CS 677 S21 Lab 3 - Pygmy

Evaluation Document

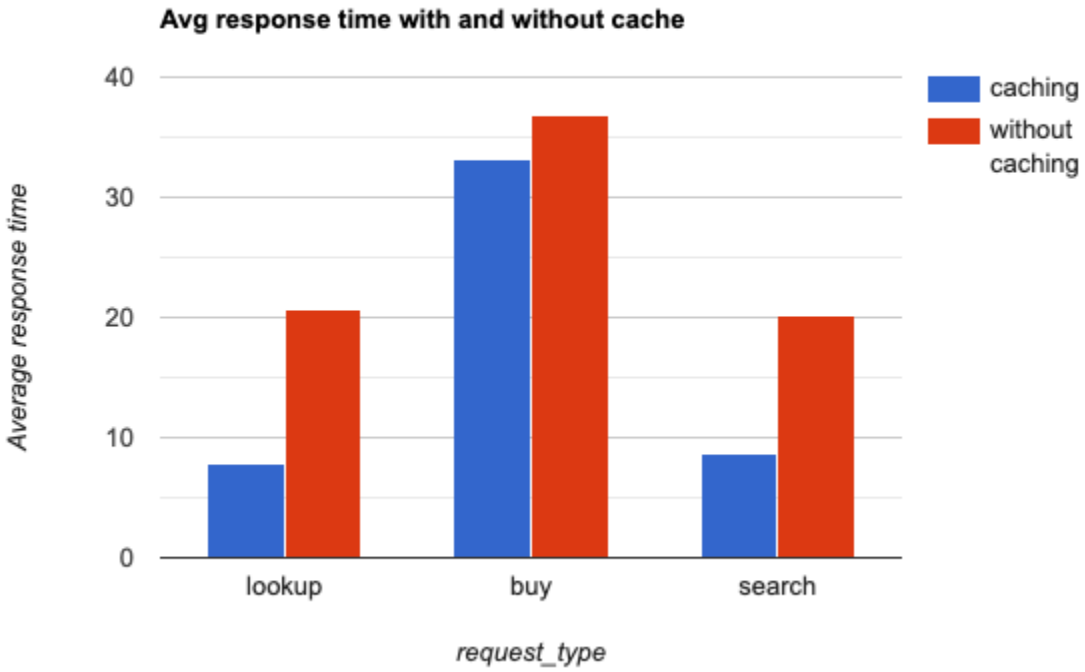
Submitted By
Yelugam Pranay Kumar
Date: 30 April, 2021

Compute the average response time (query/buy) of your new systems as before. What is the response time with and without caching? How much does caching help?

The following statistics are for reported after performing the experiments on the **docker containers locally:**

- Average time for 1000 requests with **Caching**:
  - Average time for lookup requests is: 7.95
  - Average time for buy requests is: 33.16
  - Average time for search requests is: 8.70
- Average time for 1000 requests without **Caching**:
  - Average time for lookup requests is: 20.60
  - Average time for buy requests is: 36.84
  - Average time for search requests is: 20.12

As we can see, the average time is much greater when there's no caching. The code for this can be found in the tests folder and averageTimeCache_1.py file.



Construct a simple experiment that issues orders or catalog updates (i.e., database writes) to invalidate the cache and maintain cache consistency. What is the overhead of cache consistency operations? What is the latency of a subsequent request if it sees a cache miss?

- Average time for lookup requests is: 26.796 (before caching)
- Average time for lookup requests is: 7.043 (after caching)
- Average time for buy requests is: 61.122 (buy request time & cache invalidated)
- Average time for lookup requests is: 19.894 (resp time without caching)
- Average time for search requests is: 21.699 (resp time for search req without caching)
- Average time for search requests is: 7.292 (resp time for search req with caching)
- Average time for buy requests is: 59.417 (buy request time & cache invalidated)
- Average time for search requests is: 21.163 (resp time for search req without caching)

As you can see, the latency increases and is around 11ms for a request if it sees a cache miss.

Construct an experiment to show your fault tolerance does the work as you expect. You should start your system with no failures and introduce a crash fault and show that the fault is detected and the other replica can mask this fault. Also be sure to show the process of recovery and resynchronization.

faultTolerance_3.py file achieves this experiment. It stops a server(in this case catalog-1 running on your local 8082 port), and checks if the subsequent requests are being successfully served by the alive replica(catalog-2). It performs some buy requests on the first time and stores the stock value after the last buy request. It then waits for the catalog-1 to spawn and checks if it is alive. If alive, it sends out 10 lookup requests and gathers the stock value of the item 1. The average stock values of these 10 lookup requests should match the stock value from the last buy request. If this happens, you know that the server(catalog-1) is alive and in sync with the replica.

As you can see from the screenshot above, when the server gets killed, the load balancer spawns up the catalog server again. The catalog server registers to the load balancer using the register_catalog request. Once it is done, the server issues a resync request to its replica to get synchronized.

The code for this can be found in the tests folder and faultTorlerance_3.py file. You can run after **sh run.sh** on local to see if the databases sync up after a failure.

To test the fault tolerance on aws, you can choose any server from the serverKiller.sh file and comment all the ssh commands for the other servers and execute it to kill a server running.