Building a Distributed Stock Exchange

Albert, Dean, Emeka, Gianni, and Victor

Introduction

In modern times, financial markets have experienced a rapid transformation due to the advent of modern technologies which have enabled more efficient and secure financial transactions. Distributed systems have gained significant attention in the world of stock exchanges due to their potential to be both more scalable and more reliable, factors that are critical to a stock exchange's success in providing the best platform for users to buy and sell financial assets. In this context, we built a decentralized stock exchange system that provides a friendly, fast, and reliable user experience for buying and selling stocks that has persistence, replication, and scalability.

Our code and its README are available in a public GitHub repository at https://github.com/alzh9000/cs262-final-project. We implemented the vast majority of our system and tests using Python, with some HTML and CSS for the front-end. Furthermore, a slideshow and demo describing our system are available at CS 262 Final Project Presentation 2 . These slides are especially helpful in the "Background & Motivation", "Our System", and "Why does this matter" sections for achieving better understanding of our project; for brevity, we do not repeat some of the content from those slides here.

Design Decisions

Architecture

Our architecture consists of 4 types of components: exchange, institution, client, and broker.

1. The exchange acts as the server for our distributed system and handles most of the finance logic, such as hosting and updating the orderbook, executing orders, and more. Using Paxos, we can replicate the exchange to have N replicas of the exchange running simultaneously in total, for any positive integer N, and our specific implementation modifies Paxos so that our stock exchange is N-1 fault tolerant: as long as 1 of the N servers/exchanges is alive and running, our stock exchange still works.
2. An institution directly connects to the exchange to trade and communicate requests such as sending orders, canceling orders, getting the current state of the orderbook, etc. The

institution represents trading firms, banks, and other financial institutions that are sophisticated users of the stock exchange which can directly connect to a stock exchange in the real world, without the need for a middleman.

3. A client is like an institution in the sense that a client is a user of our stock exchange who wants to trade financial assets, except unlike an institution, a client is unsophisticated and thus must communicate with the broker instead of directly with the exchange: clients represent "retail investors" in the real world. A benefit to the client of this is that communicating with the broker is easier than directly with the exchange, and the broker can provide additional convenient services to the client. This is like how in the real world, retail investors use brokers like Robinhood to trade stocks instead of directly interacting with an exchange like the New York Stock Exchange.
4. The broker acts as a middleman between the clients and the exchange. The broker can charge clients a fee for their services while also potentially providing clients extra features. The broker represents real life stock brokers. For example, Robinhood provides the service of fractional shares to its clients.

Paxos

We chose to use Paxos as our distributed consensus algorithm to achieve replication in our stock exchange system. Paxos provides fault tolerance by accepting a commit if the majority of the processes have accepted it, usually providing (N-1) / 2 fault tolerance. However, in our implementation, we decided to only accept commits if all of the processes accept them, which provides N-1 fault tolerance. While this approach ensures that the system remains running as long as at least one server is operational, our reliability testing has shown that this approach can put significant strain on the system and may not scale well when there are many servers with many clients connected to them. We observed no issues when we had client bots connected to a system of up to 5 servers for the exchange.

Paxos is an asynchronous algorithm that does not rely on time synchronization or assumptions about message delivery timing, making it suitable for use in environments with unpredictable communication delays. The algorithm has been rigorously proven to be correct, ensuring that consensus is always reached as long as a majority of nodes are functioning and can communicate with each other. Moreover, Paxos can be adapted to scale with the size of the system, making it suitable for large-scale distributed systems.

Other choices for distributed consensus algorithms include Raft, which is designed to be easier to implement than Paxos and provides similar fault tolerance and consistency guarantees. However, Raft can sometimes be less efficient than Paxos due to additional message exchanges in its leader election process. We decided to use Paxos over Raft because we think Paxos is more efficient and we have more experience with Paxos from class.

Order Book

Our order book follows price-time priority rules and relies on Python's heapq module, datetime module, and deque from the collections module. We use heaps (min and max) because of their O(log n) runtime and O(1) peek; using heaps to store bid and ask orders ensures that the highest bid and lowest ask are always at the top of their respective heaps. The order book allows for the addition of new orders, cancellation of existing orders, and order matching based on price-time priority rules. It also supports a margin/shorting system, allowing users to have negative amounts of balance or stocks.

gRPC

We used gRPC (gRPC Remote Procedure Call) to handle communication between computers over a network. gRPC is a high-performance, open-source, universal RPC framework that allows developers to build distributed systems more efficiently. gRPC uses HTTP/2 for transport, Protocol Buffers as the interface description language, and provides features such as authentication, bidirectional streaming, and flow control. In our decentralized stock exchange system, we used gRPC to facilitate efficient and secure communication between the various components, such as the clients, brokers, and exchanges. By using gRPC, we were able to take advantage of its benefits, including better performance, low latency, and high efficiency compared to other communication protocols such as REST.

To implement gRPC in our system, we first defined the service interfaces and data structures in Protocol Buffers (protobuf) format. This included specifying the messages for placing, canceling, and updating orders, as well as defining the RPC methods for interacting with the order book and the exchange. After defining the protobuf files, we used the gRPC tools to generate the server and client stubs in our preferred programming language. Next, we implemented the server-side logic for handling the RPC calls, which included processing the incoming orders, updating the order book, and executing trades. On the client-side, we used the generated stubs to make RPC calls to the server and interact with the distributed stock exchange.

Using gRPC, we implemented the following services:

- RegisterUser: Allows users to register with the broker and creates a new user account.

- DepositCash: Enables users to deposit cash into their accounts.
- GetOrderList: Retrieves a list of active orders in the order book.
- SendOrder: Places a buy/sell order for a user.
- GetStocks: Provides information about a user's stock holdings.
- GetBalance: Returns the cash balance of a user's account.
- CancelOrder: Cancels an active order placed by a user.
- OrderFill: Notifies the broker when an order is filled.

Trading Bots

To simulate a realistic trading environment and evaluate the performance and reliability of our decentralized stock exchange, we developed several trading bots with different strategies. These bots acted as autonomous agents that placed, updated, and canceled orders based on their specific trading algorithms. By introducing these bots into the system, we were able to observe the system's behavior under various conditions and stress levels.

The trading bots were implemented using the following strategies:

1. Random Strategy: This bot would place random buy and sell orders at random prices and quantities within a predefined range. This strategy was used to introduce chaos and unpredictability into the system, simulating the behavior of inexperienced traders or market noise.
2. Mean Reversion Strategy: This bot analyzed historical price data to identify potential deviations from the mean price. When the bot detected a significant deviation, it would place a buy or sell order, anticipating that the price would revert to the mean. This strategy allowed us to test the system's ability to handle more sophisticated trading algorithms and cater to more experienced traders.
3. Momentum Strategy: This bot followed trends in the market by monitoring price movements and trading volume. When the bot detected a strong trend, it would place orders in the direction of the trend, attempting to profit from the momentum. This strategy tested the system's responsiveness to sudden changes in market conditions and its ability to support high-frequency trading.

To implement these trading bots, we created a separate module that connected to our distributed stock exchange system via gRPC. The bots continuously monitored market data, executed their trading strategies, and placed orders using the gRPC-generated client stubs. This setup allowed us to evaluate the system's performance under various trading scenarios and gain valuable insights into its behavior and potential limitations.

nFaultStub Class

Initially, we used gRPC's base stub objects for broker-exchange connections; however, when an exchange server went down, the time needed for the broker to recognize this failure, find a currently running exchange, and resend its request was substantial. Ergo, we augmented the gRPC stub class to achieve N-1 fault tolerance with low latency. This class, named nFaultStub, can be used in the same way as gRPC stubs because the class wraps around a primary gRPC stub and uses its attributes. However, it also creates a backup stub that connects to a different exchange server and is consistently pinged to ensure that the connection is active. This setup enables a near-instant switch between the primary and backup connections if the primary server fails, ensuring a smooth transfer of information.

We've also incorporated a loop for connection logic to facilitate sending requests after reconnections. This feature enables the nFaultStub class to resend its request if the first request was sent to an exchange server that has died. As a result, even in the face of failure, the user only needs to send one request.

Frontend UI

We built our frontend using Flask, HTML, and CSS, allowing it to run in a browser and provide a familiar and user-friendly interface. We opted for Flask as our framework since our project is developed in Python, and Flask seamlessly integrates with the various RPC requests that need to be made on the client side. The frontend enables users to login/register, creating an account that is stored in a SQL database and protected using sha256 hashing method. Cookies are used to maintain user sessions as they navigate the website, allowing multiple users to be logged in simultaneously.

Additionally, in a separate tab on the horizontal navigation bar, users can place bids and asks. These actions automatically update the user's balance and display the changes on the website (e.g., if a user places a bid for 50 AAPL stocks at $100, $5000 is deducted from their account). If the orderbook determines a match, the stock order is displayed on each user's stock table involved in the bid/ask match.

Experiments

Speed

Latency

Stock exchange latency is seemingly unimportant, considering most customers do not actually care whether their transaction takes 1 second or 20 nanoseconds to complete. However, for certain institutional funds like market makers, latency is an important factor for their business. The exact reasoning is outside the scope of this paper, but market makers fundamentally rely on low latency to conduct their business on exchanges.

We tested latency by depositing money and sending a bid order to the exchange 1000 times with three clients connected, with the exchange running on the same network. Realistically, we would expect that latency-sensitive customers would be directly connected to the exchange, so we believe this is a fair scenario to test. We found that the mean latency of the exchange was 36.87ms, with a standard deviation of 0.4222ms.

Throughput

Stock exchanges rely on high throughput (in terms of transactions per second or TPS) in order to process a high volume of shares consistently. Since our exchange uses PAXOS in order to synchronize data across individual exchange nodes, we tested the impact of PAXOS on the throughput of our exchange. We measured the throughput of our exchange by running two bots that would buy and sell the same stock over and over through exchanges run on the local network, keeping track of how many orders were processed. This code can be seen in throughput_tests.py. Note that the exact number of transactions yielded by the test will vary depending on the network it is run on.

Having only one node for the exchange yielded an average TPS of 965.10. Two, three and four exchanges yielded an average TPS of 823.22, 770.56, and 485.11 respectively. As we expected, running more nodes for our exchange severely degraded the TPS, nearly halving it as we ran four exchanges versus one. Most of the slowdown appeared to come from failed consensus between exchanges, forcing the client to start a new request. In real life, we would expect this TPS to suffer even more if the exchanges were not run on the same network or were far apart physically, adding latency to the PAXOS functions and degrading throughput further. Overall, PAXOS may only be worth it for the exchange if the number of nodes overall is kept small. Since real exchanges are not kept online 24/7 (only around 7 hours a day), it also bears to keep in mind that the number of nodes required for near-100% uptime should be relatively small, since maintenance can be performed daily.

Reliability

To assess reliability, we evaluated two aspects: the number of failed commits from PAXOS and the time it took for a client to reconnect to a server if it went down and came back up. Analyzing the amount of disagreement in PAXOS outcomes helps us gauge system load. Although this doesn't directly cause incorrect data to be written to persistent storage, a higher number of failed commits may increase the likelihood of errors. To improve system resilience, we introduced an n-FaultStub, allowing clients to connect to two exchange servers simultaneously. This ensures a smoother transition if one server fails.

Paxos Disagreement

To test disagreement, we ran two sets of experiments: 1 set of experiments is when multiple institutions were connected directly to the exchange, and the other set of experiments is when only one broker was connected directly to the exchange, with multiple clients connected to the broker. Then, we measured how often Paxos disagreements occurred in each setting, under multiple configurations, which gave the following results:

# of institutions or brokers	Statistics on # of failed_commits
3 exchanges 3 Institutions (5 sec runtime)	mean = 542.3, standard deviation = 373.8
3 exchanges 3 Institutions (10 sec runtime)	mean = 1572.3, standard deviation = 797.3
3 exchanges 3 Institutions (15 sec runtime)	mean = 2147.5, standard deviation = 1263.8
3 exchanges 3 Institutions (20 sec runtime)	mean = 2809.2, standard deviation = 1598.2
3 exchanges 5 Institutions (5 sec runtime)	mean = 1576.5, standard deviation = 367.5
3 exchanges 5 Institutions (10 sec runtime)	mean = 2760.5, standard deviation = 929.8
3 exchanges 5 Institutions (15 sec runtime)	mean = 5665.9, standard deviation = 1573.3
3 exchanges 5 Institutions (20 sec runtime)	mean = 5588.1, standard deviation = 2386.9
3 exchanges 3 Clients on 1 Broker (5 sec runtime)	mean = 0, standard deviation = 0
3 exchanges 3 Clients on 1 Broker (10 sec runtime)	mean = 0, standard deviation = 0
4 exchanges 3 Institutions (5 sec runtime)	mean = 830.2, standard deviation = 332.3
4 exchanges 3 Institutions (10 sec runtime)	mean = 1756, standard deviation = 369.0

4 exchanges 3 Institutions (15 sec runtime)	mean = 2424.6, standard deviation = 872.4
4 exchanges 3 Institutions (20 sec runtime)	mean = 2874.4, standard deviation = 1193.8
4 exchanges 5 Institutions (5 sec runtime)	mean = 1498.3, standard deviation = 529.3
4 exchanges 5 Institutions (10 sec runtime)	mean = 2983, standard deviation = 594.5
4 exchanges 5 Institutions (15 sec runtime)	mean = 4091.4, standard deviation = 1125.6
4 exchanges 5 Institutions (20 sec runtime)	mean = 5409.3, standard deviation = 967.3

Multiple Institutions Connected To Exchanges

The number of failed commits increases as the runtime and the number of institutions/brokers increase. This is not necessarily a bad thing since Paxos is designed to handle failures and ensure eventual consistency. However, a high number of failed commits can affect the overall performance and reliability of the system. The mean number of failed commits roughly increases linearly as the runtime increases, but it can vary significantly depending on the number of institutions/brokers and exchanges. This suggests that the performance of Paxos is affected by various factors such as network latency, message loss, and the number of nodes in the system. The standard deviation of the number of failed commits can be quite high, indicating that the performance of Paxos can be unpredictable and non-deterministic. These results all make sense based on our expectations from learning about Paxos.

One Broker, with Multiple Clients, Connected To Exchanges

In this case, we had no failed commits with Paxos. This is reasonable since the broker should act as a "load bearer" so clients do not connect directly to the exchange. As a result, since the clients connect to the broker, and only that single broker connects to the exchanges, we expect there to be no disagreements because only that broker communicates with the exchanges, which matches our empirical results.

Testing Reconnections

In our decentralized stock exchange system, we also conducted performance tests to ensure fast re-connection of a broker to a backup server in the event that a primary server fails. To do this, we measured latency as the time taken from when a broker makes a gRPC call to the exchange until the corresponding response is received and compared how this latency is different when the primary server that the broker was trying to communicate with fails. On average, the connect time when the primary server does not fail was 0.00642 seconds, while the connect time after a failure of the primary server (so the gRPC call is automatically sent to a backup server instead) was 0.00859 seconds. We see that a failure of the primary server does not increase latency by much, as desired, because this means our system demonstrated a high level of fault tolerance by automatically and quickly switching to a backup stub when the primary stub encountered a gRPC error. This quick response allowed for seamless communication with minimal delay. This client-side fault tolerance ensured that even if a server was down, the broker could still send requests to another server, maintaining the overall system's functionality and speed.

Well-orderedness

We think about well-orderedness here from the stock exchange's perspective, meaning what we care about the most for well-orderedness is that trades are executed in the correct order. Specifically, we want to follow price-time priority, which is standard for most stock exchanges, meaning an incoming order will execute with the resting/existing order already in the orderbook first based on the best price, and second based on the earliest time: intuitively, we break ties in price by earliest order time. So, for example, if Bob and Alice both have existing bid orders in the orderbook for 1 share of GOOGL at $100, with Bob having placed his order the earliest and $100 being the highest bid price, and there is an incoming ask order for 1 share of GOOGL at $100, that incoming ask order will execute against Bob's bid order because Bob's bid order is the earliest. The way we designed our system to ensure well-orderedness is by having our bid and ask heaps (which make up our orderbook) sort their orders first by best price, then by earliest time, so that the bid or ask order at the top of each heap is always the highest priority order by price-time priority.

Furthermore, our exchange sets the timestamp for each order as the local time on the exchange when the exchange receives the order request. So, what determines whether Bob's order or Alice's order comes first is not which of them sends the order first in real life - what matters instead is when the exchange receives their respective orders. This is accurate to real life stock exchanges too - the timestamp of an order is always from the exchange's perspective, which is why many trading firms care so much about achieving low latency via collocation. These trading firms don't actually need to be placing their orders first in real life, they just need the exchange to receive their orders first, such as by having better connections from their servers to the exchange. This way, we will not have any timestamp conflicts between clients' orders that we can't handle because the exchange can just decide based on its own local time which orders were the ones that came first. Any conflicts between replicated exchanges about order timestamps are handled according to Paxos. We tested this by writing tests which simulated different sequences of orders and checked whether the order execution matched what we expected based on price-time priority from the exchanges perspective. Additionally, we wrote a test to check that what mattered for price-time priority was the timing from the exchange's perspective, not the client's perspective. These tests all succeeded.

Learnings and Discussion

Some of the most challenging aspects of this project included:

- Implementing the Paxos algorithm and ensuring its correctness in the context of a distributed stock exchange
- Designing an efficient and well-ordered order book that adheres to price-time priority rules
- Ensuring the system is scalable and can handle a high volume of transactions
- Developing a user-friendly frontend UI and integrating it with the backend system
- Coming up with interesting trading strategies for our trading bots
- Figuring out how to design and run our experiments in a reliable and consistent manner

Based on our experience, there are several ways we would approach this project differently in the future:

- Explore alternative consensus algorithms, such as Raft, to compare their performance and reliability with Paxos in the context of a distributed stock exchange
    - Additionally, we can try using majority voting for the Paxos algorithm to see if it handles large amounts of clients more efficiently
- Investigate more sophisticated load bearing or balancing techniques to improve the system's scalability and fault tolerance
- Implement additional trading strategies for the trading bots to increase their versatility and adaptability to different market conditions
- Improve the frontend UI's design and functionality to enhance the user experience
- Conduct more extensive testing and evaluation of the system under various conditions to better understand its limitations and areas for improvement

Conclusion

In conclusion, we have successfully built a decentralized stock exchange system that provides a friendly, fast, and reliable user experience for buying and selling stocks while ensuring persistence, replication, and scalability. By implementing the Paxos algorithm, we have achieved consensus among the networked nodes, allowing our system to be fault-tolerant and asynchronous. Our order book design adheres to price-time priority rules, and we have implemented trading bots with multiple strategies to simulate realistic trading conditions. Through extensive testing and evaluation, we have gained valuable insights into the challenges and opportunities associated with building a distributed stock exchange.

Acknowledgements

On a more personal note, we really want to thank the CS 262 team, including Professor Waldo, Varun, and Will, as well as the guest lecturers, for an amazing class and for teaching us so much! We learned so much through not just the lectures but also through the design projects and especially this final project. To whomever is reading this, we hope you have an amazing end to the semester and a happy, joyous summer!