Contents

[Part A 2](#_Toc122460920)

[Part B 9](#_Toc122460921)

[In conclusion 16](#_Toc122460922)

## Part A

1. For the first part of the experiment, we ran our base case scenario so that we could obtain some benchmarks. We created a cluster of 10 servers, each equipped with the same hardware configuration to mitigate any hardware unfairness. Each server was equipped with dual Intel Xeon 16-Core CPUs and 384GBs of RAM. We left one server exclusively for the wrk workload generator so that it would not tamper with our results. We ran our tests as follows, we kept the duration of the tests at 30 seconds each and we kept the workload generator’s threads static at 10 threads since we found out that 10 threads are the sweet spot. To measure the impact of the increase of the connections, we kept the request throughput the same @ 2000 Req/sec and we were increasing the connection count by 100 in each iteration. To measure the effect of throughput, we kept the connections fixed @ 200 connections, and we were increasing the throughput by 1000 in each iteration.

At first, we ran the base case of the HotelApp, as microservices on the same server. We ran two tests, one to evaluate the effect of the connections and another for the throughput.

After we’ve collected our baseline results, we chose one service to modify and scale out to see what effect it might had on the performance. We chose to scale the “Profile” service. For that to work, we had to create a load balancer to redirect the gRPCs from the “frontend” service to the multiple instances of the “Profile” service. We also modified the “frontend’s” service image to connect to the Load Balancer’s IP and from there to connect to the “Profile” service. For scaling-out our app we used docker swarm and we joined the 7 remaining servers to the swarm.

We started off with 4 load balancers and 4 instances of the “profile” service.

As we can see, with just 4 LBs and 4 Profile instances, we managed to create a massive bottleneck in our app. We constantly experience latency times in the 10s of seconds with periodical dips in certain iterations of the experiments. Overall, the latency was increased dramatically compared to the Single Node configuration. When we looked at the throughput of this test, we could see that not only we had high latency times, but we were also dropping connections. Looking at both graphs, we can see that when the Latency graph dips, the throughput graph peeks and vice versa. This is due to the high latency times hindering the performance and the total requests. When we looked at the Load Balancers’ logs we found out that some connections did not get forwarded and the service “Profile” was unavailable which explains the high latency times.

Moving on, we increased the number of load balancers and profile instances to 10 each and reran the tests. We found out that the latency got much better, within the millisecond region. While we fixed the connections related latency problem, when we ran the throughput test, we found out that when we passed the 3000requests/sec the latency got bad and very quickly hit the 10s of seconds again thus dropping the throughput rate and the total data transfer. We can see that we hit the maximum throughput at around 5000 requests/sec since the graph stays in the same range.

Finally, we increased the number of load balancers to 20 and we created 100 instances of the profile service. As expected, on our connections test, the latency times were almost identical to our benchmark Single Node results while our throughput test was also close to our benchmark. When we looked at the throughput graph of the throughput test, it was almost linear, just like the graph of our baseline metrics and thus our microservice implementation can be scaled by adding more instances of the services.

# Part B

1. For the second part of the experiment, we modified the “rate” service to support getting data from a database. We’ve also created a MongoDB service on our swarm to be able to transfer the data to the database. We had to modify the protocol buffer’s implementation and change the JSON format of the data structures to BSON to be able to import our data to MongoDB. Unfortunately, we could not run this part of the experiment on the servers that were used on the previous one due to Cloudlab’s maintenance schedule. We created 8 identical servers each equipped with a 10-core Intel Xeon CPU and 256GB of DDR4 RAM. The network card on these servers is slower @ 10Gbps. For the first part of the experiment, we ran the MongoDB implementation on a single server to get our baseline results.

We can see that the latency has been increased when we introduced the database. Compared to the first experiment the latency has been increased and it reaches up to 120 ms. Our throughput though, was not really affected since it stays almost the same.

When we ran the throughput test, we identified that we are capped at around 7000 requests/sec and then the throughput stays the same. We can see that the latency increases dramatically after that point.

We then scaled our services, and we created 20 instances of the MongoDB databases and 20 instances of the “rate” service and deployed the app on the swarm we created before.

As we can see, when we scaled our app to other nodes, we introduced network latency, and our latency has been increased. We also saw a drop in the throughput.

Running our throughput test showed that up to 6000 requests/sec @ 200 connections, the latency is similar to the single node implementation. After that point we see a massive increase in latency and the actual throughput stays the same meaning that we hit a ceiling.

Moving to the second part of the second experiment, we added a caching layer between the “rate” service and the MongoDB by utilizing the Memcached service. When we access the Database, we also add the item we fetch to the Memcached service for faster access times. We followed the same procedure as the rest of the tests, we initially ran our app on a single node to get our baseline results.

Running our connections test, we can see that the latency has been increased a lot and has reached the 10s of seconds and the throughput is not constant as it should have been. The results really bothered us since the latency should have dropped due to the caching service.

After running the throughput test, we saw an increase in latency after 1000 requests/sec, meaning that we after 1000 requests/sec the throughput is not increasing and we are hitting a ceiling due to the high latency times.

We can see that the throughput graph stays the same after 1000requests/sec.

We then ran our app on the swarm with 20 MongoDB databases, 20 Memcached services and 20 Rate services.

On our connections test we observed that the latency times have improved a lot and almost match the original’s. The throughput is almost constant at around 16MB. This means that by increasing the services of our app, we increase its connection handling capabilities, and we give it more headroom for more concurrent users.

When we ran our throughput test, we saw that the maximum throughput has been increased from 2000 requests/sec all the way up to 7000 requests/sec. After that point the throughput stays the same meaning that we hit an upper ceiling.

In conclusion, the takeaway here is that 1) by load balancing the gRPCs and increasing the services can result in better performance and allows us to scale our application according to our needs, 2) By using multiple caching layers we can expect drops in latency and increase in the throughput of a microservice-based application and 3) It requires careful inspection of which service requires scaling. If we scale the wrong service (i.e a service that does not have much utilization) we won’t see any performance improvements.

Notes: All the graphs and statistics are available in the excel file Report.xlsx – All the docker images used are available at the Docker repository ssofok02 for replicating our experiment.