Note: All the numerical data are in the excel file called FinalData.xlsx

Docker Hub: [ssofok02's Profile | Docker Hub](https://hub.docker.com/u/ssofok02)

Part 2:

1. We ran the monolithic implementation of the HotelMap app to get some baseline metrics so we could evaluate and compare with the microservices implementation of the app. We also modified the supplied LUA script such that it would only make requests for the search functionality of the app and ignore any other unimplemented functionalities. Furthermore, we used a newer version of the wrk which allowed us to get more accurate results for the 99.999th latency percentile and keep the throughput constant. We chose a fixed number of threads for the workload generator @ 10 threads since it was a good balance between the available cores of the machine we were using and the handling of the connections. Each test was left to run for 30 seconds. 10 tests were made, each with increasing number of connections (100-1000, increments of 100). We kept the throughput constant to get more accurate latency results.

In the microservices implementation of the app, we broke down each major component into a microservice running in a different container. A major difference in performance is present at the 99th and 99.999th percentiles. We can see that in the microservices implementation the 99.999th percentile latency is much greater than the monolithic. This suggests that some extra latency is added due to the gRPCs communication. It should be noted that in such small app, it’s normal to see loss in performance when using microservices since the latency between each service is larger than the work load of each service.

Graphical user interface, application

Description automatically generatedThis is a screenshot taken from Jaeger which we used to monitor the gRPC. We can see that the microservice “frontend” takes the most time to complete. In this example we can see the invocation of the microservice “frontend” which in turn called the “geo” microservice and then the “profile” to get the profile for the requested “geo” location. The two circled regions show the time spent in the RPC while doing nothing.

1. For the second part of the evaluation, we set up a Docker Swarm to deploy our microservices in different remote nodes. We created 6 nodes from which we used 5 to run the microservices and 1 to run the workload generator. It should be noted that each of node was configured with 256GB of very fast DDR4 RAM, had a 100Gb NIC and a 32 core CPU @ 3.0GHz. Such powerful servers resulted in the results we got. We can clearly see that the microservices implementation when distributed to other nodes, performs much better than the microservices running on a single node. When we observed each node with htop we saw that the CPU utilization of each system was constantly under 30%, even when we had the workload generator running with 1000 simultaneous connections. For reference, when we were running the microservices implementation on a single node we observed CPU utilization at around 60-80% especially at higher number of connections. Compared to the monolithic implementation though, we did not come close to its performance probably due to the latency between the gRPCs. We then traced the gRPCs with Jaeger and we got similar or faster results with the Microservices Single Node implementation.

We then calculated the average latency of each implementation and we found out that the Microservices Implementation on a Single Node was about 3.7 times slower than the Monolithic and the Microservices Swarm Implementation was about 3 time slower than the Monolithic. From these results we can conclude that the microservices implementation of this app did not result in increasing the performance but rather hindered them. => The gRPC latency is greater than the execution of each procedure, thus creating a bottleneck and dropping the performance.