DESIGN DOC

We are initializing the catalog data with 10 stocks and maintaining all the microservices host and port information, leader id, and if caching is enabled or not in a config.json file and the config file was read by all microservices at the start to get all the information. In addition to this, all the design decisions made in lab 2 are also made here. For this lab, we are using Flask to handle our API requests in the front end, catalog, and order microservices.

Caching:

We initialized an in-memory cache dictionary, in the frontend service, with stock names as its keys to maintain the stock information received from the catalog service. We are evicting the keys from the cache by following the LRU(Least Recently Used) cache policy. To achieve that, we are also storing/updating the time whenever a stock is updated/read such that to use this information in further requests to evict the stock from the cache if the cache maximum size is reached.

Whenever a lookup request arrives at the frontend service, first it checks whether the given stock is present in the cache or not, if the stock is present in the cache then we are directly getting the data from the cache and returning it directly to the client without any API call to the catalog service. If the stock is not available in the cache, then we are calling the catalog lookup API and getting the information from API, after getting the information, we are updating the cache with the received response such that it can be used in the further requests. In the same process, we are also updating the time the stock was accessed for eviction. For all the read operations of the cache, we are using the read lock and for all write operations, we are acquiring the write locks as the cache value is a shared data structure among the threads to avoid any racing conditions.

There might be a chance that the cache value can be outdated as the trade requests can happen parallelly at the order service. So to constantly update the cache, whenever a successful trade request happens at the catalog service, we are using a server-push technique to invalidate the cache. To achieve this, we exposed an API endpoint in the frontend service, so the catalog service calls this API to invalidate the cache by passing the stock name. In this API, we are deleting the stock name from the cache if it is present in the cache. Here also, we are acquiring a write lock to the cache.

We verified the cache functionality by introducing the sleep of 2 seconds in the catalog lookup API. So, for the first time when a request arrives at the frontend service, since the stock name is not present in the cache, it will get the data from the catalog service and the request takes around 2 seconds to get served. But, when we call the API with the same stock again, now the response time will be <50ms as the data is accessed from the cache. To verify the LRU eviction policy, We have given a maximum cache size of 6 and called the lookup API with 5 different stocks and called the API once again with the initial stock, and noted the response time. We got

the response time as around 2 seconds which is expected as the stock name is evicted since it is least recently accessed.
Caching data structure looks like this: {'Amazon': {'value': {'data': {'name': 'Amazon', 'price': 110, 'quantity': 100}}, 'last_access': 1682979634.2976232}}

Replication:

To replicate the order service with 3 nodes each with its own database file running on 3 different ports, we have configured the unique id for each order service in the config.json file. So, whenever the frontend service is started, we are reading the config file and get the order nodes' information and their ids, and sorting the order nodes based on their ids, and then the frontend service initializes the leader election process. The leader election process is done in the following way, firstly the frontend service gets the highest id of the order node and checks its health by calling "/ping" API, if it is successful, then the particular node is elected as leader, and updates the leader id in the config.json file such that it can be used in the future. After the leader election is done, the frontend service notifies all the other nodes through the "/notify_leader" API. In the notify_leader API, we are just storing the leader id in its memory. This process is followed until the leader is elected by checking the next highest ids.

Note: We are using try catch to know if a node is unresponsive and take further action.

Now that the leader is elected, all the trade requests from the frontend service will be forwarded to the leader node. After a successful trade request in the order service, now the data should be synchronized with the other nodes. To achieve this, we are getting the other nodes' information and for each node, we are calling the "/sync_data" API with the corresponding data for that particular trade request to update the data in the node's own database. In the sync_data API, we are just appending the request payload to the transaction history list and updating its own db file. We are doing this process by acquiring the write lock to the transaction as it is a shared data structure.

Fault Tolerance:

Now whenever a leader server is crashed, and when a trade request arrives at the frontend service since the leader node is unresponsive, it will re-elect the leader based on the algorithm defined above and updates the new leader id in the config.json file. We are handling the case where the leader node is unresponsive, by using try, except operators. Since a new leader is elected now, all the order-related requests will go to the new node and it will sync the data with the other nodes.

So, now when the crashed node (initial leader) comes back online, it will sync the data with the leader node by getting the leader id from the config.json file. For this, it will read the database file and gets the last order transaction number, and calls the "/backlog/<transaction_number>" API to the leader node. The response received from the leader node is concatenated with the already present database data. This process is done at the start of

the program. Here, the locking is not required as it is done at the start and no threads are using this data until this process is done. In the backlog API, we are iterating the transaction list and matching the transaction_number received in the request in the list, and returning the data after the found item if item is found. Initially, at the start of the whole application, the nodes will pass -1 as "transaction_number" to the leader nodes, in that case, we are just returning the entire transaction list available at the leader's node as no data is present at that time. While reading the transaction list at the leader, we are using read locks to lock the transaction data to avoid race-in conditions.

Frontend:

In the frontend service, we added an extra GET API for this lab to get the order information by passing the order number. When the order info request arrives at the frontend service, it will get the leader order node information and forwards the request to the leader node. At the order service, based on the received order number in the request, it will iterate through the transaction data and matches it with the id, and returns it if an order is found. Otherwise, it will throw a 404 error with an appropriate message. Then at the front end, for successful requests, a top-level data object is added to the response and returned to the client, and for 404 requests, a top-level error object is added to the error object and returned to the client.

Client:

Whenever a client is started with a configurable probability, the client sends a bulk of requests to the frontend service with catalog lookup requests with a random stock from the available stocks, if the request is successful and based on the probability it will send a follow-up trade request in the same session as of the lookup with random quantity and the random type and stores its local payload data in a list and also the received response in the trade request. Now, before exiting the client will validate its local order data and the server's data. This is done by iterating the before-stored list of local order data and for each order_number, it will sequentially call the order info API and validates the response with the local data. If it matches, then we are incrementing the success count, otherwise, the failed count is incremented.

Interfaces:

Catalog microservice:

1. GET /catalog/<stock_name>

Description: API to lookup stock information by passing stock_name in the URL path.

Functionality: We are handling this request by submitting it to the thread pool. We are just checking for the passed stock name in the in-memory data structure. We are acquiring the read lock when reading the data structure as it is shared among threads to avoid race-in conditions. If we found the stock name in the data structure, we are simply returning it otherwise we are returning an error with an appropriate message.

2. PUT /catalog

   Description: API to check if a stock is present or not and if present update(increment/decrement) the stock quantity based on the transaction type and increase the trading volume of the stock

   Functionality: This API is exposed to order microservice. The order service calls this API with the respective data which contains the stock_name field. This is used to check lookup function and if it passes, then we will update the quantity and trading volume in the catalog database.

Order Microservice:

1. POST /orders

   Description: API to trade stocks by sending stock name, quantity and transaction type. It then calls the catalog microservice to check if the stock is found and requested quantity is available for buy type and updates (increments/decrements) the quantity accordingly in the catalog database and returns the updated stock details to the order microservice. The order microservice then logs the received response into its database and generates the unique transaction number and returns the transaction number to frontend in JSON format. It then updates the in memory data structure. For the order replication, after the trade request is successful, it will sync its data with the other replica nodes by calling the "/sync_data" API and it syncs the data with the other nodes.
2. GET /ping

   Description: This API is used to check the health of the order nodes. It sends status: ok message if it is responsive. It is used for the leader election algorithm.
3. POST /notify_leader

   Description: This API is used to notify the other nodes that a leader has been elected. This API sends the leader node id in the request payload, in the API, the leader node id is read and it is set to a local variable in the memory
4. POST /sync_data

   Description: This API is used to synchronize the data from the leaders. On successful trade request the leader node calls this API to push the data to the other nodes. At the replica nodes, the data is appended to the transaction list and it is written to the db file. We use write locks to write the data here.
5. GET /backlog/<transaction_id>

   Description: This API is used to update the data in the replica nodes from the leader in a scenario where a crashed node comes back online after sometime. In the API, the crashed node on coming back, reads the last transaction number and passes it to the API, and in the other node, it finds the transaction number and returns the transaction from there onwards. From the received response, the crashed node updates its database file by appending it to the already present list.
6. GET /orders/<order_no>

Description: This API is used to fetch the order information based on the order id received in the request. In this API, in the transaction database, if the order no is found, then it returns the found order information to the frontend and if no order is found, then it throws an error with an appropriate error message.

Frontend Service:

1. GET /catalog/<stock_name>

   Description: This API is used to lookup the stock information in the catalog microservice with the given stock name. It returns if the given stock is found with top-level data object. Otherwise, it returns an error with top-level error object. In this lab, we are using the cache in this API to serve the requests.

2. POST /orders

   Description: This API is used to trade the requests in the order microservice. If a trade request is successful then the API returns the transaction number of the trade request with top-level data object. If the trade request fails, then it returns the error with top-level error object. In this lab, if the order service is unresponsive, then we will initialize the leader election process and re-elect the leader.

3. GET /orders/<order_no>

   Description: This API is used to get the order information for a given order number. The frontend service calls the order service for order information, if the order number is found then it returns the order information with top-level data object. If it is not found, then the API returns the error with top-level error object. In this lab, if the order service is unresponsive, then we will initialize the leader election process and re-elect the leader.

4. POST /cache

   Description: This API is used to invalidate the cache at the frontend service. On successful trade request, the catalog service calls this API to remove the stock from the cache such that to avoid the data inconsistency. So when this API is called, we will check if the stock name is present in the cache or not, if present, then we will remove it from the cache. We use write locks to write the cache data to avoid race-in conditions.