Apache Benchmark

Introduction

In this project we are going to explore basic functionalities of Apache Benchmark (ab) and implement those functionalities with **Go** language.

Some important concepts of our experiment with Apache Benchmark are:

- Number of requests to perform (n, parameter **-n**)
- Concurrency (c, parameter **-c**)
- Total time of test (t)
- Mean latency per request across all concurrent requests (calculated by t/n)
- Mean latency per request (calculated by t*c/n)
- TPS (transactions per second, calculated by n/t)
- CPU consumption
- Error in requests

Experimenting with Apache Benchmark

Following the instructions in https://github.com/jig/docker-ab, we did some experiments with ab in **docker**.
We used *docker run -d -p 8080:8080 jordi/server:http* to run a http server in localhost:8080 and *docker run --rm jordi/ab -c c -n n http://172.17.0.1:8080/* to make n requests with concurrency c to the server. We also used *top* command to observe its CPU consumption.
One example of ab's execution when c = 10 and n = 10000:

```
Server Software:
Server Hostname:        172.17.0.1
Server Port:            8080

Document Path:          /
Document Length:        23 bytes

Concurrency Level:      10
Time taken for tests:   2.162 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      1390000 bytes
HTML transferred:       230000 bytes
Requests per second:    4625.33 [#/sec] (mean)
Time per request:       2.162 [ms] (mean)
Time per request:       0.216 [ms] (mean, across all concurrent requests)
Transfer rate:          627.85 [Kbytes/sec] received
```

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 3723 root      20   0 1214936  10080   3080 S  27,6   0,3   0:55.06 docker-+
 5734 root      20   0   16252   3080   2452 R  24,3   0,1   0:00.73 ab
 2184 hmj       20   0 4562448 274860 113212 S  20,9   7,0   3:27.55 gnome-s+
 3749 root      20   0  108352   8360   3968 S  15,6   0,2   0:40.32 server
```

Figures 1 & 2: Example of ab's execution when c = 10 and n = 10000 and its CPU consumption

This table shows all results of ab's execution making requests to localhost with n = 10000 and different values of parameter c:

n	c	Total Time (s)	TPS	CPU (%)	Error (%)	Mean Latency (ms)
10000	1	5.011	1995.74	13.3	0	0.501
10000	2	3.211	3113.97	29.5	0	0.321
10000	3	2.679	3732.76	23.6	0	0.268
10000	4	2.391	4182.58	27.6	0	0.239
10000	8	1.754	5702.57	28.9	0	0.175
10000	10	2.162	4625.33	24.3	0	0.216
10000	50	1.768	5656.37	58.1	0	0.177
10000	100	1.579	6333.22	46.5	0	0.158

Figure 3: Table of results of ab's execution when n = 10000 varying c

Plot of CPU consumption:

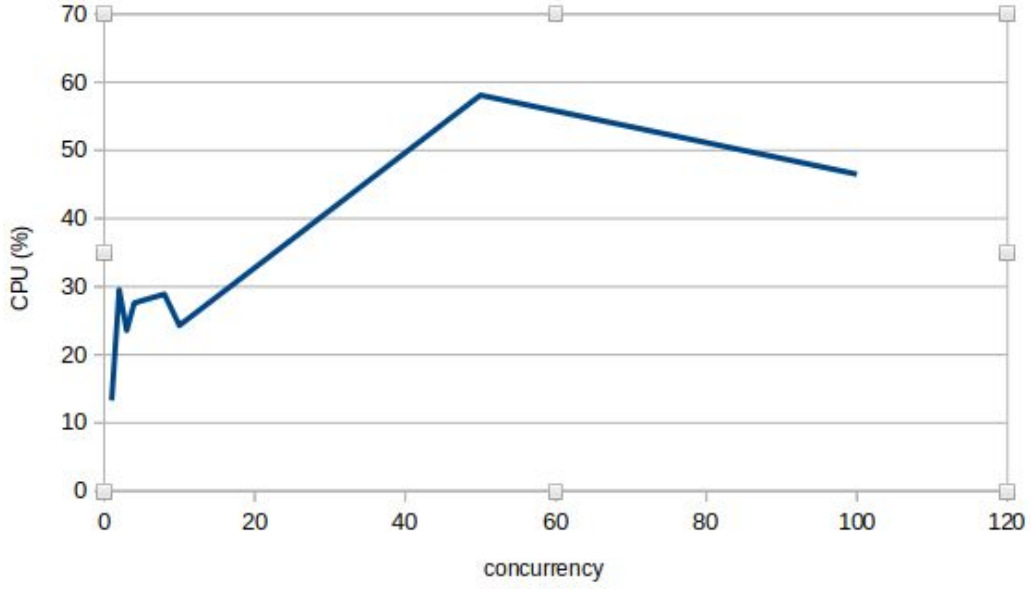

Figure 4: Plot of CPU consumption with different values of c

Plot of TPS:

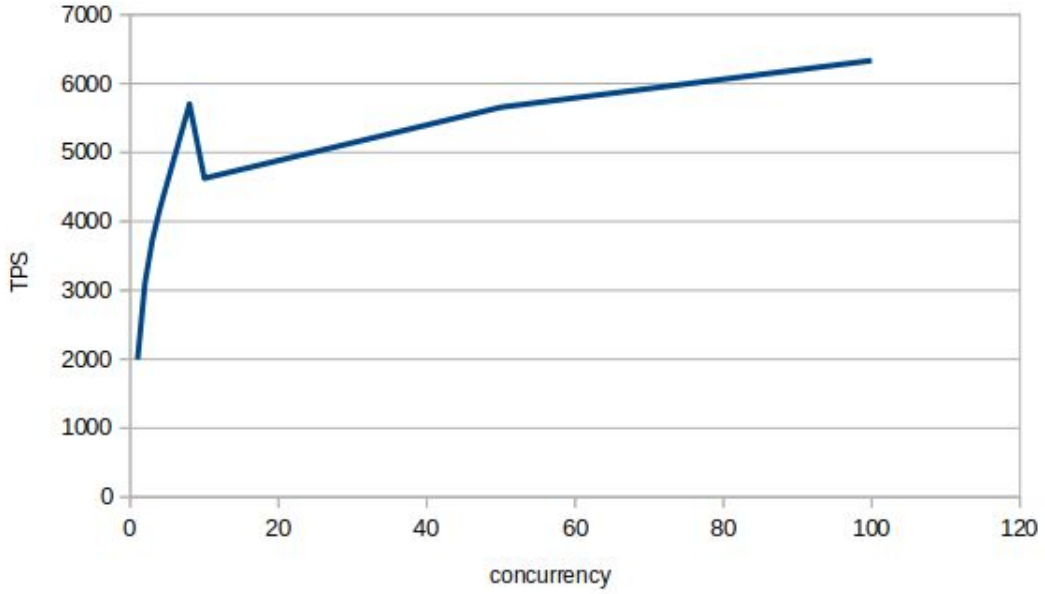

Figure 5: Plot of TPS with different values of c

Plot of Mean Latency:

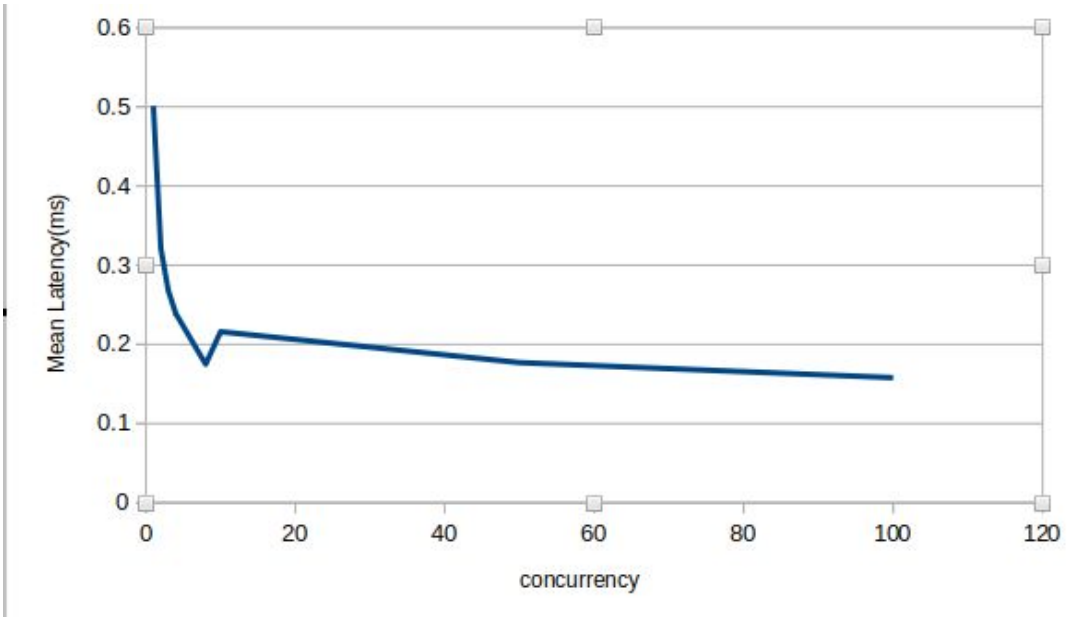

Figure 6: Plot of Mean Latency with different values of c

We can observe that in the case of localhost, we have no errors and when we increase concurrency, the TPS and CPU consumption increases and Mean Latency decreases because now we send more requests at a point of time, but we can't obtain an ideal speed-up by this as the server is not ideal and can't respond all requests with same amount of time.

Now we are going to experiment with https://www.docker.com/ and n = 100, using the command *docker run --rm jordi/ab -c c -n n https://www.docker.com/*. Example with c = 10:

```
Server Software:        nginx
Server Hostname:        www.docker.com
Server Port:            443
SSL/TLS Protocol:       TLSv1.2,ECDHE-RSA-AES128-GCM-SHA256,2048,128
TLS Server Name:        www.docker.com

Document Path:          /
Document Length:        73127 bytes

Concurrency Level:      10
Time taken for tests:   7.163 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      7406200 bytes
HTML transferred:       7312700 bytes
Requests per second:    13.96 [#/sec] (mean)
Time per request:       716.282 [ms] (mean)
Time per request:       71.628 [ms] (mean, across all concurrent requests)
Transfer rate:          1009.74 [Kbytes/sec] received
```

Figure 7: Example of ab's execution with https://www.docker.com/, n = 100, c = 10

All results of ab's execution making requests to https://www.docker.com/ with n = 100 and different values of parameter c:

n	c	Total Time(s)	TPS	CPU(%)	Error(%)	Mean Latency(ms)
100	1	21.93	4.56	5.6	0	219
100	2	13.04	7.67	10	0	130
100	3	8.7	11.49	12.3	0	87
100	4	7.89	12.67	22.9	0	78
100	8	7.409	13.5	17.5	0	74
100	10	7.163	13.96	11.3	0	71
100	25	5.743	17.41	19.6	0	57
100	50	5.348	18.7	25.2	0	53

Figure 8: Table of results of ab's execution

Plot of CPU consumption:

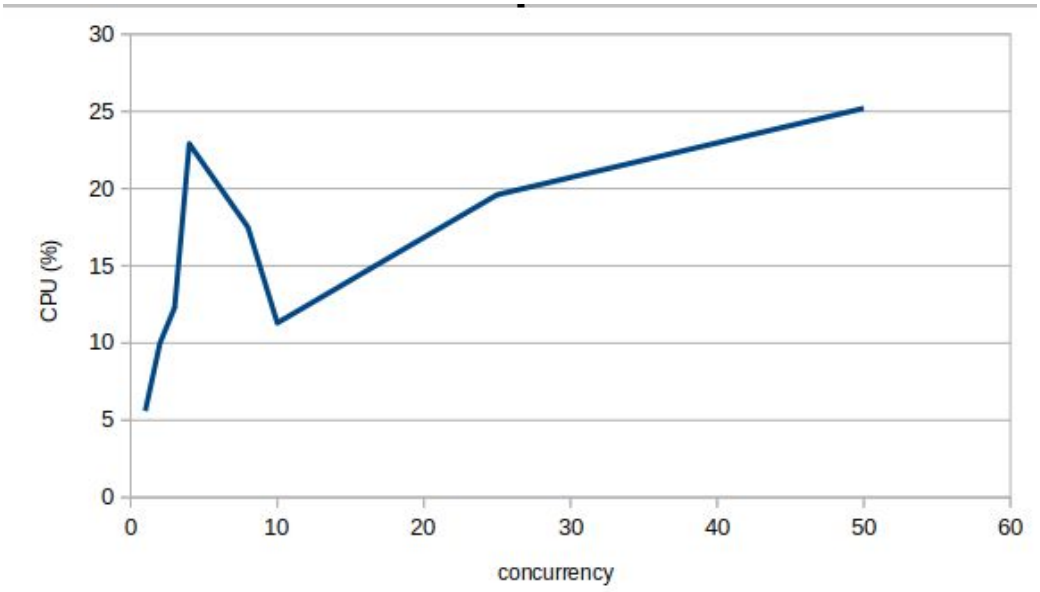

Figure 9: Plot of CPU consumption with different values of c

Plot of TPS:

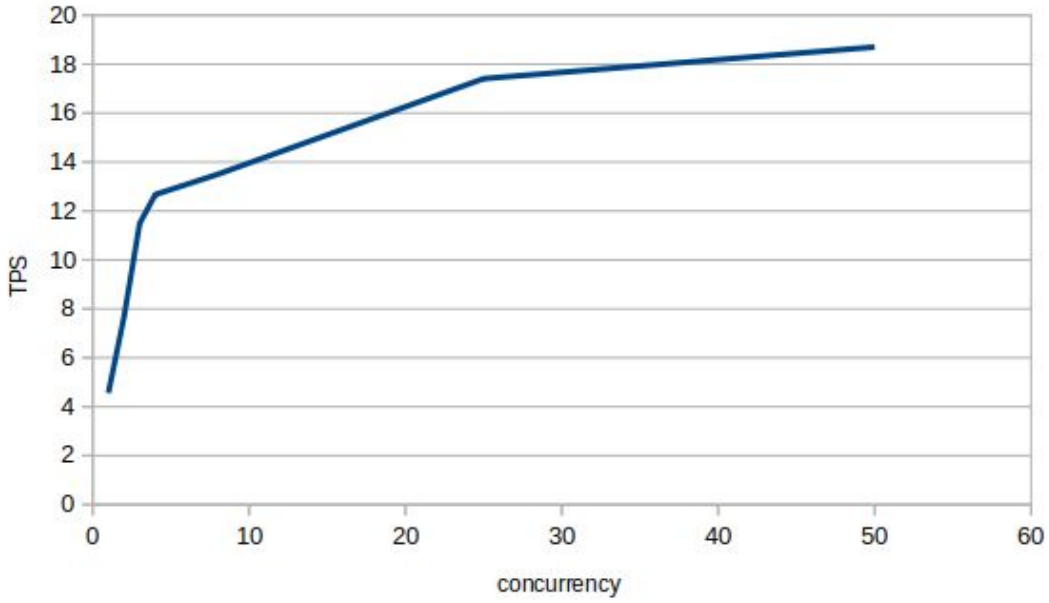

Figure 10: Plot of TPS with different values of c

Plot of Mean Latency:

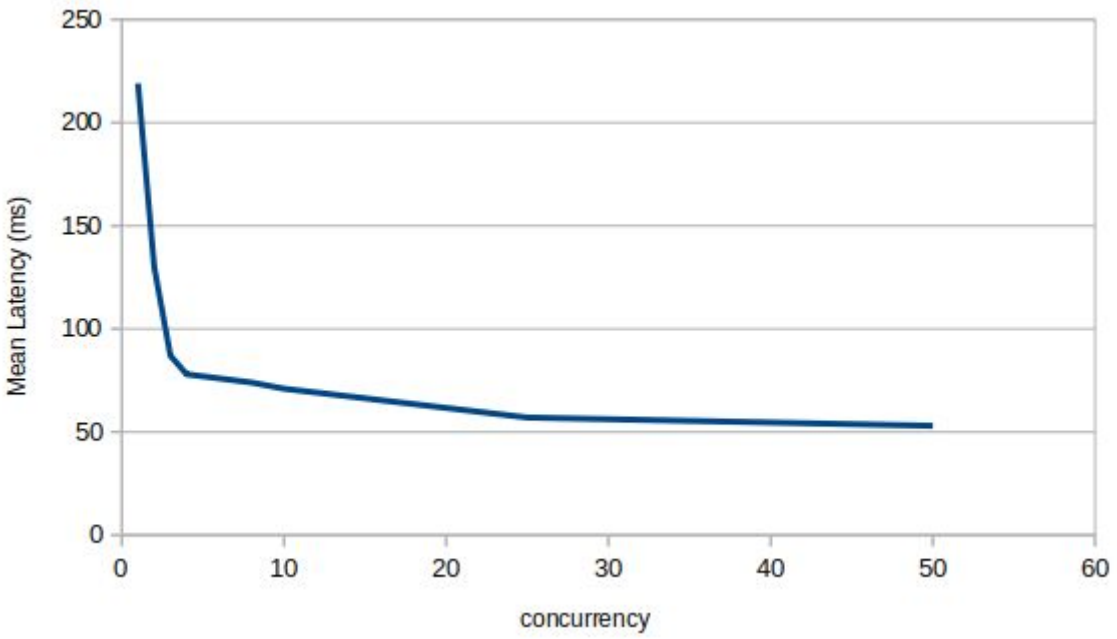

Figure 11: Plot of Mean Latency with different values of c

We can observe that the behaviour of https://www.docker.com/ is similar to the behaviour of localhost. In both cases the best value of concurrency is around 8 and 10, with which we have more TPS, less latency and not too much CPU consumption.

Another parameter of ab is **-k**, which means enable HTTP KeepAlive feature. When HTTP KeepAlive is enabled, it can reuse the previous connection, thus doesn't need to open a new connection for every request and reduces latency. For example, when we execute *docker run --rm jordi/ab -k -c 10 -n 10000 http://172.17.0.1:8080/*, the result is shown below:

```
Server Software:
Server Hostname:        172.17.0.1
Server Port:            8080

Document Path:          /
Document Length:        23 bytes

Concurrency Level:      10
Time taken for tests:   0.597 seconds
Complete requests:      10000
Failed requests:        0
Keep-Alive requests:    10000
Total transferred:      1630000 bytes
HTML transferred:       230000 bytes
Requests per second:    16738.67 [#/sec] (mean)
Time per request:       0.597 [ms] (mean)
Time per request:       0.060 [ms] (mean, across all concurrent requests)
Transfer rate:          2664.46 [Kbytes/sec] received
```

Figure 12: Example of ab's execution with KeepAlive enabled

There are 10000 Keep-Alive requests and it takes less time than the execution with same c and n but without KeepAlive.

A simple implementation of Apache Benchmark using Go

In this part of project we have implemented all basic functionalities mentioned above (**-n, -c and -k**) and also made a simple HTTP server using **Go** language. All codes of the program are in the Github repository.

In the main program, to implement the -c functionality we have used a **dynamic task decomposition** strategy, creating c goroutines to do the job and assigning jobs dynamically to every goroutine. The main program assigns jobs to all goroutines using channel ch and waits all goroutines to finish and collects results using channel cerr.

To compare the performance of our program with ab, we executed the program to test http://localhost:8080/ (with our HTTP server) and https://www.docker.com/ with the same parameters.

Results of localhost (we haven't measured the CPU consumption because the little execution time of the program makes it difficult to measure with *top*):

```
> ./main -n 10000 -c 1 http://localhost:8080/
Time taken for tests:  3.654  seconds
Complete requests:  10000
Failed requests:  0
TPS:  2736.7268746579093  [#/sec] (mean)
Time per request:  0.3654  [ms] (mean)
Time per request:  0.3654 [ms] (mean, across all concurrent requests)
```

Figure 13: Example of execution of the program

n	c	Total Time (s)	TPS	Error (%)	Mean Latency (ms)
10000	1	3.654	2736.72	0	0.3654
10000	2	2.082	4803.07	0	0.2082
10000	3	1.692	5910.16	0	0.1692
10000	4	1.51	6622.51	0	0.151
10000	8	1.34	7462.68	0	0.134
10000	10	1.289	7757.95	0	0.1289
10000	50	1.219	8203.44	0	0.1219
10000	100	1.329	7524.45	0	0.1329

Figure 14: Table of results of localhost

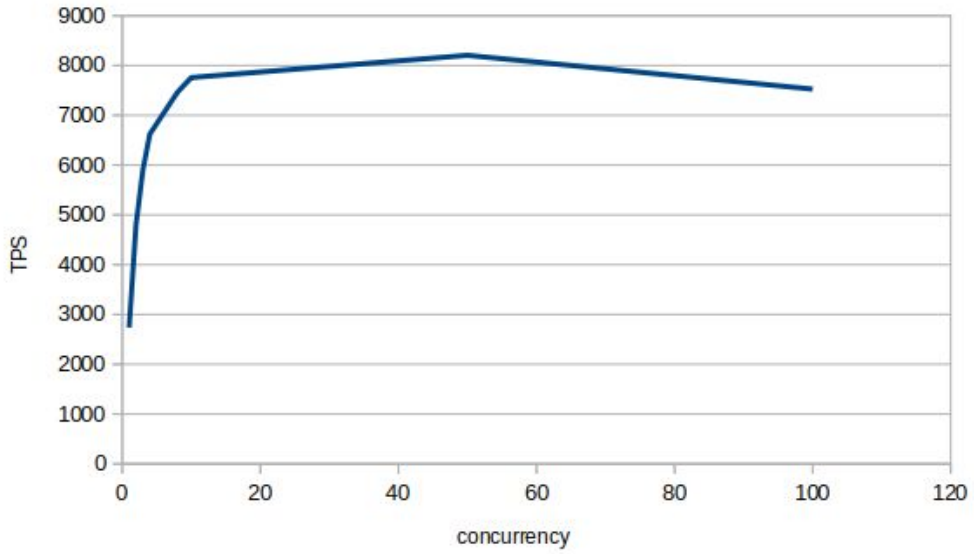

Figure 15: Plot of TPS with different values of c

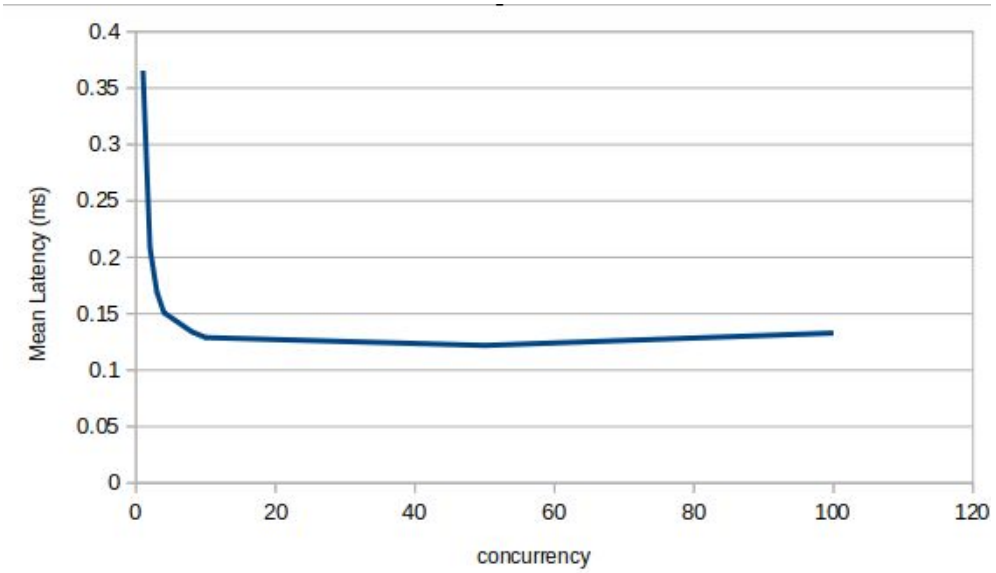

Figure 16: Plot of Mean Latency with different values of c

Results of https://www.docker.com/:

```
> ./main -n 100 -c 1 https://www.docker.com/
Time taken for tests:  6.028  seconds
Complete requests:  100
Failed requests:  0
TPS:  16.5892501658925  [#/sec] (mean)
Time per request:  60.28  [ms] (mean)
Time per request:  60.28 [ms] (mean, across all concurrent requests)
```

Figure 17: Example of execution of the program

n	c	Total Time(s)	TPS	CPU(%)	Error(%)	Mean Latency(ms)
100	1	6.028	16.58	5.6	0	60
100	2	3.064	32.63	13.2	0	30
100	3	2.29	43.66	4.3	0	22
100	4	1.703	58.72	6.3	0	17
100	8	2.571	38.89	15.9	0	25
100	10	1.633	61.23	12.3	0	16
100	25	2.084	47.98	19.9	0	20
100	50	2.197	45.51	21.9	0	21

Figure 18: Table of results of https://www.docker.com/

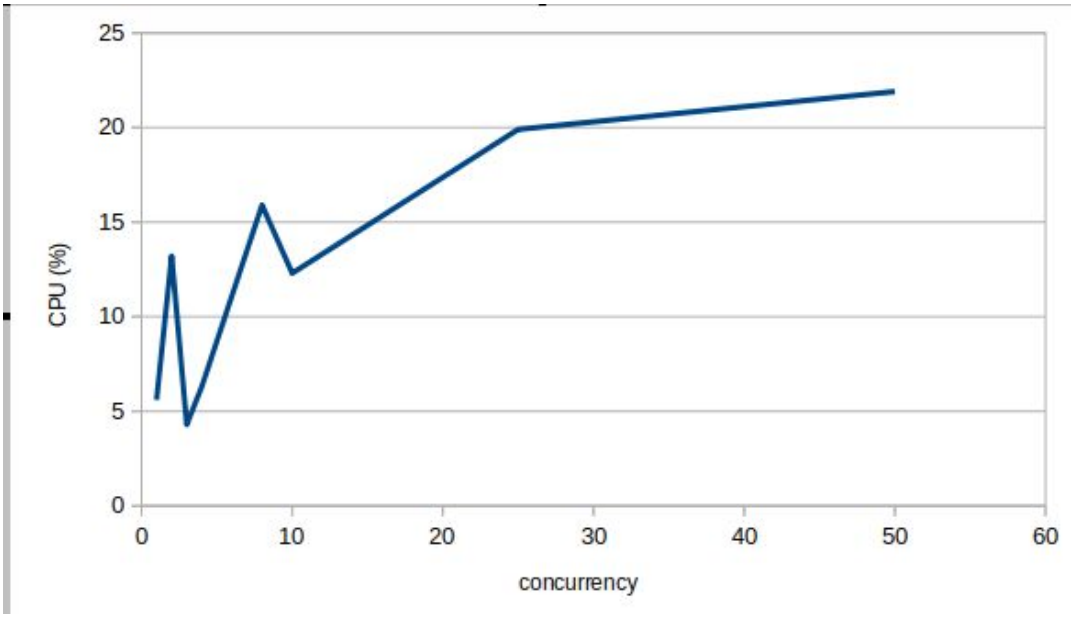

Figure 19: Plot of CPU consumption with different values of c

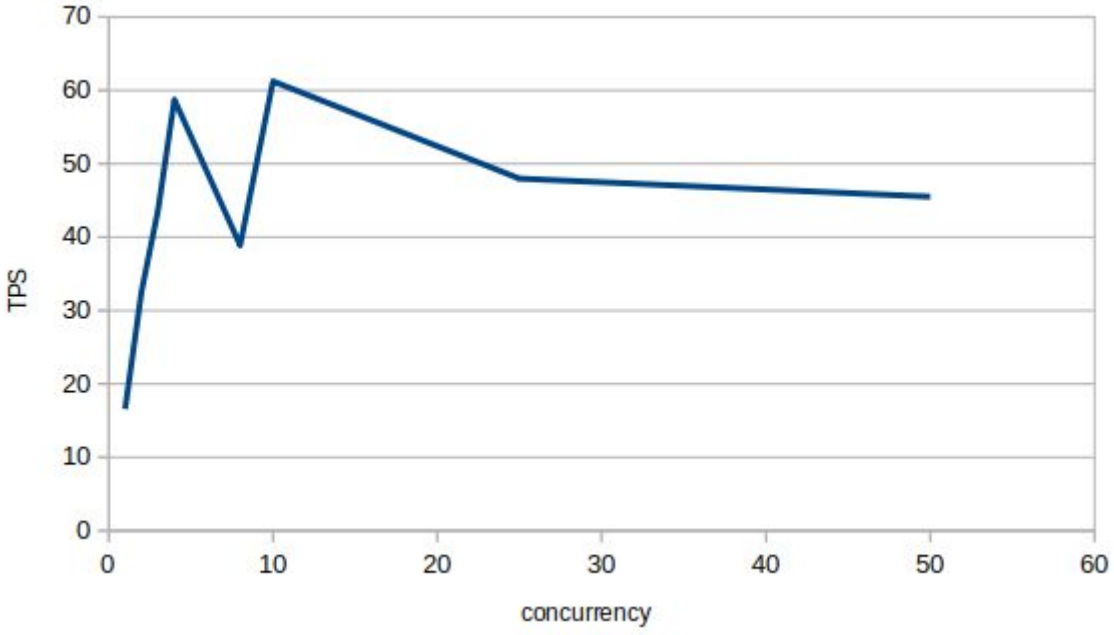

Figure 20: Plot of TPS with different values of c

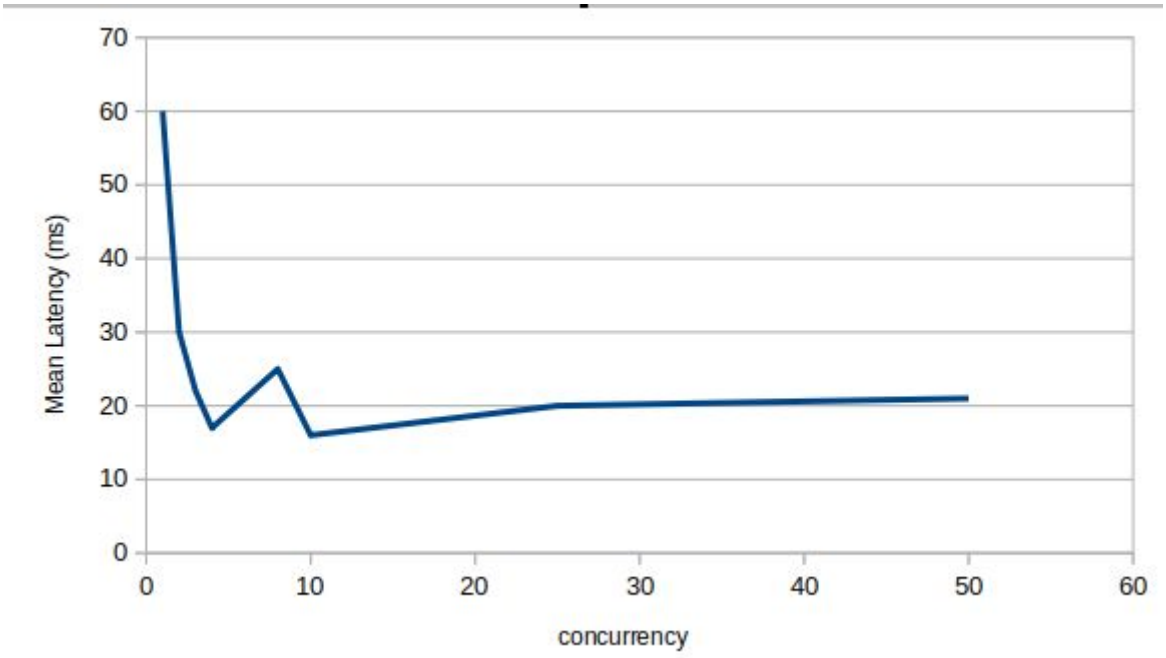

Figure 21: Plot of Mean Latency with different values of c

We can see that the **Go** version of ab performs better than the original ab because as a programming language designed for being concurrent, Go can handle this type of task much better and with more efficiency. The CPU consumption is similar to the original ab, when we increase c the CPU consumption also increases. We also observe that there is a point when we increase c it increases latency and decreases TPS probably because it overloads the server.

Conclusion

In this project we have learned basics of **docker, ab and HTTP** and how to program with **Go**. We also would like to point that the results we obtained has shown us that increasing concurrency doesn't always increase performance.

Finally, we can say that this project not only helped us to prepare for the practice, but also improved our ability of managing projects for the future.