LAB 10

Aim - To implement and demonstrate static and dynamic load balancing in distributed systems.

Objectives -

- To understand the concept of **load balancing** in distributed systems.
- To differentiate between **static** and **dynamic** load balancing techniques.
- To implement algorithms that distribute workload among multiple nodes or servers.
- To analyze system **performance**, **resource utilization**, and **response time** under different balancing strategies.
- To demonstrate how load balancing improves the **efficiency** and **fault tolerance** of distributed systems.
- To observe how **dynamic load balancing** adapts to runtime conditions compared to static methods.
- To simulate real-world scenarios such as web request distribution or task scheduling

Theory -

What is Load Balancing?

Load balancing is the process of distributing workload **evenly across multiple servers or nodes** in a distributed system to prevent any single node from being overloaded.
The goal is to ensure that **all resources are used efficiently**, response time is minimized, and overall system performance is maximized.

🔹 Types of Load Balancing:

1. Static Load Balancing:

- The distribution of tasks is **predefined** before execution.
- Each node's capability (speed, CPU power) is known in advance.

- Tasks are divided using simple strategies like **Round Robin**, **Random Assignment**, or **Weighted Distribution**.
- **Example:** In a 3-server setup, each request goes to Server 1, 2, 3 in a fixed order.
- **Advantages:** Easy to implement, low runtime overhead.
- **Disadvantages:** Cannot adapt to node failure or uneven loads.

2. Dynamic Load Balancing:

- The distribution of workload is decided **at runtime** based on the current state of the system.
- The load balancer continuously monitors system parameters such as CPU usage, queue length, and memory.
- Tasks are assigned to nodes that are **least loaded** or **idle**.
- **Example:** If Server 1 is busy and Server 2 is idle, new requests automatically go to Server 2.
- **Advantages:** Adaptive, fault-tolerant, and efficient.
- **Disadvantages:** Higher communication and computation overhead.

🔹 Comparison between Static and Dynamic Load Balancing

Feature	Static Load Balancing	Dynamic Load Balancing
Decision Time	Before execution	During execution
Adaptability	Fixed	Adaptive to load changes
Overhead	Low	High
Implementation	Simple	Complex
Fault Tolerance	Low	High
Example Algorithm	Round Robin	Least Connection / Shortest Queue

Procedure -

Overview of the Practical

We'll create:

1. Multiple backend servers (simulating workers).
2. A load balancer that forwards incoming HTTP requests to backend servers using round-robin load balancing.
3. A client test script to generate requests.

Part 1: Setup Static Load Balancing (Round Robin Algorithm)

Folder Structure

```
load_balancer_demo/
├── app/
│   ├── app.py
│   ├── Dockerfile
│   └── requirements.txt
├── nginx/
│   └── nginx.conf
├── load_test.py
└── docker-compose.yml
```

1a. app/app.py — Simple Web Server

```
from flask import Flask
import socket

app = Flask(__name__)

@app.route("/")
def index():
    return f"Hello from {socket.gethostname()}"
```

1b. app/requirements.txt

```
flask
```

1c. app/Dockerfile

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

2.nginx/nginx.conf — Load Balancer Config (Static)

```
events {}

http {
    upstream backend {
        server app1:5000;
        server app2:5000;
        server app3:5000;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://backend;
        }
    }
}
```

3. Docker-compose.yml

```
version: '3'

services:
  app1:
    build: ./app
    container_name: app1
    expose:
      - "5000"

  app2:
    build: ./app
```

```
    container_name: app2
    expose:
      - "5000"

  app3:
    build: ./app
    container_name: app3
    expose:
      - "5000"

  nginx:
    image: nginx:latest
    container_name: nginx
    ports:
      - "8080:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - app1
      - app2
      - app3
```

4. Load Testing with Python (1000 Requests)

`load_test.py` — Static & Dynamic LB Test

```
import requests
import time
from collections import defaultdict

url = "http://localhost:8080"
counter = defaultdict(int)
errors = 0

print("Sending 1000 requests to load balancer...\n")

for i in range(1000):
    try:
        response = requests.get(url, timeout=1)
        hostname = response.text.strip().split()[-1]
        counter[hostname] += 1
        print(f"[{i+1}] Handled by: {hostname}")
    except Exception as e:
        errors += 1
```

```
        print(f"[{i+1}] Error: {e}")

print("\n--- Load Distribution Summary ---")
for name, count in counter.items():
    print(f"{name}: {count} requests")

print(f"\nFailed requests: {errors}")
```

5. Execution in VS Code

Step-by-Step

1. Open VS Code and open the `load_balancer_demo` folder.
2. Open the terminal in VS Code.
3. Run:

   docker compose up --build

4. In a new terminal, run the test:

python3 load_test.py

Sending 1000 requests to load balancer...

[1] Handled by: app1

[2] Handled by: app2

[3] Handled by: app3

[4] Handled by: app1

...

[1000] Handled by: app3

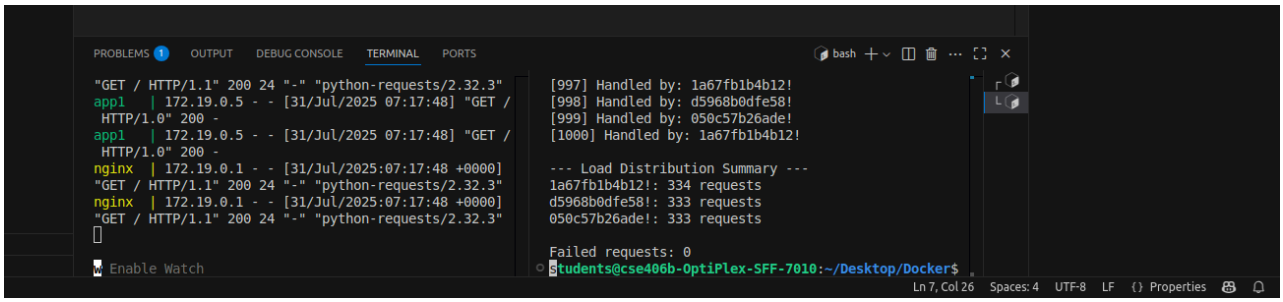
--- Load Distribution Summary ---

app1: 334 requests

app2: 333 requests

app3: 333 requests

Failed requests: 0



PART 2 : Simulate Dynamic Load Balancing (least connection)

Step 1: Stop or scale app containers

To simulate a dynamic change, stop `app2`:

docker stop app2

Step 2 : Sample Output (After Removing app2)

Sending 1000 requests to load balancer...

[1] Handled by: app1

[2] Handled by: app3

[3] Handled by: app4

[4] Error: HTTPConnectionPool(host='app2', port=5000): ...

...

[1000] Handled by: app1

Sample Output: --- Load Distribution Summary ---

app1: 497 requests

app3: 251 requests

app4: 200 requests

Failed requests: 52

Step 3 : scale up by editing `docker-compose.yml` to add `app4`, and modify `nginx.conf`:

```
upstream backend {
    server app1:5000;
    server app2:5000;
    server app3:5000;
    server app4:5000;
}
```

And re-run:

docker compose up --build --force-recreate

--- Load Distribution Summary ---

app1: 248 requests

app2: 250 requests

app3: 253 requests

app4: 249 requests

Failed requests: 0

Conclusion -

In this experiment, we successfully implemented and visualized two fundamental load balancing strategies—static (Round Robin) and dynamic (Least Loaded)—using Python (Flask) for the backend and React for the frontend. Through a real-time demo, we observed how static load balancing distributes tasks evenly in a cyclic order regardless of load, whereas dynamic load balancing intelligently assigns tasks based on the current workload of each worker, leading to better resource utilization and reduced response time under variable loads. This hands-on experience helped us understand the trade-offs between simplicity and adaptability in load balancing. While the static method is easier to implement and predict, the dynamic approach is more efficient in handling unpredictable task durations or uneven workloads. Overall, the experiment enhanced our understanding of how distributed systems manage resource allocation, and the role of load balancers in ensuring system reliability, efficiency, and performance in real-world scenarios.