**Design Document**

I expanded upon the multi-tier toy store I made in lab2 to include additional functionalities, namely caching, replication, and fault tolerance. To avoid redundancy, I will mainly discuss additional functionalities implemented for lab3, but I attached my design doc from lab2 for reference in the github repo. High level, the server side still has 3 components – frontend, catalog, and order services. The client communicates with the frontend service via REST api, and the frontend, catalog, order services communicate via sockets with each other. I deployed my system on aws where I tested its performance.

The front-end service now uses an in-memory cache for query requests. The cache’s default setting is set to be on, but can be turned off via optional command line argument -c 0. I chose to use a hashmap to store the results of the queries, and also used an additional hashmap to store the state of invalidation flags. Query requests add or update existing entries in the cache and also resets the invalidation flags to false. Successful buys and restocking set the invalidation flag to true. The cache is used for query requests for products where there is an existing entry and invalidation flag is set to false. Otherwise, the frontend will forward to catalog service to perform the query request. Additionally, the front-end service is also updated to perform leader election since the order service has multiple replicas. For leader election, it pings each replica in descending order starting with highest ID. If there is a response, it sets that node as the leader and notifies all replicas. Leader election is performed upon startup of frontend service and also if it experiences unresponsiveness from the leader node or a broken pipe error (crashing mid query). Since the different tiers on the server side communicate via sockets, I used a different socket for each function (i.e. main function for processing buys and queries, restocking, leader election, data propagation). This is to ensure that the messages will always function as expected and not interfere with other parts of the program.

The catalog service now is expanded to have 10 items in its inventory. Upon starting and every 10 seconds, it will check the inventory for any out of stock items, and if any are found, it restocks them. It sends an invalidation notification to the frontend upon restocking an item.

The order service is now replicated to help ensure fault tolerance. There are 3 replicas that are exactly the same with the exception that they each have a unique ID (1,2,3). Otherwise, the code is exactly the same and they have their own directories and their own data files. Upon starting up, each replica will reach out to the other replicas to sync up data. If a difference in order length is detected, the side with the more updated version will send the latest entries that that the outdated version is missing. This ensures that any replica that is online will always be up to date with the latest data with respect to any other online replica. Only the leader, which is selected via leader election initiated by the frontend, handles POST and GET order requests. Upon a successful order, it propagates the new entry to the follower nodes. This guarantees consistency as the follower nodes will maintain the same data as the leader. If the leader fails, the frontend will detect it, initiate leader election, and the next highest ID will become the leader and pick up right where the crashed node left off. This means that from the client side, unless all 3 replicas fail, there will be no detected difference in behavior.

The client file now also calls GET order for each successful buy it made. Each client still makes on default, but changeable via command line argument, 50 queries and potential buys in a session. The GET order calls are all made together at the end of the session, and the client does an automated check to ensure that the GET order responses from the server match those in its local buy history.

The system was deployed on aws with an m5a.large instance. I ran the server side (catalog, order replicas, frontend) on aws, and ran the clients from my local machine. The public IP address of the ec2 instance is passed as an optional parameter (local host is default) for the client.py programs.

I tested my system via various unit tests along with manual tests recorded with screenshots. These are found in the src/Test Cases directory. The test.py file still contains the original tests from lab2 to ensure that existing functionality is still working, but now includes various additional tests for the new functionalities. For instance, it has tests to make sure that GET order/<order\_number> works for normal, out of bounds ID, and invalid input cases. It has a test to check that restock is working. Additionally, there are manual tests for caching, replica data syncing/data propagation, and fault tolerance where crashes are simulated and the behavior of order replicas, frontend, and client are working as intended. For the simulated crash, I set an optional parameter -f 1 to trigger that replica to crash after 5 successful buy requests. This allowed the node to crash in the middle of the client’s sequence of orders, and verify that the client remains unaffected, next highest replica resumes control, and that the final data files are as if no crash took place. If the node comes back online, it will sync data and be back on the same page with the updated version.