CS695
Assignment 4

Docker + Request Control

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

Index:-

Introduction

- Project aims to develop a container orchestration system with enhanced load balancing and scaling features
- Objective: Provide reliable framework for managing Docker containers with optimal resource utilization
- Goals include designing custom load balancing mechanism for even request distribution
- Implementing dynamic scaling algorithms to adjust replica counts based on demand

Tools & Technologies Used

- Python
- Flask
- Celery
- Redis
- Gunicorn
- Docker
- Docker Compose
- VS Code
- Pyplot
- Bash

Design Approach:

Our system architecture consists of three components:

- Load Balancer Server: We built a custom load balancer server for fine-tuned control over routing and balancing requests across multiple backend servers, ensuring optimal performance and scalability. It uses algorithms like Round Robin and Least Connections to evenly distribute requests.
- Web Server: Our web server processes incoming requests from the load balancer and hosts the core application. It's designed to be stateless, allowing horizontal scaling by deploying multiple independent instances capable of handling requests autonomously.
- Worker to scale down container: Upon assigning a request, the load balancer sets up a task scheduler to check server activity after two minutes. If no requests were received, the container is stopped. Redis stores server information for this purpose.

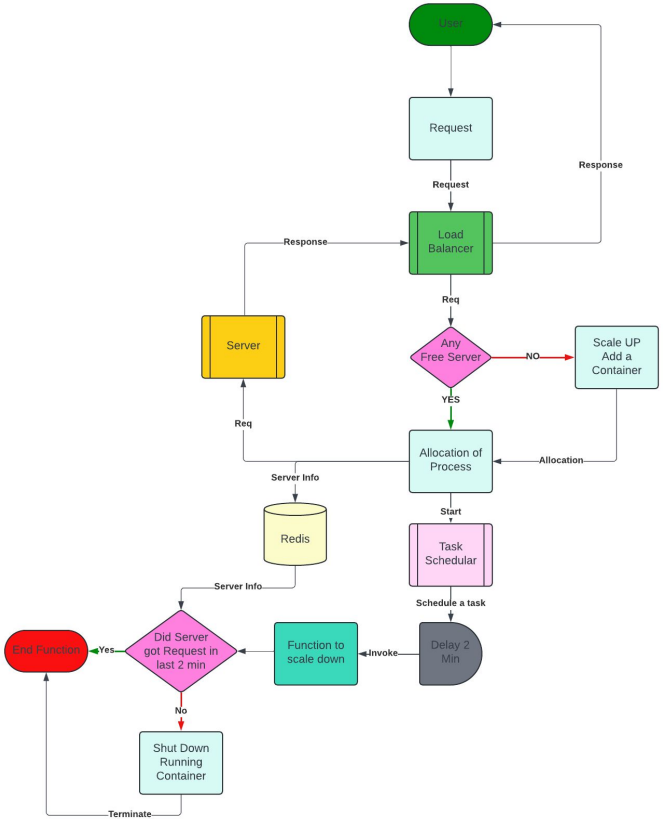


Fig: Flow Chart

Implementation

Web Server:

- Python script using Flask to create a simple web server
- When a request is made to the root URL ('/'), server logs request, retrieves server's hostname and IP address
- Server waits for 5 seconds before returning a response with "Hello World" message, including hostname
- Configured to run on port 5000, accepts requests from any IP address ('0.0.0.0') in debug mode

Implementation

Load Balancer:

- Flask application serving as a load balancer for distributing incoming requests among multiple server containers
- Initialization with a predefined number of server replicas (MIN_NO_OF_REPLICAS)
- Request handling using round robin or least connection to select a server
- Servers represented by dictionaries containing server ID or URL, number of requests, and last request time
- Scaling UP by dynamically adding a new server replica if all servers are busy
- Server information stored in Redis using key 'servers'
- After forwarding, load balancer waits for server response and returns it to the client

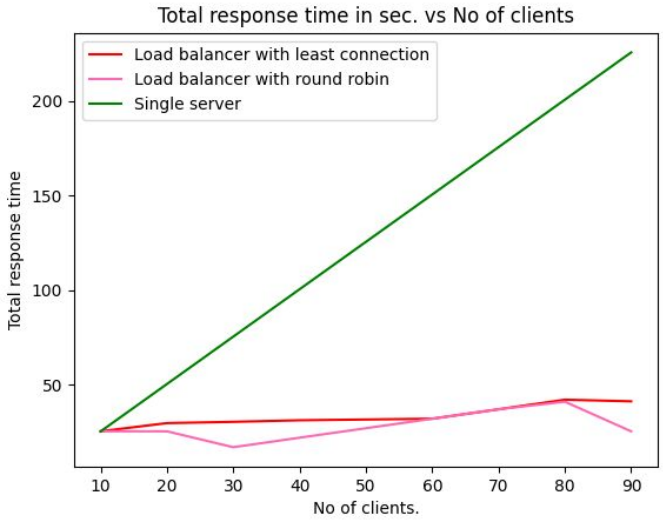
Implementation

Worker to Scale Down Servers:

- Flask application dynamically reduces server replicas to optimize resource usage
- Initialization with initial number of server replicas (MIN_NO_OF_REPLICAS)
- Monitoring load by tracking number of requests each server processes
- Retrieving server information from Redis using key 'servers'
- Identifying idle servers by checking for lack of requests over TIME_FOR_SCALE_DOWN duration
- Scaling down by removing idle server container from pool of available servers
- Updating server information in servers dictionary and Redis after scaling down
- Maintaining minimum number of server replicas (MIN_NO_OF_REPLICAS) for redundancy and availability

Performance Analysis

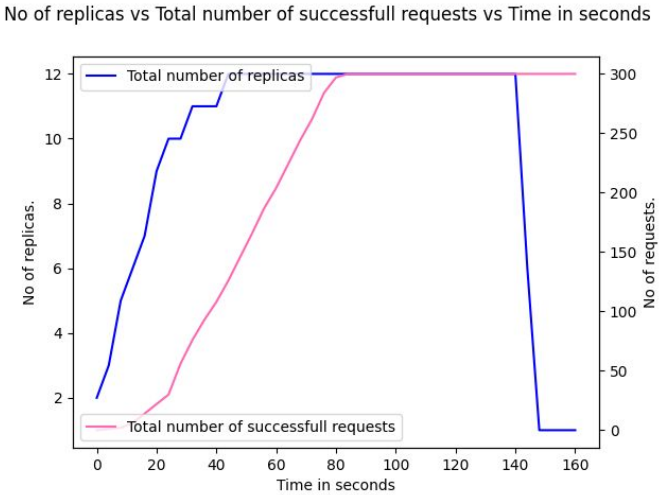
Total Response Time Experiment



- Experiment analyzed total response time for round robin, least connection, and single server approaches
- Purpose: Compare performance under varying loads
- Load balancer and scalability feature reduce total response time
- Load distributed across multiple containers allows parallel request processing
- Total response time includes travel time from user to load balancer, scheduling time, server processing time, and response delivery time

Performance Analysis

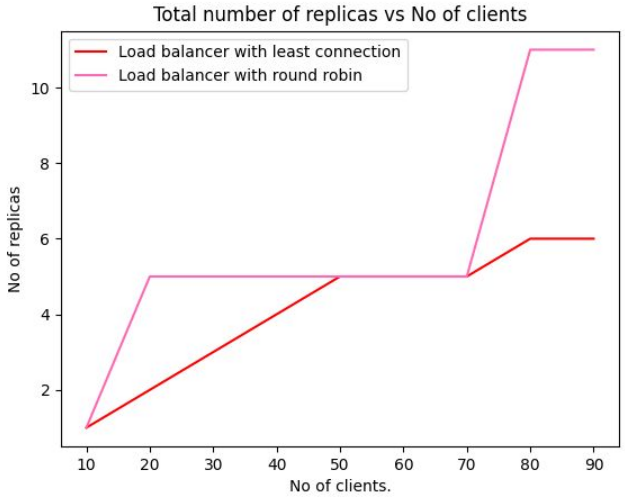
Scale Up & Scale Down Functionality Test



- Tested scalability features by sending 1000 requests to load balancer
- Plotted graph of containers over time to observe scale-up and scale-down behavior
- Load balancer dynamically adjusts container count based on workload
- Number of replicas increases with request influx, then scales down after processing all requests

Performance Analysis

Scalability Test



- Explored correlation between total containers and incoming requests for round robin and least connection strategies
- Objective: Understand how strategies adapt to increasing workload
- Least connection algorithm requires fewer replicas than round robin
- Round robin scales up container if next server cannot handle request
- Least connection increases replica only if server with least connections cannot handle request

Thank you

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