

DC EXP6

Lab Experiment: Load Balancing in Distributed Systems

1. Title

Implementation of Load Balancing Algorithms in Distributed Systems

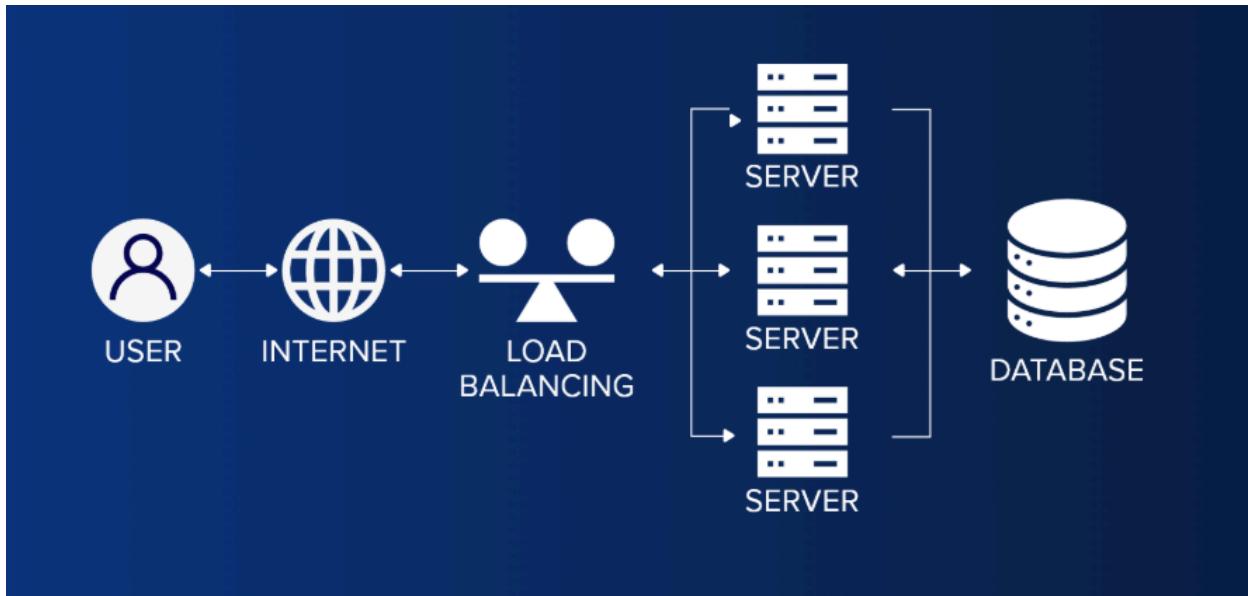
2. Aim

To implement and demonstrate **static** and **dynamic load balancing strategies** in a distributed environment, and evaluate their effect on system performance, throughput, and response time.

3. Case Study

Scenario:

In modern distributed systems, client requests must be efficiently distributed across multiple servers to prevent overload and ensure high availability. For example, large-scale platforms like Netflix, Google Cloud, and Amazon Web Services use advanced load balancers to deliver seamless performance under millions of concurrent requests.



Why Load Balancing?

Without load balancing, a single server might become overloaded while others remain idle, leading to high latency and failures. Load balancing ensures:

- **Fair distribution** of requests.
- **Scalability** by adding more servers easily.
- **High availability** through rerouting during server failures.

Real-life Example:

- **E-commerce platforms (Flipkart, Amazon)**: Load balancers distribute customer traffic during peak sales.
- **YouTube / Netflix**: Distribute video streaming requests across multiple content delivery servers.

4. Implementation

a) Environment Setup

- Language: Python (Flask, Requests library) / nodejs.
- Backend servers simulate workloads with random processing times.
- Load balancers forward client requests using chosen strategies.

- Client program generates traffic and measures metrics.

b) Load Balancing Algorithms to implement

1. **Round Robin (Static)**: Requests assigned cyclically.
2. **Random (Static)**: Requests assigned randomly.
3. **Least Connections (Dynamic)**: Request sent to server with the fewest active connections.
4. **Weighted Response Time (Dynamic)**: Request sent to server with the best response performance.

c) Example – Least-connections (Pseudo-code)

```

from flask import Flask, request
import requests, time, random

app = Flask(__name__)

#dummy servers
servers = [
    {"url": "http://localhost:5000", "connections": 0},
    {"url": "http://localhost:5001", "connections": 0},
    {"url": "http://localhost:5002", "connections": 0}
]

# Choose the server with the fewest active connections
def least_connections():
    return min(servers, key=lambda x: x["connections"])

@app.route('/process', methods=['GET'])
def balance():
    server = least_connections()
    server["connections"] += 1 # increase active count
    try:
        start_time = time.time()
    
```

```

# Forward request to chosen server
response = requests.get(f"{server['url']}/process", timeout=10)
elapsed = time.time() - start_time
print(f"Forwarded to {server['url']} | Response Time: {elapsed:.3f}s")
return response.json()

except requests.RequestException as e:
    return {"error": str(e)}, 503
finally:
    server["connections"] -= 1 # release connection after response

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8001)

```

5. Procedure

1. Start multiple backend servers ([server.py](#)) to simulate processing workloads.
2. Run the **static load balancer**.
3. Run the **dynamic load balancer**.
4. Execute the client ([client.py](#)) to send requests using each strategy:
 - Round Robin
 - Random
 - Least Connections
 - Weighted Response Time
5. Collect and compare metrics:
 - Requests served per server.
 - Average response time.
 - Load imbalance (standard deviation of request distribution).

6. Expected Output

- **Round Robin:** Perfectly balanced distribution; ignores server performance.
 - **Random:** Uneven distribution; higher imbalance.
 - **Least Connections:** Balances active connections but may have spikes.
 - **Weighted Response Time:** Improves latency but may overload faster servers.
-

7. Problem Statements

1. Implement a **Hash-based Load Balancer** (e.g., IP Hash).
 2. Modify Weighted Response to adapt dynamically with moving averages.
 3. Compare **static vs. dynamic load balancing** under high concurrency.
 4. Extend implementation with **server failure detection** and auto rerouting.
 5. Simulate **real-time chat application load balancing** across servers.
-

8. Case Studies

1. **Netflix (Dynamic Load Balancing)**
 - Uses **Zuul** and **Ribbon** for intelligent traffic routing.
 - Balances millions of video requests worldwide in real time.
2. **Amazon Elastic Load Balancer (ELB)**
 - Offers Application (ALB) and Network (NLB) load balancing.
 - Supports **round-robin, least connections, and health checks**.
3. **Google Cloud Load Balancer**
 - Provides **global load balancing** across regions.
 - Uses **latency-based routing** for faster user experience.
4. **Akamai CDN**
 - Uses **DNS-based load balancing** to direct users to the nearest edge server.
 - Ensures reduced latency in video streaming and web services.