Design Document

Implements online stock trading server using 2-tiered architecture (front end and back end) with replication, caching, leader election and database synchronization between all the 3 replicas of order server.

Design decisions:

CONFIGURATION FILE

A json config file is pre-written with all the 3 order server replicas details such as replica ID, replica PORT number and replicas specific database path. This file is used by front-end server and order server itself to fetch the ID's and other details about the replica for functionalities such as leader election and updating the database for the specific order replica etc.

FRONT-END SERVER

Python flask is used to implement the front-end http server, and requests library is used to send GET and POST requests to the backend and clients.

Data structures used:

I. **CacheFlag**
   Values: True/False
   A flag is used to switch On or Off the caching.
II. **order_replicas_status**
   {'1':False,'2':False,'3':False}
   A dictionary to maintain the alive status of all the order server replicas.
   Initially on startup all order server replicas status will be False, whenever an order server starts it will inform the front-end server upon which the above dictionary is updated.
III. **Data_dict**
   A dictionary is used to maintain the in memory cache

API Endpoints:

I. **/health_check**
   Request header: {}
   Request body: {}
   Success responses: 200
   Format: {'leader_response':'OK', 'leader_ID':3, 'leader_PORT': 5004}

II. **/update_order_replica_status/<replica_ID>**
   Request header: {}
   Request body: {}
   Success responses: 200
   Format: {'status':'OK'}

III. **/getleaderID**
   Request header: {}
   Request body: {}
   Success responses: 200
   Format: {'leader_ID':leader_ID,'leader_PORT':leader_PORT}

IV. **/queryOrder/<order_number>**
Request header: {}
Request body: {}
Success responses: 200
Format: {'data': {transaction_number': 160, ‘name': 'BoarCo', 'price': 30.25, 'quantity': 1}}
Error responses: 404
Format: {'error': {'code': 404, 'message': 'stock not found'}}

V. **/invalidate_MemCache**
Request header: {}
Request body: {}
Success responses: 200
Format**:** {'Invalidation Request' : 'Success'}
Error response: {'Invalidation Request' : 'Not Applicable'}

VI. **/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'}}

VII. **/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:

All interfaces are implemented using flask and http requests library are used to talk to the client and the backend servers.

I. **GET /health_check**
Invokes the health_check helper function. Where a GET connection is sent to the current leader to check whether the leader is alive or has crashed. If alive the leader will respond with **‘OK’.**

II. **GET /update_order_replica_status/<replica_ID>**
Whenever a new order replica is UP from a crash or is starting up for the first time, it will invoke this API endpoint inorder update the *order_replicas_status* dictionary which maintains the alive status of all the replicas. After updating dictionary leader election is invoked again because what if the newly UP order replica has the highest ID.

III. **GET /getleaderID**
GET method to return the current leader ID and leader PORT.

IV. **GET /orders/<order_number>**

Method upon receiving the request from the client will do a leader election, check whether the leader is alive using /health_check and forward the request only to the leader order service. Upon the response from the leader will forward back the data to the client.

V. **POST /invalidate_MemCache**

This POST request will be sent by the catalog server whenever the stocks database has changed so that the front-end can remove the stock data from the in memory cache.

VI. **GET /stocks/<stock_name>**

Forwards the request to catalog server. Upon the receival of data will forward the response to the client.

Caching –

A cache flag is used, if cache flag is True a dictionary is used to maintain the in memory cache for the stocks. Whenever a GET stock/<stock_name> request comes, that stock data is updated in the cache so that in future when the GET stock request for the same stock is seen, it can be fetched from the cache instead of forwarding the request to catalog server. This decreases the reponse time.

If cache flag is set to False then no in memory cache is maintained and all the GET stock request will be forwarded to the catcalog server.

VII. **POST /orders**

Forwards the request to the orders server.

Method upon receiving the request from the client will do a leader election, check whether the leader is alive using /health_check and forward the request only to the leader order service. Upon the response from the leader will forward back the data to the client.

Helper functions:

I. **leader_election (unresponsive_node,config_data=config_data)**

The above function will take the config_data from the json config file and pick the highest ID as the leader. If an unresponsive_node is passed then if that is the highest ID the function will ignore that ID as it is not responding and will pick the second highest ID as the leader.

II. **health_check_to_leader(leader_ID,leader_PORT)**

A recursive function which will send a GET request to the leader. If the leader is UP it will respond OK. Else leader election is done again by invoking function leader_election() by specifying the specific unresponsive node. Same function health_check_to_leader() is called again to check the newly elected leader status. This recursion stops when an elected leader is alive.

III. **broadcast_leader(leader_ID,config_data)**

This function will send the leader ID and leader PORT to all the order replicas that are alive.

BACKEND ORDER SERVER

On startup the current replica order server will first try to synchronize its database with the leader server by sending its last transaction number. If itself is the first server and no leader is present it will enter the except block and start the server.

After data sync the order server will send a POST request to the front-end server using helper function ***send_update_status_to_frontend()*** in order inform the front-end that the current replica has been UP. The front-end maintains a dictionary where it has all replicas alive status, upon contact by the specific order replica the dictionary will be updated.

Important variables to start the replicas –

self_ID=int(sys.argv[1])
PORT=sys.argv[2]
DB_path=sys.argv[3]
leader_ID=None
While starting the order service.py specific replica ID, PORT number and database path should be given.

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.s

I. **/health**
   Request header: {}
   Request body: {}
   Success responses: 200
   Format: {'status':'OK'}

II. **/get_order_data/<order_number>**
   Request header: {}
   Request body: {}
   Success responses: 200
   Format: {'data': {transaction_number': 160, ‘name': 'BoarCo', 'price': 30.25, 'quantity': 1}}
   Error responses: 404
   Format: {'error': {'code': 404, 'message': 'order not found'}}

III. **/sync_database**
   Request header: {}
   Request body: {}
   Success responses: 200
   Format: {'missing_rows': [[343,GameStart,buy,3],[344,BoarCo,buy,13]]}

IV. **/leader_broadcast**
   Request header: {}
   Request body: {}
   Success responses: 200
   Format: {'status':'OK'}

V. **/update_order_db**
   Request header: {}
   Request body: {}
   Success responses: 200
   Format: {'status':'OK'}

VI. **/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. GET /health
This endpoint will be invoked by the front-end server to check whether the order leader is alive or has crashed, if alive the order server will respond with status as 'OK'.

II. GET /get_order_data/<order_number>
This endpoint will be invoked by the frontend whenever it receives a query order from the client. This method will call a helper function called ***get_order_data(order_number)*** where it will look for the requested transaction number/order in its database. If present it will return the order data in a specified format to the front-end which will then forward it to the client. If not present then 'order not found' is sent.

III. POST /sync_database
This endpoint will be called by the order server replica whichever started just now in order to fetch the missing rows from the leader.

IV. POST /leader_broadcast
Invoked by the front-end server to broadcast the leader ID and PORT to all the replicas. All the replicas will save the leader ID and PORT for further use/communication with the leader.

V. POST /update_order_db
Whenever a successful trade transaction happens, the leader will update its database first and then call this POST method for all the other replicas in order to update their database.

VI. POST /trade_stocks
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.

BACKEND CATALOG SERVER

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/<stock_name>**
Request header: {Content-Type: application/json\r\n}
Request body: {}
Success response:200
Format: ‘OK’

Interfaces

I. **GET /Lookup_csv/<stock_name>**
The method will look for the request stock name in its database, 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.

II. **POST /Update_csv/<stock_name>**
This method will be invoked by the order server for a valid trade transaction. The quantity of the stock is updated in the database. If successful returns 200 else 404.

Helper functions

I. **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’.

II. **Update_csv(stock_name,quantity)** 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.

III. **update_InmemCache(stock_name)**
This helper function is used to send a invalid memory cache request to the front-end server after any changes has taken place in the database for any stock name. So that the front-end server can delete that stock data from the in memory cache as it is no longer valid.

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.

Dictionaries are used to save the orderQuery data and the successful trade data.

Design Choices:

1. A random generator is used to pick a value between 0-1. On startup first the client will send a lookup request for a stock that is randomly picked from the list.
2. After a successful lookup if the generated prob is less than the specified probability (manually configurable variable – [0,0.2,0.4,0.6,0.8]) then trade request to the same stock is done and the reponse is stored in a dictionary for few iterations.
3. Once all trade and lookup requests are done, queryOrder request is sent for the same stocks for which the trade requests were done earlier using function ***queryOrder_Status().***
4. Both the trade responses and the query order responses are saved in a dictionary and compared to check whether both are same.