Design Document

By: Aishwarya Malgonde, Sahil Jindal
Course: CS 677 | Professor Prashant Shenoy

Lab 2: Asterix and the Microservice Stock Bazaar

March 25, 2023

System Overview

The goal of this assignment is to create a set of microservices that can be used to build a simple trading platform. Specifically, we have created a Frontend service that interacts with clients, an Order service that handles order processing, and a Catalog service that manages stock catalog data.

In the second part, we have containerized all the microservices individually and defined a docker-compose file to start and shutdown all these microservices with a single command. Finally, we have performed the experiments for latency comparisons.

Architecture

We are using a microservices architecture for this assignment, by implementing the following three services, where each microservice communicates with the other -

1. The **Frontend** service will handle client interactions and make calls to the order and Catalog services as needed
2. The **Catalog service** will receive Lookup requests from the Frontend and it will retrieve and respond with stock data. It will store and manage the stock catalog data
3. The **Order service** will receive Trade requests from the Frontend process. It will call the Catalog service for Lookup and Update methods for executing the Trade request. It will store and manage the order log data
4. The **client** will make GET and POST requests to the Frontend service

Technologies Used

1. We have used **Python3** as the primary programming language for this assignment.
2. The Frontend REST API service is implemented using the Python **`http.server`** library

3. The backend interface for the catalog and order microservices is built using **PyRO5** (Python Remote Objects).
4. The client is implemented using the Python **`requests`** library
5. For data storage and management, we employed a mix of **pandas** dataframe objects for in-memory data storage and **CSV files** for persistent storage.

Design Considerations

1. Why PyRO?
    a. We decided to use PYRO because it is a remote procedure call (RPC) framework for Python
    b. It is easier to implement as it allows objects residing in one Python process to be accessed remotely by another Python process over a network, without the programmer having to explicitly write network communication code.
    c. PyRO also enables distributed computing and inter-process communication.
2. Code structure
    a. The .yml file is kept in the root directory
    b. The three docker files of every microservice is placed inside their respective folders
    c. Configurations of all microservices are clearly defined in setup.py file in all the modules inside src directory
3. Functionality
    a. The Order service processes the Trade request from the clients. It communicates with the Catalog service for Lookup requests and based on the response Updates the database.
    b. The Catalog service does Lookup and Update on the database. It does not have any order logic and just Updates the database as requested by the Order service.
    c. Front end service receives the requests from the clients and formulates and sends the response from the Order service.
    d. We have added docker support for this application. The catalog file and order log file is made such that it is persisted after container removal.

4. Ease of Running
    a. All the microservices can be started from the root directory itself with/without docker. To run the server in docker or without docker, we made sure that the server code need not be changed in any way.
    b. Each module inside the src folder has a setup file. It makes it easier for the user to configure values like host and port number before starting the system.
5. Logging
    a. We have created a custom logger in all setup.py which logs the time, script name, thread name and the message
    b. Using these logs inn the Frontend and backend services makes it is easy to see concurrency and thread-per-session behavior.

Part 1

Microservice 1 - Frontend Service

The Frontend service has two HTTP-based REST APIs and it handles client interactions with HTTP/1.1 protocol. It is built using the inbuilt `http.server` library. Following are the features of the Frontend service -

1. **Threaded** HTTP server is implemented using the `ThreadingHTTPServer` module of `http.server`. This module uses the `ThreadingMixIn` class from the `socketserver` library. The threadpool size is dynamic and it creates a new thread for each new client.
2. It checks for proper URL for all the GET and POST requests
3. If a client makes a GET request, then the Frontend service will make an RPC call over the PyRO5 proxy to the Catalog service to process the Lookup method
4. If a client makes a POST request, then the Frontend service will make an RPC call over the PyRO5 proxy to the Order service to process the Trade method
5. It uses json serialization to pack responses and unpack POST request data
6. It attaches proper headers while sending back the responses as follows -
    a. Status code
    b. Content-type as application/json

   c. Connection as "keep-alive" - Keeps the connection open for a client sending multiple requests in a session
   d. Keep-alive with timeout set as 10 sec
   e. Content-Length of response object - Helps the client read the entire response over the connection and clean it for the next request
7. The server exhibits **thread-per-session** behavior as we have attached these headers to the response - Connection, Keep-alive and Content-Length. As long as a client is sending requests over a session all the Frontend processing will happen in a single thread for that particular client

REST API Endpoints

We have implemented the following two API endpoints in the Frontend:

1. GET url - /stock/<stock-name>
   a. Sample successful response

```
{
   "data": {
      "name": "GameStart",
      "price": 17,
      "quantity": 150
   }
}
```

   b. Sample error response -

```
{"error": {
            "code": 404,
            "message": Invalid request url
         }
   }
```

2. POST url - /order/: places an order
   a. Sample payload data to be sent with request

```
{
```

```
    "name": "GameStart",
    "quantity": 1,
    "type": "sell"
}
```


b. Sample successful response


```
{
  "data": {
    "transaction_number": 10
  }
}
```


c. Sample error response


```
{"error": {
            "code": 400,
            "message": Unknow trade type
          }
  }
```


Microservice 2 - Backend Catalog service

The Catalog service maintains a list of all stocks in the stock market, and their information, particularly - price, quantity and trading volume. When the Frontend service receives a Lookup request, it will forward the request to the Catalog service. When the Order service is processing a Trade request, it will make a Lookup and Update request, as and when required.
Functionalities of the Catalog service are as follows -

1. The Catalog service is threaded, with a dynamic threadpool size ranging from 4 to 40. These configurations are added before starting the daemon request loop.
2. We have set OpenBlas threading limit to 1 to make sure that it doesn't use a large amount of resources when using pandas or numpy.
3. When the service starts up, it initializes itself from the stock database disk file
4. If the disk file is not present, then it initializes with a non-zero number of stocks of each company available for sale as can be seen in the setup file

5. When it receives a Lookup request from the Frontend service, it check if the stock name is present in the in-memory database and sends a response accordingly
6. While reading the database in a Lookup request, a read lock is acquired to ensure synchronization amongst multiple threads. It ensures that no write happens when read lock is acquired
7. When the Catalog service receives a Update request from the Order service, it can increment or decrement the number of stocks available for sale, depending on the type of Trade request
8. Here, it updates both, the in-memory stock database and the persistent CSV file, and the CSV file is written out immediately
9. While updating the databases in a Update request, a write lock is acquired to ensure synchronization amongst multiple threads. Only one thread can own this write lock and it ensures that no read happens when the write lock is acquired

PyRO Interface

Connection with the Catalog service can be established using fields -
Service name (string) - backend.catalog
Name server host (string) - 0.0.0.0 (without docker) / catalog (if using docker)
Name server port (integer) - 9090

The Catalog service exposes the following two methods -

Lookup method name - **lookup**

1. Input Parameters
    a. stock_name (string)
2. Output Parameters (tuple)
    a. code (integer) - can be any of the HTTP status codes listed in its section
    b. response (dict if successful, otherwise string)
        i. Sample response when code is 200

```
{
   "data": {
```

```
        "name": "GameStart",
        "price": 17,
        "quantity": 150
    }
}
```


ii. Sample response when code is not 200
Bad request. <stock-name> stock not found in db

Update method name - **update**

1. Input Parameters
    a. stock_name (string)
    b. change_in_quantity (integer)
2. Output Parameters (tuple)
    a. code (integer) - can be any of the HTTP status codes listed in its section
    b. message (string)

Microservice 3 - Backend Order service

When the front-end service receives a Trade request, it will forward the request to the Order service. The Order service handles the trade processing and manages the order log data. Inorder to process the Trade request, the Order service will first make a Lookup call to the Catalog service. If the Lookup response is valid, then the Order service will check if the stock could be bought or sold. If the stock can be traded, then the Order service makes an Update call to the Catalog service. Here, the Order service specifies the amount by which the stock should be incremented or decremented.

Functionalities of the Order service are as follows -

1. The Order service is threaded, with a dynamic threadpool size ranging from 4 to 40. These configurations are added before starting the daemon request loop
2. We have set OpenBlas threading limit to 1 to make sure that it doesn't use a large amount of resources when using pandas or numpy.

3. After it received a Trade request from the Frontend service, it checks if the trade information sent is valid
4. A buy trade request succeeds only if the remaining quantity of the stock is greater than the requested quantity, and the quantity is decremented
5. A sell trade request simply increases the quantity of the stock
6. In both, buy and sell trade request, the trading volume is incremented to keep track of it
7. If the trade processing is successful, the Order service increments the transaction number and returns it to the front-end service
8. This transaction number is unique, starting from 0, and if the order log is already present, then checks for the last transaction number and is Updated accordingly (This is handled in order_service/setup.py).
9. The Order service maintains the order log in a persistent manner with the help of a simple CSV file on disk as the persistent storage for the database.
10. While incrementing the transaction number and adding trade information to the order log, write lock is acquired to ensure synchronization amongst multiple threads

PyRO Interface

Connection with the Order service can be established using fields -
Service name (string) - backend.order
Name server host (string) - 0.0.0.0 (without docker) / catalog (if using docker)
Name server port (integer) - 9090

The Order service exposes the following one methods -

Trade method name - **trade**

1. Input Parameter
    a. data (dict)
        i. Sample data format with required keys

```
{
   "name": "GameStart",
   "quantity": 1,
   "type": "sell"
```

}
name (string) - stock name
quantity (integer) - trading quantity
type (string - “buy”/”sell”) - trading type

2. Output Parameters (tuple)
    a. code (integer) - can be any of the HTTP status codes listed in its section
    b. response (dict if successful, otherwise string)
        i. Sample response when code is 200

```
{
   "data": {
              "transaction_number": int(curr_txn_num)
            }
  }
```

        ii. Sample response when code is not 200
        Insufficient volume for <stock-name> stock

Data Models

We are using csv files with the following data models for this assignment:

Stock Data (**Location** - "src/back-end/catalog_service/data/stock_db.csv")
name (string)
price (float)
quantity (integer)
trading_volume (integer)

Order Log (**Location** - "src/back-end/order_service/data/order_log.csv")
transaction_number (integer)
stock_name (string)
order_type (integer)
trade_quantity (integer)

When the Catalog and Order service starts up, it initializes itself from the database disk file. If the files are not present, then they are initialized in their respective “setup.py" present in the folders of respective service, which is imported by the Catalog and Order microservices on startup.

HTTP Status Codes

The following status codes are incorporated in the front-end and backend service, which all the scenarios listed below -

1. 200 - Success
    a. GET request returns stock data from the database successfully
    b. POST request executes a trade order successfully
2. 404 - Resource not found
    a. Client makes GET/POST request on an incorrect URL
3. 400 - Bad request from client side
    a. Incorrect stock name is sent on GET/POST request by the client
    b. Invalid trade quantity is sent on POST request by the client
    c. Unknow trade type (other than “buy” or “sell”) is sent on POST request by the client
4. 500 - Internal server error
    a. Any processing / execution error on the server side
    b. Cannot complete buy transaction due to insufficient stock volume

Client

The client creates a session and makes sequential GET and POST requests with randomized stock names and payload data. If the GET request is successful with positive quantity, then the client makes the POST request with a probability (POST_probability) as defined in its setup.py file. The HTTP connection protocol is set to HTTP/1.1 for all requests. A single client will send all the requests in a single session so as to have a thread-per-session behavior on the server side.

Conclusion

By building these microservices, we have created a simple trading platform that can be used to Lookup stock catalog data, process trade orders, and interact with clients in a user-friendly way. In general, we have adopted good programming practices such as incorporating logging and utilizing code modularity.

Part 2: Containerize Your Application

In this part, we built the docker containers for each of the microservices. These containers are running independently and need to communicate with each other in order to process one request. The client sends a request to the Frontend container. The Frontend service container then communicates with the two backend service containers (catalog_service and order_service) whenever there is a GET and POST request.

Docker container helps this stock bazaar distributed system to run on any machine regardless of the OS and the version of Python it is running. This ensures that the code is running consistently without any changes when switching the machines.

While using the docker containers we ensured that each of the containers are mapped to a unique port on the machine. The two backend services, catalog and Order services, have their own pyro5 nameservers running independently from each other.

We have mounted respective data directories of Catalog and Order services so that the files can be persisted after the containers are removed.

In all the microservices setup.py file we are using a USING_DOCKER variable which helps set the proper host names for communication between them. This variable is set as True in the run.sh file, which is the entry point file for Docker. When starting the microservices in Docker, it will take the corresponding host name.

Part 3: Testing and Performance Evaluation

Test Cases

To ensure the functionality and reliability of our system, we have developed a set of test cases that cover the three microservices individually. Here, when we are testing the Frontend service,

that essentially tests the entire system, since the Frontend is making calls to the backend microservices. We have taken a comprehensive approach by testing the services with both correct and incorrect parameters to verify their error handling capabilities. We have placed our testing scripts and test output file in the src/tests directory. By thoroughly testing our system, we can identify and address any issues before they impact our clients, and ensure that our microservices are functioning as expected.

We have used Python's **unittest** library to test individual microservices. Following are some of the test scenarios covered -

Catalog Service test cases:

1. For lookup function, we checked the cases when the stock name sent is present in the database
2. For update function, we tested the update quantity and stockname. If the update quantity is not an integer the application was able to handle that and throw an error message. Similarly if the stockname was not present, the application returned a error code of 400.

Order Service:

1. Tested the trade where the client sells the stock and all information is valid
2. Tested the trade where the client buys the stock and all information is valid
3. Tested the trade where the client buys a stock larger than the amount present in the database.
4. Tested trade request where stock name is not valid
5. Trade with negative quantity.
6. Tested unknown trade type.
7. Tested trade where the payload doesn't have all the information and structure

Frontend Service

1. We tested both get and post requests.
2. We tested get and post when the url is incorrect.
3. Tested the case where the stockname is incorrect in both get and post requests.
4. For post requests we tested when the quantity was negative or invalid.

5. For post requests we also tested when there was a buy trade request but the quantity was not present in the database.

Performance measurement

We performed all the latency comparisons experiments on our host machines to have a same level field across multiple scenarios. Following are the experiments performed for comparing latency -

1. Vanilla Server vs Docker Server
2. Lookup request vs Trade request

We were also able to run our server (without virtualization) on Edlab, and ran multiple clients from our host machine. We have captured and presented these results in the Evaluation doc.

We logged the timestamps in a log file and latency was calculated using them. We measured latency from the client side as -

client latency = network latency + server latency

Where network latency accounts for sending and receiving messages, and server latency is the request processing time.

We were running into issues while trying to run Docker on Edlab. Due to this we were unable to perform comparison using two host machines. Please refer to the Output document for the error screenshots.

References

Apart from the support given by piazza and the TAs we referred the following documents:

Part 1 -

1. https://docs.python.org/3/library/http.server.html
2. https://requests.readthedocs.io/en/latest/
3. https://pyro5.readthedocs.io/en/latest/api/api.html
4. https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Part 2 -

1. https://docs.docker.com/engine/reference/builder/
2. https://docs.docker.com/compose/compose-file/
3. https://docs.docker.com/storage/volumes/