ADVANCED COMPUTER NETWORKS

MULTIPATH LOAD BALANCING
FOR SDN

AUTHOR

DHARSHAN KUMAR K S

TABLE OF CONTENTS

ABSTRACT

This is a report based on Multipath Load Balancing for SDN. The project begins with a brief introduction towards the Load Balancing.

We had discussed mainly about:

- Load Balancing
- Path Finding
- Installing Flow
- SDN Demonstration

The main idea of Load Balancing is a method of managing incoming traffic by distributing and sharing the load fairly.

They improve the capacity of links and the overall performance of applications by decreasing the burden on servers. They are usually done using middleboxes in traditional routing, but in SDN, we just use ‘match+action’ to balance loads.

Why to use Load Balancing?

- Reduce network response time
- Network Utility & Make use of idle hosts
- Harder to sniff packets gives Network security
- Increases Bandwidth due to parallel transfer

I have uploaded the multipath code in my GitHub repository, for reference of full code, you can access it using this link:

https://github.com/dharshankumar2002/Multipath-Load-Balancing/blob/main/ryu_multipath.py

RESEARCH ARTICLES

Multipath Load Balancing is important for sending packets through the network in real-time. So, there are numerous research papers available which was written by many scholars.

Few of those notable research papers are mentioned below:

1. IMPROVING LOAD BALANCING WITH MULTIPATH ROUTING
   Date published: September 2008
   DOI: 10.1109/ICCCN.2008.ECP.30
   Authored by: P. Merindol; Jean-Jacques Pansiot
   Published by: Computer Communications and Networks, 2008. ICCCN
   Objective of paper: Initially developed the Core Multipath Load balancing part
   (Oldest paper for Multipath Load Balancing)

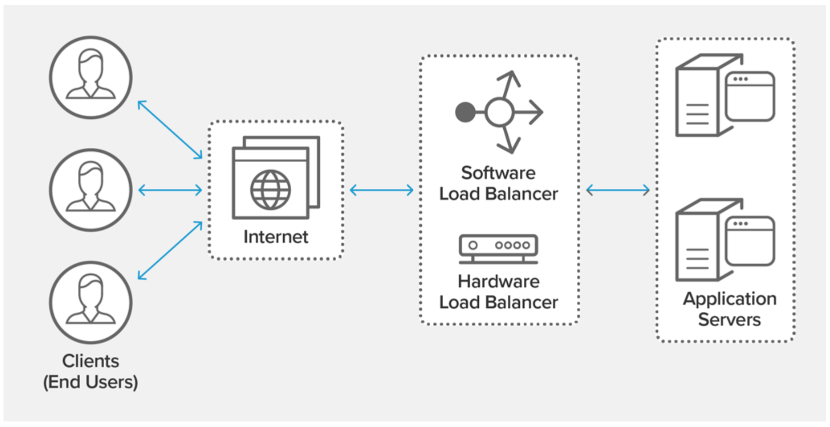
2. A Survey of Multipath Load Balancing Based on Network Stochastic Model in MANETDate published: March 2021
   DOI: 10.23919/ICACT51234.2021.9370843
   Authored by: Zhang Hui; Zhang Lingli
   Published by: International Conference on Advanced Communication Technology (ICACT)
   Objective of paper: To use load balancing in Mobile Ad-Hoc mode
   (Most Latest paper for Multipath Load Balancing)

INTRODUCTION

Load Balancing:

Load balancing refers to efficiently distributing incoming network traffic across a group of hosts.

Modern high-traffic websites must serve hundreds of thousands, if not millions, of concurrent requests from users or clients and return the correct text, images, video, or application data, all in a fast and reliable manner. To cost-effectively scale to meet these high volumes, modern computing best practice generally requires adding more servers.



Load balancers manage the flow of information between the server and an endpoint device (PC, laptop, tablet or smartphone). The server could be on-premises, in a data center or the public cloud. The server can also be physical or virtualized. The load balancer helps servers move data efficiently, optimizes the use of application delivery resources and prevents server overloads. Load balancers conduct continuous health checks on servers to ensure they can handle requests. If necessary, the load balancer removes unhealthy servers from the pool until they are restored. Some load balancers even trigger the creation of new virtualized application servers to cope with increased demand.

Traditionally, load balancers consist of a hardware appliance. Yet they are increasingly becoming software-defined. This is why load balancers are an essential part of an organization's digital strategy.

Multipath Loading:

Method of managing incoming traffic by distributing and sharing load fairly among multiple routes from source to destination hosts.

In a network, if there are multiple hosts available between the source host and the destination host, then we can split the traffic of the packets into multiple hosts. Usually, the packets will be transferred only to 'n' number of optimal paths and not to all the paths.

Types of Load Balancers:

There is a variety of load balancing methods, which use different algorithms best suited for a particular situation.

1) Least Connection Method:
   It Directs traffic to the server with the fewest active connections. Most useful when there are a large number of persistent connections in the traffic unevenly distributed between the servers.

2) Least Response Time Method:
   It directs traffic to the server with the fewest active connections and the lowest average response time.

3) Round Robin Method:
   It rotates servers by directing traffic to the first available server and then moves that server to the bottom of the queue. Most useful when servers are of equal specification and there are not many persistent connections.

4) IP Hash:
   The IP address of the client determines which server receives the request.

Load balancing has become a necessity as applications become more complex, user demand grows and traffic volume increases. Load balancers allow organizations to build flexible networks that can meet new challenges without compromising security, service or performance.

LOAD BALANCING IN SDN

Network Virtualization:

Instead of middlebox/hardware, we can implement in software through program codes

SDN controller have built-in load balancers

OpenFlow protocol v1.1 support Group Tables

Apply multiple actions to a specific flow

All – Multicast

Select – Load sharing

Indirect – Indirection

Fast Failover – Rerouting

Advantages of Load Balancing:

- Reduce network response time
- Makes use of Network Utility & Make use of idle hosts
- Harder to sniff packets. So, it gives Network security

  (Since we don't know which 2 links carry the packets)
- Increases Bandwidth due to parallel transfer of packets.

Disadvantage:

- Resources will get wasted if proper path finding algorithm is not found. The packets in one path might reach the destination host very late than the packets from another path.

- No native failure detection or fault tolerance and no dynamic load re-balancing.

Protocol Used:

Messages are transferred using TCP protocol

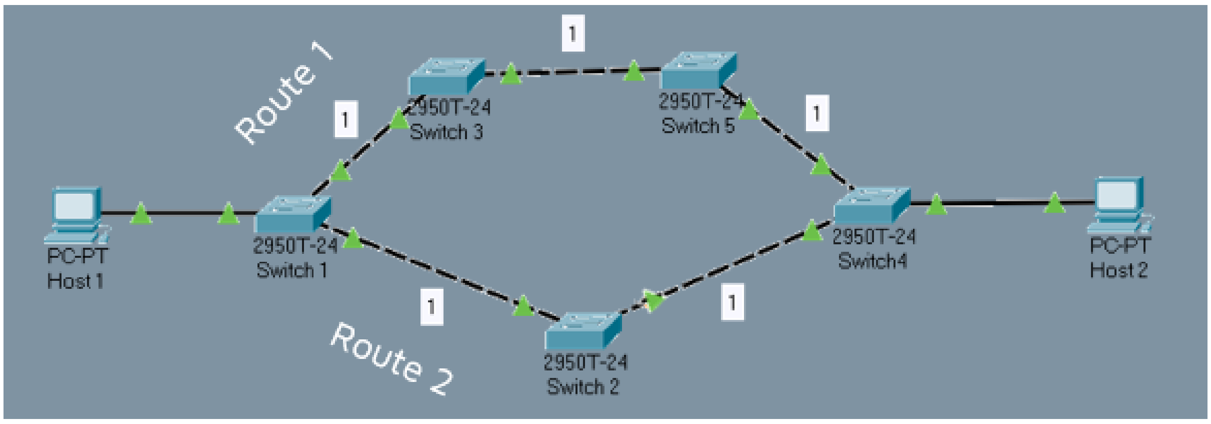
Practical example for application of Load balancing:

From MS Teams application, video packets split and travel across multiple routes. But at the destination router, in the transport layer, it again reassembles and transmits to the destination host.

PATH FINDING USING DFS

We considered few networks, which are:

Topology-1:



Source- host1

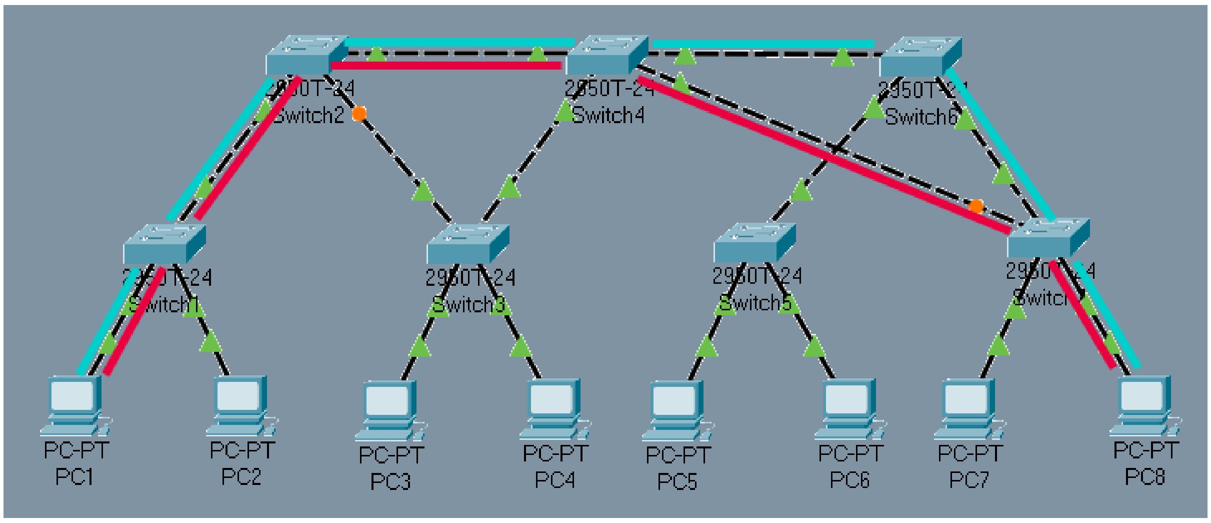
Destination- host2

<u>Paths available:</u>

Path1 – {S1-S2-S4}

Path2 – {S1-S3-S5-S4}

Topology-2:



Source- host1

Destination- host8

<u>Paths available:</u>

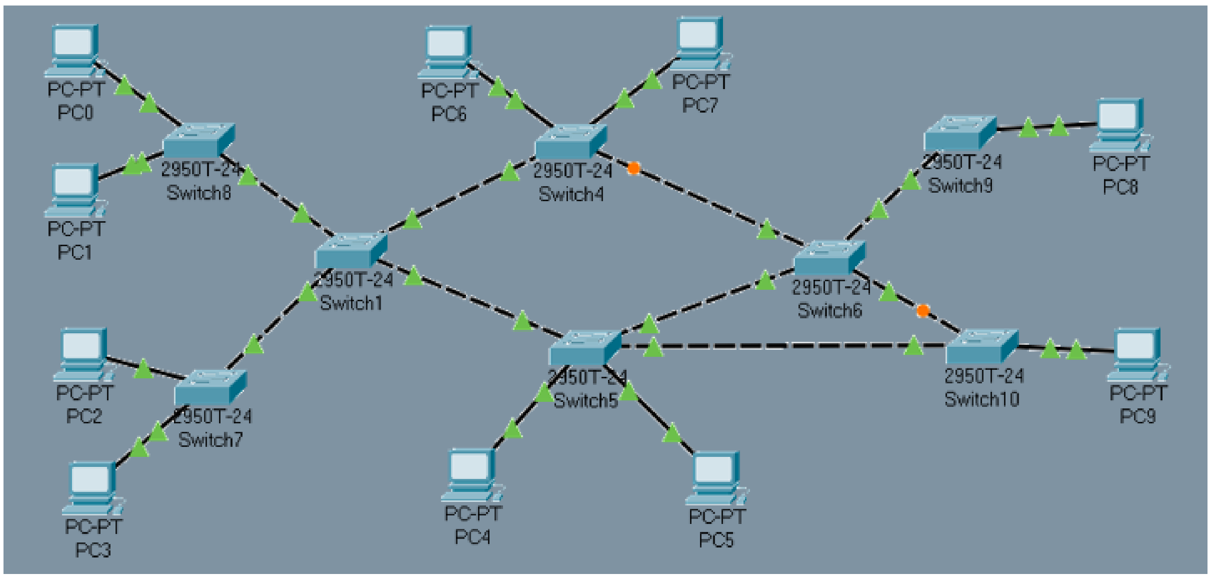
Path1 – {S1-S2-S4-S7}

Path2 – {S1-S2-S4-S6-S7}

Path3 – {S1-S2-S3-S4-S7}

Path4 – {S1-S2-S3-S4-S6-S7}

Topology-3:



Source- host1

Destination- host8

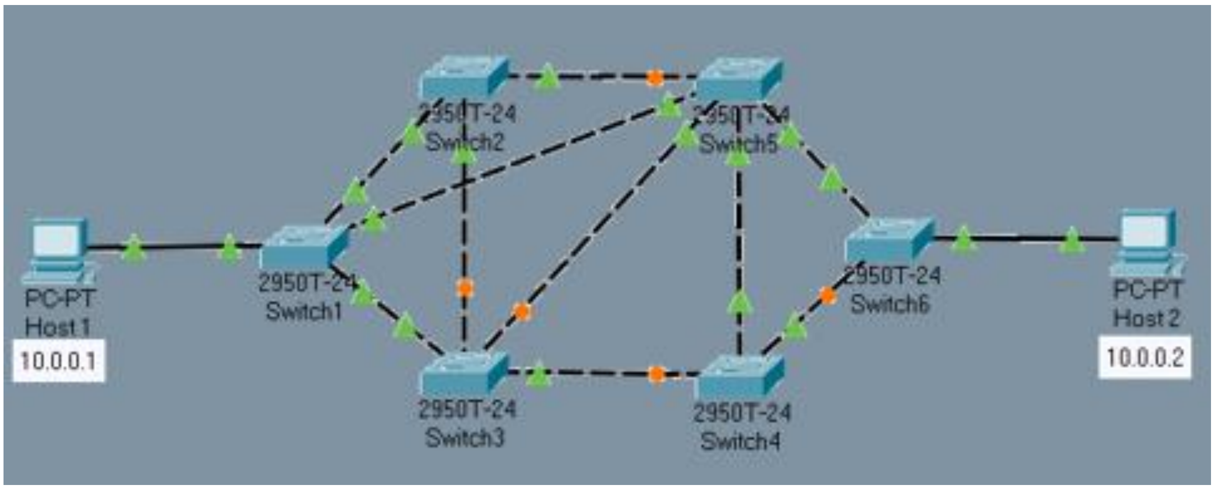
<u>Paths available:</u>

Path1 – {S8-S1-S4-S6-S9}

Path2 – {S8-S1-S5-S6-S9}

Path3 – {S8-S1-S5-S10-S6-S9}

Topology-4:



Source- host1

Destination- host2

<u>Paths available:</u>

Path1 – {S1-S5-S6}

Path2 – {S1-S5-S4-S6}

Path3 – {S1-S5-S3-S4-S6}

Path4 – {S1-S5-S2-S3-S4-S6}

Path5 – {S1-S3-S5-S6}

Path6 – {S1-S3-S5-S4-S6}

Path7 – {S1-S3-S4-S6}

Path8 – {S1-S3-S4-S5-S6}

Path9 – {S1-S3-S2-S5-S4-S6}

Path10 – {S1-S2-S3-S4-S6}

Path11 – {S1-S2-S3-S4-S5-S6}

Deep First Search:

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Steps:

1) Traverse depth-wise & Store all switches of a route in a stack

2) Find Deepest switch in network

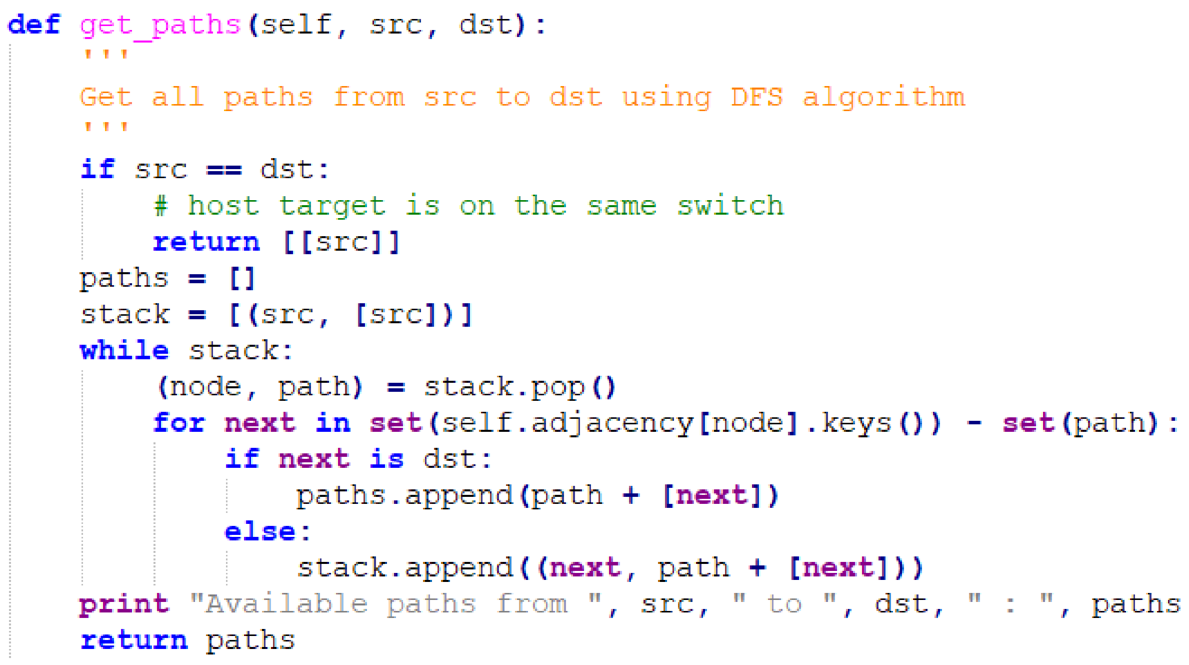
3) Backtracks to find initial switch & find other deepest switch

4) List all routes

Time Complexity: O(V+E)

Visit adjacent unvisited vertex. Make it as visited & Insert it into stack

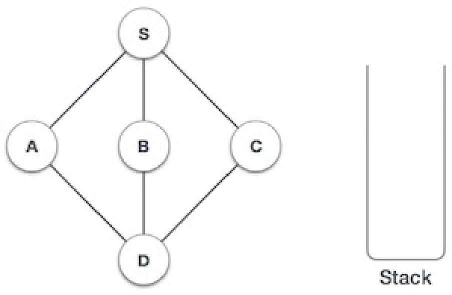
If no adjacent vertex, remove last vertex from stack



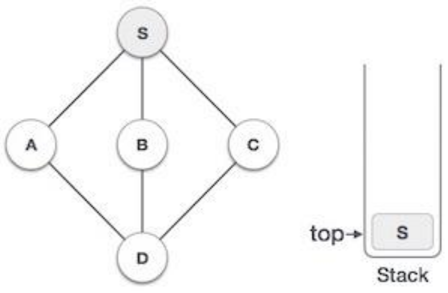
```
def get_paths(self, src, dst):
    '''
    Get all paths from src to dst using DFS algorithm
    '''
    if src == dst:
        # host target is on the same switch
        return [[src]]
    paths = []
    stack = [(src, [src])]
    while stack:
        (node, path) = stack.pop()
        for next in set(self.adjacency[node].keys()) - set(path):
            if next is dst:
                paths.append(path + [next])
            else:
                stack.append((next, path + [next]))
    print "Available paths from ", src, " to ", dst, " : ", paths
    return paths
```

Visual Illustration of DFS working:

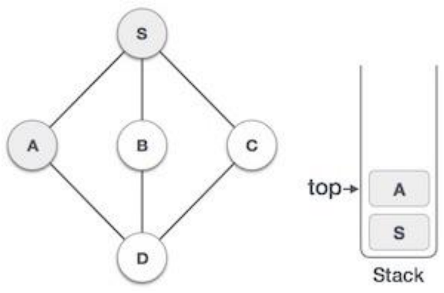
1) Initialize the stack.



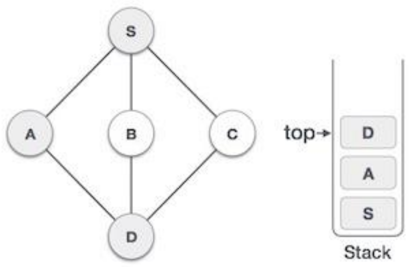
2) Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.



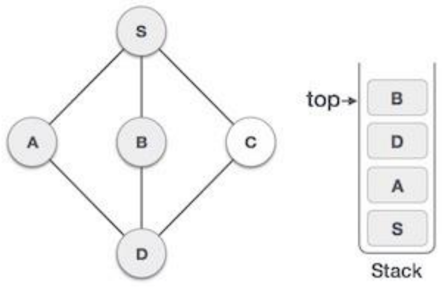
3) Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.



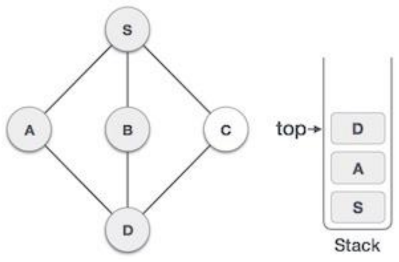
4) Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.



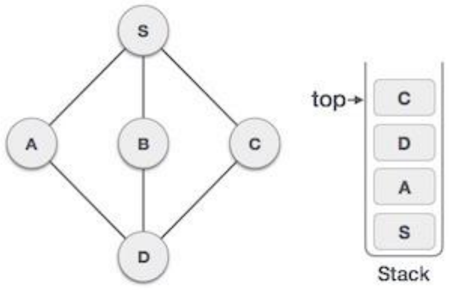
5) We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.



6) We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.



7) Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.



As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

PATH COST FINDING

DFS returns only path and not the path costs. But to calculate Bucket weights and to take the least weighted & optimal route, we need path cost.

Link Cost:

```
def get_link_cost(self, s1, s2):
    '''
    Get the link cost between two switches
    '''
    e1 = self.adjacency[s1][s2]
    e2 = self.adjacency[s2][s1]
    bl = min(self.bandwidths[s1][e1], self.bandwidths[s2][e2])
    ew = REFERENCE_BW/bl
    return ew
```

We extract the cost of each link from the adjacency matrix.

Path Cost:

```
def get_path_cost(self, path):
    '''
    Get the path cost
    '''
    cost = 0
    for i in range(len(path) - 1):
        cost += self.get_link_cost(path[i], path[i+1])
    return cost
```

We add the link costs corresponding to all the links in each path to get the path cost.

Get ‘n’ no. of Optimal paths:

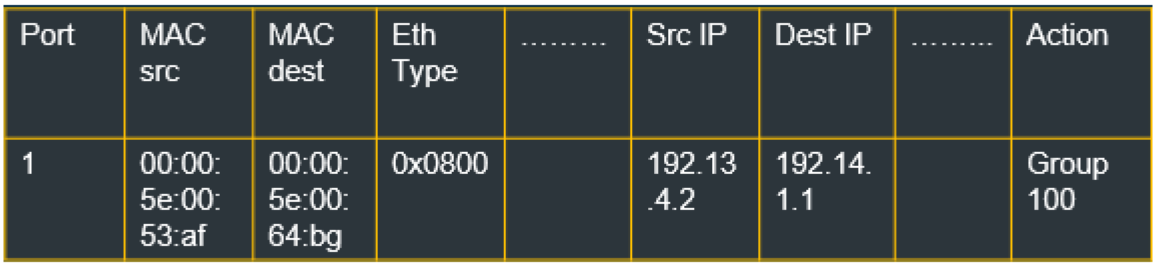
```
def get_optimal_paths(self, src, dst):
    '''
    Get the n-most optimal paths according to MAX_PATHS
    '''
    paths = self.get_paths(src, dst)
    paths_count = len(paths) if len(
        paths) < MAX_PATHS else MAX_PATHS
    return sorted(paths, key=lambda x: self.get_path_cost(x))[0:(paths_count)]
```

Even though the network might have a greater number of paths from a given source host to destination host, but we will only choose ‘n’ optimal paths with least cost among other paths. Only these ‘n’ optimal paths will carry the data packets to destination.

INSTALLING FLOW IN SWITCH

Flow Entry:
A set of actions to be applied based on a criterion of the packet headers

Match & Action uses the 5 packet headers and the flow entry in flow table to forward packet across multiple ports.



Port	MAC src	MAC dest	Eth Type	………	Src IP	Dest IP	………	Action
1	00:00:5e:00:53:af	00:00:5e:00:64:bg	0x0800		192.13.4.2	192.14.1.1		Group 100

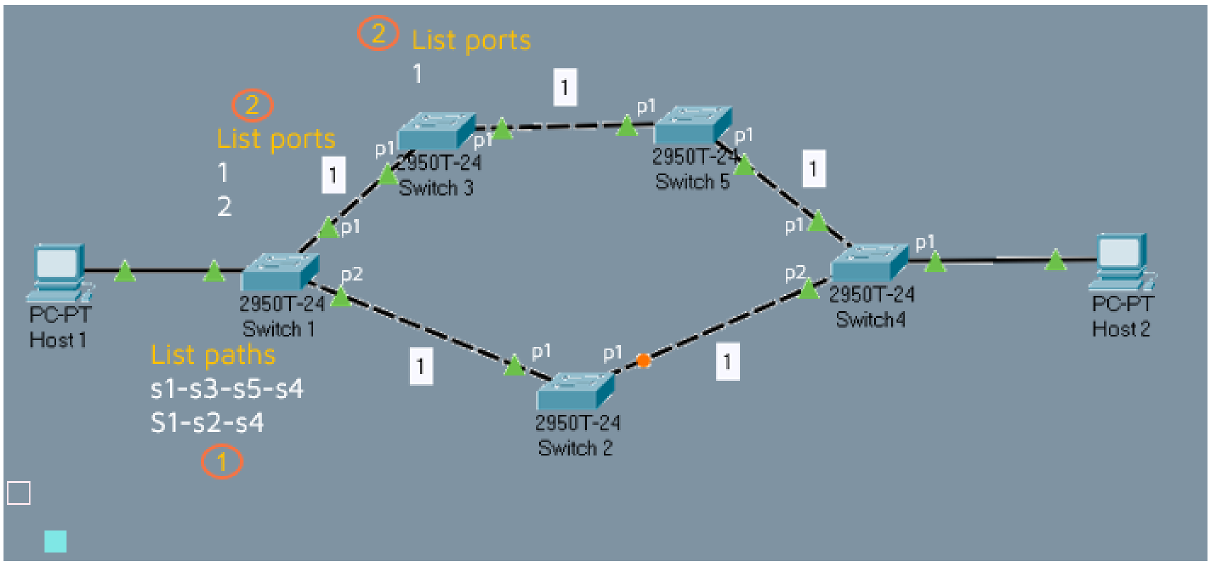
Workflow in installing packets into switch:

```
1) List all paths from source to destination.
2) Loop through all the switches that contain a path.
        1) List all the ports in the switch that contains a path.
        2) If (multiple ports in switch contain a path){
                Create a group table flow with type select  }
          else {
                Install a normal flow }
```

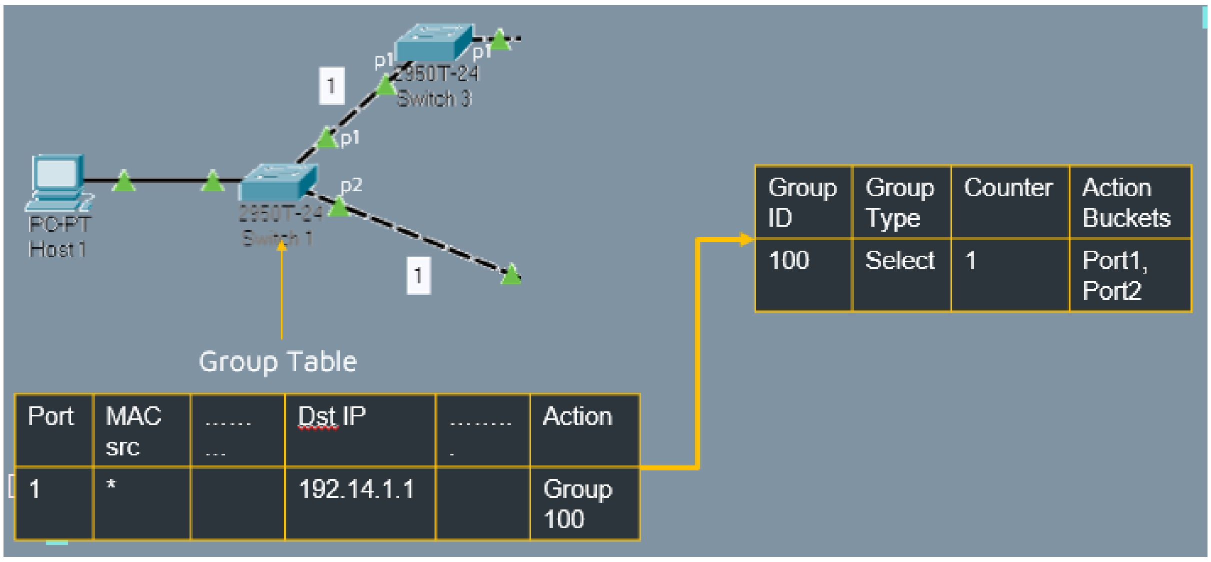
To create a group table, we create buckets (group of actions)

- **bucket weight:** weight of the bucket
- **actions:** output ports

Visual Representation of flow installation:



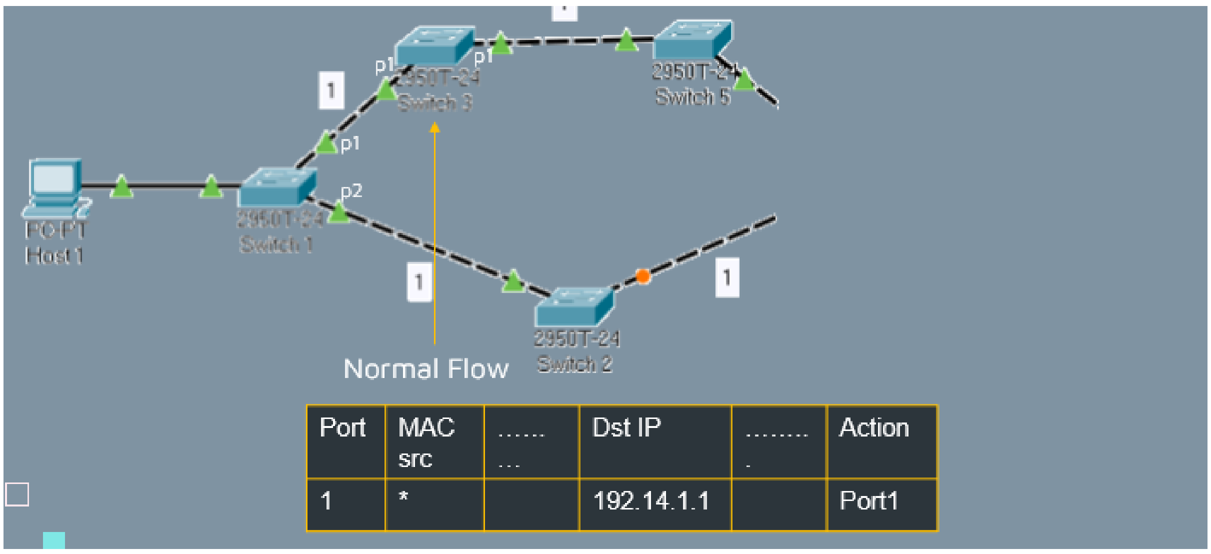
We first find the topology of the network using LLDP(Link Layer Discovery Protocol). Then we collect all the port status from the OpenFlow switches.



Group ID	Group Type	Counter	Action Buckets
100	Select	1	Port1, Port2

Port	MAC src	....	Dst IP	....	Action
1	*		192.14.1.1		Group 100

If the switch contains multiple ports leading to a path to our destination, then we use 'SELECT' type of group table to send through multiple ports as action.



Port	MAC src	………	Dst IP	………	Action
1	*		192.14.1.1		Port1

If the switch contains only a single ports leading to a path to our destination, then we insert normal flow entry in flow-table which send our packet only through one port.

Bucket Weight:

Ratio of the path weight of p with the total path weight of the available paths

$$bw(p) = \left(1 - \frac{pw(p)}{\sum_{i<n}^{i=0} pw(i)}\right) x10$$

For a path 'p':

bw - bucket weight, $0 \leq bw(p) < 10$

pw - path weight/cost

n - total no. of paths available

So, by doing this, we normalise the path weights between 0 to 10.

There may be many possible paths between the source host and destination host, but we are not going to send packets through all the paths available.

We will just take 'n' no. of optimal paths and split our packets into those paths. So, those 'n' most optimal paths with the highest bucket weights will be chosen.

Bucket weight for our topology:

Path Weight:

pw1 = (s4-s2) + (s2-s1) = 2

pw2 = (s4-s5) + (s5+s3) + (s3-s1) = 3

(Note: Here, 's4-s2' means link between host s4 to host s2)

For our topology, the path cost of 1st path is 2

and 2nd path is 3.

Bucket Weight:

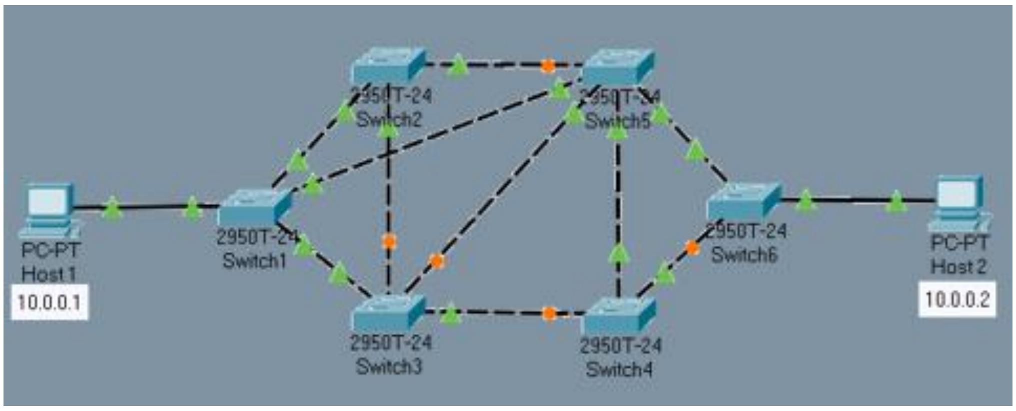
bw1 = (1 – 2/5) * 10 = 6

bw2 = (1 – 3/5) * 10 = 4

Using, bucket weight formula, we can find that bucket weight of path1 as 6

And path2 is 4.

Mininet Topology:



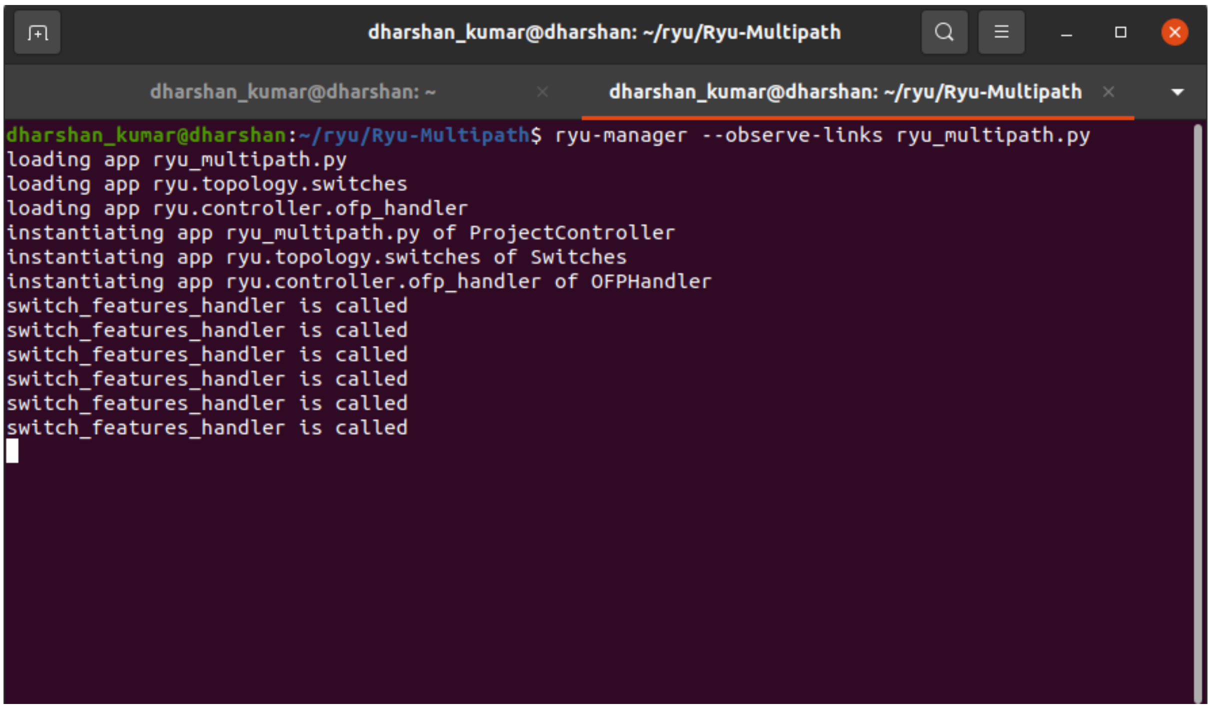
Here, you can see the network topology in this picture which we made using the cisco packet tracer.

Topology code:

```
from mininet.topo import Topo

class MyTopo( Topo ):
    "ring topology example."
    def build( self ):
        "Create custom topo."

        # Add hosts
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')

        #Add switches
        u = self.addSwitch('s1')
        v = self.addSwitch('s2')
        x = self.addSwitch('s3')
        y = self.addSwitch('s4')
        w = self.addSwitch('s5')
        z = self.addSwitch('s6')

        # Add links
        self.addLink( h1, u )
        self.addLink( u, v )
        self.addLink( u, x )
        self.addLink( u, w )
        self.addLink( v, w )
        self.addLink( v, x )
        self.addLink( x, w )
        self.addLink( x, y )
        self.addLink( y, w )
        self.addLink( y, z )
        self.addLink( z, w )
        self.addLink( z, h2 )

topos = { 'dynamic_topo': ( lambda: MyTopo() ) }
```

It is the Mininet code for this topology. We are connecting the hosts and switches as shown in the topology picture.

RYU controller:


dharshan_kumar@dharshan: ~/ryu/Ryu-Multipath
dharshan_kumar@dharshan: ~
dharshan_kumar@dharshan: ~/ryu/Ryu-Multipath

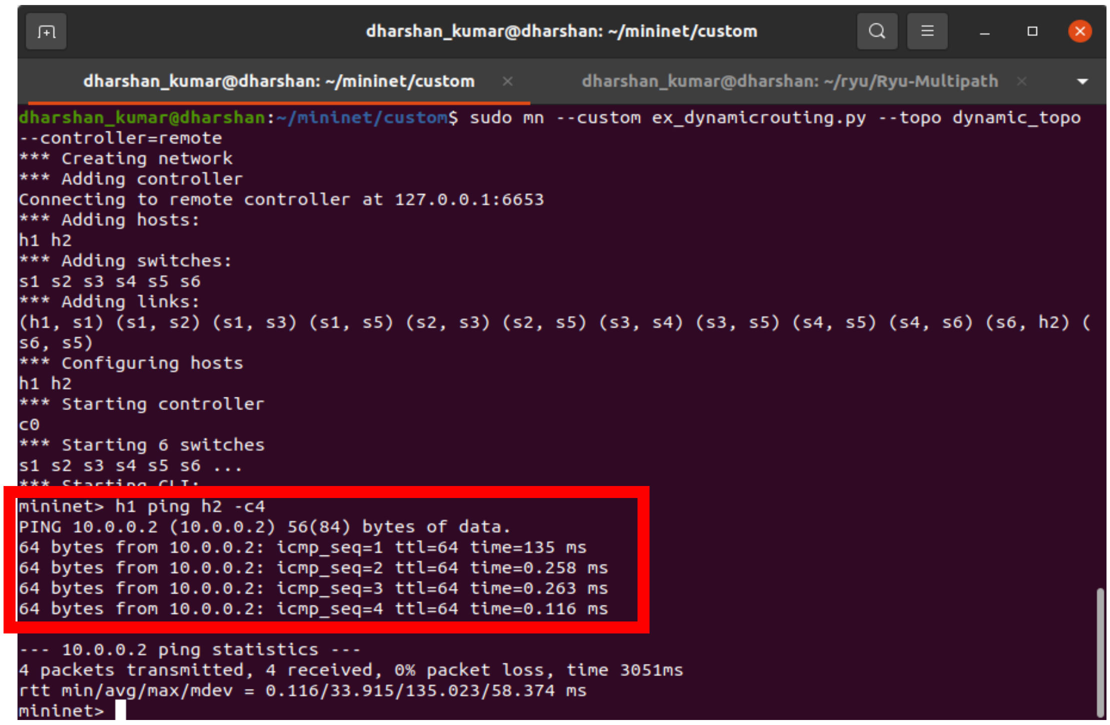
```
dharshan_kumar@dharshan:~/ryu/Ryu-Multipath$ ryu-manager --observe-links ryu_multipath.py
loading app ryu_multipath.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app ryu_multipath.py of ProjectController
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
switch_features_handler is called
switch_features_handler is called
switch_features_handler is called
switch_features_handler is called
switch_features_handler is called
switch_features_handler is called
```

Since we are implementing load balancing in SDN, so we need a controller to insert the flow entries into the switches.

So, in a command window, we are invoking the ryu-manager by loading our multipath file in it.

We also enable the processing of LLDP packets by the controller using the '--observe-links' option when starting the ryu.

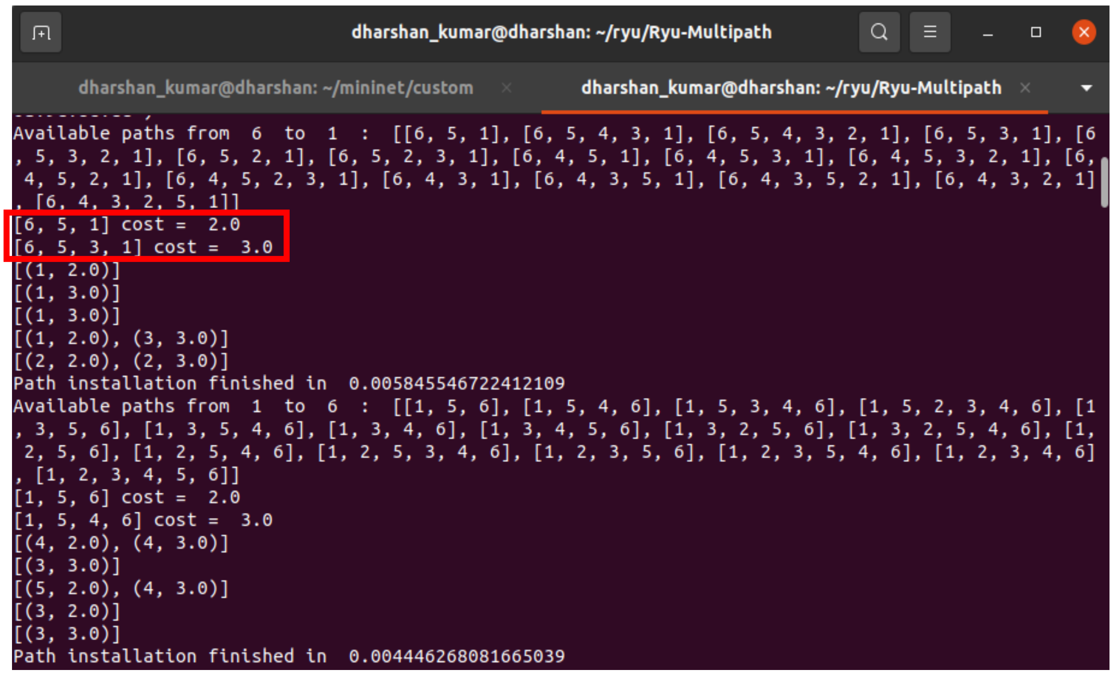
Ping:


dharshan_kumar@dharshan: ~/mininet/custom
dharshan_kumar@dharshan: ~/mininet/custom
dharshan_kumar@dharshan: ~/ryu/Ryu-Multipath

```
dharshan_kumar@dharshan:~/mininet/custom$ sudo mn --custom ex_dynamicrouting.py --topo dynamic_topo
--controller=remote
*** Creating network
*** Adding controller
Connecting to remote controller at 127.0.0.1:6653
*** Adding hosts:
h1 h2
*** Adding switches:
s1 s2 s3 s4 s5 s6
*** Adding links:
(h1, s1) (s1, s2) (s1, s3) (s1, s5) (s2, s3) (s2, s5) (s3, s4) (s3, s5) (s4, s5) (s4, s6) (s6, h2) (
s6, s5)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 6 switches
s1 s2 s3 s4 s5 s6 ...
*** Starting CLI:
mininet> h1 ping h2 -c4
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=135 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.258 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.263 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.116 ms

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3051ms
rtt min/avg/max/mdev = 0.116/33.915/135.023/58.374 ms
mininet>
```

Next, we load our topology using mininet. Then, we ping from host 1 to the other host 2. As, we can see the packets had successfully reached the destination host with 0 packet loss.

Path Finding:


dharshan_kumar@dharshan: ~/ryu/Ryu-Multipath
dharshan_kumar@dharshan: ~/mininet/custom
dharshan_kumar@dharshan: ~/ryu/Ryu-Multipath

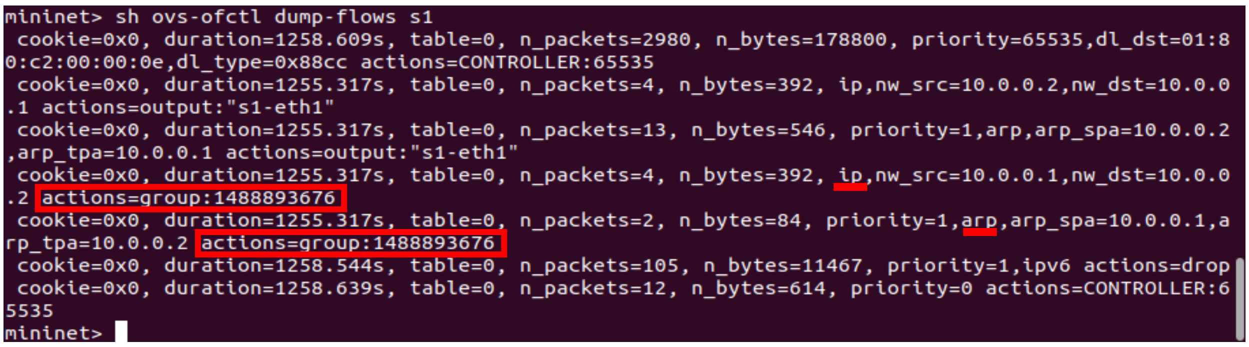
```
Available paths from  6  to  1  :  [[6, 5, 1], [6, 5, 4, 3, 1], [6, 5, 4, 3, 2, 1], [6, 5, 3, 1], [6
, 5, 3, 2, 1], [6, 5, 2, 1], [6, 5, 2, 3, 1], [6, 4, 5, 1], [6, 4, 5, 3, 1], [6, 4, 5, 3, 2, 1], [6,
 4, 5, 2, 1], [6, 4, 5, 2, 3, 1], [6, 4, 3, 1], [6, 4, 3, 5, 1], [6, 4, 3, 5, 2, 1], [6, 4, 3, 2, 1]
, [6, 4, 3, 2, 5, 1]]
[6, 5, 1] cost =  2.0
[6, 5, 3, 1] cost =  3.0
[(1, 2.0)]
[(1, 3.0)]
[(1, 3.0)]
[(1, 2.0), (3, 3.0)]
[(2, 2.0), (2, 3.0)]
Path installation finished in  0.005845546722412109
Available paths from  1  to  6  :  [[1, 5, 6], [1, 5, 4, 6], [1, 5, 3, 4, 6], [1, 5, 2, 3, 4, 6], [1
, 3, 5, 6], [1, 3, 5, 4, 6], [1, 3, 4, 6], [1, 3, 4, 5, 6], [1, 3, 2, 5, 6], [1, 3, 2, 5, 4, 6], [1,
 2, 5, 6], [1, 2, 5, 4, 6], [1, 2, 5, 3, 4, 6], [1, 2, 3, 5, 6], [1, 2, 3, 5, 4, 6], [1, 2, 3, 4, 6]
, [1, 2, 3, 4, 5, 6]]
[1, 5, 6] cost =  2.0
[1, 5, 4, 6] cost =  3.0
[(4, 2.0), (4, 3.0)]
[(3, 3.0)]
[(5, 2.0), (4, 3.0)]
[(3, 2.0)]
[(3, 3.0)]
Path installation finished in  0.004446268081665039
```

Now, we come back to the RYU command window again. Now, here we can see that all the available paths from the switch 1 to switch 6 has been listed here.

According to our code, out of all these available paths, it will only give us ‘n’ no. of optimal paths. In our case, we took ‘n’ value as 2. So, here we can see the 2 most optimal paths with their respective path costs.

The optimal path is selected based on the highest bucket weight and lowest path cost. We can also see that the Paths have been successfully installed in the switches.

Flow Entry:

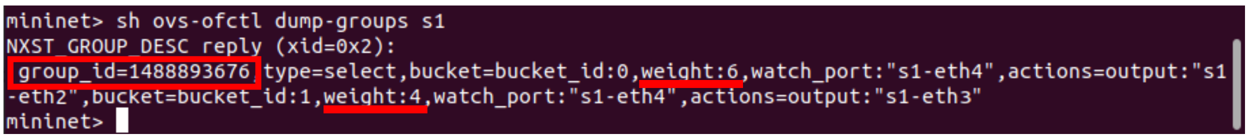


```
mininet> sh ovs-ofctl dump-flows s1
 cookie=0x0, duration=1258.609s, table=0, n_packets=2980, n_bytes=178800, priority=65535,dl_dst=01:8
0:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=1255.317s, table=0, n_packets=4, n_bytes=392, ip,nw_src=10.0.0.2,nw_dst=10.0.0
.1 actions=output:"s1-eth1"
 cookie=0x0, duration=1255.317s, table=0, n_packets=13, n_bytes=546, priority=1,arp,arp_spa=10.0.0.2
,arp_tpa=10.0.0.1 actions=output:"s1-eth1"
 cookie=0x0, duration=1255.317s, table=0, n_packets=4, n_bytes=392, ip,nw_src=10.0.0.1,nw_dst=10.0.0
.2 actions=group:1488893676
 cookie=0x0, duration=1255.317s, table=0, n_packets=2, n_bytes=84, priority=1,arp,arp_spa=10.0.0.1,a
rp_tpa=10.0.0.2 actions=group:1488893676
 cookie=0x0, duration=1258.544s, table=0, n_packets=105, n_bytes=11467, priority=1,ipv6 actions=drop
 cookie=0x0, duration=1258.639s, table=0, n_packets=12, n_bytes=614, priority=0 actions=CONTROLLER:6
5535
mininet> 
```

Now, we go to the mininet command window and display all the flow entries in the first switch ‘s1’. This switch s1 is nearest switch to the router.

We can see few flow entries here, but we can also see 2 flow entries with their action leading to a group table. This flow entries are inserted by the controller into switches. 1 flow entry for IP address and another flow entry for MAC address.

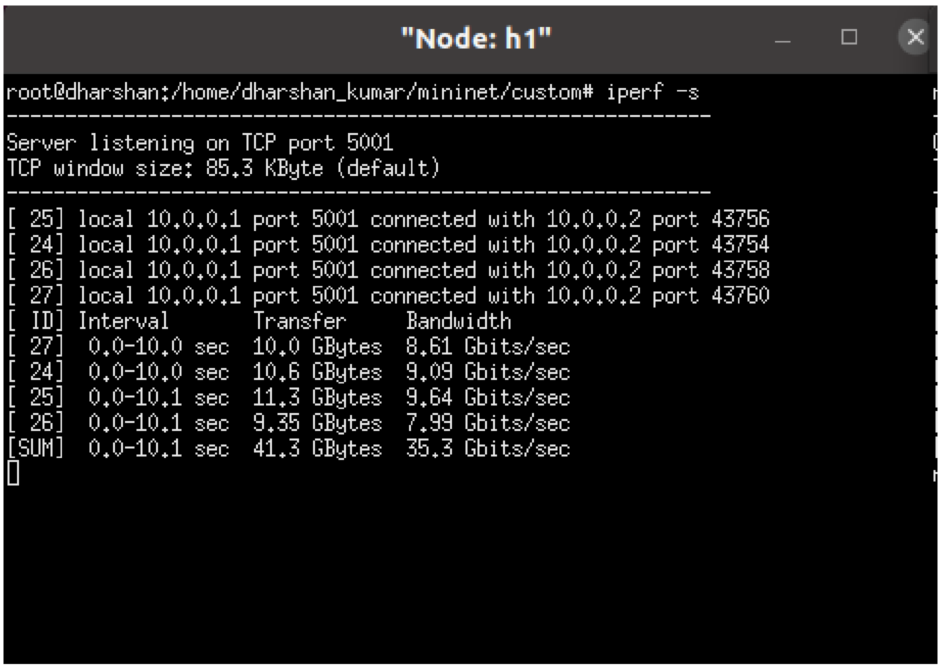
Group Table Entry:



```
mininet> sh ovs-ofctl dump-groups s1
NXST_GROUP_DESC reply (xid=0x2):
 group_id=1488893676,type=select,bucket=bucket_id:0,weight:6,watch_port:"s1-eth4",actions=output:"s1
-eth2",bucket=bucket_id:1,weight:4,watch_port:"s1-eth4",actions=output:"s1-eth3"
mininet>
```

We had also listed the group table contents.We can see the 2 actions for each path. One with bucket weight 6 and another with bucket weight of 4.

Simulate Server and Client:

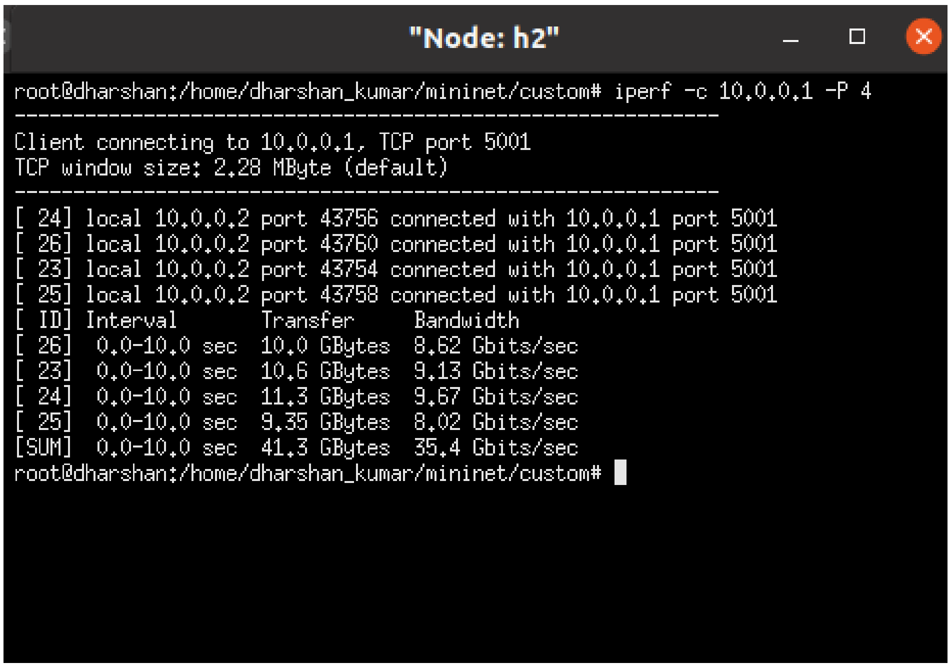


```
root@dharshan:/home/dharshan_kumar/mininet/custom# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 25] local 10.0.0.1 port 5001 connected with 10.0.0.2 port 43756
[ 24] local 10.0.0.1 port 5001 connected with 10.0.0.2 port 43754
[ 26] local 10.0.0.1 port 5001 connected with 10.0.0.2 port 43758
[ 27] local 10.0.0.1 port 5001 connected with 10.0.0.2 port 43760
[ ID] Interval       Transfer     Bandwidth
[ 27]  0.0-10.0 sec  10.0 GBytes  8.61 Gbits/sec
[ 24]  0.0-10.0 sec  10.6 GBytes  9.09 Gbits/sec
[ 25]  0.0-10.1 sec  11.3 GBytes  9.64 Gbits/sec
[ 26]  0.0-10.1 sec  9.35 GBytes  7.99 Gbits/sec
[SUM]  0.0-10.1 sec  41.3 GBytes  35.3 Gbits/sec
```

Next, we will try to simulate the host1 as client and host2 as server. By using ‘xterm’ command for host1 and host2, we open their corresponding terminal windows, then we use ‘iperf’ command to simulate server & client.

Host ‘h1’ will be the server

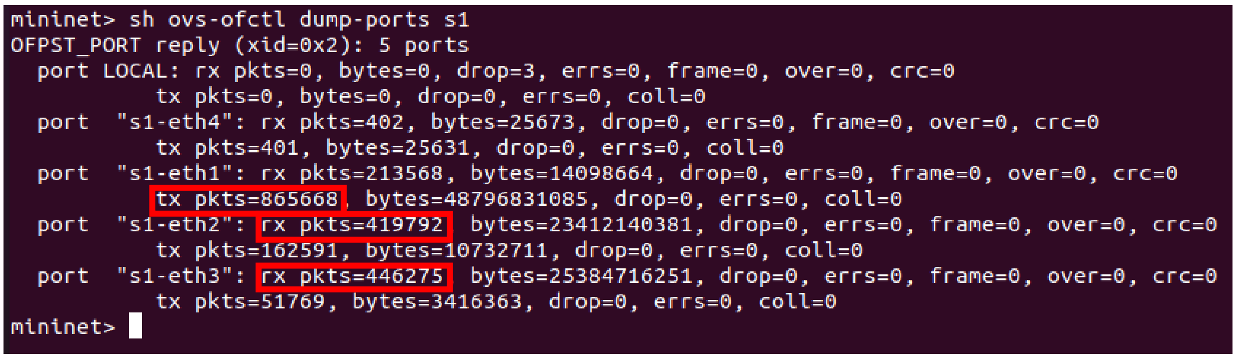
Host ‘h2’ will be the client



"Node: h2"

```
root@dharshan:/home/dharshan_kumar/mininet/custom# iperf -c 10.0.0.1 -P 4
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 2.28 MByte (default)
------------------------------------------------------------
[ 24] local 10.0.0.2 port 43756 connected with 10.0.0.1 port 5001
[ 26] local 10.0.0.2 port 43760 connected with 10.0.0.1 port 5001
[ 23] local 10.0.0.2 port 43754 connected with 10.0.0.1 port 5001
[ 25] local 10.0.0.2 port 43758 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 26]  0.0-10.0 sec  10.0 GBytes  8.62 Gbits/sec
[ 23]  0.0-10.0 sec  10.6 GBytes  9.13 Gbits/sec
[ 24]  0.0-10.0 sec  11.3 GBytes  9.67 Gbits/sec
[ 25]  0.0-10.0 sec  9.35 GBytes  8.02 Gbits/sec
[SUM]  0.0-10.0 sec  41.3 GBytes  35.4 Gbits/sec
root@dharshan:/home/dharshan_kumar/mininet/custom#
```

Checking whether Multipath occurs or not:



```
mininet> sh ovs-ofctl dump-ports s1
OFPST_PORT reply (xid=0x2): 5 ports
  port LOCAL: rx pkts=0, bytes=0, drop=3, errs=0, frame=0, over=0, crc=0
           tx pkts=0, bytes=0, drop=0, errs=0, coll=0
  port  "s1-eth4": rx pkts=402, bytes=25673, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=401, bytes=25631, drop=0, errs=0, coll=0
  port  "s1-eth1": rx pkts=213568, bytes=14098664, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=865668, bytes=48796831085, drop=0, errs=0, coll=0
  port  "s1-eth2": rx pkts=419792, bytes=23412140381, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=162591, bytes=10732711, drop=0, errs=0, coll=0
  port  "s1-eth3": rx pkts=446275, bytes=25384716251, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=51769, bytes=3416363, drop=0, errs=0, coll=0
mininet>
```

Now, after our simulation is done, we will display the port status of switch s1.

Port1 (tx): 865668

Port2 (rx): 419792

Port3 (rx): 446275

Here, ‘rx’ means total no. of packets received

And ‘tx’ means total no. of packets transmitted.

Obtained Ratio

419792 : 446275 = 2 : 1.7 = 6 : 5.1

Original Ratio

6:4

$\boxed{6 : 5.1 \sim 6 : 4}$

As we can see, the no. of packets from port1 is transmitted into port 2 & 3 with a similar ratio of bucket weights.

CONCLUSION

From this project work, we can see how we successfully implemented and tested multipath routing with load balancing using OpenFlow. Some things to note:

- We used group actions available in OpenFlow version 1.1.
- Used the “select” group action type.
- Created “buckets” in a group action.
- Calculated bucket weights from the path weight.
- Packet load-balancing was done by hashing packet headers down to the TCP headers, multiplying it with the bucket weights, and then selecting the bucket with the highest “score”.

Future Work:

May use Multipath load balancing over Fast-Failover Group table. If a link fails in one of the paths, then we can use another alternative path if there is any in the network topology.

Can test our model with multiple complex topologies with more than 20 hosts and 20 switches to check for the multipath load balancing performance.

REFERENCE

1. https://en.wikipedia.org/wiki/Load_balancing_(computing)
   (Load Balancing concepts on Wikipedia)

2. https://www.youtube.com/watch?v=XBIR88qnLoA
   (Multipath concept explanation video)

3. https://www.slideshare.net/SabeloDlamini3/multipath-load-balancing-for-sdn-data-plane
   (Reference PPT on multipath load balancing)

4. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
   (DFS concept in GeeksforGeeks)

5. https://www.nginx.com/resources/glossary/load-balancing/
   (Types of load balancing)

I have uploaded the multipath code in my GitHub repository, for reference of full code, you can access it using this link: https://github.com/dharshankumar2002/Multipath-Load-Balancing/blob/main/ryu_multipath.py