SDN-based Load Balancing and Traffic Steering

Mahesh Kutre[1], Shreya Vaidya[2], Karthik I C[1], Narva Vijay[1], Suneeta V Budihal[1], and Sumita Guddin[1]

School of Electronics and Communication Engineering,
KLE Technological University Hubballi, India
`01fe22bei408@kletech.ac.in`, `01fe21bei008@kletech.ac.in`,
`01fe22bei411@kletech.ac.in`, `01fe22bei407@kletech.ac.in`

Abstract. In this project, we used Mininet to build and configure a network with 16 hosts and 8 switches connected in various ways. Using OpenFlow controllers and a custom script called topology.py, we were able to remotely manage the network. The primary objective was to verify various pathways, data transmission methods, and latency times in order to assess the network's functionality. The network functioned well, with strong connectivity and effective data routing, according to the testing. For instance, the average round-trip time for ping requests from host h3 to host h9 was only 2.219 milliseconds, and out of 10 packets sent, none were dropped. We also examined the switch S2 data, which included the number and size of packets processed by the switch. The port "s2-eth4" received over 2.55 million packets and sent out over 1.21 million packets with no errors or lost packets, according to the output of the ovs-ofctl dump-ports s2 command .We computed multiple alternative pathways with varying prices between various nodes, such as node 2 and node 5. The network demonstrated its ability to dynamically route traffic depending on cost when it selected the shortest path, which had a cost of 3.0. These pathways took only an average of 0.0012 seconds to set up. In conclusion, this project showed how to successfully build up and maintain a dynamic network in Mininet, emphasizing effective path calculations, good performance, and minimal latency.

Keywords: Load Balancing · Traffic Steering · DFS.

1 Introduction

With the use of software applications, network traffic may be intelligently and centrally controlled thanks to the innovative network architecture known as software-defined networking, or SDN. The network's control plane and data plane are kept apart by this centralized control, which allows for flexible modifications and comprehensive network management. SDN-based load balancing in this context distributes network traffic among several servers or paths in order to keep any one server or path from being overburdened. Utilizing SDN's centralized control plane, which can dynamically redistribute traffic based on current network conditions, improves performance and resource consumption. The network demand and complexity are increasing day by day, traditional networking methods are not so good at effectively handling the dynamic behavior of the latest traffic patterns. Software Defined Networking (SDN) has a separate control plane and data plane which ensure centralized management of entire network infrastructure. This project aims to multipath load balancing and traffic steering with the help of Ryu SDN controller and Mininet. In this project we are focusing on different methods to improve network reliability and performance using multi path data transmission between source and destination nodes. Multipath routing distributes traffic across different available paths compared to traditional single-path routing. This method helps balance the load, enhancing overall throughput and network resilience. In this project, the Ryu controller is used to dynamically manage and optimize traffic flow in the network. Mininet provides the testing environment for this project, enabling the simulation of complex network topology and traffic conditions. Similar to this, SDN-based traffic steering concentrates on the dynamic routing of network traffic via particular paths to satisfy a range of requirements, including bandwidth availability, application-specific demands, and Quality of Service (QoS) standards. Network managers can design complex policies that specify traffic routing, guaranteeing best-in-class performance and dependability, by employing SDN. With this method, traffic may be redirected in real-time to prevent congestion, lower latency, and smoothly manage network outages or maintenance tasks. Within an SDN architecture, load balancing and traffic steering work together to offer a number of advantages, such as increased network performance, scalability, flexibility, and centralized management. As a result, the network infrastructure becomes more flexible, dependable, and responsive and can adjust swiftly to needs and circumstances.

2 Literature Survey

Load balancing in Software-Defined Networking (SDN) literature covers a wide range of methods and methodologies for improving network flexibility and performance, including dynamic traffic steering in service function chaining. Round-robin, least connections, and weighted round-robin are common methods for distributing traffic fairly among SDN controllers. Efforts are also directed toward efficient load adjustment among controllers via algorithms such as dynamic load balancing, which dynamically allocates workload based on real-time network conditions. Multi-path load balancing systems, which employ techniques such as ECMP (Equal-Cost Multi-Path) or variants thereof, aim to improve performance and resilience in SDN data planes. Comprehensive surveys assess the usefulness of these algorithms in maximizing resource consumption and network efficiency, developing SDN capabilities in traffic management, and improving overall system performance.

2.1 Related Works

In this [1] paper, they investigate the design and assessment of differing load-balancing algorithms inside the context of Software-Defined Networking (SDN). The authors aim to improve network acts by accurately distributing traffic and preventing bottlenecks. This content reads as if it is human-written. They test different algorithms, containing

Round Robin, Least Connection, and Weighted Round Robin, to decide their influence in balancing loads across the network. The methods include executing these algorithms in an SDN atmosphere utilizing tools like Mininet for simulation and the POX controller for network management. The authors conduct an order of tests to measure key performance metrics, including throughput, latency, and packet loss, under different network surroundings. The output displays that the Weighted Round Robin algorithm acts best in environments of balancing network load and optimizing resource utilization, providing an important enhancement in network efficiency. This research determines valuable

In this [2] paper, they investigated the performance of the round-robin load balancing method in Software-Defined Networking (SDN). The Round Robin algorithm will be implemented into an SDN controller to see how efficiently it distributes network traffic in a simple, cyclical fashion. They evaluated its performance in a simulated environment by examining parameters such as throughput, latency, packet loss, and resource utilization. They compare the round-robin algorithm with different load-balancing strategies. They highlight both its advantages and disadvantages, especially since it does not consider real-time servers. Their research provides valuable insights and recommendations to improve SDN load-balancing algorithms for future network implementation

In this [3] paper, authors define the creation of a load-balancing result particularly designed for SDN-based cloud environments. They devise an automated load balancer that uses machine learning to predict traffic trends and apply load distribution. The balancer is created with tools like Mininet and OpenDaylight and combines machine learning models accompanying classic methods like Least Connection and Round Robin. This method allows the load balancer to dynamically respond to real-time network conditions. this research demonstrates considerable increases in throughput and latency demonstrating the efficacy of applying machine learning to SDN load balancing when this is compared with previous methods[examinations into selecting appropriate load-balancing approaches for SDN.

In this [4] paper they present a creative approach to network construction and traffic administration. Their proposed design influences Software-Defined Networking (SDN) law to embellish the adaptability and adeptness assisting function chaining (SFC). The key focus displays or takes public a new traffic guiding form planned to dynamically route traffic through miscellaneous help functions in a predefined order. By decoupling the control plane from the dossier plane, SDN allows for programmable and concentrated control over network traffic, optimizing system exercise and embellishing help transfer. Here, they likely examine by what method their order raises network accomplishment, scalability, and changeability to variable duty requirements, eventually providing more adept network administration and reinforced consumer knowledge.

In this [5] paper they address the challenge of load adjustment among diversified controllers inside Software-Defined Networking (SDN) surroundings. The authors intend a method proposed to optimize the disposal of control plane tasks across diversified SDN controllers to reinforce structure reliability and depiction. The focus is likely on dynamically regulating the assigned work with the controller's established real-occasion network environments and boss powers. This approach aims to fear overload positions on some distinct controllers while ensuring adept network administration and lowering the risk of bottlenecks or losses. The expected profit would likely contain depiction versification professed improved whole dependability, lowered abeyance, and improved scalability of SDN deployments under varying traffic loads and functional sketches.

In this [6] paper, they are developing a method for optimizing request load disposal across a delivered Software-Defined Networking (SDN) boss architecture. The authors address the challenge of capably directing and weighing the assigned work among diversified SDN controllers to improve scheme openness and scalability. They likely employ algorithms that dynamically assign arriving use requests to various controllers based on palpable-opportunity verification to a degree of current workload, network traffic patterns, and boss refine ability. The wanted output from their arrangement would usually involve depiction metrics professed enhanced load disposal adeptness, reduced abeyance, and reinforced overall arrangement dependability. This technique tries to improve resource utilization across dispersed SDN controllers, enhancing network application quality of service while reducing the risk of performance degradation or bottlenecks in SDN settings.

2.2 Objectives

1)The goal of load balancing is to distribute traffic workload in order to maximize resource efficiency and service availability.
2)To accomplish security, quality of service, and content delivery.
3)Traffic management objectives and traffic steering place an emphasis on guiding traffic along predetermined paths.

2.3 Proposed Methodology

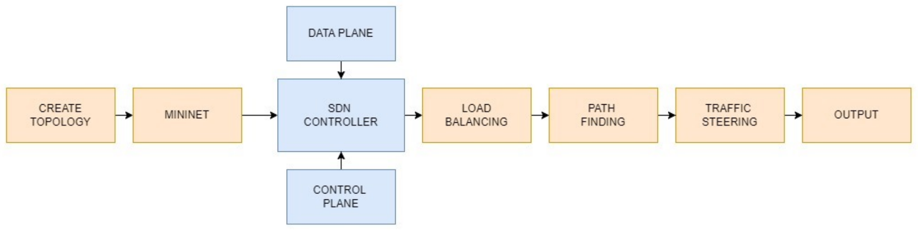


Figure 2.1: Block Diagram of the system

Figure 2.1 shows the block diagram of the system which contains different units:
Topology: Creating the network layout, which includes how switches, hosts, and links are arranged, is the task of this step. Scripts or tools for network design, are frequently created in Python or other scripting languages. The goal is to create the basic framework for the network that will be simulated or emulated.

Mininet: Mininet is a type of network emulator that builds a virtual network with switches, hosts, and links according to a predetermined topology. The goal is to eliminate the need for physical hardware by offering a realistic yet virtual environment for the development, testing, and experimentation of network setups and behaviors.

SDN Controller: The SDN controller, which communicates with network devices using protocols like OpenFlow, is the brains behind the SDN architecture. It controls the flow of data via the network. We demonstrated the output of this project using the Ryu controller. The goal is to centralize decision-making and network intelligence to facilitate dynamic resource management and configuration.

Load Balancing: In order to minimize congestion and maximize resource utilization, this block divides network traffic equally among several servers or pathways. The balancing algorithms for load methods are put into practice by the SDN controller. The goal is to minimize latency, make sure that network resources are used efficiently, and keep no one resource from becoming a bottleneck.

Path Finding: It is the process of figuring out the best paths for data packets to take from their source to their destination across the network. Computing capabilities and the shortest path are found using routing methods, such as Depth First Search. Determine the most practical and efficient routes for traffic routing while taking into account the policies and network conditions.

Traffic steering: To minimize delay or avoid busy paths, for example, traffic steering entails routing network traffic according to predetermined criteria. Instruments/Technologies: Segment routing, policy-based routing, and service function chaining (SFC). Ensure that traffic flows are controlled in accordance with performance or business policies in order to improve network quality of service (QoS) and overall efficiency.

Output: We have determined the path cost and link cost to communicate from the source to the destination after completing the load balancing and traffic steering. By applying traffic steering, packets are quickly sent from one location to another, resulting in reduced communication latency. Ultimately, this project's product is the ability to locate the right path.

2.4 Proposed Framework

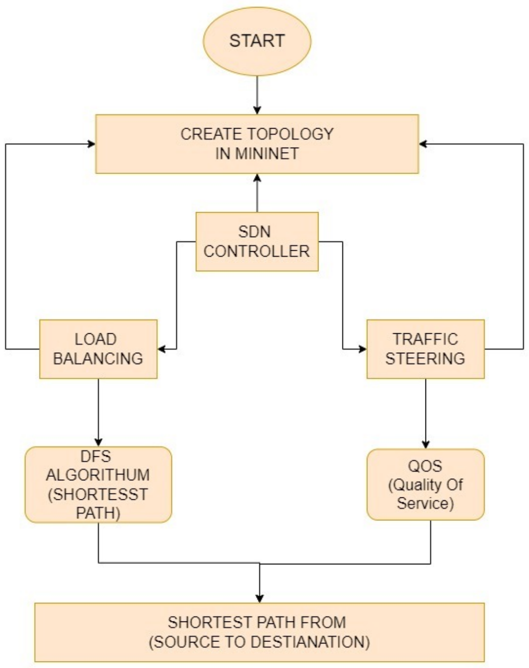

START
CREATE TOPOLOGY IN MININET
SDN CONTROLLER
LOAD BALANCING
TRAFFIC STEERING
DFS ALGORITHUM (SHORTESST PATH)
QOS (Quality Of Service)
SHORTEST PATH FROM (SOURCE TO DESTIANATION)

Figure 2.2 shows the Flowchart of the system

The figure 2.2 shows the topology was established through the use of the mininet emulator. A topology consisting of 16 hosts and 8 switches was established, and the DFS algorithm was employed to determine the shortest path. We use multipath load balancing for load balancing, and after running the code, it calculates the shortest path if the shortest path is given directly to load balancing after we have sent the path to traffic steering in the traffic steering block. It then checks the quality of service of the path at the end of this process to display the output in the Ryu environment.

3 Simulation Results

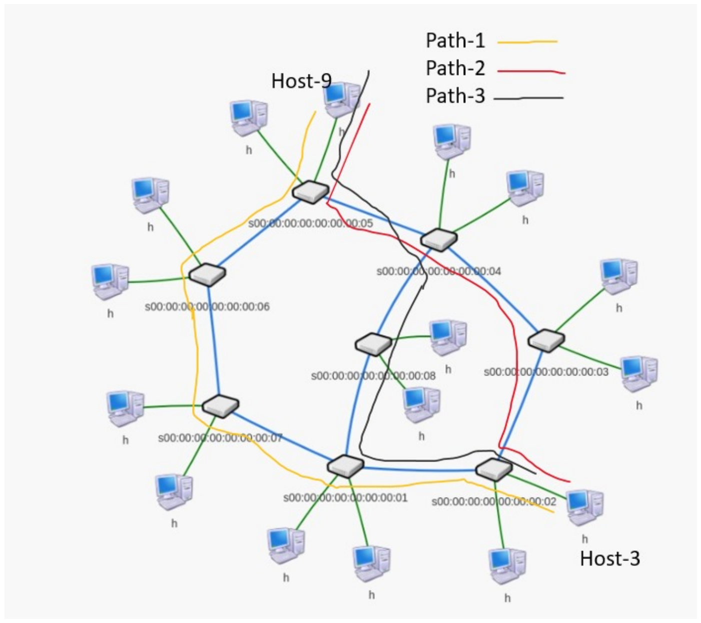
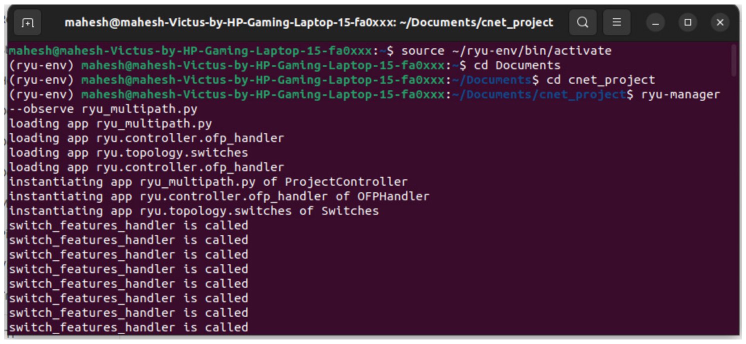


Figure 3.1: Custom Topology

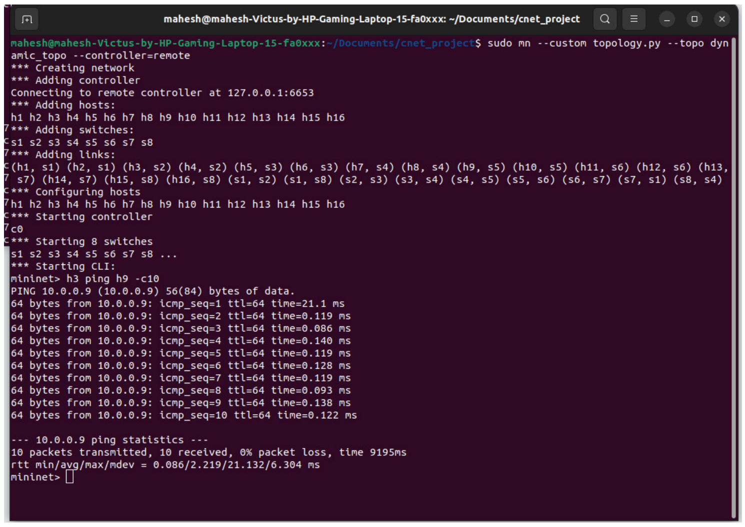
Figure 3.1 shows the topology with a total of 24 links, this setup provides redundancy and potentially better performance, as data can take different paths if one route is congested or fails. This design enhances the reliability and efficiency of the network. The figure shows a custom network topology designed with three paths connecting host-3 to host-9. The network consists of 8 switches and 16 hosts, ensuring multiple routes for data to travel between these hosts.



```
mahesh@mahesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx: ~/Documents/cnet_project
mahesh@mahesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~$ source ~/ryu-env/bin/activate
(ryu-env) mahesh@mahesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~$ cd Documents
(ryu-env) mahesh@mahesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Documents$ cd cnet_project
(ryu-env) mahesh@mahesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Documents/cnet_project$ ryu-manager
--observe ryu_multipath.py
loading app ryu_multipath.py
loading app ryu.controller.ofp_handler
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app ryu_multipath.py of ProjectController
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu.topology.switches of Switches
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
switch_features_handler is called
switch_features_handler is called
```

Figure 3.2: Running Ryu Controller

Figure 3.2 shows the activation of the Ryu SDN controller environment. The Ryu controller loads the ryu multipath.py application to manage the OpenFlow protocol and network topology. The switch features handler is called regularly to process feature messages from linked switches, allowing the controller to manage the network topology and set up the switches for multipath routing.

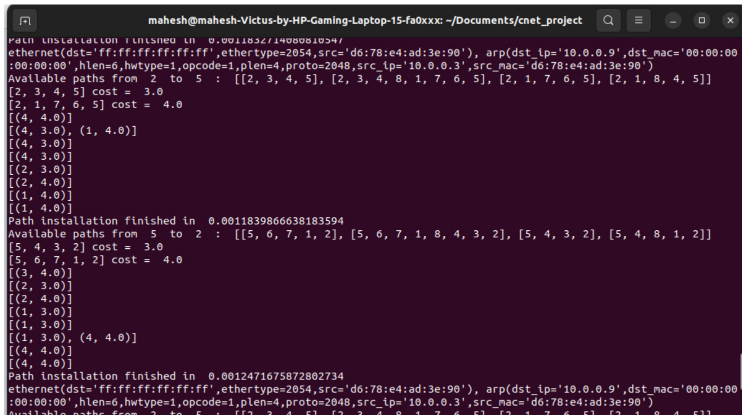


```
mahesh@mahesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx: ~/Documents/cnet_project
mahesh@mahesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:~/Documents/cnet_project$ sudo mn --custom topology.py --topo dyn
amic_topo --controller=remote
*** Creating network
*** Adding controller
Connecting to remote controller at 127.0.0.1:6653
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8
*** Adding links:
(h1, s1) (h2, s1) (h3, s2) (h4, s2) (h5, s3) (h6, s3) (h7, s4) (h8, s4) (h9, s5) (h10, s5) (h11, s6) (h12, s6) (h13,
 s7) (h14, s7) (h15, s8) (h16, s8) (s1, s2) (s1, s8) (s2, s3) (s3, s4) (s4, s5) (s5, s6) (s6, s7) (s7, s1) (s8, s4)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Starting controller
c0
*** Starting 8 switches
s1 s2 s3 s4 s5 s6 s7 s8 ...
*** Starting CLI:
mininet> h3 ping h9 -c10
PING 10.0.0.9 (10.0.0.9) 56(84) bytes of data.
64 bytes from 10.0.0.9: icmp_seq=1 ttl=64 time=21.1 ms
64 bytes from 10.0.0.9: icmp_seq=2 ttl=64 time=0.119 ms
64 bytes from 10.0.0.9: icmp_seq=3 ttl=64 time=0.086 ms
64 bytes from 10.0.0.9: icmp_seq=4 ttl=64 time=0.140 ms
64 bytes from 10.0.0.9: icmp_seq=5 ttl=64 time=0.119 ms
64 bytes from 10.0.0.9: icmp_seq=6 ttl=64 time=0.128 ms
64 bytes from 10.0.0.9: icmp_seq=7 ttl=64 time=0.119 ms
64 bytes from 10.0.0.9: icmp_seq=8 ttl=64 time=0.093 ms
64 bytes from 10.0.0.9: icmp_seq=9 ttl=64 time=0.138 ms
64 bytes from 10.0.0.9: icmp_seq=10 ttl=64 time=0.122 ms

--- 10.0.0.9 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9195ms
rtt min/avg/max/mdev = 0.086/2.219/21.132/6.304 ms
mininet> 
```

Figure 3.3: Forwarding 10 packets from h3 to h9

Figure 3.3 shows the configuration and execution of an SDN-based network simulation with Mininet and a remote Ryu controller. We are creating a network topology with 16 hosts and 8 switches, then it connects to the Ryu controller via 127.0.0.1:6653. We are using the ping command to send 10 packets from host h3 to host h9, which is successful with no packet loss.



```
mahesh@mahesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx: ~/Documents/cnet_project
Path installation finished in  0.0011832714080810547
ethernet(dst='ff:ff:ff:ff:ff:ff',ethertype=2054,src='d6:78:e4:ad:3e:90'), arp(dst_ip='10.0.0.9',dst_mac='00:00:00
:00:00:00',hlen=6,hwtype=1,opcode=1,plen=4,proto=2048,src_ip='10.0.0.3',src_mac='d6:78:e4:ad:3e:90')
Available paths from  2  to  5  :  [[2, 3, 4, 5], [2, 3, 4, 8, 1, 7, 6, 5], [2, 1, 7, 6, 5], [2, 1, 8, 4, 5]]
[2, 3, 4, 5] cost =  3.0
[2, 1, 7, 6, 5] cost =  4.0
[(4, 4.0)]
[(4, 3.0), (1, 4.0)]
[(4, 3.0)]
[(4, 3.0)]
[(2, 3.0)]
[(2, 4.0)]
[(1, 4.0)]
[(1, 4.0)]
Path installation finished in  0.0011839866638183594
Available paths from  5  to  2  :  [[5, 6, 7, 1, 2], [5, 6, 7, 1, 8, 4, 3, 2], [5, 4, 3, 2], [5, 4, 8, 1, 2]]
[5, 4, 3, 2] cost =  3.0
[5, 6, 7, 1, 2] cost =  4.0
[(3, 4.0)]
[(2, 3.0)]
[(2, 4.0)]
[(1, 3.0)]
[(1, 3.0)]
[(1, 3.0), (4, 4.0)]
[(4, 4.0)]
[(4, 4.0)]
Path installation finished in  0.0012471675872802734
ethernet(dst='ff:ff:ff:ff:ff:ff',ethertype=2054,src='d6:78:e4:ad:3e:90'), arp(dst_ip='10.0.0.9',dst_mac='00:00:00
:00:00:00',hlen=6,hwtype=1,opcode=1,plen=4,proto=2048,src_ip='10.0.0.3',src_mac='d6:78:e4:ad:3e:90')
```

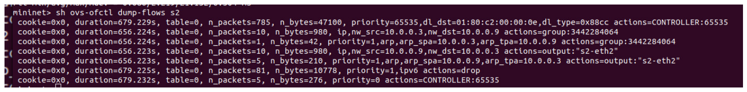
Figure 3.4: Finding paths present in topology

In Figure 3.4 We are using the DFS algorithm to find multiple paths.
1) It Traverses depth-wise and stores all switches of a route in a stack.
2) Find the deepest switch in the network.
3) Backtracks to find the initial switch find other deepest switch.

4) List all routes
We are communicating between host 2 and host 5 and vice versa. Then the controller calculates multiple available paths and displays the switch sequences and related costs. The paths are then deployed over the network, allowing for efficient traffic routing based on these computations.

$$\text{Path Cost} = \sum_{i=0}^{n-1} \text{Link Cost between } (i \text{ and } i+1)$$

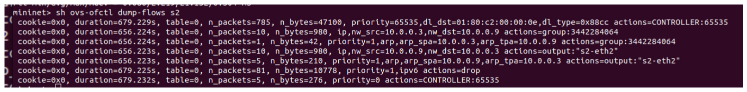
$$\text{Link Cost} = \frac{\text{REFERENCE\_BW}}{\text{Bandwidth of Link}}$$



```
mininet> sh ovs-ofctl dump-flows s2
 cookie=0x0, duration=679.229s, table=0, n_packets=785, n_bytes=47100, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=656.224s, table=0, n_packets=10, n_bytes=980, ip,nw_src=10.0.0.3,nw_dst=10.0.0.9 actions=group:3442284064
 cookie=0x0, duration=656.224s, table=0, n_packets=1, n_bytes=42, priority=1,arp,arp_spa=10.0.0.3,arp_tpa=10.0.0.9 actions=group:3442284064
 cookie=0x0, duration=656.223s, table=0, n_packets=10, n_bytes=980, ip,nw_src=10.0.0.9,nw_dst=10.0.0.3 actions=output:"s2-eth2"
 cookie=0x0, duration=656.223s, table=0, n_packets=5, n_bytes=210, priority=1,arp,arp_spa=10.0.0.9,arp_tpa=10.0.0.3 actions=output:"s2-eth2"
 cookie=0x0, duration=679.225s, table=0, n_packets=81, n_bytes=10778, priority=1,ipv6 actions=drop
 cookie=0x0, duration=679.232s, table=0, n_packets=5, n_bytes=276, priority=0 actions=CONTROLLER:65535
```

Figure 3.5: Flow Entry

Figure 3.5 shows how various types of traffic should be processed and forwarded across the SDN infrastructure. They concentrate on efficiently routing and directing traffic within the specified network topology and policies. These flow entries in the OpenFlow switch specify how packets are treated according to various criteria. The first item routes packets with a defined destination MAC address to the controller. Entries that handle IP traffic between hosts 10.0.0.3 and 10.0.0.9 utilize actions=group: 3442284064 for load balancing or multipath routing. ARP packets between certain hosts are sent to specific ports, IPv6 traffic is dropped, and other packets are routed to the controller, resulting in improved network performance and administration.

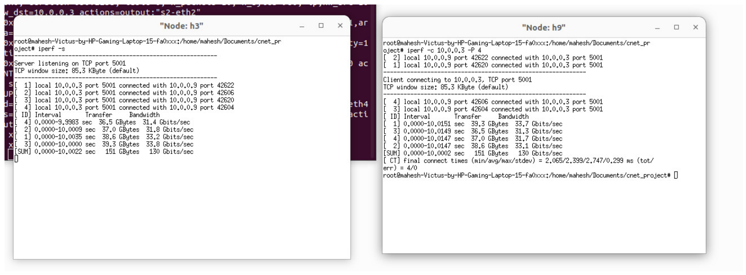


```
mininet> sh ovs-ofctl dump-flows s2
 cookie=0x0, duration=679.229s, table=0, n_packets=785, n_bytes=47100, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=656.224s, table=0, n_packets=10, n_bytes=980, ip,nw_src=10.0.0.3,nw_dst=10.0.0.9 actions=group:3442284064
 cookie=0x0, duration=656.224s, table=0, n_packets=1, n_bytes=42, priority=1,arp,arp_spa=10.0.0.3,arp_tpa=10.0.0.9 actions=group:3442284064
 cookie=0x0, duration=656.223s, table=0, n_packets=10, n_bytes=980, ip,nw_src=10.0.0.9,nw_dst=10.0.0.3 actions=output:"s2-eth2"
 cookie=0x0, duration=656.223s, table=0, n_packets=5, n_bytes=210, priority=1,arp,arp_spa=10.0.0.9,arp_tpa=10.0.0.3 actions=output:"s2-eth2"
 cookie=0x0, duration=679.225s, table=0, n_packets=81, n_bytes=10778, priority=1,ipv6 actions=drop
 cookie=0x0, duration=679.232s, table=0, n_packets=5, n_bytes=276, priority=0 actions=CONTROLLER:65535
```

Figure 3.6: Group Table Entry

Figure 3.6 shows the ovs ofctl dump-groups command output for group id 3442284064. Switch s2 displays a select type group configuration with two buckets. Every bucket has a specific weight: Bucket 0, with a weight of 6, directs all traffic to port s2 eth4, whereas Bucket 1, with a weight of 4, divides traffic between ports s2 eth1 and s2 eth. This setting allows the switch to dynamically distribute incoming traffic based on the provided weights, which helps in improving network performance by balancing the load across different.

$$\text{Bucket Weight} = \left(1 - \frac{\text{Path Weight}}{\text{Sum of Path Weights}}\right) \times 10$$

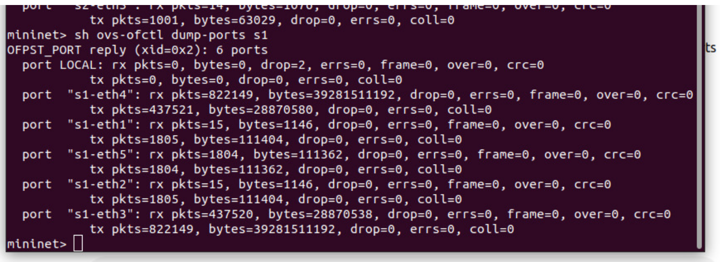


```
"Node: h3"
root@mahesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:/home/mahesh/Documents/cnet_pr
oject# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.3 port 5001 connected with 10.0.0.9 port 42622
[  2] local 10.0.0.3 port 5001 connected with 10.0.0.9 port 42606
[  3] local 10.0.0.3 port 5001 connected with 10.0.0.9 port 42620
[  4] local 10.0.0.3 port 5001 connected with 10.0.0.9 port 42604
[ ID] Interval       Transfer     Bandwidth
[  4] 0.0000-9.9983 sec  36.5 GBytes  31.4 Gbits/sec
[  2] 0.0000-10.0009 sec  37.0 GBytes  31.8 Gbits/sec
[  1] 0.0000-10.0035 sec  38.6 GBytes  33.2 Gbits/sec
[  3] 0.0000-10.0000 sec  39.3 GBytes  33.8 Gbits/sec
[SUM] 0.0000-10.0022 sec   151 GBytes   130 Gbits/sec
```

```
"Node: h9"
root@mahesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:/home/mahesh/Documents/cnet_pr
oject# iperf -c 10.0.0.3 -P 4
[  2] local 10.0.0.9 port 42622 connected with 10.0.0.3 port 5001
[  1] local 10.0.0.9 port 42620 connected with 10.0.0.3 port 5001
------------------------------------------------------------
Client connecting to 10.0.0.3, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.0.0.9 port 42606 connected with 10.0.0.3 port 5001
[  3] local 10.0.0.9 port 42604 connected with 10.0.0.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0151 sec  39.3 GBytes  33.7 Gbits/sec
[  3] 0.0000-10.0149 sec  36.5 GBytes  31.3 Gbits/sec
[  4] 0.0000-10.0147 sec  37.0 GBytes  31.7 Gbits/sec
[  2] 0.0000-10.0147 sec  38.6 GBytes  33.1 Gbits/sec
[SUM] 0.0000-10.0002 sec   151 GBytes   130 Gbits/sec
[ CT] final connect times (min/avg/max/stdev) = 2.065/2.399/2.747/0.299 ms (tot/
err) = 4/0
root@mahesh-Victus-by-HP-Gaming-Laptop-15-fa0xxx:/home/mahesh/Documents/cnet_project# 
```

Figure 3.7: Simulate Server and Client

Figure 3.7 shows the iperf test between hosts h3 and h9 identifies h3 as the server (iperf -s) and h9 as the client (iperf -c 10.0.0.3 -P 4). Multiple concurrent TCP connections were created between them, allowing for high-speed data transfers with bandwidth ranging from around 31.3 Gbits/sec to 33.8 Gbits/sec. This indicates strong network performance between h3 and h9, exhibiting efficient use of network resources for data delivery.



```
            tx pkts=1001, bytes=63029, drop=0, errs=0, coll=0
mininet> sh ovs-ofctl dump-ports s1
OFPST_PORT reply (xid=0x2): 6 ports
  port LOCAL: rx pkts=0, bytes=0, drop=2, errs=0, frame=0, over=0, crc=0
            tx pkts=0, bytes=0, drop=0, errs=0, coll=0
  port  "s1-eth4": rx pkts=822149, bytes=39281511192, drop=0, errs=0, frame=0, over=0, crc=0
            tx pkts=437521, bytes=28870580, drop=0, errs=0, coll=0
  port  "s1-eth1": rx pkts=15, bytes=1146, drop=0, errs=0, frame=0, over=0, crc=0
            tx pkts=1805, bytes=111404, drop=0, errs=0, coll=0
  port  "s1-eth5": rx pkts=1804, bytes=111362, drop=0, errs=0, frame=0, over=0, crc=0
            tx pkts=1804, bytes=111362, drop=0, errs=0, coll=0
  port  "s1-eth2": rx pkts=15, bytes=1146, drop=0, errs=0, frame=0, over=0, crc=0
            tx pkts=1805, bytes=111404, drop=0, errs=0, coll=0
  port  "s1-eth3": rx pkts=437520, bytes=28870538, drop=0, errs=0, frame=0, over=0, crc=0
            tx pkts=822149, bytes=39281511192, drop=0, errs=0, coll=0
mininet> 
```

Figure 3.8: Server and Client

Figure 3.8 shows The ovs ofctl dump-ports s1 command output for switch s1 in Mininet revealing that data is being transmitted and received across its ports (s1 eth1 to s1 eth5). Notably, s1 eth4 and s1 eth3 manage high traffic levels, sending 437521 and 822149 packets, respectively, with appropriate byte counts. These statistics show stable network functioning with no reported packet drops, errors, or collisions (errs or coll), indicating effective data flow management and normal network circumstances.

$$\text{REFERENCE\_BW\_Bps} = \frac{10000000 \text{ bps}}{8} = 1250000 \text{ Bps}$$

$$\text{Threshold} = \text{REFERENCE\_BW\_Bps} \times 0.7 = 1250000 \times 0.7 = 875000 \text{ Bps}$$

The ratios for each port (s2 eth1: 31,448.23 Bps, s2 eth2: 130,373.27 Bps, s2 eth3: 0.051 Bps, and s2 eth4: 98,925.19 Bps) are calculated by adding the total bytes received and transmitted and dividing by the reference bandwidth (REFERENCE BW Bps = 1,250,000 Bps). This computation compares current traffic demand in bytes per second to available capacity. The barrier is set at 70 percent of REFERENCE BW Bps, which equals 875,000 Bps. By comparing each port's ratio to this threshold, it is clear that all ports are working efficiently at less than 70 percent capacity, ensuring excellent network performance and appropriate bandwidth control.

4 Conclusion

In this project we have managed various types of traffic, including ARP, IP, and Ethernet frames, and have reduced the path cost and bucket weight related to Quality of Service (QoS). Although QoS indirectly supports this project by enhancing overall performance, we observed that the average time increased when conducting a normal ping test, compared to using the load balancing code. By using the load balancing code we have achieved the network performance, congestion, and reliability. Implementing an SDN-based load balancing solution with Mininet and the Ryu controller results in a considerable improvement in traffic management and network performance. our project effectively illustrates the ability of SDN to dynamically control network traffic by utilizing advanced load-balancing algorithms and efficient traffic steering policies. We improved resource utilization and network resiliency by developing a simulated network topology in Mininet and then using the Ryu controller to compute optimal pathways and distribute traffic accordingly. This solution solves the inefficiencies of traditional networking systems by delivering a scalable, adaptive, and high-performance network architecture.

5 Future scope

The future sphere for this SDN-located load balancing resolution is hopeful. As network demands evolve, enhancing the current exercise accompanying progressive load adjust algorithms and integrating machine intelligence for traffic indicator and mechanization is crucial. Realworld deployment will validate performance in enterprise networks, data centers, and Internet of Things (IoT) atmospheres. Security augmentations, containing advanced warning discovery, are essential for strong activities.. Interoperability accompanying SDN controllers and network instruments will aid adoption. Integrating edge and cloud controls can sharpen traffic allocation and correct use conduct. These progress will invigorate SDN's role in designing effective, flexible network infrastructures for developing digital surroundings to a degree smart metropolises, autonomous cars, and telemedicine.

References

[1] Yassin Abdulkarim Hamdalla Omer, Amin Babiker A. Mustafa, and Ashraf G. Abdalla. Performance analysis of round robin load balancing in sdn. In *2020 International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)*, pages 1–5, 2021.

[2] Senthil Prabakaran and Ramalakshmi Ramar. Software defined network: load balancing algorithm design and analysis. *Int. Arab J. Inf. Technol.*, 18(3):312–318, 2021.

[3] Kannan Govindarajan and Vivekanandan Suresh Kumar. An intelligent load balancer for software defined networking (sdn) based cloud infrastructure. In *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–6, 2017.

[4] Hajar Hantouti and Nabil Benamar. A novel sdn-based architecture and traffic steering method for service function chaining. In *2018 International Conference on Selected Topics in Mobile and Wireless Networking (MoWNeT)*, pages 1–8, 2018.

[5] Kai-Yu Wang, Shang-Juh Kao, and Ming-Tsung Kao. An efficient load adjustment for balancing multiple controllers in reliable sdn systems. In *2018 IEEE International Conference on Applied System Invention (ICASI)*, pages 593–596, 2018.

[6] Kenji Hikichi, Toshio Soumiya, and Akiko Yamada. Dynamic application load balancing in distributed sdn controller. In *2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 1–6, 2016.