Evaluation Doc

For this part, we started t2.medium EC2 instance in the AWS and deployed our application there. We deployed our application in the remote server by following the below:

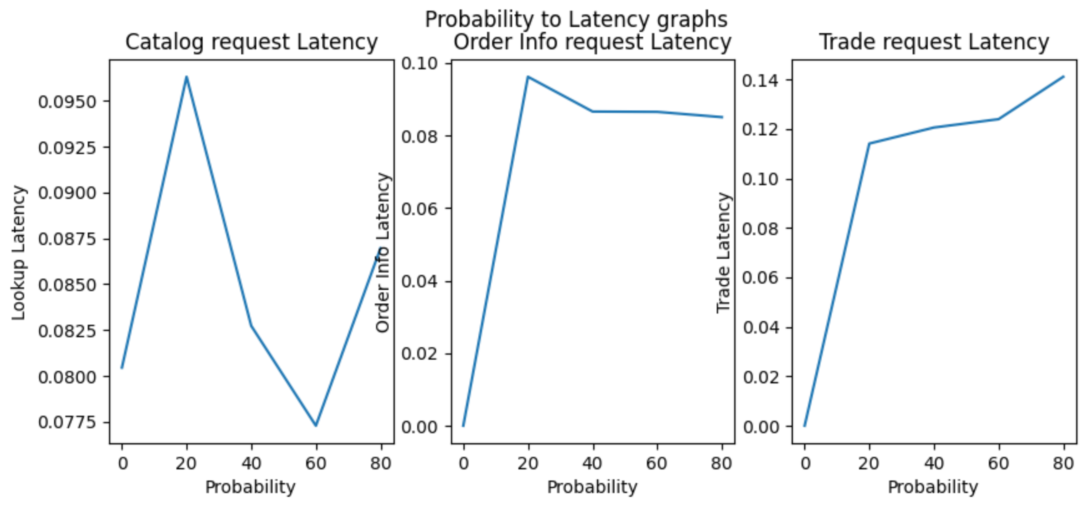
1. We cloned our repo from git
2. We installed the necessary linux packages, to install pip3 library we require python3.7. But by default we get python3.6 in the AWS instance. So to install all the dependencies we have written a script 'install.sh'. On running this script, it will install all the dependencies including Flask, as we are using Flask in our application. The script is located in `src/install.sh`
3. In `src/config.json` file we are maintaining all the ports for the respective microservices. So to run our application with the defined ports, first we need to change the ports in the `src/backend.sh` and `src/frontend.sh` scripts for the backend and frontend respectively. We also need to open the necessary ports to the public with the following command.aws *ec2 authorize-security-group-ingress --group-name default --protocol tcp --port <port_no> --cidr 0.0.0.0/0.* Replace port_no with the required port number.
4. Now to run our backend services, run `bash backend.sh` located at `src/backend.sh` and to run frontend, run `bash frontend.sh` located at `src/frontend.sh`. Please run in the same order as frontend have dependencies on the order service.
5. We can get the public ip of our machine using `curl ifconfig.me` command and replace the frontend's host in the config.json to test the client program in your local machine.
6. We stopped our application by this command `kill $(lsof -t -i:<port_no>)`. Replace port_no with the appropriate port number.

Plots:

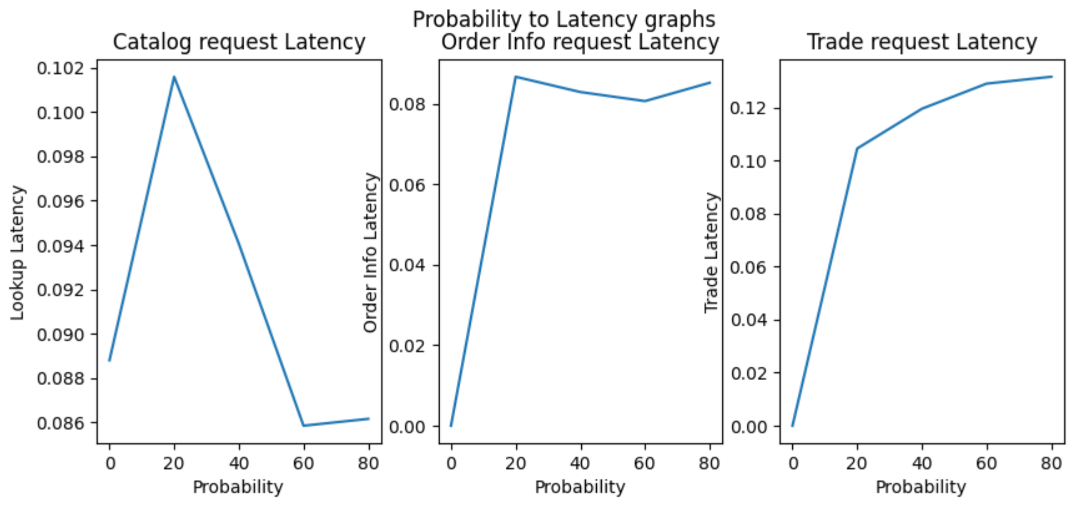
We plotted the plots by varying the probability from 0 to 80 in steps of 20 and running the 5 clients concurrently and each client sends 100 sequential requests to the frontend service. The average of the latency is calculated for each probability and the following plots were plotted. One with cache enabled and one with cache disabled.

Plots for probability vs latency for 3 different requests namely lookup, trade and order_info requests.

With cache enabled:



Without cache:



By observing the plots, since caching functionality is only enabled for lookup requests, we can see that the latency for requests with cache is lesser than the latency for requests without cache. This is an expected behavior as caching reduces the latency.

<u>**Questions:**</u>

Q) Can the clients notice the failures?

A) No, clients didn't notice the failures

Q) Do all the order service replicas end up with the same database file?

A) Yes, all the order service replicas ended up with the same database file.