Load Balancing using Round Robin : Phase 1

Group 6 : Akhil Karrothu (ak8367) Sri Rachana Achyuthuni (sa9547)

ABSTRACT

For our term project, we are planning to implement Round Robin Approach for Load Balancing in Cloud Computing. We would also be comparing this Load Balancing approach with two other known Load Balancing Approaches - AWS Classic Load Balancer [2] and HAProxy Load Balancer [4]. This report presents a detailed description of our implementation of Round Robin Algorithm in Nginx [5], the deployment of a Docker container [3] with Nginx and web server containers with Apache [1].

1. INTRODUCTION

Load Balancing is a popular technique introduced commonly in Cloud Computing to ensure that the load on each server in a cluster is distributed equally. Load Balancing ensures high performance and throughput. In cloud computing, it facilitates distribution amongst servers placed anywhere in the world, thus, decreasing the response time. In case of failures, further requests are transmitted to other alive servers in the cluster maintaining reliability of the application.

There are multiple ways of implementing Load Balancing. The most common and effective way is to have a proxy server which stands as the gateway or the door for all incoming requests. This proxy server will then redirect requests to the web servers that stand behind the proxy server. The incoming server is usually unfamiliar with which web server it is interacting as the IP address that reflects is of the proxy server.

In our project, for the preliminary testing, we have deployed Apache containers as web servers as discussed in section 1. In section 2.2, we have have described the implementation of Round Robin in Nginx proxy server and then, deployed it in a Docker container (section 2.3). This acts as the gateway to the web server containers. We have hit the proxy server several times in different conditions to obtain some preliminary result, the details of which are provided in section 3.

2. IMPLEMENTATION

There are three main components in the implementation, the details of which are described in detail in this section.

2.1 Deploying Apache Container

Apache is one very well known web server which is used frequently for hosting web sites. To launch the container with Apache in it, we have created a Dockerfile which has

```
FROM ubuntu:16.04
RUN apt-get -y update && apt-get -y install apache2 && rm -rf /var/lib/apt/lists/*
CMD apachectl -DFOREGROUND
```

Figure 1: Dockerfile for deploying Apache Web Server

```
<!DOCTYPE html>
<html>
<head>
<title>Hello</title>
</head>
<body>

<h1>Hello!</h1>
<p>This is from container 1</p>

</body>
</html>
```

Figure 2: index.html file being hosted in Web Server

base image - Ubuntu16.04 Then, in the Dockerfile, we have specified the installation of Apache modules and the command to start Apache service in the foreground. The image of the Dockerfile can be found in Figure 1. Next, we have manually added an index.html file in */var/www/html* within the container which is the directory from which the files are hosted in Apache. The index.html file added can be seen in Figure 2

2.2 Round Robin in Nginx

We have implemented a Round Robin algorithm in Java using interfaces and classes. However, we discovered that integration of custom Load Balancing algorithm in Nginx needs some heavy-lifting and will also compromise on the efficiency. Thus, we chose to use the in-built modules of Nginx to implement Round Robin Algorithm. This includes setting up an upstream block of servers and then routing the traffic to this upstream in the nginx.conf file. The complete file has been provided.

2.3 Deploying Nginx container

After the modification of the nginx.conf file, we have created a Dockerfile for deploying nginx with the modified nginx.conf file by copying the nginx.conf file from the local di-

```
FROM nginx
COPY ./nginx.conf /etc/nginx/nginx.conf
```

Figure 3: Dockerfile used for hosting nginx

Figure 4: index.html file being rendered from container by Nginx

rectory to the */etc/nginx/* folder. For this, the base image used is the enterprise image of nginx which is automatically started as a foreground process, once deployed. The content of the file can be seen in Figure 3. We have created a bind between the localhost's 8080 port and the container's port 80 on which Nginx is listening.

3. PRELIMINARY TESTS AND RESULTS

In this section, we will be looking into some basic tests conducted and the results obtained.

In order to test the effectiveness of the system that we have built, we have deployed 5 Apache web server containers whose IP addresses have been added to the upstream of the Nginx container. Then, we have repeatedly hit on the running Nginx container by visiting - *http://127.0.0.1:8080*

We found that every time the site is hit, in a round robin fashion, each container's index.html page was being displayed. An example can be seen in Figure .

Then, we killed one of the containers. Without modifying the upstream configuration in Nginx, we have repeated tried to reach the server. We found that the requests do not get dropped. Once the container whose IP has been configured in the upstream is not found, the request is routed to the next available container in the round robin sequence itself. This takes a couple more seconds for the very first time. However, from the next time on wards, the round robin sequence continues without any additional time on all the available containers.

We have also generated multiple simultaneous requests from multiple browsers and found that the round robin sequence is withheld quite well. In 9/10 browser sessions, the round robin sequence took place as expected. In one session, due to the concurrency perhaps, it took a couple of seconds for the round robin to executed as expected.

When all the web server containers were killed, as expected, the load balancing Nginx server rendered a 502 Bad Gateway error as indicated in Figure 5 which means that the backend server which the proxy is trying to connect to has sent a bad response.

4. CONCLUSION & FUTURE WORK

After the preliminary tests and implementation, we have found that the system works well with Docker containers as load balancer and web servers. Since Docker is a freely available software, the cost of implementation was zero. The time taken for creating the entire system was about 3 days. However, the steps have been provided with the specific commands and the files have been provided. Thus, the reconstruction of the system took about 20 minutes. The time taken for the load balancer to render the page was quite quick - about 10 milliseconds - since the docker network is being used and the proxy container is local itself.

In the next phase of the project, we would be implementing the entire system on an EC2 server. We are expecting the cost and time taken for the page to render would increase. We would also be comparing the system that we have built with standard existing Load Balancers such as EC2 Classic Load Balancer and HAProxy.

Figure 5: 502 Bad Gateway response from Nginx

5. REFERENCES

[1] Apache. `https://www.apache.org`. Accessed: 2020-10-23.

[2] Aws classic load balancer. `https://aws.amazon.com/elasticloadbalancing/`. Accessed: 2020-09-25.

[3] Docker. `https://en.wikipedia.org/wiki/Docker_(software)`. Accessed: 2020-10-23.

[4] Haproxy. `http://www.haproxy.org`. Accessed: 2020-09-25.

[5] Nginx. `https://www.nginx.com`. Accessed: 2020-10-23.