SYSTEM DESIGN DOCUMENT

for

Trading Application

COMPSCI 677 Lab 3

Prepared by: Mudit Chaudhary (32607978)
Girish Baviskar (33976648)

Contents

1 Introduction

1.1 Purpose

This document specifies the system design for the microservices-based trading application.

1.2 Intended Audience and Reading Suggestions

This document is intended for developers, testers, documentation writers, and the grading staff for the project. This document does not specify the technical details, evaluation results, and user manual.

1.3 Tech Stack

We use the following tech stack for the trading application:

- Python
- Docker
- Flask web framework

1.4 Deployment

We deploy the application on AWS EC2 `t2.micro`.

2 System Design

In our system, the microservices communicate using HTTP requests using REST-based architecture. We develop microservices using Flask web framework to provide RESTful APIs. All our microservices use thread-per-request threadpool concurrency model. We use a threadpool of size 1000 for each service using Gunicorn.

2.1 Frontend Service

The Frontend Service communicates with the catalog service and order service using thread-per-request. The Frontend also deals with determining the validity of requests. The Frontend also provides a cache for stock lookup request with a functionality to invalidate caches. The cache invalidation interface is provided as an API call which requires a secret key for authentication. Secret key protects cache invalidation from an external client.

2.1.1 Leader election

Frontend service also performs leader election of the order service replicas. It first sends a ping request to all the possible replicas. From the replicas that answer the ping requests, it chooses the replica with the highest id. The leader's id is then pushed to all the replicas. Leader election is performed whenever an order or order lookup requests fails.

2.2 Catalog Service

The Catalog Service provides the functionality to lookup and update the stock database. However, it can only update a particular stock's quantity and trading volume and does not perform additional checks if an update is valid as that responsibility is handled by the order service. In our implementation we use a thread-per-request model for lookup requests. Catalog service also sends cache invalidation request to the frontend whenever a stock is updated in the catalog.

2.2.1 Database

We provide a simple persistent database using Pysos. We make it thread safe by using reader-writer locks. The reader locks can be accessed by multiple requests simultaneously but writer locks blocks the resources from concurrent threads except the one that

has acquired the writer lock. The reader-writer locks improve the throughput of our application.

2.3 Order Service

Order Service provides the functionality of determining the validity of an order and carrying out the trade by communicating with the Catalog Service. It first performs lookup of the stock through the catalog service and then determines the validity of the order. Once the validity is determined, it send the update request to the catalog service to increase/decrease the stock quantity based on the trade type. Order service also maintains a database to log the transactions, which is then used to handle order lookup requests. To handle the order requests, we use synchronization locks and TransactionHandler. The locks ensures catalog consistency for order requests. We also implement fault tolerance and replication of order service, which is described in subsequent sections.

2.3.1 TransactionHandler

TransactionHandler performs thread-safe transactions. The transactions/order are performed in a thread-safe manner i.e., the process of lookup and update are under a lock. Only one thread can lookup and update at a time. This avoids any inconsistency in the catalog database due to multiple threads.

2.3.2 Replication and Fault Tolerance

Order services can be replicated and a leader is chosen by the frontend to handle the requests. We perform automated replication using Docker compose's scale functionality. All replicas maintain their own transaction databases and keep them synchronized with the leader. To maintain consistency between the replica databases, we use the following triggers to perform synchronization with the leader replica:

- **Startup**: As soon as a replica starts, it queries frontend for a leader. If a leader is found, it performs database synchronization with the leader by sending it a synchronization request. This trigger maintains database consistency in case the replica crashes and restarts.
- **Leader announcement**: Synchronization with the new leader is performed for all the replicas, whenever frontend announces a new leader.
- **Order request**: Whenever leader completes an order request, it sends a synchronization request to the replicas.

3 API specifications

3.1 Frontend Service

3.1.1 Lookup request

- **Method Access:** HTTP
- **URL:** GET http://hostname:port/stocks/stockname
- **Response Codes:** 200 (Success), 400 (Failure)
- **Sample Success:** {"data":{"name":"GameStart","price": 15.99,"quantity":100}}
- **Sample Failure:** {"error":{"code": 404,"message":"stock not found"}}

3.1.2 Order request

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/orders
- **Response Codes:** 200 (Success), 400 (Failure)
- **Content-Type**: application/json
- **Sample Request Body**: {"name":"GameStart","type":"sell","quantity":100}
- **Sample Success:** {"data":{"transaction_number":10}}
- **Sample Failure:** {"error":{"code": 404,"message":"transaction unsuccessful"}}

3.1.3 Order lookup request

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/orders/order_id
- **Response Codes:** 200 (Success), 400 (Failure)
- **Sample Success:** {"data":{"transactionID":10, "name":stockName, "type":orderType, "quantity": orderQuantity}}
- **Sample Failure:** {"error":{"code": 404,"message":"order not found"}}

3.1.4 Cache invalidation request

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/invalidate/secret_key/stock_name
- **Response Codes:** 200 (Success), 511 (Failure)
- **Sample Success:** {`"status": "success"`}
- **Sample Failure:** {`"status": "fail"`}

3.1.5 Query leader request

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/secret_key/whoisleader
- **Response Codes:** 200 (Success), 511 (Failure)
- **Sample Success:** `leader_address`
- **Sample Failure:** `"None"`

3.2 Catalog Service

3.2.1 Lookup request

- **Method Access:** HTTP
- **URL:** GET http://hostname:port/lookup/stockname
- **Response Codes:** 200 (Success), 400 (Failure)
- **Sample Success:** {`"data":{"name":"GameStart","price": 15.99,"quantity":100}`}
- **Sample Failure:** {`"error":{"code": 404,"message":"stock not found"}`}

3.2.2 Update request

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/update
- **Response Codes:** 200 (Success), 400 (Failure)
- **Content-Type**: application/json
- **Sample Request Body**: {`"name":"GameStart","quantity":-100`}

3.2.3 Reset catalog request

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/clearDB/secret_key
- **Response Codes:** 200 (Success), 511 (Failure)
- **Sample Success:** {`"status": "cleared"`}
- **Sample Failure:** {`"status": "not authenticated"`}

3.3 Order Service

3.3.1 Order request

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/order
- **Response Codes:** 200 (Success), 400 (Failure)
- **Content-Type**: application/json
- **Sample Request Body**: {`"name":"GameStart","type":"sell","quantity":100`}
- **Sample Success:** {`"data":{"transaction_number":10}`}
- **Sample Failure:** {`"error":{"code": 404,"message":"transaction unsuccessful"}`}

3.3.2 Order lookup request

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/orders/order_id
- **Response Codes:** 200 (Success), 400 (Failure)
- **Sample Success:** {`"data":{"transactionID":10, "name":stockName, "type":orderType, "quantity": orderQuantity}`}
- **Sample Failure:** {`"error":{"code": 404,"message":"order not found"}`}

3.3.3 Ping request

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/ping
- **Response Codes:** 200 (Success)
- **Sample Success:** `replica_id`
- **Sample Failure:** `N/A`

3.3.4 Perform sync with leader request (Sent by replica to the leader)

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/syncSendMulti/transaction_id_after_which_logs_are_missing
- **Response Codes:** 200 (Success)
- **Sample Success:** `[{"transactionID":10, "name":stockName, "type":orderType, "quantity": orderQuantity}, {"transactionID":11, ...}...}]`
- **Sample Failure:** `N/A`

3.3.5 Leader initiated sync request (Sent by leader to replicas)

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/syncReceive
- **Response Codes:** 200 (Success)
- **Content-Type**: application/json
- **Sample Request Body**: `{"transactionID":10, "name":stockName, "type":orderType, "quantity": orderQuantity}`
- **Sample Success:** `{"status": "true"}`
- **Sample Failure:** `N/A`

3.3.6 Announce leader request

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/announceLeader/leader_id
- **Response Codes:** 200 (Success), 500 (Fail)
- **Sample Success:** `"success"`
- **Sample Failure:** `"fail"`

3.3.7 Get leader request

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/getLeader
- **Response Codes:** 200 (Success)
- **Sample Success:** `leader address`
- **Sample Failure:** `N/A`

3.3.8 Reset database request

- **Method Access:** HTTP
- **URL:** POST http://hostname:port/clearDB/secret_key
- **Response Codes:** 200 (Success), 511 (Failure)
- **Sample Success:** {`"status": "cleared"`}
- **Sample Failure:** {`"status": "not authenticated"`}