Design Doucment

Lab 3 Caching, Replication and Fault Tolerance

Jurgen Yu Jeff Mao

May 5, 2022

1 Introduction

This Lab contains implementations for a toy store server in two tier architecture, a client to make HTTP RESTful requests to the server and utilities for assisting operation. This project extends Lab2 but with a huge overhaul to the architecture. All three services is re-implemented with Spring Boot 2 framework, bring in enhancement to system functionality, modularity and robustness.

2 Server

The serer has three parts implemented for three microservices. They are, respectively, the frontend service for exposing HTTP RESTful API to network and handle client requests, the catalog service and order service feature-specific tasks and updating persistant data stores on its disk space. In a two tier architecture, the frontend service is regarded as upper tier/layer, and are directly accessed by users; The catalog and order serives is regarded as lower layer that can only be invoked by server as needed, however, we do provide access to individual services for easy of debugging. The spring boot framework will provide routing and hosting for our APIs. An incoming request is routed to corresponding controller endpoints and our code takes over from there.
All services now run in isolated docker containers each consists of a spring boot application. They expose ports assgined as confifured for accessing. Communication amoung services is done through `RESTFul API` interface.
It is also noted that due to the inherent nature of spring framework, the applciation is granted multi-threading capability for all services. In case of an potential conflicts in critical data, we used Map structure for all sensitive data and wrapped it in `Collections.synchronizedMap` to maintain operational safety while allowing threading.

3 Front-End

Frontend service is implemented under `frontend` and it itself is a complete spring boot aplication that can work independently.The fronend service uses thread-per-request scheme in its operation. The end points include `GET /products /<product_name>` for product querying, `POST /orders` to order products, `GET /orders/<order_number>` to query order information and `GET /invalidate/<product_name>` for catalog services to request for invalidating certain product in the frontend server's cache. At start up, front end service would retrive service hosts from environment variables for catalog and three order replicas. It would also quest each replca and select the order service with biggest id as the leader. The front end service only communicates with leader replca before the leader is found unresponsive which triggers another leader election in frontend service.
The frontend service would cache product data in its memory for future queries untill a invalidate request on `GET /invalidate/<product_name>` is made from catalog to remove the cached information to cooperate with stock change.

3.1 Catalog

Catalog service is implemented in `catalog` and it itself is a complete spring boot application. The service exposes `GET /query /<product_name>` for querying products, `POST /orders` for stock update according to ordering requests, and `OPTIONS /reset` for internal use to reset all stocks and local persistent data. The catalog service will restock products every 10 seconds. An invalidation request is send to frontend service to remove product from its cache when a pruchase or restocking causes quantity change in products.
The catalog service saves stock information to local disk when it is modified, during restart, the catalog service will read from local files to recover previous state.

3.2 Order

Order service is implemented in `order` and it itself is a complete spring boot application. The service exposes `POST /orders` for purchasing a product, `GET /order /<order_number>` for retriving order information `OPTIONS /reset` for internal use to reset local persistent data, `GET /ping` for checking status of order service, `POST /sync` for leader order service to request this service to update with new orders and `GET /getUnsyncedOrderDataList /<next_order_number>` for other order services to request on this service during recovery to get untracked orders during its crash-recovery period.
The order service saves order informations to local files and reads it during restart to recover previous orders. After which it also sends request to live order replicas to retrive any missing order between its last successful order and current state.
Once a order service process an order request, it pushed sync requests to other replicas to make sure all replcias has the same state.

4 Docker

All services are contained in its own container at exectution, each module can communicate through docker network bridge. Images are built from a bare-bone linux to keep at minimum complexity. We utilize `docker-compose` to assemble five services into a running app. Build scripts are provided in the project folder.