CSC301 Writeup

Group Members:

Marcelo Alexander Ponce
Jai Dey
Navroop Chandwani
Yash Dave
Bennet Ashbey Dela Rosa

Group github: https://github.com/yashdave727/csc301_a2
(See a2architecture.pdf on the github for most up to date performance metrics)

A1 Architecture

Here is the architecture we started out with from A1 (provided by Jai)

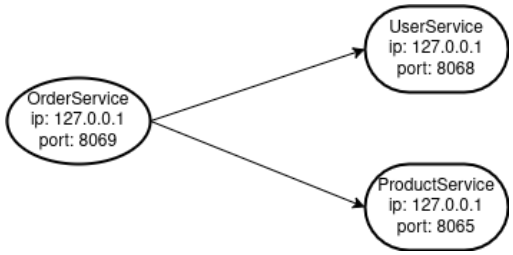

Figure 1: (not shown: both user and product services have their own "database" consisting of a text file which it reads/writes on)

Using modified versions of sender.py (included in the utility/ directory), the following tests were ran on this base architecture: (for all tests, we set N = 10,000)
(Note: We didn't run all tests because we figured this would be too slow anyways)

Name	Description	Result (time total, req's per second) or Didn't run if we didn't run it
- create_N_users.py	Create N many users	About 737s, about 13.6 requests per second
- create_N_products.py	Create N many products	About 740s, about 13.5 requests per second
- place_N_orders.py	Place N many orders	Didn't run

- update_N_users.py	Update N many users	1073.6s, about 9.3 requests per second
- update_N_products.py	Update N many products	1101.5s, about 9.1 requests per second
- get_N_users.py	Get N many users	502.4s, about 20 requests per second
- get_N_products.py	Get N many products	509s, about 19.6 requests per second
- get_N_order_history.py	Get N many order history	Didn't run
- stress_test.py	Randomly get a user, product, order history, or place a random valid order, runs multiple processes	Didn't run
- stress_test.c	Same as stress_test.py, but also sends random valid update commands to user, product, is POSIX multithreaded	Didn't run

Optimizations

We figured that having the OrderService as the entrypoint would be a bottleneck for the entire system. Thus, we decided that the newly implemented ISCS should be the entrypoint for the system. At this point in our implementation, iscs.py simply redirected requests based on the endpoint to one of UserService, ProductService or OrderService.

File I/O is incredibly slow, with search algorithms through the file having an O(N) runtime since it was not organized in any meaningful way. Thus, in this step we introduced the dockerized database. The following relations were created in the dockerized database (with appropriate attributes):

- users
- orders
- products

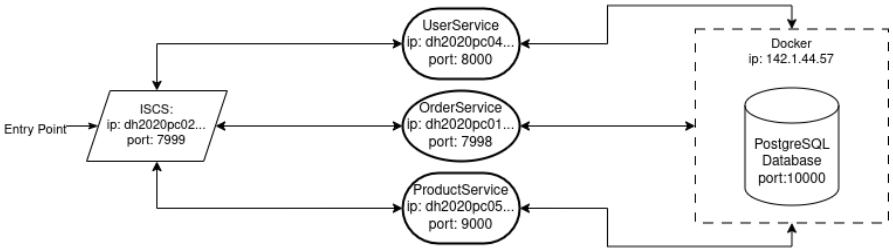

Figure 2: Updated architecture with the ISCS

The same N tests were performed and the following was recorded
(Similar to before, for all tests, we set N = 10,000):

Name	Description	Result (time total, req's per second) or Didn't run if we didn't run it
- create_N_users.py	Create N many users	433s, 23.0 reqs per second
- create_N_products.py	Create N many products	435s, 23.0 reqs per second
- place_N_orders.py	Place N many orders	too long, we cut it at 960s, measured 4.5 reqs per second
- get_N_users.py	Get N many users	453s, 22.0 reqs per second
- get_N_products.py	Get N many products	440s, 22.5 reqs per second
- get_N_order_history.py	Get N many order history	443s, 22.6 reqs per second
- stress_test.py	Randomly get a user, product, order history, or place a random valid order, runs multiple processes	Ran for 120s, 104 reqs per second
- stress_test.c	Same as stress_test.py, but also sends random valid update commands to user, product, is POSIX multithreaded	Ran for 120s, 606 reqs per second

NOTE: We approximately doubled the reqs per second from the a1 starting architecture However, this may not be an entirely accurate speedup, it might be more (The starting architecture was ran locally on a laptop that is potentially faster than the lab machines).

And based on these findings, we figured if we wanted to achieve 1000+ reqs per second, we would want to have multiple user and product services. This leads to the following architecture:

To be able to handle these changes, iscs.py was upgraded to redirect incoming http requests to services in a Round Robin manner.

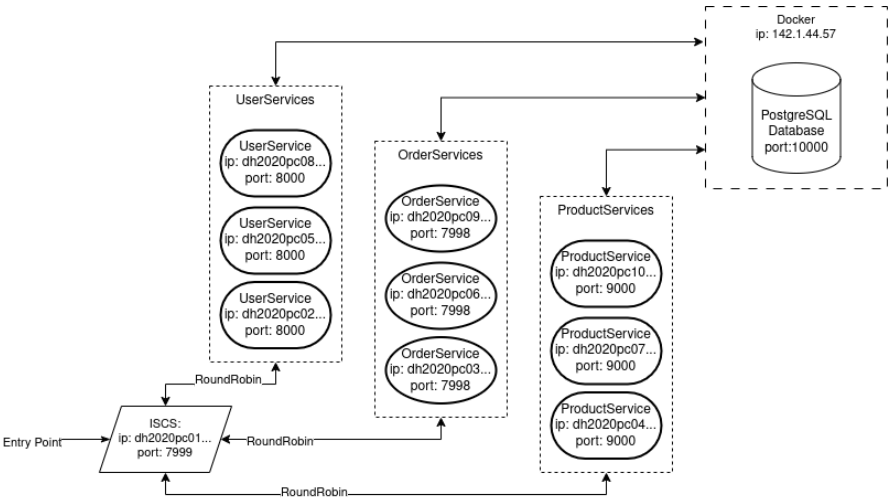

Figure 3: The architecture with multiple (3) user, product and order services to achieve as many requests per second as possible. (This is the architecture we plan to demo)

In our implementation, we decided to use 3 order services, and 3 product services.
Tested on the same tests, we got the following results:

Name	Description	Result (time total, req's per second) or Didn't run if we didn't run it
- create_N_users.py	Create N many users	
- create_N_products.py	Create N many products	
- place_N_orders.py	Place N many orders	
- get_N_users.py	Get N many users	

- get_N_products.py	Get N many products	
- get_N_order_history.py	Get N many order history	
- stress_test.py	Randomly get a user, product, order history, or place a random valid order	

Future Optimizations - Redis Cache

For further implementation, Redis cache is a performance enhancement that can be implemented to a PostgreSQL database within a Docker container. These potential optimizations can be:

- Faster data retrieval due to less database querying
    - Redis accesses data faster than PostgreSQL due to being an in-memory data storage. Thus, allowing for a much faster retrieval of data and an overall better and improved performance for the services.

- Less workload on the PostgreSQL database
    - Since PostgreSQL can be slow when many read and write (POST and GET requests) operations are done, we can use Redis to serve the frequently or "most requested" data in the database. Thus, allowing the users/clients of the services to access the cache instead of querying the database each time.

Future Optimizations - Database Pooling

Database Pooling is also another optimization we plan to implement in time for the demo. These potential optimizations include:

- Reuse of established database connections
    - All of the structures associated with a database connection take a non zero amount of time to create, and using database pooling allows each service's threads to reuse established database connections, which allows the services to save time by eliminating unnecessary reconnections.