Microservices Design

For front-end and backend microservices, code from lab2 was reused. Modifications to the previous code were made to add replication, caching, and fault tolerance support.

- **Overview**:

  The application consists of two back-end microservices: catalog and order services, and one front-end microservice. The front-end service and the two back-end services use grpc for communication among themselves. The front-end microservice is a REST API server. Clients communicate with the front-end service using HTTP-based REST APIs.

- **Communication**:

  Communication between the front-end server and the two back-end services is handled using grpc. The proto file specifying the servicer classes, methods, errors, and message structure “service_rpc.proto” is present in the proto directory. The python files generated using this proto file are also present in the same directory.

  The communication between clients and the front-end server occurs through HTTP-based REST APIs. The front-end server only supports GET and POST requests for lookup and trade requests respectively.

- **Working:** Based on the type of request (GET/POST), the front end will call the catalog service if the request is GET to look up stock details, order number, cache or to invalidate cache, and return a data object or error object to the client. If the request is of type POST, the front end will call the order service, which will call the catalog service for lookup and update of the database if the request is valid and return an appropriate response. The front end will return a data object containing the transaction number back to the client if the trade was successful or an error object if the trade was unsuccessful.

- **Multiple Instances of Order service:**

  In our implementation, instances of order service can be started by running multiple instances of the src/order/orderService.py file. It accepts the service Id number, database file path, port number, and host through environment variables.

- **Selection of leader by the front end :**
  Once the order services start, then the front-end service is started. The ids, ports and hosts of running order services are provided to the front end through “*ORDER_ID*”, “ORDER_PORTS”, “ORDER_HOSTS” environment variables. The front end reads the ids, ports and hosts and selects the service with the highest id. Then it attempts to create a grpc channel and send a health check request. If an appropriate response is received, that service is elected as the leader. Then all the running instances of order service are notified of the selected leader by the front end. This is done using the “*setLeader”* method. When a request is received by the front end for stock trade, the front end makes a call to the leader. Update of follower services is taken care of by the leader service.

- **Syncing amongst instances of order service:**
  Once a selected leader order service receives a trade request from the front end, it proceeds to carry out *lookup* and *buy_or_sell_stock* calls to the catalog service. If the service is successful, the service sends an *update_db* call to other replicas to update their database. In the *update_db* method, the follower’s database is updated with the information it gets. The information includes the stockname, quantity, order_type, and transaction_number of the transaction sent by the leader for update. This is done using a write lock to maintain consistency.

- **Fault Tolerance:**
  When the front-end tries to make a call to orderService but doesn't get a response, a new leader is selected by the front end. This is done by selecting from the rest of the replicas, the replica with the next largest service id as the leader.
  The front end continues to forward new requests to the new leader. Once a service recovers from a crash, the front end doesn’t initiate a leader selection but keeps the current leader as the leader for the subsequent requests. In case the trade or order query request to the order service fails if the leader goes down, it initiates a leader election and selects the replica with the highest service id. It attempts to create a grpc channel and send a health check request. If an appropriate response is received, it sends the information about the current leader to all the replicas including the recovered service.
  On the recovered order service’s side, as soon as the update replica comes up, the *synchronize_database* method is called to update its database with current leader and other replicas. Inside *synchronize_database,* the *send_db_data* method is called by the recovered services, which fetches updated transactions from other running service replicas. If a replica replies with some transactions that the recovered service has missed, it updates its

database with the response it gets from *send_db_data*. This fetched data is then written to the recovered service's database. After that, the recovered service acts as a follower.

- **Cache invalidation of front-end service:** After a successful Buy/Sell request that is once the update operation of the stock_data.csv file in the "buy_or_sell" method is successful at the catalog service, the catalog service sends an HTTP request to the front-end service to invalidate its cache. It first establishes an HTTP connection with the front-end service and sends the stockname for cache invalidation. After receiving the response from the front-end service confirming the cache invalidation, it closes the connection.

- **Error Handling:** Errors are handled mostly using error fields in the responses returned by service methods. The error fields are a type of enum specified in the proto file and can have values "NO_ERROR", "INVALID_STOCKNAME", "INSUFFICIENT_QUANTITY", "INTERNAL_ERROR" and "INVALID_REQUEST", "DB_UPDATE_ERROR", "INVALID_ORDERNUMBER", depending on the type of error occurred.

```
enum ERROR_CODES{
   NO_ERROR = 0;
   INVALID_STOCKNAME = 1;
   INTERNAL_ERROR = 2;
   INSUFFICIENT_QUANTITY = 3;
   INVALID_REQUEST = 4;
   DB_UPDATE_ERROR = 5;
   INVALID_ORDERNUMBER = 6;
}
```

- **Database Files:**

  The catalog and order services interact with database files to persist data. The catalog service interacts with the "stock_data.csv" file to lookup stock information and write updated data to the database. The stock data has data stored in csv format where the first row contains stock names, second contains stock prices, third row contains the quantity available to sell and the fourth row contains the overall traded volume of that stock.

  The order service interacts with the "transaction_logs_serviceid.txt" file to store transaction logs. The logs are stored in following format :

  1176 - Stockname: GameStart Quantity: 1 Order: sell

  Here, the first field is the transaction number, followed by stockname, quantity and type of order. The service also reads the logs file at startup to get the last transaction number. Subsequent transaction numbers are generated by incrementing this transaction number. If the transaction log file is empty (such as at first startup), then the transaction number initializes to 0.

Each order service replica has a separate transaction database file named transaction_log_<service_id>.txt where it stores information about the transactions.

- **Design**:

a. **Catalog Service:** The GRPC server running the catalog service is multithreaded using the futures.ThreadPoolExecutor library. The number of threads is obtained from the environment variable “MAX_WORKER_THRESHOLD_CATALOG” with a default value of 5 and the host and port of the service are obtained from the environment variables “CATALOG_HOST” with a default value of “0.0.0.0” and “CATALOG_PORT” with a default value of 6000 respectively. The catalog service is responsible for the lookup and update of the database. The database file used here is named “stock_data.csv” to store the stock catalog’s information. The file contains attributes of the stocks such as price, quantity, and volume.
Data is read by the service as pandas dataframe. The constructor creates a read-write lock from the “readerwriterlock” library and then loads the data from the “stock_data.csv” into “data_file”.
The service has 2 methods: lookup() and buy_or_sell(). The lookup method is used for getting information about stock and the buy_or_sell method is used for updating the database according to the trade request received by the order service. The lookup method accepts a *lookupRequestMessage* as input which is defined in the proto file and its output is *lookupResponseMessage*. The *lookupRequestMessage* has one field “stockname”. The *lookupResponseMessage* has the fields error, stockname, quantity, and price. The method first acquires the read lock and then looks up the stock name in the catalog. If present, it returns the stockname, quantity, and price of the stock inside a *lookupResponseMessage* along with an error field with “NO_ERROR”. If the requested stock name is not present then the method sends a response with the error value “INVALID_STOCKNAME”. If any other error occurs, it sends a response with the error value “INTERNAL_ERROR”.
The buy_or_sell method is responsible for updating the database values as per requests received. It accepts an *orderRequestMessage* and returns *orderResponseMessage. orderRequestMessage* has three fields stockname, quantity, and type. *orderResponseMessage* has one field error. If the type field is “buy”, the method first acquires a write lock and then it decreases the available quantity of stock in the database by the order request’s quantity, and if the type is “sell” then it first acquires a write lock, and increases the value of the database’s quantity corresponding to stockname by the supplied value. It also increments the total traded volume of the stock for both types of requests. The updated values are persisted in the database file immediately after each update operation while holding

the write lock in order to avoid any inconsistency. The read-write locks are used from *readerwriterlock* library to handle consistency issues. A write lock is acquired, then an update operation is carried out for the database, and new data is persisted in the database file. After persisting the data, it sends cache invalidation request to the front-end service by using an HTTP connection.

The method also returns an error field in the *orderResponseMessage* to the order service*.* Its value depends on whether the update operation was successful, or if any other error occurred.

b. **Order Service:** The GRPC server running the order service is multithreaded using the futures.ThreadPoolExecutor library. The number of threads is obtained from the environment variable “MAX_WORKER_THRESHOLD_ORDER” with a default value of 5 and the host and port of the service are obtained from the environment variables “ORDER_HOST” with a default value of “0.0.0.0” and “ORDER_PORT” with a default value of 6001 respectively. The constructor creates a GRPC channel for communicating with the catalog service. It also creates a read-write lock from the “readerwriterlock” library. The catalog service host and port are obtained from the environment variables “CATALOG_HOST " and “CATALOG_PORT”.This is done in order to dynamically assign the host and port from environment variables during running the service. The Order service is responsible for handling the trade requests received from the front end. This service has one method “trade” which is called from the front end. Input is given as a *tradeRequestMessage,* which has three fields namely stockname, quantity, and type. It returns a *tradeResponseMessage*. *tradeResponseMessage* has two fields namely error and transaction_number. The service first validates the received quantity and type. If found invalid, it returns an error with the value “INVALID_REQUEST”. If quantity and type are found valid, then the service proceeds and does a lookup call with the catalog service to get the stock’s details. If the call is successful, further checks are done to ensure if a buy request is made then the requested quantity is available to be traded. If the quantity is not available, the “INSUFFICIENT_QUANTITY” error is returned; else, the service proceeds to call the buy_or_sell method of catalog service to update the catalog. If the call succeeds then after acquiring the write lock, a transaction number is generated and persisted along with details of the requests such as stock name, type, and quantity in the “transaction_logs.txt” file, present in the data directory. The transaction_logs text file contains all successful transactions. The service returns the transaction_number along with “NO_ERROR” to the front end for successful operation. Else if the operation is unsuccessful, an “INTERNAL_ERROR” is returned as the value of the error field of *tradeResponseMessage.* The service also has *synchronize_database, update_db, send_db_data* methods, and their usage is mentioned in the “Syncing amongst instances of order service” and “Fault Tolerance” sections. The “lookupOrder” method of the order service is used to query existing orders in the transactions log file of the order service. It takes “lookupOrderRequestMessage” request which contains the order_number field for lookup. It reads the transactions log file after acquiring a lock and checks if a transaction number in the file matches the order/transaction number of the query. If it finds any matching transaction number, it returns a response "lookupOrderResponseMessage" containing the fields: error=pb2.NO_ERROR, number=transaction number, name=stockname, type=trade_type and quantity=trade_quantity. If there's no transaction number in the order service file that matches the query order, it returns a lookupOrderResponseMessage with error=pb2.INVALID_ORDERNUMBER , indicating an invalid order number. If some other issue occurs during the execution of the method, it sends a lookupOrderResponseMessage with error=pb2.INTERNAL_ERROR.

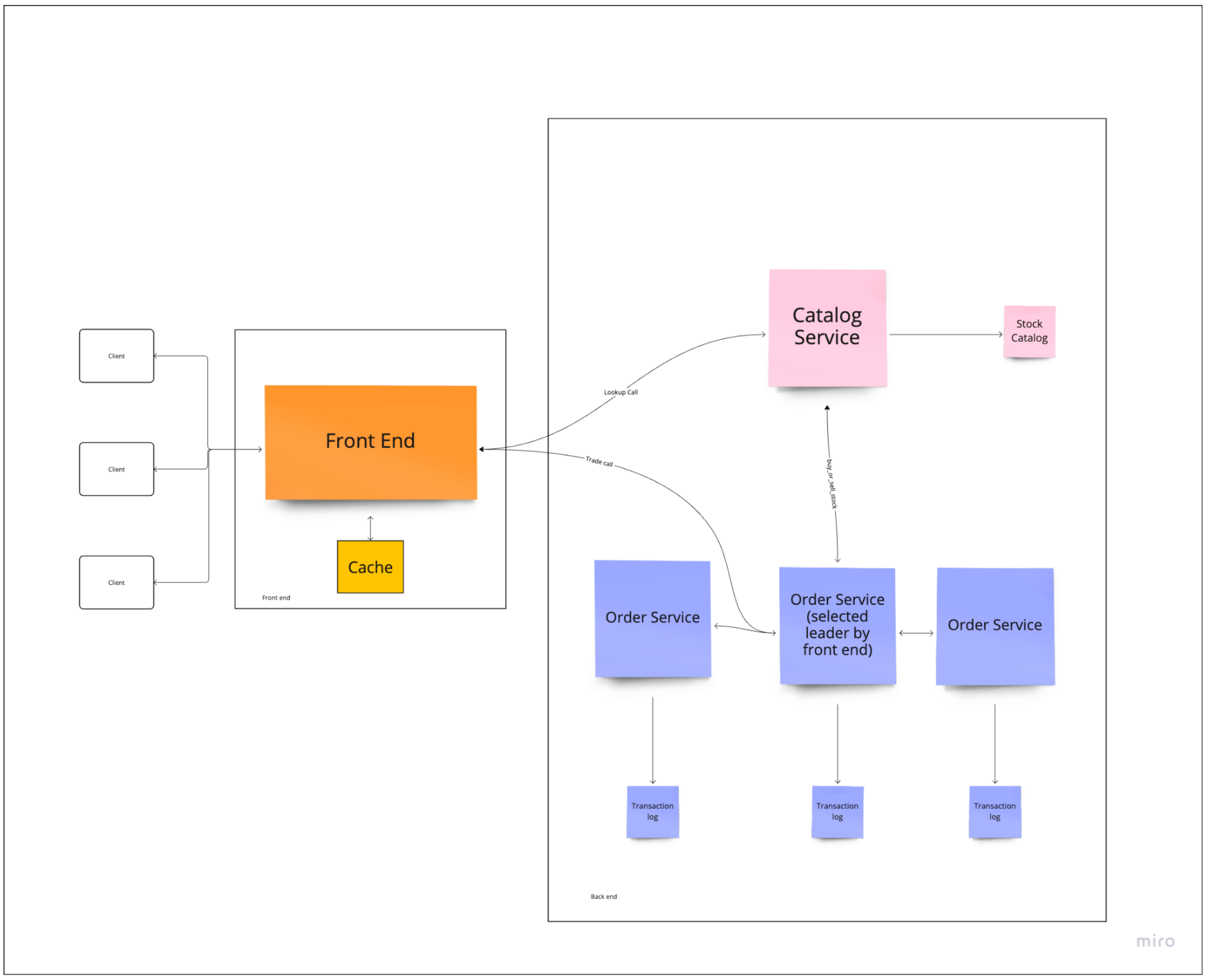

Figure. service design

c. **Front-End Service:** The frontend server is a REST API HTTP server.

The HTTP protocol version is set as HTTP/1.1 for getting persistent connections.

A subclass of the “http.server.BaseHTTPRequestHandler” is created. This subclass is named “MyHTTPHandlerClass”. It overrides the do_GET() and do_POST() methods of the BaseHTTPRequestHandler class. The front-end HTTP server is instantiated as an object of the “ThreadingHTTPServer” class in order to support multithreading. Thus, multiple clients can connect to the server and each client session runs on a single thread. The hostname and port of the HTTP server are obtained from the environment variables "FRONTEND_HOST" and "FRONTEND_PORT" respectively. This eases the process of dynamically assigning the host and port while running the front-end service.

“MyHTTPHandlerClass” is passed to ThreadingHTTPServer as the Request Handler class. After creating the object of the ThreadingServer with the “MyHTTPHandlerClass” as the request handler class, the serve_forever() method is called in order to start the HTTP server.

The constructor of the “MyHTTPHandlerClass” class creates a GRPC channel for communicating with catalog service and a GRPC channel for communicating with order service. The host of the catalog service is obtained from the environment variable “CATALOG_HOST” and the port is obtained from the environment variable “CATALOG_PORT”. The host of the order service is obtained from the environment variable “ORDER_HOST” and the port is obtained from the environment variable “ORDER_PORT”. It also elects the leader of order services and notifies all order service replicas. The host and port of the two services are taken from environment variables so that the host and port can be changed easily with minimal effort. The environment variables “CATALOG_HOST” and “CATALOG_PORT” specify the hostname and port for the catalog service. The environment variables “ORDER_PORTS”, “ORDER_HOSTS”, and “ORDER_ID” specify the list of ports, hostnames, and service ids of all the order service instances respectively.

“handle_one_request()” is called whenever a request is executed by the “MyHTTPHandlerClass” class. The method calls the “handle_one_request()” method of the superclass BaseHTTPRequestHandler and prints the thread currently executing the request and the client address for debugging.

convert_json_string() method takes a JSON string as an argument and encodes and returns the JSON string into bytes using “encode('utf-8')”.

create_and_send_response() method takes status_code, content_type, content_length, and response as arguments and writes and sends the HTTP response along with the HTTP headers “Content-type” and "Content-Length".

do_GET() is the method called by the HTTP server for lookup requests and cache invalidation requests from the client. The client sends HTTP GET requests to the HTTP server.

LRUCache is a class implemented for LRU cache to cache stockname lookups. It uses locks for accessing any key in the cache to maintain consistency.
It has a method “get” to get the cached value if key is present in cache. The “put” method updates the value of a key. If the cache capacity is 0 the “put” method doesn’t do anything and just returns. It pops the last used key if th size of the cache exceeds the capacity. The “invalidate” method removes a key from the cache if it is present.

The front-end service cache size can be set using the environment variable “CACHE_SIZE“ and the default value is set to 5. It was set as such for testing where the number of stocknames is 10.

There are 4 operations supported by the front-end service for GET requests:

1. Stockname lookup:

The stock name for lookup is sent in the URL for the GET method. The “path” variable contains the URL of the GET request. It is parsed in order to get the stock name. If the URL is not of the format "/stocks/<stock_name>", then the HTTP server sends a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 400,
      "message": "Invalid GET request/URL"
   }
}
```
The HTTP status code sent is 400 with reason/message “Bad Request”

In case of a valid URL, the stockname is obtained from the parsed URL. Then, the stockname is looked up in the cache. If stockname is cached, the cached stock information for the stockname is returned to the client. The stockname, price, and quantity obtained from the cache are used to send the HTTP response to the client. The response is a JSON reply with a top-level data object:

```
{
   "data": {
      "name": "GameStart",
      "price": 15.99,
      "quantity": 100
   }
}
```

If the cache lookup didn't return a value for the stockname, then the front-end service makes a lookup call to catalog service. Then, based on the response received from the catalog service, different responses are sent back to the client. If the response error is "NO_ERROR", then it refers to a successful lookup. The stockname, price and quantity obtained from the response is used to send the HTTP response to the client. The lookup response from the catalog service is also cached here. The stockname, price and quantity are stored in the cache as a single JSON string with the stockname as key, for example "{'data': {'name': 'GameStart', 'price': 15.989999771118164, 'quantity': 9012}}". After caching the response, an HTTP response sent to the client which is a JSON reply with a top-level data object:

```
{
   "data": {
      "name": "GameStart",
      "price": 15.99,
      "quantity": 100
   }
}
```

The HTTP status code sent is 200 with reason/message "OK"

In case the stock name was not found and the response error from catalog service is "INVALID_STOCKNAME", the HTTP server sends a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 404,
      "message": "stock not found"
   }
}
```

The HTTP status code sent is 404 with reason/message "Not Found"

In case of some other error, where the response error from catalog service is "INTERNAL_ERROR", the HTTP server sends a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 500,
      "message": "Lookup Failed due to internal error"
   }
}
```

The HTTP status code sent is 500 with reason/message “Internal Server Error”

2. Order number lookup:

The order number for lookup is sent in the URL for the GET method. The “path” variable contains the URL of the GET request. It is parsed in order to get the order number. If the URL is not of the format "/orders/<order_number>", then the HTTP server sends a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 400,
      "message": "Invalid GET request/URL"
   }
}
```

The HTTP status code sent is 400 with reason/message “Bad Request”

In case of a valid URL, the order number is obtained from the parsed URL. The front-end service makes a lookup call to the order service. Then, based on the response received from the order service, different responses are sent back to the client. If the response error is “NO_ERROR”, then it refers to a successful lookup. The number, name, type, and quantity obtained from the response is used to send the HTTP response to the client. The HTTP response sent to the client is a JSON reply with a top-level data object:

```
{
   "data": {
      “number”: 165
      "name": "GameStart",
      "type": “sell”,
      "quantity": 100
   }
}
```

In case the order number was not found and the response error from order service is “INVALID_ORDERNUMBER”, the HTTP server sends a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 404,
      "message": "order number not found”
   }
}
```

The HTTP status code sent is 404 with reason/message “Not Found”

In case of some other error, where the response error from order service is “INTERNAL_ERROR”, the HTTP server sends a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 500,
      "message": "Lookup Failed due to internal error"
   }
}
```

The HTTP status code sent is 500 with reason/message “Internal Server Error”

3. Cache Invalidation:

The stockname for lookup is sent in the URL for the GET method. The “path” variable contains the URL of the GET request. It is parsed in order to get the order number. If the URL is not of the format "/stocksCache/<stock_name>", then the HTTP server sends a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 400,
      "message": "Invalid GET request/URL"
   }
}
```
The HTTP status code sent is 400 with reason/message “Bad Request”

In case of a valid URL, the stockname is obtained from the parsed URL. The stockname is invalidated in the front-end cache. If the cache invalidation is successful that is if the key was removed if present in the cache, then the code and message is used to send the HTTP response to the client. The HTTP response sent to the client is a JSON reply with a top-level data object:

```
{
   "data": {
      “code”: 200
      "message": "Cache Invalidation done"
   }
}
```

The HTTP status code sent is 200 with reason/message “OK”

In case the cache invalidation failed, the HTTP server sends a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 500,
      "message": "Cache Invalidation Failed due to internal error”
   }
}
```

The HTTP status code sent is 500 with reason/message “Internal Server Error”

4. Front-end cache stockname lookup:

This method is used to lookup a stockname from cache only. This was used for testing purpose in our unit test. The stock name for lookup is sent in the URL for the GET method. The “path” variable contains the URL of the GET request. It is parsed in order to get the stock name. If the URL is not of the format "/stocksCacheLookup/<stock_name>", then the HTTP server sends a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 400,
      "message": "Invalid GET request/URL"
   }
}
```

The HTTP status code sent is 400 with reason/message “Bad Request”

In case of a valid URL, the stockname is obtained from the parsed URL. Then, the stockname is looked up in the cache. If stockname is cached, the cached stock information for the stockname is returned to the client. The stockname, price, and quantity obtained from the cache are used to send the HTTP response to the client. The response is a JSON reply with a top-level data object:

```
{
   "data": {
      "name": "GameStart",
      "price": 15.99,
      "quantity": 100
   }
}
```

In case the stock name was not found in cache, the HTTP server sends a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 404,
      "message": "stock not found”
   }
}
```

The HTTP status code sent is 404 with reason/message “Not Found”

do_POST() is the method called by the HTTP server for trade requests from the client. The client sends HTTP POST requests to the HTTP server. The client attaches a JSON body to the POST request to provide the information needed for the order name, quantity and type.
The “path” variable contains the URL of the POST request. If the URL is not of the format "/orders", then the HTTP server sends a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 400,
      "message": "Invalid POST request/URL"
   }
}
```
The HTTP status code sent is 400 with reason/message “Bad Request”

The server reads the JSON object attached by the client to the POST request as the first step since if the URL is incorrect then, returning the HTTP response without reading it can cause errors, as it this JSON object will be read in the subsequent request. Hence, after reading the request’s JSON object the URL format is checked. The request is read using the rfile.read() method and the amount of bytes to be read is determined by the header “Content-Length”.

If the header “Content-type” is not set to "application/json" or if any of “name”, “quantity” and “type” is not present in the JSON object, then the HTTP server sends a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 400,
      "message": "Invalid request- JSON object should contain name, quantity and
type"
   }
}
```
The HTTP status code sent is 400 with reason/message “Bad Request”

In case of valid JSON object, the order information(name, quantity and type) is populated from the received JSON object. Then the HTTP server makes trade call to order service using the GRPC channel created.

In case the trade request was successful from the order service and the response error from the order service was “NO_ERROR”, the HTTP server sends the JSON object containing the Transaction number. The transaction number is obtained from the response from the order service.

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

The HTTP status code sent is 200 with reason/message “OK”

In case the trade request was not successful from the order service due to invalid order type and the response error from the order service was “INVALID_REQUEST”, the HTTP server a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 400,
      "message": "Order type is invalid, only buy/sell are accepted"
   }
}
```

The HTTP status code sent is 400 with reason/message “Bad Request”

In case the trade request was not successful from the order service due to invalid stock name and the response error from the order service was “INVALID_STOCKNAME”, the HTTP server a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 404,
      "message": "stock not found"
   }
}
```

The HTTP status code sent is 404 with reason/message “Not Found”

In case the trade request was not successful from the order service due to insufficient quantity and the response error from the order service was “INSUFFICIENT_QUANTITY”, the HTTP server a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 400,
      "message": "Available quantity to buy is less than requested quantity”
   }
}
```

The HTTP status code sent is 400 with reason/message “Bad Request”

In case the trade request was not successful from the order service due to some other error and the response error from the order service was “INTERNAL_ERROR”, the HTTP server a JSON reply with a top-level error object:

```
{
   "error": {
      "code": 500,
      "message": "Stock could not be traded due to internal error"
   }
}
```

The HTTP status code sent is 500 with reason/message “Internal Server Error”

d. **Client:** The client takes the HTTP server host and port as command line arguments and is called using: python3 client.py <host> <ip> <p>.
The client is implemented using the “http.client” library. The client establishes a connection with the HTTP server using the method HTTPConnection() of the “http.client” library. Then, for a number of iterations in the while loop, it sends a lookup request of a randomly selected stock name from a list of stock names. It sends a GET request to the server using the request() method. Then, it reads the response obtained from the server, and if the lookup was successful and the client received a JSON reply with the top-level data object, it sets the variable stock_quantity using the ‘data’ object’s 'quantity' field. If the lookup failed, the "stock_quantity" is set as 0, so a trade request is not sent for this failed lookup. Then, a random number is generated, and if this number is less than or equal to the probability “p”(passed as a command-line argument) and the stock_quantity is greater than 0, then a Trade request is sent to the server. This sequence of lookup and trade requests is sent over the same connection. The client records the order number and order information if a trade request was successful. Before exiting, the client retrieves the order information of each order that was made using the order number lookup request, and checks whether the server reply matches the locally stored order information. The connection is closed after all the client order lookup requests are done. Since the HTTP protocol version is set to HTTP/1.1, the connection is persistent and is thus a requests-per-session model.