Performance Evaluation Report

Voicu Gabriel - 341C5

Assignments completed: I, II(except bonus), III, IV

Self-evaluation: 9/10

A. System Limits Analysis

- How many requests can be handled by a single machine?

It depends on the number of concurrent requests. I tried to crash the workers directly without using the load balancer, just by making direct GET requests to my deployed servers in Heroku and I managed to overload one of the servers (503 response code with service time > 30s) after around 5000 calls, using asyncronous requests with a batch size of 1500 (concurrent connections). If this batch is relatively small (400-500), the server won't timeout.

- What is the latency of each region?

The question is rather vague, but I decided to calculate the latency by subtracting work time from response time where there is no load (even though it's not really correct because the latency depends on multiple factors), just to have a relatively good metric to see the differences between regions. So I consider the latency to be the time it takes between sending and receiving the message from server.

On my system, the average latency for EMEA is ~350ms, for ASIA is ~580ms and for US ~450ms. If I am connected to my phone's hotspot, the response times for all regions are ~150ms higher.

- What is the computation time for a work request?

The work time is ~19-21ms. It does not change no matter how many I requests I give because the servers don't do very much computational processing.

- What is the response time of the worker when there is no load?

I consider the response time of the worker to be the service time that can be found in Heroku logs. On average, it's around 21 ms.

- What is the latency introduced by the forwarding unit?

I consider the latency to be the difference between the response times of calling the workers directly (like GET https://gabrielvoicu200-worker-emea-0.herokuapp.com/work) and through the forwarding unit. The latency seems to be between 30-35ms.

The forwarding unit opens a new TCP connection to a worker at every request, so the TCP handshake is done everytime, unlike what internet browsers do, appending the "Connection:keep-alive" header every time and therefore, keeping the TCP connection persistent for a given timeout. If the connection is persistent, the response time decreases dramatically, but I did not consider this in my calculation because it wouldn't have been fair.

- How many requests must be given in order for the forwarding unit to become the bottleneck of the system? How would you solve this issue?

It depends. On my system (WSL 2), I was limited by the number of maximum number of opened sockets. ulimit -n is 1024 on my machine and if I run the python program with 10000 requests for example and the batch size of 1000 (concurrent connections), the forwarding unit crashes because the number of open file descriptors is too big. To solve this, I can increase the maximum number of file descriptors, but on WSL 2 I don't think it's possible. Furthermore, I am also limited by the specifications of my system (number of processor cores, their raw power and network connectivity) and the fact that I can't call the forwarding unit from other machines.

- What is your estimation regarding the latency introduced by Heroku?

I estimate the latency to be 'connect' parameter from the logs + service time when there is no computational work. From my tests, I could deduce that the latency is between 5-10ms.

- What downsides do you see in the current architecture design?

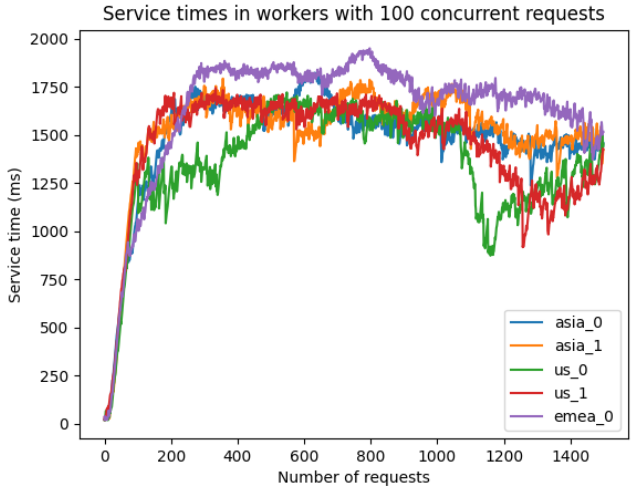
I think the way the forwarding unit routes the requests to the workers is not really efficient and is not designed for performance. As I said earlier, no TCP connection is persistent in this architecture, so even if we make 10 consecutive requests to the forwarding unit, we cannot see a decrease in response time after the first one. Secondly, on real load balancers, there are multiple optimizations that are done automatically which improve security and performance, ensuring the best end-user experience.

B. Load Balancing Policies Comparison

For the implementation I used the grequests Python library, which helps you making asynchronous HTTP requests.

Firstly, I wanted to find out how are the servers behaving when there is a huge load on them. So I made 1500 asynchronous requests on each worker with 100 concurrent connections, which means 100 requests are sent simultaneously and as soon as one of them is completed the next one is sent. I used only 1500 because that is the maximum number of logs Heroku is keeping for free use.

Then, I made a bash script which parse the logs from heroku servers and write the service times in a file for each worker. Then I read the files in my test script to create a plot and see how the service times are distributed.



As you can see, when the first 100 requests are made, the service times are relatively small (10 - 1000ms). After that, the workers are having a hard time responding, the average being aroung 1600ms. There are also some spikes that can be seen in us_0 plot, the times decreasing dramatically and then increasing again.

As far as the actual implementation is concerned, I used 5 policies:

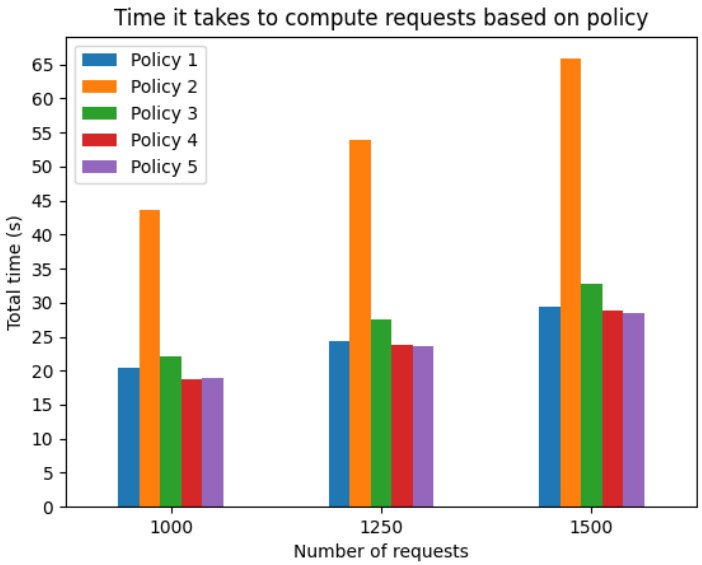
- Policy 1: Calling a random worker with batch size of 100

- Policy 2: Calling a random worker with batch size of 10

- Policy 3: Calling only EMEA worker with batch size of 100

- Policy 4: Round-robin with batch size of 100

- Policy 5: Round-robin on regions with batch size of 100



I used batch size of 100 in almost all the policies because when I tested, I tried with a variety of combinations and came to the conclusion that this batch size does not matter anymore after a certain limit, the final time it takes computing N requests being almost identical. The only difference is that when I increase the batch size, the response times and service times increase. That's normal because the servers start being overloaded with lots of concurrent connections. Before I started this intensive work, I "warmed up" all the workers to have relevant results by sending 1500 requests to each of them.

The slowest policy is the second one and it's normal. With only 10 simultaneous requests at a time, it's certainly a bottleneck for my system. I chose this policy because I can see much easier what the workers can handle and what my system is capable of doing to send multiple requests.

Then the second slowest one is the policy that calls EMEA continuously. Here we can see that we have around ~3 seconds more than any other policy except 2. I expected the difference to be much bigger, but Heroku does a great job with its workers.

To run the python program: ./main.py <number_of_requests> <policy_number> <should_start_servers>. The third argument specifies if the servers should be "woken up" or not. The default is false.