Stock Matching

Design Document

System Architecture:

End users: Web App. Input order information, and send object with type (uid

Request Handler API: Possibly Lambda or VM/ECS cluster, performs simple updation of ticker document on DB, then performs all possible transactions. Can implement publishing messages to AWS SQS, which can then be picked up by

DynamoDB: Decentralized object storage, focusing on fast read/write performance. Configured to get consistent writes.

Matcher: Lambda/ECS cluster that

Data Design:

The payload for the request by the end user web app will be in the format (**str: ticker, long: timestamp, str:uid, int: value)**

The buy list, buy_user list, sell_list and sell_user lists can be separated using the unique ticker id, and Stored in DB in following format:

```
{
    ticker_id: [[buy_list], [buy_user_list], [sell_list], [sell_user_list]],
    ...
}
```

Interface Design:

The end user sends a request to the API gateway, which requests data for the relevant ticker from the DB, and depending on the ticker it modifies the buy/buy_user or sell/sell_user lists, finds matches, and updates record on database.

Security Design: secured tunnel for endpoint to external users;

Performance Design

An Autoscaling cluster should be enough to process the given throughput of 100k messages/sec, considering the read latency for DynamoDB is a few milliseconds.

Requirements:

- Fault Tolerant / Highly Available.
    - Achieved through VM instances in cluster performing independently, with redundancy.
- Modular.
    - Achived through decoupling several systems in our infrastructure.

Possible Improvements:

- Implementing notification mechanism for end user
- binary search adds/reads for all instances of reads/writes from user/buy/sell lists.

APIs:

/order : POST : Posts an order to database

/get_orders_ticker: GET : Gets all orders for a ticker

/get_orders_uid : GET : Gets all standing orders for a user

/cancel_order : POST : Cancels an order by its value, uid, ticker and order type.