Strathmore
UNIVERSITY

LOAD BALANCER REPORT

REPORT PREPARED BY:

Mbugua Nathan Ng'ethe- 115104

Mutende Arabella Fanisheba - 139133

Nalugala Venessa Chebukwa - 145646

Caleb Asira Etemesi - 146565

INSTRUCTOR

Mr. H. Talo, Distributed Systems Unit

REPOSITORY

https://github.com/nerdistry/Customizable-Load-Balancer

\

Server

The server endpoints are defined in (file main.rs), the following endpoints are present

- /heartbeat: Heartbeat functionality, return server status indicating which servers were on and their status to a heartbeat request
- /home : A simple request handled by the load balancer indicating which server would handle the request without necessarily forwarding the request to the server
- /add : Add a new server to the load balancers
- /rm: Remove a server from load balancers
- /metrics: Prometheus metrics
- /rep: Returns replica status

All other unknown endpoints are passed to the backend server

Consistent hashing

The hash function used for the initial server was

```
fn hash_virtual_server(container_id: usize, vs_index: usize, total_slots: usize) -> usize {
  ((37 * container_id * vs_index) ^ (container_id | vs_index)) % total_slots
}
```

This is not a cryptographically secure hash, but it's a good enough mixer to distribute servers in an arbitrary way

This hash was modified from the original suggested hash (shown below) since the original hash was predictable and distributed servers in a linear way (server 1, server 2, server 3) which could easily lead to overloading of a server if the pattern was predictable.

```
fn hash_virtual_server(container_id: usize, vs_index: usize) -> usize {
  (container_id + vs_index + 2 * vs_index + 25) % TOTAL_SLOTS
}
```

The main meat of the consistent hashing file (src/consistent_hashing.rs) is in initialize(&self) function.

We start by creating a random number from a seeded generator (to ensure reproducibility) and assign that as the server ids. The assignment specified that the range of id's is assumed to be 6 digits so we generate id's in this range

```
// Use a seeded wyrand rng
let mut ranger = nanorand::rand::WyRand::new_seed(32422312);
// Create server containers
for i in 0..self.num_containers {
  self.servers.push(Arc::new(ServerContainer {
          // gen
    id: ranger.generate_range(100_000..999_999),
    name: format!("Server-{}", i),
```

```
    }));
}
```

We then go and distribute slots to servers, to prevent division by zero, we check that the server isn't empty.

In case of hash conflicts, we increment by 1 until we find an empty slot, this is guaranteed to terminate since we loop only 512 times to insert 512 items.

```
if !self.servers.is_empty() {
    // Create virtual servers for each server container
    let virtual_servers_per_container = TOTAL_SLOTS  / self.servers.len();

    let mut hash_map = self.hash_map.write().unwrap();
    // virtual_servers_per_container is the number of times a single server will
    // be duplicated in our slot map, aka number of slots a mapping of virtual server
    // to physical server exist for each physical server
    for i in 0..virtual_servers_per_container {
        for container in &self.servers {
            //  hash the server to get the slot
            let mut slot = hash_virtual_server(container.id, i, TOTAL_SLOTS);

            // Apply linear probing if there's a conflict
            while hash_map.contains_key(&slot) {
                slot = (slot + 1) % TOTAL_SLOTS;
            }

            hash_map.insert(
                slot,
                Arc::new(VirtualServer {
                    server_container: container.clone(),
                    slot,
                }),
            );
        }
    }
}
```

When a request comes in, we can either match directly, (server->slot) but in case we miss, we do a simple linear probe (get_server_container)

```
// Direct match
if let Some(vs) = hash_map.get(&slot) {
    return Some(vs.server_container.clone());
}

// Linear probing to find the nearest slot with a virtual server
for i in 1..TOTAL_SLOTS {
    let check_slot = (slot + i) % self.slots;
    if let Some(vs) = hash_map.get(&check_slot) {
        return Some(vs.server_container.clone());
    }
}
```

Example

An example of using consistent hashing with three servers and 512 slots is given below.

The first 21 slots are filled in the following order

slot=0 name=Server-2 slot=1 name=Server-2 slot=2 name=Server-2
slot=3 name=Server-0 slot=4 name=Server-1 slot=5 name=Server-2
slot=6 name=Server-0 slot=7 name=Server-1 slot=8 name=Server-0
slot=9 name=Server-1 slot=10 name=Server-0 slot=11 name=Server-2
slot=12 name=Server-2 slot=13 name=Server-1 slot=14 name=Server-1
slot=15 name=Server-2 slot=16 name=Server-0 slot=17 name=Server-1
slot=18 name=Server-1 slot=19 name=Server-1 slot=20 name=Server-1

The distribution is the following

Server 0: 5
Server 1: 9
Server 2: 7

While focusing on a small distribution (only 20 slots), one may think that distribution is biased to one server, but the consistent hash will skew the other slots to have more servers so that in the end of all slots (512), the servers per slot will be 512/self.servers.length() (in our case, 170 or 171)

Analysis

The load balancer was sent 100 requests synchronously and prometheus was used to do statistics.

Above is a cumulative graph of request distribution as time continues, (cumulative graph), the load balancer had 4 servers for which it distributed requests to the server

The command to add the servers was

`curl "http://localhost:5001/add" -X POST -H "Content-Type: application/json" -d '{"N":1,"hostnames":["hate","love","big","small"]}'`

The script use for requests was the following

```
#!/bin/bash
j=1
for i in {1..100}
do
  start="$j day ago";
  end="${i} day ago";
  j=$((j+1));
  start_date="date -d '${start}' '+%Y-%m-%d'";
  end_date="date -d '${end}' '+%Y-%m-%d'";
```

```
  ss=$(eval "$start_date");
  ee=$(eval "$end_date");
  #echo  $ee,$ss,$start_date,$end_date

  curl "http://localhost:5001/neo?start_date=$ss&&end_date=$ee" >> /dev/null
  sleep 5
done
```

It calls the load balancer 100 times varying the parameters in order to simulate variable latency,

From the above requests we collected the following metrics

1. Request latency
2. Request distribution
3. Total Requests
4. Requests per single time

1. Total Requests

A graph of total requests received by load balancer with increase in time

The requests start at zero and steadily increase with time up until 100
Total runtime was 13minutes 4 seconds 805 milliseconds

2. Request distribution

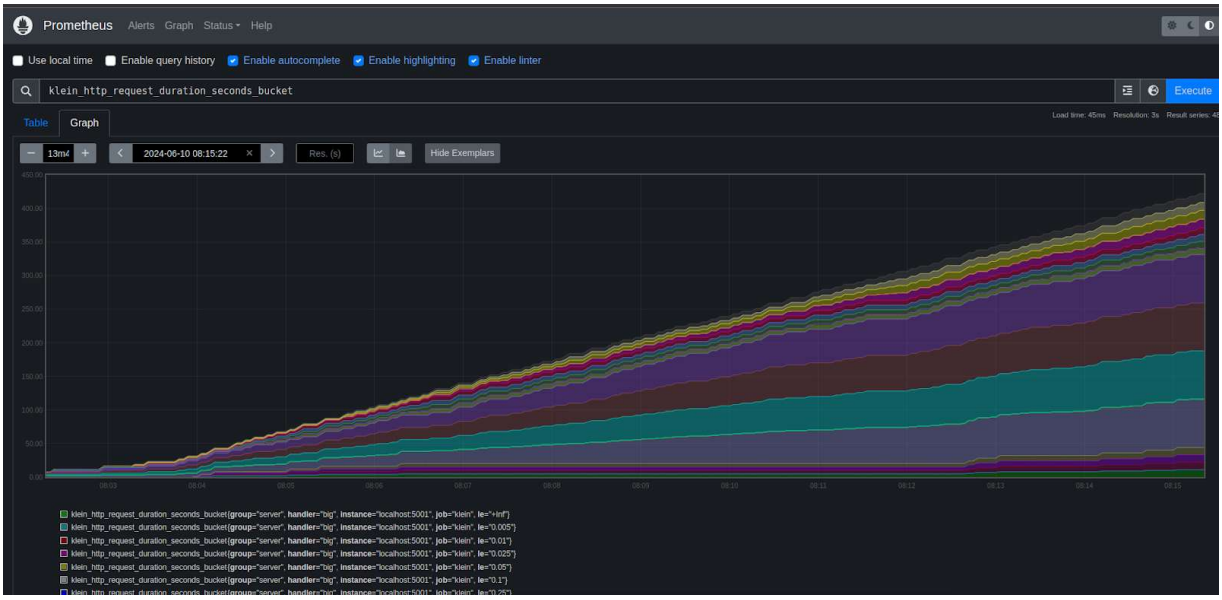
Above, we have request distribution for the duration of requests.
From this we can see that the hash function was biased to the backend server “hate”(in light blue) while other servers got lighter loads, while this may seem worrisome, it’s a consequence of the hash function used, to improve the distribution, the best solution would be to modify the hash function for requests. (it is not in any way distributive)

```
fn hash_request(req_id: usize, total_slots: usize) -> usize {
  (req_id + 2 * req_id + 17) % total_slots
}
```

The hash function for requests is shown above, a weak hash with very small distributive property.

Strathmore UNIVERSITY

3. Request latency



This graph is a bit difficult to interpret, but indicates how long a request took in the backend, prometheus groups requests according to latency range, inf stands for requests that never got a response

Request latencies, (tabular) were

Name	Latency
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="big", **instance**="localhost:5001", **job**="klein", **le**="+Inf"}	5
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="big", **instance**="localhost:5001", **job**="klein", **le**="10"}	5
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="big", **instance**="localhost:5001", **job**="klein", **le**="5"}	5
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="love", **instance**="localhost:5001", **job**="klein", **le**="5"}	6
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="love", **instance**="localhost:5001", **job**="klein", **le**="2.5"}	6

klein_http_request_duration_seconds_bucket{**group**="server", **handler**="love", **instance**="localhost:5001", **job**="klein", **le**="10"}	6
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="love", **instance**="localhost:5001", **job**="klein", **le**="+Inf"}	6
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="small", **instance**="localhost:5001", **job**="klein", **le**="2.5"}	6
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="small", **instance**="localhost:5001", **job**="klein", **le**="5"}	7
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="small", **instance**="localhost:5001", **job**="klein", **le**="+Inf"}	7
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="small", **instance**="localhost:5001", **job**="klein", **le**="10"}	7
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="hate", **instance**="localhost:5001", **job**="klein", **le**="5"}	49
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="hate", **instance**="localhost:5001", **job**="klein", **le**="2.5"}	49
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="hate", **instance**="localhost:5001", **job**="klein", **le**="10"}	49
klein_http_request_duration_seconds_bucket{**group**="server", **handler**="hate", **instance**="localhost:5001", **job**="klein", **le**="+Inf"}	49

From here, we can see that all requests took less than 2.5 seconds for most requests, no requests timed out, which indicates all servers are on.