

## Part 1: Implementing a Customizable Load Balancer

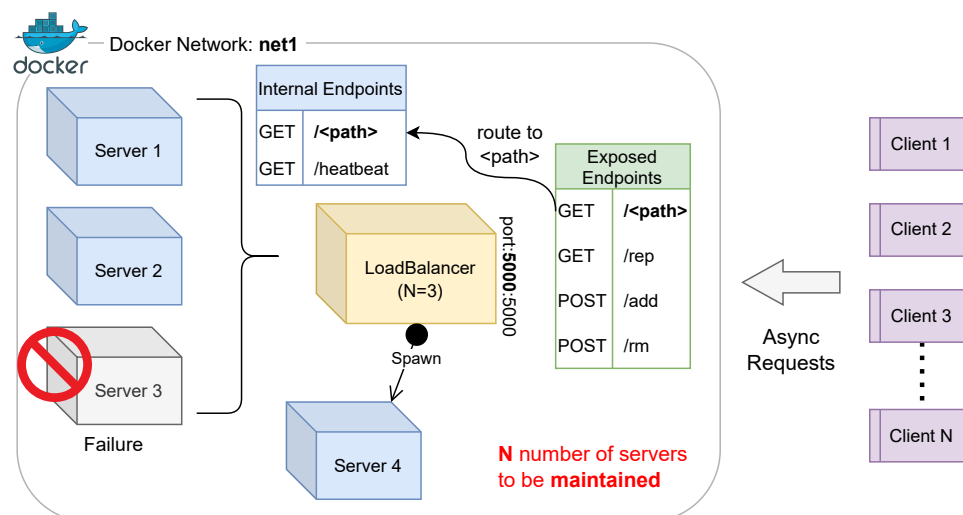


Fig. 1: System Diagram of Load Balancer

### 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.

```
1 Response Json = {
2     "message" : "Hello from Server: [ID]",
3     "status" : "successful"
4 },
5 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.

```
1 Response [EMPTY],
2 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:nl as mentioned in Fig. 1. An example response is shown below.

```
1 Response Json = {
2   "message" : {
3     "N" : 3,
4     "replicas" : ["Server 1", "Server 2", "Server 3"]
5   },
6   "status" : "successful"
7 },
8 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.

```
1 Payload Json= {
2   "n" : 4,
3   "hostnames" : ["S5", "S4", "S10", "S11"]
4 }
5 Response Json = {
6   "message" : {
7     "N" : 7,
8     "replicas" : ["Server 1", "Server 2", "Server 3", "S5", "S4", "S10", "S11"]
9   },
10  "status" : "successful"
11 },
12 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.

```

1 Payload Json= {
2   "n" : 2,
3   "hostnames" : ["S5", "S4", "S10", "S11"]
4 }
5 Response Json = {
6   "message" : "<Error> Length of hostname list is more than newly added instances",
7   "status" : "failure"
8 },
9 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.

```

1 Payload Json= {
2   "n" : 3,
3   "hostnames" : ["S5", "S4"]
4 }
5 Response Json = {
6   "message" : {
7     "N" : 4,
8     "replicas" : ["Server 1", "Server 3", "S10", "S11"] /*See "Server 2" is choosen
    randomly to be deleted along with mentioned "S5" & "S4"*/
9   },
10  "status" : "successful"
11 },
12 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.

```

1 Payload Json= {
2   "n" : 2,
3   "hostnames" : ["S5", "S4", "S10", "S11"]
4 }
5 Response Json = {
6   "message" : "<Error> Length of hostname list is more than removable instances",
7   "status" : "failure"
8 },
9 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.

```

1 Response Json = {
2   "message" : "<Error> '/other' endpoint does not exist in server replicas",
3   "status" : "failure"
4 },
5 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.

```

1 # Docker: latest [version 20.10.23, build 7155243]
2 ~$ sudo apt-get update
3 ~$ sudo apt-get install ca-certificates curl gnupg lsb-release
4 ~$ sudo mkdir -p /etc/apt/keyrings
5 ~$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/
   keyrings/docker.gpg
6 ~$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https
   ://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources
   .list.d/docker.list > /dev/null
7 ~$ sudo apt-get update
8 ~$ sudo apt-get install docker-ce docker-ce-cli containerd.io
9
10
11 # Docker-compose standalone [version v2.15.1]
12 ~$ curl -SL https://github.com/docker/compose/releases/download/v2.15.1/docker-compose-
   linux-x86_64 -o /usr/local/bin/docker-compose
13 ~$ sudo chmod +x /usr/local/bin/docker-compose
14 ~$ 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.

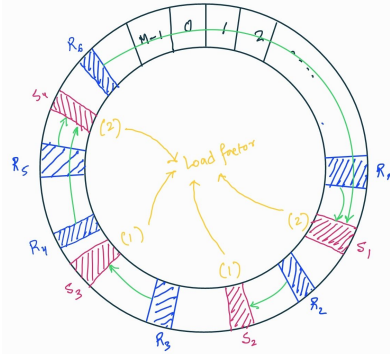


Fig. 2: Data Structure to implement a Consistent Hash Map

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  $\Phi$ . 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<sup>th</sup> server to the system to handle more client requests. The new server will be placed in the consistent map structure using the hash function  $\Phi$  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.

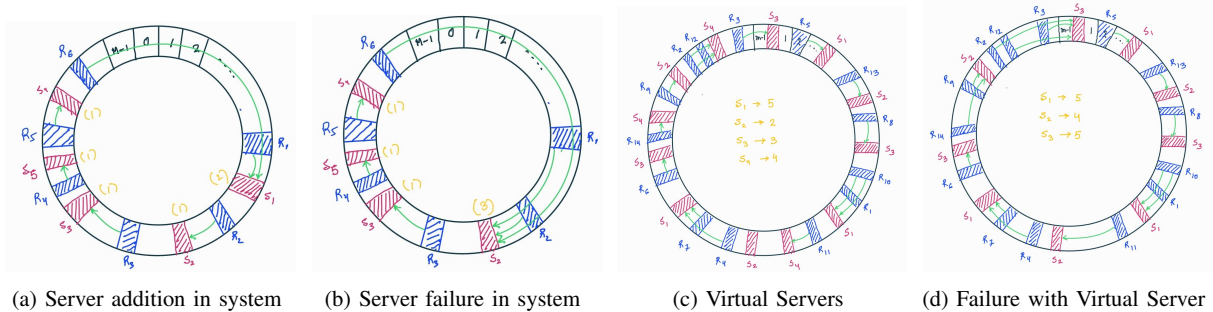


Fig. 3: Consistent Hash-map in different scenarios

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  $\Phi$  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.

```

1 #docker-compose.yml file
2 #-----
3 version: "3.9"
4 services:
5   servicename:
6     build: ./path # Path to the Dockerfile
7     image: imagenameA
8     container_name: containernameA
9     ports:
10      - "hostport:internalport" # Exposing port to host
11     volumes:
12      - /var/run/docker.sock:/var/run/docker.sock
13      # This share host's the docker daemon with the container. Now, the container can spawn
14      # other containers just like host
15     privileged: true # This should be true to share the docker daemon
16     networks:
17       net1: # Need to define net1 in the compose file
18       aliases:
19         - hostnameA # Usually same as containername
20     environment:
21       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 you can take help from <https://github.com/prasenjit52282/shardQ/blob/main/manager/Dockerfile> file (Line 15-33).

```

1 #Spawning containerB from imageB
2 #-----
3 res=os.popen(f'sudo docker run --name containerB --network net1 --network-alias containerB -e
4   VAR1=v1 -e VAR2=v2 -d ImageB:latest').read()
5 if len(res)==0:
6   print("Unable to start containerB")
7 else:
8   print("successfully started containerB")
9 #Removing containerB
10 #-----
11 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/](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/l11.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, Communication, Electronic and Automation Control Conference (IMCEC)*, pp. 2387–2391, 2018.

## Part 2: Implementing a Scalable Database with Sharding

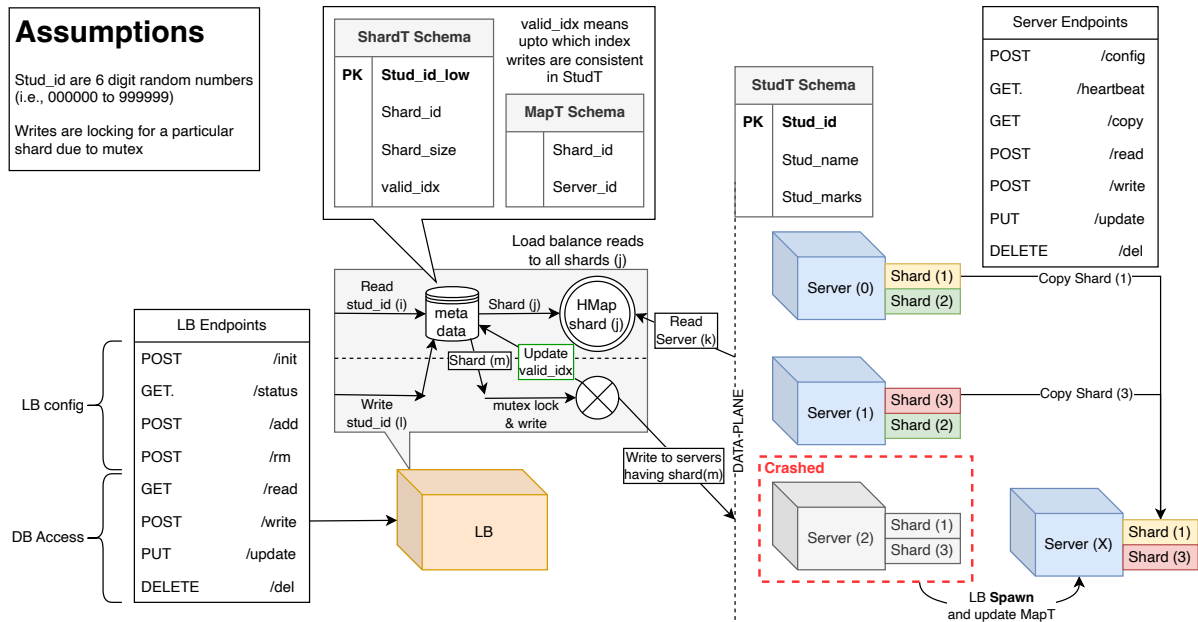


Fig. 1: System Diagram of a Sharded Database

### I. OVERVIEW

In this assignment, you have to implement a sharded database that stores only one table *StudT* in multiple shards distributed across several server containers. This is an incremental project so that you can reuse the codebase from the first assignment. A system diagram of the sharded database is shown in Fig. 1. Here, shards are subparts of the database that only manage a limited number of entries (i.e., *shard\_size* as shown in the diagram). Shards can be replicated across multiple server containers to enable parallel read capabilities. For this assignment, we assume that write requests are blocking for a particular shard. Thus, if two write requests are scheduled simultaneously on shard (i), one of them will wait for the other to complete. However, Parallel writing to different shards, for instance, shard (i) and shard (j), is possible. The system's current design provides scaling in two ways: (i) Read speed with more shard replicas and (ii) Database size with more shards and servers.

#### 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

The server containers handles shards of the *StudT* (**Stud\_id: Number, Stud\_name: String, Stud\_marks: Number**) table. Each server container can manage a different set of shard replicas of the distributed database, as shown in Fig. 1. The endpoints to handle requests to a specific server are described as follows:

- 1) **Endpoint (/config, method=POST):** This endpoint initializes the shard tables in the server database after the container is loaded. The shards are configured according to the request payload. An example request-response pair is shown below.

```
1 Payload Json= {
2   "schema": { "columns": ["Stud_id", "Stud_name", "Stud_marks"],
3               "dtypes": ["Number", "String", "String"] }
4   "shards": ["sh1", "sh2"]
5 }
```

```

6 Response Json = {
7     "message" : "Server0:sh1, Server0:sh2 configured",
8     "status" : "success"
9 },
10 Response Code = 200

```

- 2) **Endpoint (/heartbeat, method=GET):** This endpoint sends heartbeat responses upon request. The heartbeat endpoint is further used to identify failures in the set of server containers maintained in the distributed database. Therefore, you could send an empty response with a valid response code.

```

1 Response [EMPTY],
2 Response Code = 200

```

- 3) **Endpoint (/copy, method=GET):** This endpoint returns all data entries corresponding to one shard table in the server container. Copy endpoint is further used to populate shard tables from replicas in case a particular server container fails, as shown in Fig. 1. An example request-response pair is shown below.

```

1 Payload Json= {
2     "shards":["sh1","sh2"]
3 }
4 Response Json = {
5     "sh1" : [{ "Stud_id":1232, "Stud_name":ABC, "Stud_marks":25},
6               { "Stud_id":1234, "Stud_name":DEF, "Stud_marks":28},
7               ...],
8     "sh2" : [{ "Stud_id":2255, "Stud_name":GHI, "Stud_marks":27},
9               { "Stud_id":2535, "Stud_name":JKL, "Stud_marks":23},
10              ...],
11     "status" : "success"
12 },
13 Response Code = 200

```

- 4) **Endpoint (/read, method=POST):** This endpoint reads data entries from a shard in a particular server container. The endpoint expects a range of *Stud\_ids* (low, high) and *Shard\_id* that has to be read from the server container.

```

1 Payload Json= {
2     "shard":"sh2",
3     "Stud_id": { "low":2235, "high":2555}
4 }
5 Response Json = {
6     "data" : [{ "Stud_id":2535, "Stud_name":JKL, "Stud_marks":23},
7                { "Stud_id":2536, "Stud_name":MNO, "Stud_marks":22},
8                .....
9                { "Stud_id":2554, "Stud_name":XYZ, "Stud_marks":25},
10               { "Stud_id":2255, "Stud_name":GHI, "Stud_marks":27}],
11     "status" : "success"
12 },
13 Response Code = 200

```

- 5) **Endpoint (/write, method=POST):** This endpoint writes data entries in a shard in a particular server container. The endpoint expects multiple entries to be written in the server container along with *Shard\_id* and the current index for the shard. An example request-response pair is shown below.

```

1 Payload Json= {
2     "shard":"sh2",
3     "curr_idx": 507
4     "data": [{ "Stud_id":2255, "Stud_name":GHI, "Stud_marks":27}, ...] /* 5 entries */
5 }
6 Response Json = {
7     "message": "Data entries added",
8     "current_idx": 512, /* 5 entries added */
9     "status" : "success"
10 },
11 Response Code = 200

```

- 6) **Endpoint (/update, method=PUT):** This endpoint updates a particular data entry in a shard in a particular server container. The endpoint expects only one entry to be updated in the server container along with *Shard\_id*. An example request-response pair is shown below.

```

1 Payload Json= {
2     "shard":"sh2",
3     "Stud_id":2255,
4     "data": { "Stud_id":2255, "Stud_name":GHI, "Stud_marks":28} /* see marks got updated */
5 }
6 Response Json = {
7     "message": "Data entry for Stud_id:2255 updated",
8     "status" : "success"
9 },
10 Response Code = 200

```

- 7) **Endpoint (/del, method=DELETE):** This endpoint deletes a particular data entry (based on *Stud\_id*) in a shard in a particular server container. The endpoint expects only one entry to be deleted in the server container along with *Shard\_id*. An example request-response pair is shown below.



```

1 Payload Json= {
2     "shard":"sh1",
3     "Stud_id":2255
4 }
5 Response Json={
6     "message": "Data entry with Stud_id:2255 removed",
7     "status" : "success"
8 },
9 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: IMPROVE THE LOAD BALANCER

You have implemented the load balancer with consistent hashing in the previous assignment. All the previous assumptions and implementation details still hold. However, you must modify and integrate a few features to the load balancer to make it work for assignment 2. In the system diagram shown in Fig. 1 you can observe that the load balancer manages *Stud\_id* → *Shard\_id* → *Server\_id* mapping with two data tables in the **metadata**. The table schemas are as follows:

- 1) **ShardT** (*Stud\_id\_low*: Number, *Shard\_id*: Number, *Shard\_size*:Number, *valid\_idx*:Number)
- 2) **MapT** (*Shard\_id*: Number, *Server\_id*: Number)

#### A. Assumptions

To connect the distributed database assignment with the earlier load balancer assignment, you should realize that you need to maintain consistent hashmaps for each of the shards. The consistent hashmaps can be identified with the *Shard\_id*, and the hashmap entries will be populated by the replicas of the shard.

- There are only 4 (sh1, sh2, sh3, sh4) shards in the database.
- Each shard has 3 replicas across the servers.
- There are 6 servers having shards in configuration:

```

1 {
2     "N":6
3     "schema":{"columns":["Stud_id","Stud_name","Stud_marks"],
4         "dtypes":["Number","String","String"]}
5     "shards":[{"Stud_id_low":0, "Shard_id": "sh1", "Shard_size":4096},
6         {"Stud_id_low":4096, "Shard_id": "sh2", "Shard_size":4096},
7         {"Stud_id_low":8192, "Shard_id": "sh3", "Shard_size":4096},
8         {"Stud_id_low":12288, "Shard_id": "sh4", "Shard_size":4096}]
9     "servers":{"Server0":["sh1","sh2"],
10         "Server1":["sh3","sh4"],
11         "Server3":["sh1","sh3"],
12         "Server4":["sh4","sh2"],
13         "Server5":["sh1","sh4"],
14         "Server6":["sh3","sh2"]}
15 }

```

- Thus consistent hashmap for shard sh1 will contain entries for servers {Server0:sh1, Server3:sh1, Server5:sh1}. Similarly each shard would have corresponding hashmap. Other constants for hashmap are as follows:
  - 1) Total number of slots in the consistent hash map ( $\#slots$ ) = 512
  - 2) Number of virtual servers for each server container ( $K$ ) =  $\log(512) = 9$
  - 3) Function for request mapping  $H(i)$ , & virtual server mapping  $\Phi(i, j)$  is what you found works best for your load balancer implementation.

#### B. Endpoints

The read requests to a particular shard (i) will be load balanced with the consistent hash among all shard (i) replicas across all the server containers. Therefore, parallel read requests can be serviced. In case of write requests, the load balancer ensures consistent write to all the appropriate shard replicas with a mutex lock. Hence, writes are blocking for a particular shard and can only be parallelized across different shard instances (e.g., not replicas). The endpoints of the load balancer are as follows.

- 1) **Endpoint (/init, method=POST):** This endpoint initializes the distributed database across different shards and replicas in the server containers. As administrators, we should provide the configuration of shards and their placements in the set of server containers (place randomly if not specified). An example request-response pair is shown below.

```

1 Payload Json= {
2     "N":3
3     "schema":{"columns":["Stud_id","Stud_name","Stud_marks"],
4         "dtypes":["Number","String","String"]}
5     "shards":[{"Stud_id_low":0, "Shard_id": "sh1", "Shard_size":4096},
6         {"Stud_id_low":4096, "Shard_id": "sh2", "Shard_size":4096},
7         {"Stud_id_low":8192, "Shard_id": "sh3", "Shard_size":4096},]
8     "servers":{"Server0":["sh1","sh2"],
9         "Server1":["sh2","sh3"],

```



```

10         "Server2":["sh1","sh3"]}
11     }
12     Response Json={
13         "message" : "Configured Database",
14         "status" : "success"
15     },
16     Response Code = 200

```

- 2) **Endpoint (/status, method=GET):** This endpoint sends the database configurations upon request. The configuration is set via the `/init` endpoint; thus, a valid response according to the above `/init` example is shown below.

```

1 Response Json= {
2     "N":3
3     "schema":{"columns":["Stud_id","Stud_name","Stud_marks"],
4               "dtypes":["Number","String","String"]}
5     "shards":[{"Stud_id_low":0, "Shard_id": "sh1", "Shard_size":4096},
6               {"Stud_id_low":4096, "Shard_id": "sh2", "Shard_size":4096},
7               {"Stud_id_low":8192, "Shard_id": "sh3", "Shard_size":4096},]
8     "servers":{"Server0":["sh1","sh2"],
9               "Server1":["sh2","sh3"],
10              "Server2":["sh1","sh3"]}
11 }
12 Response Code = 200

```

- 3) **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, their server names, and the shard placements. An example request and response is below.

```

1 Payload Json= {
2     "n" : 2,
3     new_shards:[{"Stud_id_low":12288, "Shard_id": "sh5", "Shard_size":4096}]
4     "servers" : {"Server4":["sh3","sh5"], /* new shards must be defined */
5                 "Server[5]":["sh2","sh5"]}
6 }
7 Response Json ={
8     "N":5,
9     "message" : "Add Server:4 and Server:58127", /* server id is randomly set in case of
10    Server[5] */
11     "status" : "successful"
12 },
13 Response Code = 200

```

Perform simple sanity checks on the request payload and ensure that servers mentioned in the Payload are less than or equal to newly added instances unless through an error. Server IDs are preferably set randomly to make the consistent hashing algorithm work efficiently. Therefore, you can also define a template for servernames (shown above) where ids are randomly set by the load balancer.

```

1 Payload Json= {
2     "n" : 3, /* wrong input n > len(servers) */
3     new_shards:[{"Stud_id_low":12288, "Shard_id": "sh5", "Shard_size":4096}]
4     "servers" : {"Server4":["sh3","sh5"],
5                 "Server[5]":["sh2","sh5"]}
6 }
7 Response Json ={
8     "message" : "<Error> Number of new servers (n) is greater than newly added instances",
9     "status" : "failure"
10 },
11 Response Code = 400

```

- 4) **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 server names in a list. An example request and response is below.

```

1 Payload Json= {
2     "n" : 2,
3     "servers" : ["Server4"]
4 }
5 Response Json ={
6     "message" : {
7         "N" : 3,
8         "servers" : ["Server1", "Server4"] /*See "Server1" is choosen randomly to be
9         deleted along with mentioned "Server4" */
10     },
11     "status" : "successful"
12 },
13 Response Code = 200

```

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

```

1 Payload Json= {
2     "n" : 2,          /* wrong input n < len(servers) */
3     "servers" : ["Server1", "Server4", "Server2"]
4 }
5 Response Json = {
6     "message" : "<Error> Length of server list is more than removable instances",
7     "status" : "failure"
8 },
9 Response Code = 400

```

- 5) **Endpoint (/read, method=POST):** Based on the consistent hashing algorithm, this endpoint reads data entries from the shard replicas across all server containers. The endpoint expects a range of *Stud\_ids* (low, high) to be read from the distributed database. The load balancer derives all the required shards to be read from the provided *Stud\_id* range and accumulates all read results before sending them to the client.

*Hint:* Due to our indexing (*Stud\_id\_low*, *Stud\_id\_low+Shard\_size*) rule for each shard, we can easily find the shard entry from the *Stud\_id*. (Calculation: *Stud\_id\_low*+ *Stud\_id % Shard\_size* entry will have the corresponding *Shard\_id*).

```

1 Payload Json= {
2     "Stud_id": {"low":1000, "high":8889}
3 }
4 Response Json = {
5     "shards_queried": ["sh1", "sh2", "sh3"]
6     "data" : [{"Stud_id":1000, "Stud_name":PQR, "Stud_marks":23},
7               {"Stud_id":1001, "Stud_name":STV, "Stud_marks":22},
8               ....,
9               {"Stud_id":8888, "Stud_name":ZQN, "Stud_marks":65},
10              {"Stud_id":8889, "Stud_name":BYHS, "Stud_marks":76}],
11     "status" : "success"
12 },
13 Response Code = 200

```

- 6) **Endpoint (/write, method=POST):** This endpoint writes data entries in the distributed database. The endpoint expects multiple entries that are to be written in the server containers. The load balancer schedules each write to its corresponding shard replicas and ensures data consistency using mutex locks for a particular shard and its replicas. The general write work flow will be like: (1) **Get Shard\_ids from Stud\_ids and group writes for each shard** → **For each shard Do:** (2a) **Take mutex lock for Shard\_id (m)** → (2b) **Get all servers (set S) having replicas of Shard\_id (m)** → (2c) **Write entries in all servers (set S) in Shard\_id (m)** → (2d) **Update the valid\_idx of Shard\_id (m) in the metadata if writes are successful** → (2e) **Release the mutex lock for Shard\_id (m)**. An example request-response pair is shown.

```

1 Payload Json= {
2     "data": [{"Stud_id":2255, "Stud_name":GHI, "Stud_marks":27},
3             {"Stud_id":3524, "Stud_name":JKBFSFS, "Stud_marks":56}
4             ...
5             {"Stud_id":1005, "Stud_name":YUBAAD, "Stud_marks":100}] /* 100 entries */
6 }
7 Response Json = {
8     "message": "100 Data entries added",
9     "status" : "success"
10 },
11 Response Code = 200

```

- 7) **Endpoint (/update, method=PUT):** This endpoint updates a particular data entry (based on *Stud\_id*) in the distributed database. The load balancer retrieves all the shard replicas and their corresponding server instances where the entry has to be updated. The endpoint expects only one entry to be updated. An example request-response pair is shown. *Hint:* Note that the updates are blocking for a particular shard and its replicas to maintain data consistency.

```

1 Payload Json= {
2     "Stud_id": "2255",
3     "data": {"Stud_id":2255, "Stud_name":GHI, "Stud_marks":30} /* see marks got updated */
4 }
5 Response Json = {
6     "message": "Data entry for Stud_id: 2255 updated",
7     "status" : "success"
8 },
9 Response Code = 200

```

- 8) **Endpoint (/del, method=DELETE):** This endpoint deletes a particular data entry (based on *Stud\_id*) in the distributed database. The load balancer retrieves all the shard replicas and their corresponding server instances where the entry has to be deleted. The endpoint expects only one entry that is to be deleted. An example request-response pair is shown. *Hint:* Note that the deletes are blocking for a particular shard and its replicas to maintain data consistency.

```

1 Payload Json= {
2     "Stud_id":2255
3 }
4 Response Json = {
5     "message": "Data entry with Stud_id:2255 removed from all replicas",
6     "status" : "success"
7 },
8 Response Code = 200

```

#### IV. TASK3: ANALYSIS

To analyze the performance of the developed distributed database. You need to perform four subtasks. As mentioned earlier, the design provides scaling in two ways: (i) Read speed with more shard replicas and (ii) Size with more shards and servers.

- A-1 Report the read and write speed for 10000 writes and 10000 reads in the default configuration given in task 2.
- A-2 Increase the number of shard replicas (to 7) from the configuration (init endpoint). Report the write speed down for 10000 writes and read speed up for 10000 reads.
- A-3 Increase the number of Servers (to 10) by adding new servers and increase the number of shards (shard to 6, shard replicas to 8). Define the (init endpoint) configurations according to your choice. Report the write speed up for 10000 writes and read speed up for 10000 reads.
- A-4 Finally, check all the endpoints and ensure their correctness. Manually drop a server container and show that the load balancer spawns a new container and copies the shard entries from other replicas.

#### APPENDIX

We encourage you to use MYSQL:8.0 to implement the data tables in this assignment. However, you can use any database you want. To run the database and a flask app in the same container, you can follow the method below.

```

1 #Dockerfile
2 #-----
3 FROM mysql:8.0-debian
4
5 COPY deploy.sh /always-initdb.d/          #here the flask app deploy script is copied
6 COPY . /bkr
7 WORKDIR /bkr
8
9 RUN apt-get update
10 RUN apt-get install -y python3
11 RUN apt-get install -y python3-pip
12
13 RUN pip install --upgrade pip
14 RUN pip install -r requirements.txt
15
16 ENV MYSQL_ROOT_PASSWORD="abc"    #host='localhost', user='root',password='abc'
17
18 EXPOSE 5000

```

Whatever is copied in the '/always-initdb.d/' folder will be run after the database is initialized. Therefore, the Flask app can connect to the local database with root credentials (mysql.connector.connect(host='localhost',user='root',password='abc').

```

1 #deploy.sh file
2 #-----
3 #!/bin/bash
4 python3 flaskApp.py &

```

For reference, you can observe how the Dockerfile and deploy.sh file is coded for <https://github.com/prasenj52282/shardQ/tree/main/broker>. For other databases (i.e., MongoDB, Cassandra, etc.), similar always-init scripts can be run when the database initializes in every startup. Feel free to choose any database of your choice.

#### A. Grading Scheme

- TASK1: Server - 40 %
- TASK2: Improve the load balancer - 30 %
- TASK3: Analysis - 20 %
- Documentation & Code Optimizations - 10 %

#### 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/](https://docs.docker.com/get-started/08_using_compose/), 2024.
- [3] makefiletutorial, "Learn makefiles with the tastiest examples." <https://makefiletutorial.com/>, 2024.
- [4] 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.
- [5] T. Roughgarden and G. Valiant, "Cs168: The modern algorithmic toolbox lecture 1: Introduction and consistent hashing." <https://web.stanford.edu/class/cs168/l11.pdf>, 2022.
- [6] DockerHub, "Mysql:8.0 docker image source in dockerhub." [https://hub.docker.com/\\_/mysql](https://hub.docker.com/_/mysql), 2024.