1 AWS Deployment:

1.1 Steps Taken on First Runtime:

1. Followed steps according to lablet 5 to get credentials and .pem file.
2. Start m5a.large instance:
   aws ec2 run-instances –image-id ami-0d73480446600f555 –instance-type m5a.large –key-name vockey > instance.json
3. Get public DNS and public IP address of instance:
   aws ec2 describe-instances –instance-id i-07bb30c815e72cfe8
   (”PublicDnsName”: ”ec2-52-73-0-179.compute-1.amazonaws.com”, ”PublicIpAddress”: ”52.73.0.179”,)
4. Changed permission of labsuer.pem file:
   chmod 400 labuser.pem
5. Authorized port 56893 since this is the port that the front end is hosted on, and needs to be able to be connected to from the outside:
   aws ec2 authorize-security-group-ingress –group-name default –protocol tcp –port 56893 –cidr 0.0.0.0/0
6. SSH into aws instance:
   ssh -i labsuser.pem ubuntu@ec2-52-73-0-179.compute-1.amazonaws.com
7. Update and upgrade ubuntu:
   sudo apt update && sudo apt upgrade
8. Install python 3.11 and pip:
   sudo add-apt-repository ppa:deadsnakes/ppa
   sudo apt install python3.11
   sudo apt install python3.11 python3.11-distutils
   curl -sS https://bootstrap.pypa.io/get-pip.py — python3.11
9. Install grpcio and protobuf packages:
   python3.11 -m pip install grpcio
   python3.11 -m pip install protobuf
10. SCP files from local to aws server:
    scp -i labsuser.pem -r ”C:\Users\thewi\Desktop\CS677\lab3\src” ubuntu@ec2-52-73-0-179.compute-1.amazonaws.com:./

1.2 Steps Taken on Testing:

- Start the catalog service:
  1. SSH into aws instance:
     ssh -i labsuser.pem ubuntu@ec2-52-73-0-179.compute-1.amazonaws.com
  2. cd src/catalog-service
  3. python3.11 catalogServer.py
- Start the order service replicas:
  1. SSH into aws instance:
     ssh -i labsuser.pem ubuntu@ec2-52-73-0-179.compute-1.amazonaws.com
  2. cd src/order-service
  3. python3.11 orderServer.py 56891 3 172.31.16.170 56893 1
  4. SSH into aws instance:
     ssh -i labsuser.pem ubuntu@ec2-52-73-0-179.compute-1.amazonaws.com

5. cd src/order-service
6. python3.11 orderServer.py 56890 2 172.31.16.170 56893 1
7. SSH into aws instance:
   ssh -i labsuser.pem ubuntu@ec2-52-73-0-179.compute-1.amazonaws.com
8. cd src/order-service
9. python3.11 orderServer.py 56889 1 172.31.16.170 56893 1

- Start the front-end service:

1. SSH into aws instance:
   ssh -i labsuser.pem ubuntu@ec2-52-73-0-179.compute-1.amazonaws.com
2. cd src/front-end
3. python3.11 frontend.py 3 172.31.16.170:56891 2 172.31.16.170:56890 1 172.31.16.170:56889 1

2 Evaluation and Performance Measurement

2.1 General Observations:

- In general, from table 1 and 2, we can see that as the value of p increases, the latency of catalog lookup requests, and order trade requests increase. This is as expected, since as the number of trade requests increases, the overall system will be taxed more as a whole, since while catalog lookup requests simply call the lookup function of the catalog service, order trade requests require a order trade request (to record the transaction and ensure its viability), which calls a catalog lookup request (to ensure the stock is valid), and a catalog trade request (to update stock information in the catalog database). As such, as the number of trades increases, the catalog server becomes even more saturated which leads to a higher latency. Interestingly, we can see that order query lookup requests also increase in latency as p increases. This could be due to the fact that order query requests are serviced by the order service, and as the number of trades increase, the order service will be more saturated, which means that the order query requests may have to wait for threads to be available in order to be serviced.

2.2 Effects of Caching:

- The effects of caching can be seen from table 1 and 2. We can see that on average, caching saves roughly 20 milliseconds on catalog lookup requests ($\sim$15% difference), and roughly 30 milliseconds on order trade requests ($\sim$19% difference). Trivially we can see why caching would save time on catalog lookup requests, as it results in less calls to the catalog service itself which reduces latency. The reason why order trade request latency decreases with caching may not be as obvious. The reason for the decrease in latency is because caching reduces the number of calls to the catalog service, which means that it may not get as saturated as easily. Because the catalog service is not as saturated, the catalog requests that are necessary for each order trade request can be serviced sooner, which causes a lower latency for order trade requests as a whole.

2.3 Fault Tolerance Observations:

- In our experiments, we tested the four following scenarios:

  1. Leader crashes after execution begins.
  2. Follower crashes after execution begins.
  3. Leader crashes and rejoins after execution begins.
  4. Follower crashes and rejoins after execution begins.

- Tests were conducted with 100 catalog lookup requests were sent. The p value was set as $p = 0.5$ and caching was enabled.

- From fig 3 and fig 5, we can see that on leader death and follower death, without any rejoining, the latency of requests are in line with what we would typically expect, or only slightly higher. As such, we have determined that the crashing of the leader replica or a follower replica are transparent to the clients. In this case, the remaining replicas have consistent database files, as shown in the test cases. The only overhead of a leader crash is the time it would take for the front end to attempt to ping the leader replica, and then the time it would take to execute the leader election algorithm (Which is fast, since we do not attempt to ping the ID corresponding to the crashed leader again, so we never have to wait for a timeout.). The only overhead of a follower crash is the time it would take for the leader replica to attempt to ping the follower, and then the time it would take to remove the follower from the leaders list of followers. Trivially these are both executed quite fast, and as such leader and follower crashes are relatively transparent to the client.

- From fig 4 and fig 6, we can see that on leader death and follower death, with rejoining, the latency of order trade requests significantly increases. This is the case, because on a rejoin, the rejoining replica must synchronize with the leader replica, which is done through rpc streaming. As long as a rejoining replica is streaming information with the leader replica, the number of threads that the leader replica can use is limited. (i.e. synchronizing causes the leader replica to be saturated) As such, this causes a bottleneck in the throughput of regular order trade requests that are being sent to the leader replica, which in turn causes the latency to increase. Therefore, when a leader or follower crashes and rejoins, the clients will be able to notice the failure and as such is not transparent to the clients. All replica database files end up the same, as shown in the test cases.

- In conclusion, through implementation techniques, on leader crash leader election is handled quite fast, and as such is quite transparent to the clients. On follower crash, checking follower death by the leader replica is handled quite fast, and as such is quite transparent to the clients. On rejoin, for both a prior follower or leader replica, the synchronization process saturates the current leader, and as such results in a higher latency for the regular order trade requests, which is therefore not transparent to the clients.

	$p = 0.0$	$p = 0.2$	$p = 0.4$	$p = 0.6$	$p = 0.8$	$p = 1.0$	Average Latency
Catalog Lookup Request	94.97	116.17	110.12	120.25	130.78	112.3	113.92
Order Trade Request		112.38	132.47	136.67	138.28	108.9	125.38
Order Query Request		97.2	94	93.5	112	94	98.14

Table 1: Average Latency of Requests with Caching. Five clients were connected to the stock server and average latency was measured for each type of request. Time is measured in milliseconds. For each trial, 50 catalog lookup requests were sent. The number of order trade requests are conditioned on the given p value relative to the number of catalog lookup requests.

	$p = 0.0$	$p = 0.2$	$p = 0.4$	$p = 0.6$	$p = 0.8$	$p = 1.0$	Average Latency
Catalog Lookup Request	134.8	114.5	134.0129	132.177	152.701	131.923	133.353
Order Trade Request		145.24	147.492	162.257	175.9625	151.177	156.426
Order Query Request		95.58	132.333	153.125	141.158	112.678	126.975

Table 2: Average Latency of Requests without Caching. Five clients were connected to the stock server and average latency was measured for each type of request. Time is measured in milliseconds. For each trial, 50 catalog lookup requests were sent. The number of order trade requests are conditioned on the given p value relative to the number of catalog lookup requests.

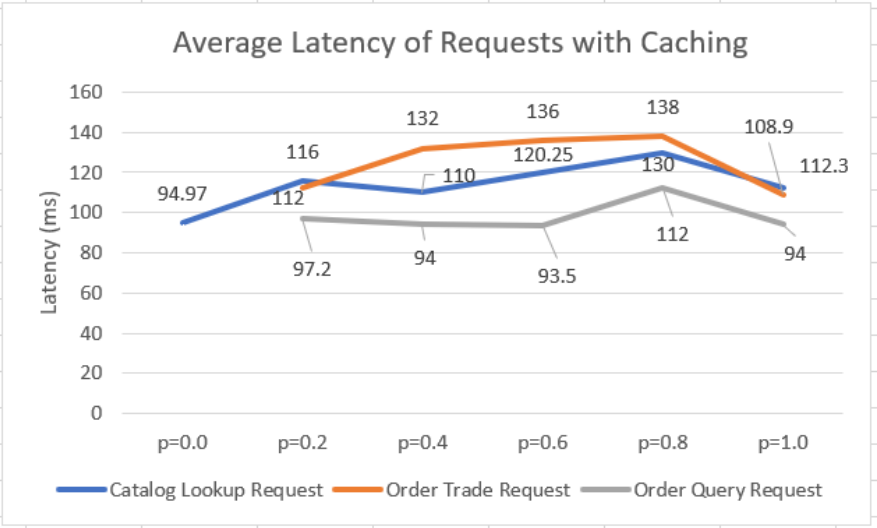

Figure 1: Average Latency of Requests for Stock Server with Caching. The probability that trade requests followed look up requests were iterated by 20%.

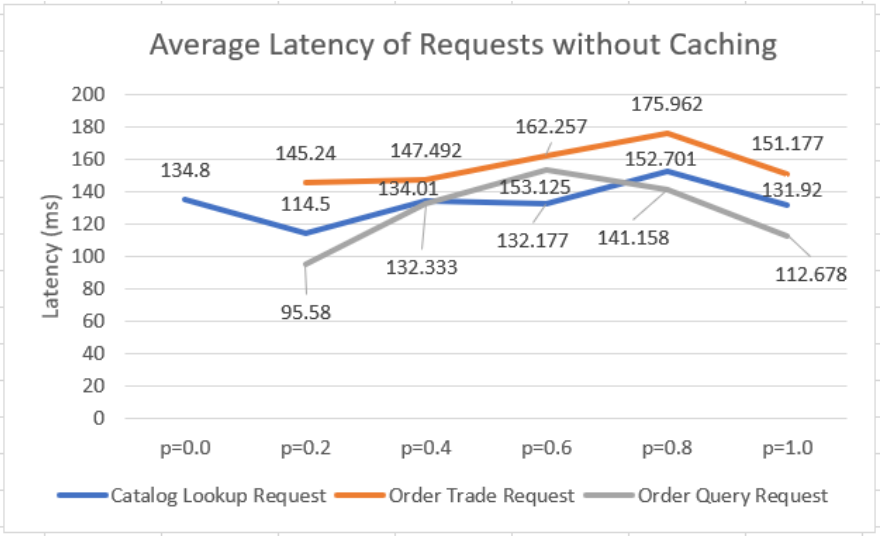

Figure 2: Average Latency of Requests for Stock Server without Caching. The probability that trade requests followed look up requests were iterated by 20%. Results are shown with caching not enabled.

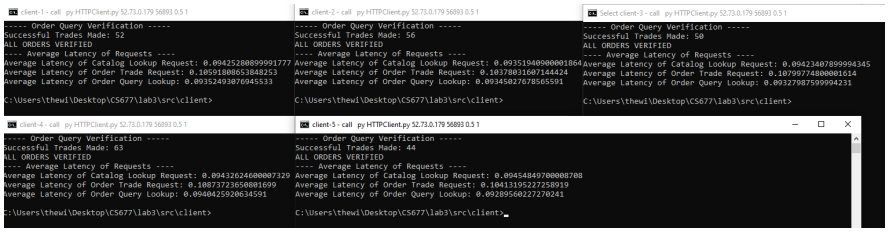

Figure 3: Latency of 5 Clients to Server. Caching was enabled with a p value of 0.5. Leader was randomly crashed during execution.

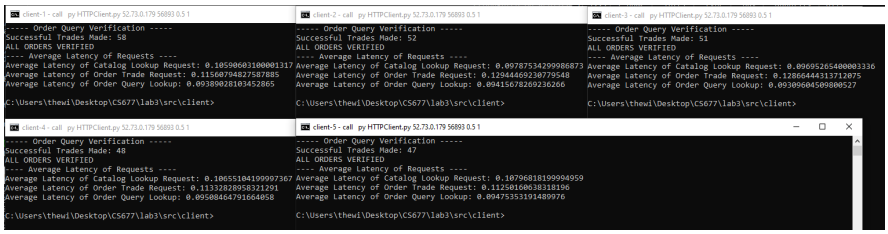

Figure 4: Latency of 5 Clients to Server. Caching was enabled with a p value of 0.5. Leader was randomly crashed during execution and subsequently restarted in order to rejoin.

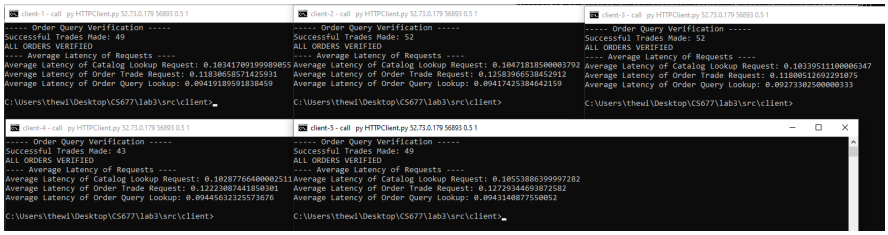

Figure 5: Latency of 5 Clients to Server. Caching was enabled with a p value of 0.5. A follower was randomly crashed during execution.

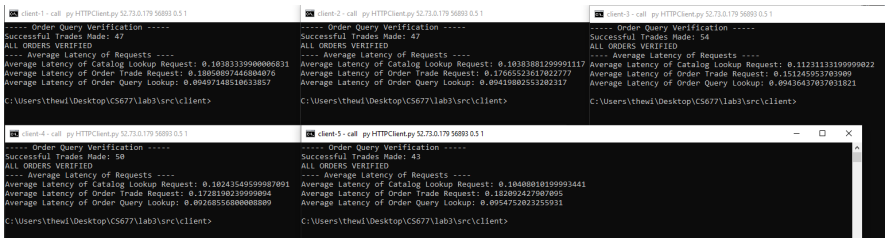

Figure 6: Latency of 5 Clients to Server. Caching was enabled with a p value of 0.5. A follower was randomly crashed during execution and subsequently restarted in order to rejoin.

3 Figures for Result Posterity:

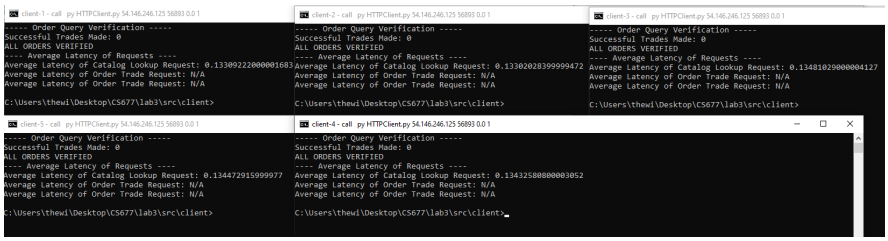

Figure 7: Latency of 5 Clients to Server. No caching was enabled with a p value of 0.0.

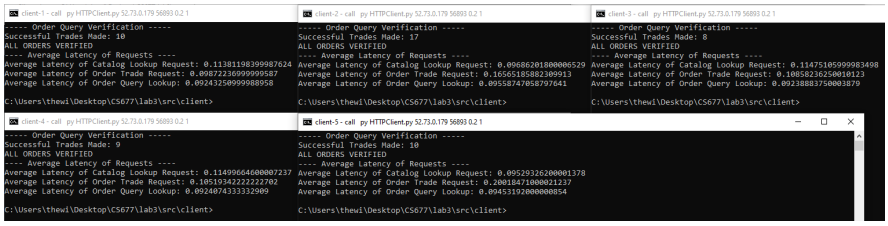

Figure 8: Latency of 5 Clients to Server. No caching was enabled with a p value of 0.2.

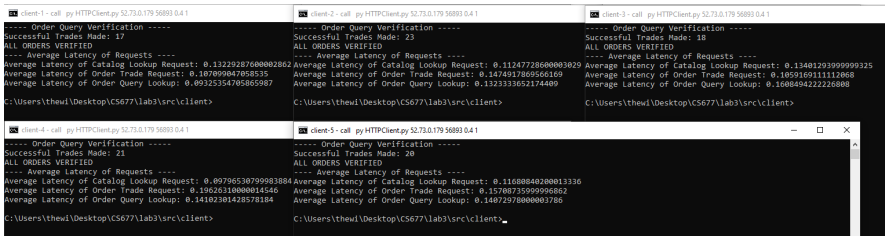

Figure 9: Latency of 5 Clients to Server. No caching was enabled with a p value of 0.4.

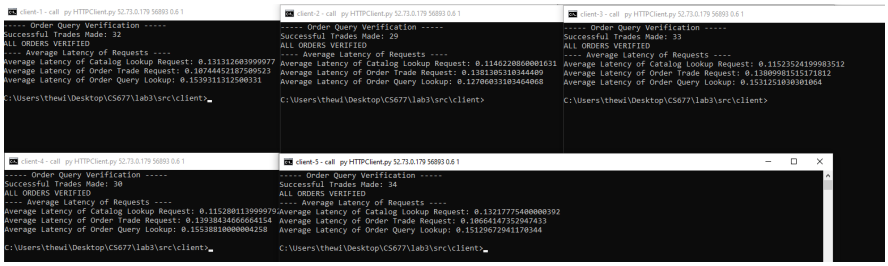

Figure 10: Latency of 5 Clients to Server. No caching was enabled with a p value of 0.6.

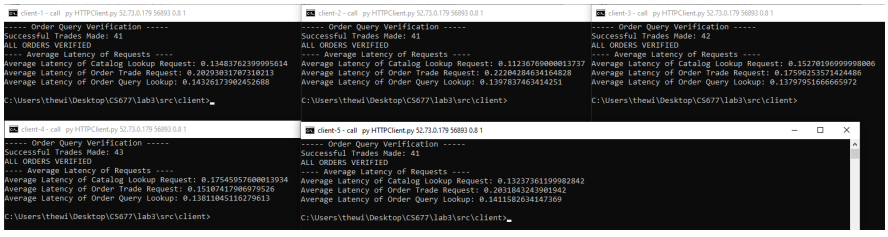

Figure 11: Latency of 5 Clients to Server. No caching was enabled with a p value of 0.8.

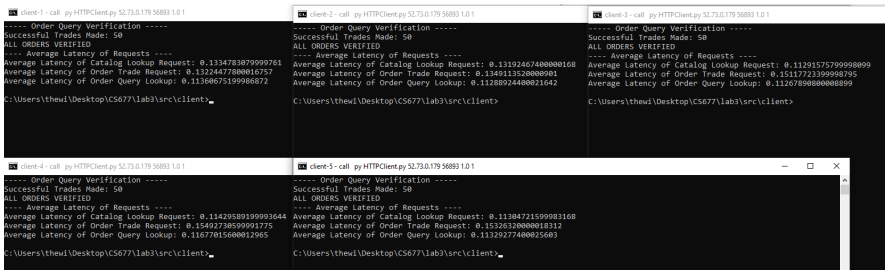

Figure 12: Latency of 5 Clients to Server. No caching was enabled with a p value of 1.0.

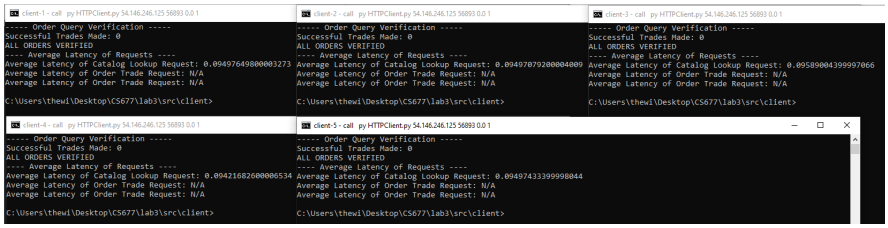

Figure 13: Latency of 5 Clients to Server. Caching was enabled with a p value of 0.0.

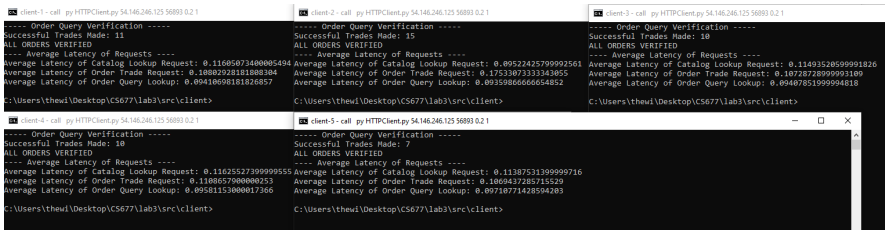

Figure 14: Latency of 5 Clients to Server. Caching was enabled with a p value of 0.2.

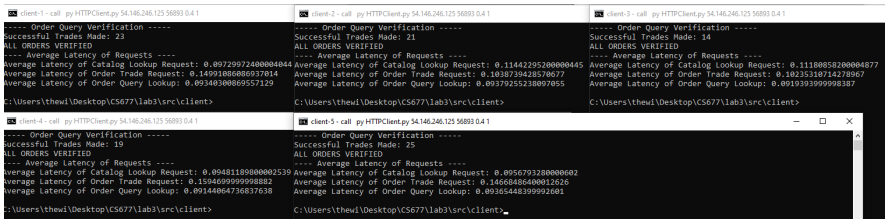

Figure 15: Latency of 5 Clients to Server. Caching was enabled with a p value of 0.4.

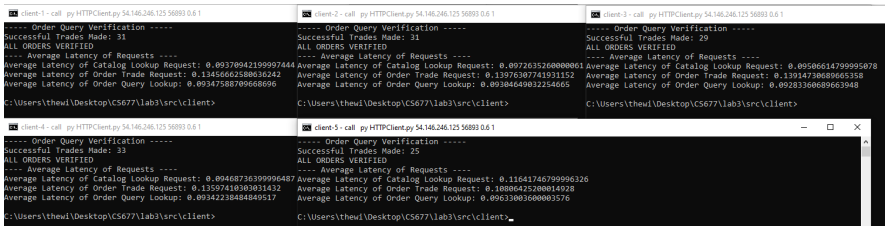

Figure 16: Latency of 5 Clients to Server. Caching was enabled with a p value of 0.6.

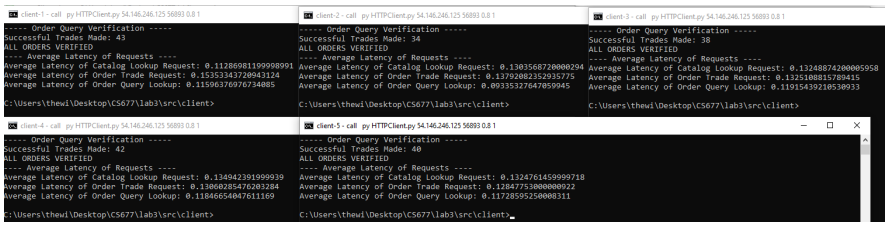

Figure 17: Latency of 5 Clients to Server. Caching was enabled with a p value of 0.8.

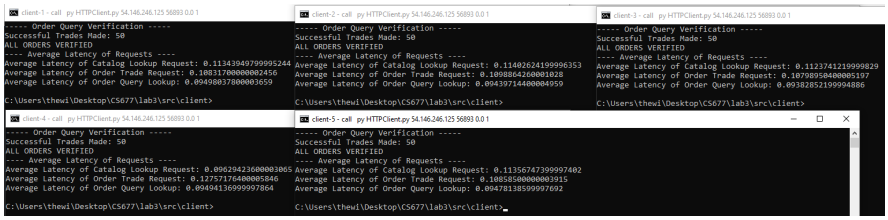

Figure 18: Latency of 5 Clients to Server. Caching was enabled with a p value of 1.0.