Performance Evaluation Assignment: Load balancer

Andrei Gabriel POPESCU
Group 343C4

December 1, 2020

Abstract

In this assignment we had to implement a load balancer with at least 5 load balancing policies that can handle a number N of user requests trying to offer the best performance it can possibly deliver. The assignment consists mainly of two parts: implementation and the most important one: testing/evaluation. The implementation part is a very simple one and takes advantages of useful Python 3 libraries for solving network requests. Regarding the testing part, things tend to be a little bit more complicated as is pretty hard to figure out some technical aspects of the general architecture and even some aspects of the load balancing policies.

Resolved parts of the assignment

- Prerequisites - solved
- Implementation - solved
- Evaluation - solved
- Bonus - encapsulate all in a docker container - solved

* Considering the work behind my implementation I want to think that a suitable grade is 11.0, with respect for the fact that the way I tested the application is in a purely personal manner and there is always space for improvements, but for this specific moment I think is in good shape.

1 General architecture

The main components of the project are organized respecting the above description in: implementation (which consists of all the helper files and functions that can help the LoadBalancer class to work smoothly) and testing (the testing part is centered in the file called $load - balancer - tester.py$ which calls a test suite from helper files). One thing that should be mentioned is the fact that the project aims to be very modular and very easy to change in future and for this specific reason I decided to use configuration files for any parameters that could be changed.

2 Implementation

The main purpose of a load balancer is to reduce the response time as much as possible by alternating the instances of the server which has to do the job. To implement a load balancer I needed a couple of load balancing policies (algorithms) to route more efficient the requests. In the next subsection there are 3 examples of policies that are used in this project.

2.1 Load balancing policies

In this project there are used a number of 6 load balacing policies: round robin, weighted round robin, least connection, weighted least connection, fixed weighting and randomized static. Part of them are using a custom scoring technique for choosing which endpoint is suitable and when.

2.1.1 How the scoring technique works

In order to implement a weighted algorithm the main problem is how you can assign a score to each endpoint and how accurate are these scores in time. My method was to perform a certain number of requests and to calculate the mean response time for each endpoint. But the endpoints from the same region and not only can have pretty similar response times, and to solve this, the mean response time is normalized and then all these values go trough a softmax function to make then uniformly distributed (as probabilities). The abnormal response time from the start of each machine is solved by wake them up before executing any other part of the code, in parallel.

3 Testing Methodology and Results

3.1 General architecture

* All tests were performed with 100 requests for calculating the average values.

- Number of maximum requests per instance:
  - emea-0 limit: 25, us-0 limit: 41, us-1 limit: 11, asia-0 limit: 4, asia-1 limit: 28

  These numbers can seem a little bit to small at first sight but they are the result of a stress test with continuous requests on each endpoint in which for every step the response time is compared with the current average response time, calculated using exponential moving average with dynamic parameters (1), and if is higher with more than it with ϵ time units we stop. In this way we can see an exponential drop of performance for each server.

$$m_t = r_t * \alpha + m_{t-1} * (1 - \alpha) \tag{1}$$

$$\alpha = \frac{2}{N + 1} \tag{2}$$

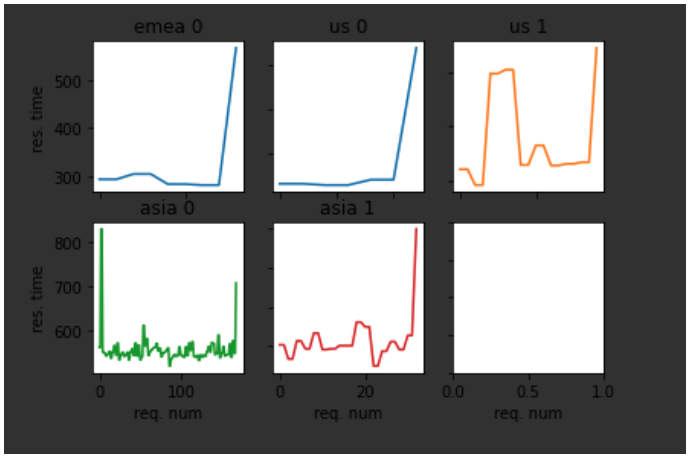

Figure 1: Performance dropout represented as an exponential increase in response time calculated after a number of iterations.

- (Average) Latency per region:
  - emea latency: 391.117, us latency: 409.092, asia latency: 611.876

  Latency is calculated as the sum of response time, internal server work time and an additional latency introduced by the forwarding unit.
- (Average) Work time per region:
  - emea work time: 19.52, us work time: 19.89, asia work time: 19.58
- (Average) Response time without load:
  - emea-0 time: 292.9, us-0 time: 480.1, us-1 time: 433.9, asia-0 time 536.9, asia-1 time: 720.5

  To obtains these numbers between requests there is a specific time δt to ensure that the server has no load for next request.
- (Average) Forwarding unit latency estimation:
  - emea: 0.312, us: 0.484, asia: 0.587

  The above results are the additional latency that were added when calculating the latency of each endpoint, obtained by estimating the computation time to deliver a request response.
- How many requests must be given in order for the forwarding unit to become the bottleneck of the system: The problem appears when the time elapsed between sending and receiving a request is greater than the server response time and the additional server work time, and this can happen when there is an exponential dropout of performance or when there are is another thread who is blocking the process. Basically, the number of requests depends of specific server and depends on specific regions, but this problem can be solved only on current system by enhancing the process resources and even wait for network to be free.
- What is your estimation regarding the latency introduced by Heroku: For Eastern Europe the Heroku servers are hosted in Germany, Frankfurt to be more precise and by calculating the distance between Bucharest and Frankfurt (1789.7 km) we can estimate the latency introduced by Heroku at about: 6 time units from response time and work time together.

- Downsides in the current architecture design: When there are frequent simultaneous client requests even for efficient load balancing techniques, servers severely get overloaded, forming traffic congestion.

  Since it is centralized in one single forwarding unit, if it fails, client requests are not accomplished. Therefore, client-server lacks the robustness of a good network.

3.2 Load balancing policies

As mentioned before the task of the load balancer is to minimize the average response time for a number N of requests that come from user, but in the same time a load balancing policy should be easy to implement and should not require a lot of resources, as the machine on which the forwarding unit runs is more exposed to possible system fails, so the main features that are analyzed in the next part are: average response time, complexity and predictability to system fails.

3.2.1 Analysis

- Complexity is denoted as a natural number between 1 and 5, with 1 being a not complex policy to implement and maintain and 5 being the most complex to implement and maintain.
- Predictability to fails is a statistical value calculated after the formula: $p = \frac{nfr}{tr}$ where nfr is the number of failed requests in current test batch. Furthermore, in this predictability value is included the probability for a thread to fail in a busy memory and busy CPU environment.

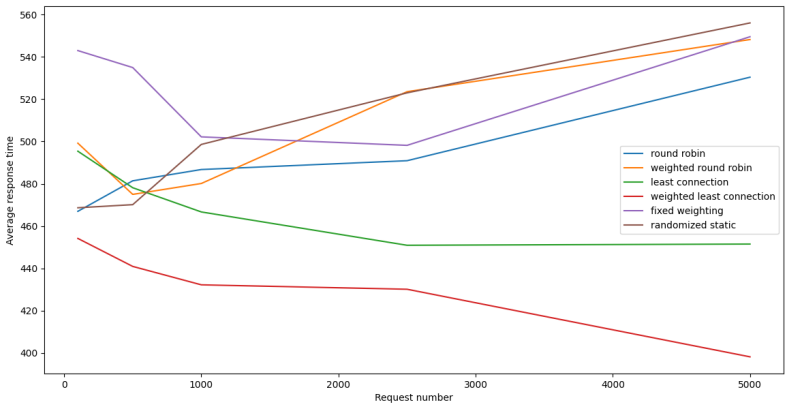

Figure 2: Over time average response time for all policies

Policy	Response time (ms)	Complexity	Predictability
Round Robin	481.4	1	5%
Weighted Round Robin	474.98	2	5%
Least Connection	495.4	4	15%
Weighted Least Connection	454.14	5	17%
Fixed Weighting	543.02	2	10%
Randomized Static	468.64	1	7%

For the numbers above there were tested all endpoints at 1000 requests. The test suite covered the response of all policies to different continuous requests which were progressively increased as follows: 100, 500, 1000, 2500, 5000. The results are represented in the next plot.

4 Conclusion

As we observe in the table above there is a correlation between complexity and performance, as the most complex to implement policy is the one with the best response time in stress tests, but this depends a lot by a couple of factors, such as the number of requests or internal state of the host machine. Is the number of request is small Round Robin or even Randomized Static has much better performance in contrast with Least Connection and Weighted Least Connection (in their case the problem is that the architecture of producer consumer implies the fact that there are constantly over a period of time things to produce and things to consume).

One the other hand we cannot conclude that one policy rule them all and the best choice in real world projects would be to select the policy based on a predefined strategy considering the number of requests.

Bonus

To resolve the network problem of the container I had to specify the current local host to be the one on the docker and in this way the building process does not include the ”–network host” parameter.