Final Report: gRPC Look-Aside Load Balancing Assignment

1. Introduction

This assignment demonstrates a distributed client–server system implemented using gRPC with a look-aside load balancing model. The system is composed of three major components:

- **Load Balancer (LB) Server:**
  Maintains an up-to-date list of available backend servers, receives registration and periodic load updates, and selects the most appropriate server to handle incoming client requests based on configurable load balancing policies.

- **Backend Servers:**
  These servers register themselves with the LB server and periodically report their current load (simulated for testing). They process client computational requests (a dummy task such as reversing a string).

- **Clients:**
  Clients first query the LB server for a suitable backend server and then directly send their compute tasks to the chosen server.

Additionally, two scale testing scripts were developed:

- **scale_servers.py:** Spawns multiple backend server processes (allowing simulation of 10 or 15 servers).
- **scale_clients.py:** Simulates 100 concurrent client requests to evaluate performance by measuring response times, throughput, and backend usage distribution.

---

2. Service Definition and RPCs

a. Protocol Buffers and gRPC Service Definitions

Our services are defined in the `load_balancer.proto` file. The key messages and service interfaces include:

- **Messages:**
  - **BackendInfo:** Contains details of a backend server (unique ID, IP address, port, and a load metric).
  - **RegisterResponse:** Sent by the LB server to acknowledge successful registration or load update.
  - **LoadReport:** Used by backend servers to report their current load.
  - **ServerRequest:** Contains a client identifier and the requested service name; sent by clients to the LB server.
  - **TaskRequest/TaskResponse:** Define the dummy compute task and its corresponding result.
- **Services:**
  - **LoadBalancer Service:**
    Provides three RPCs:
    - `RegisterBackend(BackendInfo) returns (RegisterResponse)` – for backend registration.
    - `ReportLoad(LoadReport) returns (RegisterResponse)` – for backend load reporting.
    - `GetBackend(ServerRequest) returns (BackendInfo)` – for clients to query for an optimal backend.
  - **Compute Service:**
    Provides `ComputeTask(TaskRequest) returns (TaskResponse)` for processing compute tasks.

b. Design Rationale

Using Protocol Buffers ensures a strongly typed and language-independent interface. This modular design clearly separates responsibilities:

- The LB server manages dynamic backend discovery and selection.
- Backend servers focus on processing tasks and reporting their load.
- Clients remain simple by delegating the decision of which backend to contact to the LB server.

---

3. Load Balancer Functionality

The LB server (implemented in **lb_server.py**) provides the following key functionalities:

a. Maintaining a List of Available Servers

A global list, `backend_servers`, holds the details of all registered backend servers. This list is updated as servers register or report updated load, with thread safety ensured by a lock.

b. Server Registration and Load Reporting

Registration:
When a backend server starts, it calls the `RegisterBackend` RPC. If the server is not already registered, its details are appended to the list:

```
if not exists:

    print(f"Registering backend: {request.id} at 
{request.address}:{request.port}")

    backend_servers.append(request)
```

Load Reporting:
Backend servers periodically invoke the `ReportLoad` RPC to update their load metric. The LB server then updates the corresponding record in the list:

```
backend_servers[i] = load_balancer_pb2.BackendInfo(

    id=backend.id,

    address=backend.address,

    port=backend.port,

    load=request.load

)
```

c. Responding to Client Queries

When a client calls the `GetBackend` RPC, the LB server selects an appropriate backend based on the configured load balancing policy and returns its details. This dynamic selection is critical for effectively distributing client requests.

---

4. Load Balancing Policies

The LB server supports three load balancing policies, selectable via the CLI (`--policy`). These policies determine how client requests are distributed among backend servers.

a. Pick First Policy

Definition & Implementation:
The "Pick First" policy always selects the first backend in the list, irrespective of its load. This is the simplest approach and works well in homogeneous and low-load scenarios.

```
if self.policy == "pick_first":

    selected = backend_servers[0]
```

b. Round Robin Policy

Definition & Implementation:
The "Round Robin" policy cycles sequentially through the list of backend servers, ensuring each server receives an equal share of requests.

```
 elif self.policy == "round_robin":

    selected = backend_servers[self.round_robin_index]

    self.round_robin_index = (self.round_robin_index + 1) %
len(backend_servers)
```

c. Least Load Policy

Definition & Implementation:
The "Least Load" policy selects the backend server with the lowest current load. In our current implementation, if multiple backends have nearly equal minimal load (within a defined tolerance), one is chosen at random. This approach distributes the requests among multiple servers with low load, rather than concentrating all requests on a single server.

```
elif self.policy == "least_load":

    min_load = min(b.load for b in backend_servers)

    tolerance = 0.001

    candidates = [b for b in backend_servers if abs(b.load - min_load)
< tolerance]

    if candidates:

        selected = random.choice(candidates)

    else:

        selected = min(backend_servers, key=lambda b: b.load)
```

5. Scale Testing and Analysis

a. Test Setup

We evaluated system performance using two scenarios:

1. **Test Case 1:**
   - **Configuration:** 10 backend servers and 100 concurrent clients.
   - **Scripts Used:**
     - **scale_servers.py:** Spawns 10 backend server processes.
     - **scale_clients.py:** Simulates 100 concurrent client requests.
2. **Test Case 2:**
   - **Configuration:** 15 backend servers and 100 concurrent clients.
   - **Scripts Used:**
     - **scale_servers.py:** Spawns 15 backend server processes.
     - **scale_clients.py:** Simulates 100 concurrent client requests.

Definitions:

- **Response Time:** The time elapsed from when a client sends a request to when it receives a response. This metric indicates how quickly the system processes requests.
- **Throughput:**Throughput is generally defined as the overall rate at which the system (including both the client and server components) processes requests. In our context, it measures how many client requests are successfully processed per second by the entire system—from when a client sends a request, the LB routes it, and the backend returns a response. So, it encompasses both the client-side and the server-side contributions.
- **Load Distribution:** Refers to how evenly client requests are spread across the available backend servers. Balanced load distribution prevents any single server from becoming a bottleneck.

b. Test Results Summary

Test Case 1 (10 Servers, 100 Clients):

- **Pick First Policy:**

  - Average Response Time: 6.701 sec
  - Max Response Time: 12.780 sec
  - Min Response Time: 0.526 sec
  - Throughput: 7.49 req/sec
  - Backend Usage: All 100 requests routed to *backend6*

- **Round Robin Policy:**
    - Average Response Time: 1.235 sec
    - Max Response Time: 1.951 sec
    - Min Response Time: 0.546 sec
    - Throughput: 36.16 req/sec
    - Backend Usage: exactly 10 requests per backend server
- **Least Load Policy:**
    - Average Response Time: 1.234 sec
    - Max Response Time: 2.198 sec
    - Min Response Time: 0.521 sec
    - Throughput: 35.28 req/sec
    - backend_usage_least_load = {
    - "backend3": 7, "backend8": 6, "backend6": 11, "backend4": 13, "backend5": 8,
    - "backend1": 10, "backend9": 12, "backend2": 13, "backend7": 9, "backend10": 11}

Test Case 2 (15 Servers, 100 Clients):

- **Pick First Policy:**
    - Average Response Time: 7.013 sec
    - Max Response Time: 15.544 sec
    - Min Response Time: 0.526 sec
    - Throughput: 6.14 req/sec
    - Backend Usage: All 100 requests routed to *backend2*
- **Round Robin Policy:**
    - Average Response Time: 2.801 sec
    - Max Response Time: 4.873 sec
    - Min Response Time: 0.563 sec
    - Throughput: 17.50 req/sec
    - Backend Usage: Requests distributed roughly evenly (e.g., *backend9: 7, backend5: 7, backend3: 7, backend8: 7, backend13: 6, etc.*)
- **Least Load Policy:**
    - Average Response Time: 1.254 sec
    - Max Response Time: 1.999 sec
    - Min Response Time: 0.562 sec
    - Throughput: 38.61 req/sec
    - Backend Usage: backend_usage_least_load = {"backend3": 9, "backend8": 11, "backend15": 8, "backend6": 6, "backend4": 5, "backend2": 10, "backend10": 9,"backend14": 6, "backend5": 7, "backend1": 7, "backend11": 4, "backend13": 7, "backend9": 5, "backend7": 5, "backend12": 1}

c. How the LB Server Manages Load

The LB server dynamically maintains the list of backend servers along with their load metrics. When a client issues a query:

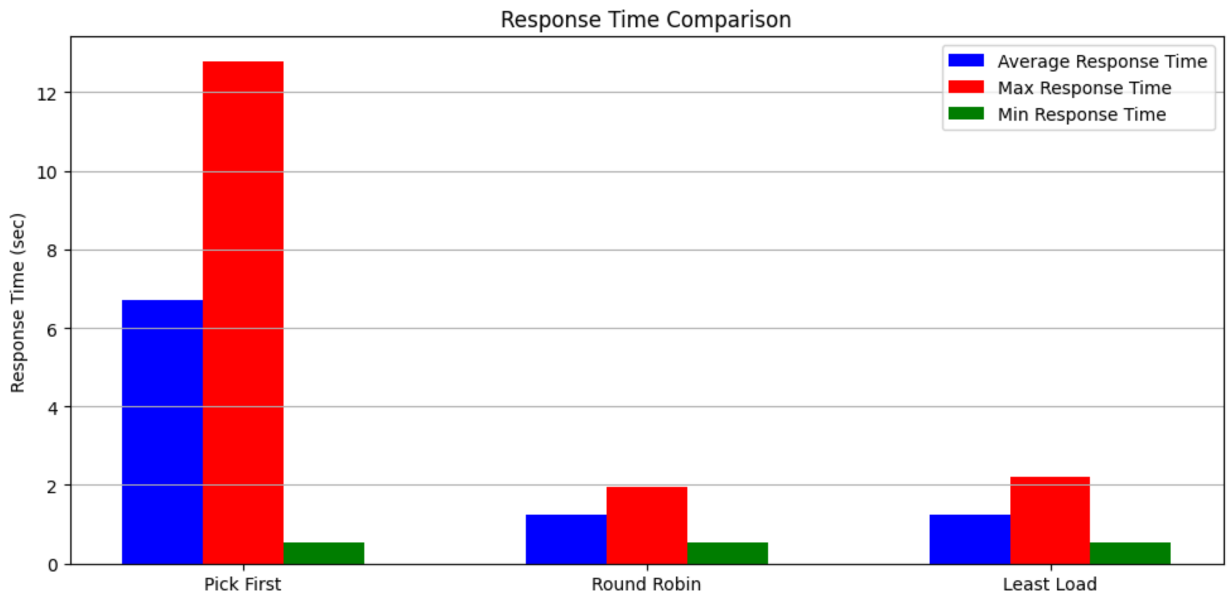
- The **Pick First** policy routes all requests to the first server, leading to potential overload.
- The **Round Robin** policy cycles through all available servers, resulting in balanced load distribution.
- The **Least Load** policy now distributes requests among multiple servers that have nearly equal minimal load, preventing a single server from being overwhelmed.

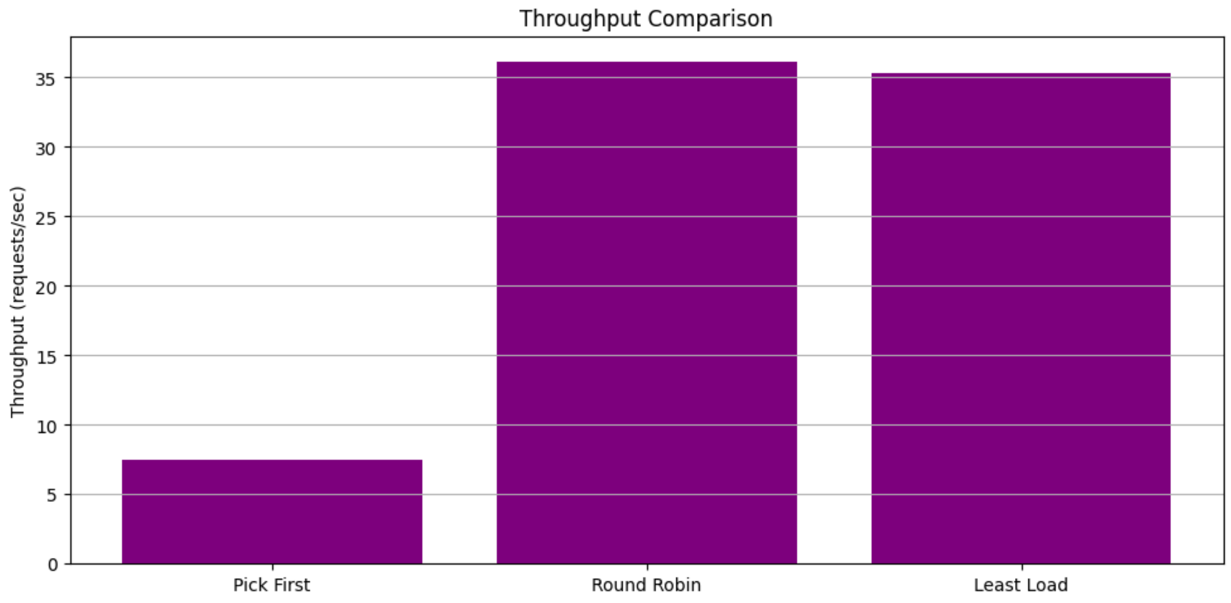
e. Scale Testing Scripts

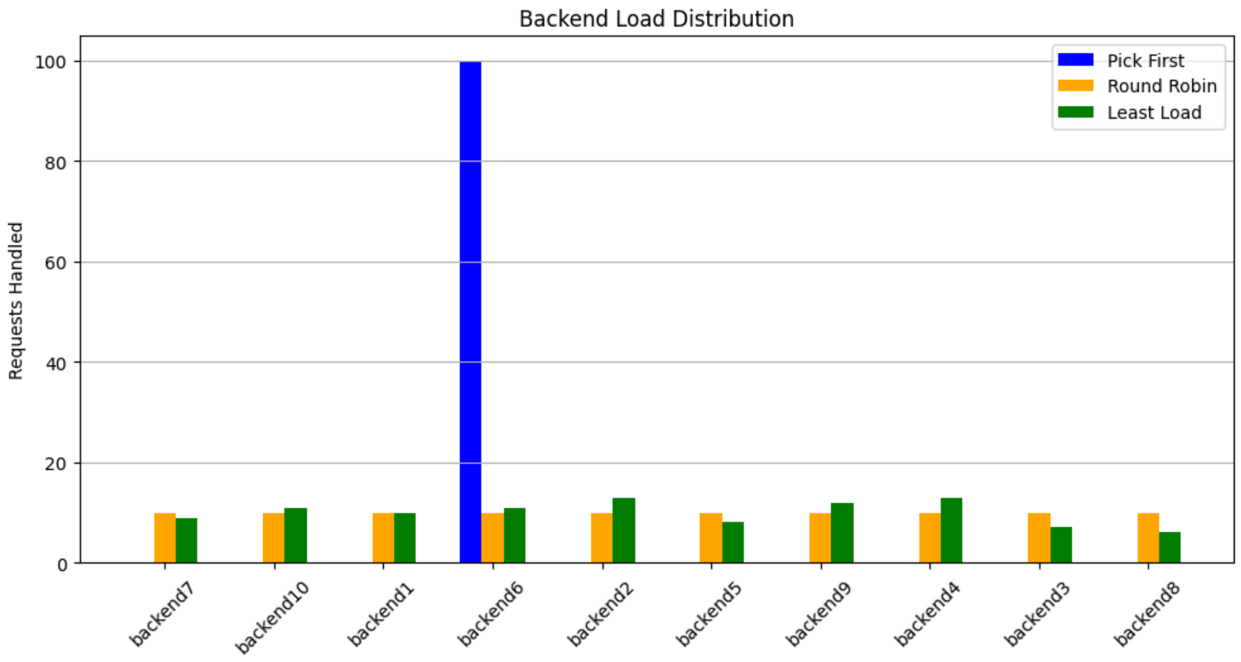
- **scale_servers.py:**
  This script automates the spawning of multiple backend server processes (configurable to 10 or 15 servers), enabling simulation of a realistic multi-server environment.

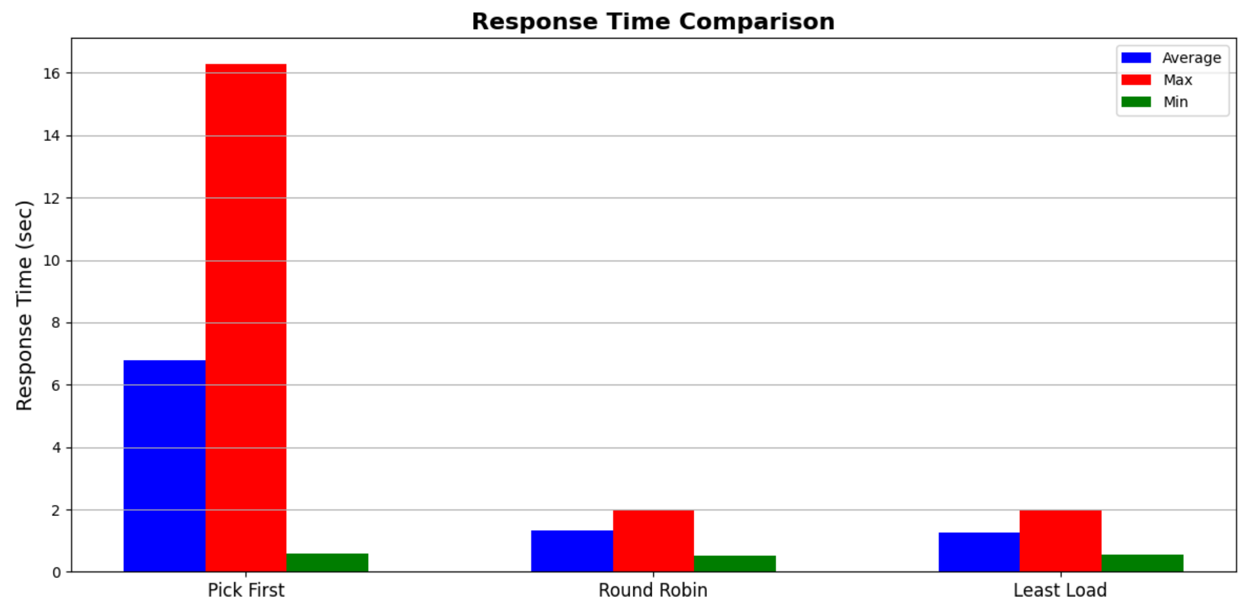
- **scale_clients.py:**
  This script simulates 100 concurrent client requests to the LB server. It measures response times, calculates throughput, and records backend usage distribution. These metrics are used to analyze and compare the performance of the different load balancing policies.

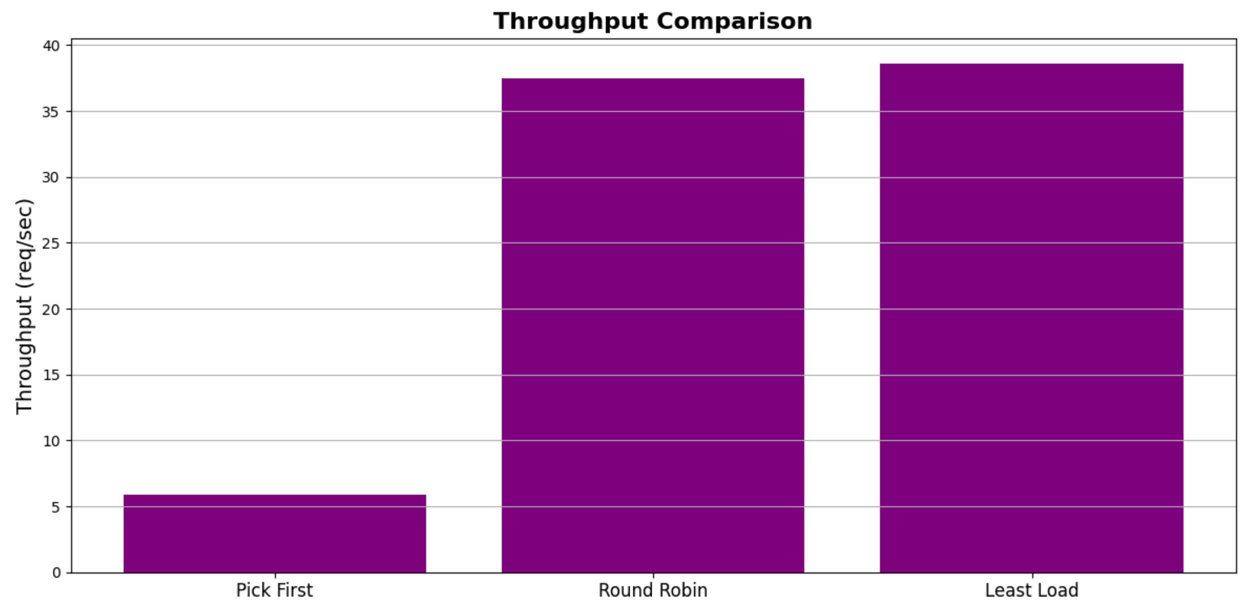
- *For a visual reference and the corresponding code on which the below 2 graphs are generated, please see the Google Colab link provided: Code Link*

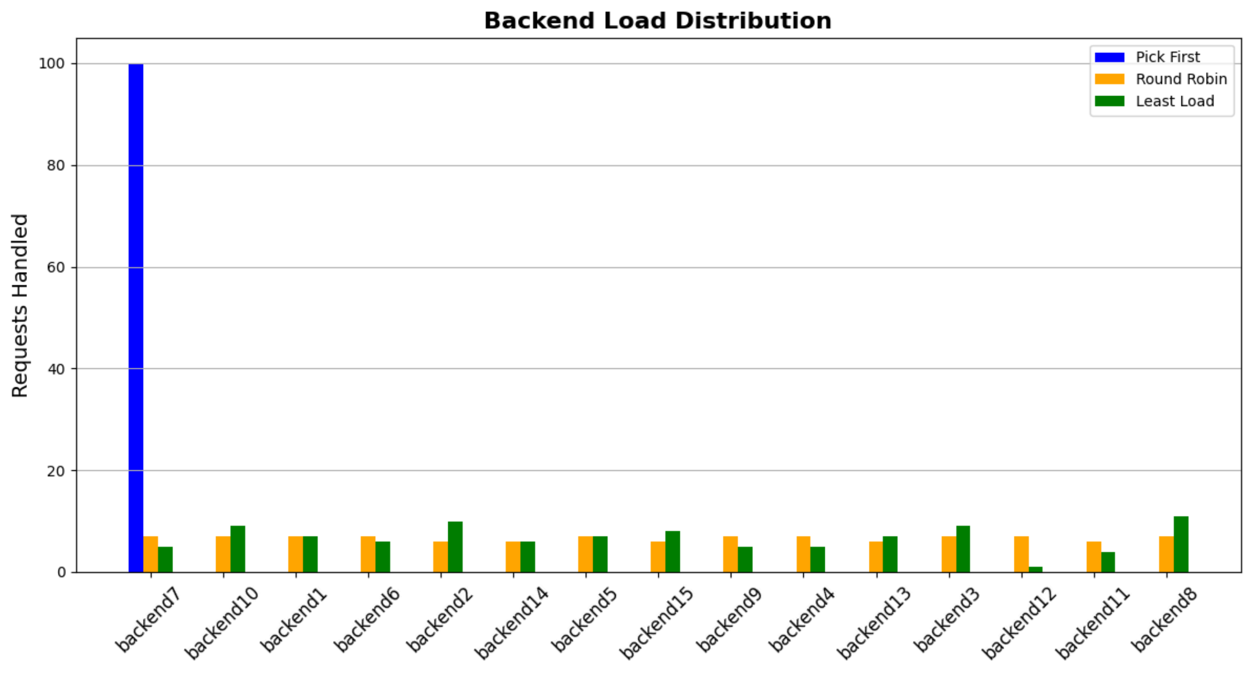

Response Time Comparison
Response Time (sec)
0
2
4
6
8
10
12
Average Response Time
Max Response Time
Min Response Time
Pick First
Round Robin
Least Load


Throughput Comparison
Throughput (requests/sec)
0
5
10
15
20
25
30
35
Pick First
Round Robin
Least Load


Backend Load Distribution
Requests Handled
0
20
40
60
80
100
Pick First
Round Robin
Least Load
backend7
backend10
backend1
backend6
backend2
backend5
backend9
backend4
backend3
backend8


Response Time Comparison
Average
Max
Min
Response Time (sec)
0
2
4
6
8
10
12
14
16
Pick First
Round Robin
Least Load


Throughput Comparison
Throughput (req/sec)
0
5
10
15
20
25
30
35
40
Pick First
Round Robin
Least Load


Backend Load Distribution
Pick First
Round Robin
Least Load
Requests Handled
0
20
40
60
80
100
backend7
backend10
backend1
backend6
backend2
backend14
backend5
backend15
backend9
backend4
backend13
backend3
backend12
backend11
backend8

6. Conclusion

This assignment successfully demonstrates a gRPC-based look-aside load balancing system with dynamic backend registration and real-time load reporting. The LB server effectively manages multiple clients and servers by:

- Maintaining an up-to-date list of backend servers.
- Responding to client queries with a valid server selection based on the chosen load balancing policy.
- Allowing the load balancing policy to be easily switched via a CLI argument for testing purposes.

Our performance analysis, using both 10 and 15 backend servers with 100 concurrent clients, indicates that:

- **Pick First** consistently routes all requests to a single server, leading to high response times and low throughput.
- **Round Robin** evenly distributes requests among servers, resulting in lower response times and higher throughput.
- The **Least Load** policy now distributes requests among multiple servers with nearly equal minimal load, achieving excellent performance (average response time of 1.254 sec and throughput of 38.61 req/sec in the 15-server test).

Overall, the LB server effectively manages load in a multi-client, multi-server environment. The scale testing scripts (**scale_servers.py** and **scale_clients.py**) facilitate comprehensive performance evaluation, and the Round Robin and Least Load policies demonstrate superior performance compared to the Pick First policy in this assignment.