Part 1 Design Document

part 1 implements online stock trading server using 2-tiered architecture (front end and back end)

Design decisions:

FRONT-END SERVER

1. **Threading**

A thread pool with sufficient maximum workers is used in the front-end server in order to handle all the incoming clients. A thread per session model is used so that each client gets its own thread until the client wishes to close the connection.

2. **Sockets**

Python sockets are used for communication between the client and the front-end sever.

API Endpoints:

I. **/stocks/<stock_name>**
Request header: {}
Request body: {}
Success responses: 200
Format: {'data': {'name': 'BoarCo', 'price': 30.25, 'quantity': 1}}
Error responses: 404
Format: {'error': {'code': 404, 'message': 'stock not found'}}

II. **/orders**
Request header: {Content-Type: application/json\r\nContent-Length: {len(request_body)}}
Request body: {"name": stock_name, "quantity": quantity, "type": “buy/sell”}
Success responses: 200
Format: {'data': {'transaction_number': 275}}
Error response: 404
Format: {'error': {'code': 404, 'message': 'stock not found'}}
Error reponse: 422
Format: {'error': {'code': 422, 'message': 'max trading volume exceeded'}}

Interfaces:

A simple http request handler class is implemented to handle the incoming requests from the clients. The **decode_request** method receives the socket data from the client, decodes and parses it to determine the type of request [GET or POST] and forwards it to the do_GET or do_POST methods defined in the same class.

I. **GET /stocks/<stock_name>**
Forwards the request to catalog server
Http client library is used to forward the request to the back-end services. The do_GET method opens a http connection with catalog server in order to forward the GET request from the client and the same connection is used to get the response back.

II. **POST /orders**
Forwards the request to the orders server
The do_POST method also uses http connection to send and receive the http request and response from the orders service.

Http response

The http responses received from the backend services are now used to build a response in the required format using the **_build_http_response** helper function. The result to be sent to the client is built in form of a dictionary and encoded in json format with appropriate http header and body. This response adhering the http protocol is sent back to the client using sockets.

Http response codes and status used :

200 – OK for successful lookup or trading
404 – Stock not found if requested stock is not present in the database
422 – Maximum trading volume exceeded if client wants to buy/sell more than the limit set

CLIENT

Data structure:

Python lists are used for the client to randomly select the stock name, type of trade (buy/sell) and the quantity. The lists also contain invalid stock names and quantities which exceeds the maximum trading volume to test the error conditions.

Design

1. The client is designed to first send a lookup request for a stock name, if the stock has quantity greater than zero then a trade request for the same stock is sent with a probability >0.5
2. Multiple clients can also be opened simultaneously to test concurrent requests.
3. Python sockets are used to establish connection with the front-end server.

SYNCHRONIZATION (Read_Write_Lock.py)

A simple read write lock class is implemented to ensure that the catalog and orders database can be used by multiple threads without causing any issues.

1. The **acquire_read()** method is used to acquire the read lock. It first acquires the general lock to update the internal state of the ReadWriteLock object, incrementing the _read_count variable to indicate that a thread is about to start reading from the resource. If this is the first thread to start reading (_read_count == 1), it acquires the write lock to prevent any threads from writing to the resource. Finally, it acquires the read lock to allow the thread to start reading from the resource.
2. The **release_read()** method is used to release the read lock. It first releases the read lock itself and then acquires the general lock to update the internal state of the ReadWriteLock object, decrementing the _read_count variable to indicate that the thread has finished reading from the resource. If this was the last thread to finish reading (_read_count == 0), it releases the write lock to allow any waiting threads to start writing to the resource.
3. The **acquire_write()** method is used to acquire the write lock. It simply acquires the write lock itself to prevent any other threads from reading from or writing to the resource.
4. The **release_write()** method is used to release the write lock. It simply releases the write lock itself to allow other threads to start accessing the resource.

CATALOG SERVER

Python socketserver library is used to start the server on port 8001 and wait for incoming requests.

On startup, the catalog server will open and read the database file and put it into a dictionary for further usage.

Database: A csv file is used to store all the stock data. The file has stock name, quantity, max trading volume and price of each stock.

API Endpoints

I. **/Lookup_csv/<stock_name>**
Request header: {}
Request body: {}
Success response: 200
Format: {'name': 'BoarCo', 'price': 30.25, 'quantity': 1}
Error response: 404
Format: ‘stock not found’

II. **/Update_csv**
Request header: {Content-Type: application/json\r\n}
Request body: {}
Success response:200
Format: ‘OK’

Interfaces

I. **GET /Lookup_csv/<stock_name>**
The do_GET method of MyHandler handles incoming GET requests and retrieves stock information based on the stock name provided in the endpoint. It first extracts the stock name from the endpoint, calls the Lookup_csv function to retrieve the stock information, and sends the response back to the client as a serialized json object with the appropriate status code. If the stock is found status code is sent as 200, else it is sent as 404 when stock is not present in the DB. http client python library is used to send the response back to the front-end server or to the catalog.

II. **POST /Update_csv**
The do_POST method of MyHandler class handles incoming POST requests and updates the stock quantity for the given stock name. It first reads the incoming data from the request body, deserializes it as a json object, extracts the stock name and quantity fields, and calls the Update_csv function to update the stock information in the CSV file. It then sends a success response back to the client as a serialized json object with the appropriate status code. Only orders service sends a POST request to the catalog service to update the database after a successful transaction.

III. **Lookup_csv**
The lookup_csv API is designed to fetch the stock requested (if present) from the stock_DB.csv database and load it into a dictionary. A read lock is used as the stock database file maybe accessed by multiple threads/clients. Then the method will return the dictionary with the name, price and quantity as its keys with respective fetched values from the DB. If the requested stock is not present then the API will return a string ‘stock not found’.

IV. **Update_csv** method is used to update the stock_DB.csv database file after any successful trading. Write lock is acquired in order to ensure that no changes can be made by other thread when updating the data.

ORDER SERVER

The server listens on port 8080 for incoming requests. The MyHandler class is defined, which extends the BaseHTTPRequestHandler class and implements the do_POST() method to handle incoming POST requests.

Database: A csv file is used to maintain log of all successful trading transactions. Which has transaction number, stock name that was bought or sold, type of trade and quantity.

API Endpoints:

I. **/trade_stocks**

Request header: {Content-Type: application/json\r\nContent-Length: {len(request_body)}}
Request body: {"name": stock_name, "quantity": quantity, "type": “buy/sell”}
Success responses: 200
Format: 'transaction_number': 275
Error response: 404
Format: 'stock not found'
Error reponse: 422
Format: 'max trading volume exceeded'

Interfaces:

I. **POST /trade_stocks**

The MyHandler class's do_POST() method extracts the request endpoint and calls the trade_stocks() function to handle the trade request. The response data and response code returned by trade_stocks() are sent back as a JSON-encoded message to the client.

II. **Trade_stocks()**

The trade_stocks() function first fetches the last transaction number from the database file.

The function processes the incoming POST request, opens up a connection with catalog server to get whether the requested trading stock name is present in the database or not. If the stock name is not present in the database then the function returns error code 404 saying stock not found.

If the stock is present in the database, then the next condition check is whether the quantity the client wants to buy is more or less than the max trading volume, if the quantity exceeds the trading volume an error code 422 UnProcessable entity/ max trading volume exceeded is sent back to the client.

If the stock is present and the requested quantity is in limits, then the transaction number is incremented and successful transaction is updated in orders DB, quantity is updated based on buy/sell and request is sent to catalog server to update the database with the new quantity. Finally, transaction number is sent back to the front-end server will will then be forwarded to the client.