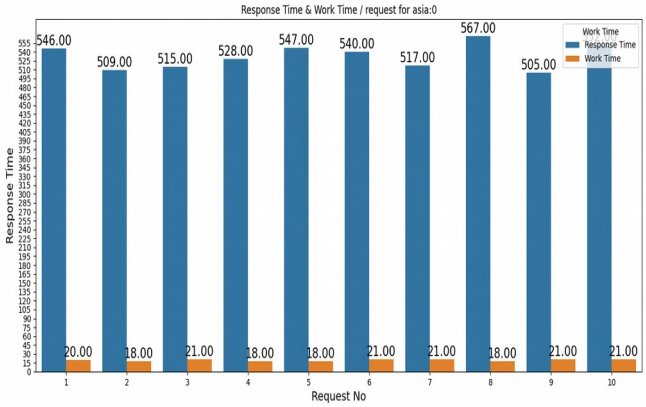
Assignment Report

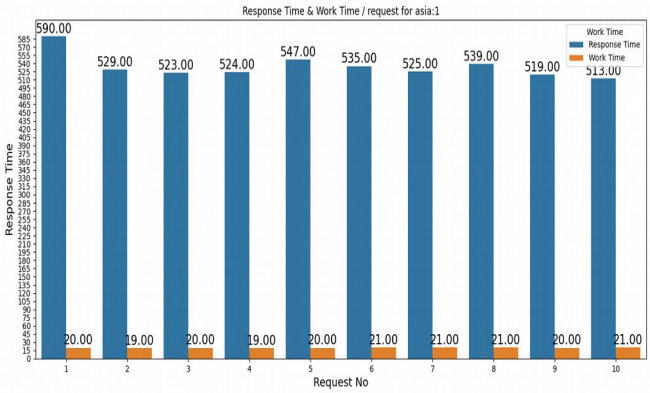
Olaru Gabriel Iulian, 342C2

All mandatory parts of the assignment were completed.
I expect to get full marks ^-^.

Evaluation strategies

☐ In order to visualize the basic behaviour of each machine (work time vs network time of a request) a batch of 10 requests was sent to each of them and the data was plotted. More details about the implementation can be found in *src/data_collection/request_time_assertion.py* . Some plots are presented below, but more can be found under *data/* .





☐ In order to asses each machine individually, Python threads were used to emulate concurrent users sending requests to the endpoints. Each user (thread) sends a batch of **N** requests (taken as command line input) – the requests in each batch are sent synchronously. More details about the implementation can be found in *src/data_collection/stress_test.py* .

☐ Relevant data (such as time spent over the network, time spent working, etc) is saved in a log file, under *data/stress_<region>_<machine_id>.log* .

☐ The logs are then parsed to compute relevant information, such as average time spent per request, average time spent on the network, etc. Details about this can be found in *src/data_collection/compute_avg_request_time.py* .

☐ More users (threads) were added with different batch sizes, until some of them were getting connection refused by the servers. The number that each machine can handle in about **1000 users**, sending batches of **10 requests** each. Further calculations are described below.

☐ A similar approach is used to test the performance of each load balancer. The difference is that instead on targeting one URL, the endpoint is chosen by the load balancer's policy.

Stress test results

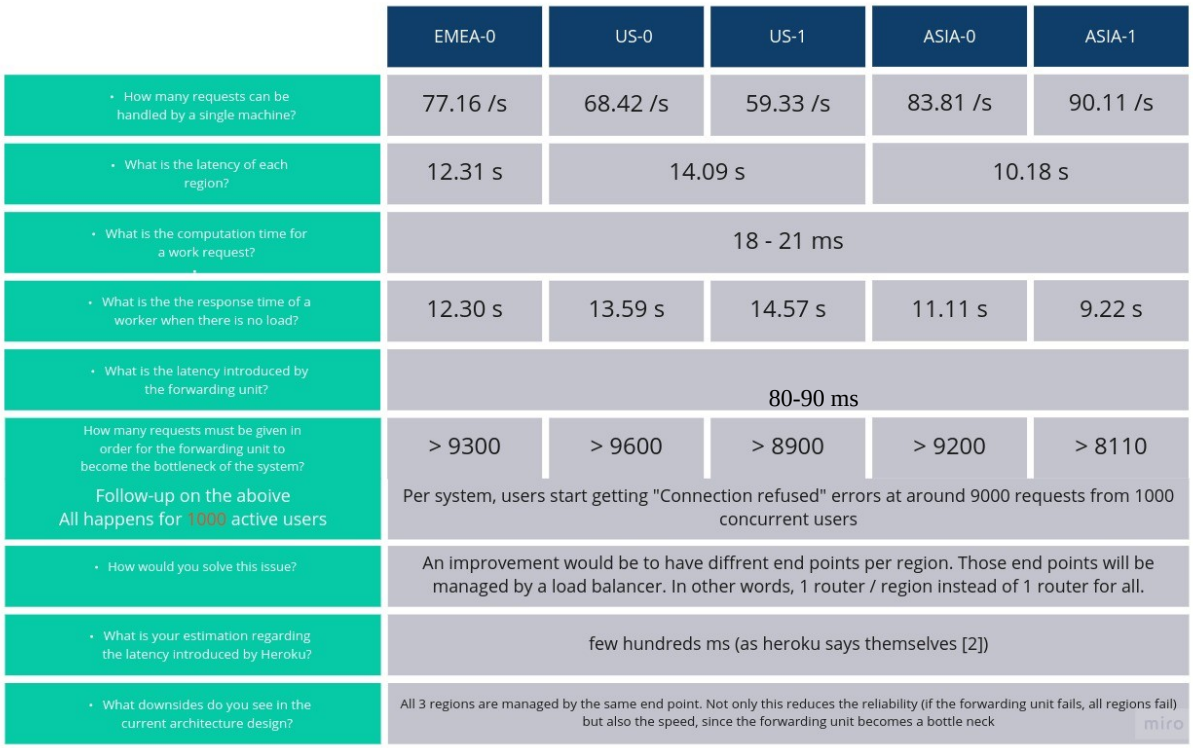
□ The number of requests / second each machine can process is represents how much traffic it can handle. This was calculated with the formula [1]: **r** = **n** / (Tresponse + Tthink), where n = concurrent users, Tresponse = request average response time, Tthink = delays caused by user (here it nears 0, since the script does not introduce artificial wait time in between requests).

☐ The average latency of each region in calculated by the script, from the logs collected. This is done by summing up the total number of requests served (not counting those which got timed out) and dividing the total time by this.

☐ The average work time vs network time was determined from the request response, in a similar manner.

☐ The latency introduced by Heroku was written in their documentation[2].

☐ Exact numbers and further answers to questions regarding the **limits of the system** can be found in the table below.



	EMEA-0	US-0	US-1	ASIA-0	ASIA-1
• How many requests can be handled by a single machine?	77.16 /s	68.42 /s	59.33 /s	83.81 /s	90.11 /s
• What is the latency of each region?	12.31 s	14.09 s		10.18 s	
• What is the computation time for a work request?	18 - 21 ms				
• What is the the response time of a worker when there is no load?	12.30 s	13.59 s	14.57 s	11.11 s	9.22 s
• What is the latency introduced by the forwarding unit?	80-90 ms				
How many requests must be given in order for the forwarding unit to become the bottleneck of the system?	> 9300	> 9600	> 8900	> 9200	> 8110
Follow-up on the aboive All happens for 1000 active users	Per system, users start getting "Connection refused" errors at around 9000 requests from 1000 concurrent users				
• How would you solve this issue?	An improvement would be to have diffrent end points per region. Those end points will be managed by a load balancer. In other words, 1 router / region instead of 1 router for all.				
• What is your estimation regarding the latency introduced by Heroku?	few hundreds ms (as heroku says themselves [2])				
• What downsides do you see in the current architecture design?	All 3 regions are managed by the same end point. Not only this reduces the reliability (if the forwarding unit fails, all regions fail) but also the speed, since the forwarding unit becomes a bottle neck				

Load Balancer Comparison

The following policies have bee implemented(more details about each load balancing policy can be found under *src/load_balancers*):

- **Random**: choosing at random from a pool of URLs – generates a load distribution based on the python's random number generator. Can also serve as a base case in comparing with others.
- **Round Robin**: choosing endpoints in a round robin fashion – one after another, as if they were in a circle
- **Weighted Round Robin**: same as round robin, but this time the **weight** of each server is taken into account. The weight is calculated based on how much traffic each machine can handle (see requests/ second in table above).
- **Least Connections**[3]: choosing an endpoint based on how many active users are connected to it. A user is considered connected just before sending the first request from a batch. The weights of each server are also taken into account, same as above.
- **Source IP Hash**: a key is generated from the addresses of a user and an endpoint. The key maps the user to a specific endpoint, to which they will send requestrs. The coice of endpoint is made based on this mapping.

The graphs below show a comparison of the policies in how much traffic they can handle, how quick they can serve queries, on average, with the same number of clients, but batches of different sizes.

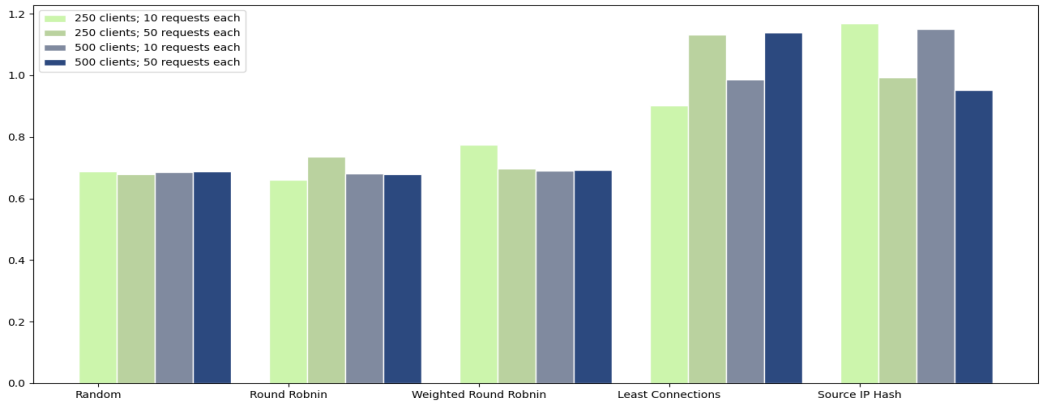
The number of concurrent users is given by the thread count, and the request batch size of each thread represents how much a user interacts with the system. We will compare how fast the requests are being processed, on average and how many requests are actually served. This will give us an idea of the trade off between latency introduced vs throughput (or how much traffic can be handled) There are 4 scenarios in which the load balancer policies are compared:

- **low** number of **users**, **not** so **active** (250 threads, 10 requests each)

- **low** number of **users**, **very active** (250 threads, 50 requests each)
- **high** number of **users**, **not** so **active** (500 threads, 10 requests each)
- **high** number of **users**, **very active** (500 threads, 50 requests each)

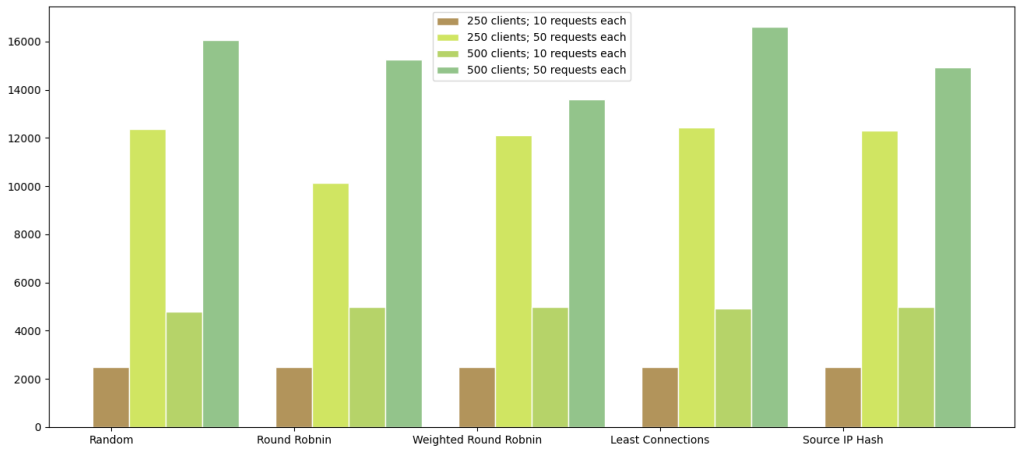
After this numbers, the system reaches its limits and the users begin to get disconnected. The bar blot below represents the response times of each policy in all different scenarios. We don't see much fluctuation in the response times of each policy across the scenarios, but when compared to each other we can observe that **Source IP Hash** introduces the most latency when there are more active users, followed by **Least Connections.**

Scenarios	Random	Robin	WRR	LC	IpHash
250, 10	0.688	0.661	0.775	0.902	1.169
250, 50	0.679	0.736	0.696	1.131	0.994
500, 10	0.685	0.681	0.691	0.985	1.151
500, 50	0.688	0.678	0.692	1.138	0.953



Below we can see how much of the requests sent by the users were actually served (because some got rejected due to the system reaching its limits). We can confirm again that even tough **Round Robin , Random,** and **Weighted Round Robin** introduce less latency, **Least Connections** and **Source IP Hash** ensure that more requests are served, therefore more traffic is possible.

Scenarios	Random	Robin	WRR	LC	IpHash
250, 10	2500	2500	2500	2500	2500
250, 50	12372	10147	12111	12417	12313
500, 10	4780	5000	5000	4921	5000



Bibliography

[1] https://docs.oracle.com/cd/E19879-01/820-4342/abfcj/index.html
[2] https://devcenter.heroku.com/articles/add-on-performance-expectations#latency
[3] http://kb.linuxvirtualserver.org/wiki/Least-Connection_Scheduling
[4] https://en.wikipedia.org/wiki/Weighted_round_robin
[5] https://www.educative.io/courses/grokking-the-system-design-interview/3jEwl04BL7Q#Load-Balancing-Algorithms