Design document

Problem Statement

To design a distributed application using a multi-tier architecture and microservices that serves HTTP based Rest APIs.

1. To provide an HTTP GET request that can be used to perform a lookup on a stock.
2. To provide an HTTP POST request that can be used to perform a trade on a stock.

Implementation Details

Language Used: Python
Framework and libraries: Python HTTP Server, gRPC, Python HTTP Client

Application endpoints

The frontend application exposes the following HTTP endpoints:

- /hello?name=<name> - Returns a hello response with the given name.
  Sample request: GET *http://localhost:26111/hello?name=user*
  Sample response: *Hello, user!*
- /stocks?stockname=<stockname> - Returns the price and volume of the given stockname.
  Sample request: GET *http://localhost:26111/stocks?stockname=stock1*
  Sample response:
  ```
  {
    "data": {
      "name": "stock1",
      "price": 25.78,
      "quantity": 10
    }
  }
  ```
  Sample error response:
  ```
  {
    "error": {
      "code": 404,
      "message": "stock not found"
    }
  }
  ```
- /orders - Accepts a POST request with a JSON body containing the name, quantity, and trade type (BUY/SELL) of a stock to be traded.
  Sample Input Payload:
  ```
  {
    "name":"stock1",
    "quantity":10,
    "type":"BUY"
  }
  ```
  Sample request: POST *http://localhost:26111/orders*
  Sample response:
  ```
  {
    "data": {
      "transaction_number": 3
    }
  }
  ```
  Sample error response:
  ```
  {
  ```

```
  "error": {
     "code": 404,
     "message": "Trading not permitted due to insufficient volume"
  }
}
```


Check attached postman collection to send sample requests.

Proto Design

The proto contains the following messages:

- *LookupRequest*: Contains the name of the stock (string) to look up.
- *LookupResponse*: Contains the stock name (string), price (double), and volume(int).
- *UpdateRequest*: Contains the stock name(string), trade type(BUY or SELL), and quantity(int).
- *UpdateResponse*: Contains the stock name(string) and status(int).
- *TradeRequest*: Contains the stock name(string), trade type(BUY or SELL), and quantity(int).
- *TradeResponse*: Contains the stock name(string), status(int), and transaction number(int).
- *Empty*: An empty message.

The proto file also provides the interfaces for the Catalog Service and Order Service
The **CatalogService** interface has the following methods:

- The ***Lookup()*** method takes a *LookupRequest* message and returns a *LookupResponse* message. This method is to perform a lookup on a stock. The LookupResponse will have price and volume values as -1 if the stock is not present, otherwise the corresponding values.
- The ***Update()*** method takes a *UpdateRequest* message and returns a *UpdateResponse*. This method is used for performing an update on a stock depending on the trade type. The status field in the response will have 1 for success, 0 for suspended trading, and -1 otherwise.

The **OrderService** interface has the following methods:

- The ***Trade()*** method takes a *TradeInput* message and returns a *TradeResponse*. This method is used for registering an order and performing the trade. The status field in the response will be 1 for success, 0 for suspended trading, and -1 otherwise.
- The ***Save***() method takes an *Empty* message and returns an *Empty* message. This method is used for saving the contents of the in-memory database to a file. This function is not used in the code.

Data model

The *Stock* data model is used to store the details of the stock. The Stock data model has the following attributes and functions.
Attributes:

- *name* : used to store the name of the stock
- *price*: used to store the price of the stock. Pythons' Decimal module is used to handle the precision of the stock price with 2 decimal places to avoid floating point arithmetic issues.
- *volume*: used to store the current available volume

Methods:

- The ***__init__(name, price, vol)*** constructor method to initialize the stock object.
- The ***getPrice()*** method returns the price of the stock.
- The ***trade(quantity, trade_type)*** method is used to perform trading on the stock. It takes two parameters: the quantity of the stock being traded and the type of the trade, which can be either 0 for BUY and 1 for SELL. The

method only performs BUY if there is a sufficient quantity available. The method returns 1 if the trade was successful, returns 0 if the trade cannot be processed because of insufficient volume, and -1 otherwise.
- The ***to_string()*** method converts the stock object to a readable string.

Frontend-Service

The frontend service serves the lookup and order REST APIs. The frontend service is designed as a Python HTTP server using the BaseHTTPRequestHandler and HTTPServer classes. The frontend service overrides its do_GET() and do_POST() methods to handle GET and POST requests, respectively. The frontend service is multi-threaded using the ThreadingMixIn class to allow concurrent handling of multiple requests. The frontend service implements a thread-per-session model and maintains a session with client until the client exits.
The server communicates with the other microservices services using gRPC to perform the lookup and order operations. The services are defined using Google Protobuf (.proto) and generated stubs are used to communicate with the services.

The frontend calls the Lookup function of Catalog service to perform Lookup operation. The frontend calls the Trade function of Order service to place an order.

Catlog Service

The catalog service implements the *CatalogService* gRPC interfaces. The service implements the following methods:

- **Lookup**: The Lookup function is responsible for retrieving the price and volume of a given stock. The Lookup method takes the name of a stock as input and returns its current price and trading volume, if present in the catalog. If the stock is not present in the catalog, the function returns -1 for both price and volume.
- **Update**: The update function is responsible for updating the stock object after performing a trade on the stock. The Update method takes the name of a stock, quantity to trade, and type of trade (BUY or SELL) as input. If the stock is present in the catalog, the function performs the trade and returns 1 if the trade is successful and 0 if the trade is suspended. If the stock is not present in the catalog or the function fails, the function returns -1.

The catalog service uses an in-built thread pool to handle concurrent requests. The maximum number of threads is configurable and is read from the *'catalog_threadpool_size'* field in the *config.py* file at start-up.
The catalog service also maintains an in-memory database to store the stocks along with their information. The catalog service also writes to a *stockDB.txt* file to persist the stock information even when the server stops. The stock details are updated in the in-memory database and *stockDB.txt* during each update call.

Order Service

The order service implements the *OrderService* gRPC interface. The service implements the following methods:

- **Trade**: The trade method takes the name of the stock, quantity to trade, and type of trade (BUY or SELL) as input. The method forwards the request to the Catalog Service to check the availability of the stock and perform the trade request. If the trade was successful, the method generates a transaction number and returns the transaction number along with the trade status of 1. The method returns a transaction number of -1 if the trade is unsuccessful.
- **Save**: This method will dump the latest database changes to the disk. This function is not used at the moment.

The order service uses an in-built thread pool to handle concurrent requests. The maximum number of threads is configurable and is read from the *'order_threadpool_size'* field in the *config.py* file at start-up.
The order service also maintains an in-memory database to store the order log and order history. The order service also writes to a stockOrderDB.txt file to persist the order history. The order details are logged into the in-memory database and *stockOrderDB.txt* file after each transaction. The data from the in-memory database is also dumped into the stockOrderDB.txt file when the application exits.

Client

The client uses Python's ‘http.client’ library to send HTTP requests to the frontend service. Once the client opens a connection with the frontend service it can send multiple requests over that connection. The client has two functions to test different scenarios.

- The ***testNormalWorking()*** function is used to check if the application produces the expected results for both the Lookup and Order endpoints. It sends multiple Lookup and Order Requests to test all the scenarios.
- The ***testSession()*** function establishes a session with the frontend and sends multiple Lookup requests and optionally Order requests based on the probability ‘p’. The probability and the number of requests are both configurable and are included in the config.py file under field names ‘num_session_requests’ and ‘prob’ respectively.

Data Flow

1. The client establishes a session with the frontend service.
2. The client sends an HTTP request to the frontend service.
3. The frontend services parses the request to extract the path.
4. If the path is '/hello', a hello response is returned with the given name.
5. If the path is '/stocks', the frontend service sends a lookup request to the catalog service to get the price and volume of the given stock name.
6. If the path is '/orders', the frontend service sends a trade request to the order service with the name, quantity, and trade type of the stock to be traded. The order service logs the trade and forwards the request to the catalog service to complete the trade.
7. The response from the backend services is converted to a JSON response.
8. The JSON response is returned as an HTTP response to the client.
9. The connection remains open for other requests from the client.

Error handling

Since the resources such as in-memory stocks database and file databases can be accessed by multiple threads simultaneously, locks should be used to maintain synchronization and ensure that the data is consistent and accurate. Simple locks are used when accessing shared resources. try-except blocks are used to handle unknown errors and prevent the application from failing. In addition, the frontend service only accepts requests to specific paths and sends 404 error for all other requests. The Catalog and Order services handle scenarios such as invalid stock names, and performing BUY when insufficient quantities are available.

Logging

The application uses the custom logger provided in src/shared/util/logging.py to make it convenient to track down issues. The logger object is configured to log messages of all levels and output them to the console. The log messages are formatted with a timestamp, the name of the service, the name and ID of the thread that generated the message, the level of the message, and the message itself.

References

- Python HTTP server
- gRPC
- Geeksforgeeks python HTTP server
- Python HTTP Client