Abdul Bagasarawala (abagas3)
Jason (Xingjian) Gong (xg8)
Nikhil Paidipally (npaidi2)

IE 497: Final Project Report
Mini-Exchange

Introduction

As part of our final project for IE 497 @ the University of Illinois, we implemented a Mini-Exchange in Python. The exchange consists of 3 fully independent components that can be deployed on separate machines/servers:

1. Gateway
2. Order Matching Engine
3. Frontend / Graphical User Interface

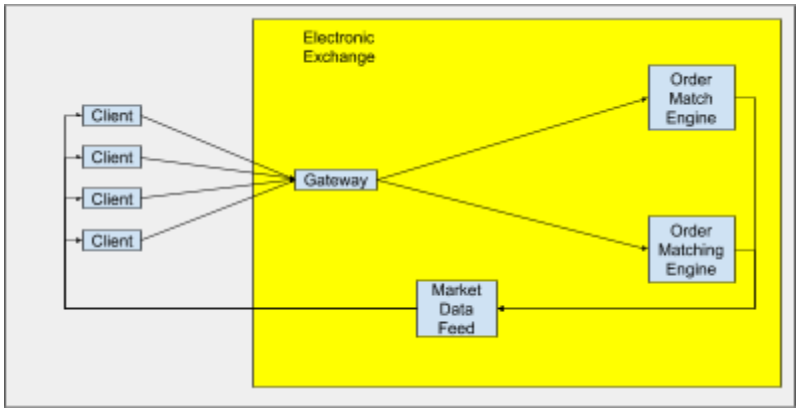
Instead of using our own simple protocol for communication, we decided to make our exchange compliant with the FIX (version 4.2) protocol. In addition, our project mirrors actual exchanges (like NYSE and CME group) in industry in that it allows clients to connect and communicate using TCP connections rather than a REST API. Initially, we we had also planned on having a ticker plant that sends out market data to allow users to construct an anonymized order book. However, due to timing constraints and other priorities (focusing a bit more on the front-end, for example), we were not able to fully complete a binary-protocol ticker plant.

We chose to work on this project since we believed that it would be a good chance to get hands-on-experience with the knowledge that we had gained in class. Specifically, we had learned about the various parts of the exchanges at a high-level, the anatomy of an order book, as well as about the FIX protocol and this project provided us with an opportunity to synthesize these ideas together in practice.

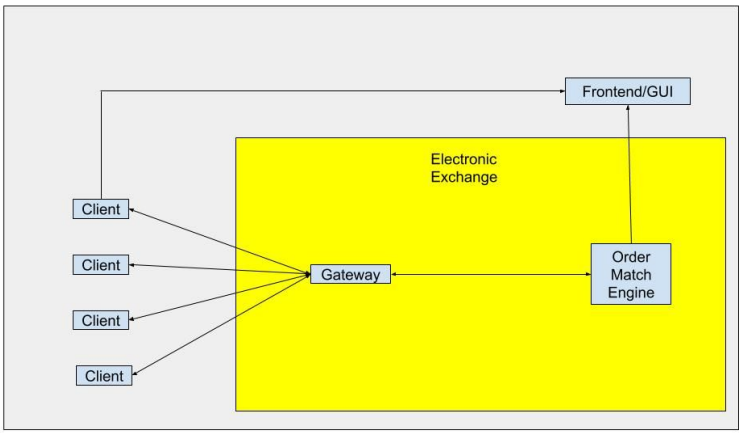
Abdul Bagasarawala (abagas3)
Jason (Xingjian) Gong (xg8)
Nikhil Paidipally (npaidi2)

Exchange Architecture

Initial Architecture:



Final Architecture:



Abdul Bagasarawala (abagas3)
Jason (Xingjian) Gong (xg8)
Nikhil Paidipally (npaidi2)

Implementation Details/Features

Gateway:

In our project, the gateway is the primary interface that clients have with the exchange. Similar to “real” exchanges and in line with the FIX protocol, clients first establish a TCP connection with our gateway and then send a Logon message to authenticate and *establish* a FIX session with the gateway. The gateway validates the Logon message and upon success, responds with a Logon message to confirm to the client that a session has been established. If the client sends an invalid Logon message or sends a different message prior to establishing a FIX session, the gateway responds with a Logout message and terminates the TCP connection [1].

After a FIX session has been established, the client is then able to send order-related messages. Specifically, our exchange currently supports submitting new limit orders (New Order Single), editing existing orders (Order Cancel/Replace Request), and cancelling existing orders (Order Cancel Request). For each order-related message that is received from the client, the gateway first performs session-level validation on the message. Specifically, the gateway validates the standard header (checking comp_ids, message types, etc), checks for message gaps by comparing the actual sequence number with the expected incoming sequence number, and ensures that all required tags for a given message are present. Next, the gateway performs message-specific validation. For instance, in the case of a New Order Single message, the gateway would ensure the symbol value is not empty, price and quantity are floating point value and non-negative, the transact time is a valid timestamp, etc. If at any point, the message fails validation, the gateway generates a Reject message and sends it to the client. However, if the message passes validation, it is forwarded to the order matching engine for further validation. When a response is received from the order matching engine, the gateway parses the data into a FIX message, adds outgoing sequence number and the sending time to the header, and sends it to the appropriate client.

In code, the gateway logic is split across various files. Specifically, there is a `FixSession` class that maintains the state for a single client and is responsible for validating incoming messages, processing valid messages and forwarding them to the order matching engine, and receiving responses from the matching engine, and transmitting them back to the client. The actual networking code and logic, however, is handled by a non-blocking TCP server that accepts connections from clients, reads/writes data from/to sockets, maintains messages queues for the FIX sessions to place outgoing messages into, and handles disconnect events.

While we do use a third-party library (SimpleFix) [2] to assist us in the encoding and decoding of the FIX messages, we deliberately chose **not** to use a FIX engine such as QuickFIX that abstracts away many of the concepts of the gateway. Rather, we wanted to get first-hand experience with reading FIX messages (using the body length), validating messages,

constructing messages using tags and values, as well as maintaining the state necessary for a FIX session. Although it made the gateway quite a bit more involved and required a few weeks to implement, it was overall a worthwhile exercise to learn and use the FIX protocol in greater detail.

Order Matching Engine

The purpose of the order matching engine is to receive four specific order types, buy_limit, sell_limit, modify, and cancel from the gateway, which maintained persistent connections with clients, and process the orders. If the incoming order was a buy or sell order, it would attempt to match the buy and sell limit orders with the appropriate orders within the orderbook. The matched orders would then be removed from the order book and all of the trades would be sent to both the gateway and ticker plant to notify clients that their orders have been filled, either partially or in its entirety. If no matches were found or the order isn't filled in its entirety, the order would then be added to the orderbook to be matched with future orders. Modify and cancel orders would be be processed similarly by matching these orders with the order in the orderbook with the same previous order id, and the matched order would get processed respective to the order type and new parameters.

Initially, our plan was to have a multithreaded order matching engine with the main thread communicating with the gateway, child threads performing matching for a specific instrument, and another child thread handling communications with the ticker plant. This would've theoretically increased throughput as the matching process would be able to handle multiple orders of the different instruments simultaneously as opposed to only doing one at a time.

However, due to the shortcomings in python's multiprocessing libraries, we had issues regarding the thread-safe data structures. Our problem occured when child processes added outgoing messages to a process-safe queue, and the parent thread reads data from queue and tries to send it. If the send call only managed to transmit part of the data, there's no good way to remember that information. We ended up compromising by making the order matching engine entirely single-threaded, thus solving our earlier issue.

Frontend / Graphical User Interface

We used the Plotly and Dash python framework to create an interactive web-based application for our exchange. This platform is a data visualization tool to help users analyze and track market data over a period of time.

Users can selected which stock/market to plot with a drop down menu at the top of the page. We also implemented a checklist option to select which set of orders and trades to plot. In the first tab, we are plotting the order book consisting of sell and buy orders. The order book lists the number of shares being bid or offered at each price point. Right below the order book, there is a

slider to choose how much data to plot on the graph at one time. Along with the graph, all the buy and sell orders in the market are displayed in tables, which are sorted by the price. In this tab, users can also submit orders by typing in the correct price and volume of the order into input text boxes. Users can choose to use the market valuation for the price which is just the price of the latest trade done on the market. When the user presses the submit button, a pop up notification will show up to confirm if the user is sure about submitting this order.

In the second tab, we are plotting all of the trades for a stock in a scatter plot. Users can use the selector at the bottom to specify the range of dates to plot. When users hover over a data point, the small table below will update the to show more information about a trade based on the selected point. On the side of the graph, there is a graduated bar which will also update to reflect the volume of the trade that the user is hovering over. We decided to implement this bar because the graph only plots the date and price; we believe that volume is a very important factor to look at when analyzing market data. Lastly, a table of all the trades done on the market with more information is at the bottom.

Currently, we are reading data from CSV file as pandas data frames. When a user submits an order, the order is written to a text file. All the graphs and tables in this application update with correct information everytime the dropdown menu and checklist values are changed. In addition, the website automatically updates all the components every 15 seconds in case the CSV files have been updated or changed.
To build this section of the project, we first built a simple version of the two graphs and various tables. Then, we started to add features such as range selectors, drop down menus, and tabs. We had to use callbacks to update the graphs and tables when the value of a component changes. Finally, we styled the components by using separate CSS files and external stylesheets. Using Dash and Plotly made the process much simpler, and we were able to use a list of predefined components. The documentation was really good, so it was easy for us to use this new technology.

Abdul Bagasarawala (abagas3)
Jason (Xingjian) Gong (xg8)
Nikhil Paidipally (npaidi2)

TCP/Networking/Epoll

When we initially implemented the TCP server for the gateway, we were using a multithreaded server that would spawn a fresh thread each time a new connection was received. While that approach was fairly straightforward to implement, upon further research, we learned that such a strategy does not scale well due to a variety of reasons. Specifically, with a large number of clients/threads, we would run into problems of exhausting virtual memory since each thread needs a dedicated amount of stack memory/frame. In addition, context switching is quite computationally expensive and should ideally be minimized when performance is of concern. The C10K problem [3] outlines the issue in much greater detail. Finally, since our project is implemented in Python, we would not be able to achieve "true" multithreading due to the existence of the Global Interpreter Lock.

Consequently, after performing online research and discussing the problem as a group, we decided to use non-blocking single-threaded I/O and Linux's `epoll` function to implement our TCP server(s). While that decision made our project less portable (the servers could only be deployed on Linux machines) and a bit more difficult to implement, it should allow the exchange to scale much better and be able to efficiently serve a large number of clients. In addition, it was a challenging and intriguing learning experience that exposed us to a new technology.

In code, we implemented the networking logic in a modular fashion to allow for maximum code reuse (and minimum code duplication) since the servers for the gateway and the order matching engine were quite similar. We accomplished this by making an abstract non-blocking `TCPServer` class that implemented the common code and having the gateway and order matching engine servers inherit from the abstract class and implement specific/unique features that they needed.

FIX Functionality

As mentioned above, we specifically decided against using our own simple protocol for data communication. Rather, we used the FIX protocol that is commonly used in industry to communicate with "real" exchanges. Doing so allowed us to gain a deeper understanding on FIX, its intricacies, as well as the best practices, controversies, limitations, and other details of the protocol. We tried to follow the FIX specification as closely as possible (outlined by the FIX Wiki and Onix Solutions) [4 and 5], but did deviate from one requirement of sending multiple execution reports (discussed in more detail below).

Abdul Bagasarawala (abagas3)
Jason (Xingjian) Gong (xg8)
Nikhil Paidipally (npaidi2)

Development Process

Testing

Although we had neglected writing test cases during the initial stages of the project and relied instead on manual checks using the command line, we rectified our approach as we progressed through the project.

Specifically, we wrote over 40 unit tests to test ensure that our code worked correctly in isolation and produced the expected results with a variety of inputs. In addition, we also wrote 10 integration tests that verified our exchange's functionality as a whole. We took advantage of the Fixtures features provided by the PyTest framework that allowed us to run our integration tests in an automated fashion. In other words, instead of relying on uncertain timing dependencies, we used fixtures to ensure that the gateway did not start up until the matching engine was fully loaded and that the clients did not attempt to connect to the gateway until it was fully initialized.

Git

In an effort to simulate development practices used in industry, we heavily utilized the various features that GitLab had to offer. In particular, we created **issues** to outline our plan for the project, assign tasks to the relevant group members, and keep up with progress on the features. We also made `master` a protected branch (no direct pushes allowed) and used **separate branches** to commit and test our work before opening **merge/pull requests** to the master branch. Finally, we also set up a simple **continuous integration pipeline** with a Google Cloud Platform VM that would run all of the tests for our project whenever someone pushed new code or opened a pull request (to ensure that existing functionality was not inadvertently broken).

Best Practices/Code Styles

While writing code, we worked hard to ensure that our work followed best development practices such as writing modular and general code so that it can easily be extended and updated to support more actions (such as different FIX message and order types) in the future. We also included logging, inserted the appropriate amount of documentation/comments, extracted common code into functions and shared utils as well as paid attention to minor details like using constants instead of magic numbers and having descriptive variables and function names. In addition, we used a consistent style guide (PEP8) along with a VS Code plugin that would automatically format our files whenever we saved.

Future Improvements

- Complete implementation of the ticker plant
    - Send out Level 2 or Level 3 market data over multicast UDP
    - Use efficient binary format such an MDP 3.0/Simple Binary Encoding or FAST (FIX Adapted for STreaming) [MDP 2.0]
- Support more order types (market, fill or kill, etc)
- Data Persistence
    - Use a database to persist order book and historical data to disk
    - Seed order matching engine with previous data when re-initializing
- Expand FIX protocol support
    - Handle more FIX messages such as Resend requests, test requests, order status, etc
- Use a thread/process pool along with the single-threaded epoll server to take advantage of multiple CPU cores
- Failure Handling
- Translate project into faster compiled languages such as C/C++, Golang or Java

Statement of Contribution:

- Abdul: Worked on the gateway and networking parts of the code. Refactored the order matching engine module. Integrated FIX protocol in the project. Wrote automated test cases for the project
- Jason**:** Worked on the front-end for our exchange. Built an interactive dashboard to plot all the buy and sell orders along with all the trades done on the market.
- Nikhil: Worked on initial design and code of the Mini-Exchange. Worked on the second version of the matching engine. Helped implement real-time updating for the front-end.

Abdul Bagasarawala (abagas3)
Jason (Xingjian) Gong (xg8)
Nikhil Paidipally (npaidi2)

References

0. GitLab Project: https://gitlab.engr.illinois.edu/npaidi2/ie_497_mini_exchange_project
1. https://javarevisited.blogspot.com/2011/02/fix-protocol-session-or-admin-messages.html
2. https://github.com/da4089/simplefix
3. https://en.wikipedia.org/wiki/C10k_problem
4. http://fixwiki.org/fixwiki/FIXwiki
5. https://www.onixs.biz/fix-dictionary/4.2/msgs_by_msg_type.html
6. https://fixparser.targetcompid.com