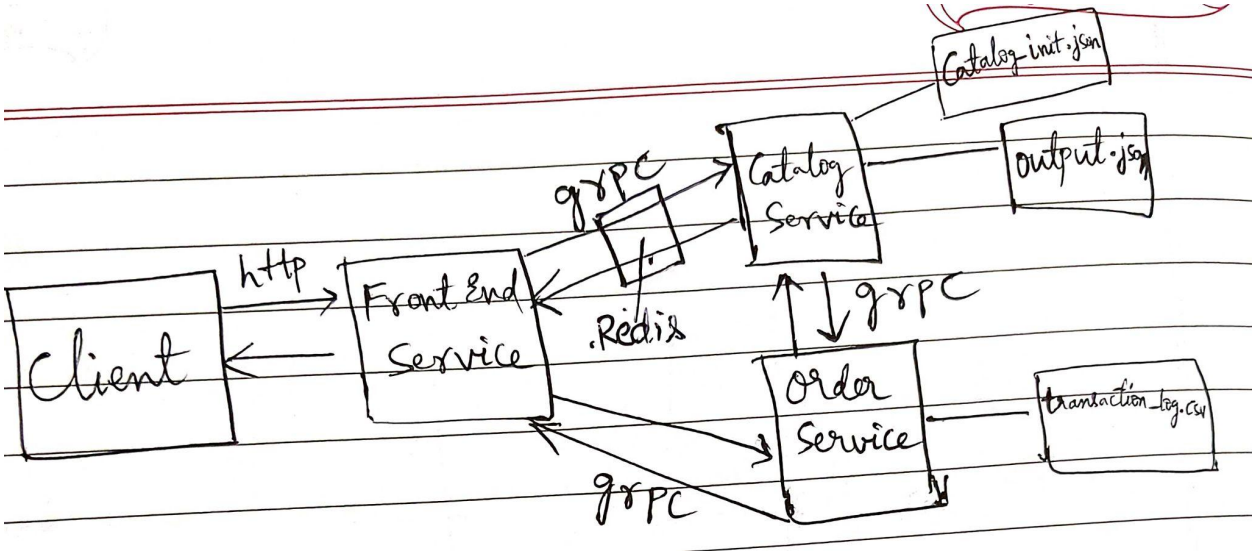
Design Document Lab 3

Architecture:



1. Catalog service uses catalog_init.json to initialize the trade volumes on every new start and output.json contains the final state of the catalog on server exit. If the server is up again, then output.json would be the initial state of the server. In order to initialize the server with catalog_init, get a git pull or clear output.json.
2. transaction_log.csv maintains the history of transactions present in the order service.
   Every time the server is rebuilt, the transaction number is continued from the last transaction_id in transaction_log.csv. In order to reinitialize the server with transaction_id=0, get a git pull again or clear transaction_log.csv.
   Each order_service has it's own transaction_log.csv i.e service with id one has transaction_log_1.csv and likewise respectively.
3. Output.json and transaction_log.csv of running order service containers are updated after every successful order request.

Communication between services:

Client to Frontend: HTTP

All internal calls after frontend service use gRPC as the internal communication protocol to handle client requests.

Whereas all replication related strategies like database synchronization communicate via http requests.

Database:

1. Output.json for catalog service
2. transaction_log_i.csv for order_service_i (where i is the service id of order_service)

Front End Service-

The frontend service now has been refactored and used the Flask framework in place of the previous implementation. Just like last time, the GET and POST requests are implemented for stocks and orders respectively. In addition to these two, a new GET request was also implemented which gives us a response for a particular order number. We are utilizing the grpc channels of both catalog and order services in run_lookup and run_order methods respectively, which help us provide necessary results for those requests.

API Contracts::

1. GET /stocks/<stock_name>

   Params: provide the name of the stock to fetch
   Body: None
   Sample Request: http://0.0.0.0:8081/api/frontend_service/trade/stocks/GameStart
   Sample Response: "data": {
   "name": "GameStart",
   "price": 10.0,
   "quantity": 100.0
   }

2. POST /orders

   Params: None
   Body: None
   Sample POST: http://0.0.0.0:8081/api/frontend_service/trade/orders,json={
   'name': “GameStart”,
   'quantity': 5,
   'type': “sell”
   }
   Sample Response: data": {
   "transaction_number": 1
   }

3. GET/orders/<order_number>

   Params: provide the order number
   Body: None
   Sample Request: http://0.0.0.0:8081/api/frontend_service/trade/orders/2

Sample Response: {
"success": true,
"data": {
"number": "2",
"name": "FishCo",
"type": "sell",
"quantity": "4.0"
}
}

4. GET /ping

   Params: None
   Body: None
   Sample Request: http://0.0.0.0:8081/api/frontend_service/ping
   Sample Response: {
   "success": true,
   "data": {
   "data": "pong"
   }
   }

Note: Standard and basic http errors like 400's are handled.

Catalog Service:

The Catalog Service implements two GRPC interfaces:

1. Lookup
2. Trade

Lookup Method: Frontend calls this method on the catalog service to make a lookup. Only the stock_name argument needs to be passed, and we can expect the return to be success and stock_ details.

Return values:
{-1, {}} if stock_name isn't present
{1, {name:"meta", quantity:"10", status:"0"}} if stock status is active.
{0, {}} if stock status is suspended.

Trade Method: Order Service calls this method on catalog service to execute an order it received from frontend.

Args: stock_name, trade_volume, type
Return: {success}

Success Values:
1 - Successful trade
0 - if trade volume < 0 & if stock is suspended to trade
-1 - if stock_name is invalid
-2 - if trade_volume > stock quantity

The catalog service now also takes care of caching using redis.

Order Service:
Implements one GRPC interface Order

Order Method: Take an order request from Frontend Service

Args:stock_name, trade_volume, type
Return: {success, transaction_id}

Return Values:
{1, 1} - transaction_id is legit only if success is 1
{0, -1} - transaction_id=-1 if order fails.

API Contracts::

1. GET /ping

   Params: None
   Body: None
   Sample Request: http://0.0.0.0:8081/api/order_service/ping
   Sample Response: {
   "success": true,
   "data": {
   "data": "pong"
   }

}

2. GET /query/<order_number>
   Params: provide the order number
   Body: None
   Sample Request: http://0.0.0.0:8081/api/order_service/order_query/query/2
   Sample Response: ["2", "FishCo", "sell", "3.0"]

3. GET sync_db
   Params: None
   Body: None
   Sample Request: http://127.0.0.1:6298/api/order_service/sync/sync_db
   Sample Response: synchronizes databases between replicas

How caching works?

In the updated system architecture, a caching server (Redis) has been introduced to improve lookup request efficiency and keep the frontend updated with the latest stock details. Previously, the frontend would directly query the catalog service for stock information, which was time-consuming and resource-intensive. With Redis as an intermediary layer, the frontend now communicates with Redis for stock details, resulting in improved response times and reduced load on the catalog service. So, a read through cache mechanism was implemented for stock lookup i.e. if stock details are present in cache then they are directly returned if not a query is made to the catalog service.

To maintain accurate stock information, a mechanism has been implemented for handling changes. When a new trade or order is placed, the catalog service invalidates the corresponding stock details in the cache, indicating that certain cached stock details are no longer valid. Redis then takes appropriate action to remove or update the invalidated entries, ensuring subsequent lookup requests receive accurate and current information by retrieving from the database of the catalog service. By implementing this invalidation mechanism, the system ensures that the frontend receives the most recent stock details while benefiting from the caching server's performance advantages.

How replication works?

In our system architecture, we have implemented a setup consisting of three replicas of the order service, each with its own dedicated database files. To ensure efficient trade requests handling, we have incorporated a leader election algorithm within the frontend service. This algorithm

selects a leader from the available replicas and directs all trade requests exclusively to the leader. Additionally, the frontend notifies the other replicas about the elected leader.

When a trade occurs, the leader takes the responsibility of informing the other healthy replicas about the new transaction. This synchronization ensures that all replicas have updated information regarding the trades. However, in the event that the leader becomes unresponsive or goes down, the frontend quickly detects this situation by doing a healthcheck and initiates a new leader election process. This ensures the continuous operation of the system even in the face of failures.

The newly elected leader assumes all the responsibilities previously carried out by the previous leader and also checks its database state with the last consistent replica and will take an action to synchronize if required before handling any further trade requests. It takes over the handling of trade requests and notifies the other replicas whenever new trades take place. This process guarantees that all replicas' database files remain synchronized, and no information is lost during the transition.

Tradeoffs:

1. The final trade for an order request is implemented in the catalog service. So, in this design, order service is adding an extra layer of latency to the execution.
2. For now, the system isn't completely fault-tolerant due to a lack of replication in components like frontend service, catalog service and cache layer. Also, in heavy load scenarios, front end would contribute to the highest decrease in performance in comparison to other components as HTTP is the communication protocol it interfaces with the client (gRPC is faster compared to HTTP).
3. Also, the replication and synchronization of databases of order replicas is consistent only if only one container goes down at any point in time. If more than one goes down, then the state of the transaction log becomes inconsistent and can never be retrieved in the current system. Although it can be corrected with implementation of complex synchronization strategies when all the services are back up again (requires cross synchronization across all databases of all the replicas).