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.
3. To provide an HTTP GET request that can be used to retrieve information about an existing order.

To implement caching in frontend to reduce latency of stock lookup.
To replicate the order service to make it fault tolerant.
To deploy it in AWS.

Implementation Details

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

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"
  }
}
```


- /orders?order-number=<order id> - Returns the order information of an existing order
  Sample request: GET *http://localhost:26111/orders?order-number=1*
  Sample response:


```
{
  "data": {
    "number": 1,
    "name": "stock3",
    "type": "SELL",
    "quantity": 25
  }
}
```


  Sample error response:


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


  AWS Endpoints:

  hello GET - *http://ec2-3-238-71-234.compute-1.amazonaws.com:26111/hello?name=user*

  stock lookup GET - http://ec2-3-238-71-234.compute-1.amazonaws.com:26111/stocks?stockname=stock1

  Order Lookup GET - http://ec2-3-238-71-234.compute-1.amazonaws.com:26111/stocks?stockname=stock1

  Post Order POST - http://ec2-3-238-71-234.compute-1.amazonaws.com:26111/orders

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.
- OrderLookupRequest: Contains the id of the order to look up.
- OrderLookupResponse: Contains the order information along with the status of the request. Contains the order id(int), status of the request(1, 0, or -1), name of the stock(string), trade type (BUY or SELL) , and quantity (int).
- CacheInvalidateRequest: Contains the name of the stock (string) that has to be invalidated from the cache.
- AliveResponse: Contains a boolean is_alive indicating the status of the service.
- OrderDBItem: Contains the information about an order. Contains the order id (int), name of the stock (string), trade type (BUY or SELL), and quantity (int).
- SyncRequest: Contains the highest id of the order (int) present in the in-memory database.
- SetLeaderRequest: Contains the id of the leader node (int).
- GetLeaderResponse: Contains the id of the leader node (int).

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.
- The OrderLookup() method takes a OrderLookupRequest message containing the order id. It performs a lookup on the order id in the database and returns an OrderLookupResponse containing the id of the order, the name of the stock in the order, the type of the trade performed, and the quantity traded. It also contains a status field, which is 1 if order is found, 0 if order is not present, -1 otherwise.
- The StreamDBUpdates() method is used to stream CacheInvalidateRequest when any transaction is performed on a stock. The CacheInvalidateRequest contains the name of the stock that has been traded and has to be invalidated from cache.
- The IsAlive method is used to check whether a particular order service is alive or not. It takes an Empty message and returns a response containing a bool is_alive which tells whether a service is alive or not.
- The SyncOrderRequest() method takes an OrderDBItem message as input which contains stockname, quantity, trade type and transaction id, and adds the order details to the recipient service's database. It returns an empty message.
- The SyncOrderDB() method takes in a SyncRequest message which contains the maximum transaction id at the sender service and streams back OrderDBItem messages which contains the order details of all the transactions greater than maximum transaction id received.
- The SetLeader() method is used to set the elected leader. This message is sent by frontend to all the order service. The input message SetLeaderRequest contains the elected leader_id.

- The GetLeader() method is used to get the elected leader from an order service. The response message contains the leader_id.

Data model

1. 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.

2. The *Order* data model is used to store the details of an order. The Order data model has the following attributes and functions.

Attributes:

- order_id: The transaction number of the order.
- stockname: The name of the stock that was traded in the order.
- trade_type: The type of the trade (BUY or SELL)
- quantity: The quantity of the stock traded in the order.

Methods:

- The __init__(order_id, stockname, trade_type, quantity) constructor to initialize the order object.
- The to_string() method converts the order object to a readable string.

Cache

The implementation of the LRU Cache is based on the doubly linked list and dictionary data structures. The doubly linked list is used to maintain the order of items in the cache, while the dictionary is used to provide quick access to items in the cache. The LRUCache class is used to implement the LRU Cache. The maximum size of the cache is configurable (specified by *cache_size* in *config.py*). When the cache is full, the least recently used item is evicted from the cache. When an item is accessed, it is moved to the front of the doubly linked list, indicating that it is the most recently used item. The class contains the following methods:

Methods:

- The constructor initializes the cache with a specified size. It initializes the current size to zero and creates an empty dictionary to store the cache items. It also creates a doubly linked list with a head and tail node.
- The get(key) method is used to retrieve an item from the cache. It takes a key as an argument and returns the corresponding value if the key exists in the cache, otherwise, it returns None. If the item is present in the cache, it is moved to the front of the doubly linked list, indicating that it is the most recently used item. If the item is not present in the cache, None is returned.
- The put(key, value) method is used to insert or update an item in the cache. It takes a key and value as arguments. If the key already exists in the cache, the method removes the item from the cache and updates the cache with the new item. The new item is added to the front of the doubly linked list, indicating that it is the most

recently used item. If the cache is full, the method removes the least recently used item before inserting the new item.

- The remove(key) method is used to remove an item from the cache. It takes the key of the item to be removed as an argument. If the key is present in the cache, it removes the item from the cache and its corresponding node from the linked list.

Leader election:

Since we are maintaining multiple replicas of order services, we need to decide on an order service that the frontend makes requests to. Otherwise we need to send requests to all the order services from frontend which is not ideal/scalable (frontend needs to keep track of all the order services - overhead at frontend). So the frontend elects a leader and sends all the requests to that leader. The leader will take the responsibility of keeping the other services in sync.
Here we use a simple leader election algorithm. The frontend selects an alive order service with the highest id. In our code we assume the ids are in the same order as ports. So we send isAlive requests to all the services in reverse order of ports and choose the one that responds with is_alive true.

Replication:

Replication is an important feature in distributed systems. Whenever a leader service goes down, we need a backup service that can take care of incoming requests. For this we run a set of replica order services so that even if a few services go down, one of the alive ones can take care of the incoming requests. But in order for a new service to accept requests and respond correctly we need a copy of the leader database. For this we implement replication of data at all of the services so that they are all in sync and a client wouldn't know if some failure occurs to the leader service and a new service is elected as leader. Whenever an order is successfully processed at leader service we send the information to all the replicas. The replica services will store the data in their local dbs. This way we can make sure that all the replicas' databases are in sync.

Fault Tolerance:

One important thing in distributed systems is that we need to always serve the user requests even if some of the servers go down. For this we implement replication and add some strategies on top of that to make them fault tolerant. As said in replication, whenever a leader service goes down, we elect a new leader and start serving the requests.But when the service comes online again, we need to make sure that its data is in sync with the other replicas. In order to do this, in our code we did the following: whenever a service starts, we communicate with other replicas and get the leader id. Once we know the leader, we send the last transaction id and get all the orders that happened after that transaction at the leader. This way we can make sure that the services that came online from the crash will be in sync and can start serving requests correctly if other services go down.

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 and the OrderLookup function of Order Service to place an order and retrieve information of an order respectively.

the frontend also maintains a Least Recently Used (LRU) cache to reduce the latency of stock query requests. The cache can be enabled or disabled using *enable_cache* parameter in *config.py.* The cache stores the lookup information of the stock until it is either invalidated or evicted. To ensure consistency, the frontend subscribes to the invalidate requests from the Order service. The stock is removed from the cache upon receiving an invalidated request.

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.
- **OrderLookup:** The orderlookup method takes the order id to be looked up and checks in the stockorder_db. If it finds an order with a given order id, it will return the stock name, quantity traded, trade type (BUY or SELL) and the status (value 1). If the order is not present in the database, we just return status=0. If the request execution is failed, we return -1.
- **StreamDBUpdates:** This function keeps on sending trade information to the frontend to invalidate the cache for that particular stock. The frontend subscribes to these changes on start. Whenever a trade is processed successfully, that stockname is sent to the frontend and the cache is invalidated.
- **IsAlive:** This function is used to check if a service is alive or not. Whenever a service receives this request it replies with a boolean is_alive=true. This shows that the service is up and running and serve any further requests.
- **SetLeader:** This method is used to set the elected leader id. The frontend services elects a leader and once a leader is selected, it will broadcast the id to all the order replica services.
- **GetLeader:** This method will send the leader id to the requested service. The response will have leader_id.
- **SyncOrderRequest:** This method is used to sync the latest processed trade request at leader service. The leader service calls this method with orderid, stock name, trade type and quantity. The recipient service stores the order to the local service db.
- **SyncOrderDB:** This method is used to sync DB data with a service that just came from crash/started.This will take the transaction id as input and send all the orders with order ids greater than this id. The data is sent in a stream fashion.
- **replicate_order:** This method will send the latest successfully processed data to all the other replica services.

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. After sending all the requests it validates all the orders by placing order lookup requests.

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?stockname=<>', the frontend service returns the price and volume of the given stock name from the cache if it is present in the cache. Otherwise, it 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. If the path is '/orders?order-number=<>', the frontend service sends a order lookup request containing order number to the order service. The Order service returns the information of the order if the order is present in the database.
8. The response from the backend services is converted to a JSON response.
9. The JSON response is returned as an HTTP response to the client.
10. 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.

AWS Deployment Details

All microservices are deployed on an AWS *m5a.large* EC2 instance. The public dns of the instance is *ec2-3-238-71-234.compute-1.amazonaws.com*

References

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