StockNet - A Distributed Trading Competition

John Lee[1] and David Simonetti[1]

[1]https://github.com/David-Simonetti-ND/Stock-Market-Simulator.git

May 3, 2023

Abstract

We designed a system that emulates a distributed trading competition using an artificial market simulation. Individual clients play a gameified simulation of the stock market to compete for the highest net worth. To inject interesting behavior into our system, the market consists of a basket of 5 stocks and clients always receive delayed information.

Our system consists of 4 different units: a stock market simulator, a broker/load-balancer, replicator servers, and clients. Clients connect to the broker using a client endpoint and subscribe to the stock market simulator. The broker is connected to a set of broker-replicators and shuttle requests between clients and replicators. The broker is also subscribed to the simulator with a higher quality data stream.

We evaluate our system using 3 criteria - consistency, throughput, and latency. We measure how our performance change with differing amount of clients and broker-replicator servers. Lastly, since our system was initially designed to be an enjoyable game, we observe the performance of several strategies over time.

Contents

1 Purpose/Motivation **2**
- 1.1 Design . . . 2
- 1.2 Goals . . . 3
- 1.3 Challenges . . . 3

2 Architecture **4**
- 2.1 Definitions . . . 4
- 2.2 Name Server . . . 5
- 2.3 Stock Market Simulator (SMS) . . . 5
- 2.4 Broker/Load Balancer . . . 6
- 2.5 Replicator . . . 8
- 2.6 Clients . . . 10

3 Evaluation **11**
- 3.1 Basic Logic and Error Checking . . . 11
- 3.2 Consistency . . . 12
- 3.3 Throughput . . . 12
- 3.4 Latency and Fairness . . . 13
- 3.5 Simulator . . . 14
- 3.6 Strategy . . . 15

4 Conclusion and Future Work **16**

1 Purpose/Motivation

Our system is designed to emulate a distributed trading competition using an artificial market simulation. Individual users can develop automated strategies or utilize an interactive client to play a gameified simulation of the stock market and compete for the highest net worth. Our game should appeal to those fascinated by market behavior, the mathematically inclined, and those with competitive spirits.

1.1 Design

To enter the game, each user authenticates/registers an account to a broker, initializes the account with a net worth of 100k and subscribes to a live stream of market updates. Based on market information, the user can then implement its own strategy of selecting between five stocks to purchase or sell.

The broker, which is the entity responsible for managing client operations and has the most recent price information, updates the clients positions when it receives buy/sell requests. To do this, the broker utilizes a system of replicators which each store a portion of client information. The actual client requests are carried out by the replicator servers, with the broker serving as a load-balancer in forwarding a portion of requests to them. Each request sent to a replicator is also forwarded asynchronously, so that multiple client requests can be processed by replicators simultaneously. The replicator chosen for each request is assigned using a hash of the username of the client. This improves the overall throughput of the system because multiple client requests can be processed concurrently, ideally balanced between replicators.

A global leader-board is asynchronously updated every minute to display the top 10 clients with highest net worth. If a client chooses to voluntarily leave or disconnects, its current positions and balance are saved by its hash-assigned replicator so that the client may reconnect at any point. Clients can be assured that they will never lose their account information, as their accounts are persistently stored no matter how many crashes may occur in the system.

One purposeful design decision of the simulation is that the client's current information is not perfect. The client will receive information that is consistently out of date compared to the stock broker. This means that the clients will have to deal with stale information when making their buy/sell requests, and this will generate interesting variability on trading strategies. However, every client receives information with the same delay, so there is still a fair playing field between competing clients.

Another facet of the simulation is that we will create clients that have different trading strategies to try and make money. Some might make random buy and sell decisions, while others might try to make predictions based on the historical behavior of a given stock. The different trading strategies, a basket of stocks to choose from, and information delay interplay to paint an interesting picture on how the performance of different strategies are varyingly affected.

Since the stock market actually operates through a very complex system of supply and demand, we drastically reduce the project complexity through a series of reasonable assumptions and simplifications in market dynamics as follows:

1. The market consists of only 5 stocks - Tesla (TSLA), Microsoft (MSFT), Amazon (AMZN), Nvidia (NVDA), Apple (AAPL).
2. Every stock is simulated as follows:
   (a) Load pre-downloaded OHLC (open-high-low-close) minute data from actual markets over the last month.
   (b) For each minute, the price updated at evenly spaced ticks using a Gaussian random walk from the open price to the close price, with the variance set as a proportion of the range.
3. Each stock has infinite volume/liquidity, meaning that an individual's purchase does not impact the market in any way.
4. Buy/sell orders are limited to at-the-price execution, meaning only the amount of desired may be specified and the actual cost/profit depends on the current price of the stock.
5. Only long-side positions are allowed, meaning you cannot go short against a stock.

6. Clients receive artificially delayed market information in order to emulate real delays in information. This also adds interesting dynamics to the trading strategies.

7. Each user is only allowed one concurrent instance of a given username connected to the broker at any time. Users are allowed to have multiple clients with different usernames if they desire to test out different strategies.

8. Because the real life stock market should never "crash", we tolerate that our Simulation and Broker servers are single points of failure, as either crashing would be equivalent to the New York Stock Exchange going down. The system is designed to be recoverable in the event they do crash, but the game will halt until these servers are restarted. In the case that a replicator or client crashes, the rest of the system will continue to function while they are being restarted.

1.2 Goals

We have three main goals our system needs to accomplish. These are ordered in terms of relative importance to the system.

1. Maintain **consistency** of client accounts and service over time. This means that a client should never lose a stock they have purchased or any money sitting in their account from server failure. This also means that the system should be able to recover in the event that any of the servers crash.

2. Maximizing total **throughput** of the broker system. This will be measured as the amount of buy/sell requests the broker can handle for all connected clients per second.

3. Minimizing individual client **latency**. This is measured as the time it takes a client to send a request to the time that the clients receives confirmation that the request was successful. Furthermore, we wish to maintain fairness of client latency. Low-frequency traders and high frequency traders should experience no discrepancy in operation time.

1.3 Challenges

The essential/potential challenges of implementing this project follow from the three goals laid out above

1. The interplay between consistency of throughput/latency was a challenge. Ensuring the consistency of client account, for example maintaining their stock purchases over time, is at odds with having low latency and throughput. This is because consistency is achieved by slow operations which flush to disk. We also needed to strike a balance between ensuring consistency while also enabling high frequency trades. The usage of multiple replicators in the system helped alleviate this tension.

2. Fairness of the system could be tough to balance. We needed to ensure that no preference was given to one client above the others, and that client requests are processed in the order the server receives them. There is also a consideration of access to information. All clients should receive information about stock prices at the same time, and two purchase requests sent concurrently should use the same stock price.

There were other challenges to overcome considering other parts of our overall design.

- Having multiple servers cooperate with each other. Each server has autonomy in its own internal functioning, but must work together to create the stock market simulation as a whole. Making a server wait on a result from another server could grind the system to a halt, and the system needed to be carefully designed to avoid deadlock situations.

- Client operations should be processed in the order they are sent. A client should not have to worry about their buy order being processed before being sending a sell order.

- Delivering large quantities of stock market information to many connected clients. There are 5 stocks in the simulation, and sending out updates to potentially 1000s of clients many times a second could prove challenging on server bandwidth.

- Implementing the time delay between client and broker. The simulation server needed finer granularity updates to distinguish between data sent to connected clients compared to the connected broker. The simulator also needed to temporarily store stock information before the delay ends so it can be sent to clients.
- Implementing various trading strategies to be able to handle stale information cases. Stock market client programs needed to handle the case they perform a buy that was possible according to the client's out of date information but impossible with the broker's fresh info. The client will have to be able to adapt to flaws in its strategy introduced by this fact.
- The usage of replicators in the system presented a challenge as it adds another layer of latency to the system, as well as a fair hashing algorithm so one replicator is not overwhelmed. We also needed a system to handle replicator crashes so that the entire system does not collapse.

2 Architecture

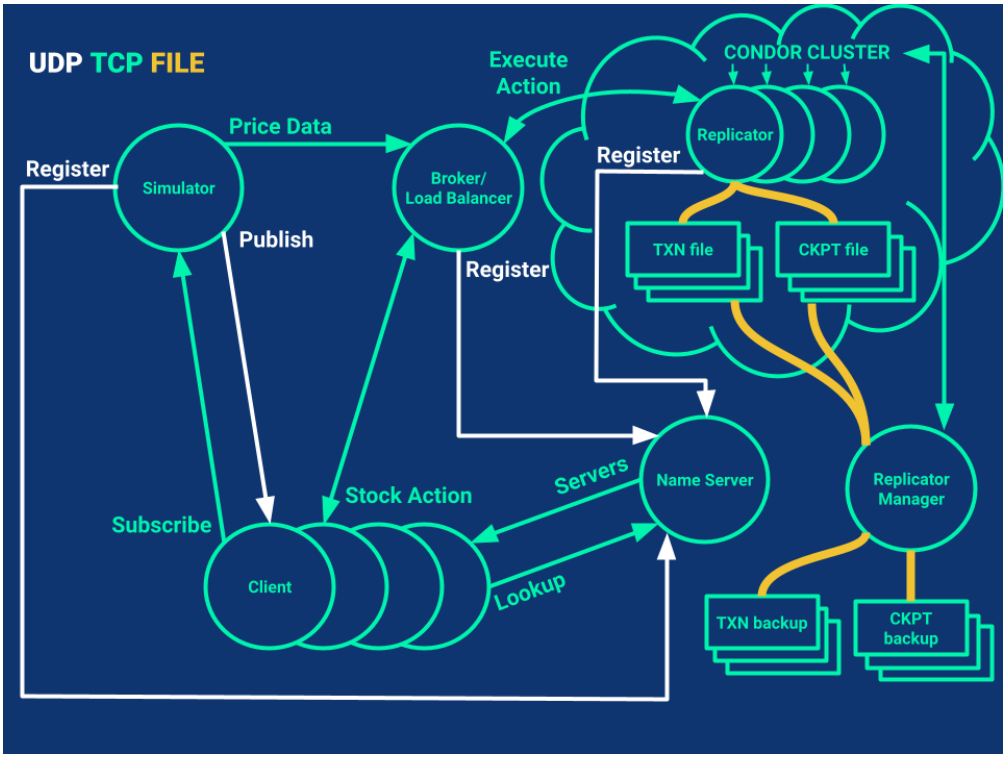

Figure 1: Overview of System

2.1 Definitions

There are four main units that consist our system. A brief overview of each is given in this starting section to provide context, and detailed information about their interactions are covered in following sections.

1. **Simulator**: The simulation server completes the task of simulating the stock market. It uses a publish/subscribe system in order to deliver price information to clients with a specified delay. It also has a direct connection to the broker to provide it with updated stock information. The simulator is located in the top left of figure 1.
2. **Replicator**: The replicator is a server that is used by the broker to handle a portion of client requests and information. For example, if there are 10 replicators in the system, then each replicator will store and perform requests for 1/10 of all clients. Requests are forwarded to the replicator for the broker

via hashing the username of the client and taking the modulus by the number of replicators. Client requests are asynchronously forwarded from the broker on to the replicator in order to be completed, and then returned to the client through the broker once they are finished. The replicator uses a transaction and check pointing system that allows itself to recover from a crash. The replicators are run as condor jobs, and are located in the top right of figure 1.

3. **Client**: The client is the endpoint for the user to enter in the system. It allows the user to connect to the broker and perform register/buy/sell/balance/leaderboard operations. Each user can create as many clients as they want, but they must have unique names. Clients are located in the bottom middle of figure 1.

4. **Broker/Load Balancer**: The broker is connected to all clients and all replicators in the system. It receives incoming client requests, forwards it to a replication server via a hashing algorithm, and then asynchronously completes the given request for that client. The broker is located in the top middle of figure 1.

2.2 Name Server

We use the centralized name server, the ND catalog server, to register the various servers inside our distributed system. The simulator, broker, and replicator servers periodically update the name server every minute using a json UDP scheme, informing it of the server's host, port, type, and project name. Each server will use the same project name but have a unique and different type depending on the function of that server. For example, the one simulator will always have the type stockmarketsim. These unique name that will be further discussed in the later sections. This enables our broker to be found by clients through a HTTP GET request to the name server. It also enables the broker to make connections to the simulation server and the replicator servers.

2.3 Stock Market Simulator (SMS)

The description of the Stock Market Simulator will cover a general overview of the component, details of the simulation, its connection to the broker, and the publish-subscribe system used with clients.

Overview: The Stock Market Simulator is a simple server performs the simulation of the prices of the 5 stocks on the stock market and informs the other components of the system of these prices. There is only every one simulator active at any time in the system, and it always has a fixed type, stockmarketsim, so that the broker and clients can easily look it up from the name server. For a "market" of 5 stocks, the simulator simulates each stock according to the scheme described in the Design section. The simulator has 2 types of connections - a TCP connection to the broker and a UDP Pub-Sub scheme to the set of subscribed clients. The mechanics of these systems are pictured in figures 2 and 3 will be further described in the sections below.

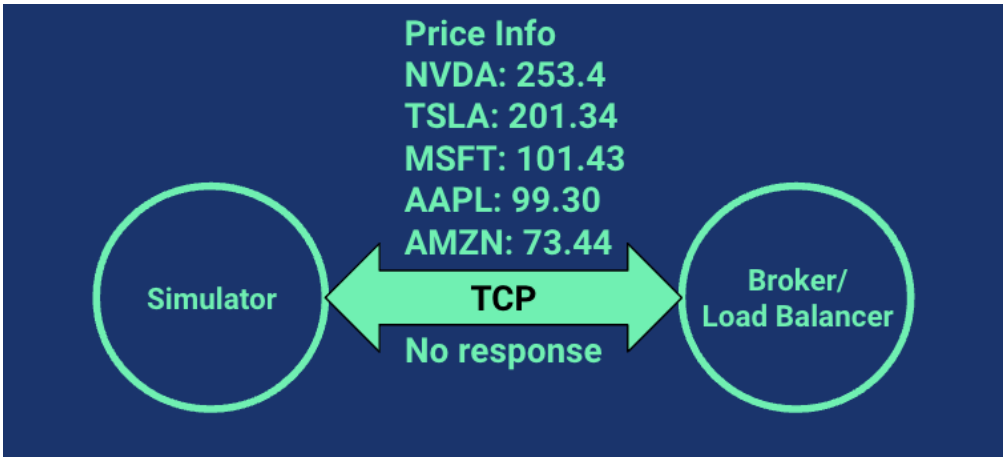

Figure 2: Price Update TCP example between Broker and Simulator

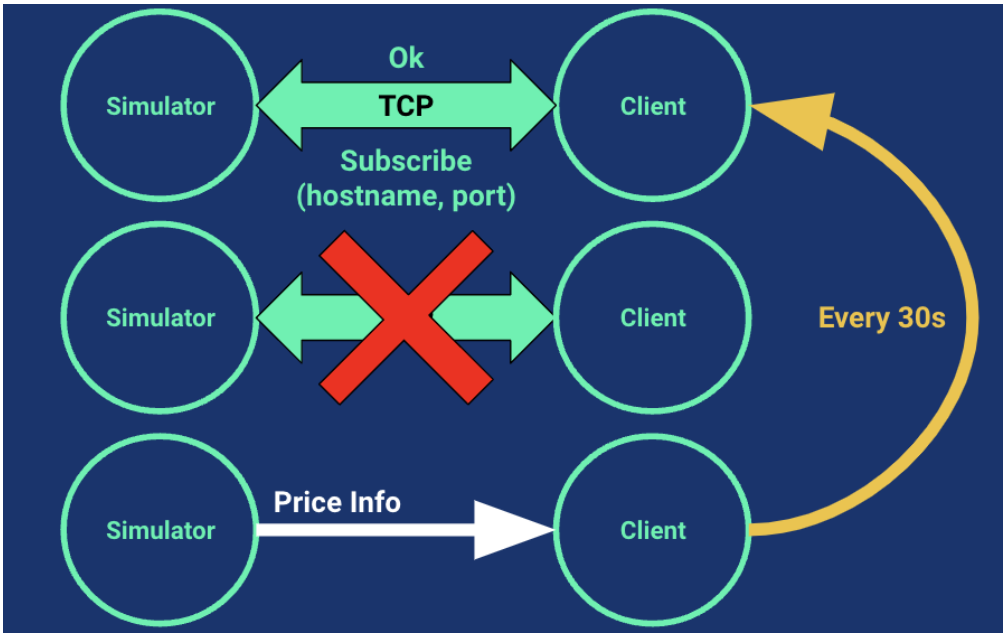

Figure 3: Pub-Sub UDP example between Client and Simulator

Simulation: In the course of simulation, we define two separate rates at which data is processed. First, we set an update rate, or tick, of 10 milliseconds, at which the stock price for every ticker is updated. Similarly, we set a publish interval of 100 ms, where the simulator publishes the most recent tick of stock information to the broker and delayed data of 5 update ticks (50 ms) to a queue of subscribed clients. This delay can be adjusted as desired.

Broker Connection: The simulator's TCP connection to the broker enables a "reliable" source of stock information. On startup, the simulator passively waits for the broker to connect and make a TCP connection. The broker sends a JSON encoded message to the simulator to indicate that it is the broker. If the broker were to crash and be restarted, the simulator will close the dead connection and send stock information to the new broker. The active connection is pictured in figure 2, where the simulator will send information to the broker forever and expects no response. In contrast, data is published unreliably but efficiently using a publish/subscribe system to the clients through a json UDP message.

Publish-Subscribe System: To subscribe to the broker, a client first initializes a temporary TCP connection with the simulator, sending the hostname and port it wishes to subscribe on encoded in JSON. This process is shown in figure 3. The simulator then sends an acknowledgement back letting the client know they are subscribed and closes the TCP connection. After subscribing, clients are added to a queue from which they are removed if they do not resubscribe periodically (30 seconds in our scheme). This unsubscribing allows the simulator to be free of junk clients who have crashed or disconnected. While in the queue, clients are send stock price updates every 100ms via a UDP message consisting of a JSON encoded dictionary of prices.

2.4 Broker/Load Balancer

The description of the Stock Market Broker will cover the following: a general overview of the component, what it does on startup, how it handles incoming client requests, how it completes pending client requests, key data structures, error handling, and the leaderboard.

Overview: The broker/load balancer serves as the front end for all incoming client requests. It is connected to all clients and all replicators in the system. When a client request comes in, the broker forwards this request asynchronously to a hash-assigned replicator server which actually complete the request. Once the request is completed by the replicator, it returns the response to the broker which forwards it back to the client. In each simulation run, there can only be one broker running at one time.

Startup: To start the broker, one must specify how many replicators that the given simulation will be using. On the startup of the broker, it will immediately poll the name server to find out how to connect to

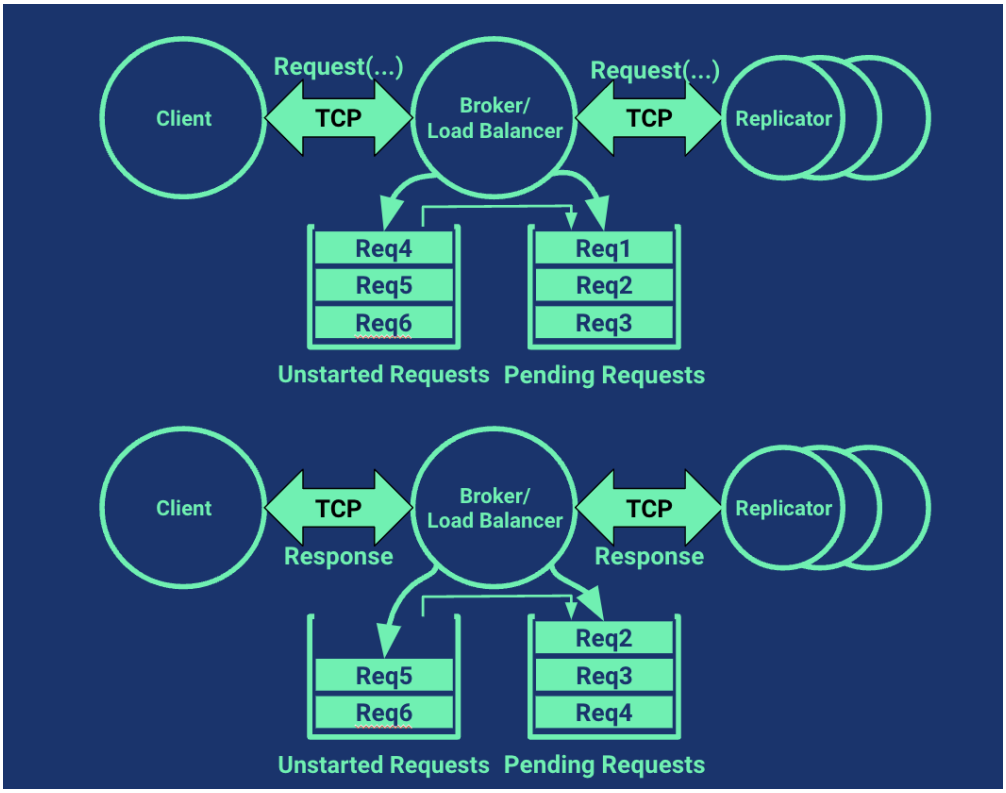

Figure 4: Buy RPC call example between Broker and Client, with load balancing forwarding the requests to different brokers. The unstarted requests and pending requests data structures also appear.

the simulator, and it will keep retrying this process until it gains a connection to the simulator. Afterwords, if the broker was started with a specified n number of replicators, it will try to establish a TCP connection to each of the replicators in order. The name of each is consistent for every run of the system, for if n=3 for example, then the broker will connect to the replicator with id 0, rid 1, and id 2 in that order. If it cannot connect to one of these replicator servers, it will retry until it gains a successful connection. Afterwards, it will send a message to the name server saying that its type is stockmarketbroker and is available for incoming connections.

Handling client requests: After the broker's startup process is complete, it will wait for clients to connect. While it waits, it keeps keeping itself updated on the updated stock market data periodically received from the simulator. Once it receives a client connection and request request, the broker will ensure the request is properly formatted. The details of client requests are covered in the client subsection of the architecture section. After validation, the broker computes a hash of the client's username and takes the modulo by the total number of replicators in the system. The result of this is the replicator server number to which the client's request will map to. For example, if the client's username hash is 13 and there are 3 replicator servers, the request will be mapped to replicator with id 1. After computing this, the broker will asynchronously send off the request to be handled by the hash-assigned replicator. After sending off the request, the broker will move onto handling other client requests, and the original client is left waiting until the broker returns the response from its request. This process is shown in the bottom half of figure 4.

Completing client requests: While the broker is waiting for incoming requests, it will also poll the replicators to see if any request has completed. If a request is ready, the broker will forward the response to the original client. The these response messages are described in further detail in the replicator section. If the client disconnects before the request is returned, the response message is discarded. In addition, the broker only accepts one concurrent request from each client, so clients must wait for the first response before sending another request. If the replicator disconnects or crashes before returning its response, it returns a JSON message to the client indicating that the database server has crashed. If this occurs, the client must

issue another request to check its current balance in order to see if its request went through or not. More detailed error handling and recover will be covered below. The process of completing client requests is shown in the bottom half of figure 4.

Key data structures: The broker maintains a couple of key data structures in order to complete client request. These two structures are shown in figure four, and they are the unstarted requests dictionary and the pending requests dictionary. The pending requests dictionary maps busy replicator ids to the corresponding client requests. For example, if there are 10 replicators in the system, and 1, 3, 5 are currently busy, then the dictionary will have entries for keys 1, 3, and 5 mapping to the current ongoing requests for that replicator. This allows the broker to know if a replicator is currently busy on a request and allows the broker to know where the response should go.

The unstarted requests dictionary maps all replicator numbers to queues. The queues consist of all the client requests which are yet to be started for that replicator id. For example, if clients Bob, Alice, and Tom all map to replicator 2 and Bob's get sent off first, then Alice and Tom will have their requests placed in the queue under key 2 in the order they came in. The broker will then iterate over this data structure every so often and after completing a request to attempt to start waiting client requests. Using the queue structure ensures fairness as each client will eventually get to the front of the queue and get their request serviced. For example, in figure 4 once request 1 was processed by the replicator, request 2 was able to be started as it was next in line for that replicator.

Error handling-broker crash: The broker is semi-stateless, in that it contains state in its unstarted requests and pending requests dictionary but this information can be lost without any consistency problems. In the case that a broker crashes, all of the replicators will still complete the requests currently in the pending requests dictionary, with the response being discarded. All of the unstarted requests will also just be lost and need to be resent. This means that, in the case the broker crashes, is up to the client to poll for its balance between requests to see if its last request finished completely or not. This means that all operations are atomic if the broker crashes, it just forces the clients to check if they actually occured or not. This does mean the broker is a single point of failure in the system, but this is acceptable because as long as it gets restarted after crashing, the system can continue to function with no loss in consistency.

Error handling-replicator crash: In the case a replicator crashes, the broker will accumulate client requests hash-assigned to that replicator in its unstarted requests queue. Once the replicator is reconnected, then the broker can begin iterating over these requests and have them serviced in the order they came in. In this way, request clients make will always be serviced eventually as long as replicators are restarted when they crash. Additionally, in the case that a single replicator goes down in a system with n replicators, its represents only a 1/n loss in throughput of the system, as the other n - 1 replicators can continue to function.

Leader-board Finally, the broker maintains the leader-board for the system. The leader-board is asynchronously updated every minute, iterating through each replicator and using a special json request to poll them for their client information. The replicators will respond with a json encoded dictionary containing client names mapping to their net worth. Using this information, the broker can compute the highest net worth of all players in the system. When receiving a client request for the leader-board, the current leader-board is sent as a response to the client, which means it can be up to 1 minute out of date. The slight staleness of this information is a trade off because computing the leader-board is an incredibly expensive operation. Consider the case where there are 200 brokers, which means that one leader-board operation will generates 200 requests to replicators.

2.5 Replicator

The description of the Replicator will cover the following: a general overview of the component, how it completes requests, request security, replicator consistency, and its usage with condor.

Overview The replicators serve to store all client information and to execute client requests forwarded to it by the broker. Each replicator performs requests for a portion of all clients proportional to the total number of replicators in the system. For example, if there are 10 replicators in the system, each will process roughly 1/10 of all clients and requests. Replicators are hash-assigned clients based on their unique id. The id is given to the replicator on the command line, which is just a number from 0 to (n - 1), where n is the number of replicators in the system. In this way, the usage of multiple replicators allows the system to utilize the disk capacity of multiple different machines as well as greatly increase the overall throughput of the system. Especially considering that buy and sell operations require flushes to disk to ensure consistency,

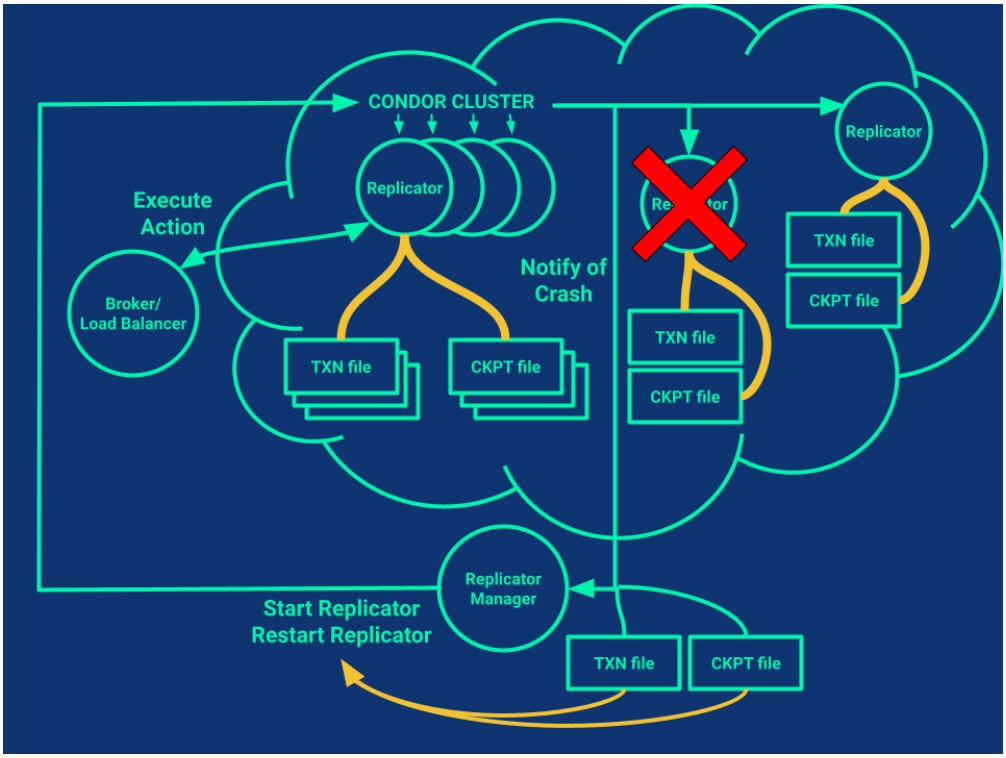

Figure 5: The structure of how replicators are deployed using condor via the Replication Manager. Also visualized is a replicator crashing, condor reporting the crash along with the current state of its txn and ckpt files to the manager. The manager then proceeds to restart that specific replicator from its exact state.

maintaining many replicators in the system enables a massive multiplication of throughput. They also serve as a natural failure domain, where one replicator can cleanly fail and be restarted.

Request completion: The main function of the replicator is to service client requests. Client requests are forwarded to their hash-assigned replicator by the broker. The broker discovers the name of a given replicator based on its unique id. For example, in a system with 3 replicators, there would be a replicator with id 0, id 1, and id 2. All clients that have a username hashing to id 2 would be sent to the replicator with id 2. The main API calls that clients can perform in the requests forwarded to the replicator are as follows. All responses to these requests are sent as JSON encoded strings.

1. Registration Operation
   (a) Clients must register themselves with the system before they can begin to make buy and sell requests. This causes the replicator to create a new StockMarketUser object for that client and enables them to begin trading. If the user already exists, an appropriate error message is return. Otherwise, ok is returned.
2. Buy/Sell operations
   (a) If a buy or sell operation arrives, the broker checks whether the client has enough resources to complete the operation. A success or failure response is sent to the client. Here are two examples of a failure. If a client attempts to buy 100 shares of TSLA at $10 each, but only has $900, an error is returned. Similarly, if a client tries to sell 100 shares of TSLA, but only owns 90, an error is also returned
3. Requesting positions and balance
   (a) Each client's balance is tracked using a hashmap mapping usernames to stock and cash information, so the correct client's positions and balances are sent as a response. The response is sent as

a dictionary with cash and the stock tickers are keys, with the amount of cash they have and the amount of each stock as values.

Security: As a matter of security, clients must include a password along with their username. This password is checked every time the client attempts an operation, and if it does match the password provided at account creation then the request is denied. An appropriate error message is returned in this case.

Consistency: The registration and buy/sell actions are kept consistent by the usage of a transaction and checkpointing system. Every time an operation comes by that changes the balance of a client's account, it is first written to a transaction file which is flushed to disk to maintain a permanent record of the operation. In addition, every 100 operations this transaction log is compressed into a checkpoint file that contains the entire current state of the replicator in a file. In this way, if the replicator were to crash, it can be restarted with the same replicator number and rebuilt the exact same pre-crash state using the transaction and checkpoint log.

Usage with condor: To encourage ease of use in the system, we have created a replicator manager service which interfaces with HTCondor jobs in order to be able to easily create up to hundreds of replicators. The replication manager is separate from replicators and only serves to manage the replicator jobs deployed on condor. This process is shown in figure 5, with the replication manager on the bottom and the condor replicator jobs shown on the top part of the figure. The manager begins by starting up a number of replicators via condor jobs each with their own id in series, e.g. id 0, id 1, etc. Each of these jobs is started with a log file that reports on the status of that replicator job.

Once they are started, the replication manager will continually watch these log files. If the log reports that the job has terminated in any way-that it has crashed or otherwise-then the replicator manager will step in and restart the replicator with the same replicator number. In addition, when the replicator crashes, condor will bring back the current transaction and checkpoint files to the replication manager. The replication manager then restarts the replicator with these same files so that it continue in the exact state that it crashed in. This is shown in figure 5, as the replicator job that has the giant red X over it hash crashed and the replicator job all the way on the right of the screen was started to replace it. In this way, the replication manager allows for any number of replicators to crash as many times as they want in the course of the simulation. Of course as a result, the replicator manager becomes a single point of failure, but it can just be easily restarted if that were to happen.

2.6 Clients

The description of the client will cover the following: a general overview of the component, how it receives stock prices, how it makes requests, and error handling.

Overview: Clients serve as the endpoint for the player of the stock market game, enabling the user to interface with the system. Each client connects to the Name Server via an HTTP connection, querying for the broker and the simulator. The client first establishes a TCP connection with the simulator and subscribes to stock updates by sending it a socket that it will listen on. A background thread is also started to continually perform the resubscribe process every 30 seconds. The resubscribe time is randomly chosen and slightly less than 30 seconds so that not every client will ask the simulator to resubscribe at the exact same time. The client then proceeds to make a TCP connection to the broker, retrying until a connection made.

Receiving stock prices: Once the client is subscribed to the simulator, the client will begin receiving a stream of json UDP packets containing stock prices from the simulator, which can then be used to inform trading decisions. Of course this stock information is slighty out of date with the real value, so the client will have to account for this when making their trade requests.

Making requests: In order to perform requests on their account, clients utilize a TCP json RPC paradigm with 4 main API calls to the broker system. Figure 4 shows the overall process the client request must go through in order to be executed. Each call includes the username and password of the client in order to ensure that the client is properly authenticated before purchase. The valid API calls are as follows:

1. Register the client account. This operation causes the replicator to create an account for the client with the given username and password. The account will start with 100,000 dollars to work with. If the account already exists, an appropriate error message will be returned.

2. Sending Buy/Sell requests to the broker. The client includes the stock ticker that is requested for purchase as well as the amount of that stock they wish to buy, sent as a JSON encoded dictionary. The request is validated by the replicator before being executed, so the client could receive a notification that for example they did not have enough money to complete the purchase. The replicator also ensures that the client has enough stock to sell when a sell request is made. As a response, the client gets back how much money the order cost/gained them in total, because they are unaware of current price information when they sent their request.

3. Request information on the client's positions and balances. The replicator will return a nicely formatted JSON encoded dictionary containing the current cash the client has on hand as well as the amounts of each stock that they own.

4. Polling for the leader-board of client net worth. The client sends this request to the broker, which will handle this request itself instead of forwarding it to a replicator. The leader-board operation is discussed in more detail under the broker section.

Error handling: One thing to note is error recovery for the client. There are a couple of cases to consider for the client:

1. The simulator crashes. In this case, the client will stop receiving stock updates, and basically the whole simulation comes to a halt as the broker will refuse to service more requests until it reconnects. The client will simply have to wait until the simulator gets restarted and then request to resubscribe.

2. The broker crashes. In this case, the client does not know the state of their request. It could have either made it to the replicator and been committed or it could have been lost somewhere in between. In this case, the client must reconnect to the broker when it comes back online and request its account balances. In this way it can keep track of whether or not its latest buy or sell went through. Each buy/sell operation is atomic, so the request either completed or did not.

3. The replicator which is hash-assigned to the client crashes. In this case, the client must just simply sit tight until the replicator comes back online. The broker will continuously retry to service the clients request until the replicator is able to be restarted. The client will then receive the forwarded response from the broker when it is finally completed.

4. The client crashes. In this case, the client be restarted and reconnect to the broker. It can then make a request to see their account balance. The client will then be get their account information, and they can resubscribe to the simulator and continue trading.

3 Evaluation

We perform several tests to evaluate the correctness and performance of our system, particularly how they achieve the 3 goals mentioned earlier.

3.1 Basic Logic and Error Checking

Using a interactive client, we manually test various combinations of API calls to ensure that our system correctly implements the mathematical calculations of asset exchange, leaderboard tracking, error checking, and other general logic such as user authentication. A visualization of such testing is not present in the paper.

For user authentication, we attempt to send any operation both for an unregistered user, as well as with the incorrect password. Both instances result in failure, which is intended. Of the API calls, the balance and leaderboard operations never result in error as long as the user is valid since they require no user input. All we confirm is that the values are mathematically correct and the leaderboard periodically updates. Register, buy, and sell operations, however, can result in error even in the presence of valid user usage. We test that if a client does not yet exist, register creates a new user, and if the client does not exist, it provides a failure. For asset-exchange operations, we test that correct conversions are made when the user has sufficient resources, and that an failure occurs returned if the user does not. Lastly, we error check what happens when a user requests an invalid stock or amount. These tests cover all possible interactions of a client with our servers.

3.2 Consistency

Although consistency was a primary goal for our system, consistency verification is difficult to accomplish numerically and visually. Rather, verification must be accomplished by logically insuring our design achieves the desired semantics.

Our system implements a checkpoint and transaction scheme for each replicator, persistently saving each clients information. Furthermore, clients can never influence the accounts of other clients and our load balancer ensures that each client's operations are handled sequentially. This ensures client requests are handled in order and there are no race conditions.

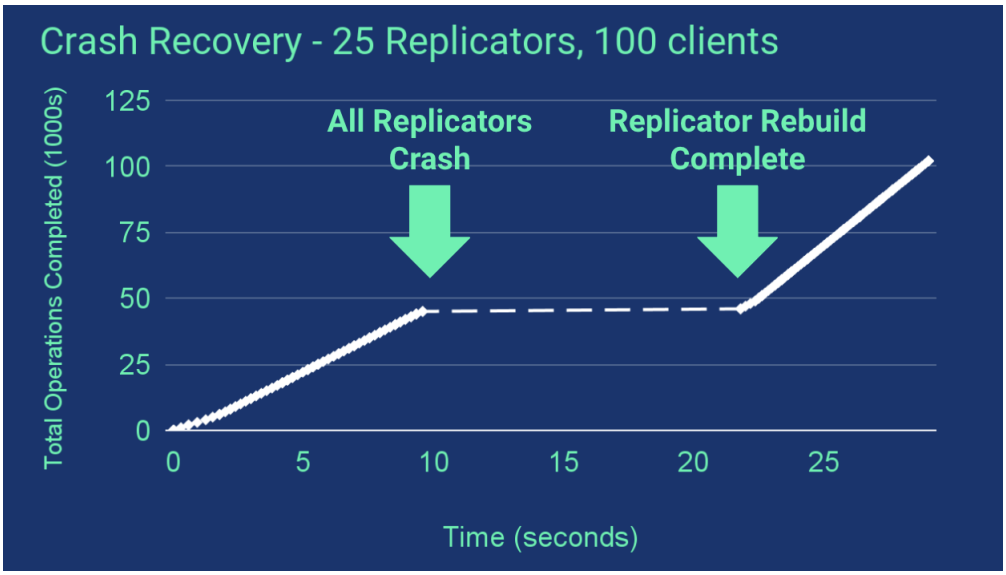

Figure 6: Data gathered from a run where we intentionally crashed all replicators and the replicator manager was able to restart them

On the server side, we were able to test the reconstruction of our replication servers by observing the total requests completed over time. To test this, we constructed the experiment seen in Figure 6. We created an example system with 100 clients and 25 replicator servers where we intentionally crashed all replicators midway through the run. We observe that initially our servers are servicing clients normally. At 10 seconds in, we crash all replicators and observe that no clients are being serviced. During this time, our replication manager is going through and restarting each replicator so that at $\sim$ 23 seconds in, our server returns to servicing clients at the same rate. Because the replication manager restarts the replicators in the exact state they crashed in, the system maintains consistency despite the crash.

3.3 Throughput

We test the total throughput of our system on 50, 100, 200, and 500 clients using a different number of replication servers. Initially, from 1-15 replication servers, the total throughput increases drastically for all clients. Then it starts leveling off. In an ideal system, we would only see this leveling off when the number of clients reaches the number of servers, since each server would begin handling more than 1 client. However, since requests must pass through a single broker, we believe it is a source of bottleneck for our system. The total throughput cannot increase beyond the concurrency capabilities of the broker. Despite this, we see that the throughput increases from 50 operations/second with one replicator to around 7000 operations/second at its peak around 15-20 replicators. This represents the success of the replicator system at increasing the overall throughput of the system and meets our second goal.

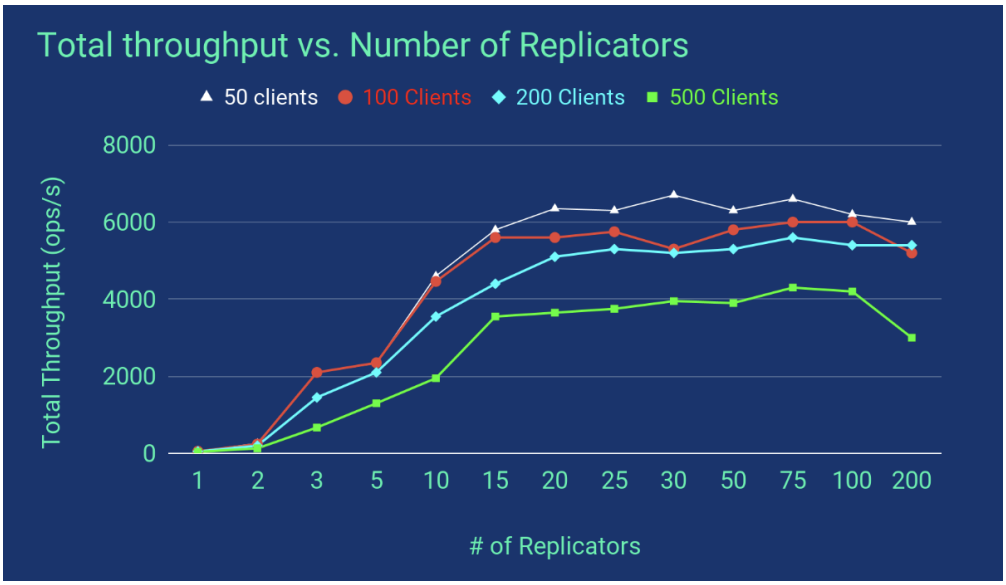

Figure 7: Data gathered from running the broker with different configurations of clients and replicators

3.4 Latency and Fairness

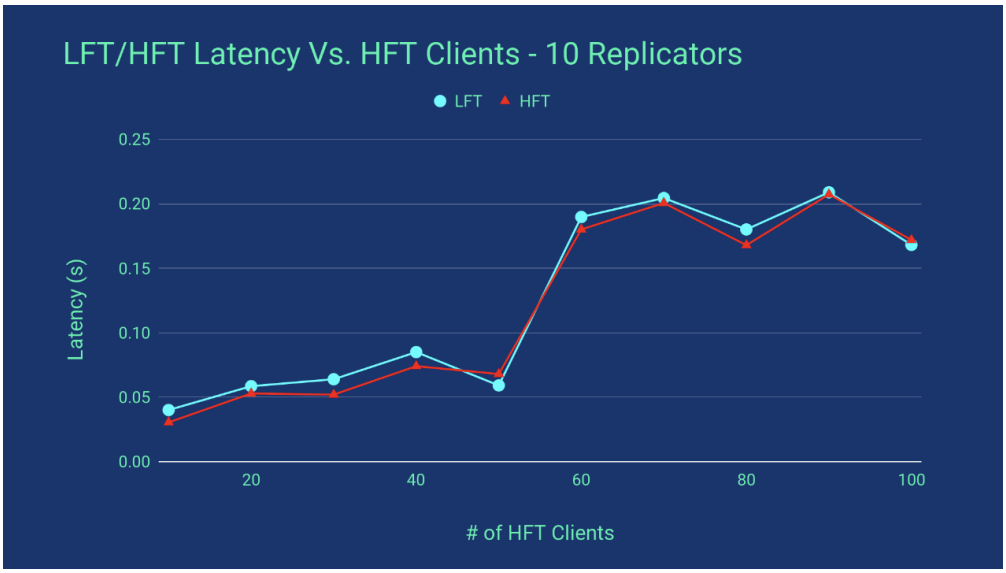

Figure 8: Measured latency of client operations for different numbers of total clients in the system. The blue line is a low frequency trading client and the red line is a high frequency trading client

We evaluate the latency of both a high-frequency and low-frequency client as the number of high-frequency traders increase. Overall, we observe that the latency for both goes up as the number of high-frequency clients increase, which is to be expected. We do see a significant jump around 50-60 clients, which we suspect to be the point our single broker again becomes a bottleneck. For both HFT and LFT, the latency is insignificantly different, so we believe our goal of fairness is sufficiently met.

3.5 Simulator

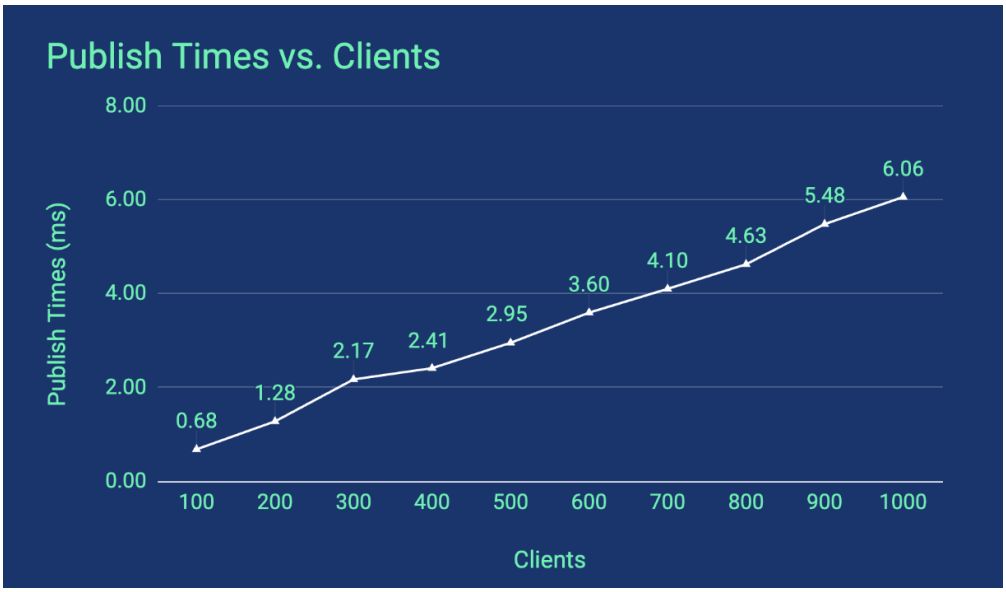

Figure 9: Measured publish times each publish interval for an increasing number of concurrent client connections.

We wished to ensure that our simulator was not another source of bottleneck, so we tested how long it would take in each publish interval to send all UDP packets to an increasing number of clients. We observe that even with 1000 clients, the publish time is only 6ms, which is below our update rate of 10ms and the actual publish interval of 100ms. 1000 clients is already more clients than our architecture can currently handle, but the linear nature of the graph suggests we could extend even further.

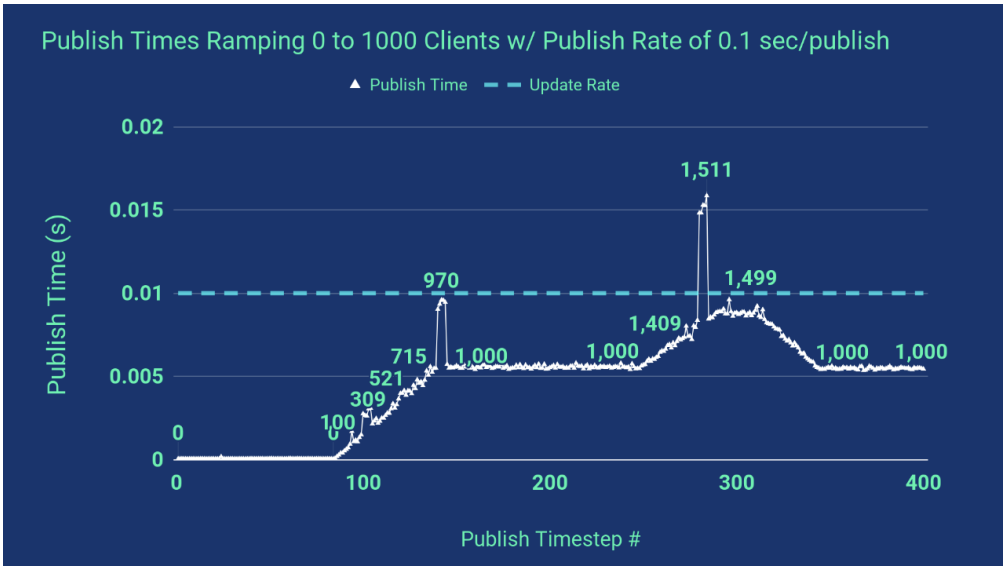

Figure 10: Measured publish times for 1000 clients as they resubscribe and are unsubscribed.

Furthermore, since we implement a pub/sub scheme that might result in temporarily duplicated clients, we also measured how the publish times changed as 1000 clients were consistently resubscribing in Figure 10. The data shows that the initial ramp to 1000 clients is roughly linear, as expected from Figure 9. The

publish times then level off at 6ms (again consistent with Figure 9) for a while, and then begin increasing as clients resubscribe before they are disconnected. In our scheme, our clients are resubscribing uniformly at $80 - 90\%$ of the simulator flush rate, resulting in around only half of the clients being duplicated at any time. In the worst case however, there can be at most 2 times the number of true clients connected.

We observe that the publish times do near the update rate, but this is not a problem. Should publish times rates ever go beyond our update rate, we could easily fix the problem by making updates asynchronous rather than sequential like we did. Lastly, there are two spikes in performance which we are unable to explain, but we likely attribute them to performance hiccups or expensive context switching.

3.6 Strategy

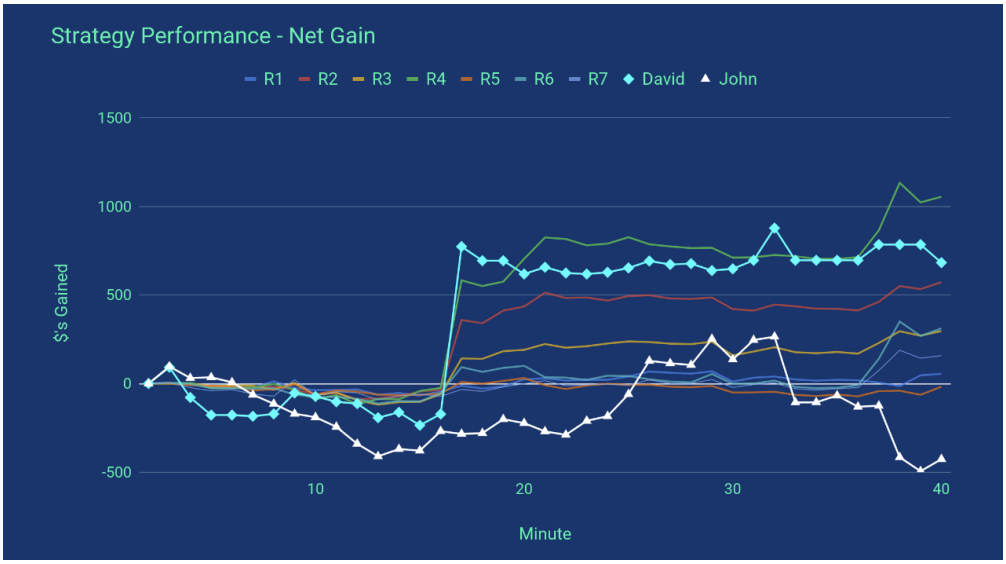

Figure 11: The net gain for 7 random clients (R1-R7), John's strategy (John) and David's strategy (David) over the course of 40 minutes.

Since the main motivation of building StockNet was to provide an entertaining competition between strategies, we implemented 3 different strategies.

1. The first strategy (R1-R7) is purely a random client who chooses to either buy or sell once every second. It uniformly selects 1-15 shares of a random stock to perform the operation on.

2. The second strategy (John) was developed by John Lee. Essentially, for the first 100 publishes, it tracks the probability of each stock to go up or down as well as by how much. It then computes the expected value of each stock, selects the highest value, and all-ins on that choice for the next 500 publishes.

3. The final strategy (David), by David Simonetti, takes the second strategy and inverses it by selecting the worst expected value stock. The idea behind this is to make a gamble on stocks that the market would consider undervalued in the hopes that the price corrects and increases after purchase.

While this plot is purely for amusement, we can see that John's strategy gets significantly outperformed by David's strategy and all the random clients.

Analyzing why certain strategies outperformed other strategies, we can look at David's strategy first. The reason that David's strategy outperformed all of the other strategies except for random number 6 was because it made a very lucky stock pick at around 15 minutes. The David strategy's net worth jumped about 750 dollars in a very short span of time, which seems to indicate David's strategy was very lucky in picking a good stock. Because David's strategy purposefully chooses the stock with the worst prospects, perhaps it was an undervalued stock that other people were passing up and not noticing for its value. The purchase of this undervalued stock at the right moment proved to be a good investment when the rest of the market came up, and it made up for the generally mediocre performance of David's strategy for the rest of the run.

Looking at the random strategies, it seems like they all generally made money, so it seems to be that the market was generally increasing over the period of time that the simulation ran. Random player 2 and player 6 managed to make some very impressive gains, but the majority of these gains were from the same lucky stock pick that David's strategy made at 15 minutes.

Looking at John's strategy, it performed the worst of all 9 players tested. This is interesting because John's strategy tried to calculate which stock was most likely to keep increasing in the near future to purchase. Perhaps this was a worse strategy because stock that were increasing might have already started decreasing by the time the information got to the client because of the information delay in the system. It could also be that John's strategy got extraordinarily unlucky.

4 Conclusion and Future Work

StockNet is a distributed trading competition that simulates an artificial market simulation. Clients can join to the game to test out their different trading strategies and try and make the most money in the context of an explicit information gap. StockNet focuses on three major goals: maintaining the consistency of the system over time, maximizing total throughput of the system, and minimize the latency of the system for all clients fairly. We achieve the goal of consistency by having robust error-handling protocols and utilizing checkpoint and transaction features on replicator servers. We achieve the goal of maximizing throughput by harnessing the computational and disk resources of multiple replicators in order to greatly boost the overall throughput. We achieve the goal of fairness and low latency by ensuring that all clients are serviced in the order their requests arrive. Overall, StockNet is a consistent, high throughput, and low latency distributed stock market simulation.

Future work would most likely focus around increasing the number of broker/load-balancer servers from one to a variable amount. Right now, our testing indicates that our system levels out around 7000 requests per second no matter how many replicator servers are in the system. This leads us to believe that the single broker serves at a bottleneck in the system. This is also supported by the generally lower throughputs at higher levels of clients, indicating that our system is overwhelmed with hundreds of concurrent clients. If we could alter the system to allow multiple brokers and for clients to forward their request to any of the brokers, then perhaps the system could continue to become scalable and achieve even higher throughputs.