Distributed Electronic Stock Exchange

Haneul Shin
Department of Computer Science
Harvard University
Cambridge, MA 02138
haneulshin@college.harvard.edu

Catherine Yeo
Department of Computer Science
Harvard University
Cambridge, MA 02138
cyeo@college.harvard.edu

May 7, 2023

Abstract

We present an implementation of a distributed electronic stock exchange. The exchange allows both human traders and automated market makers to create accounts and make trades with each other on the exchange server. The proposed and constructed distributed system leverages a streaming RPC and heartbeat mechanism to provide scalability, reliability, persistence, and fault tolerance. The system design takes into consideration several key functionalities, including real-time access to market and user data, fair trade matching algorithms (quantified by price-time priority and the RegNMS Order Protection Rule), and remote server-server and server-client communication to ensure an efficient and easy-to-use electronic stock exchange that can handle large-scale trading volumes. We also evaluate our system on the basis of unit tests for accuracy and market metrics, as well as general testing for speed and latency.

Our code can be found on GitHub.

1 Introduction

Electronic stock exchanges are a natural application of distributed systems. There the exchange itself, which is the centralized server that allows communication between different clients that wish to make trades with each other, as well as the traders, who would like to make trades with each other through communication with the exchange as well as access real-time marketdata regarding the open orders in the market and information on their own positions in securities.

Hence, we present an implementation of an electronic stock exchange. There are three players in this market, which we briefly outline here and describe the implementations of in greater detail in Section 2.

1.1 Three Players in the Market

1. **Exchange server**: The exchange server is the centralized server that acts as the electronic exchange. The sever maintains connections with clients, provides them real-time access to

market data, receives trade orders from clients, matches and executes trades, and sends log messages back to clients.

2. **Trader client**: The trader client allows a human trader to authenticate and log in, and to make manual trades by interacting with an interface. The client also allows the trader to access real-time market data as well as their own positions in securities.

3. **Market maker client**: The market maker client allows traders to generate an automated market making bot that sends their desired trade orders to the market.

1.2 Preliminaries: Financial Terminology

To contextualize our system, we begin by explaining necessary financial context and vocabulary in brief.

An *order* is a request to buy or sell a security at a specific price and size. When an investor places an order, it is sent to a trading network, such as an exchange or broker-dealer, where it is matched with a corresponding order from another party, known is the *counterparty.*

A *trade* occurs when two parties agree to buy and sell a security at a specific price and size. Trades are executed when orders are matched on a trading network (the exchange server in this project's case).

Importantly, an order is not a trade. When an order is matched with another order, then it becomes a trade, and is no longer an order since the request to buy or sell has been satisfied.

The *size* of an order or trade refers to the quantity of a security being bought or sold. The size can be specified in terms of the number of shares or the value of the order. The *price* of an order or trade refers to the amount for which a security is bought or sold, specified in a given unit of currency (in our case, USD).

The *dir*, short for *direction*, refers to the direction of an order or trade. An order can be either a buy order (an order to buy a security) or a sell order (an order to sell a security). Note that a trade is then a transaction in which one party buys and the counterparty sells. The *sgn*, or sign, of a trade is a numerical representation of the direction of a trade. A buy order corresponds to a `sgn` of 1, and a sell order corresponds to a `sgn` of -1.

Open orders refer to buy or sell orders that have been posted but not yet been matched with a corresponding sell or buy order, respectively.

An *order book* is a table of all the buy and sell orders for a particular security on a trading venue, sorted by price, and under the hood, time. Users can see all prices and the corresponding cumulative sizes of open orders at those prices in the market, sorted in decreasing order of competitive price. Under the hood, for a given price, the trading network also accounts for sorting by the time at which different clients posted their orders. The order book shows the demand and supply of the security at different prices and can help traders make informed trading decisions.

Positions refer to an investor's holdings of each particular security, expressed in the number of shares.

2 System Design: Server

The server stores, updates, and renders the below dictionaries of market data and user data, allows for remote server-server and server-client communication, and implements a trade matching algorithm that satisfies price-time priority and the RegNMS Order Protection Rule to enforce the fair matching of trades.

2.1 Market Data

2.1.1 Open Orders

For each `dir` in [`buy`, `sell`] and `symbol` in `symbols`, `open_orders`[`dir`][`symbol`] is a *sorted* dictionary mapping `price` to a queue of open orders. Using a sorted dictionary ensures that we can easily retrieve the most competitively priced orders at any given moment in logarithmic time. The queue is represented as follows:

$$\texttt{open\_orders}[\texttt{dir}][\texttt{symbol}][\texttt{price}] = [[\texttt{username}_1, \texttt{quantity}_1], \ldots, [\texttt{username}_n, \texttt{quantity}_n]]$$

This a queue of the open orders in direction `dir` in `symbol` at `price`. Each element of the queue represents an individual order: the user that made the order and the size of that order. The queue stays sorted by time priority. The sorted dictionary structure for the prices, together with the queue for the individual orders, allows the server to efficiently enforce *price-time priority* and the *RegNMS Order Protection Rule*, both of which are described in Section **??**.

2.1.2 Order Book

For each `dir` in [`buy`, `sell`] and `symbol` in `symbols`, `order_book`[`dir`][`symbol`] is a *sorted* dictionary mapping `price` to the total quantity of shares available at that price. Once again, using a sorted dictionary ensures that we can easily retrieve the most competitively priced orders at any given moment in logarithmic time. The data structure represents the total quantity as follows:

$$\texttt{open\_orders}[\texttt{dir}][\texttt{symbol}][\texttt{price}] = \texttt{size}$$

Importantly, note that while the `open_orders` is private information that is only known to the server (the identities of the entity that posted an order most remain anonymous, by law), the `order_book` is the public order book that can be accessed by anyone. In other words, the server computations that match trades utilize `open_orders`, whereas the clients are given access to `order_book`.

2.2 User Data

2.2.1 Positions

For each `user` in `usernames` and `symbol` in `symbols`, `positions` maps `user`, `symbol` to the user's position in that symbol:

$$\texttt{positions}[\texttt{user}][\texttt{symbol}] = \texttt{position}$$

The position is stored in shares if `symbol` is a security, and in dollars if `symbol` is USD. Thus, `positions` is a nested dictionary, which again allows for logarithmic updates and retrieval.

2.2.2 Messages

For each `user` in `usernames`, `messages` maps `user` to the queue of the user's messages, which are sent to the user by the server when orders are posted or when trades occur, i.e.

$$\texttt{messages}[\texttt{user}] = [\texttt{message}_1, \ldots, \texttt{message}_n]$$

2.2.3 User Status

For each `user` in `usernames`, `user_status` maps `user` to a boolean value indicating whether they are online or not.

2.2.4 Passwords

For each `user` in `usernames`, `passwords` maps `user` to their password. We have added criteria such as minimum length and usage of different types of characters (lowercase, uppercase, numbers, special characters) to greater security in password selection.

2.3 Functionalities

2.3.1 Server-Server Communication

There are three replications of the server, which ensures 2-fault tolerance in the case that any two of them go down. To simulate a process for leader election, each server sends a hearbeat to the others, so that each server can detect whether the other two are currently active. If the leader is not heartbeating, then one the other of the two leaders with the least server id is set as the new leader. Once the new leader is chosen, if any client tries to send a new request to the server, it will be notified of the server disconnection and will be connected to the new leader. The client will also receive a prompt to repeat its most recent message that was directed to the old leader.

To ensure 2-fault tolerance and persistence, every time a request is received from the client that can cause a change in either user or market data, all dictionaries storing server data and user data are sent to each of the other active servers using `send_server_data_to_servers()` as well as are saved *locally* using `save_data_locally()`.

First, the local updates ensure persistence on any given machine: in the case that it is the only machine remaining, then any updates must occur through that server, so saving locally suffices. Second, sending the data to other servers preserves 2-fault tolerance: if there are multiple servers, then sending the server data from the leader to each non-leader for each update ensures that the other machines also have up-to-date information, in the case that they are later assigned as the leader. This functionality was tested by running three leaders in tandem and halting one and/or two of them.

2.3.2 Server-Client Communications

All server-client communications are also executed entirely through gRPC, so that remote servers and clients can transfer information in the absence of a shared machine. In particular, we use a gRPC wire protocol that allows clients to send requests to create an account, log in, initialize and render their orderbook, or post orders to make trades. The clients then receive real-time updates on the state of the order book, as well as a real-time message log notifying them of orders that have been posted, trades that have been matched, whether a leader has been changed, and each time the order book is updated.

We discuss the details of the client-side in Section 3.

2.3.3 Trade Matching Algorithm

Each time a user posts a trade in the form of a buy or sell order, the server runs a trade matching algorithm that receives orders posted from clients and matches any existing trades, satisfying both price-time priority and the RegNMS Protection Rule, and posts the remaining unmatched trades to the market, as follows.

1. While the order has remaining unmatched shares, compute `best_price`: the best price currently offered on the order book is (this preserves *price*-time priority). This is done in logarithmic time since `open_orders[symbol][opp]` is a *sorted* dictionary with prices as keys.
    (a) If that price is not sufficiently competitive, in that it is not equal to or a better price than what the current user is asking for, then proceed to Step 3.
    (b) Otherwise, while there are open orders remaining at `best_price`, iterate over them in order (this preserves price-*time* priority). For each such order, while there are unmatched shares remaining in the original user's order, do the following.
        i. Determine `cur_size`, the size of the trade with the current counterparty (the person the user is currently matching trades with), which is the minimum of the remaining unmatched shares of the original order and the shares that the counterparty has left in the order.
        ii. Execute this trade for the *counterparty* at the price that was originally offered in the counterparty's order (this enforces RegNMS Order Protection). Send the coun-

terparty a success message that the trade was executed. (All trades made by the current user with each counterparty are later executed at once below).

2. If any of the shares of the user's order were matched in Step 1, determine the average price that the trades occurred at and execute the full size of all the trades at once. Send the user a success message that the trades were executed.

3. If there are remaining unmatched shares of the user's order, post a trade order for this number of shares for the price that the user originally requested.

2.3.4 Price-Time Priority

The above trade matching algorithm satisfies price-time priority, which is an essential feature of fair trade matching. Specifically, for a given posted order, if there is more than one opposing order (a sell order for a buy order and a buy order for a sell order), then the counterparty orders to trade with are matched as follows. First, orders are given priority by *price*: the most competitive price receives first right to the trade, and any shares that can be matched are matched before moving onto the next order. Next, for a given price, orders are given priority by *time*: the order that was posted first is matched with first, before moving on to the next order.

2.3.5 RegNMS Order Protection Rule

The RegNMS Order Protection Rule states that a trader who posts an order should receive the best possible price available. For instance, if trader A posts an order to buy 100 shares of AAPL for a price of \$170.50 per share, and a trader B already had an open order to sell 50 shares of AAPL for a price of \$170.35, which is less than trader A was offering to pay, then if this is the most competitive price, then A's trade will first be matched with trader B for a price of \$170.35 per share instead of the requested \$170.50. The remaining $100 - 50 = 50$ of A's shares will be matched with the next most competitive open order, again sorted by price-time priority. If there are no more orders to be matched with, the remaining unmatched shares will be posted in an open order, this time at A's specified price of \$170.50 per share.

3 System Design: Clients

We have constructed two types of clients: 1) the trader client, which renders a graphical user interface for a human trader to post orders on, and 2) the market maker client, which generates automated trades via the command line.

Both types of clients receive real-time data from the server that inform them on changes in leaders, updates on orders that have been posted and trades that have been matched, and most importantly, updates to the order book as they happen. Each of the clients continuously listens for these updates and performs.

Specifically, the client runs a thread that calls the `listen_for_messages()` function. The function continuously listens for one of three types of messages:

1. Leader update: The client then updates the leader set in its local memory and interacts with the new leader.
2. Order book update: If the order book has been updated (which occurs on the order of hundreds of milliseconds), the client then updates its internally stored order book, so that the up-to-date order book is rendered in real-time.
3. All other messages: These messages include response messages regarding successful posted trade orders, matched trades, as well as position limit warnings. The client will display these messages in real-time to its message log.

These real-time updates are essential for the clients to render rapid real-time marketdata in their order books to make up-to-date decisions about their trades.

3.1 Trader Client

The trader client offers the following functionalities:

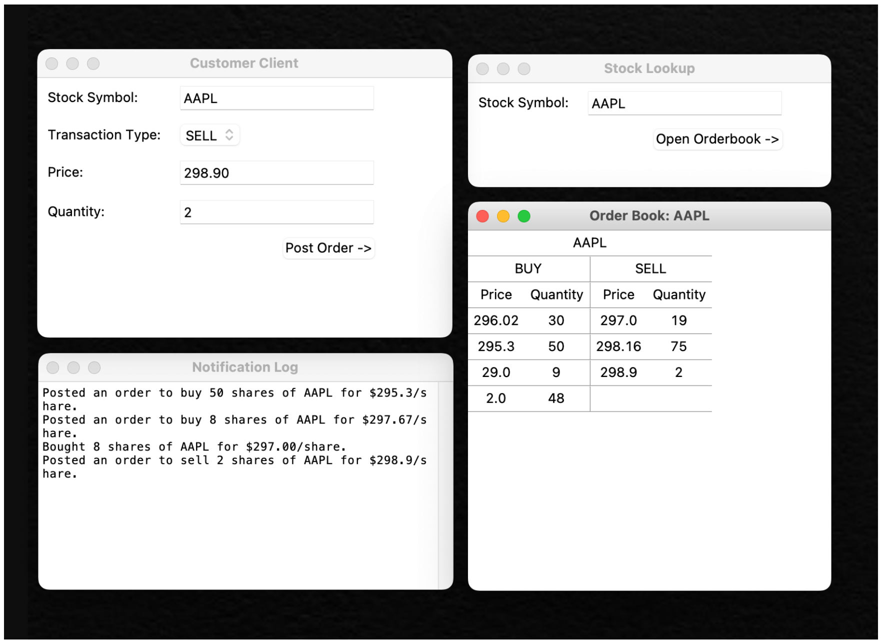
- Registers an account, or authenticates an existing account and logs in.
- Posts trade orders.
- Renders the order book (for any inputted symbol) and trade notification/message log in real-time.

As we can see in Figure 1, on the top left we have our interface to post new trade orders where a trader inputs their desired stock symbol, price, and quantity of shares, while also selling their transaction type (buy or sell). Once the trader hits the "Post Order" button, the order will be logged in the "Notification Log" window and added to the "Order Book" window in real-time. There is also a stock lookup window to launch an order book view for each stock symbol.

3.2 Market Maker Client

The market maker client operates similarly to the trader client, but adds the important functionality of being able to create an automated market making bot that generates trade orders continuously. Using a similar UI to that of the client, the user can input the desired security, a price to act as a fair value to center their trading on, a size, and whether they want to generate markets for only buy orders, only sell orders, or both. The market maker client then generates a bot that continuously generates the specified trades in batches of 10 trades for each specified direction. The market maker client thus allows for the simulation of a real market in which trades are constantly made in both directions continuously.

As with the trader client, the orders are sent to the server, where they are processed and matched and the order book is updated.



Stock Symbol: AAPL
Transaction Type: SELL
Price: 298.90
Quantity: 2

Stock Symbol: AAPL

AAPL

BUY		SELL	
Price	Quantity	Price	Quantity
296.02	30	297.0	19
295.3	50	298.16	75
29.0	9	298.9	2
2.0	48		

```
Posted an order to buy 50 shares of AAPL for $295.3/s
hare.
Posted an order to buy 8 shares of AAPL for $297.67/s
hare.
Bought 8 shares of AAPL for $297.00/share.
Posted an order to sell 2 shares of AAPL for $298.9/s
hare.
```

Figure 1: Screenshot of the trader client UI.

4 Evaluation and Testing

Evaluation of our system is separated into discussions of technical and market features.

4.1 Technical Features

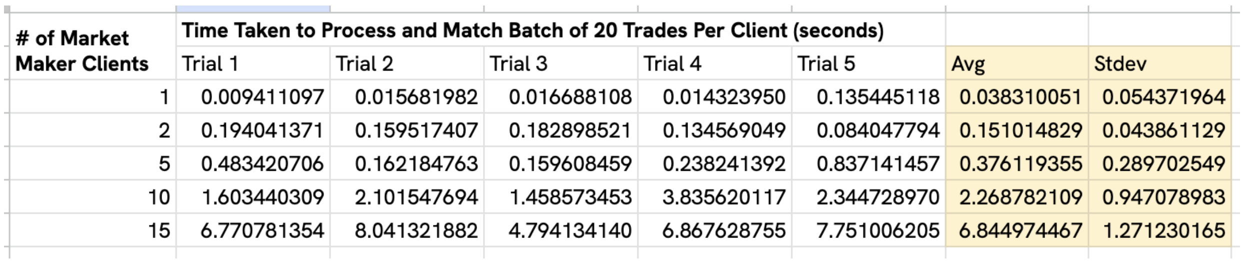
4.1.1 Unit Tests

Our unit tests (found in `testing.py`) assesses the correctness and evaluates important cases (especially edge cases) of different key steps of our process, including:

- Account handling: valid account creation with unique new username, valid user login process, password construction that satisfies set criteria.
- Orders and trades: order posting logic, trade matching logic (which specifically tests across a comprehensive range of different scenarios, edge cases, price-time priority, RegNMS, and average pricing).
- Distributed components: heartbeat mechanism initialization, election leader finding process.

4.1.2 Latency and Speed

To benchmark the speed performance of our server-client interactions, we used our automated market maker clients to conduct experiments (found in `evaluation.py`) evaluating the time taken to process and match a batch of 20 trades generated per market maker client. This was done while varying the number of market maker clients operating in parallel (specifically 1, 2, 5, 10, and 15 clients). Then, we ran each experiment for 5 trials to compute the average and standard deviation. The full results table can be seen in Figure 2.



# of Market Maker Clients	Time Taken to Process and Match Batch of 20 Trades Per Client (seconds)						
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Avg	Stdev
1	0.009411097	0.015681982	0.016688108	0.014323950	0.135445118	0.038310051	0.054371964
2	0.194041371	0.159517407	0.182898521	0.134569049	0.084047794	0.151014829	0.043861129
5	0.483420706	0.162184763	0.159608459	0.238241392	0.837141457	0.376119355	0.289702549
10	1.603440309	2.101547694	1.458573453	3.835620117	2.344728970	2.268782109	0.947078983
15	6.770781354	8.041321882	4.794134140	6.867628755	7.751006205	6.844974467	1.271230165

Figure 2: Table displaying experimental results evaluating the time taken to process and match a batch of 20 trades per market maker client when varying the number of market maker clients operating in parallel.

In our experimental results, we observe that the processing time scales as the number of market maker clients increases. This intuitively makes sense because with more market maker clients, more trades are being generated and posted, and thus more time is spent server-side to match the larger volume of trades. The standard deviation in processing time also increases as the number of market maker clients increases; with more trades being generated, there is a higher variance in what happens among the trades (e.g. 20% of trades being matched and 80% of trades being matched are both possible scenarios, but will differ drastically in total trade processing and matching time).

Another evaluation of latency and speed we considered carefully was the process in how we constructed and optimized the clients – both trader client and market maker client – to ensure and render real-time updates (e.g. for the order book). For example, our order book is stored as a dictionary; we iterated through many versions of how we saved, persisted, and updated the order book with methods such as `json.dumps`, pickling, updating the order book only when a modification to it has been detected, etc. in order to improve towards real-time updates and renders. We started out at updating at intervals such as every 2 seconds and continued to optimize downwards to improve our speed.

4.1.3 Server-Server and Server-Client Communication

All server-server and server-client interaction was written through gRPC to ensure that remote servers and clients can communicate with each other in the absence of a shared machine. This feature was tested by connecting to remote AWS servers and ensuring that all interactions ran as anticipated.