Akshay Patidar (23M0792)
Ravi Patidar (23M0796)

Report

CS695 Assignment 4

Design and Implementation of Container Orchestration with Custom Load Balancing and Scaling

Background: Containerization technology, particularly Docker, has revolutionized software development and deployment. Containers offer lightweight, portable, and isolated environments for applications, making them ideal for microservices architecture. However, as applications scale, managing a large number of containers becomes increasingly complex. Moreover, ensuring optimal performance and reliability requires sophisticated load balancing and scaling mechanisms.

Load balancing ensures that incoming requests are evenly distributed among multiple instances of the same application, enhancing performance and reliability. Dynamic scaling adjusts the number of container instances based on factors like request load.

Problem Statement: The increasing adoption of containerized applications necessitates robust container orchestration systems capable of managing Docker containers efficiently. However, existing solutions often lack custom load balancing and scaling features, hindering their ability to distribute incoming requests effectively and adapt to fluctuating workloads. This project aims to develop a comprehensive container orchestration system with enhanced load balancing and scaling capabilities to address these challenges. The system's primary objective is to provide a reliable framework for managing Docker containers, ensuring optimal resource utilization and responsiveness. Key goals include designing a custom load balancing mechanism to evenly distribute requests

among multiple container replicas, implementing dynamic scaling algorithms to adjust replica counts based on demand.

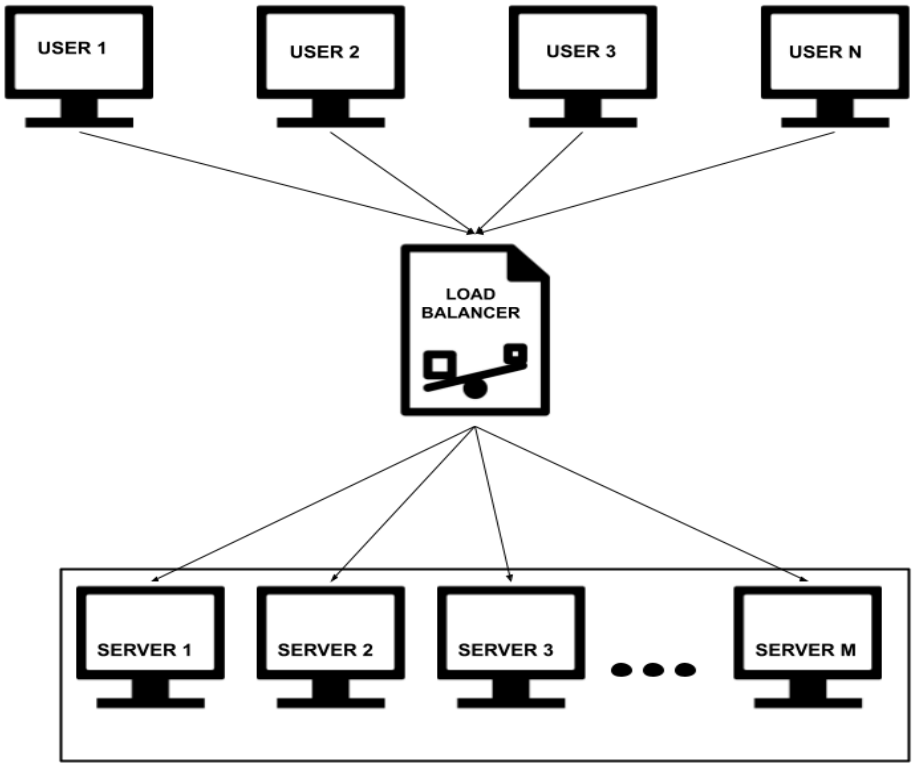


Fig: Request of N users shared to M server using Load Balancer

Design Details: Our system architecture consists of three components: a Load Balancer Server and a Web Server and a Worker to scale down containers. Let's delve into each component's design:

1. **Custom Load Balancer Server:** We have developed a dedicated load balancer server instead of relying on Docker's built-in load balancing features. This custom server provides us with greater control and flexibility over load balancing strategies and configurations. The load balancer server is responsible for receiving incoming requests and distributing them among multiple backend servers. It acts as the entry point for all client requests and performs load balancing based on predefined algorithms. The load balancer

monitors the availability of backend servers and dynamically adjusts routing to ensure optimal performance if no server is free the load balancer will scale the server. Algorithms such as Round Robin and Least Connections are implemented to distribute requests evenly across backend servers.

2. **Web Server:** The Web Server plays a critical role in our system by processing incoming requests that are forwarded by the load balancer. It hosts the core application or service responsible for fulfilling client requests. To handle varying loads, we can deploy multiple instances of the request handling server. These servers are meticulously designed to be stateless, a crucial characteristic that enables them to scale horizontally and manage requests independently. Each instance of the request handling server operates as a standalone unit, capable of processing requests without reliance on the state of other servers. This independence ensures that the system can efficiently handle increasing loads by simply adding more server instances.

3. **Worker to scale down container:** When a load balancer assigns a request to a server container, it also sets up a task scheduler to trigger a function after two minutes (adjustable) to check if the server received any requests during that period. If requests were received, the function will terminate. If no requests were received, the container is stopped and function will terminated. This scheduler runs the task for each request, and we utilize an in-memory database, Redis, to store server information (server name or domain, number of requests currently handled by that server, last request time).

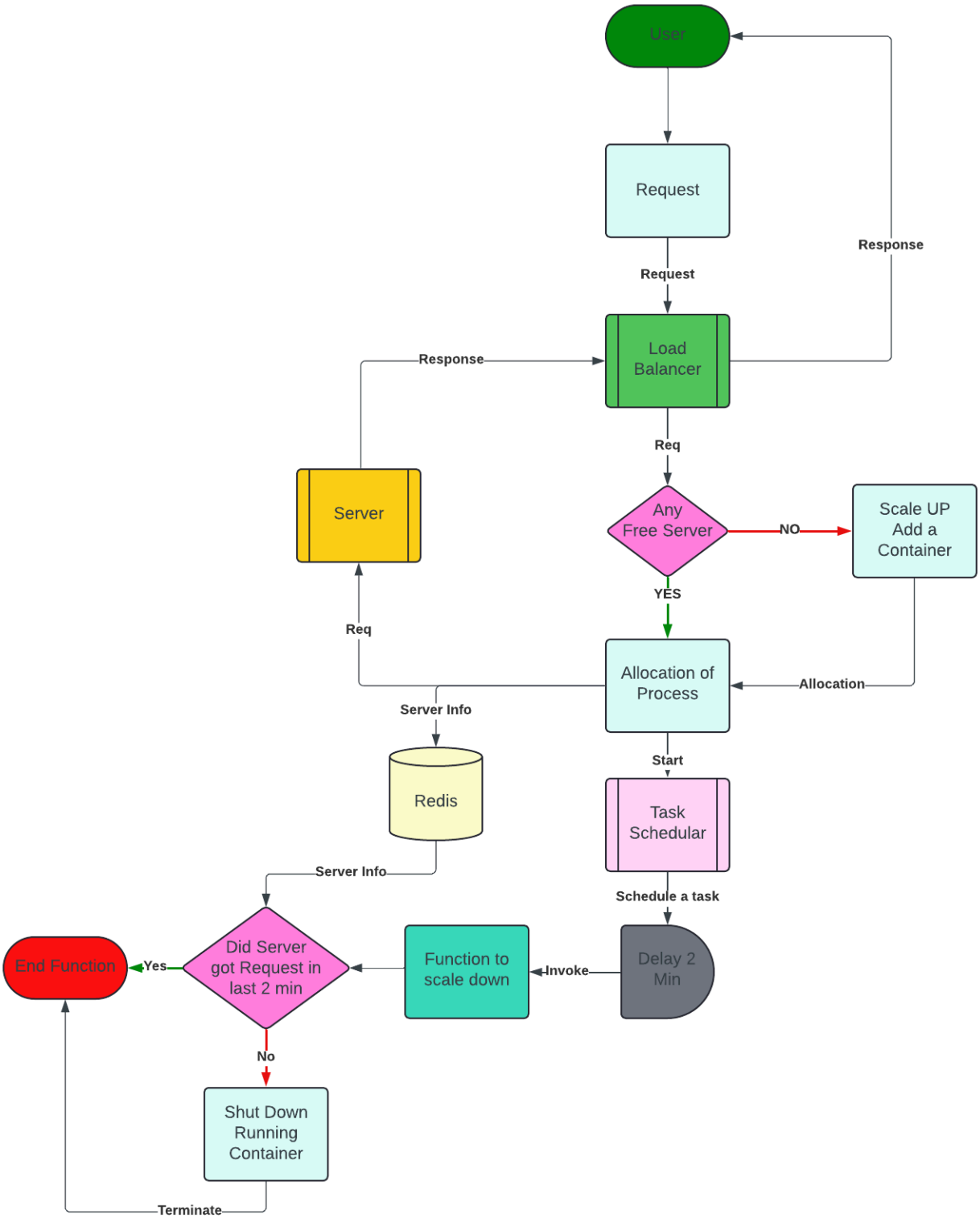


Fig: Control Flow Diagram

Implementation: Now that we have outlined the objectives and challenges, let's delve into the implementation details. We'll discuss the various components and technologies used to build our container orchestration system with custom load balancing and scaling features.

Tools & Technology Used:

- **Python:** Programming Language
- **Flask:** Flask is a lightweight Python web framework for building web applications quickly and with minimal overhead.
- **Celery:** Celery is a distributed task queue framework for Python, enabling asynchronous task execution across multiple workers.
- **Redis:** Redis is an open-source, in-memory data structure store used as a database, cache, and message broker.
- **Gunicorn:** Gunicorn is a Python WSGI HTTP server for UNIX, capable of serving web applications efficiently and with high concurrency.
- **Docker:** Docker is an open-source platform for developing, shipping, and running applications in containers.
- **Docker Compose:** Docker Compose is a tool for defining and running multi-container Docker applications, simplifying the process of managing complex application setups.
- **VS Code:** IDE
- **Pyplot:** Pyplot is a plotting library used for creating interactive visualizations in Python.
- **Bash:** Bash is a command-line interpreter for Unix-like operating systems, providing a text-based interface to interact with the system. It allows users to execute commands, automate tasks, and write scripts for various system operations.

Web Server: This Python script creates a simple web server using Flask, a lightweight web framework. When a request is made to the root URL ('/'), the server logs that the request has been received, retrieves the hostname and IP address of the server, and then waits for 5 seconds (adjustable). After the delay, it returns a response with a "Hello World" message, including the server's hostname. The server is configured to run on port 5000 and accepts requests from any IP

address ('0.0.0.0') in debug mode, which provides additional logging and error messages.

```python
app = Flask(__name__)

@app.route('/')
def hello_world():
    logging.error("Request received")
    hostname = socket.gethostname()
    IPAddr = socket.gethostbyname(hostname)
    # if (IPAddr == socket.gethostbyname('loadbalancer-server-3')):
    time.sleep(5)

    return f'{random.random()} <h1>Hello World from server {socket.gethostname()}<h1>'
```

Fig: Web Server Code

Load Balancer: This Flask application serves as a load balancer for distributing incoming requests among multiple server containers. The application is designed to dynamically scale the number of server replicas based on demand and distribute incoming requests accordingly.

Here's how it works:

- **Initialization:** The application initializes with a predefined number of server replicas (MIN_NO_OF_REPLICAS). Each server replica is represented by a dictionary containing information such as the server's ID orURL, number of requests, and the time of the last request.
- **Request Handling:** When a request is received, the load balancer uses either round robin or least connection (adjustable) to selects a server to handle the request. It checks each server's load by comparing the number of requests it has already handled (SERVER_MAX_REQUESTS). If a server is available (i.e., it hasn't reached its maximum requests), the load balancer assigns the request to that server.
- **Scaling UP:** If all server replicas are busy handling requests, the load balancer dynamically scales up by adding a new server replica. This is achieved by running a Docker command to scale the number of server containers. The load balancer then updates its list of servers with the new replica and assigns the request to it.

- **Server Information**: Each server replica is represented by a dictionary entry in the servers dictionary. This entry contains details such as the server's ID or URL, number of requests currently processed, and the timestamp of the last request.
- **Redis Integration:** The server information stored in the servers dictionary is then serialized into JSON format and stored in Redis using a key-value pair. The key used for storing this information in Redis is 'servers'.
- **Request Forwarding:** Once a server is selected, the load balancer forwards the request to the chosen server, forwarding the request method, parameters, and body as needed.
- **Request Handlling:** After forwarding the request, the load balancer waits for a response from the server. Once received, it returns the response to the client.

Worker to Scale Down: The scaling down process in the Flask application involves dynamically reducing the number of server replicas when the load decreases to prevent overprovisioning and optimize resource usage.

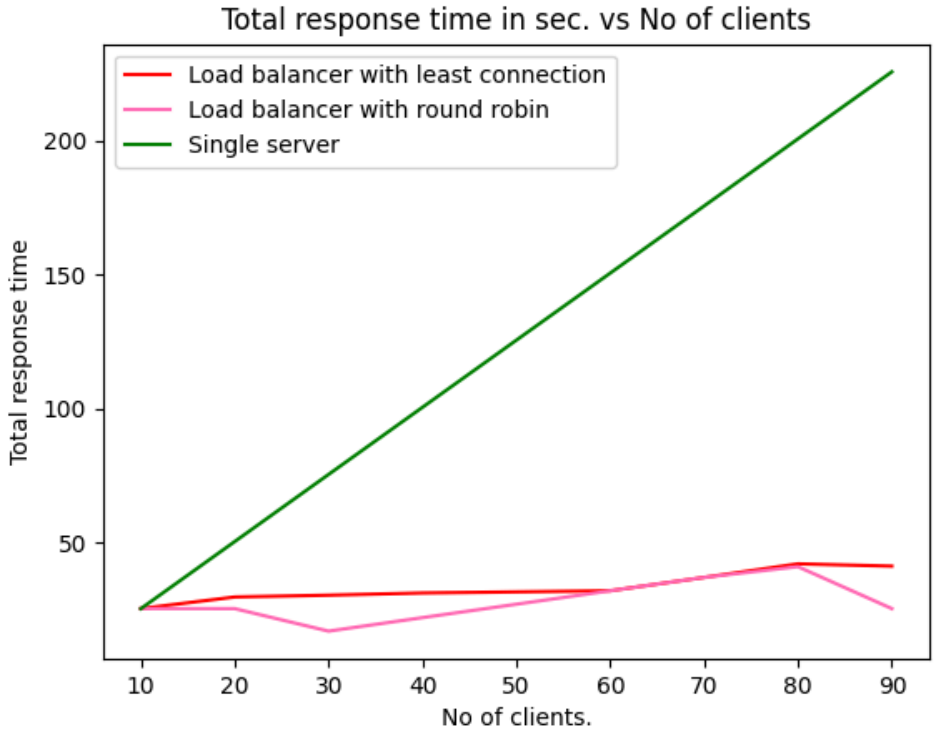
- **Initialization: T**he application starts with an initial number of server replicas (MIN_NO_OF_REPLICAS). Each server replica is monitored for activity and load.
- **Monitoring Load:** As requests are handled by the server replicas, the application keeps track of the number of requests each server has processed. When a server reaches its maximum number of requests (SERVER_MAX_REQUESTS), it's considered busy.
- **Retrieving Information:** Whenever the application needs to access server information (such as checking server status, identifying idle servers, or updating server metrics), it retrieves the serialized JSON data from Redis using the 'servers' key.
- **Determining Idle Servers:** The load balancer periodically checks the server replicas to identify idle servers. An idle server is one that hasn't received any requests for a specific period (determined by TIME_FOR_SCALE_DOWN). If a server has been idle for this duration, it's considered eligible for scaling down.

- **Scaling Down:** Once an idle server is identified, the load balancer schedules a task to scale down the number of server replicas. This task removes the idle server container from the pool of available servers. The load balancer then updates its list of servers to reflect the reduced number of replicas.
- **Updating Information:** After performing operations such as scaling down or updating server metrics, the application updates the server information in the servers dictionary and then serializes it back into JSON format. This updated data is then stored back in Redis, replacing the previous information.
- **Ensuring Redundancy:** The application maintains a minimum number of server replicas (MIN_NO_OF_REPLICAS) even after scaling down. This ensures that there's always a certain level of redundancy and availability in the system, even during low-load periods.

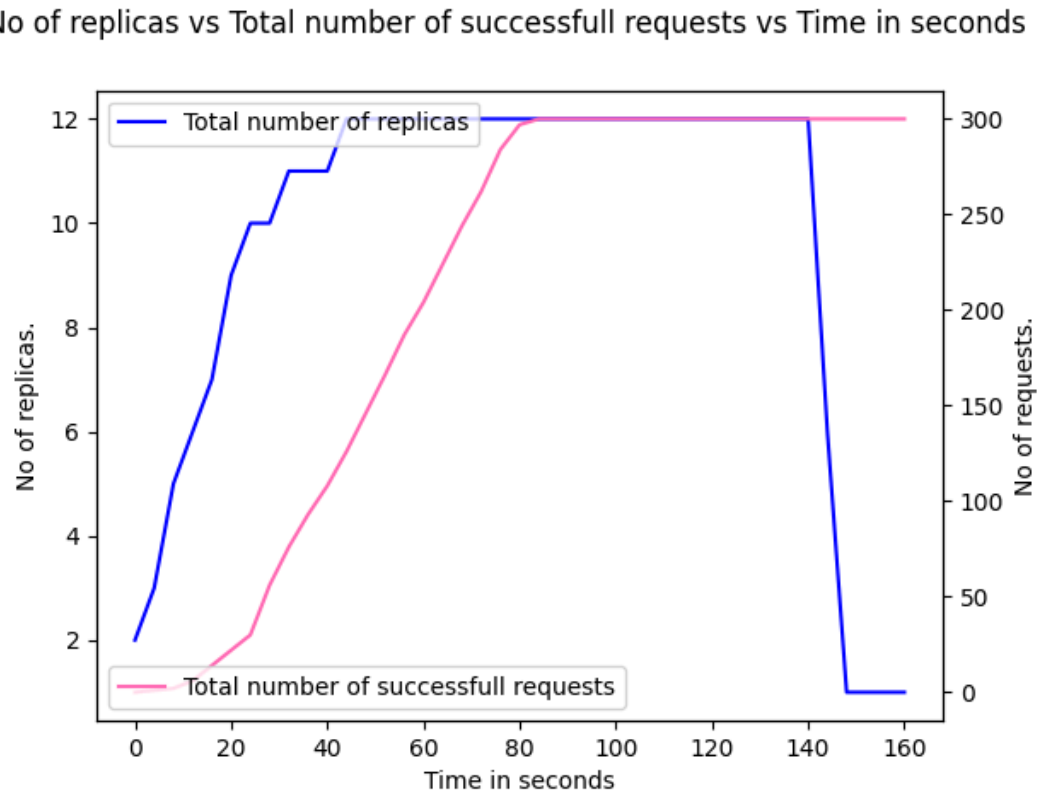
Experimental Evoluation: We conducted three experiments to evaluate the performance of different load balancing algorithms and the scalability features:

1. **Total Response Time Experiment:** We conducted an experiment to analyze the total response time for two load balancing strategies – round robin, least connection, and single server approach – while varying the number of incoming requests. The purpose was to compare the performance of each algorithm under different loads.

From this experiment, we found that using the load balancer and its scalability feature reduces the total response time for incoming requests. Because the load is distributed across multiple containers and the requests can process in parallel as compared to single server where requests are process in sequential order. The total response time includes the time taken for a request to travel from the user to the load balancer, the load balancer's scheduling time, the time taken by the server to handle the request and send the response back to the load balancer, and finally the time to deliver the response from the load balancer to the client.



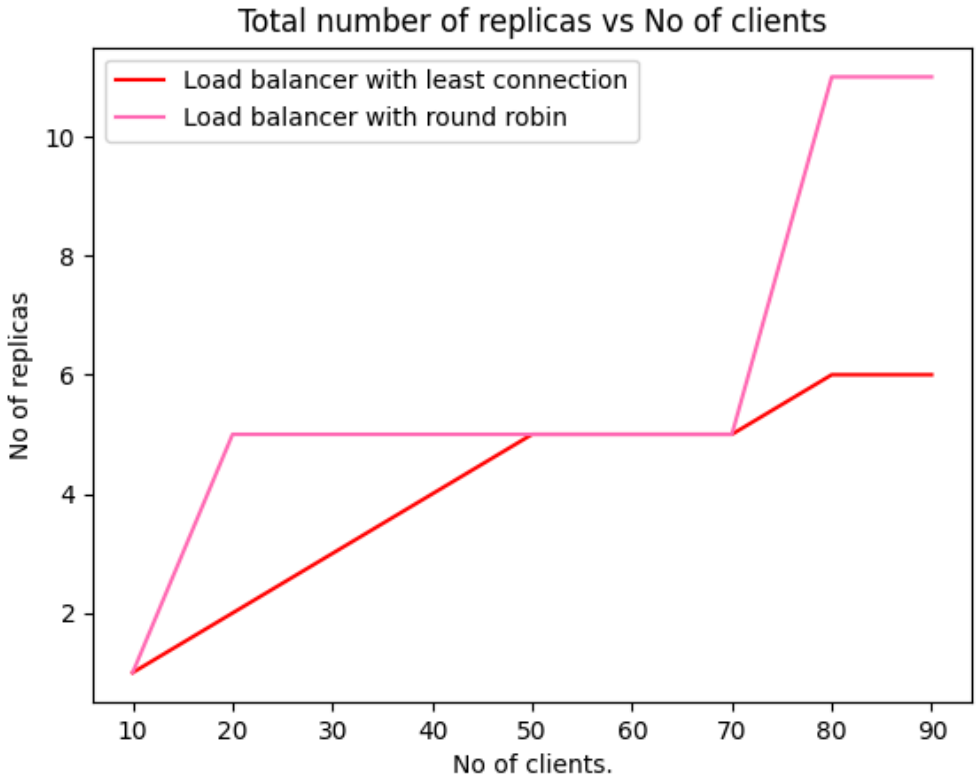
2. **Scale Up and Scale Down Feature Test:** To test the scalability features of the load balancer, we sent 1000 requests to the load balancer to schedule them across containers. We then plotted a graph showing the number of containers over time to observe the scale-up and scale-down behavior of the system. This test was conducted to ensure that the load balancer can dynamically adjust the number of containers based on the workload The below graphs shows that when request increases the number of replica increases and then after processing all the requests the number of replicas scale down.



3. **Container Scaling Experiment:** In this experiment, we explored how the total number of containers correlates with the number of incoming requests for round robin and least connection load balancing strategies. The objective was to comprehend how these strategies adapt to increasing workload demands.

We discovered that the load balancer using the least connection algorithm requires fewer replicas compared to round robin. In round robin, we check the next server's capacity to process a request; if it cannot, we scale up a container. However, in the

case of least connection, we only increase the replica if the server with the least connections is unable to handle the request.



Discussions:

- For scaling down, our initial approach involved checking for idle servers upon every incoming request. However, this method had a drawback: if a large number of requests came in, the load balancer would generate numerous replicas. When the request rate decreased, all these containers would continue running, consuming resources until new requests arrived. This led to inefficient resource utilization.

  To address this issue, we explored different solutions. One attempt involved creating a thread that scanned all servers to identify idle ones. While this theoretically worked, the process of scanning every server upon each request

consumed a considerable amount of time and resources, especially as the number of servers increased.

After further consideration, we developed a more optimized solution. We introduced a worker mechanism that scheduled tasks to check server activity after a fixed interval of time, which was adjustable. If a server received a request during this interval, it remained active. However, if no request arrived, the server was stopped and scaled down. This approach allowed us to effectively manage resources in an asynchronous manner, ensuring that idle servers were appropriately scaled down without the overhead of continuous scanning.

- To prevent the load balancer server from becoming a bottleneck for incoming requests, we employed Gunicorn to handle multiple requests simultaneously.

Thankyou

Team: Genuine Team
Ravi Patidar (23M0796)
Akshay Patidar (23M0792)