**Design Document**

I implemented the toy store as a multitier system with three microservices in python. High-level, it follows the specifications in a straightforward manner. The front-end service communicates with clients via HTTP rest based API’s where it receives GET and POST orders. It forwards these requests to the catalog and order services. The catalog and order services are separate and each contain their own persistent data files. Catalog service can read and write to the toy store catalog, and order service communicates with catalog service regarding buy requests, saving successful orders in its own data log. I chose to have all three services communicate via simple sockets. They all have their own folders which contain their respective docker files. There is one docker-compose.yml file in the src folder that ties these three build files together I passed an optional command line argument for the three services -d where d=1 (default) specifies that the server is running through docker containers and otherwise it is run on the elnux3 server.

The front-end service takes an optional parameter -z (default 1) which specifies how many clients to wait for before starting threads. This is to ensure consistent measurement of latency. For the server itself, I leveraged http.server package with the ThreadingHTTPServer class to allow it to be multi-threaded (thread-per-request model). The server takes as an argument a httpHandler class which processes the GET and POST requests it receives from clients over the HTTP connection. Then as explained in the intro, it passes the GET request to catalog service to run query method and passes the POST request to the order service which will also work with catalog service to process buy request. It responds to the client with the results if the calls were valid or the relevant error messages if something unexpected occurred.

The catalog service uses a thread pool of max size 20 threads to handle client requests from the front-end and order services. I first designed the interface exposing two methods, Query() and Buy() which the front-end and order services call respectively via sockets. I saved the toy store catalog in the form of JSON in a /data/catalog.txt file which contains the names, prices, and quantity of the toys. Query method reads from the catalog and Buy requests will decrement the catalog if the item is in-stock. I used read and write locks for query() and buy() respectively to guarantee mutual exclusion since the service is multi-threaded.

The order service also has many of the same elements as the catalog service. It also has its own thread pool of size 20 to handle requests from the front-end. It also has its own persistent /data/orders.txt which contain the order log in JSON format. It uses a write lock to guarantee mutual exclusion of this file when its threads access that file. It receives POST requests from the front-end which it forwards to the catalog service to perform buy() method if valid. If buy order succeeded, it writes to the order.txt and passes the order number to the front-end.

I implemented the client file in its own directory as well. If the servers are running on elnux3, the clients can be run on different machines and also with multiple clients at the same time. If the servers are being run through docker containers, since elnux3 doesn’t currently support docker at the time of the project, I ran the clients on the same machine as the docker containers, connecting over local host with port forwarding. The client file takes as optional parameters -p (default 0.7) and -n (default 50). The p parameter is the probability that the client will perform each POST request if its query request returns in-stock. This adjustable parameter is to allow the user to vary the proportion of POST requests in the session. The n parameter is the number of iterations of GET and potential POST requests. I also time the GET and POST requests starting from the second iteration to guarantee consistency as the first iteration involves some sync up time for all clients to connect. The client returns the average latency of GET and POST requests at the end. As mentioned already, the client file interfaces with front-end and it uses the requests module to communicate with the front-end HTTP server. It uses the same connection over the entire session.

As mentioned already, the optional parameter -d allows the servers to either be run on elnux3 or as dockerized containers. For dockerized containers, each service has its own container. The docker-compose.yml file stitches them together, so that they can connect to each other using the aliases from the docker-compose file. They can be run separately too in which case they accept an environmental variable with specified container IP’s. To allow for persistent storage I mount the data files as volumes for the directories containing catalog.txt and orders.txt. Each service has its own docker build file in its own directory.

I tested my system via various unit tests in the /src/Test Cases/tests.py file. For these test cases I tested that the GET and POST client calls performed as expected with both normal inputs and inputs that would generate exceptions. For example, for POST, I ran a test that queried the catalog to get initial quantity, ran two successive POST requests, and ran another query to get the final quantity. This verified that the catalog.txt was decrementing properly and persistently with each Buy() method and that the order ID’s were returned properly and were persistent as well. I also did visual tests when I ran the servers both on elnux3 and via docker. Print statements helped illustrate that the microservices and the clients were being processed as planned, and I also checked the results over multiple docker sessions to make sure the results were persistent.