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.

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.