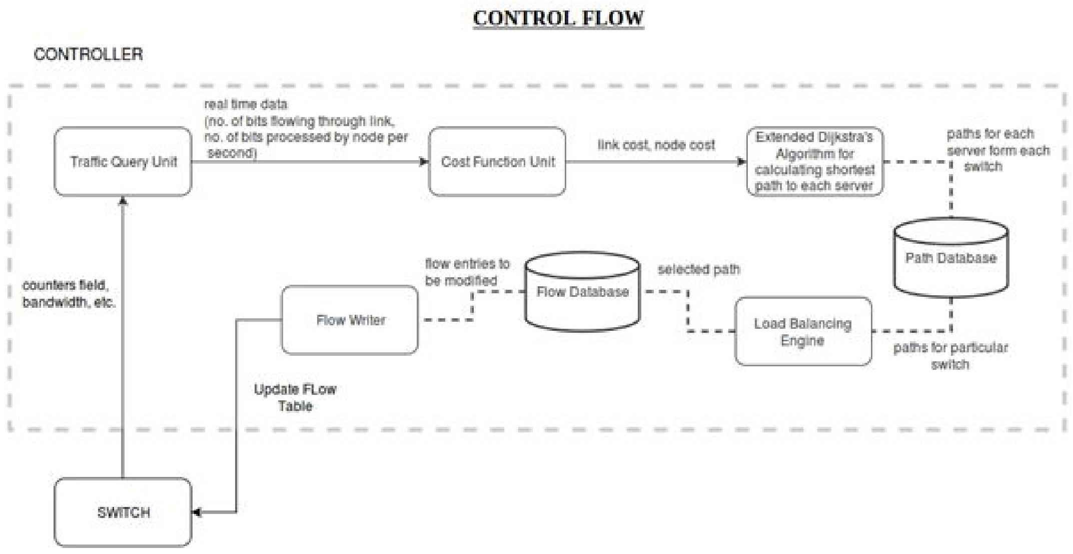
4. Design and Development Plan

4.1 High Level Design



Topology discovery
Packet-in event
Cost calculation
Use Extended Dijkstra's Algorithm to find shortest path
Yes
If destination is virtual ip
No
Regular forwarding
Select the server with minimum cost
No
If server switch utilization < threshold
Yes
Write flows on all switches in the path selected
Select path with minimum cost and server switch utilization below threshold

Figure 2: Data Flow



CONTROL FLOW
CONTROLLER
Traffic Query Unit
real time data (no. of bits flowing through link, no. of bits processed by node per second)
Cost Function Unit
link cost, node cost
Extended Dijkstra's Algorithm for calculating shortest path to each server
paths for each server form each switch
Path Database
counters field, bandwidth, etc.
flow entries to be modified
Flow Database
selected path
Flow Writer
Load Balancing Engine
paths for particular switch
Update FLow Table
SWITCH

Figure 3: Control Flow

4.2 Low Level Design

When a host requests to connect to server arrives at a OpenVSwitch, the switch asks the controller for the path to be followed by host request packets.
RYU Controller running on the Controller node uses RYU REST API for gathering the information required by Traffic Query Unit.
Node Cost and Link Cost are calculated by Cost Function for all the nodes and links in the topology.
Extended Dijkstra's Algorithm then uses the Node Costs and Link Costs to find the minimal cost to reach each server from the requested host connected switch.
Load Balancing Engine then selects the minimal cost server by considering the server utilization and selects an appropriate server for serving the requests and then the flow for reaching the selected server is pushed on all the nodes in the path by Flow Writer and thus communication between hosts and servers is established.

4.2.1 Extended Dijkstra's Algorithm

Extended Dijkstra's Algorithm is used to calculate minimum cost path considering node cost along with the link cost for a client to connect to a server.

Link Cost(LC) is the ratio of number of bits passing through the link to the link bandwidth.

$$Link\ Cost(LC) = \frac{\Sigma\ number\ of\ bits\ passing\ through\ the\ link}{Link\ Bandwidth}$$

Node Cost(NC) is the ratio of number of bits processed by node per second to the number of bits that node can process per second which is node Capability.

$$Node\ Cost\ (NC) = \frac{\Sigma\ number\ of\ bits\ processed\ by\ node}{Number\ of\ bits\ node\ can\ process\ per\ second}$$

Once the shortest path is calculated by the controller, it will query the link between server-switch(switch connected directly to server) and server using traffic query unit to check the link load.

Link Load is the ratio of current traffic between server-switch and server link to maximum bandwidth of server-switch and server link.

$$Server\ Link\ Load = \frac{\Sigma\ current\ traffic\ between\ server{-}switch\ and\ server\ link}{Link\ Bandwidth}$$

Extended Dijkstra's Algorithm:

The extended Dijkstra's Algorithm is similar to original Dijkstra's Algorithm. The difference is that the extended Algorithm also takes into consideration node cost along with the link cost.

For a graph G(V, E) with following parameters:
s : source node
V : set of nodes
E : set of links
lw : link weight
nw : node weight
d[u] : distance of current shortest path from source s to destination u
p[u] : previous node to u

Input to Dijkstra's Algorithm : G=(V,E), lw, nw, s

Extended Dijkstra's Algorithm pseudocode:

```
1. d[s]←0; d[u]←∞, for each u≠s, u∈V
2. Insert u with key d[u] into the priority queue Q, for each u∈V
3. while(Q≠Φ)
4. u←Extract-Min(Q)
5. for each v adjacent to u
6. ifd[v]>d[u]+lw[u,v]+nw[u] then
7. d[v]←d[u]+lw[u,v]+nw[u]
8. p[v]←d[u]
```

The Algorithm is similar to original Dijkstra's Algorithm with the modification of adding node cost of every node along the path in step 6 and step 7.
The Algorithm returns Shortest path from source s to destination u

4.2.2 Load-Balancing Algorithm:

When a new request comes to the controller, Extended Dijkstra's Algorithm calculates minimum cost path from client to every Server-Switch (Switch that is directly connected to Server and only one Server is connected to one Server-Switch).
Depending on the server utilization and the cost to reach the server, the Load Balancing Algorithm selects the minimal path for serving the client request taking into consideration the threshold prespecified for servers.

sw_{src} : source switch
S_{dst} : set of Servers
Θ : prespecified threshold of Servers
kl_i : link load of link connecting server S_i and server switch sw_i <S_i, sw_i>

```
1. P←eDijkstra($sw_{src}$ , $S_{dst}$ )
```

```
2. for every $p_i \in P$
3. if $p_i$.server.kl > $\Theta$ then move $p_i$ from P to Q
4. if P=$\Phi$ then
5.        s $\leftarrow$ min(P).server
6. else
7.        s $\leftarrow$ min(Q).server
8. return s
```

In line 1, list P is formed which contains cost of each server from source switch sw_{src}.

In line 3, p_i.server.kl is the current utilization of link connecting server and server switch. If the current link utilization is more than threshold Θ, then the server is moved from list P to list Q.

We repeat the above step for all the entries in list P.
After this step, list P will contain server list who are running above threshold and list Q will contain servers that are running above threshold value.

After complete scanning of list P, we select the server with minimum cost from list P for serving the client request.
But if list P is empty, meaning that all the servers are running above threshold Θ, then a server with minimum cost from list Q is selected for serving the client request.

Thus, the Load Balancing Algorithm selects the minimum cost server which is running below threshold for serving the client request and if all the servers are running above threshold then the server with minimum cost is selected for serving the client request.

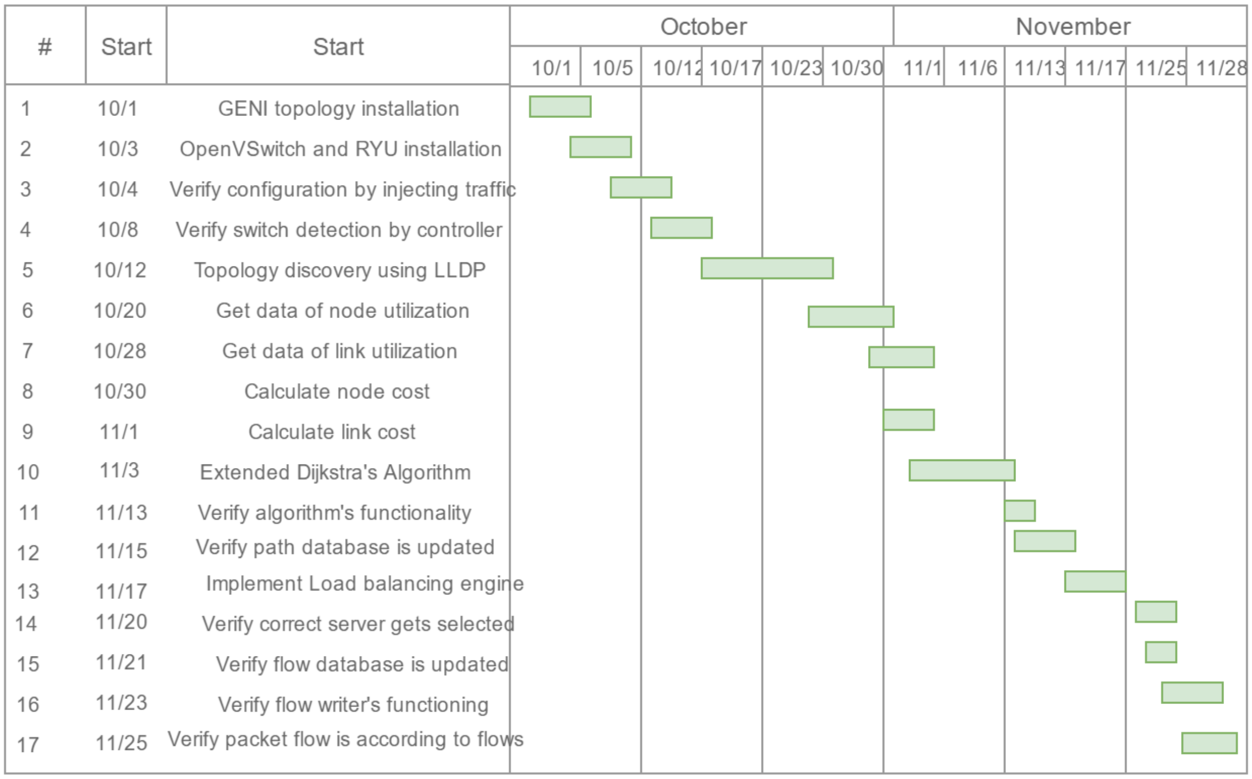
4.3 Team Composition and per member responsibilities

Task	**Accountable Person**
Test Environment	
Provision switch nodes, host nodes and 1 controller using GENI	Bhushan
Install OpenVSwitch on switch nodes, RYU controller on controller node	Bhushan
Verify that basic switch and controller configuration is correct by injecting traffic to check reachability of hosts	Rishabh
Traffic Query Unit	
Verify that switches are detected by	Rishabh

Controller	
Implement topology discovery using LLDP	Bhushan
Get data of node utilization	Rishabh
Get data of link utilization	Bhushan
Cost Function	
Calculate Node Cost	Prachi
Calculate Link Cost	Prachi
Extended Dijkstra’s Algorithm	
Implement Extended Dijkstra’s Algorithm	Bhushan
Verify that Extended Algorithm calculates cost for each server-switch from the source switch	Bhushan
Verify that Path Database is updated successfully	Bhushan
Load-Balancing Engine	
Implement Load Balancing Engine Algorithm	Rishabh
Verify that Load Balancing Engine selects appropriate server for the request	Rishabh
Verify that flow database is updated successfully	Rishabh
FLow Writer	
Verify that Flow Writer writes appropriate flow on the switches	Divya
Verify that packets follow the path written by Flow Writer	Divya

Table 1: Per-member distribution

4.4 Development Phases



#	Start	Start
1	10/1	GENI topology installation
2	10/3	OpenVSwitch and RYU installation
3	10/4	Verify configuration by injecting traffic
4	10/8	Verify switch detection by controller
5	10/12	Topology discovery using LLDP
6	10/20	Get data of node utilization
7	10/28	Get data of link utilization
8	10/30	Calculate node cost
9	11/1	Calculate link cost
10	11/3	Extended Dijkstra's Algorithm
11	11/13	Verify algorithm's functionality
12	11/15	Verify path database is updated
13	11/17	Implement Load balancing engine
14	11/20	Verify correct server gets selected
15	11/21	Verify flow database is updated
16	11/23	Verify flow writer's functioning
17	11/25	Verify packet flow is according to flows

Figure 4: Timeline

4.5 Timeline

The project timeline is divided into two phases. Figure 4 shows the timeline.
Phase 1: 10/01/2016 to 11/13/2016
Phase 2: 11/13/2016 to 11/28/2016

4.6 Per Member Learning Objectives

Rishabh Rathore ● To learn about real time network data querying and network cost calculation.
Bhushan Deshmukh ● To learn about shortest path algorithms and load balancing algorithms on large networks.
Prachi Kelkar

● To learn about Openflow and network programming, routing algorithms and networking programming.
Divya Guttikonda ● To learn about Openflow and SDN technology, routing algorithms,storing data in openflow database and networking programming.

Table 2: Per Member learning objectives

5. Implementation

5.1 Traffic Query Unit

- We are setting up the below topology in the EXOGENI environment, where we have 11 Hosts,9 openflow switches, 2 server switches and 2 server.
- We have written "Topology Discovery() " program which utilizes the rest_topology application and information like properties of switches, properties of links , their bandwidth and status in the network and stores entire details and the network structure in a directedgraph.

5.1.1 Topology discovery:

1. Whenever a packet comes to controller, controller discovers entire network topology using ryu/ryu/app/rest_topology.py application running on localhost.

```
# get all the switches
http://localhost:8080/v1.0/topology/switches
 Returns complete information of every switch on the network.
```

Example information of one switch:

```
[{"ports": [{"hw_addr": "fa:16:3e:00:3f:b1", "name": "eth1", "port_no":
"00000001", "dpid": "00008af704a6d842"}, {"hw_addr": "fa:16:3e:00:25:8f",
"name": "eth2", "port_no": "00000002", "dpid": "00008af704a6d842"},
{"hw_addr": "fa:16:3e:00:24:b5", "name": "eth3", "port_no": "00000003",
"dpid": "00008af704a6d842"}], "dpid": "00008af704a6d842"}
```

```
  # get the switch
 http://localhost:8080/v1.0/topology/switches<switch_dpid>
 Returns information of a switch with given dpid(data path id)

  # get all the links
 http://localhost:8080/v1.0/topology/links
 Returns information regarding links connected in the network

 # get the links of a switch
  http://localhost:8080/v1.0/topology/links<switch_dpid>
 Returns information regarding links connected to a switch with given id
```

Example output

```
[{"src": {"hw_addr": "fa:16:3e:00:27:8d", "name": "eth2", "port_no":
"00000002", "dpid": "0000aa66e817ab48"}, "dst": {"hw_addr":
"fa:16:3e:00:2d:c6", "name": "eth3", "port_no": "00000003", "dpid":
"0000b26a0843ce45"}}
```

2. A Directed Graph G is created using Networkx package, all the above topology information is stored in the graph with the nodes as switches and the links between a pair of switches as the edges on the graph.

```
# creating directedgraph using networkx
     G = nx.DiGraph();
     # adding switch to directedgraph
     self.G.add_node(dpid1_dec, weight=1, Bcount=val, DiffBcount=0)
     # adding links to directedgraph
self.G.add_edge(src_dpid_dec,      dst_dpid_dec,       src_port=src_port,
dst_port=dst_port, weight =1)
```

5.1.2 Traffic Querying

1. Traffic Query unit queries all the switches, links and Server-Switches (Server connected Switches) for the real time traffic on them
2. To implement this 2 threads are created
    a. Thread 1 : running TopologyDiscovery() module which discovers all the links and nodes in the topology and calculates the node statistics(number of Bytes received by node per second) and link statistics(number of Bytes transferred by link per second). This information is queried every second for the entire topology.
    b. Thread 2 : running server_switch_monitor() module which queries the number of Bytes transferred by Server and Server-Switch link every second which is then further used to check if the Server Switch is running above threshold value
3. Using `http://localhost:8080/stats/port<switch_dpid>` switch status is retrieved, which consists of number of packets processed, dropped by the switch, number of bytes involved in the process etc.
4. Link bandwidth calculation is done by adding the number of Bytes transferred and received by the destination ports connecting the switch
5. For the Server Switch, the number of rx_bytes and tx_bytes are calculated for the Server-Switch port connecting the Server-Switch and the Server and the difference is stored in the list ServerInfolist mapping the Server-Switch DPID to port_byte_diff storing the number of Bytes passing through the link per second.

5.2 Cost Function Unit

Cost Function Unit is responsible for calculating node and link costs of each and every switch and connecting links respectively.

The Cost Function Unit takes in the data generated by TopologyDicsovery() module for the calculation of link cost and node cost.

Link Cost is Calculated as the ratio of number of Bytes passing through the link per second to link Bandwidth

Node Cost is calculated as the ratio of number of Bytes received by Node per second to the node capacity

```
dst_node_weight = (float(diffBcount_dst)/float(self.NODE_CAPACITY))
```

diffBcount_dst is the number of Bytes received by node from all the ports per second
We started iperf to contact the server and found out that the maximum value of number of Bytes received by the Switch Connected to maximum bandwidth node and to bring the node cost and link cost to comparable figures the NODE_CAPACITY value selected as 1000000

The node is added with following parameters:
self.G.add_node(dpid1_dec, weight=1, Bcount=val, DiffBcount=diffBcount, timestamp=ts, timestamp_diff=timestamp_diff1)
dpid1_dec : DPID of node(switch)
weight : initial weight value of node
Bcount : Bytes count of total number of Bytes received on the switch per secong
DiffBcount : Difference of number of bytes received on the switch(node) [er second
timestamp : timestamp of storing the information
Timestamp_diff : difference of timestamp

The sum of link weight and destination node weight is set as the total link weight and is added to the link.
Link is added as following:

```
self.G.add_edge(src_dpid_dec,dst_dpid_dec,src_port=src_port, dst_port=dst_port,
Bcount_dst=bcount_dst,          DiffBcount=diffBcount,          weight=weight1,
bandwidth=bandwidth1, timestamp=ts)
```

Where
src_dpid_dec : source dpid of link
dst_dpid_dec : destination dpid of link
src_port : source port of link
dst_port : destination port of the link
Bcount_dst : number of bytes received by destination node per second
diffBcount : number of bytes received on link per second
weight : link weight (sum of link + destination node weight)
bandwidth : link bandwidth
timestamp : timestamp of data storage

The link weight and destination node weight are calculated and stored in the link as weight1

```
weight1 = float(link_weight + dst_node_weight)
```

weight1: weight stored in the link as sum of link weight and destination node weight

link_weight : weight of link
dst_node_weight : destination node weight

5.3 Shortest path Algorithm

`The Shortest path is calculated using dijkstra_shortest_path function` available on networkx Graph.
The sum of link cost and destination node cost is stored as link cost in the weight field of the link as per the Extended Dijkstra's Algorithm which takes into consideration the node cost and link cost for path selection.

The Shortest Path is calculated as:

```
nx.dijkstra_path_length(self.G, origin, destination, 'weight')
Where:
```

self.G : Graph containing the information of all the links and nodes
origin : origin from where we want to calculate the Shortest Path (it will be server switch asking for the path to serve host packets to server)
destination : the list of all Server-Switches
weight : the link parameter containing sum of link cost and destination node cost
The Shortest Path Algorithm calculates the Shortest Path to reach destination from origin based on the weight parameter

Here, the Shortest Path to reach each destination(Server Switch) from Host Connected Switch is calculated.

The output of Shortest Path Algorithm is stored in the following lists
path_len LIST #stores path length for each PATH
path_all LIST #stores path to reach each SSW

5.4 Load Balancer Engine

Two LISTS P and Q are created to parse over all the Shortest paths in LIST for separation of paths into paths which have Server Switch running below threshold and Server Switch running above threshold.

```
P = []
P_cost = []
Q = []
Q_cost = []
```

The LISTS are initialised as empty.
All the Shortest Paths to reach all the Server-Switches in list path_all are parsed and if some Server Switch is running above threshold, then
that path is added in list Q and cost of reaching that path is added in list Q_cost

else,
The path is added in list P and cost of reaching that path is added in list P_cost

After parsing over all the paths in LIST path_all[], we check

if,
LIST P is not empty, then the path with minimum cost in list P is selected

else,
select minimum cost path from list Q

For selection of threshold value, we tried to overload the server switch by running iperf and tried to bring the Server-Switch to bottleneck value.
We observed that maximum number of packets flowing through Server-Switch and Server link was maximum 137000 Bytes based on the topology nodes and the links used in the topology.
So the threshold value was set to 100000 Bytes per second for Server-Switch and Server link which is 73% of maximum value.
So the threshold value selected is 73%.
As long as the link of Server is running below threshold that particular server is selected based on the path cost for the nodes and once the server starts running above threshold value(73%) all the further host requests are redirected to the other server unless it's utilization goes above threshold.

The selected path is stored in LIST selpath[] which is further given to flow writer for writing the flow and serving the host request.

5.5 Flow writer

A packet can be forwarded in a network by following one of the following strategies.

- If the switch has flow entry for a destination given in packet, then packet is forwarded to next hop in normal way.
- If the switch does not have a flow entry for the destination given in packet, then packets are sent to the controller through Packet-In function. The controller retrieves the network topology, calculates shortest path for that destination and adds that flow entry in the switch flow table
- Once the switch has flow entries, it forwards the packets in usual manner.

Flow addition is done through following steps:

1. We have considered following parameters to add flows
   `In_port` :: input port of the packet,
   `Ipv4_src`: host ipv4 address ,
   `Virtual_ip` : virtual ip of the servers(192.168.1.10),
   out_port:outgoing port to reach the server,
   server_ip: ip of the server selected by the best path algorithm,
   server_mac:mac of the server selected by the best path algorithm ,
   `PRIORITY1`: openflow priority value of the pushed flow

2. Best path is taken as input by the flow writer, retrieves the input port(IN_PORT) and output port(OUT_PORT) of the switches along the path.

```
if count != 0: #we are at first switch in the path
IN_PORT =int(self.G[selpath[i-1]][selpath[i]]['src_port'])

if count != length-1: #we are not at last switch in the  path
OUT_PORT =int(self.G[selpath[i]][selpath[i+1]]['dst_port'])
```

Where selpath is the selected path, count defines the position of switch in the path.

3. These details along with the other flow parameters like source and destination addresses(IP),DPID etc are passed to methods path_switch_add_flows() or server_switch_add_flows() to add flows in the switches along the path or in the server switch respectively..

```
self.path_switch_add_flows(DPID, IN_PORT,  IPV4_SRC,  IPV4_DST,  OUT_PORT,
server_ip, self.PRIORITY1)

if count == length-1: #we are at last switch in the path
OUT_PORT = 1
server_mac = self.ServerInfolist[DPID]['server_mac']
self.server_switch_add_flows(DPID, IN_PORT, IPV4_SRC, IPV4_DST, OUT_PORT,
server_ip, server_mac, self.PRIORITY1)
print "Flow : ",DPID, IN_PORT, OUT_PORT
self.PRIORITY1 = self.PRIORITY1 + 1
count=count+1
```

4. path_switch_add_flows(in_port, ipv4_src, virtual_ip, out_port, server_ip, server_mac, PRIORITY1) takes the switch dpid and flow parameters to install 2 flows, one each for incoming and outgoing packets for a specific host and the server selected.

5. server_switch_add_flows(self, dpid, in_port, ipv4_src, virtual_ip, out_port, server_ip, server_mac, PRIORITY1) takes the server-switch dpid and flow parameters to install 2 flows, one each for incoming and outgoing packets for a specific host and the server selected.

6. For the packets from host to server, the installed flow rewrites the virtual ip and virtual mac with the real ip and mac of the server selected and forwards on the port connected to the server.

7. For the packets from server to host, the installed flow rewrites the real ip and real mac in the source of packets with the virtual ip and mac and forwards on the port connected to the host end.

```
server_switch_to_server[ACTIONS][1][FIELD] = "ipv4_src"
```

```
server_switch_to_server[ACTIONS][1][VALUE] = virtual_ip
```

8. Flow writer module also pushes flows to instruct switches along the path forward arp request and replies on the appropriate port to avoid flooding of packets in case of loops in the network and avoids broadcast storm.

```
arp_request= {
'dpid': dpid,
'cookie': 1,
'cookie_mask': 1,
'table_id': 0,
'idle_timeout': 3000,
'hard_timeout': 3000,
'priority': PRIORITY1,
'Flags':1,
'Match':{
      'In_port':out_port,
      'Eth_type':0x0806,
      'Arp_spa':server_ip,
      'Arp_tpa':ipv4_src,
      'Arp_op':'1'
       },
'actions':[{'type':"OUTPUT",'port':in_port}]}

arp_reply= {
'dpid': dpid,
'cookie': 1,
'cookie_mask': 1,
'table_id': 0,
'idle_timeout': 3000,
'hard_timeout': 3000,
'priority': PRIORITY1,
'Flags':1,
'Match':{
      'In_port':in_port,
      'Eth_type':0x0806,
      'Arp_spa':ipv4_src,
      'Arp_tpa':server_ip,
      'arp_op':'2'},
 'actions':[{'type':"OUTPUT",'port':out_port}]}
```

9. Our application sends a POST request to "http://localhost:8080/stats/flowentry/add" in order to add flows on the switches along the shortest path selected by the algorithm, where ryu.app.ofctl_rest application is running on localhost . We have implemented this using "json" requests.

```
r= requests.post(url_add,data=json.dumps(server_switch_to_server))
```

10. Example Flows:

Path_switch:

```
cookie=0x1,   duration=131.632s,   table=0,   n_packets=3,   n_bytes=168,
idle_timeout=3000,            hard_timeout=3000,            send_flow_rem
priority=22222,arp,in_port=3,arp_spa=192.168.1.11,arp_tpa=192.168.1.106,ar
p_op=1 actions=output:2

cookie=0x1,   duration=131.628s,   table=0,   n_packets=3,   n_bytes=168,
idle_timeout=3000,            hard_timeout=3000,            send_flow_rem
priority=22222,arp,in_port=2,arp_spa=192.168.1.106,arp_tpa=192.168.1.11,ar
p_op=2 actions=output:3

cookie=0x1,  duration=131.639s,  table=0,  n_packets=129,  n_bytes=12642,
idle_timeout=50000,           hard_timeout=50000,           send_flow_rem
priority=22222,ip,in_port=2,nw_src=192.168.1.106,nw_dst=192.168.1.10
actions=output:3

cookie=0x1,  duration=131.636s,  table=0,  n_packets=129,  n_bytes=12642,
idle_timeout=50000,           hard_timeout=50000,           send_flow_rem
priority=22222,ip,in_port=3,nw_src=192.168.1.10,nw_dst=192.168.1.106
actions=output:2
```

Server_switch: (rewrites virual ip and mac withreal and vice-versa)

```
cookie=0x1,   duration=14.472s,   table=0,   n_packets=1,   n_bytes=56,
idle_timeout=3000,            hard_timeout=3000,            send_flow_rem
priority=11113,arp,in_port=1,arp_spa=192.168.1.11,arp_tpa=192.168.1.101,ar
p_op=1 actions=output:2

cookie=0x1,   duration=14.468s,   table=0,   n_packets=1,   n_bytes=56,
idle_timeout=3000,            hard_timeout=3000,            send_flow_rem
priority=11113,arp,in_port=2,arp_spa=192.168.1.101,arp_tpa=192.168.1.11,ar
p_op=2 actions=output:1

cookie=0x1,   duration=14.465s,   table=0,   n_packets=5,   n_bytes=490,
idle_timeout=50000,           hard_timeout=50000,           send_flow_rem
priority=11113,ip,in_port=1,nw_src=192.168.1.11,nw_dst=192.168.1.101
actions=set_field:aa:aa:aa:aa:aa:aa->eth_src,set_field:192.168.1.10->ip_sr
c,output:2

cookie=0x1,   duration=14.476s,   table=0,   n_packets=5,   n_bytes=490,
idle_timeout=50000,           hard_timeout=50000,           send_flow_rem
priority=11113,ip,in_port=2,nw_src=192.168.1.101,nw_dst=192.168.1.10
actions=set_field:fa:16:3e:00:6c:a8->eth_dst,set_field:192.168.1.11->ip_ds
t,output:1
```

6. Verification and Validation Plan

6.1 Test Application

- The testing of the controller code is done using a series of unit test cases.
- Also for the purpose of testing the code, the values of different links and the entire topology will be hard coded and the algorithm will be tested on those.

Rishabh Rathore ● Design and implement test network in GENI
Bhushan Deshmukh ● Generate test traffic for network testing
Prachi Kelkar ● Base case testing with round robin and unit weighted dijkstra's algorithm
Divya Guttikonda ● End to end network testing with load-balancing and results comparison

Table 3: Per-member distribution

6.2 Test Environment

The topology we will use for the demo is shown below.

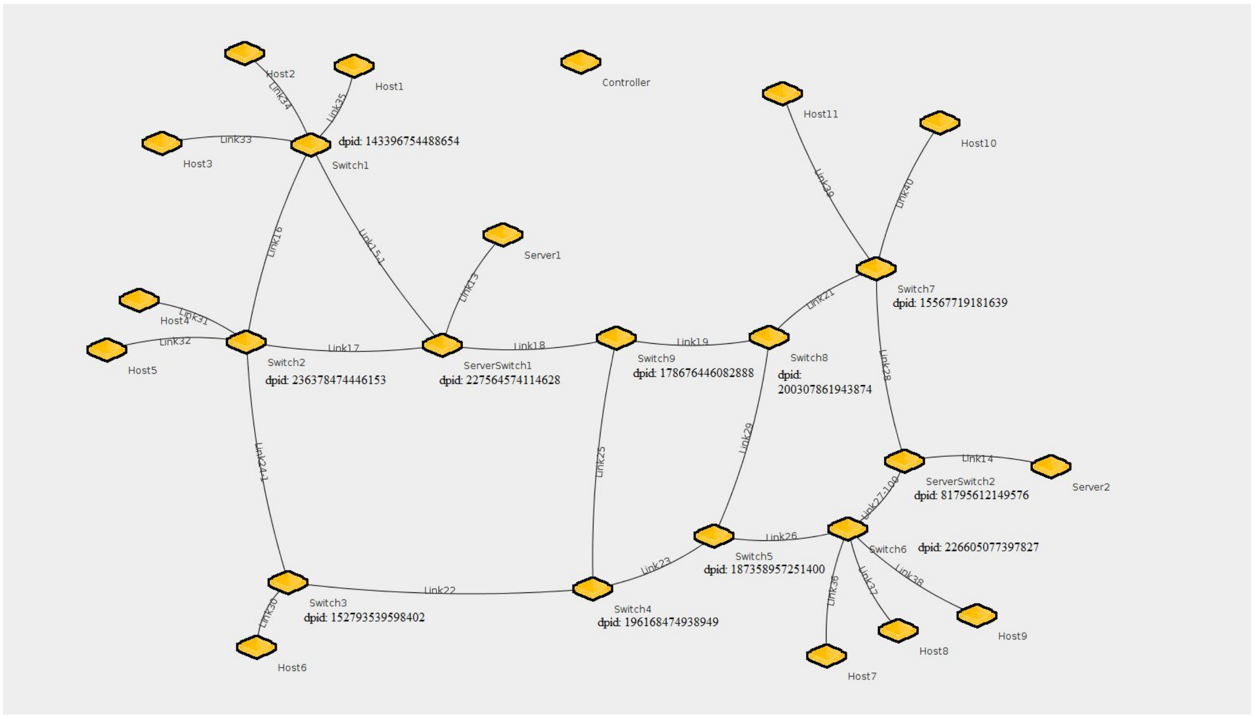


Figure5: Topology

There are 11 Hosts, 9 path ovs switches, 2 server ovs switches and 2 servers and 1 Controller. The specifications for all these systems are given below:

Network Component	Resource Type	Flukes Name	RSpec Name	Cores	RAM	Disk(s)	OS
Hosts, Openflow Switches	VM	XOSmall	xo.small	1	1GB	10GB	Ubuntu 14.04
Servers, Server Switches	VM	XOMedium	xo.medium	1	3GB	25GB	Ubuntu 14.04
Controller	VM	XOLarge	xo.large	2	6GB	50GB	Ubuntu 14.04

Table4: Topology Specifications

Link “27-100” is a 100kbps link.

Links “15-1” and “24-1” are 1 Mbps links.

All other links are 10Mbps.

Component Name	Datapath ID	IP Address
ServerSwitch1	227564574114628	128.194.6.183
ServerSwitch2	81795612149576	128.194.6.202
Switch1	143396754488654	128.194.6.200
Switch2	236378474446153	128.194.6.198
Switch3	152793539598402	128.194.6.194
Switch4	196168474938949	128.194.6.187
Switch5	187358957251400	128.194.6.191
Switch6	226605077397827	128.194.6.186
Switch7	15567719181639	128.194.6.201
Switch8	200307861943874	128.194.6.190
Switch9	178676446082888	128.194.6.188

Table5: Component Specifications

Virtual IP for servers: 192.168.1.10

Virtual MAC for servers: aa:aa:aa:aa:aa:aa

Network Components	**IP Address**
Host1	192.168.1.101
Host2	192.168.1.102
Host3	192.168.1.103
Host4	192.168.1.104
Host5	192.168.1.105
Host6	192.168.1.106
Host7	192.168.1.107
Host8	192.168.1.108
Host9	192.168.1.109
Host10	192.168.1.110
Host11	192.168.1.111
Server1	192.168.1.11
Server2	192.168.1.12

Table6: Component Specifications

OpenVSwitch Configuration Steps:

Do not give any IP address on any interface of the GENI system and run the following commands to make it a OpenVSwitch:

```
>>sudo apt-get update
>>sudo apt-get install openvswitch-switch
>>sudo apt-get install openvswitch-common
>>sudo apt-get install openvswitch-controller
```

Now create a bridge and add all the ports to the switch:

```
>>ovs-vsctl add-br <bridge_name>
>>ovs-vsctl add-port <bridge_name> <port_no>
```

Set controller for the switch using the following command

```
>>ovs-vsctl set-controller <bridge_name> tcp:<controller_ip>:6633
```

Controller (RYU) Configuration Steps:

Controller will contact the switches through management IP
```
>>apt-get install python-dev
>>pip install -U pip setuptools
>>hash -r
>>time sudo apt-get install python-eventlet python-routes python-webob >>python-paramiko
>>pip install ryu
>>pip install tinyrpc
>>git clone git://github.com/osrg/ryu.git
>>ryu-manager <program_file>
```

Controller starts running and it can start serving the requests from switches coming on the management IP of GENI.

6.3 Test Plan

- Base case scenario used will be round robin and unit-weighted dijkstra's algorithm to select a server to which the traffic will be directed for a particular host connected to the switches.
- End to end network latency will be measured using ping tool by sending 300 packets of 10000 bytes each for 30 seconds.
- Iperf like utility will used to generate TCP data stream from clients to servers and and calculate the average throughput on each client.

6.3.1 Test cases

6.3.1.1 Unit Testing

1. **Verify that switches are detected by Controller**
   Aim: Verify that OVS connects to controller after doing the required configuration on the controller
   Procedure:
    1. Verify OVS is connected to controller using the command "ovs-vsctl show br0 -O OpenFlow13"
    2. Start RYU controller module on the controller node using the command "ryu-manager"
    3. Run **ryu.app.rest_topology** application on local host.
    4. Verify that controller is able to detect nodes connected to it using the command
       ```
       curl -X GET http://localhost::8080/v1.0/topology/switches
       ```

   Success Criteria: Controller is seen as connected in the switch and the REST API GET request shows the switches connected to the Controller

```
Nodes :

ServerSwitch1                dpid:227564574114628
ServerSwitch2                dpid:81795612149576

Switch1                      dpid:143396754488654
Switch2                      dpid:236378474446153
Switch3                      dpid:152793539598402
Switch4                      dpid:196168474938949
Switch5                      dpid:187358957251400
Switch6                      dpid:226605077397827
Switch7                      dpid:15567719181639
Switch8                      dpid:200307861943874
Switch9                      dpid:178676446082888
```

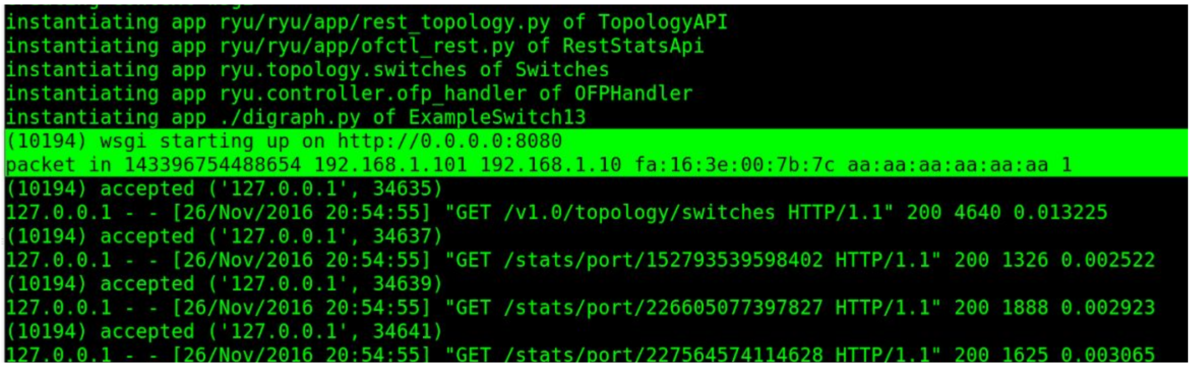
2. **Verify that links are detected by Controller**
   Aim: Verify that OVS connects to controller after doing the required configuration on the controller
   Procedure:
   1. Verify OVS switches are connected to controller using the command "ovs-vsctl show br0 -O OpenFlow13"
   2. Start RYU controller module on the controller node using the command "ryu-manager"
   3. Run **ryu.app.rest_topology** application on local host.
   4. Verify that controller is able to detect links connected to using the command `curl -X GET http://localhost::8080/v1.0/topology/links`

```
Links :

1.Link between switch 3 and switch 2
 (152793539598402, 236378474446153)
2.Link between switch 3 and switch 4
(152793539598402, 196168474938949)
3.Link between switch 6 and server switch 2
 (226605077397827, 81795612149576)
4.Link between switch 6 and switch 5
 (226605077397827, 187358957251400)
5.Link between server switch 1 and switch 9
 (227564574114628, 178676446082888)
6.Link between server switch 1 and switch 2
 (227564574114628, 236378474446153)
7.Link between server switch 1 and switch 1
 (227564574114628, 143396754488654)
8.Link between switch 4 and switch 5
(196168474938949, 187358957251400)
9.Link between switch 4 and switch 9
 (196168474938949, 178676446082888)
10.Link between switch 4 and switch 3
 (196168474938949, 152793539598402)
11.Link between switch 7 and server switch 2
 (15567719181639, 81795612149576)
```

```
12.Link between switch 7 and switch 8
 (15567719181639, 200307861943874)
13.Link between server switch 2 and switch 6
 (81795612149576, 226605077397827)
14.Link between server switch 2 and switch 7
 (81795612149576, 15567719181639)
15.Link between switch 2 and switch 3
 (236378474446153, 152793539598402)
16.Link between switch 2 and server switch 1
 (236378474446153, 227564574114628)
17.Link between switch 2 and switch 1
 (236378474446153, 143396754488654)
18.Link between switch 8 and switch 5
 (200307861943874, 187358957251400)
19.Link between switch 8 and switch 9
 (200307861943874, 178676446082888)
20.Link between switch 8 and switch 7
 (200307861943874, 15567719181639)
21.Link between switch 1 and switch 2
 (143396754488654, 236378474446153)
22.Link between switch 1 and server switch 1
 (143396754488654, 227564574114628)
23.Link between switch 9 and switch 8
 (178676446082888, 200307861943874)
24.Link between switch 9 and switch 1
 (178676446082888, 227564574114628)
25.Link between switch 9 and switch 4
 (178676446082888, 196168474938949)
26.Link between switch 5 and switch 8
 (187358957251400, 200307861943874)
27.Link between switch 5 and switch 6
 (187358957251400, 226605077397827)
28.Link between switch 5 and switch 4
 (187358957251400, 196168474938949)
```

3. **Verify incoming packets handling**
   Aim: Verify controller detects and handles incoming packets
   Procedure:

   1. Verify OVS is connected to controller using the command ”`ovs-vsctl show br0 -O OpenFlow13`”
   2. Start RYU controller module on the controller node using the command “`ryu-manager`”
   3. Run**ryu.app.rest_topology ,ryu.app.ofctl_rest, ryu.controller.ofp_handler** applications which will call Packet-In handler for each incoming packet at the controller.
   4. Ping server virtual IP (192.168.1.10) from Host1 and observe controller log.



```
instantiating app ryu/ryu/app/rest_topology.py of TopologyAPI
instantiating app ryu/ryu/app/ofctl_rest.py of RestStatsApi
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ./digraph.py of ExampleSwitch13
(10194) wsgi starting up on http://0.0.0.0:8080
packet in 143396754488654 192.168.1.101 192.168.1.10 fa:16:3e:00:7b:7c aa:aa:aa:aa:aa:aa 1
(10194) accepted ('127.0.0.1', 34635)
127.0.0.1 - - [26/Nov/2016 20:54:55] "GET /v1.0/topology/switches HTTP/1.1" 200 4640 0.013225
(10194) accepted ('127.0.0.1', 34637)
127.0.0.1 - - [26/Nov/2016 20:54:55] "GET /stats/port/152793539598402 HTTP/1.1" 200 1326 0.002522
(10194) accepted ('127.0.0.1', 34639)
127.0.0.1 - - [26/Nov/2016 20:54:55] "GET /stats/port/226605077397827 HTTP/1.1" 200 1888 0.002923
(10194) accepted ('127.0.0.1', 34641)
127.0.0.1 - - [26/Nov/2016 20:54:55] "GET /stats/port/227564574114628 HTTP/1.1" 200 1625 0.003065
```

Success Criteria: Packet handler log is visible on controller node.

4. Verify Topology discovery

Aim: Verify that controller gets the complete information of the topology.

Procedure:

1. Verify OVS switches are connected to controller using the command "`ovs-vsctl show br0 -O OpenFlow13`"
2. Start RYU controller module on the controller node using the command "`ryu-manager`"
3. Run **ryu.app.ofctl_rest** application and the Topology Discovery module

Output :

```
Nodes :

ServerSwitch1                  dpid:227564574114628
ServerSwitch2                  dpid:81795612149576

Switch1                        dpid:143396754488654
Switch2                        dpid:236378474446153
Switch3                        dpid:152793539598402
Switch4                        dpid:196168474938949
Switch5                        dpid:187358957251400
Switch6                        dpid:226605077397827
Switch7                        dpid:15567719181639
Switch8                        dpid:200307861943874
Switch9                        dpid:178676446082888
```

`Links :`

```
1.Link between switch 3 and switch 2
  (152793539598402, 236378474446153, {'src_port': '00000003', 'dst_port':
'00000003'}),
2.Link between switch 3 and switch 4
(152793539598402, 196168474938949, {'src_port': '00000001', 'dst_port':
'00000001'}),
3.Link between switch 6 and server switch 2
  (226605077397827, 81795612149576, {'src_port': '00000001', 'dst_port':
'00000003'})
4.Link between switch 6 and switch 5
```

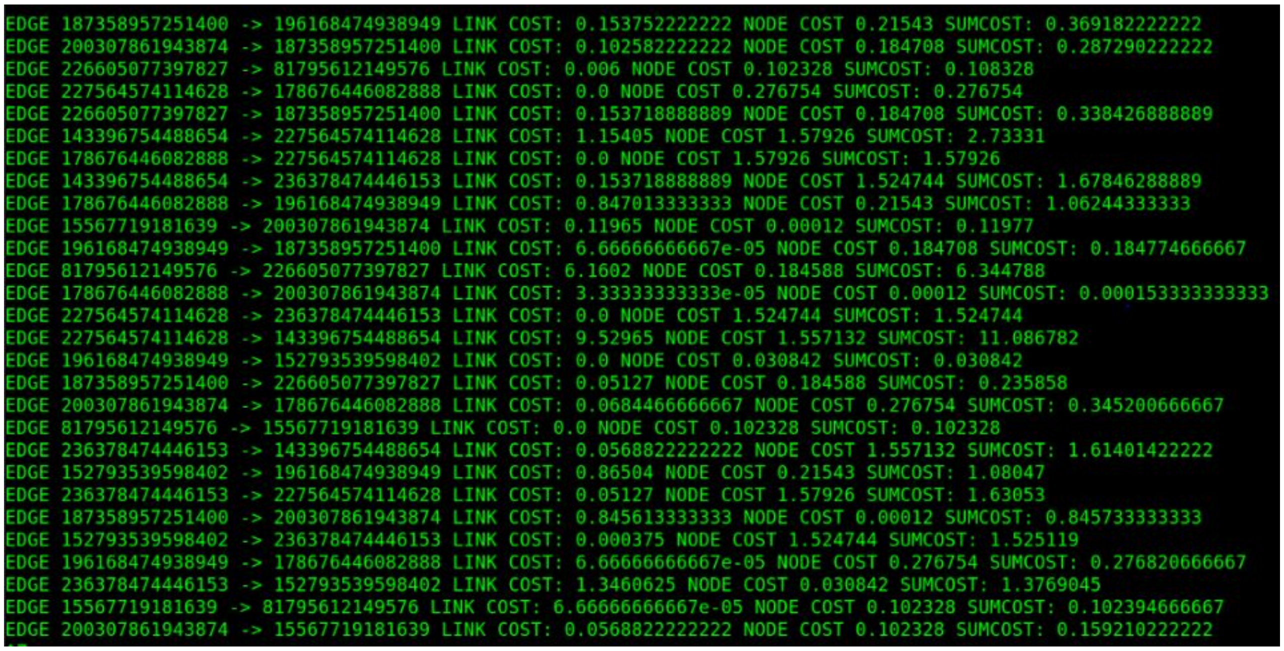
```
 (226605077397827, 187358957251400, {'src_port': '00000004', 'dst_port':
'00000001'})
5.Link between server switch 1 and switch 9
 (227564574114628, 178676446082888, {'src_port': '00000003', 'dst_port':
'00000002'})
6.Link between server switch 1 and switch 2
 (227564574114628, 236378474446153, {'src_port': '00000004', 'dst_port':
'00000001'})
7.Link between server switch 1 and switch 1
 (227564574114628, 143396754488654, {'src_port': '00000002', 'dst_port':
'00000003'})
8.Link between switch 4 and switch 5
(196168474938949, 187358957251400, {'src_port': '00000003', 'dst_port':
'00000002'})
9.Link between switch 4 and switch 9
 (196168474938949, 178676446082888, {'src_port': '00000002', 'dst_port':
'00000003'})
10.Link between switch 4 and switch 3
 (196168474938949, 152793539598402, {'src_port': '00000001', 'dst_port':
'00000001'})
11.Link between switch 7 and server switch 2
  (15567719181639, 81795612149576, {'src_port': '00000002', 'dst_port':
'00000002'})
12.Link between switch 7 and switch 8
  (15567719181639, 200307861943874, {'src_port': '00000003', 'dst_port':
'00000002'})
13.Link between server switch 2 and switch 6
  (81795612149576, 226605077397827, {'src_port': '00000003', 'dst_port':
'00000001'})
14.Link between server switch 2 and switch 7
  (81795612149576, 15567719181639, {'src_port': '00000002', 'dst_port':
'00000002'})
15.Link between switch 2 and switch 3
 (236378474446153, 152793539598402, {'src_port': '00000003', 'dst_port':
'00000003'})
16.Link between switch 2 and server switch 1
 (236378474446153, 227564574114628, {'src_port': '00000001', 'dst_port':
'00000004'})
17.Link between switch 2 and switch 1
 (236378474446153, 143396754488654, {'src_port': '00000004', 'dst_port':
'00000002'})
18.Link between switch 8 and switch 5
 (200307861943874, 187358957251400, {'src_port': '00000003', 'dst_port':
'00000003'})
19.Link between switch 8 and switch 9
 (200307861943874, 178676446082888, {'src_port': '00000001', 'dst_port':
'00000001'})
20.Link between switch 8 and switch 7
  (200307861943874, 15567719181639, {'src_port': '00000002', 'dst_port':
'00000003'})
21.Link between switch 1 and switch 2
 (143396754488654, 236378474446153, {'src_port': '00000002', 'dst_port':
'00000004'})
22.Link between switch 1 and server switch 1
```

```
 (143396754488654, 227564574114628, {'src_port': '00000003', 'dst_port':
'00000002'})
23.Link between switch 9 and switch 8
 (178676446082888, 200307861943874, {'src_port': '00000001', 'dst_port':
'00000001'})
24.Link between switch 9 and switch 1
 (178676446082888, 227564574114628, {'src_port': '00000002', 'dst_port':
'00000003'})
25.Link between switch 9 and switch 4
 (178676446082888, 196168474938949, {'src_port': '00000003', 'dst_port':
'00000002'})
26.Link between switch 5 and switch 8
 (187358957251400, 200307861943874, {'src_port': '00000003', 'dst_port':
'00000003'})
27.Link between switch 5 and switch 6
 (187358957251400, 226605077397827, {'src_port': '00000001', 'dst_port':
'00000004'})
28.Link between switch 5 and switch 4
 (187358957251400, 196168474938949, {'src_port': '00000002', 'dst_port':
'00000003'})
```

Success Criteria: Controller is able to locate existence of switch and links between switches.

5. **Verify calculation of link cost and node cost.**
   Aim: Verify that the Cost Function Unit is calculating the correct link and node cost for implementation of Extended Dijkstra's Algorithm.
   Procedure:
   1. Start RYU controller module on the controller node using the command "ryu-manager extended-load-balancer"
   2. Ping the server from multiple hosts and observe the value of link and node costs on the controller output.

   Output:



```
EDGE 187358957251400 -> 196168474938949 LINK COST: 0.153752222222 NODE COST 0.21543 SUMCOST: 0.369182222222
EDGE 200307861943874 -> 187358957251400 LINK COST: 0.102582222222 NODE COST 0.184708 SUMCOST: 0.287290222222
EDGE 226605077397827 -> 81795612149576 LINK COST: 0.006 NODE COST 0.102328 SUMCOST: 0.108328
EDGE 227564574114628 -> 178676446082888 LINK COST: 0.0 NODE COST 0.276754 SUMCOST: 0.276754
EDGE 226605077397827 -> 187358957251400 LINK COST: 0.153718888889 NODE COST 0.184708 SUMCOST: 0.338426888889
EDGE 143396754488654 -> 227564574114628 LINK COST: 1.15405 NODE COST 1.57926 SUMCOST: 2.73331
EDGE 178676446082888 -> 227564574114628 LINK COST: 0.0 NODE COST 1.57926 SUMCOST: 1.57926
EDGE 143396754488654 -> 236378474446153 LINK COST: 0.153718888889 NODE COST 1.524744 SUMCOST: 1.67846288889
EDGE 178676446082888 -> 196168474938949 LINK COST: 0.847013333333 NODE COST 0.21543 SUMCOST: 1.06244333333
EDGE 15567719181639 -> 200307861943874 LINK COST: 0.11965 NODE COST 0.00012 SUMCOST: 0.11977
EDGE 196168474938949 -> 187358957251400 LINK COST: 6.66666666667e-05 NODE COST 0.184708 SUMCOST: 0.184774666667
EDGE 81795612149576 -> 226605077397827 LINK COST: 6.1602 NODE COST 0.184588 SUMCOST: 6.344788
EDGE 178676446082888 -> 200307861943874 LINK COST: 3.33333333333e-05 NODE COST 0.00012 SUMCOST: 0.000153333333333
EDGE 227564574114628 -> 236378474446153 LINK COST: 0.0 NODE COST 1.524744 SUMCOST: 1.524744
EDGE 227564574114628 -> 143396754488654 LINK COST: 9.52965 NODE COST 1.557132 SUMCOST: 11.086782
EDGE 196168474938949 -> 152793539598402 LINK COST: 0.0 NODE COST 0.030842 SUMCOST: 0.030842
EDGE 187358957251400 -> 226605077397827 LINK COST: 0.05127 NODE COST 0.184588 SUMCOST: 0.235858
EDGE 200307861943874 -> 178676446082888 LINK COST: 0.0684466666667 NODE COST 0.276754 SUMCOST: 0.345200666667
EDGE 81795612149576 -> 15567719181639 LINK COST: 0.0 NODE COST 0.102328 SUMCOST: 0.102328
EDGE 236378474446153 -> 143396754488654 LINK COST: 0.0568822222222 NODE COST 1.557132 SUMCOST: 1.61401422222
EDGE 152793539598402 -> 196168474938949 LINK COST: 0.86504 NODE COST 0.21543 SUMCOST: 1.08047
EDGE 236378474446153 -> 227564574114628 LINK COST: 0.05127 NODE COST 1.57926 SUMCOST: 1.63053
EDGE 187358957251400 -> 200307861943874 LINK COST: 0.845613333333 NODE COST 0.00012 SUMCOST: 0.845733333333
EDGE 152793539598402 -> 236378474446153 LINK COST: 0.000375 NODE COST 1.524744 SUMCOST: 1.525119
EDGE 196168474938949 -> 178676446082888 LINK COST: 6.66666666667e-05 NODE COST 0.276754 SUMCOST: 0.276820666667
EDGE 236378474446153 -> 152793539598402 LINK COST: 1.3460625 NODE COST 0.030842 SUMCOST: 1.3769045
EDGE 15567719181639 -> 81795612149576 LINK COST: 6.66666666667e-05 NODE COST 0.102328 SUMCOST: 0.102394666667
EDGE 200307861943874 -> 15567719181639 LINK COST: 0.0568822222222 NODE COST 0.102328 SUMCOST: 0.159210222222
```

Link and node costs

Success Criteria: Link and node costs are calculated and verified.

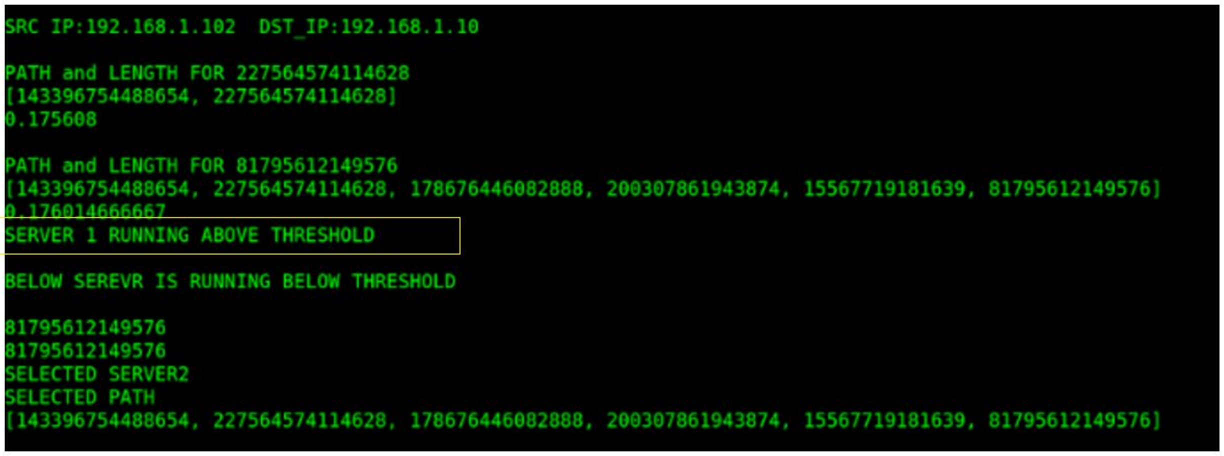
6. **Verify calculation of server-link threshold**

   Aim: Verify that Load Balancer (thread running server switch monitor function in the code) is able to get information about bits processed by link.

   Procedure:

   1. Start RYU controller module on the controller node using the command "ryu-manager extended-load-balancer"
   2. Run iperf server on the server1 and client on host1.
   3. Ping server from host2 and observe controller output.

   Output:

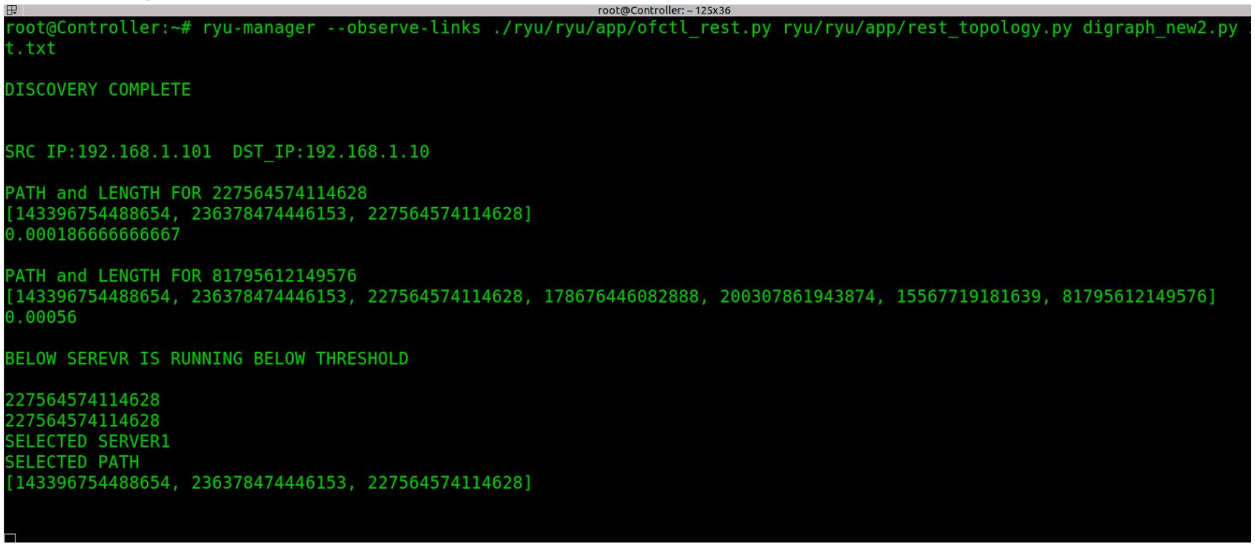


```
SRC IP:192.168.1.102  DST_IP:192.168.1.10

PATH and LENGTH FOR 227564574114628
[143396754488654, 227564574114628]
0.175608

PATH and LENGTH FOR 81795612149576
[143396754488654, 227564574114628, 178676446082888, 200307861943874, 15567719181639, 81795612149576]
0.176014666667
SERVER 1 RUNNING ABOVE THRESHOLD

BELOW SEREVR IS RUNNING BELOW THRESHOLD

81795612149576
81795612149576
SELECTED SERVER2
SELECTED PATH
[143396754488654, 227564574114628, 178676446082888, 200307861943874, 15567719181639, 81795612149576]
```

   Success Criteria: Controller selects correct path based on threshold value of server-link. It is verified that selected server is running above threshold.

7. **Verify that Extended Algorithm calculates cost for each server-switch from the source switch**
   Aim: Verify that Extended Dijkstra's Algorithm calculates cost for each server-switch by traversing through minimal cost path
   Procedure:
    1. Start RYU controller module on the controller node using the command "ryu-manager extended-load-balancer"
    2. Ping server from Host1.
    3. Observe controller output.

   Output:

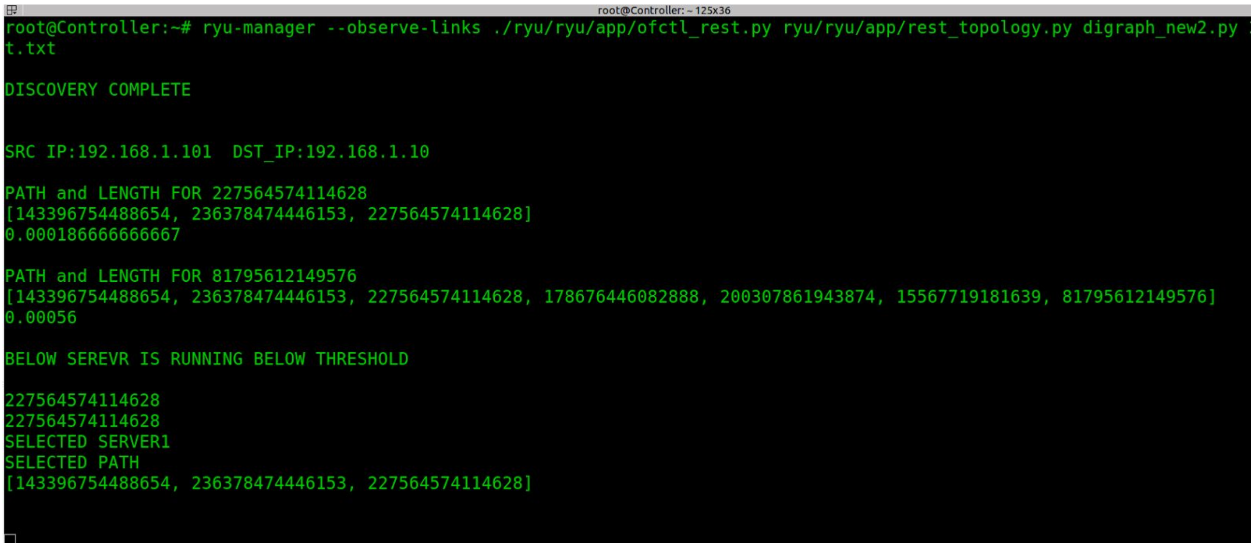


```
root@Controller:~# ryu-manager --observe-links ./ryu/ryu/app/ofctl_rest.py ryu/ryu/app/rest_topology.py digraph_new2.py
t.txt

DISCOVERY COMPLETE


SRC IP:192.168.1.101  DST_IP:192.168.1.10

PATH and LENGTH FOR 227564574114628
[143396754488654, 236378474446153, 227564574114628]
0.000186666666667

PATH and LENGTH FOR 81795612149576
[143396754488654, 236378474446153, 227564574114628, 178676446082888, 200307861943874, 15567719181639, 81795612149576]
0.00056

BELOW SEREVR IS RUNNING BELOW THRESHOLD

227564574114628
227564574114628
SELECTED SERVER1
SELECTED PATH
[143396754488654, 236378474446153, 227564574114628]
```

   Success Criteria: Shortest paths are calculated for both servers .

8. **Verify that Load Balancing Engine selects appropriate server for the request**
   Aim: Verify that minimal cost path is selected for serving the request
   Procedure:
    1. Start RYU controller module on the controller node using the command "ryu-manager extended-load-balancer"
    2. Ping server from host1.
    3. Observe Controller output.

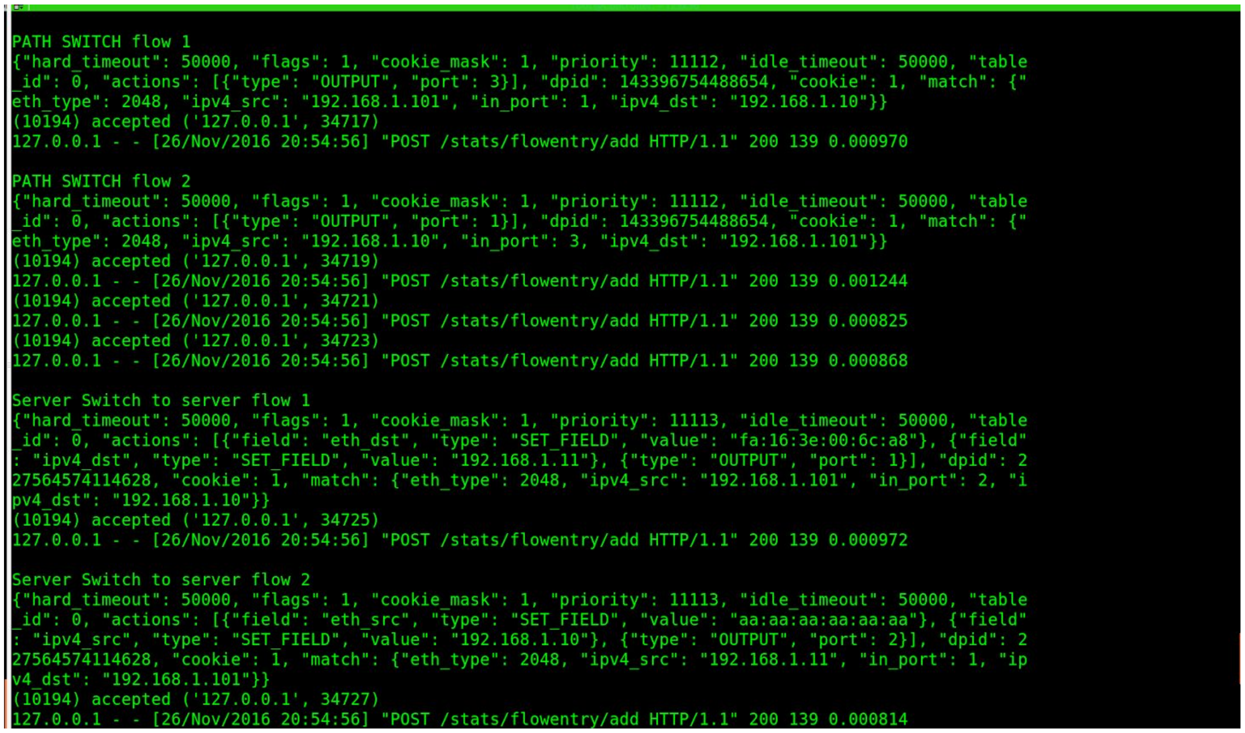
   Output:



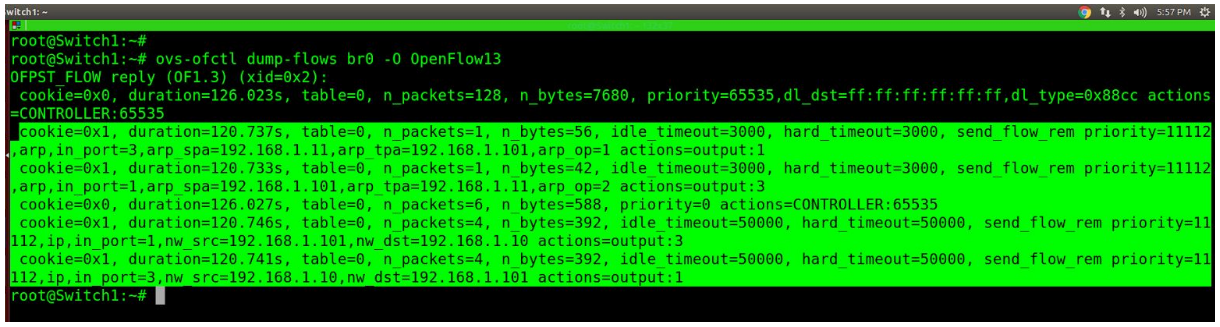
```
root@Controller:~# ryu-manager --observe-links ./ryu/ryu/app/ofctl_rest.py ryu/ryu/app/rest_topology.py digraph_new2.py
t.txt

DISCOVERY COMPLETE


SRC IP:192.168.1.101  DST_IP:192.168.1.10

PATH and LENGTH FOR 227564574114628
[143396754488654, 236378474446153, 227564574114628]
0.000186666666667

PATH and LENGTH FOR 81795612149576
[143396754488654, 236378474446153, 227564574114628, 178676446082888, 200307861943874, 15567719181639, 81795612149576]
0.00056

BELOW SEREVR IS RUNNING BELOW THRESHOLD

227564574114628
227564574114628
SELECTED SERVER1
SELECTED PATH
[143396754488654, 236378474446153, 227564574114628]
```

Success Criteria: Load Balancing Engine should select minimal cost path for a server below threshold for serving the client request.

9. **Verify that Flow Writer writes appropriate flow on the switches**
   Aim: Verify that Flow writer writes flows to all the switches in the path as per flow in flow database
   Procedure:
   1. Start RYU controller module on the controller node using the command “`ryu-manager extended-load-balancer`”
   2. Configure controller IP on OVS using the command ”`ovs-vsctl set-controller br0 tcp:128.194.6.173:6633`”
   3. Verify OVS is connected to controller using the command ”`ovs-vsctl show br0 -O OpenFlow13`”
   4. Run the flow writer module in digraph.py program which will add flows using ryu.app.ofctl_rest api running on local host
   5. Let the flow Writer write the flow on each switch in the path using the command
      `Curl -X POST http://localhost/stats/flow/add/<switch_dpid>`
   6. Verify on each of the switch that particular flow is written as per the entries for selected flow in flow database using the command:
      `ovs-ofctl dump-flows br0 -O OpenFlow13`



```
PATH SWITCH flow 1
{"hard_timeout": 50000, "flags": 1, "cookie_mask": 1, "priority": 11112, "idle_timeout": 50000, "table
_id": 0, "actions": [{"type": "OUTPUT", "port": 3}], "dpid": 143396754488654, "cookie": 1, "match": {"
eth_type": 2048, "ipv4_src": "192.168.1.101", "in_port": 1, "ipv4_dst": "192.168.1.10"}}
(10194) accepted ('127.0.0.1', 34717)
127.0.0.1 - - [26/Nov/2016 20:54:56] "POST /stats/flowentry/add HTTP/1.1" 200 139 0.000970

PATH SWITCH flow 2
{"hard_timeout": 50000, "flags": 1, "cookie_mask": 1, "priority": 11112, "idle_timeout": 50000, "table
_id": 0, "actions": [{"type": "OUTPUT", "port": 1}], "dpid": 143396754488654, "cookie": 1, "match": {"
eth_type": 2048, "ipv4_src": "192.168.1.10", "in_port": 3, "ipv4_dst": "192.168.1.101"}}
(10194) accepted ('127.0.0.1', 34719)
127.0.0.1 - - [26/Nov/2016 20:54:56] "POST /stats/flowentry/add HTTP/1.1" 200 139 0.001244
(10194) accepted ('127.0.0.1', 34721)
127.0.0.1 - - [26/Nov/2016 20:54:56] "POST /stats/flowentry/add HTTP/1.1" 200 139 0.000825
(10194) accepted ('127.0.0.1', 34723)
127.0.0.1 - - [26/Nov/2016 20:54:56] "POST /stats/flowentry/add HTTP/1.1" 200 139 0.000868

Server Switch to server flow 1
{"hard_timeout": 50000, "flags": 1, "cookie_mask": 1, "priority": 11113, "idle_timeout": 50000, "table
_id": 0, "actions": [{"field": "eth_dst", "type": "SET_FIELD", "value": "fa:16:3e:00:6c:a8"}, {"field"
: "ipv4_dst", "type": "SET_FIELD", "value": "192.168.1.11"}, {"type": "OUTPUT", "port": 1}], "dpid": 2
27564574114628, "cookie": 1, "match": {"eth_type": 2048, "ipv4_src": "192.168.1.101", "in_port": 2, "i
pv4_dst": "192.168.1.10"}}
(10194) accepted ('127.0.0.1', 34725)
127.0.0.1 - - [26/Nov/2016 20:54:56] "POST /stats/flowentry/add HTTP/1.1" 200 139 0.000972

Server Switch to server flow 2
{"hard_timeout": 50000, "flags": 1, "cookie_mask": 1, "priority": 11113, "idle_timeout": 50000, "table
_id": 0, "actions": [{"field": "eth_src", "type": "SET_FIELD", "value": "aa:aa:aa:aa:aa:aa"}, {"field"
: "ipv4_src", "type": "SET_FIELD", "value": "192.168.1.10"}, {"type": "OUTPUT", "port": 2}], "dpid": 2
27564574114628, "cookie": 1, "match": {"eth_type": 2048, "ipv4_src": "192.168.1.11", "in_port": 1, "ip
v4_dst": "192.168.1.101"}}
(10194) accepted ('127.0.0.1', 34727)
127.0.0.1 - - [26/Nov/2016 20:54:56] "POST /stats/flowentry/add HTTP/1.1" 200 139 0.000814
```



```
root@Switch1:~#
root@Switch1:~# ovs-ofctl dump-flows br0 -O OpenFlow13
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=126.023s, table=0, n_packets=128, n_bytes=7680, priority=65535,dl_dst=ff:ff:ff:ff:ff:ff,dl_type=0x88cc actions
=CONTROLLER:65535
 cookie=0x1, duration=120.737s, table=0, n_packets=1, n_bytes=56, idle_timeout=3000, hard_timeout=3000, send_flow_rem priority=11112
,arp,in_port=3,arp_spa=192.168.1.11,arp_tpa=192.168.1.101,arp_op=1 actions=output:1
 cookie=0x1, duration=120.733s, table=0, n_packets=1, n_bytes=42, idle_timeout=3000, hard_timeout=3000, send_flow_rem priority=11112
,arp,in_port=1,arp_spa=192.168.1.101,arp_tpa=192.168.1.11,arp_op=2 actions=output:3
 cookie=0x0, duration=126.027s, table=0, n_packets=6, n_bytes=588, priority=0 actions=CONTROLLER:65535
 cookie=0x1, duration=120.746s, table=0, n_packets=4, n_bytes=392, idle_timeout=50000, hard_timeout=50000, send_flow_rem priority=11
112,ip,in_port=1,nw_src=192.168.1.101,nw_dst=192.168.1.10 actions=output:3
 cookie=0x1, duration=120.741s, table=0, n_packets=4, n_bytes=392, idle_timeout=50000, hard_timeout=50000, send_flow_rem priority=11
112,ip,in_port=3,nw_src=192.168.1.10,nw_dst=192.168.1.101 actions=output:1
root@Switch1:~#
```

Success Criteria: As seen from the dump-flows output, correct flows were written.

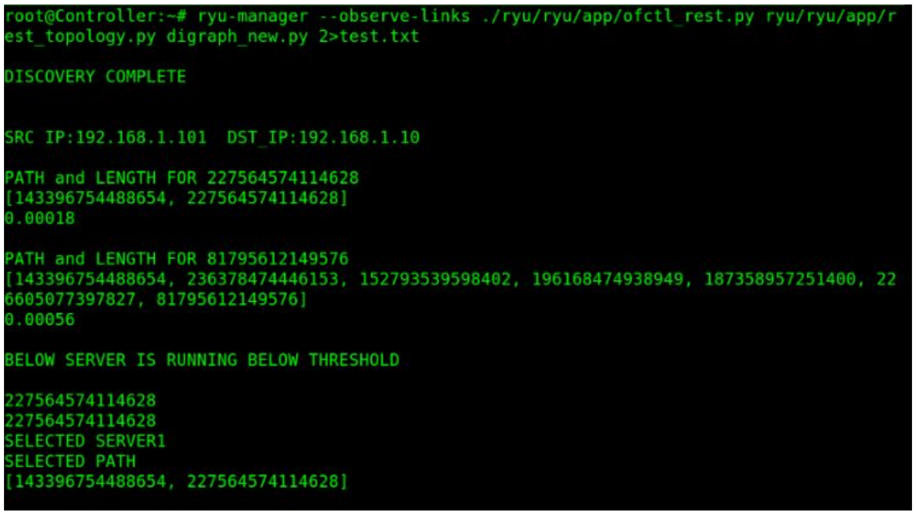
10. Verify that packets follow the path written by Flow Writer

Aim: Verify that datapath followed for traffic is correct and according to the path decided

Procedure:

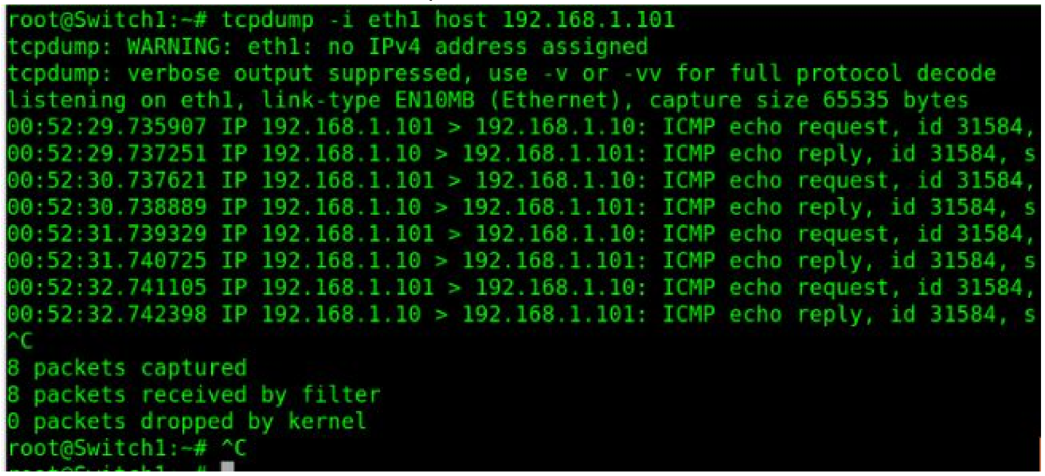
1. Start RYU controller module on the controller node using the command "`ryu-manager extended-load-balancer`"
2. Configure controller IP on OVS using the command "`ovs-vsctl set-controller br0 tcp:128.194.6.173:6633`"
3. Verify OVS is connected to controller using the command "`ovs-vsctl show br0 -O OpenFlow13`"
4. Generate traffic from a host and observe tcpdump outputs on the switches along the selected path displayed by the controller.

Success Criteria: Screenshots showing outputs of tcpdump for Switch1, Switch2 and ServerSwitch1 indicate the path followed by the packet.



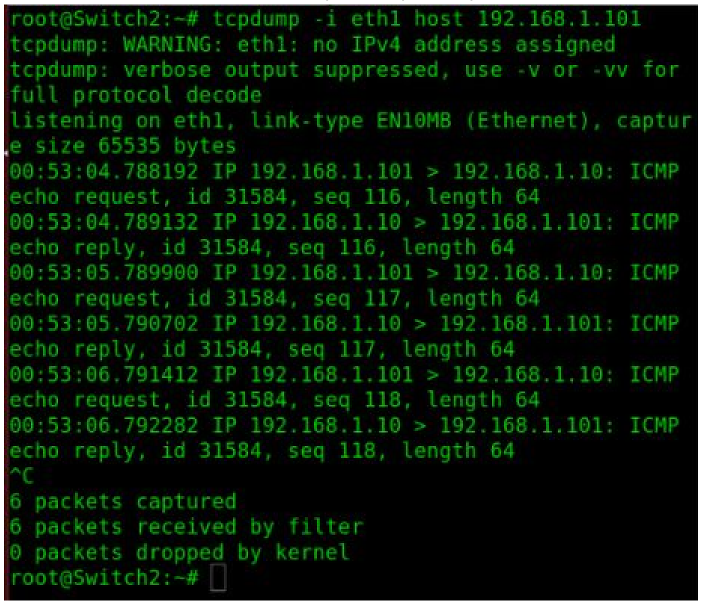
```
root@Controller:~# ryu-manager --observe-links ./ryu/ryu/app/ofctl_rest.py ryu/ryu/app/r
est_topology.py digraph_new.py 2>test.txt

DISCOVERY COMPLETE


SRC IP:192.168.1.101  DST_IP:192.168.1.10

PATH and LENGTH FOR 227564574114628
[143396754488654, 227564574114628]
0.00018

PATH and LENGTH FOR 81795612149576
[143396754488654, 236378474446153, 152793539598402, 196168474938949, 187358957251400, 22
6605077397827, 81795612149576]
0.00056

BELOW SERVER IS RUNNING BELOW THRESHOLD

227564574114628
227564574114628
SELECTED SERVER1
SELECTED PATH
[143396754488654, 227564574114628]
```

Which path has lowest cost.



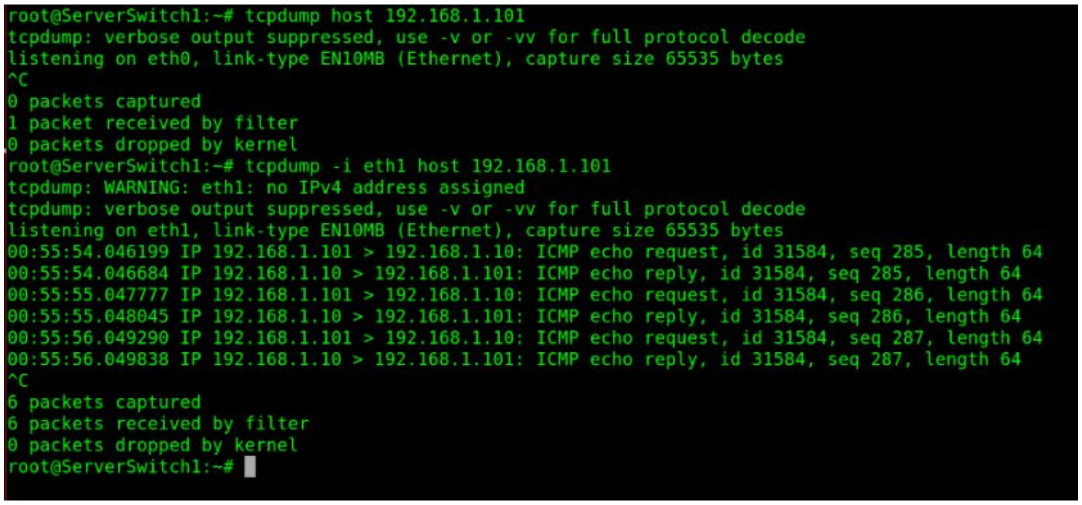
```
root@Switch1:~# tcpdump -i eth1 host 192.168.1.101
tcpdump: WARNING: eth1: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 65535 bytes
00:52:29.735907 IP 192.168.1.101 > 192.168.1.10: ICMP echo request, id 31584,
00:52:29.737251 IP 192.168.1.10 > 192.168.1.101: ICMP echo reply, id 31584, s
00:52:30.737621 IP 192.168.1.101 > 192.168.1.10: ICMP echo request, id 31584,
00:52:30.738889 IP 192.168.1.10 > 192.168.1.101: ICMP echo reply, id 31584, s
00:52:31.739329 IP 192.168.1.101 > 192.168.1.10: ICMP echo request, id 31584,
00:52:31.740725 IP 192.168.1.10 > 192.168.1.101: ICMP echo reply, id 31584, s
00:52:32.741105 IP 192.168.1.101 > 192.168.1.10: ICMP echo request, id 31584,
00:52:32.742398 IP 192.168.1.10 > 192.168.1.101: ICMP echo reply, id 31584, s
^C
8 packets captured
8 packets received by filter
0 packets dropped by kernel
root@Switch1:~# ^C
```

Switch1 tcpdump output



```
root@Switch2:~# tcpdump -i eth1 host 192.168.1.101
tcpdump: WARNING: eth1: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for
full protocol decode
listening on eth1, link-type EN10MB (Ethernet), captur
e size 65535 bytes
00:53:04.788192 IP 192.168.1.101 > 192.168.1.10: ICMP
echo request, id 31584, seq 116, length 64
00:53:04.789132 IP 192.168.1.10 > 192.168.1.101: ICMP
echo reply, id 31584, seq 116, length 64
00:53:05.789900 IP 192.168.1.101 > 192.168.1.10: ICMP
echo request, id 31584, seq 117, length 64
00:53:05.790702 IP 192.168.1.10 > 192.168.1.101: ICMP
echo reply, id 31584, seq 117, length 64
00:53:06.791412 IP 192.168.1.101 > 192.168.1.10: ICMP
echo request, id 31584, seq 118, length 64
00:53:06.792282 IP 192.168.1.10 > 192.168.1.101: ICMP
echo reply, id 31584, seq 118, length 64
^C
6 packets captured
6 packets received by filter
0 packets dropped by kernel
root@Switch2:~#
```

Switch2 tcpdump output

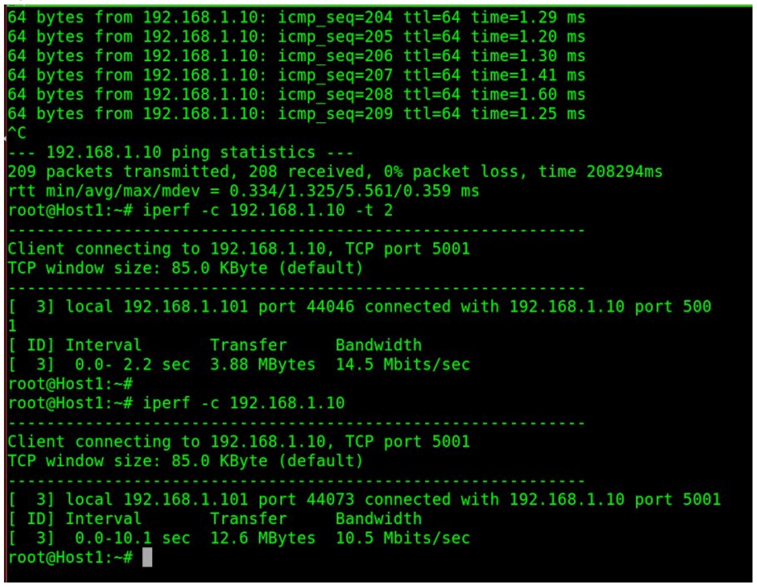


```
root@ServerSwitch1:~# tcpdump host 192.168.1.101
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
^C
0 packets captured
1 packet received by filter
0 packets dropped by kernel
root@ServerSwitch1:~# tcpdump -i eth1 host 192.168.1.101
tcpdump: WARNING: eth1: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 65535 bytes
00:55:54.046199 IP 192.168.1.101 > 192.168.1.10: ICMP echo request, id 31584, seq 285, length 64
00:55:54.046684 IP 192.168.1.10 > 192.168.1.101: ICMP echo reply, id 31584, seq 285, length 64
00:55:55.047777 IP 192.168.1.101 > 192.168.1.10: ICMP echo request, id 31584, seq 286, length 64
00:55:55.048045 IP 192.168.1.10 > 192.168.1.101: ICMP echo reply, id 31584, seq 286, length 64
00:55:56.049290 IP 192.168.1.101 > 192.168.1.10: ICMP echo request, id 31584, seq 287, length 64
00:55:56.049838 IP 192.168.1.10 > 192.168.1.101: ICMP echo reply, id 31584, seq 287, length 64
^C
6 packets captured
6 packets received by filter
0 packets dropped by kernel
root@ServerSwitch1:~#
```

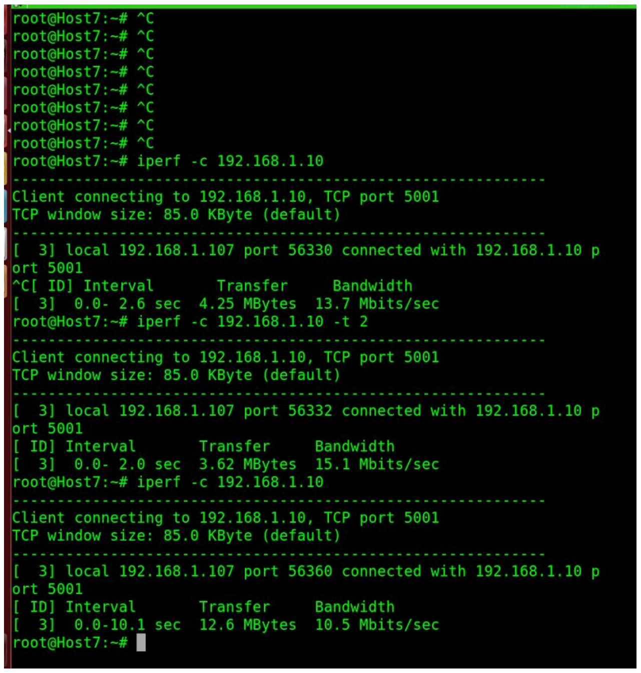
ServerSwitch1 tcpdump output

5.3.1.2 Integration Testing

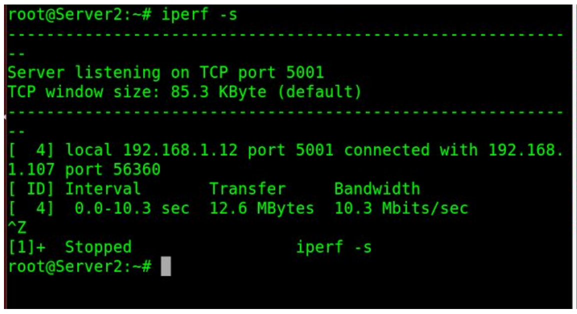
1. **Load balancing by selecting different servers for different hosts.**
   Aim: verify that both the servers are being used to serve the clients.
   Procedure:
   1. Start RYU controller module on the controller node using the command "ryu-manager extended-load-balancer"
   2. Verify that the controller application gets the information about all the switches and links in the topology.
   3. Start iperf server on both the servers.
   4. Start iperf client on Host1 to connect with the server.
   5. Start iperf client on Host7 to connect with the server.



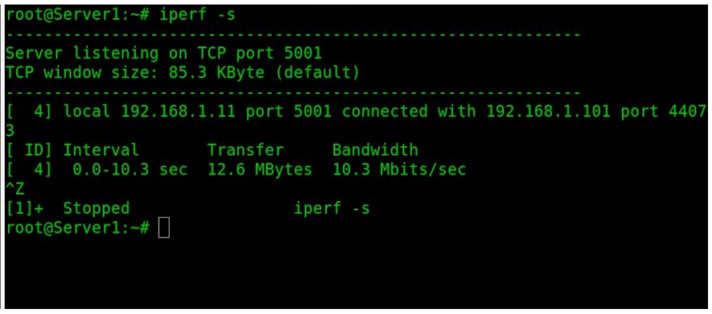
```
64 bytes from 192.168.1.10: icmp_seq=204 ttl=64 time=1.29 ms
64 bytes from 192.168.1.10: icmp_seq=205 ttl=64 time=1.20 ms
64 bytes from 192.168.1.10: icmp_seq=206 ttl=64 time=1.30 ms
64 bytes from 192.168.1.10: icmp_seq=207 ttl=64 time=1.41 ms
64 bytes from 192.168.1.10: icmp_seq=208 ttl=64 time=1.60 ms
64 bytes from 192.168.1.10: icmp_seq=209 ttl=64 time=1.25 ms
^C
--- 192.168.1.10 ping statistics ---
209 packets transmitted, 208 received, 0% packet loss, time 208294ms
rtt min/avg/max/mdev = 0.334/1.325/5.561/0.359 ms
root@Host1:~# iperf -c 192.168.1.10 -t 2
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.101 port 44046 connected with 192.168.1.10 port 500
1
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 2.2 sec  3.88 MBytes  14.5 Mbits/sec
root@Host1:~#
root@Host1:~# iperf -c 192.168.1.10
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.101 port 44073 connected with 192.168.1.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.1 sec  12.6 MBytes  10.5 Mbits/sec
root@Host1:~#
```



```
root@Host7:~# ^C
root@Host7:~# ^C
root@Host7:~# ^C
root@Host7:~# ^C
root@Host7:~# ^C
root@Host7:~# ^C
root@Host7:~# ^C
root@Host7:~# ^C
root@Host7:~# iperf -c 192.168.1.10
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.107 port 56330 connected with 192.168.1.10 p
ort 5001
^C[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 2.6 sec  4.25 MBytes  13.7 Mbits/sec
root@Host7:~# iperf -c 192.168.1.10 -t 2
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.107 port 56332 connected with 192.168.1.10 p
ort 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 2.0 sec  3.62 MBytes  15.1 Mbits/sec
root@Host7:~# iperf -c 192.168.1.10
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.107 port 56360 connected with 192.168.1.10 p
ort 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.1 sec  12.6 MBytes  10.5 Mbits/sec
root@Host7:~# 
```



```
root@Server2:~# iperf -s
--------------------------------------------------------------
--
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
--------------------------------------------------------------
--
[  4] local 192.168.1.12 port 5001 connected with 192.168.
1.107 port 56360
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.3 sec  12.6 MBytes  10.3 Mbits/sec
^Z
[1]+  Stopped                 iperf -s
root@Server2:~# 
```

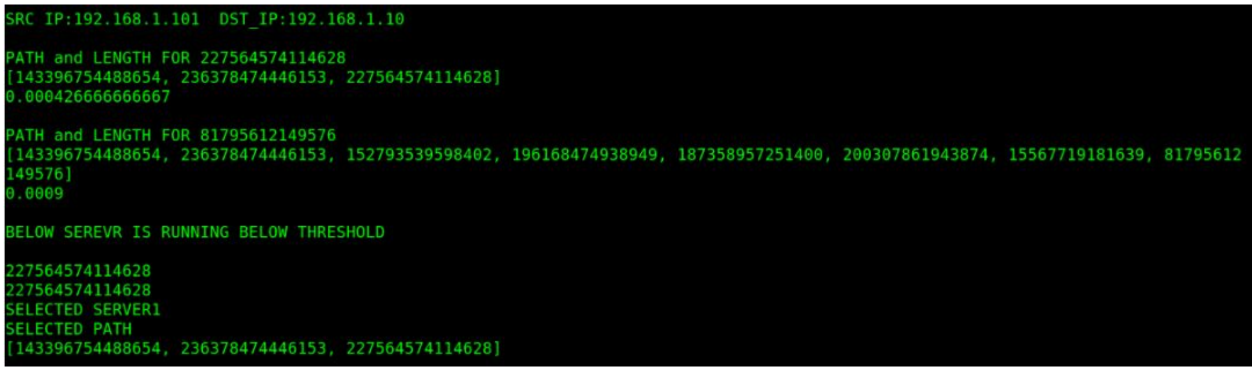


```
root@Server1:~# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 192.168.1.11 port 5001 connected with 192.168.1.101 port 4407
3
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.3 sec  12.6 MBytes  10.3 Mbits/sec
^Z
[1]+  Stopped                 iperf -s
root@Server1:~# 
```

Success Criteria: Server1 communicates with Host1 and Server2 communicates with Host7.

2. **Selection of cost efficient path to reach the nearest server where shortest path is not necessarily optimum.**
   Aim: Verify that path selection is cost efficient.
   Procedure:
   1. Start RYU controller module on the controller node using the command "ryu-manager extended-load-balancer"
   2. Ping successfully from Host1 to virtual IP.
   3. Display switch datapath Ids in the path selected.

   Output:

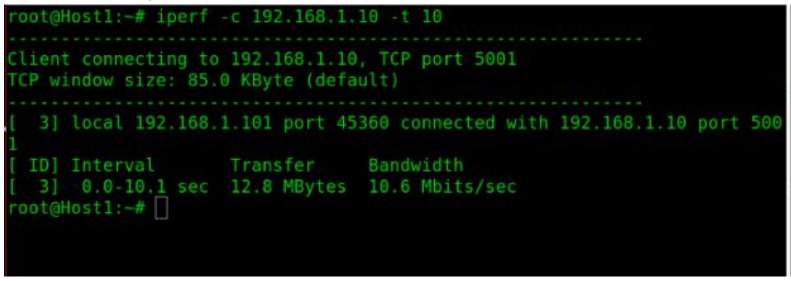


```
SRC IP:192.168.1.101  DST_IP:192.168.1.10

PATH and LENGTH FOR 227564574114628
[143396754488654, 236378474446153, 227564574114628]
0.000426666666667

PATH and LENGTH FOR 81795612149576
[143396754488654, 236378474446153, 152793539598402, 196168474938949, 187358957251400, 200307861943874, 15567719181639, 81795612
149576]
0.0009

BELOW SEREVR IS RUNNING BELOW THRESHOLD

227564574114628
227564574114628
SELECTED SERVER1
SELECTED PATH
[143396754488654, 236378474446153, 227564574114628]
```

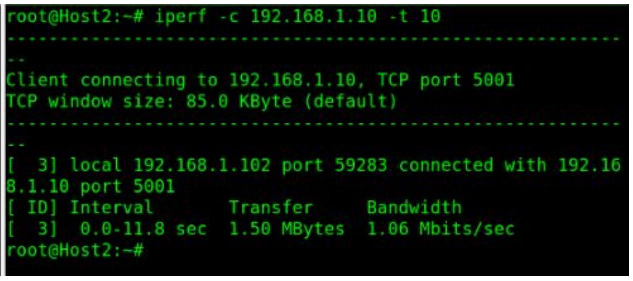
Success Criteria: The nearest server is Server1 and although the shortest path for Host1 to reach server1 is through Switch1->link15-1->ServerSwitch1, the selected path is Switch1->link16->Switch2->link17->ServerSwitch1. This is because links in the selected path are all 10Mbps while that in the shortest path is 1Mbps.

3. **Congestion control by directing traffic to a different server when the utilization of the server-switch link overshoots threshold**
   Aim: A different server should be selected for serving new host request if the
   Procedure:
   1. Start RYU controller module on the controller node using the command "ryu-manager extended-load-balancer"
   2. Start iperf server on both the Servers.
   3. Start iperf client on Host1.
   4. Start iperf client on Host2.

Output:

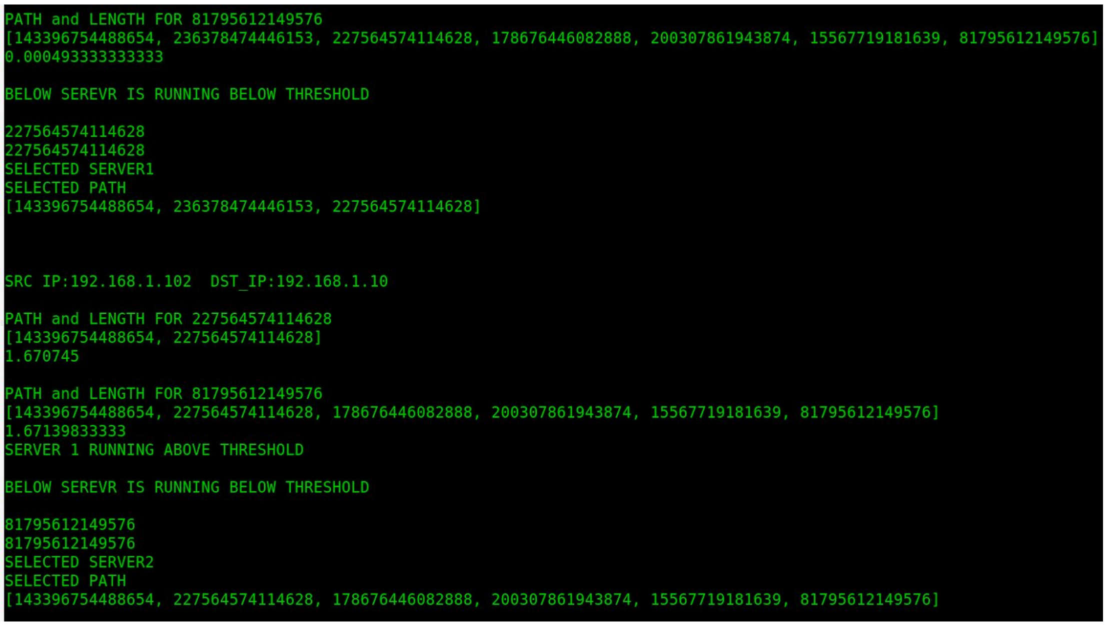


```
root@Host1:~# iperf -c 192.168.1.10 -t 10
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.101 port 45360 connected with 192.168.1.10 port 500
1
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.1 sec  12.8 MBytes  10.6 Mbits/sec
root@Host1:~# 
```



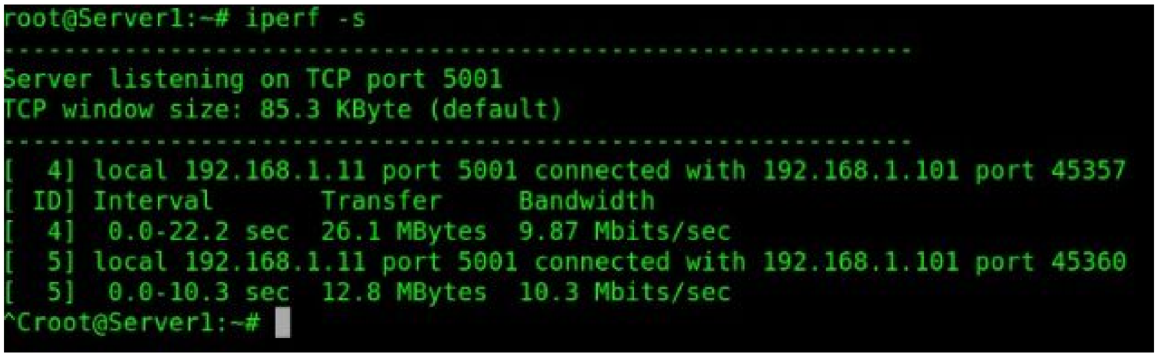
```
root@Host2:~# iperf -c 192.168.1.10 -t 10
------------------------------------------------------------
--
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
--
[  3] local 192.168.1.102 port 59283 connected with 192.16
8.1.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-11.8 sec  1.50 MBytes  1.06 Mbits/sec
root@Host2:~#
```

Hosts

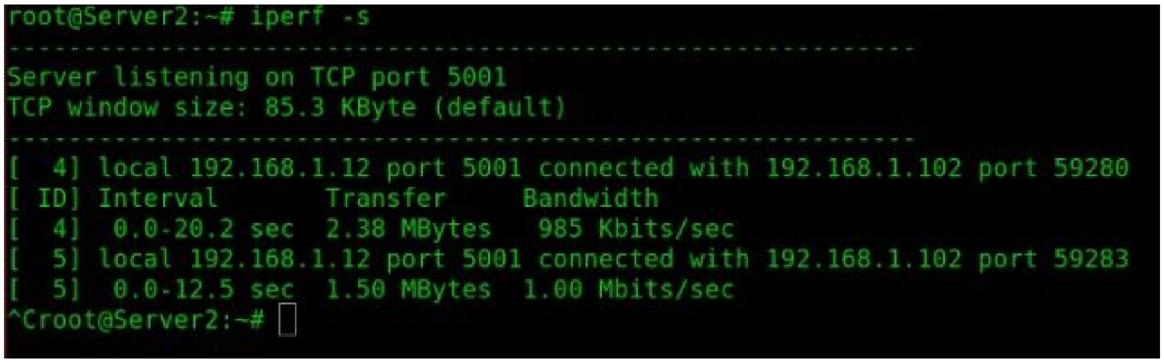


```
PATH and LENGTH FOR 81795612149576
[143396754488654, 236378474446153, 227564574114628, 178676446082888, 200307861943874, 15567719181639, 81795612149576]
0.000493333333333

BELOW SEREVR IS RUNNING BELOW THRESHOLD

227564574114628
227564574114628
SELECTED SERVER1
SELECTED PATH
[143396754488654, 236378474446153, 227564574114628]



SRC IP:192.168.1.102  DST_IP:192.168.1.10

PATH and LENGTH FOR 227564574114628
[143396754488654, 227564574114628]
1.670745

PATH and LENGTH FOR 81795612149576
[143396754488654, 227564574114628, 178676446082888, 200307861943874, 15567719181639, 81795612149576]
1.67139833333
SERVER 1 RUNNING ABOVE THRESHOLD

BELOW SEREVR IS RUNNING BELOW THRESHOLD

81795612149576
81795612149576
SELECTED SERVER2
SELECTED PATH
[143396754488654, 227564574114628, 178676446082888, 200307861943874, 15567719181639, 81795612149576]
```

Client



```
root@Server1:~# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 192.168.1.11 port 5001 connected with 192.168.1.101 port 45357
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-22.2 sec  26.1 MBytes  9.87 Mbits/sec
[  5] local 192.168.1.11 port 5001 connected with 192.168.1.101 port 45360
[  5]  0.0-10.3 sec  12.8 MBytes  10.3 Mbits/sec
^Croot@Server1:~# 
```



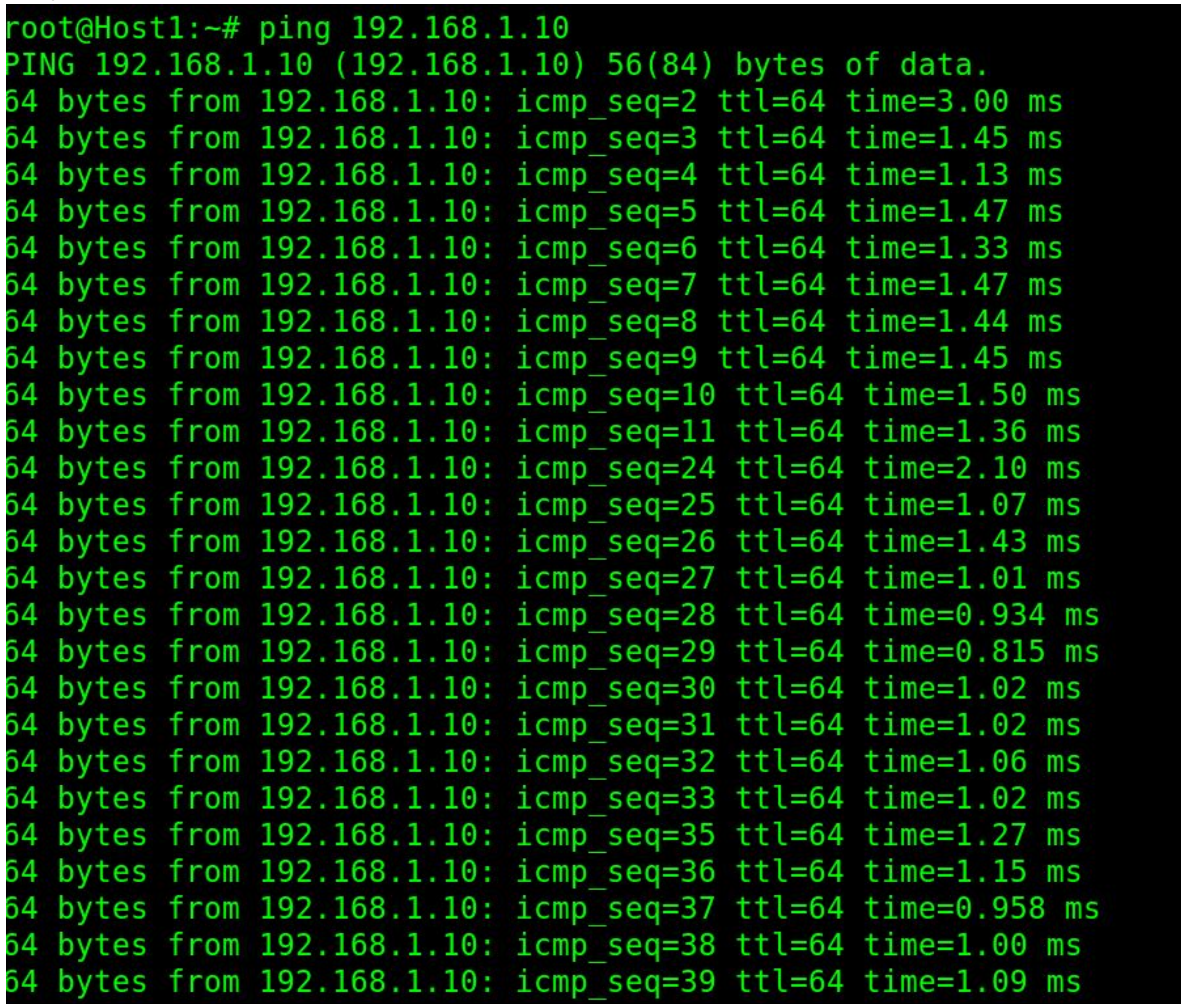
```
root@Server2:~# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 192.168.1.12 port 5001 connected with 192.168.1.102 port 59280
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-20.2 sec  2.38 MBytes   985 Kbits/sec
[  5] local 192.168.1.12 port 5001 connected with 192.168.1.102 port 59283
[  5]  0.0-12.5 sec  1.50 MBytes  1.00 Mbits/sec
^Croot@Server2:~# 
```

Servers

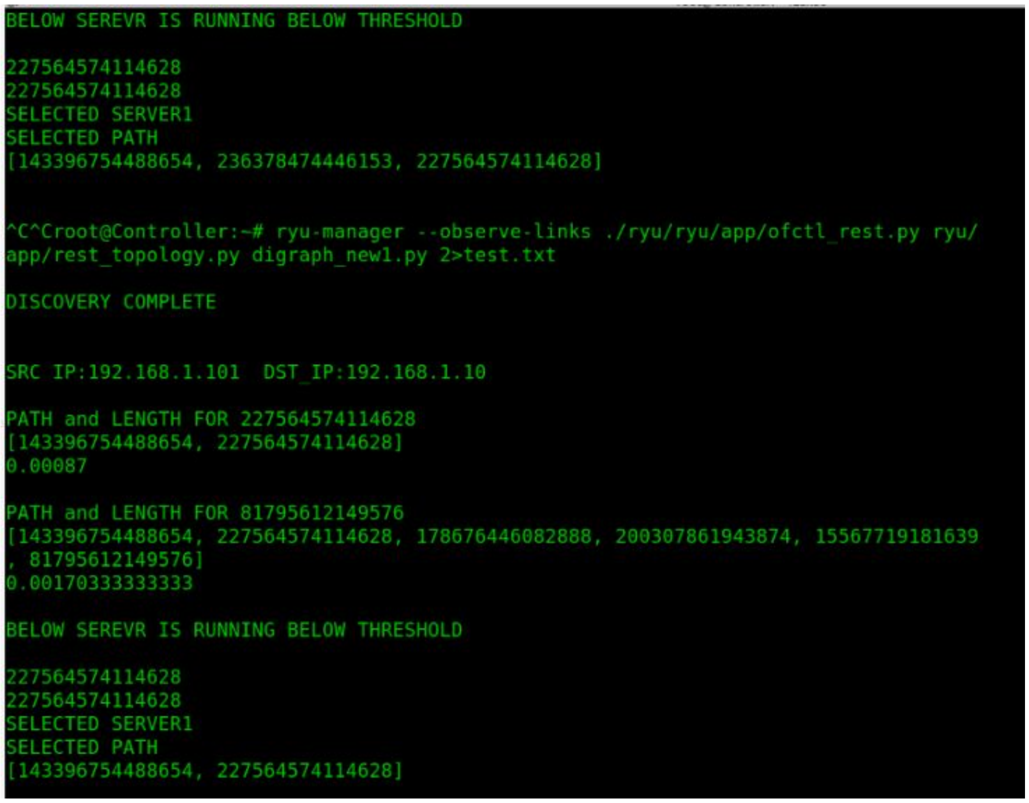
Success Criteria: Server1 is selected for Host1 connection and since threshold for Server1 link is exceeded because of iperf, Server2 will be selected for Host2 communication. Here the connection is reestablished with the Link15-1 in the path. This link has bandwidth 1Mbps unlike the alternative path with links 16 and 17 which are both 1Mbps. However, even then the said path was selected because the links 16 and 17 were being used for communication with Server2 and hence, the cost for the path involving link15-1 was lower than the other path.

4. **Connection between switch and controller fails**
   Aim: Controller should provide different path to the server if switch in the path fails
   Procedure:
   1. Start RYU controller module on the controller node using the command "ryu-manager extended-load-balancer"
   2. Start ping from host 1 to server.
   3. Verify that server 1 is selected.
   4. Stop Switch2 which is along the path to reach server1.
   5. Observe the ping replies and controller output.

   Output:



```
root@Host1:~# ping 192.168.1.10
PING 192.168.1.10 (192.168.1.10) 56(84) bytes of data.
64 bytes from 192.168.1.10: icmp_seq=2 ttl=64 time=3.00 ms
64 bytes from 192.168.1.10: icmp_seq=3 ttl=64 time=1.45 ms
64 bytes from 192.168.1.10: icmp_seq=4 ttl=64 time=1.13 ms
64 bytes from 192.168.1.10: icmp_seq=5 ttl=64 time=1.47 ms
64 bytes from 192.168.1.10: icmp_seq=6 ttl=64 time=1.33 ms
64 bytes from 192.168.1.10: icmp_seq=7 ttl=64 time=1.47 ms
64 bytes from 192.168.1.10: icmp_seq=8 ttl=64 time=1.44 ms
64 bytes from 192.168.1.10: icmp_seq=9 ttl=64 time=1.45 ms
64 bytes from 192.168.1.10: icmp_seq=10 ttl=64 time=1.50 ms
64 bytes from 192.168.1.10: icmp_seq=11 ttl=64 time=1.36 ms
64 bytes from 192.168.1.10: icmp_seq=24 ttl=64 time=2.10 ms
64 bytes from 192.168.1.10: icmp_seq=25 ttl=64 time=1.07 ms
64 bytes from 192.168.1.10: icmp_seq=26 ttl=64 time=1.43 ms
64 bytes from 192.168.1.10: icmp_seq=27 ttl=64 time=1.01 ms
64 bytes from 192.168.1.10: icmp_seq=28 ttl=64 time=0.934 ms
64 bytes from 192.168.1.10: icmp_seq=29 ttl=64 time=0.815 ms
64 bytes from 192.168.1.10: icmp_seq=30 ttl=64 time=1.02 ms
64 bytes from 192.168.1.10: icmp_seq=31 ttl=64 time=1.02 ms
64 bytes from 192.168.1.10: icmp_seq=32 ttl=64 time=1.06 ms
64 bytes from 192.168.1.10: icmp_seq=33 ttl=64 time=1.02 ms
64 bytes from 192.168.1.10: icmp_seq=35 ttl=64 time=1.27 ms
64 bytes from 192.168.1.10: icmp_seq=36 ttl=64 time=1.15 ms
64 bytes from 192.168.1.10: icmp_seq=37 ttl=64 time=0.958 ms
64 bytes from 192.168.1.10: icmp_seq=38 ttl=64 time=1.00 ms
64 bytes from 192.168.1.10: icmp_seq=39 ttl=64 time=1.09 ms
```



```
BELOW SEREVR IS RUNNING BELOW THRESHOLD

227564574114628
227564574114628
SELECTED SERVER1
SELECTED PATH
[143396754488654, 236378474446153, 227564574114628]


^C^Croot@Controller:~# ryu-manager --observe-links ./ryu/ryu/app/ofctl_rest.py ryu/
app/rest_topology.py digraph_new1.py 2>test.txt

DISCOVERY COMPLETE


SRC IP:192.168.1.101  DST_IP:192.168.1.10

PATH and LENGTH FOR 227564574114628
[143396754488654, 227564574114628]
0.00087

PATH and LENGTH FOR 81795612149576
[143396754488654, 227564574114628, 178676446082888, 200307861943874, 15567719181639
, 81795612149576]
0.00170333333333

BELOW SEREVR IS RUNNING BELOW THRESHOLD

227564574114628
227564574114628
SELECTED SERVER1
SELECTED PATH
[143396754488654, 227564574114628]
```

Success Criteria: A new path is selected for serving the request and which specifies connection problem between switch and controller in the previous path and ping reply continues.

5.4 Demo Playbook

Demo ID	Scenario	Observation	Conclusion
1.	Start the RYU controller and verify all the switches connect to controller	Controller shows log of OVS switches connected	Switches are successfully connected to the controller
2.	Start the base case - round-robin application on the controller and generate traffic for the servers from multiple hosts.	High end to end latency and packet loss observed	Due to congestion in the network and the server links, delay and loss is observed in the network.
2.	Start the base case -unit - weight application on the	High end to end latency and packet loss observed	Due to congestion in the network and the

	controller and generate traffic for the servers from multiple hosts.	because of congestion in the network.	server links, delay and loss is observed in the network.
3.	Stop the controller and start the load balancing application	Switches connect to the controller which now runs improved load balancing algorithm	Switches are successfully connected to the load balancing controller
4.	Generate traffic for the servers from multiple hosts.	Lower end to end latency is observed.	Since load balancing controller selects the best server, lower latency than base case is observed.
5.	Generate traffic such that one of the server-link utilization overshoots the threshold value	Traffic is redirected to another server that is below threshold value with minimum packet loss.	Congestion on the server links is avoided by dynamically switching to another server thus improving latency.
6.	Generate Traffic such that the path cost to the server increases due to heavy traffic on the selected path	Traffic is redirected to the server on alternate path that has lower cost	Congestion on the network is avoided by dynamically switching paths and hence lower latency than base case is observed
7.	Generate traffic for the servers and shut down a switch that belongs on of the selected path	Path cost is recalculated and traffic is forwarded through the new best path.	Controller does route recalculation as soon as a link goes down.

Table 7: Demo Plan

7.Self Study

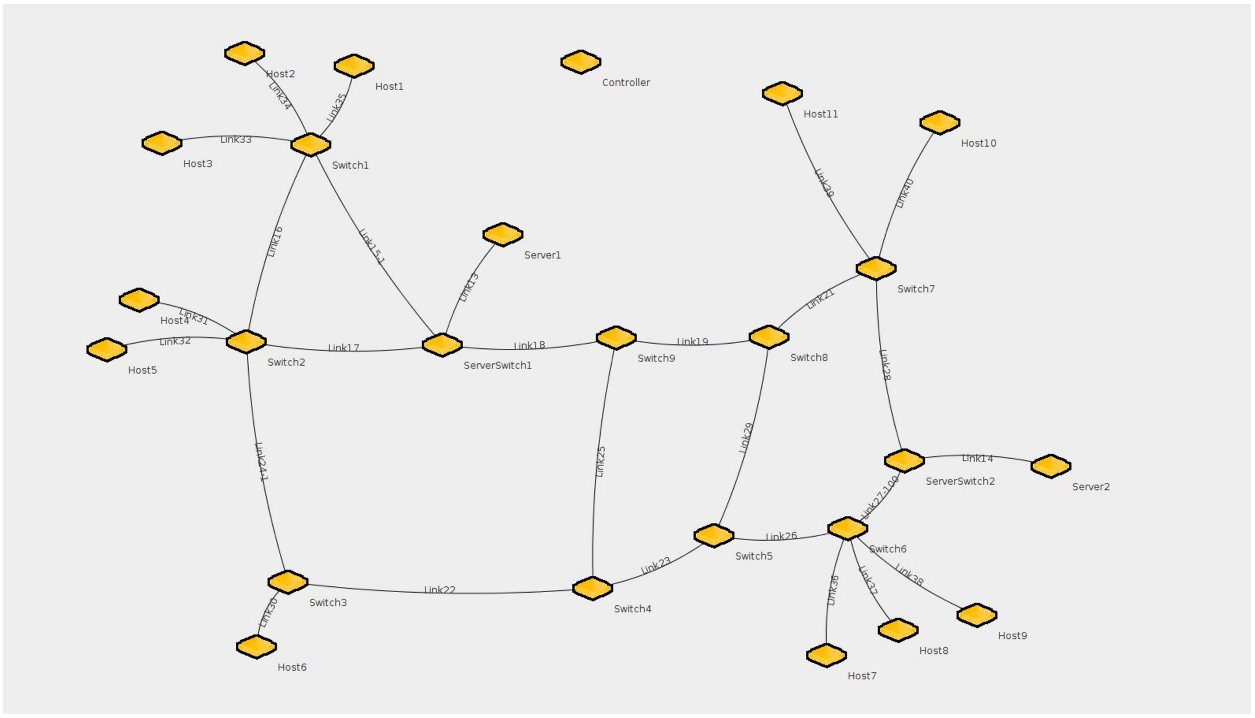


Figure6: Topology

7.1 Platform:

Initially, the platform for the project was intended to be netlabs. However, due to insufficient lab resources, the platform was shifted to GENI. Instead of using limited physical resources available in netlabs, GENI gives the freedom to build bigger topologies by using VMs.

7.2 Controller:

POX was supposed to be the controller to be used as proposed originally. The project was continued using RYU controller since POX is almost obsolete.

7.3 Topology discovery:

A new module was added in component decomposition which is topology discovery. When a new packet comes to the controller, the module discovers the entire topology and stores the properties such as switch and link properties, their utilization and bandwidth in a graph.

7.4 Base Case Description:

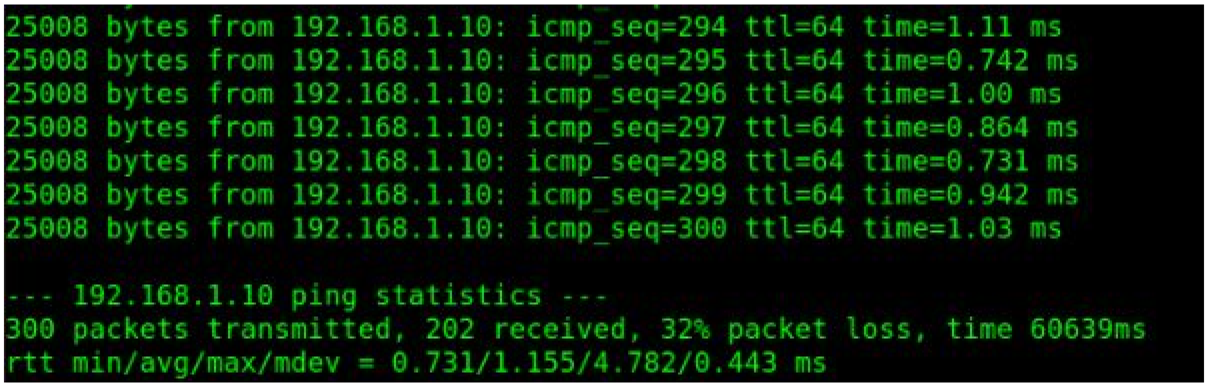
The extended dijkstra's algorithm for load balancing algorithm will be compared with the base case scenario of round robin algorithm. In round robin algorithm, the client requests are forwarded to each servers in turn.For the topology shown, the first client request to Server1. The next client request will be forwarded to Server2 and this will continue. There is no priority for forwarding requests.

The second base case being considered is unit-weighted dijkstra's algorithm. In unit-weighted dijkstra's algorithm, all link weights in dijkstra's algorithm are considered to be one. The unit-weighted Dijkstra's algorithm will return the path with the minimum hop counts for a pair of a source node and a destination node. It will not consider the congestion in the paths.

7.5 Comparison with base cases:

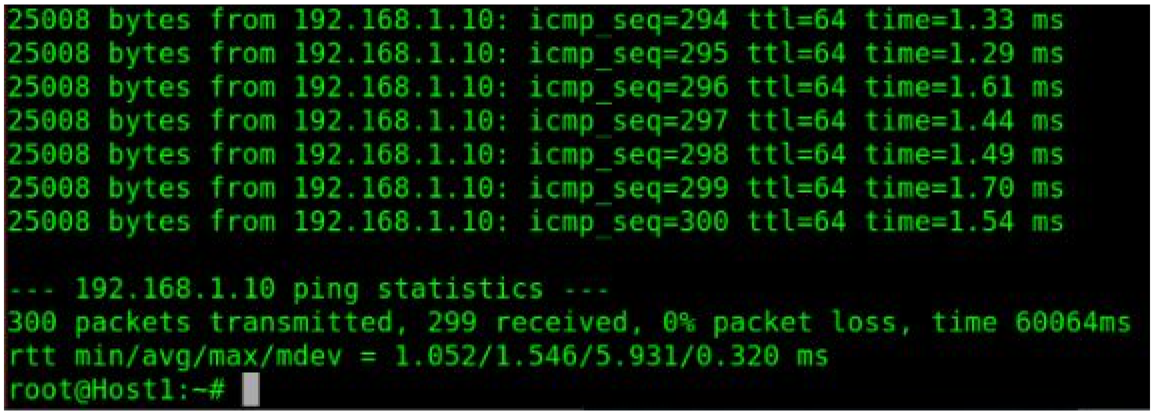
Unit weighted Dijkstra's algorithm:

When host1 pings the server, server1 being the closest is selected. The path selected is Switch1->link15-1->ServerSwitch1. The link15-1 has bandwidth 1Mbps. When a ping command "`ping 192.168.1.10 -s 25000 -i 0.2 -c 300`" is executed , packet loss is observed.



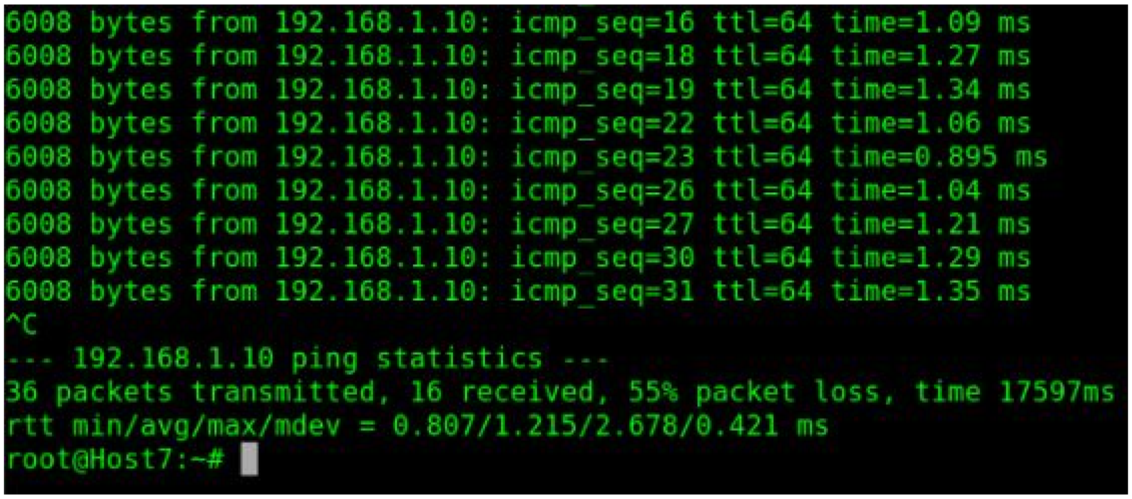
```
25008 bytes from 192.168.1.10: icmp_seq=294 ttl=64 time=1.11 ms
25008 bytes from 192.168.1.10: icmp_seq=295 ttl=64 time=0.742 ms
25008 bytes from 192.168.1.10: icmp_seq=296 ttl=64 time=1.00 ms
25008 bytes from 192.168.1.10: icmp_seq=297 ttl=64 time=0.864 ms
25008 bytes from 192.168.1.10: icmp_seq=298 ttl=64 time=0.731 ms
25008 bytes from 192.168.1.10: icmp_seq=299 ttl=64 time=0.942 ms
25008 bytes from 192.168.1.10: icmp_seq=300 ttl=64 time=1.03 ms

--- 192.168.1.10 ping statistics ---
300 packets transmitted, 202 received, 32% packet loss, time 60639ms
rtt min/avg/max/mdev = 0.731/1.155/4.782/0.443 ms
```

The path selected for Extended Dijkstra's algorithm takes into consideration link and node cost which causes path Switch1->link16->Switch2->link17->ServerSwitch1 to be selected. All links in this path have bandwidth 10Mbps. This is why no packet loss is observed for the same ping command, i.e: more optimum path was selected.



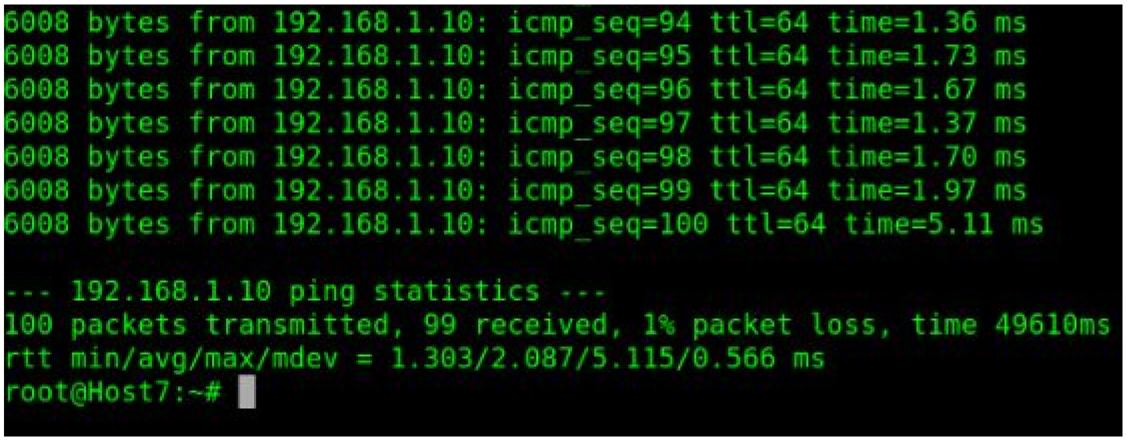
```
25008 bytes from 192.168.1.10: icmp_seq=294 ttl=64 time=1.33 ms
25008 bytes from 192.168.1.10: icmp_seq=295 ttl=64 time=1.29 ms
25008 bytes from 192.168.1.10: icmp_seq=296 ttl=64 time=1.61 ms
25008 bytes from 192.168.1.10: icmp_seq=297 ttl=64 time=1.44 ms
25008 bytes from 192.168.1.10: icmp_seq=298 ttl=64 time=1.49 ms
25008 bytes from 192.168.1.10: icmp_seq=299 ttl=64 time=1.70 ms
25008 bytes from 192.168.1.10: icmp_seq=300 ttl=64 time=1.54 ms

--- 192.168.1.10 ping statistics ---
300 packets transmitted, 299 received, 0% packet loss, time 60064ms
rtt min/avg/max/mdev = 1.052/1.546/5.931/0.320 ms
root@Host1:~# 
```

When host7 pings the server, server2 being the closest is selected. The path selected is Switch6->link27-100->ServerSwitch2. The link27-100 has bandwidth 100kbps. When a ping command "`ping 192.168.1.10 -s 6000 -i 0.5 -c 100`" is implemented, packet loss is observed.



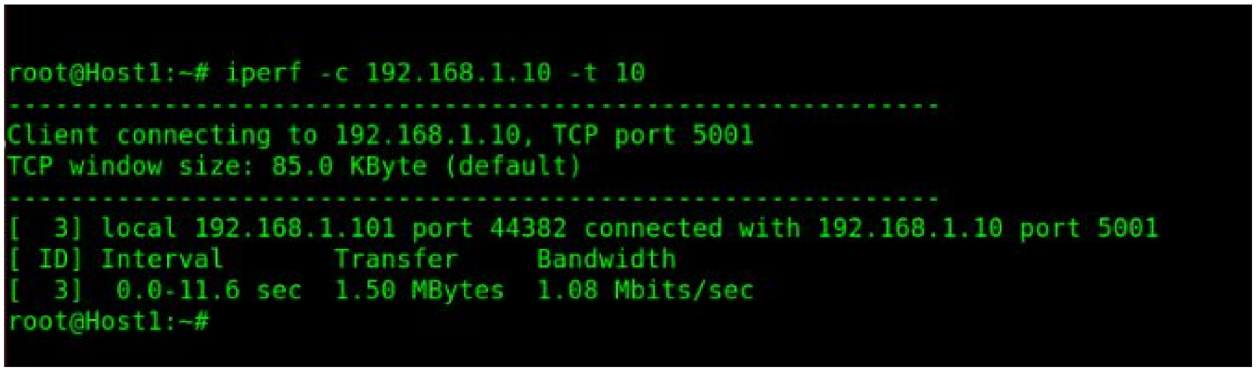
```
6008 bytes from 192.168.1.10: icmp_seq=16 ttl=64 time=1.09 ms
6008 bytes from 192.168.1.10: icmp_seq=18 ttl=64 time=1.27 ms
6008 bytes from 192.168.1.10: icmp_seq=19 ttl=64 time=1.34 ms
6008 bytes from 192.168.1.10: icmp_seq=22 ttl=64 time=1.06 ms
6008 bytes from 192.168.1.10: icmp_seq=23 ttl=64 time=0.895 ms
6008 bytes from 192.168.1.10: icmp_seq=26 ttl=64 time=1.04 ms
6008 bytes from 192.168.1.10: icmp_seq=27 ttl=64 time=1.21 ms
6008 bytes from 192.168.1.10: icmp_seq=30 ttl=64 time=1.29 ms
6008 bytes from 192.168.1.10: icmp_seq=31 ttl=64 time=1.35 ms
^C
--- 192.168.1.10 ping statistics ---
36 packets transmitted, 16 received, 55% packet loss, time 17597ms
rtt min/avg/max/mdev = 0.807/1.215/2.678/0.421 ms
root@Host7:~#
```

The path selected for Extended Dijkstra's algorithm takes into consideration link and node cost which causes path Switch6->link26->Switch5->link29->Switch8->link21->Switch7->link28->ServerSwitch2 to be selected. All links in this path have bandwidth 10Mbps. This is why no packet loss is observed for the same ping command, i.e: more optimum path was selected.



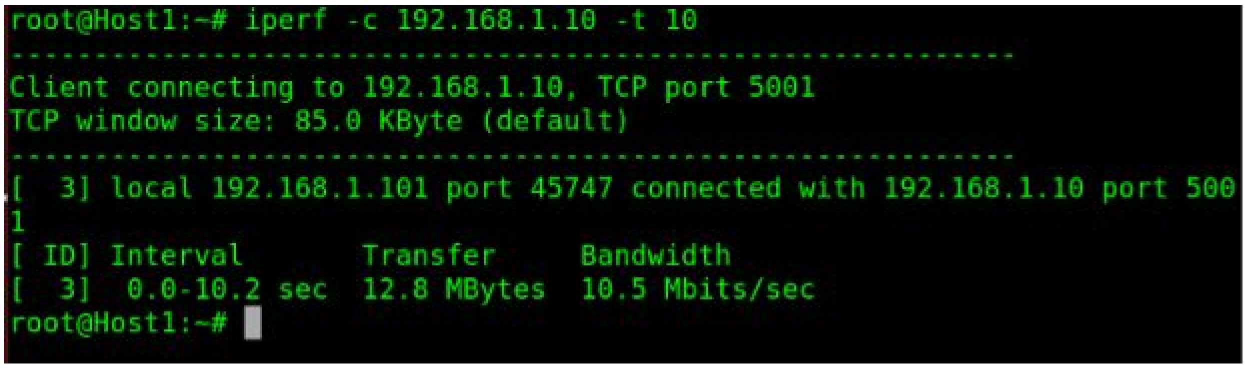
```
6008 bytes from 192.168.1.10: icmp_seq=94 ttl=64 time=1.36 ms
6008 bytes from 192.168.1.10: icmp_seq=95 ttl=64 time=1.73 ms
6008 bytes from 192.168.1.10: icmp_seq=96 ttl=64 time=1.67 ms
6008 bytes from 192.168.1.10: icmp_seq=97 ttl=64 time=1.37 ms
6008 bytes from 192.168.1.10: icmp_seq=98 ttl=64 time=1.70 ms
6008 bytes from 192.168.1.10: icmp_seq=99 ttl=64 time=1.97 ms
6008 bytes from 192.168.1.10: icmp_seq=100 ttl=64 time=5.11 ms

--- 192.168.1.10 ping statistics ---
100 packets transmitted, 99 received, 1% packet loss, time 49610ms
rtt min/avg/max/mdev = 1.303/2.087/5.115/0.566 ms
root@Host7:~#
```

In unit-weighted Dijkstra's algorithm, both the servers are kept in server mode using iperf. Set Host1 in client mode using the command "`iperf -c 192.168.1.10 -t 10`". The path selected is Switch1->link15-1->ServerSwitch1.The output gives the bandwidth as 1.08Mbps. This is because link15-1 has bandwidth 1Mbps.



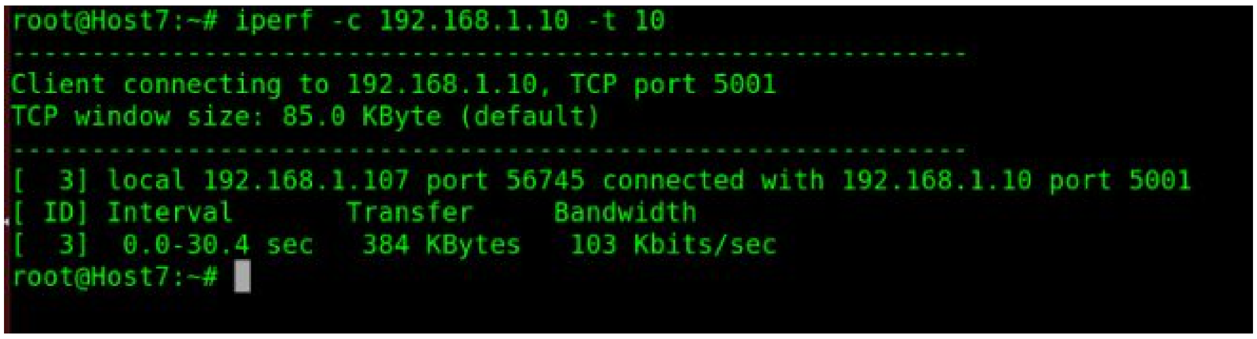
```
root@Host1:~# iperf -c 192.168.1.10 -t 10
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.101 port 44382 connected with 192.168.1.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-11.6 sec  1.50 MBytes  1.08 Mbits/sec
root@Host1:~#
```

For Extended Dijkstra's algorithm, the same procedure is repeated. Switch1->link16->Switch2->link17->ServerSwitch1. The bandwidth is observed to be 10.5Mbps. This is because all links in the path selected have bandwidth 10Mbps.



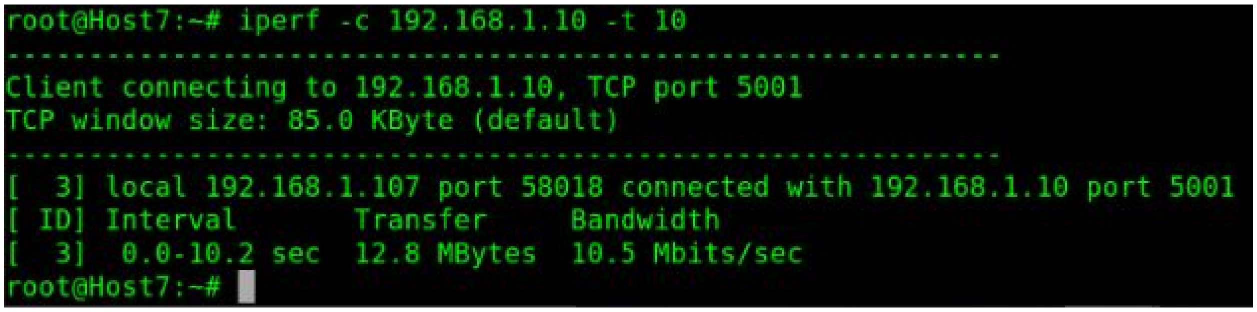
```
root@Host1:~# iperf -c 192.168.1.10 -t 10
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.101 port 45747 connected with 192.168.1.10 port 500
1
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.2 sec  12.8 MBytes  10.5 Mbits/sec
root@Host1:~#
```

In unit-weighted Dijkstra's algorithm, both the servers are kept in server mode using iperf. Set Host7 in client mode using the command "iperf -c 192.168.1.10 -t 10". The path selected is Switch6->link27-100->ServerSwitch2. The output gives the bandwidth as 103kbps. This is because link27-100 has bandwidth 100kbps.



```
root@Host7:~# iperf -c 192.168.1.10 -t 10
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.107 port 56745 connected with 192.168.1.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.4 sec   384 KBytes   103 Kbits/sec
root@Host7:~#
```

For Extended Dijkstra's algorithm, the same procedure is repeated. The path selected is Switch6->link26->Switch5->link29->Switch8->link21->Switch7->link28->ServerSwitch2. The bandwidth is observed to be 10.5Mbps. This is because all links in the path selected have bandwidth 10Mbps.



```
root@Host7:~# iperf -c 192.168.1.10 -t 10
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.107 port 58018 connected with 192.168.1.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.2 sec  12.8 MBytes  10.5 Mbits/sec
root@Host7:~# 
```

A comparison between Extended Dijkstra's Algorithm and Unit Weighted Dijkstra's Algorithm is shown in the graph below. The graph has number of hosts on x axis and bandwidth achieved on y axis. As seen, the bandwidth achieved using Extended Weighted Dijkstra's algorithm is more than that for Unit Weighted Dijkstra's Algorithm.

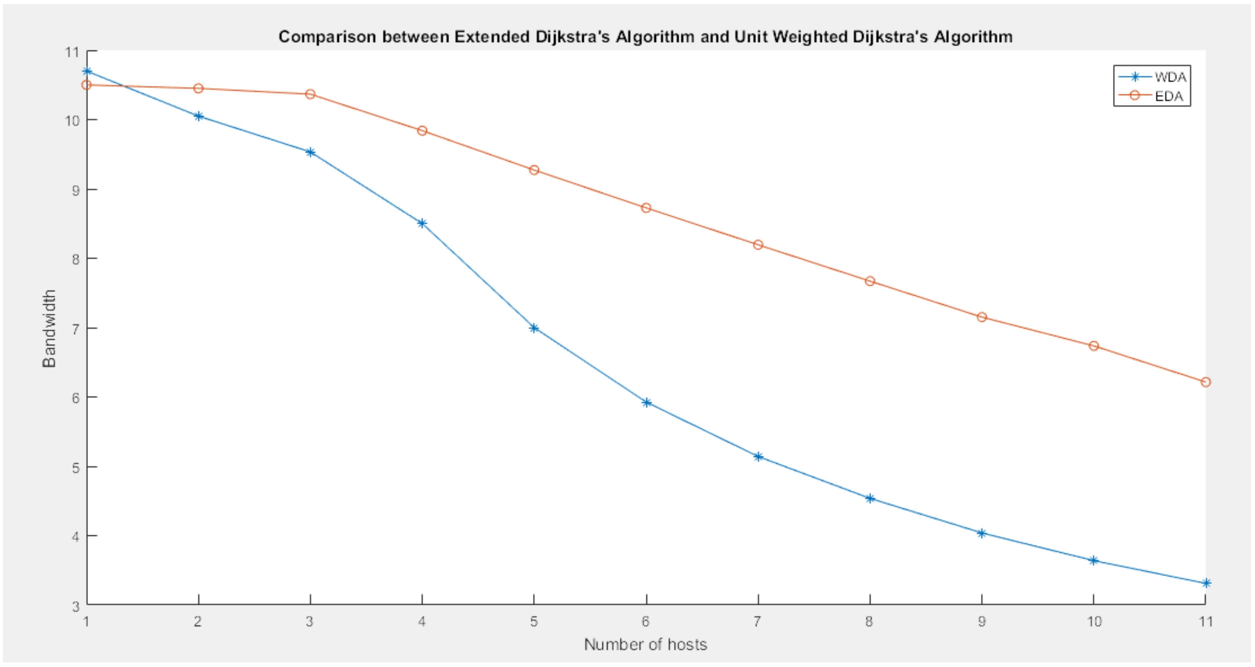
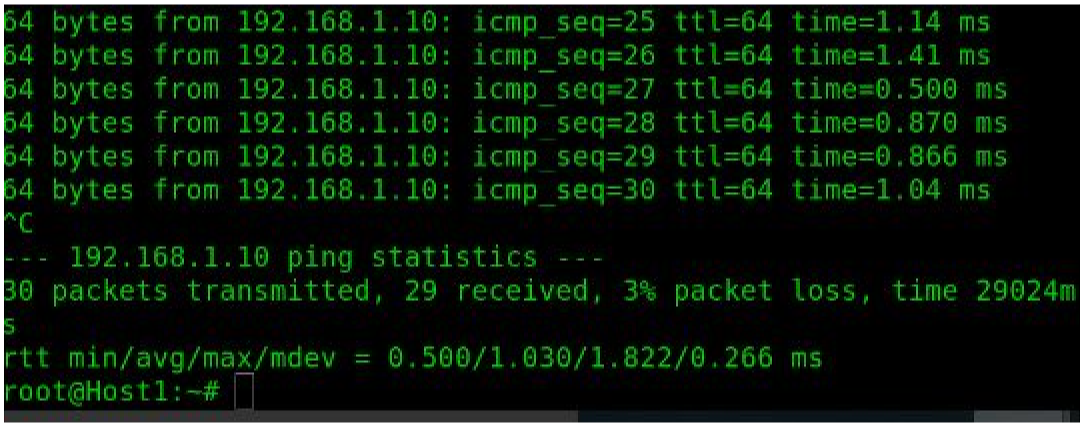


Figure7: Comparison of Bandwidth for Extended Dijkstra's Algorithm and Unit-Weighted Dijkstra's Algorithm

Round-robin Algorithm:

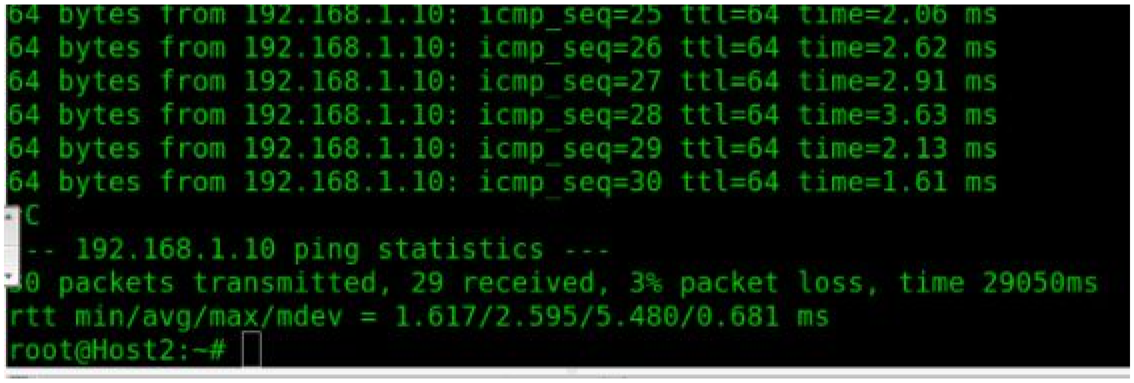
When Host1 pings the server, server1 was selected. The average round-trip time was observed to be 1.03ms.



```
64 bytes from 192.168.1.10: icmp_seq=25 ttl=64 time=1.14 ms
64 bytes from 192.168.1.10: icmp_seq=26 ttl=64 time=1.41 ms
64 bytes from 192.168.1.10: icmp_seq=27 ttl=64 time=0.500 ms
64 bytes from 192.168.1.10: icmp_seq=28 ttl=64 time=0.870 ms
64 bytes from 192.168.1.10: icmp_seq=29 ttl=64 time=0.866 ms
64 bytes from 192.168.1.10: icmp_seq=30 ttl=64 time=1.04 ms
^C
--- 192.168.1.10 ping statistics ---
30 packets transmitted, 29 received, 3% packet loss, time 29024m
s
rtt min/avg/max/mdev = 0.500/1.030/1.822/0.266 ms
root@Host1:~# 
```

Host1 pings Server1

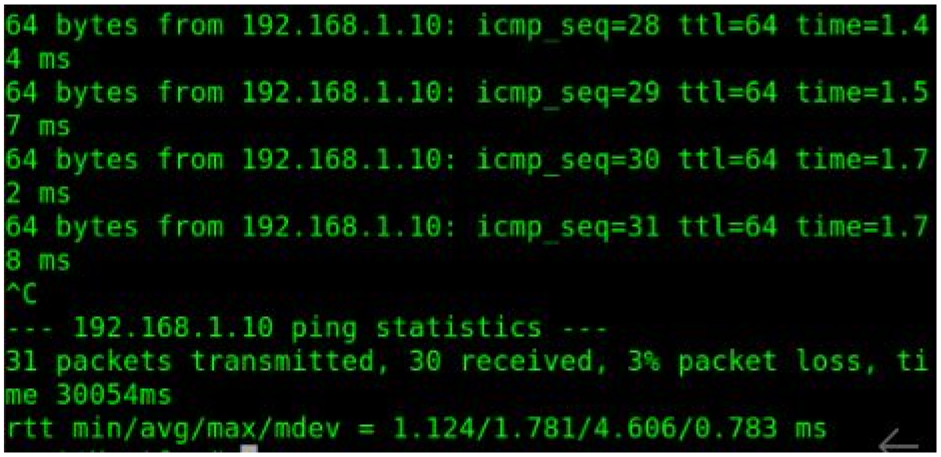
When Host2 pings the server, server2 was selected. The average round-trip time was observed to have increased to 2.595ms.



```
64 bytes from 192.168.1.10: icmp_seq=25 ttl=64 time=2.06 ms
64 bytes from 192.168.1.10: icmp_seq=26 ttl=64 time=2.62 ms
64 bytes from 192.168.1.10: icmp_seq=27 ttl=64 time=2.91 ms
64 bytes from 192.168.1.10: icmp_seq=28 ttl=64 time=3.63 ms
64 bytes from 192.168.1.10: icmp_seq=29 ttl=64 time=2.13 ms
64 bytes from 192.168.1.10: icmp_seq=30 ttl=64 time=1.61 ms
^C
--- 192.168.1.10 ping statistics ---
30 packets transmitted, 29 received, 3% packet loss, time 29050ms
rtt min/avg/max/mdev = 1.617/2.595/5.480/0.681 ms
root@Host2:~# 
```

Host2 pings Server2

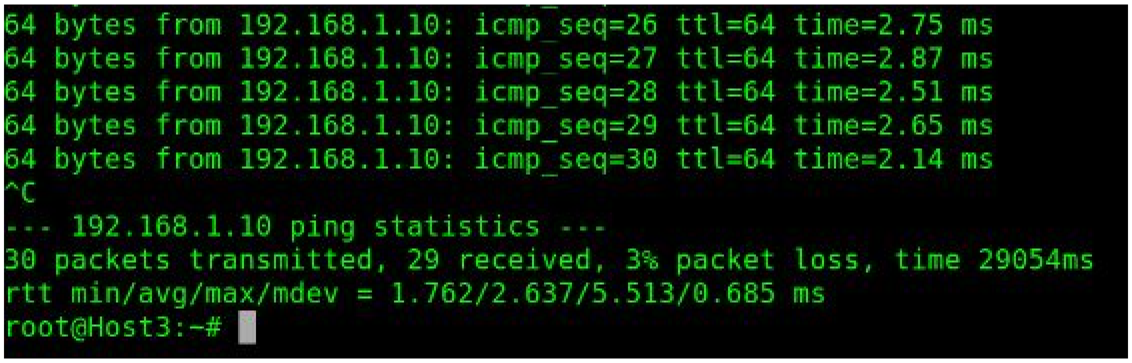
When Host6 pings the server, server1 was selected. The average round-trip time was observed to be 1.781ms.



```
64 bytes from 192.168.1.10: icmp_seq=28 ttl=64 time=1.4
4 ms
64 bytes from 192.168.1.10: icmp_seq=29 ttl=64 time=1.5
7 ms
64 bytes from 192.168.1.10: icmp_seq=30 ttl=64 time=1.7
2 ms
64 bytes from 192.168.1.10: icmp_seq=31 ttl=64 time=1.7
8 ms
^C
--- 192.168.1.10 ping statistics ---
31 packets transmitted, 30 received, 3% packet loss, ti
me 30054ms
rtt min/avg/max/mdev = 1.124/1.781/4.606/0.783 ms
```

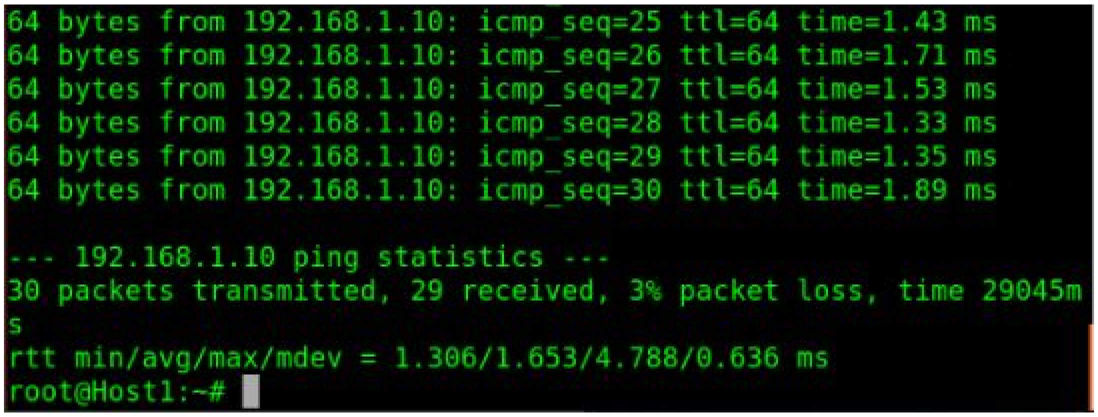
Host6 pings Server 1

When Host3 pings the server, server2 was selected. The average round-trip time was observed to have increased to 2.637ms.

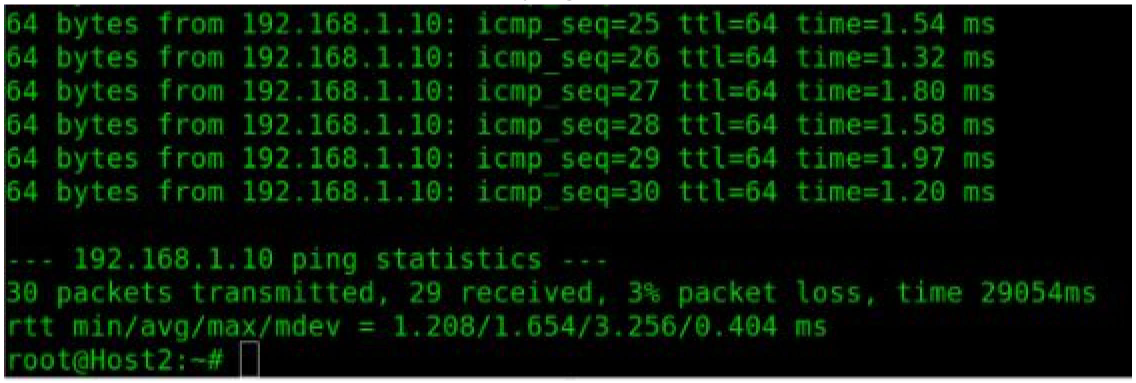

```
64 bytes from 192.168.1.10: icmp_seq=26 ttl=64 time=2.75 ms
64 bytes from 192.168.1.10: icmp_seq=27 ttl=64 time=2.87 ms
64 bytes from 192.168.1.10: icmp_seq=28 ttl=64 time=2.51 ms
64 bytes from 192.168.1.10: icmp_seq=29 ttl=64 time=2.65 ms
64 bytes from 192.168.1.10: icmp_seq=30 ttl=64 time=2.14 ms
^C
--- 192.168.1.10 ping statistics ---
30 packets transmitted, 29 received, 3% packet loss, time 29054ms
rtt min/avg/max/mdev = 1.762/2.637/5.513/0.685 ms
root@Host3:~#
```

Host3 pings Server2`

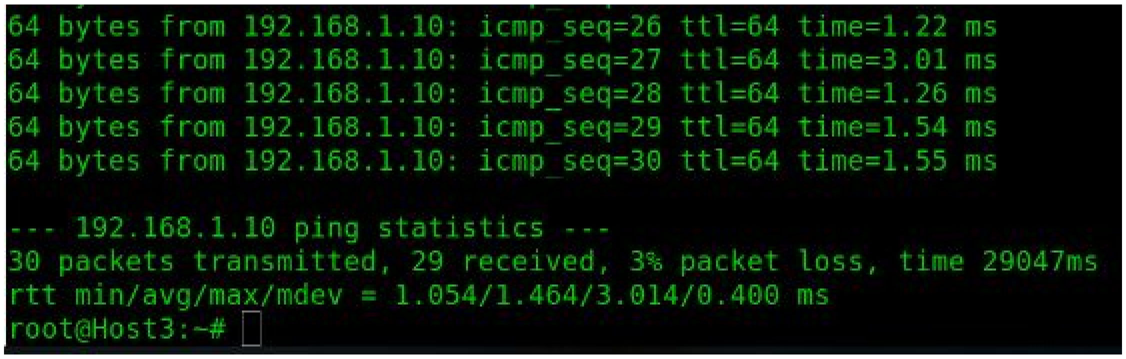
For Extended Dijkstra's Algorithm,
Since all the 3 hosts, Host1, Host2 and Host3 communicate with Server1 based on the shortest calculated path with server below threshold, all average round trip times are observed to be less than that for round robin algorithm. The round trip times are observed to be 1.653ms, 1.654ms and 1.464ms for Host1, Host2 and Host3 respectively.


```
64 bytes from 192.168.1.10: icmp_seq=25 ttl=64 time=1.43 ms
64 bytes from 192.168.1.10: icmp_seq=26 ttl=64 time=1.71 ms
64 bytes from 192.168.1.10: icmp_seq=27 ttl=64 time=1.53 ms
64 bytes from 192.168.1.10: icmp_seq=28 ttl=64 time=1.33 ms
64 bytes from 192.168.1.10: icmp_seq=29 ttl=64 time=1.35 ms
64 bytes from 192.168.1.10: icmp_seq=30 ttl=64 time=1.89 ms

--- 192.168.1.10 ping statistics ---
30 packets transmitted, 29 received, 3% packet loss, time 29045m
s
rtt min/avg/max/mdev = 1.306/1.653/4.788/0.636 ms
root@Host1:~#
```

Host1 pings Server1


```
64 bytes from 192.168.1.10: icmp_seq=25 ttl=64 time=1.54 ms
64 bytes from 192.168.1.10: icmp_seq=26 ttl=64 time=1.32 ms
64 bytes from 192.168.1.10: icmp_seq=27 ttl=64 time=1.80 ms
64 bytes from 192.168.1.10: icmp_seq=28 ttl=64 time=1.58 ms
64 bytes from 192.168.1.10: icmp_seq=29 ttl=64 time=1.97 ms
64 bytes from 192.168.1.10: icmp_seq=30 ttl=64 time=1.20 ms

--- 192.168.1.10 ping statistics ---
30 packets transmitted, 29 received, 3% packet loss, time 29054ms
rtt min/avg/max/mdev = 1.208/1.654/3.256/0.404 ms
root@Host2:~#
```

Host2 pings Server1



```
64 bytes from 192.168.1.10: icmp_seq=26 ttl=64 time=1.22 ms
64 bytes from 192.168.1.10: icmp_seq=27 ttl=64 time=3.01 ms
64 bytes from 192.168.1.10: icmp_seq=28 ttl=64 time=1.26 ms
64 bytes from 192.168.1.10: icmp_seq=29 ttl=64 time=1.54 ms
64 bytes from 192.168.1.10: icmp_seq=30 ttl=64 time=1.55 ms

--- 192.168.1.10 ping statistics ---
30 packets transmitted, 29 received, 3% packet loss, time 29047ms
rtt min/avg/max/mdev = 1.054/1.464/3.014/0.400 ms
root@Host3:~# 
```

Host3 pings Server1

A comparison graph between average round trip times for Extended Dijkstra's Algorithm and Round Robin Algorithm is shown below.

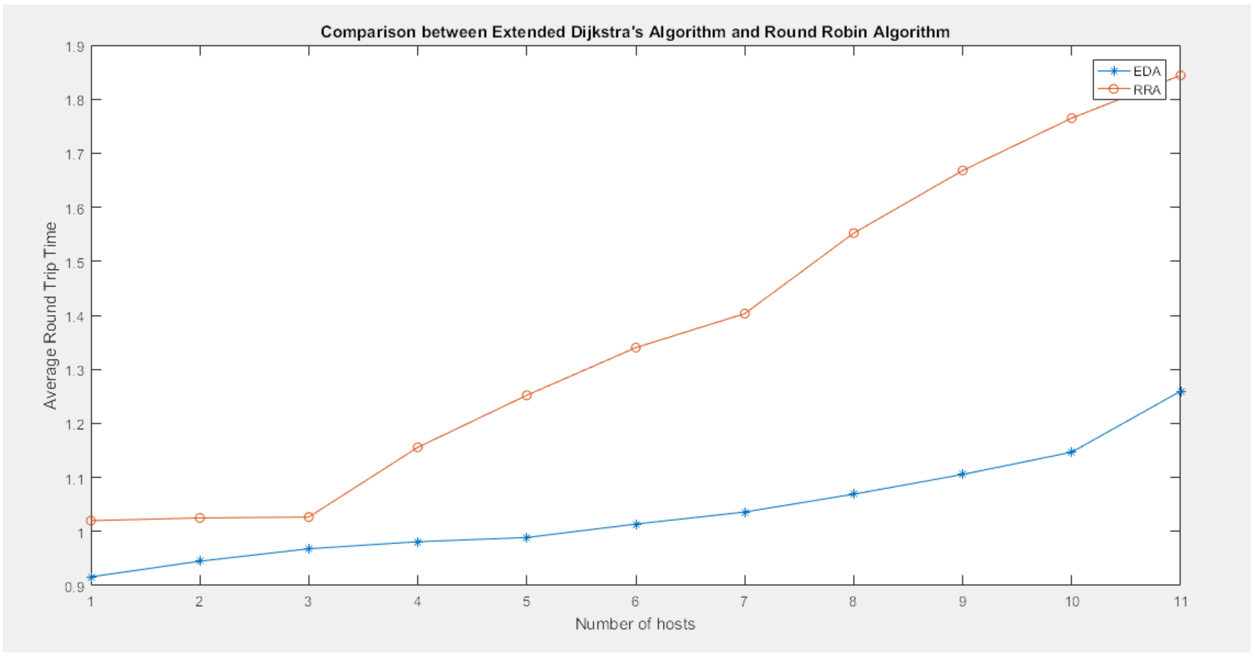


Figure8: Comparison of average round trip time for Extended Dijkstra's Algorithm and Round Robin Algorithm

The Extended Dijkstra's Algorithm gives a better average round trip time than Round Robin Algorithm.

8. Reflection

The Extended Dijkstra's algorithm forms the fundamental component of this project. We spent a substantial portion of time deploying and trying out various approaches to select shortest path between a client and server on Openflow network using GENI platform. We read, discussed and referred to Openflow Specification 1.3 documentation, Ryu Controller documentation and some below mentioned reference papers while coming up with the development plan. The Rest API document of ryu controller explained the implementations

and functionality regarding detection of switches ,links,flow modifications and enabled us to come up with the framework for our modified components in the controller.
Nodes used in this project run on virtual Linux machines. This was achieved by using ExoGENI. A thorough understanding of how openflow works on Linux kernel was essential. For this we studied the openflow installation guide. We also extensively analyzed the compared our basecase approaches using Unit weighted Algorithm and Round-robin algorithm with our extended dijkstra's algorithm approach .
We also learned about various traffic generators available in GENI like tmix ubuntu based Iperf . Scapy was very easy and convenient to use and GENI tmix provided a very detailed GUI to configure traffic streams. After a long debate, we finalized Iperf as our traffic generator for this project as it appeared to be the 'best of both worlds'. We also utilized "tcpdump" output packet streams on the receiver node.

Lastly, for strengthening our fundamental concepts about Internet Protocols and Networks, we referred the book "TCP/IP Illustrated: The Protocols" by W. Richard Stevens. This book helped us resolve our general doubts and queries concerning network related configuration and commands on linux distribution environments.

8.1 Per-member learning experiences

Bhushan:

Working on the project was a great learning for me and I improved my knowledge on the things mentioned below:

1. Network Architecture
2. OpenFlow actual working and the significance of Controller and Switches in OpenFlow
3. Topology Analysis (information that is attached with each switch and link in OpenFlow)
4. Python Coding : I worked on the language for the first time but it was a great learning experience to build the complete project from scratch in a completely new coding language
5. Network Parameter Analysis (various ways to find end-to-end latency and packet loss)
6. Networking functionality available in Python(eg. Networkx for graph creation)
7. Traffic Analysis (ability to predict the path that a connection will establish based on the algorithm used for selection of path)
8. Implementation and use of threads
9. Analysis of datagram received on OpenFlow Controller
10. Various events that can be triggered in OpenFlow (eg. PacketIn Event, Switch Features Event)

Divya :

Through this project i got exposed to SDN technology, especially learned a lot about OpenFlow, studied and analysed about different openflow controllers like POX Controller,RYU controller,Big Switch network controller. Along with them I gained knowledge about :

1. OpenFlow architecture
2. Functioning of flow pipeline processing
3. Implementation of openflow network using ExoGENI

4. Usage of RYU REST applications and their implementation
5. How to implement different algorithms on the network using python programming.
6. Usage of different network monitoring tools like tmix, iperf,tcpdump etc.
7. Various network based linux commands and configurations
8. How a network should be designed to achieve better performance

Rishabh:

The project provided a great insight on how to work on OpenFlow based networks with the help of ExoGENI that can be used to build almost any network topology desired. The things I will take away from this experience are:

1. Functioning of OpenVSwitch and OpenFlow Controllers.
2. Ability to install and modify existing flows dynamically through controller APIs to direct network traffic successfully.
3. How to make use of virtual IP and virtual MAC address and implement server load balancing .
4. Analysis of network traffic statistics and using it meaningfully to calculate costs and find the best cost path over the network.
5. Python Coding to call RYU controller APIs and catch events to trigger specific code.
6. I learned the importance of interface level network troubleshooting and captured traffic analysis to find exactly where exactly is the problem in the packet forwarding that is causing unsuccessful communication.
7. Analyse a network topology and discover its components in a dynamic environment.

Prachi:

During the course of the project, I learnt a lot of new and interesting things about how networking actually works.

1. I learnt that softwares like ExoGENI can help us create complex topologies to study how a particular algorithm will work. It was a great experience to actually watch the flow of packets through switches and it helped strengthen my concepts.
2. How OpenFlow works and the importance of OpenVSwitches and OpenFlow controllers.
3. The difference between an algorithm for a static network and an algorithm for a dynamic network.
4. How to implement algorithms for a dynamic environment.
5. Python Coding

8.2 What went as expected

Logically, we implemented our idea exactly as we proposed in most of the cases. We designed and implemented proposed components as mentioned previously. After implementation of base case and in our approach we got the results according to our expectations i.e our approach improved average end-to-end latency, reduced packet loss when analysed with round-robin and unit-weighted based server load balancing algorithms.

8.3 What went different from expected

Initially we were assigned to NetLabs but as we can not accommodate our topology in it, we had to shift to ExoGENI platform. We considered using POX controller to control our openflow network but it is deprecated and does not support some of our requirements so, we replaced it with Ryu controller.
Initially, we thought the implementation would be easy but a lot of problems arose during the actual implementation from the problem in topology discovery because of LLDP packets to the arp flooding due to loop in the topology but in the end it was a great learning experience.

9. References

1. Extending Dijkstra's Shortest Path Algorithm for Software Defined Networking: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6996609
2. GENI: http://www.exogeni.net/
3. OpenFlow: https://www.opennetworking.org/sdn-resources/openflow
4. Open vSwitch: https://github.com/openvswitch/ovs
5. RYU: https://osrg.github.io/ryu/
6. REST API Ryu: http://ryu.readthedocs.io/en/latest/app/ofctl_rest.html
7. RYU installation Guide : https://github.com/sdnds-tw/ryuInstallHelper
8. OpenVSwitch installation guide : https://www.nanog.org/meetings/nanog57/presentations/Monday/mon.tutorial.SmallWallace.OpenFlow.24.pdf
9. Python tutorial : http://www.python-course.eu/course.php
10. RYU Guide : http://www.pica8.com/document/v2.6/html/openflow-tutorials/
11. RYU REST GUIDE: http://ryu.readthedocs.io/en/latest/app/ofctl_rest.html
12. SDN GUIDE : http://sdnhub.org/tutorials/ryu/
13. OPENFLOW Information: http://archive.openflow.org/wp/learnmore/
14. Networkx GUIDE: https://networkx.github.io/documentation/networkx-1.10/tutorial/index.html
15. Python Threading: https://docs.python.org/2/library/threading.html