COMPUTER NETWORKS PROJECT
GROUP-8

NAME	ROLL NUMBER
BOLLA NEHASREE	CS22BTECH11012
MANE POOJA VINOD	CS22BTECH11035
NALAVOLU CHETANA	CS22BTECH11042
NETHI KEERTHANA	CS22BTECH11043
SURBHI	CS22BTECH11057
TUMARADA PADMAJA	CS22BTECH11059

Problem Statement:-

Efficient Load Balancing Mechanism in software defined Networks

Objective:-

Implement a response time-based load balancing strategy using an OpenFlow switch and evaluate its performance compared to RR and Random Strategies.

Commands to run:-

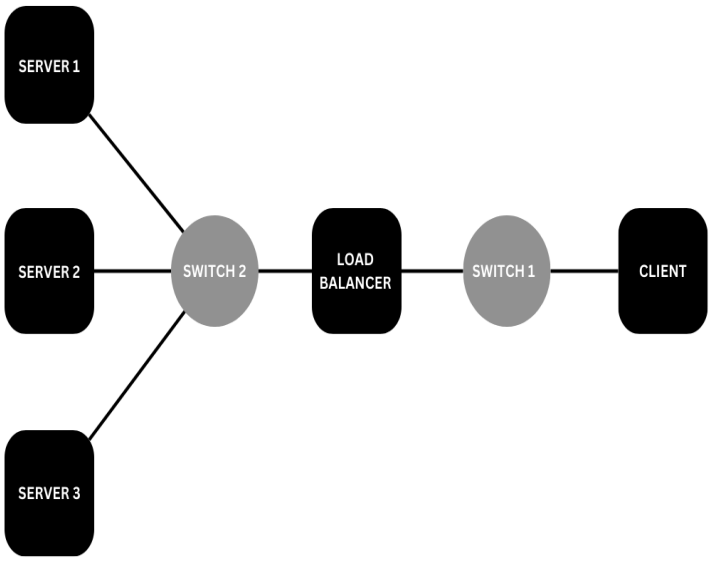
To open mininet with a custom we have to write : sudo mn --custom topology.py --topo loadbalancer --controller=default --ipbase=10.0.0.0/24

In mininet:

- xterm server1
- xterm server2
- xterm server3
- xterm lb
- xterm client

In server1 : python3.6 backend_server1.py
In server2: python3.6 backend_server2.py
In server3: python3.6 backend_server3.py
In lb: python3.6 load_balancer.py
In client: python3.6 client.py

System Block-diagram:-



Code Explanation:-

Server:

Simulates a backend server for load balancing.

- ➔ **Configuration:**
  Server1 IP: 10.0.0.2, Port: 8081.
  Server2 IP: 10.0.0.3, Port: 8082.
  Server3 IP: 10.0.0.4, Port: 8083.
  TCP server (socket.AF_INET, socket.SOCK_STREAM).
- ➔ **Core Functions**:
  Accepts client connections via load balancer and logs requests.
  Send a static HTTP response: Welcome to Server 1.
- ➔ **Connection Handling**:

Iterative processing (no concurrency).
Closes client connection after responding.

➔ **Limitations**:
There is no generalized server generation for dynamic topology.

Client:
Sends 100 simulated HTTP GET requests to a load balancer for testing.

➔ **Configuration**:
Load Balancer IP: 10.0.0.100.
Port: 8888.
Request: Basic HTTP GET request.

➔ **Request Handling**:
Iteratively sends requests (1 to 100).
Prints request and response for each interaction.
Handles errors gracefully with exception handling

➔ **Socket Usage**:
Creates a new socket (socket.AF_INET, socket.SOCK_STREAM) for each request.
Closes the socket after processing each request.

➔ **Delay Mechanism:**
Introduces a 0.1-second delay between requests to avoid overwhelming the load balancer.

➔ **Limitation**:
There is no generalized server generation for dynamic topology. We introduced multiple threads instead of multiple clients

Load_balancer:
Code implements a Load Balancer that forwards client requests to a set of backend servers. It supports three different load balancing algorithms: **Round Robin, Random, and Least Response Time.**

Load Balancing Algorithms:

- **Round Robin**: Distributes requests to servers in a cyclic order.
- **Random**: Randomly selects a backend server for each request.
- **Least Response Time**: Selects the server with the fastest response time.

Variables:

- **backend_servers:** List of backend server IPs and ports.

- **current_server**: Tracks the server in use for Round Robin.
- **SERVER_LOAD**: List tracking the load (active requests) on each server.
- **CLIENT_QUEUE** and **SERVER_QUEUE**: Queues for managing requests and responses.
- **request_metrics**: Stores metrics for throughput, response time, and latency.
- **queue**: Used to manage requests waiting to be processed by each server.

Functions:

- **is_server_available(ip, port)**: Checks if a backend server is available via socket connection.
- **get_response_time(ip, port)**: Measures the round-trip response time of a server using a "PING".
- **server_queues**: Dictionary where each key corresponds to a backend server index, and the value is a queue that stores client connections waiting to be processed by that server.
- **find_best_server()**: Selects the server with the least load.
- **forward_request(client_socket, server_ip, server_port, session_metrics)**: Forwards client requests to the selected server and tracks metrics.
- **update_response_time**: Calculates the average response time for the given server by updating the entry in SERVER_RESPONSE_TIMES.

Load Balancing Handlers:

- **round_robin_handler(client_socket, session_metrics):** Implements Round Robin load balancing.
- **random_handler(client_socket, session_metrics):** Implements Random load balancing.
- **least_response_time_handler(client_socket, session_metrics):** Implements Least Response Time method.
- **process_queue**: Processes client requests from a queue associated with a specific server. If the server is available, it forwards the request; otherwise, it retries later by re-queuing the request.

Load Balancer Function (load_balancer):

- Listens for incoming client connections, forwards requests to backend servers using the selected method, and tracks metrics (latency, response time, throughput).
- Handles each connection in a separate thread for concurrency.
- Calculates and logs session-level metrics (averages of latency, response time, and throughput) after each client request.
- Updates overall metrics across all sessions.

Threading:

- Utilizes threads to handle multiple concurrent client connections, improving scalability.

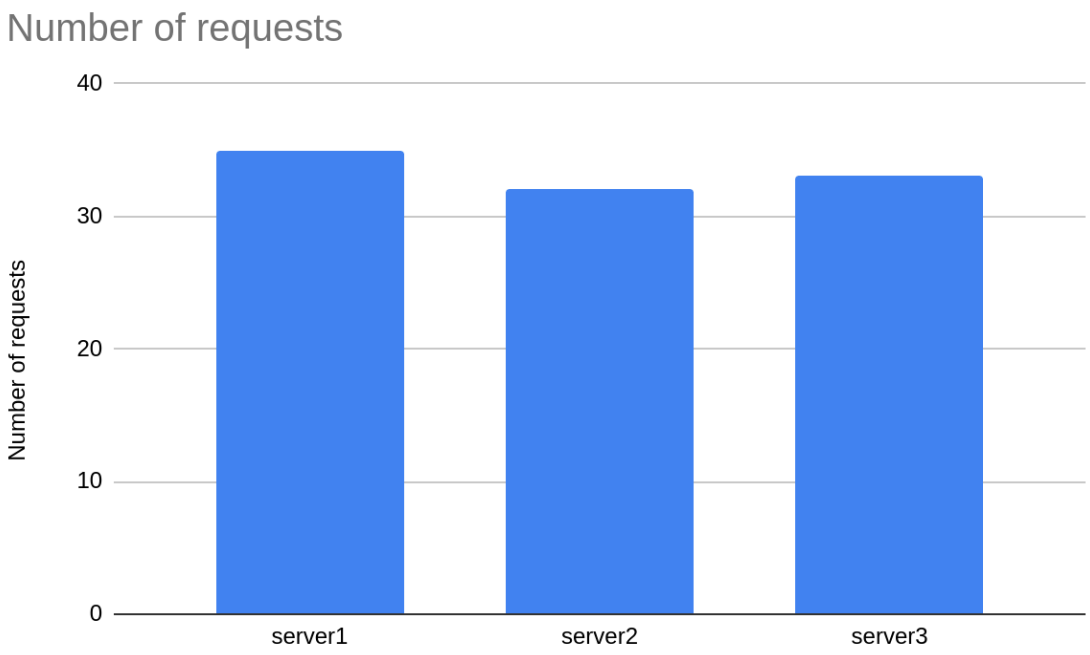
Metrics Calculation:

- Tracks per-request metrics like response time, latency, and throughput.
- Calculates session-level and overall metrics.
- Logs metrics for performance evaluation.

GRAPHS AND METRICS:

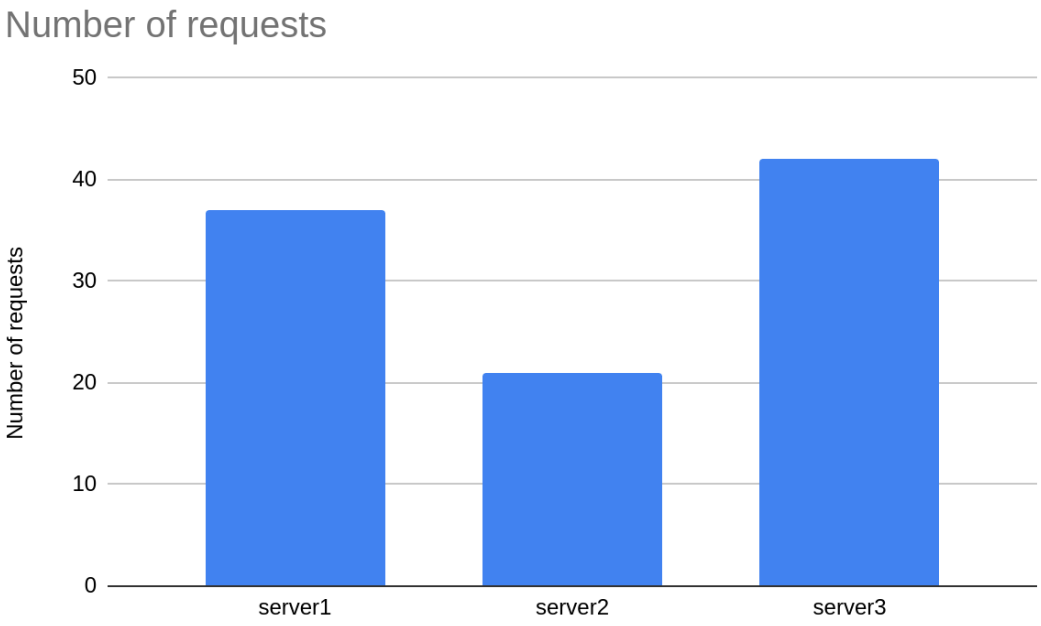
CASE1: All Servers are up and running

Number of requests Vs Servers:

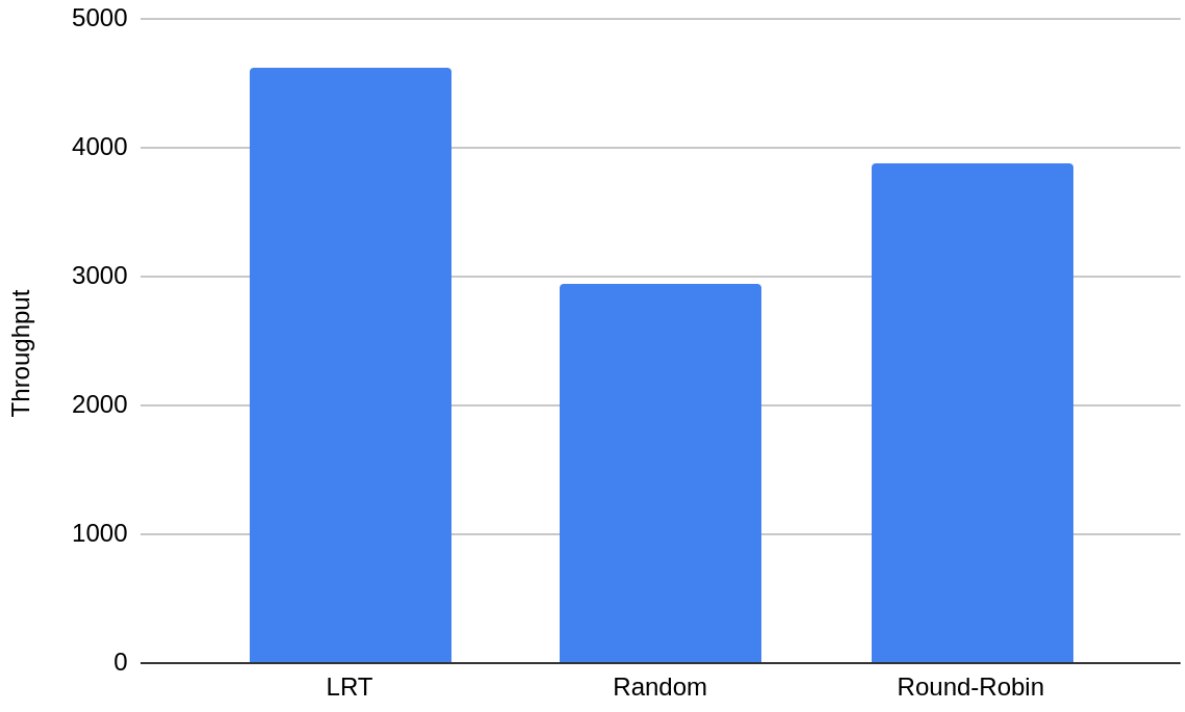


CASE2: Server 2 is randomly kept in sleep for 1 second:

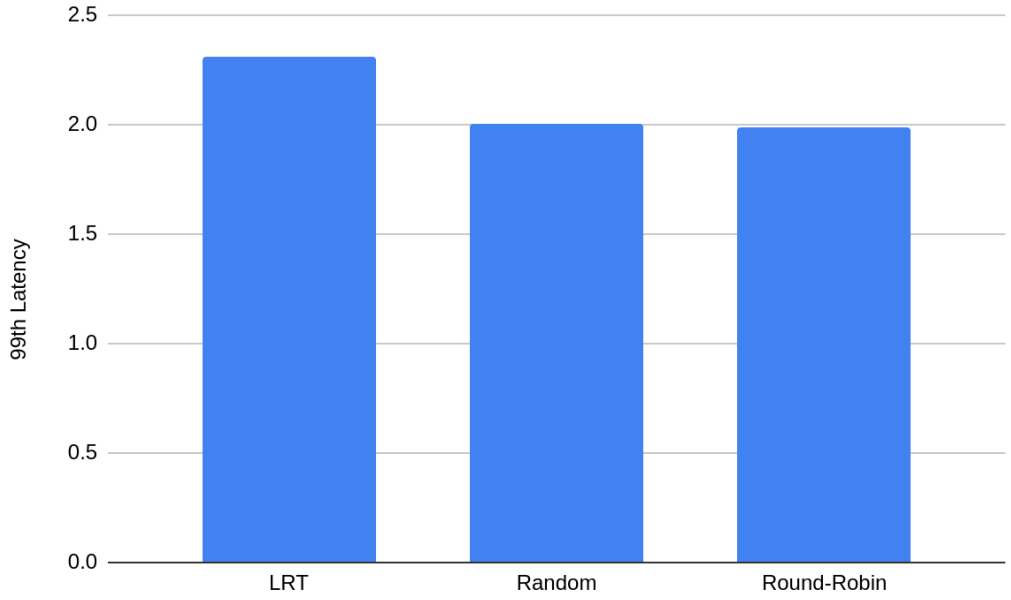
Number of requests Vs Servers:



Throughput graph:



Latency graph:



Observations:

CASE1:

The number of requests are similar over all the servers.

CASE2:

The number of requests in server2 are less than server1 and server3 and server3 has received the most number of requests.

The latency and throughput of Least Response time is higher than the algorithms Random and Round-Robin.

Analysis:

CASE1:

All servers have similar throughput because all the servers have similar networking conditions.

CASE2:

Since server 2 is kept in sleep randomly for a second it has less number of requests compared to server1 and 3.In this case the queue of server 2 is constant at a value whereas 1 and 3 handles the requests which leads to decrease in queue size.So servers 1 and 3 are preferred over server 2.