CS60002: Distributed Systems

Assignment 1: Implementing a Customizable Load Balancer

Date: Jan 10th, 2024

Target Deadline (For sharing the git repo): January 24, 2024

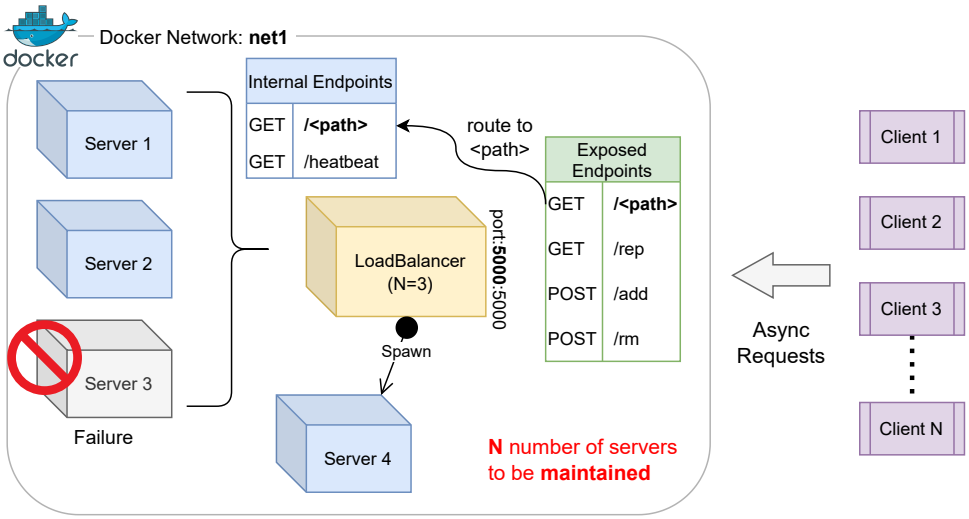


Fig. 1: System Diagram of Load Balancer

I. Overview

In this assignment, you have to implement a load balancer that routes the requests coming from several clients asynchronously among several servers so that the load is nearly evenly distributed among them. In order to scale a particular service with increasing clients, load balancers are used to manage multiple replicas of the service to improve resource utilization and throughput. In the real world, there are various use cases of such constructs in distributed caching systems, distributed database management systems, network traffic systems, etc.

To efficiently distribute the requests coming from the clients, a load balancer uses a consistent hashing data structure. The consistent hashing algorithm is described thoroughly with examples in Appendix A. You have to deploy the load balancer and servers within a Docker network as shown in Fig. 1. The load balancer is exposed to the clients through the APIs shown in the diagram (details on the APIs are given further). There should always be N servers present to handle the requests. In the event of failure, new replicas of the server will be spawned by the load balancer to handle the requests.

A. Coding Environment

- **OS:** Ubuntu 20.04 LTS or above
- **Docker:** Version 20.10.23 or above
- **Languages:** C++, Python (preferable), Java, or any other language of your choice

B. Submission Details

- Write clean and well-documented code.
- Add `README` file and mention the design choices, assumptions, testing, and performance analysis.
- Add `Makefile` to deploy and run your code.

Your Code should be version-controlled using Git, and the GitHub repository link must be shared before the deadline. Please note that the contribution of each group member is essential to learn from these assignments. Thus, we will inspect the commit logs to award marks to individual group members. An example submission from previous course session can be found at https://github.com/prasenjit52282/shardQ for your reference.

II. Task1: Server

For this assignment, we focus on the load-balancing aspects of a distributed application. Therefore, the server-side code is minimal. In this task, you need to implement a simple web server that accepts HTTP requests on `port 5000` in the below endpoints.

1) **Endpoint (/home, method=GET):** This endpoint returns a string with a unique identifier to distinguish among the replicated server containers. For instance, if a client requests this endpoint and the load balancer schedules the request to `Server: 3`, then an example return string would be `Hello from Server: 3`.
Hint: Server ID can be set as an env variable while running a container instance from the docker image of the server.

```
Response Json ={
    "message" : "Hello from Server: [ID]",
    "status" : "successful"
},
Response Code = 200
```

2) **Endpoint (/heartbeat, method=GET):** This endpoint sends heartbeat responses upon request. The load balancer further uses the heartbeat endpoint to identify failures in the set of containers maintained by it. Therefore, you could send an empty response with a valid response code.

```
Response [EMPTY],
Response Code = 200
```

Finally, write a `Dockerfile` to containerize the server as an image and make it deployable for the subsequent tasks. Note that two containers can communicate via hostnames in a docker network (Docker provides a built-in DNS service that allows containers to resolve hostnames to the correct IP addresses within the same Docker network).

III. Task2: Consistent Hashing

In this task, you need to implement a consistent hash map using an array, linked list, or any other data structure. This map data structure details are given in Appendix A. Use the following parameters and hash functions for your implementation.

- Number of Server Containers managed by the load balancer (N) = 3
- Total number of slots in the consistent hash map ($\#slots$) = 512
- Number of virtual servers for each server container (K) = $\log(512)$ = 9
- Hash function for request mapping $H(i) = i^2 + 2i + 17$
- Hash function for virtual server mapping $\Phi(i, j) = i^2 + j^2 + 2j + 25$

Hint: Two client requests can be mapped to the same slot of the hash map. However, in case there is a conflict between two server instances, apply *Linear* or *Quadratic* probing to find the next suitable slot.
Note that server containers and virtual servers are different. Server containers are the number of containers the load balancer will manage to handle requests. A virtual server is a theoretical concept that repeats the location of server containers in the consistent hash, which helps better distribute the load in case of failure. Virtual servers have nothing to do with the actual number of server containers. For example, we can have N=5 server containers in the load balancer, when in the consistent hash, we can replicate each server K=20 times and treat each entry as a virtual server. See the explanation in the Appendix.

IV. Task3: Load Balancer

In this task, you need to build a load balancer container [1] that uses the consistent hashing data structure from Task 2 to manage a set of N web server containers from Task 1. The container also provides HTTP endpoints to modify configurations or check the status of the managed web server replicas. The primary task of the load balancer container is to route the client requests to one of the server replicas so that the overall load is equally distributed across the available replicas. Apart from that, the load balancer is also responsible for maintaining N replicas even in case of failure by spawning new instances, where the new instance's hostname (also container name) is randomly generated. Only the load balancer endpoints are exposed at the host system at `port 5000`. The endpoints are described as follows:

1) **Endpoint (/rep, method=GET):** This endpoint only returns the status of the replicas managed by the load balancer. The response contains the number of replicas and their hostname in the docker internal network:n1 as mentioned in Fig. 1. An example response is shown below.

```
Response Json ={
    "message" : {
        "N" : 3,
        "replicas" : ["Server 1", "Server 2", "Server 3"]
    },
    "status" : "successful"
},
Response Code = 200
```

2) **Endpoint (/add, method=POST):** This endpoint adds new server instances in the load balancer to scale up with increasing client numbers in the system. The endpoint expects a JSON payload that mentions the number of new instances and their preferred hostnames (same as the container name in docker) in a list. An example request and response is below.

```
Payload Json= {
    "n" : 4,
    "hostnames" : ["S5", "S4", "S10", "S11"]
}
Response Json ={
    "message" : {
        "N" : 7,
        "replicas" : ["Server 1", "Server 2", "Server 3", "S5", "S4", "S10", "S11"]
    },
    "status" : "successful"
},
Response Code = 200
```

Perform simple sanity checks on the request payload and ensure that hostnames mentioned in the Payload are less than or equal to newly added instances. Note that the hostnames are preferably set. One can never set the hostnames. In that case, the hostnames (container names) are set randomly. However, sending a hostname list with greater length than newly added instances will result in an error.

```
Payload Json= {
    "n" : 2,
    "hostnames" : ["S5", "S4", "S10", "S11"]
}
Response Json ={
    "message" : "<Error> Length of hostname list is more than newly added instances",
    "status" : "failure"
},
Response Code = 400
```

3) **Endpoint (/rm, method=DELETE):** This endpoint removes server instances in the load balancer to scale down with decreasing client or system maintenance. The endpoint expects a JSON payload that mentions the number of instances to be removed and their preferred hostnames (same as container name in docker) in a list. An example request and response is below.

```
Payload Json= {
    "n" : 3,
    "hostnames" : ["S5", "S4"]
}
Response Json ={
    "message" : {
        "N" : 4,
        "replicas" : ["Server 1", "Server 3", "S10", "S11"] /*See "Server 2" is choosen randomly to be deleted along with mentioned "S5" & "S4"*/
    },
    "status" : "successful"
},
Response Code = 200
```

Perform simple sanity checks on the request payload and ensure that hostnames mentioned in the Payload are less than or equal to the number of instances to be removed. Note that the hostnames are preferably mentioned with the delete request. One can never set the hostnames. In that case, the hostnames (also container names) are randomly selected for removal. However, sending a hostname list with a greater length than the number of removable instances will result in an error.

```
Payload Json= {
    "n" : 2,
    "hostnames" : ["S5", "S4", "S10", "S11"]
}
Response Json ={
    "message" : "<Error> Length of hostname list is more than removable instances",
    "status" : "failure"
},
Response Code = 400
```

4) **Endpoint (/<path>, method=GET):** Request in this endpoint gets routed to a server replica as scheduled by the consistent hashing algorithm of the load balancer. According to Task 1, the web server has a ”/home” endpoint. Thus, a GET request to ”/home” in the load balancer would give a valid response that routes from a server replica. Requesting an endpoint that is not registered with the web server will cause an error, as shown below.

```
Response Json ={
    "message" : "<Error> '/other' endpoint does not exist in server replicas",
    "status" : "failure"
},
Response Code = 400
```

Finally, write a `Dockerfile` to containerize the load balancer. Also, write a `docker-compose` [2] file as well as a `Makefile` [3] to easily deploy the whole stack in a Ubuntu environment. Use the parameters defined in Task 2 as default values for the load balancer container. Later, you can modify the number of server containers requesting the above endpoints.

V. TASK4: ANALYSIS

In this task, you test and analyze the performance of your load balancer implementation in different scenarios. You need to show how it distributes the load among the server containers and how promptly it recovers from server container failure. The `README` file must contain the observations and explanations of the following experiments.

A-1 Launch 10000 async requests on $N = 3$ server containers and report the request count handled by each server instance in a bar chart. Explain your observations in the graph and your view on the performance.

A-2 Next, increment N from 2 to 6 and launch 10000 requests on each such increment. Report the average load of the servers at each run in a line chart. Explain your observations in the graph and your view on the scalability of the load balancer implementation.

A-3 Test all endpoints of the load balancer and show that in case of server failure, the load balancer spawns a new instance quickly to handle the load.

A-4 Finally, modify the hash functions $H(i)$, $\Phi(i, j)$ and report the observations from (A-1) and (A-2).

APPENDIX

A. Installation of Docker

You need to run the following code snippets to install the docker daemon and the docker-compose in a standard Ubuntu (above 20.04) system. Please ensure your system has an internet connection before attempting to do any installation.

```
# Docker: latest [version 20.10.23, build 7155243]
~$ sudo apt-get update
~$ sudo apt-get install ca-certificates curl gnupg lsb-release
~$ sudo mkdir -p /etc/apt/keyrings
~$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/
    keyrings/docker.gpg
~$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https
    ://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources
    .list.d/docker.list > /dev/null
~$ sudo apt-get update
~$ sudo apt-get install docker-ce docker-ce-cli containerd.io


# Docker-compose standalone [version v2.15.1]
~S sudo curl -SL https://github.com/docker/compose/releases/download/v2.15.1/docker-compose-
    linux-x86_64 -o /usr/local/bin/docker-compose
~$ sudo chmod +x /usr/local/bin/docker-compose
~$ sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
```

B. Consistent Hashing

Consistent hashing [4, 5] has a unique hashing structure that is circular instead of linear to avoid many shifts of data in the event of the addition of resources to the system. Load Balancer uses consistent hashing to distribute client requests evenly among the server instances (i.e., balancing the system load). Moreover, consistent hashing technique is also used in distributed caching systems for better utilization of resources.

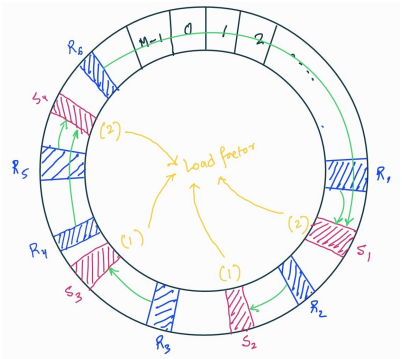
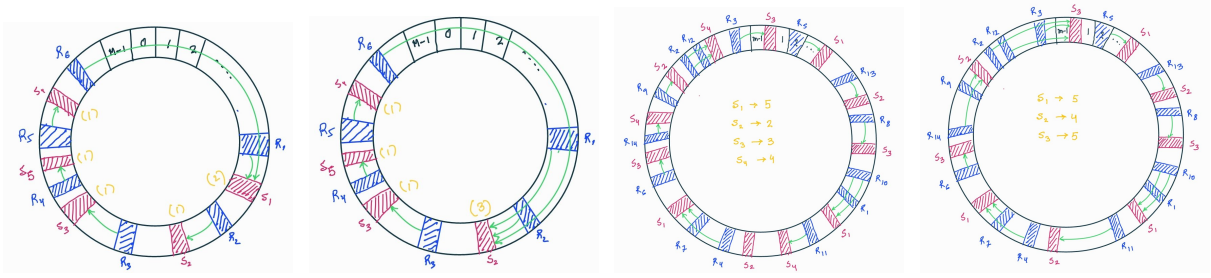


Fig. 2: Data Structure to implement a Consistent Hash Map

1) Consistent Hash Map: For any client request, a unique ID R_{id} is associated with it. For simplicity, you can assume that the request IDs are 6-digit random numbers (e.g., 132574). Requests will be mapped to the indexes or slots of a circular data structure as shown in Fig. 2, having M slots in total. The slot mapping is done with the help of a hash function H. Therefore, slot number $slot_n \leftarrow H(R_{id})\%M$. The servers with unique ID S_{id} are also placed in the slots of the data structure using another hash function Φ. Thus, server slots are computed as $slot_n \leftarrow \Phi(S_{id})\%M$. The requests are assigned to the server in clockwise order, which is present in the nearest slot.

2) Adding New Server Instance: Let's consider adding a 5^{th} server to the system to handle more client requests. The new server will be placed in the consistent map structure using the hash function Φ that is used for server mapping. Only the assignment of R_4 is changed from S_4 to S_5 as shown in Fig. 3a. Here, the change in each of the server's loads will be much less than the linear structure of the map. This property gives better performance in load balancing for concurrent client requests.

3) Failure of Server Instance: Now, if server S_1 suddenly gets down due to a power outage or network outage, the requests that were scheduled on S_1 will be shifted to server S_2 as per the clock-wise allocation logic. In this scenario, the majority load of the system comes to S_2 server instead of evenly distributing across four available servers as shown in Fig. 3b. This problem is further solved using the virtual server concept.



(a) Server addition in system (b) Server failure in system (c) Virtual Servers (d) Failure with Virtual Server

Fig. 3: Consistent Hash-map in different scenarios

4) Virtual Server: Instead of placing a single instance of a physical server in the circular hash-map structure, more than one replica of a physical server is mapped into the structure. Thus, we can use two variables i and j to represent a virtual server $S_{(i,j)}$ and use them as inputs to the hash function Φ to map the servers with minimum conflicts. Here, i represents the server ID, and j represents the virtual server replica ID of server S_i. Thus, a virtual server slot number $slot_n \leftarrow \Phi(i,j)\%M$.

We have shown four servers, each with three virtual instances as shown in Fig. 3c. Now, suppose server S_4 gets down. Then, all the instances of S_4 will be removed from the structure, and the requests that were assigned to instances of S_4 will be served by the server, which is present in the next slot in a clockwise direction. We can see that the load is readjusted evenly after failure from Fig. 3d. Statistically, $K = \log(M)$ virtual servers work best to distribute the load across the physical server instances equally.

C. Implementation Hints

1) Privileged containers: A privileged container is a container that has all the capabilities of the host machine, which lifts all the limitations regular containers have. This means that privileged containers can do almost every action that can be performed directly on the host. Privileged containers can spawn other containers, manage the host network card, remove containers, etc. The below docker-compose file shows some tags you can use while defining a service.

```
#docker-compose.yml file
#-----------------------
version: "3.9"
services:
  servicename:
    build: ./path # Path to the Dockerfile
    image: imagenameA
    container_name: containernameA
    ports:
      - "hostport:internalport" # Exposing port to host
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      # This share host's the docker daemon with the container. Now, the container can spawn
    other containers just like host
    privileged: true # This should be true to share the docker daemon
    networks:
      net1: # Need to define net1 in the compose file
        aliases:
          - hostnameA # Usually same as containername
    environment:
      VAR1: VALUE1 # Environment variables accessed with os.environ['VAR1'].
```

2) Spawning containerB from containerA: Assuming that containerA has host privileges, we can spawn or remove any container from containerA. Let's say we need to launch a container instance, having docker image ImageB in the system. The below code will run a containerB from that image and attach it to the net1 internal network with hostname containerB. Similarly, we can also remove any container from the privileged container. To install `sudo` and `docker daemon` within a container your can take help from https://github.com/prasenjit52282/shardQ/blob/main/manager/Dockerfile file (Line 15-33).

```
#Spawning containerB from imageB
#-------------------------------
res=os.popen(f'sudo docker run --name containerB --network net1 --network-alias containerB -e
    VAR1=v1 -e VAR2=v2  -d ImageB:latest').read()
if len(res)==0:
    print("Unable to start containerB")
else:
    print("successfully started containerB")

#Removing containerB
#-------------------
os.system('sudo docker stop containerB && sudo docker rm containerB')
```

D. Grading Scheme

- TASK1: Server - 20 %
- TASK2: Consistent Hashing - 30 %
- TASK3: Load Balancer - 30 %
- TASK4: Analysis - 20 %

References

[1] Docker, "What is a container?." https://docs.docker.com/guides/get-started/, 2024.

[2] Docker, "Use docker compose." https://docs.docker.com/get-started/08_using_compose/, 2024.

[3] makefiletutorial, "Learn makefiles with the tastiest examples." https://makefiletutorial.com/, 2024.

[4] T. Roughgarden and G. Valiant, "Cs168: The modern algorithmic toolbox lecture 1: Introduction and consistent hashing." https://web.stanford.edu/class/cs168/l/l1.pdf, 2022.

[5] J. Li, Y. Nie, and S. Zhou, "A dynamic load balancing algorithm based on consistent hash," in *2018 2nd IEEE Advanced Information Management,Communicates,Electronic and Automation Control Conference (IMCEC)*, pp. 2387–2391, 2018.