677 Lab 3: Asterix and Double Trouble -- Replication, Caching, and Fault Tolerance

| Lavanika Srinivasaraghavan (lsrinivasara@umass.edu)

1 Introduction and Architecture

In this lab I implemented a two-tier distributed microservices-based application for stock bazaar. The application is composed of microservices built using Spring Boot. The application uses LRU based caching, leader-follower replication, fault tolerance and state synchronization. There are 3 microservices:

1. Front-end service: receives client requests via HTTP-based REST APIs (`GET /stocks/<name>, POST /orders, GET /orders/<order_number>, etc`) and forwards them to the appropriate back-end service(Catalog/Order).
2. Catalog Service: stores stock information (name, price, quantity, volume) and supports updates.
3. Order Service: interacts with Catalog to update stock information, handles buy/sell requests, supports replication (3 replicas, different replica.id) and ensures consistency across all 3.

All services support concurrent request handling through Spring Boot's default embedded Apache Tomcat server which has an inbuilt thread pool.

2 Front-end Service

- REST APIs exposed to clients:
  - `GET /stocks/<stock_name>` to return stock info with caching.
  - `POST /orders` to place a buy or sell order, forwards to leader
  - .`GET /orders/<order_id>` to retrieve order details through leader.
- Spring Boot (@Restcontroller), RestTemplate for internal REST calls, `LinkedHashMap` for in-memory LRU Cache (configurable via `cache.size` in `application.yml`)
- Maintains information on the leader node of the Order service and updates the leader in case of a failure.
- Thread safe using synchronized(cache) blocks

3 Catalog Service

- Loads stock data through stocks.csv using @PostConstruct
- Stores stock volume in ConcurrentHashMap<String, Integer> which is also thread safe
- Catalog is updated when orders are processed
- On startup, the Catalog loads data from a CSV (`catalog.csv`) via Java IO `FileReader, BufferedReader`.
- For buy/sell changes, it modifies the in-memory map and may overwrite the CSV file.
- After each trade, sends a POST to frontend's invalidation API endpoint

4 Order Service (replicated)

- Each replica runs with unique replica.id and port (9091, 9092, 9093), highest ID is the leader
- API endpoints:
  - `GET /orders/<order_id>`
  - `POST /orders`
  - `POST /orders/replicate` (to followers, each stores order using its local `ConcurrentHashMap`)
- Ping to check who current leader is, if it is a follower, endpoint hit is: /orders/sync
- Concurrency: ConcurrentHashMap<Integer, Map<String, Object>> to store orders
- If the leader is killed/crashed, the request is re-attempted without disturbing the client.
- Synchronization of orders happens when a crashed replica comes back online, which is validated by querying /orders/{id} on all replicas, they all return the same content.
- Fault transparency is maintained because clients do not see any errors even when leaders fail in the middle of a request.

5 Deployment and Testing

- All microservices are runnable using `mvn spring-boot:run` with separate profiles
- test/ folder contains REST API tests for:
  - Stock querying
  - Cache replacement
  - Order creation
  - Leader failure simulation
- Logs are stored in .log files configured through `logback-spring.xml`

6 References

1. Oracle Java Documentation for `com.sun.net.httpserver` https://docs.oracle.com/javase/8/docs/jre/api/net/httpserver/spec/
2. Spring Initalizr: https://start.spring.io/
3. Spring Boot documentation https://docs.spring.io/spring-boot/index.html
4. Logback documentation https://logback.qos.ch/documentation.html
5. LinkedHashMap LRU Cache design https://www.youtube.com/watch?v=ITQd43aSWOo&ab_channel=RishiSrivastava