# Project Report: Orchestrating Docker Containers with Load Balancing

Ratnesh Patel(23M0784), Kumaran Karthikeyan(23M0803)

May 3, 2024

**Subject: Topics in Virtualization and Cloud Computing CS695**

**Abstract**

This project report gives the details of the implementation of the load balancer that forwards the incoming requests to the appropriate container based on the exposed port number of the containers. The containers on which the request to be forwarded are being selected based on the protocols being used , such as round robin ,least connection and probabilistic policies. The load balancer also scales up and down based on the number of active connections. This reports outlines the problem statement, approach used, key findings and interesting aspects of the project.

## 1 Problem Statement

The project aims to build the docker containers and make their separate replicas, with each replica running some services such as HTTP server. Further it involves building the load balancer that steers requests across multiple replicas of the same application running within the containers.

## 2 Approach

The first step is to build the containers and run some service within the container and make the replicas of the same application running within the containers. The next step is to build the load balancer that steers the requests across different replicas based on the policy being used and scales up and down.

## 2.1 Designing the containers and their replicas

The containers has been designed using the docker compose. Steps to build the docker containers:

1. We built an app.py(the HTTP service) which needs to be run within the container replicas. app.py is a flask server.
2. We made an image for the app.py using the Dockerfile.
3. With the help of yaml file , we designed two services, the web service and the redis service which will be storing the number being incremented for every HTTP request( this number will be shown as the output when the response from the HTTP server will be received)
4. We run the app.py with gunicorn in multithreaded way, to serve multiple request at a time.
5. Now the docker containers are good to run with the help of docker compose. Now we ran the command *docker compose -f compose.yaml up –build -d –scale web =*$< NO\_OF\_REPLICAS >$ , where compose.yaml is our yaml file for running two services mentioned in step 3. This will create multiple replicas of the web services.

By following the above steps, the docker containers will be created with multiple replicas of the web services.

## 2.2 Designing the Loadbalancer

we used flask framework of python to build the loadbalancer server. We have implemented Loadbalancer with following policies:

1. Round Robin: The requests are forwarded to the replicas one after the other.
2. Least Connection: The requests are forwarded to the replica with the least connection.
3. Probabilistic model: In this approach we give a random weight to a container and based on the cumulative probability, we will forward the requests.

Loadbalancer design steps:

1. Our Loadbalancer server is running at the host machine, at port number 8080.

2. The replicas are running within the containers, with different port numbers being exposed to the host.

3. Loadbalancer runs *docker ps*(each time before forwarding the request to the replicas) and extracts the replica URL along with their exposed port numbers.

4. Based on the policy being used, our loadbalancer extracts the replica URL(using the step 3) of the replica where the request to be forwarded.

5. Our loadbalancer is running using gunicorn, which helps us making our loadbalancer multithreaded.

Autoscaling of our loadbalancer:

1. Our loadbalancer checks the number of active connections.

2. If the number of active connections is greater than the number of web service replicas, then we scale up by one server.

3. If the number of replicas is greater than three and less than the number of web services, then we scale the server down by one.

4. Once the server replicas reaches above three, they aren't scaled down less than three, as we are maintaining some minimum number of replicas to run every time.

How clients can access the service:

1. Clients need to send the request to the url: $http://< host - ip >$: 8080

2. The loadbalancer will forward the requests to the different replicas, as described above.

3. The loadbalancer will receive the response from the replica.

4. At the end the loadbalancer will forward the response to the client.

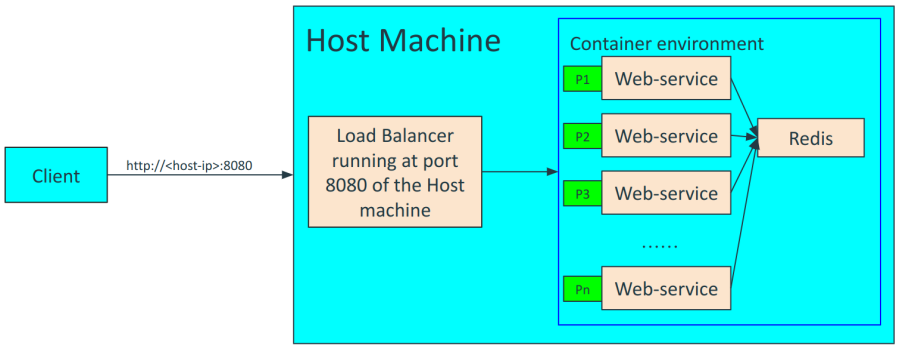


Figure 1: Design of our loadbalancer(P1,P2,P3,..Pn are the exposed ports of the web replicas)

Figure 1 shows our design of the load balancer.

# 3 Findings

We performed the performance analysis of our loadbalancer for different policies using Apache jmeter. We measured the throughput, and error rate for the loadbalancer built on three policies, i.e. round-robin, least connection based, and probabilistic model.
This is our following setup in Apache jmeter with which we stress tested it:

1. Number of threads = 500. This will be the maximum concurrent users
2. Ramp up period = 10. Time period for increasing the number of users
3. Loop count = 5. These are the number of times it will be looped through

We ran the entire test for 3 minutes 15 seconds for each thread group.

We noticed that the maximum throughput was approx 7 requests/second which is expected, as we were running 7 threads for our loadbalancer in the gunicorn config setup and the web server was sleeping for 1 second before giving the response. At a time the throughput graph flattens and slowly starts degrading due to resource contention. We got the maximum value for round robin algorithm and probabilistic algorithm(as we were using additional sorting algorithm in least-connection based policy , we argue that the implementation can be improved by a few optimizations).

We noticed the least throughput of around 2.9 requests/second for the least connectivity algorithm.

Round robin has the highest error rate of 35%. This is because round robin uses no heuristics to determine which container to forward the request to. Probabilistic had the lowest of approximately zero.

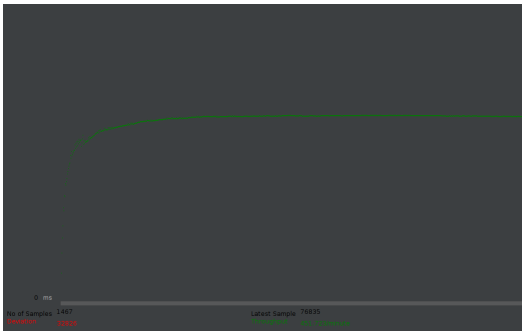


Figure 2: Throughput for round-robin policy

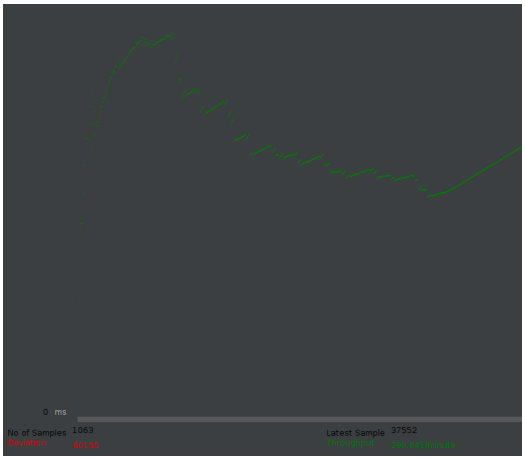


Figure 3: Throughput of least-connection based policy

Figure 4: Throughput of probabilistic based policy

We also found out that the scaling helped in improving the throughput of all the three policies.

# 4 Interesting Aspects

Several interesting aspects which emerged during the project are:

1. Load Balancing Policies: Exploring different load balancing policies like(distributing the traffic to the server with least CPU utilization) could further optimize traffic distribution.

2. Health Monitoring: Health checks could be integrated for the container instances, which would enable the loadbalancer to identify and remove unhealthy replicas from the replica pool.

3. Scaling techniques: A good scaling technique can handle the traffic well. A scaling can be done by analyzing the overall load on the individual servers, and based on that some heuristics can be applied to scale the servers up or down, based on the load on the system.

# 5 Conclusion

In this project we successfully implemented a loadbalancer which scales up and down based on the number of active connections. Our loadbalancer was implemented using three policies, round-robin, least connection based and probabilistic based. Our loadbalancer allows for efficient request routing across multiple replicas of an application, which enhances the scalability and fault tolerance. We provide a foundation for further exploration of advanced load balancing stratigies, health monitoring, and better scaling techniques.

# References

[1] Docker. "Getting Started with Docker Compose." Docker. https://docs.docker.com/compose/gettingstarted/.

[2] Apache JMeter. *Apache JMeter*. https://jmeter.apache.org/.