MODELING AND SIMULATION OF ORDER BOOK DYNAMICS: A STUDY ON FINANCIAL MARKET MICROSTRUCTURE

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Ruize Ren

May 2023

ABSTRACT

This thesis presents two projects for Julia developers in financial studies. The first project focuses on the Limit Order Book (LOB), which is a useful tool in studying financial market microstructure. As trading volume grows, efficient data structures become more important for reducing latency and modeling accuracy. To address this challenge, the VLLimitOrderBook.jl package is proposed as a software solution that simulates the order book dynamics and captures the interaction between market participants and exchanges. This package is validated via data feed messages from Nasdaq TotalViewITCH. It serves as an effective tool for researchers and traders to study order book behavior and develop trading strategies. The second project involves the development of a WebSocket API for PQPolygonSDK.jl, a software development kit for Polygon.io financial data warehouse. We provide an analysis of data volume and latency, as well as a discussion of the WebSocket structure and its future development.

BIOGRAPHICAL SKETCH

Ruize Ren was born in Anyang, an ancient city in Henan, China. In 2016, he went to Tianjin University for his undergraduate studies in the field of Chemical Engineering and Technology. After graduating in 2020, he deferred his graduate program at Cornell for one year and went back to Tianjin University to study computer science. He came to Cornell in 2021 and began graduate studies in Professor Jeffrey Varner's research group. During the summer of 2023, he went to Bellevue, Seattle for an internship as a Software Development Engineer at Amazon. Beyond his academic life, he also spent a lot of time in weight lifting and boxing.

This work is dedicated to my parents, Haimin Ren and Hongjing Dong.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Professor Jeffery D. Varner, whose dedication and patience help me to be a better researcher. His constant support and guidance enabled me to pursue my career aspirations and dreams. In the meanwhile, I would also like to extend my appreciation to Professor Kenneth P. Birman, whose guidance help me to develop a keen interest and professional knowledge in Cloud Computing. I would also like to express my genuine appreciation to Aaron Wheeler, my research mentor who always provides me with insightful comments and constructive suggestions, I am grateful for that. I would like to express my thanks to my awesome labmates: Sandra Vadhin, Abhinav Adhikari, Jacob Belding, Divya Lakshmi, Aravind Suresh, and Mavis Ofori-Brown. Finally, I would like to thank my parents, for their unyielding support and love, which have been a constant source of strength and inspiration throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 History of Stock Market Exchanges

For centuries, individuals engaged in buying and selling will have gathered at marketplaces to engage in exchange activities. During those activities, the seller presents an initial offer price for their product or service, and the buyer usually counters with a lower bid price. When the seller observes the disparity, he or she may reduce the offer by a tiny amount, while the buyer may raise the bid by a small amount. This process will continue until both parties reach a mutual agreement or one side chooses to end the negotiation[27].

The history of commerce can be traced back to the merchants of Venice in the thirteenth century. Due to its strategic position between East and West, Venice plays a major role in Mediterranean commerce. By trading debts, purchasing government debt issues, and selling debt issues to individual investors, Europe's moneylenders filled the void left by the bankers. To expand their business, they also established centralized financial institutions and public administrators, such as deposit banks and foreign exchange banks, which have a significant impact on the contemporary economy. Venetian bankers would carry slates containing information about the different issues for sale and meet with customers, much like a contemporary broker [42]. In 1531, Belgium established a stock market in Antwerp[19]. During that time, unlike current stock exchanges, they did not deal with actual stocks. Instead, moneylenders and brokers usually gathered there to negotiate business, government, and individual debt issues, primarily through promissory notes and bonds. Although various types

of business-financier partnerships could generate income similar to stocks, there were no official shares that were traded during the 1500s[47].

In the 1600s, a British joint-stock company, the East India Company, was created to exploit surplus value from local residents. These group of merchants from London engaged in sea voyages and often encountered numerous risks, including pirates, extreme weather, and poor navigation systems[25]. To mitigate the risk and protect their fortunes, ship owners sought investors who would provide financial support for the voyage. In return, those investors will get a share of profit. However, these early companies typically lasted for a single voyage. Then they dissolved and form a new one for the next voyage[25]. Investors often spread their risk and increase the chance of success by investing in multiple ventures at the same time. The formation of the East India companies revolutionized the business model. This company was developed because of the problems of "distant trade" which always holds a large amount of capital. Instead of raising money through voyages, these companies issued stock that paid dividends on all of their voyages operations. These were the first modern joint-stock companies, which enabled the companies to demand higher prices for capital accumulation, and business growth. With support from the royals, who prohibited competition, the companies generated significant profits for investors[47]. The bloom does not last very long. During that time, there is no regulation or limitation for issuing the same stocks, so the South Sea Company (SSC) gathered huge funds by reissuing numerous stocks. The other company noticed this similar profitability and rushed to offer new shares for various ridiculous ventures. The bubble burst when the SSC failed to pay any dividends on their profits. The subsequent crash caused the government to ban the issuing of shares, which lasted until 1825[48].

The first stock exchange was officially formed in 1773 in London (LSE)[6]. It did not become the largest exchange due to government restrictions. In 1792, the New York Stock Exchange (NYSE) was founded, and it quickly beat London Stock Exchange, becoming the world's largest stock exchange. The NYSE has been the center of American finance for over two centuries and has played a critical role in the US economy. The NASDAQ (National Association of Securities Dealers Automated Quotations Stock Market) was established in 1971, and it quickly became a rival to the NYSE. Unlike the NYSE, which operated as an auction market, the NASDAQ was a dealer market[22]. The NASDAQ was dominated by technology companies, such as Amazon, Apple, Google, etc. In recent years there are also non-tech companies coming to NASDAQ. It is widely regarded as the stock exchange for the new economy. The biggest difference between NYSE and NASDAQ is that the NASDAQ is a dealer market, however, the NYSE is an auction market[10]. In an auction market, buyers and sellers submit competitive bids and offers and the highest bid and the lowest offer get matched[1]. In a dealer market, dealers list prices that they would like to buy and sell for a certain security[2].

In the early times, orders were placed by phone or in person, and transactions were conducted on a trading floor. This is very inefficient and could limit trading strategies and volumes. With the development of technology, electronic trading is introduced. This innovation changed the way the stock market operates, making trading more efficient and accessible to a wider range of investors. Nowadays, the limit order book (LOB) mechanism has taken up to half of the markets[45]. The Swiss, Helsinki, Hong Kong, Toronto, Shenzhen, Tokyo, and Vancouver Stock Exchanges, together with Euronext and the Australian Securities Exchange, all now operate as pure LOBs[34, 41]. The NASDAQ, the London

Stock Exchange (LSE), and the New York Stock Exchange (NYSE) all operate a LOB system[28, 33].

LOB systems provide investors with a transparent view of the market depth (level of prices) and liquidity, enabling them to make thorough trading decisions. Liquidity means the ability to be traded without affecting market prices for a certain security. However, as trading scales, both software and hardware solutions become increasingly important to meet the requirement of low latency[36]. In this review, we discussed the development of stock market exchanges. We then explored the basic concepts of LOB systems. We went through recent research on LOB performance optimization from both software and hardware ways.

1.2 Basic Concept Overview

The limit order book is a crucial component of contemporary financial markets that combines orders for certain securities from multiple buyers and sellers. Typically, the highest buying price is lower than the lowest selling price. This process resembles an auction in which buyers and sellers compete to achieve their respective goals. For buyers, the purpose is to obtain an item at the lowest possible price, while sellers want to sell their product at the highest possible price without deterring prospective buyers. Nevertheless, putting a price that is too low may not ensure fulfillment, since a competitor bidder may offer a greater price, while setting a price that is too high may repel prospective purchasers. Ultimately, the most competitive bid matches the most attractive ask, resulting in the transaction's execution.

In the limit order book, a buy limit order means that a person is willing to buy a share of stock at or below a certain price. On the other hand, a sell limit order means that a person is willing to sell some share of stock at or above a certain price. The highest buy price is referred to as the bid price, and the lowest ask price is referred to as the ask price. The difference between the bid/ask price is referred to as the bid/ask spread. A brokerage or market maker can profit on the spread between the bid and ask prices. They purchase the stock at the bid price and sell the price at the ask price. The bid/ask spread can be used to measure the liquidity of the market, which refers to the ability to trade a stock without affecting its prices (Fig. 1.1).

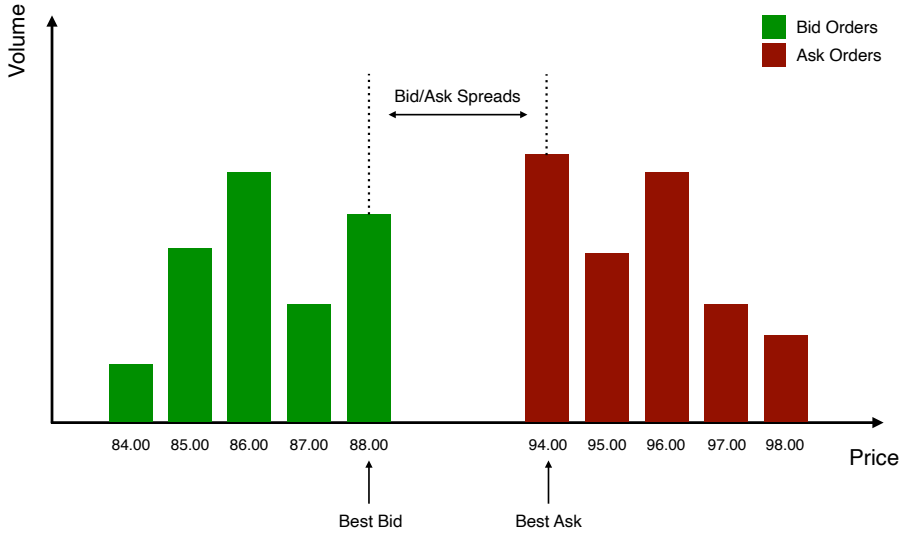

Figure 1.1: A snapshot of limit order book for ticker XYZ at time T

More than half of the world's stock exchanges are order-driven, which all center around the limit order exchanges[45]. Therefore, studying the limit order

book is especially important. By contrast, market orders are not stored in the order book and are executed immediately. This type of order prioritizes execution above order prices, as opposed to the limit order. In addition, there exists a third type of order, the stop order, where the transaction gets triggered as a market order as long as they are reaching the market price. There are other orders on the market; we shall clarify this throughout our discussion of validation.

There are three order fill options in the stock market: Immediate or cancel (IOC), Fill or Kill (FOK), and All or None (AON). The IOC fill option is to fill an order immediately while eliminating the unfulfilled section. This option allows for partial fill. The FOK fill option means the current order can only be either filled or killed, no partial fill is allowed. If the order is not filled, it will be removed from the order book immediately. The AON fill option specifies that a current order must be filled entirely or not executed until it is expired or canceled by the user.

Most exchanges use two execution algorithms for order-matching engines. One is the price-time algorithm, which is used by most exchanges, and the other one is the pro-rata algorithm. The price-time algorithm matches the order at the best available price with a higher time priority. For example, if bid order A is for 89.00 USD and bid order B is for 90.00 USD, the B order will have higher priority and get executed first. If both orders are at the same price level, the order that comes earlier to the queue will have a higher priority. The price time algorithm discourages the order at a later time priority, but it is computationally more demanding than the other algorithm since more market participants want to place many small orders at different prices[11]. The pro-rata algorithm, on the other hand, allows the orders to be filled in proportion to their share of the

price. Unlike the price-time algorithm, the pro-rata algorithm can guarantee access to the order with lower time priority at the same price level. Securities may have varying prices across different exchanges, creating an opportunity for individuals to purchase and sell these securities simultaneously at different exchanges in order to profit from price differences. This practice is known as arbitrage and occurs when markets are not efficient or when foreign exchanges are involved [21]. It is important to note that prices seen on platforms such as Yahoo Finance or Robinhood are based on the NBBO (National Best Bid and Offer), which searches all available exchanges or venues to obtain the highest bid price and the lowest ask price [23].

When it comes to investing in the stock market, investors have various options to choose from individual securities, unit investment trusts (UITs), exchange-traded funds (ETFs), and mutual funds. Individual securities are fungible and tradable, which is used to collect capital from the market[20]. UITs, ETFs, and mutual funds are groups of securities, but they differ in several ways. Mutual funds are more actively traded than UITs and are not bought or sold unless significant changes occur, such as a bankruptcy or corporate merger[17, 8]. ETFs are similar to mutual funds, but they can be traded like regular stocks and usually track an underlying index, sector, commodity, or other assets[4]. On the other hand, mutual funds are a type of investment vehicle consisting of a portfolio of stocks, bonds, or other securities, which are professionally managed and are usually charged annual fees, expense ratios, or commissions[8].

The feature or properties of the limit order book, such as order depth information, and the changes of the bid/ask spread, are very valuable for a certain financial asset. This data is widely used for various trading algorithms[46, 26],

market anomaly detection[32, 24, 43], price formation[24, 40]. Despite the importance of LOB data, there are obstacles to acquiring them since most vendors do not provide order book data or need an expensive subscription fee, which could prevent researchers from accessing it. In addition, some data is completely unavailable to the public as some trades are executed through dark pools. Therefore, it is crucial to develop a Limit Order Book object that simulates order dynamics and provides a reference for researchers.

1.3 Ways to Pursue Efficiency

The pursuit of effective trading solutions remains an ongoing endeavor. Particularly for those who engage in High-Frequency Trading (HFT), minimizing latency has become an increasingly critical concern. HFT trading strategies typically operate within a timescale of 1ms [35], which has a significant impact on market quality. In order to lower the latency of the limit order book system, it is essential to explore efficient hardware and software solutions.

There are many hardware solutions to reduce the latency, Denholm, et al.[29] present a reconfigurable approach for A/B (two identical feeds) arbitration that compensates for missing packets in the stock market. This method could provide flexibility for various operational modes such as prioritizing low latency or high data reliability. He, et al.[36] provide a hardware-friendly order book update algorithm and designed a fixed tick data structure that could be easily mapped to hardware. The algorithm running on a FPGA-based solution can process 1.2-1.5 million messages per second with a throughput of 10Gb/s and a latency of 132-288 nanoseconds. Wary, et al.[36] explore parallelism and re-

configurability of FPGA and achieve 133 times the speed of the corresponding software implementation.

Software solutions also play a crucial role in reducing order book latency. An efficient data structure can significantly impact the performance of an order book system. Jhon et al. [38] concluded that while arrays are suitable for quick insertion and extraction, balanced tree data structures such as AVL tree should be used when dealing with big price levels and order volumes. With our VL-LimitOrderBook package, order queues at various price levels are stored in the AVL tree. Four different methods were compared based on their algorithmic complexity (Table 1.1). Method 1 involves allocating an array of the price levels, with each price level using a linked list to store the order queue appended to that price. The subscript i in the n_i is the length of chain in level i. Method 2 is an extension of method 1 but adds a hash table (open addressing) for reference ID. This hash table uses reference ID for the key and store order address node for the value. The average case is better than method 1. Method 3 is also an extension of method 1 but applies a hash table in a linked list (chaining) implementation. The subscript j in the n_j is the length of chain in level j of corresponding reference id in the hash table. Method 4 is derived from method 1 but replaces the hash table with a balanced search tree data structure such as AVL tree/B Tree. This tree uses reference ID as the key for the tree node which also contains a pointer to the price node.

Table 1.1: Operation complexity for different data structures in order books

Methods	**Methods 1**	**Methods 2**	**Methods 3**	**Methods 4**
Insert, average case	*O(1)*	*O(1)*	*O(1)*	*O(logn)*
Insert, worst case	*O(1)*	*O(n)*	*O(1)*	*O(logn)*
Delete, average case	*O(1)*	*O(1)*	*O(1)*	*O(logn)*
Delete, worst case	*O(1)*	*O(n)*	*O(1)*	*O(logn)*
Cancel using price, average case	$O(n_i)$	$O(n_i)$	$O(n_i)$	$O(logn + n_i)$
Cancel using price, worst case	*O(n)*	*O(n)*	*O(n)*	*O(logn)*
Cancel using Ref-ID, average case	*O(n)*	*O(n)*	$O(n_j)$	*O(logn)*
Cancel using Ref-ID, worst case	*O(n)*	*O(n)*	*O(n)*	*O(logn)*
Modify quantity, timestamp not changed, average case	$O(n_i)$	$O(n_i)$	$O(n_i)$	$min(O(logn), O(n_i))$
Modify quantity, timestamp not changed, worst case	*O(n)*	*O(n)*	*O(n)*	*O(logn)*
Modify quantity, timestamp changed, average case	$O(n_i + 1)$	$O(n_i + 1)$	$O(n_i + 1)$	$min(O(logn), O(n_i))$
Modify quantity, timestamp changed, worst case	*O(n)*	*O(n)*	*O(n)*	*O(logn)*
Modify price, average case	$O(n_i + 1)$	$O(n_i + 1)$	$O(n_i + 1)$	$min(O(logn), O(n_i))$
Modify price, worst case	*O(n)*	*O(n)*	*O(n)*	*O(logn)*

CHAPTER 2

A MODEL FOR SIMULATION OF ELECTRONIC ORDER BOOK DYNAMICS

2.1 Abstract

The Limit Order book (LOB) is a fundamental component in the financial market microstructure that records a list of limit orders from potential buyers and sellers for a specific venue. LOB system has taken up to half of the markets[45]. However, as the volume of trading messages grows, the efficiency of data structures becomes more and more important for reducing latency and accurately modeling order behavior. Therefore, we proposed a software solution - VLLimitOrderBook.jl package - that could precisely simulate the order book dynamics and captures the interaction between market participants and exchanges. To validate the effectiveness and accuracy of our VLLimitOrderBook package, we used real-world historical data feed messages from Nasdaq TotalViewITCH[3]. We conducted a series of experiments to test the performance of our package. This package provides an effective tool for researchers and traders to study the behavior of the order book and perform trading strategies.

2.2 Background

The order book model has been studied for years, but due to the complexity of the orders, it's challenging to capture all market behaviors. Rosu, et al. [44] presents a tractable equilibrium dynamic model for an order-driven market,

which allows people to modify and cancel the order as they want. This stylized model can deliver the shape of the limit order book and its evolution in time. Similarly, Foucault, et al. [31], used an equilibrium dynamic model and produced predictable relationships such as spread, trading frequency, market frequency, etc. But those models involve unknown parameters or relationships that influence agent preferences, making them difficult to use in applications. Cliff, et al.[27] has released their model for simulating a centralized financial market, which has been successfully used for teaching and research purpose. However, it uses the Python 2.7 version in which they use dictionary data structures to store price data. The implementation for dictionary in Python is a hash table and open addressing to resolve hash collision. As mentioned earlier, the hash table data structure is generally less effective than a balanced tree. One could use a novel data structure to improve performance. Jericevich, et al[39] developed a low latency, high throughput matching engine, which provides a realistic platform for agent-based models simulation. However, their order book was implemented in B+ tree, which has limitations in-memory operations. Hollifield, et al.[37] developed a model based on the trade-offs between order prices, execution probabilities, and picking off risks. They conducted a semiparametric test using data from the Stockholm Stock Exchange. Famer, et al.[30] investigated the cause of large fluctuations in prices on the London Stock Exchange at the microscopic level of individual events. They concluded that the distribution of large price changes is related to the gaps in the limit order book, which are caused by the granularity of order flow. Both works highlight the complexities involved in financial markets and the various factors that contribute to price movements and optimal order submissions.

Currently, most venues either do not provide level 2 order book data or

charge a very expensive subscription fee. Additionally, there are currently no available packages for the Limit order book developed in Julia. As such, there is a clear need for a package that offers an order book construction with real historical data validation, which could be a valuable tool for traders and research scientists. Another advantage of this package can be applied to adaptive machine learning. By utilizing the package as real-time instructive feedback, it can assist ML models in improving and optimizing training results. These features make our package an innovative solution and can be applied to many fields of research.

In this study, we built a package that can simulate the order book dynamics. Specifically, the implementations of the package were based on fixed tick order book data structure and AVL Trees Model[36, 38]. We adopted LimitOrderBook.jl[12] and made further modifications to it. The original package LimitOrderBook.jl implemented the submission and cancellation of market orders and limit orders. It also provided information regarding the order book, such as the depth of the book, total bid/ask volume, and best bid/ask prices. However, several things could have been improved with the original package. Toward these issues, we fixed several order matching and account tracking issues present in the original package. Further, the package were extended to include additional filling options. We also implemented functionality to load or persist the state of the entire order book as a comma-separated value (CSV) file. Finally, to validate the accuracy and correctness, we applied data feed messages from Nasdaq TotalViewITCH[9], a direct data feed product offered by the Nasdaq Stock Market, LLC. This package is open source and available on GitHub[15]. Our package is maintained on the Varnerlab GitHub site at Cornell University. The code is available under an MIT software license.

An order book instance contains two *OneSideBook* objects, the ask side, and the bid side (Fig. 2.1). Each side uses an AVL tree to store different orders at different price levels and contains other general information including volume and best prices.

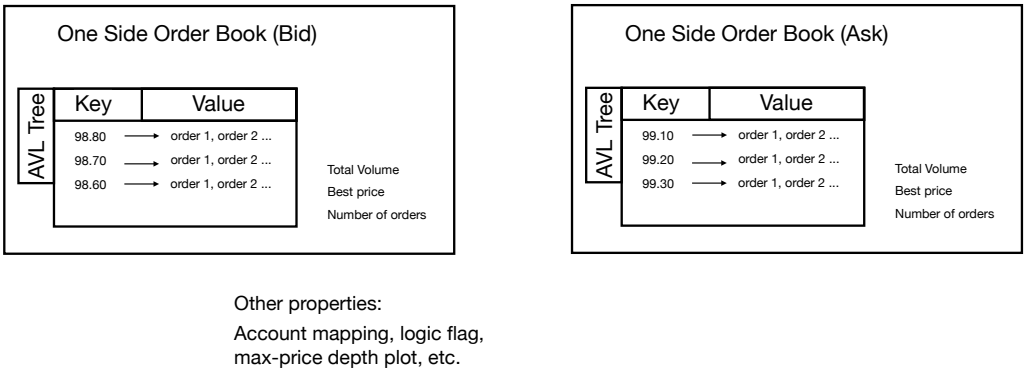

Figure 2.1: An instance of limit order book in VLLimitOrderBook.jl package

2.3 Contribution

We have addressed the bug mentioned in the introduction by updating the account mapping in the order-matching process. The original order book did not track specific user behaviors, as a result the account information was not updated when an order was matched and removed from the order book. Furthermore, the original order book allowed for cross-match limit orders on one side of the book, which is not allowed on most exchanges. It's counter-intuitive. To address this issue, we threw errors in the package to when this scenario happened.

We tested every single branch of the new code since the original package has limited test coverage. Additionally, the original package did not provide

the functionality to read from CSV files, so we added a function to load the order book state from CSV files. In terms of order filling, the original package only offered static filling types. However, some orders may have different filling types from others. As such, we changed the *OrderTraits* component to a mutable structure. We will go into more detail about this modification in the *Discussion* section. To improve efficiency, we also modified the *Order* data structure to be mutable to increase performance, which is helpful when orders are partially filled or canceled. Lastly, we provided a detailed discussion on testing the performance of our package, which we will cover later in this thesis.

2.4 Results

In this part, we will discuss the testing result for the package. Following is the hardware specification (Table 2.1).

Table 2.1: Hardware specification for benchmark and performance analysis

Model	**Operating System**	**RAM**	**CPU Specification**
Macbook Air	MacOX 13.1	16 Gb	M1, 8-core CPU
Dell Desktop	Ubuntu 19.04	16 Gb	Intel Core i7-6700 CPU @ 3.40 GHz * 8

2.4.1 Package Performance Evaluation

Time of Constructing Order Book

We evaluated the order book package by placing different ranges of limit orders to test its efficiency for a specific ticker across different volumes. The result showed that there is a linear relationship between the time required to build an order book and the total number of orders placed (Figs. 2.2, 2.3). The processing rate decreases as the total number of limit orders increases. This is due to the larger instance of the limit order book (LOB) when more orders are placed, which causes the system to slow down. In addition, there are some outliers at the beginning of the graph. Each data point is an average of five identical experiments, so the outliers could be the randomness in the system since we only varied the total number of limit orders in the experiment.

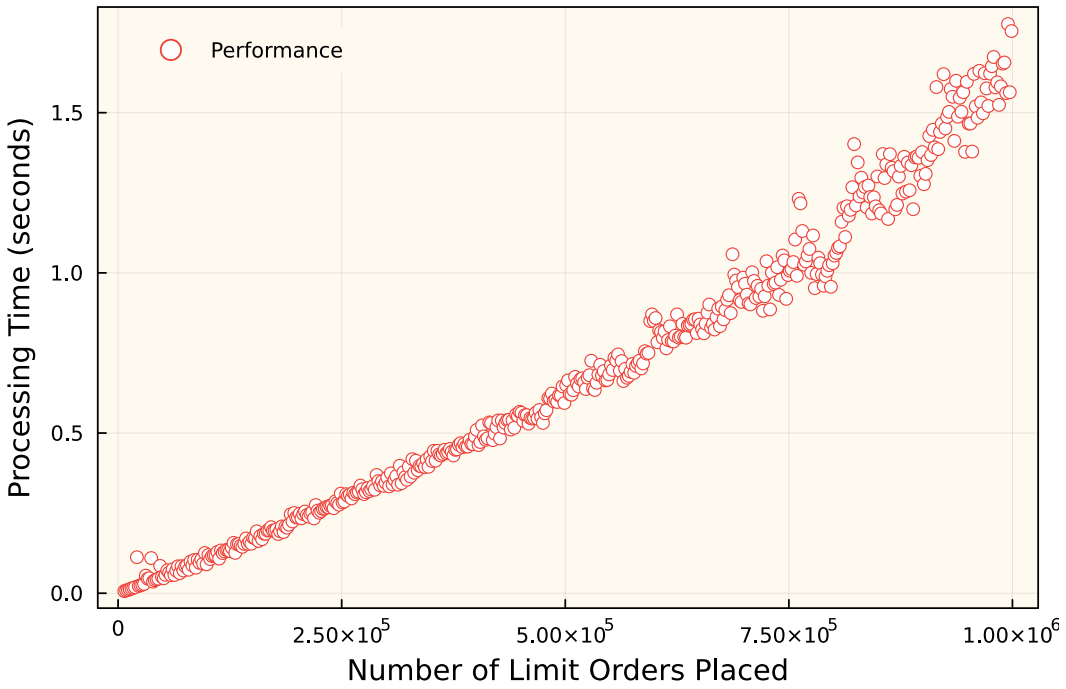

Figure 2.2: The relationship between order book construction time and the total number of placed limit orders

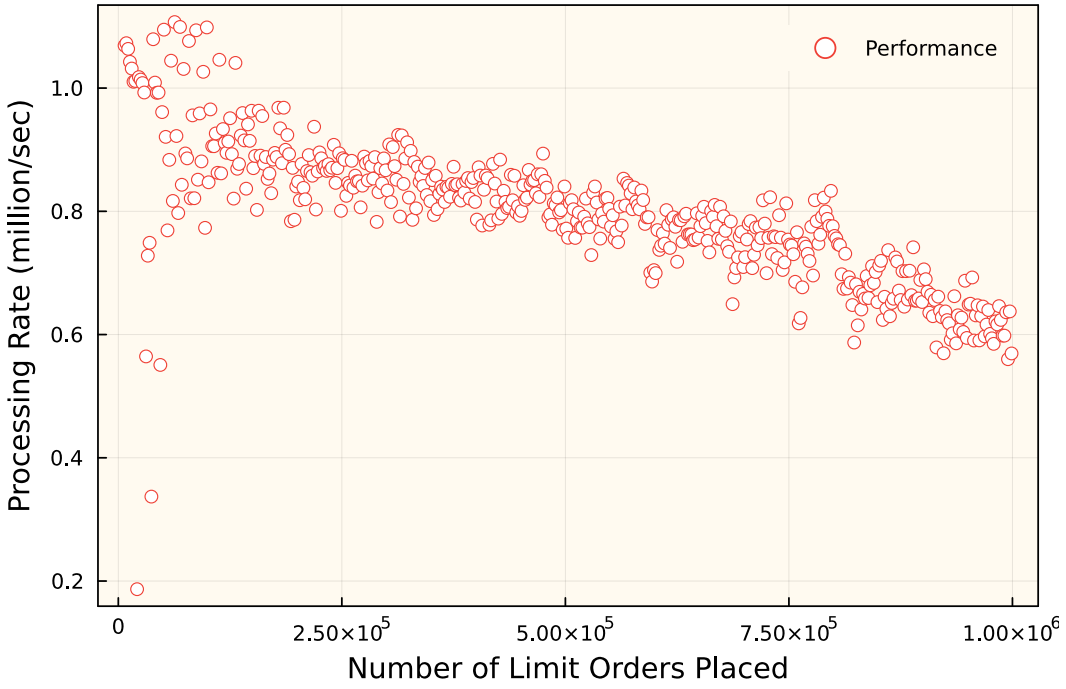

Figure 2.3: The relationship between message processing rate and the total number of placed limit orders

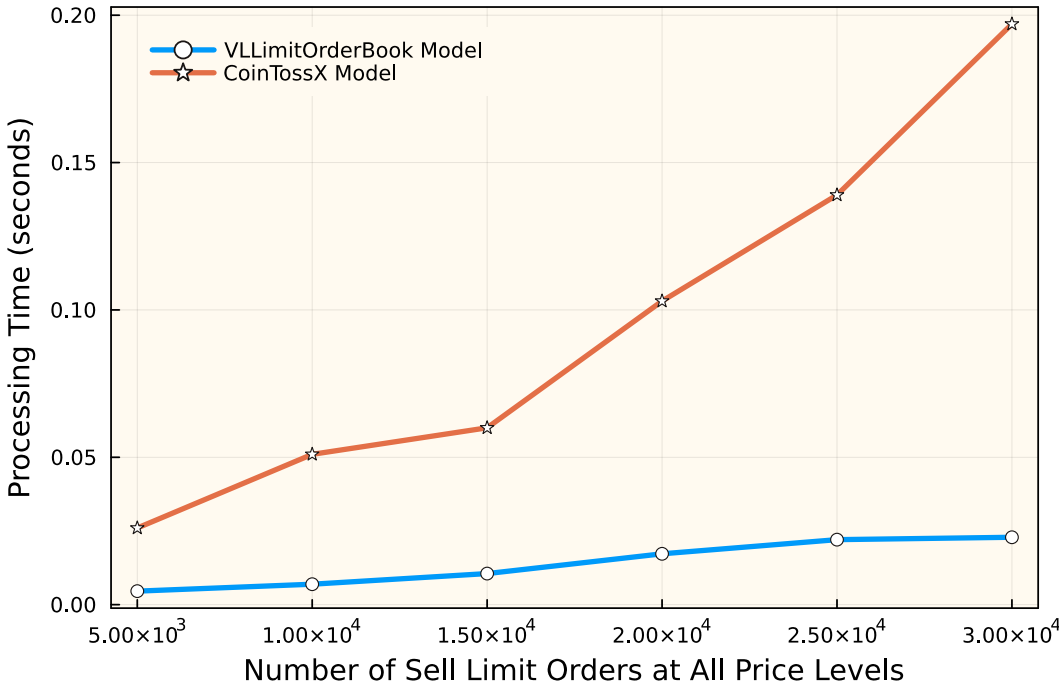

Figure 2.4: Model Performance Comparison: VLLimitOrder and CoinTossX

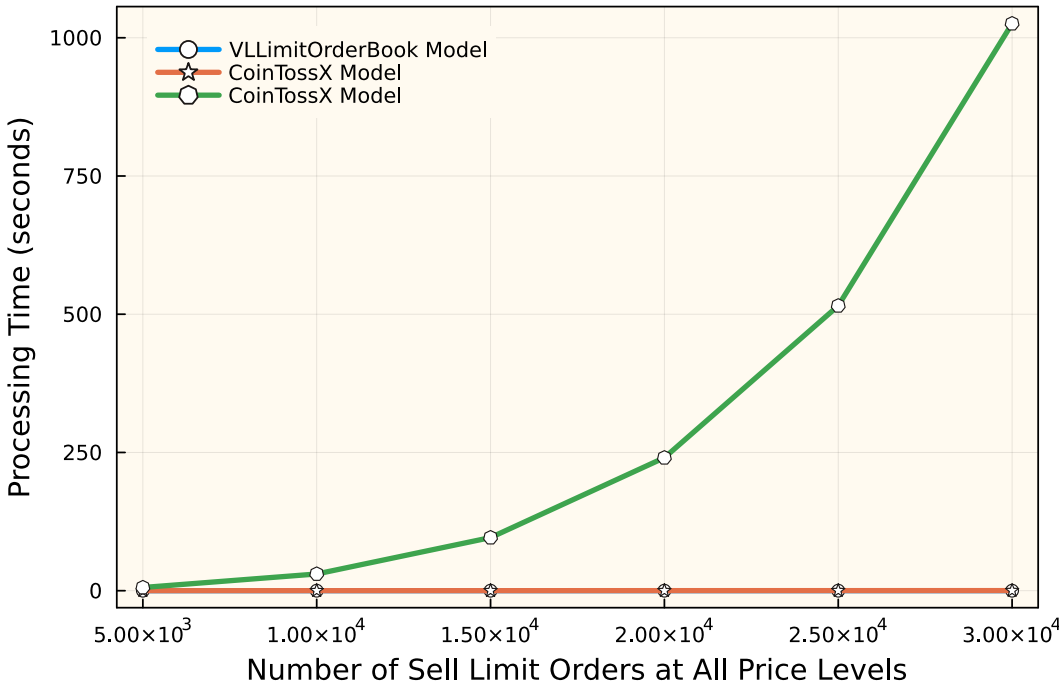

Figure 2.5: Model Performance Comparison: VLLimitOrder, CoinTossX, and BSE

Jericevich et al.[39] mention two ways to evaluate an order book system: throughput per second (TPS) and latency. TPS refers to the number of order operations per second, which includes various order behaviors such as submitting limit/market orders, canceling orders, etc. They used the Hawkes process to generate different order types and then derived other order attributes such as volume and prices. Their latency was calculated based on throughput. In this thesis, our performance was compared by the processing time of placing the same number of limit orders (Figs 2.4, 2.5). Our VLLimitOrderBook model demonstrates the best performance, taking the least amount of time to process the same number of limit orders. BSE takes the longest time because it builds an order book from the beginning every time a new order comes in. The implementation of VLLimitOrderBook and CoinTossX models is similar; the main difference is that CoinTossX uses a B-tree to store price levels, while VLLimitOrderBook employs an AVL tree. AVL tree data structures enable efficient look-up operations, particularly when executed in memory. However, if memory is limited or the transaction is sufficiently large, we could switch to the B/B+ tree. Another point worth noting is that these three models were implemented in different programming languages: BSE was implemented in Python and CoinTossX was implemented in Java. An additional factor contributing to the superior performance of our model compared to the other two is that Julia typically exhibits better performance than Python and Java. Future work could involve implementing and comparing the models using the same programming language to further evaluate their performance.

Effect of Different Insert Positions

Those tests were conducted to determine whether inserting an order at different price levels would affect processing time. To do so, we built an order book with one million price levels on both sides and measured the time it took for one insertion at different levels. Results indicated that the processing time was approximately the same, regardless of the price at which the order was inserted (Fig. 2.6). The mean processing time was $4.723 * 10^{-7}$, with a maximum value of $3.698 * 10^{-6}$, a minimum value of $3.478 * 10^{-7}$, and a standard deviation of $1.127 * 10^{-7}$. Our range of price levels is from 20 USD to 200 USD. When we insert a limit order into the order book, our model first searches the price level and then appends that order at the end of that price queue. Hence the difference could be the time to locate that specific price key, which has a maximum of $O(logn)$ time complexity, n is the total number of price levels. This explains why 70% of mean processing time varies from approximately $3.5 * 10^{-7}$ to $5.8 * 10^{-7}$ seconds, although the maximum and minimum values vary at each execution. Each data point is an average of five identical experiments, so the outliers could be attributed to randomness in the system, given that the only variable was the insertion of limit orders at different price levels.

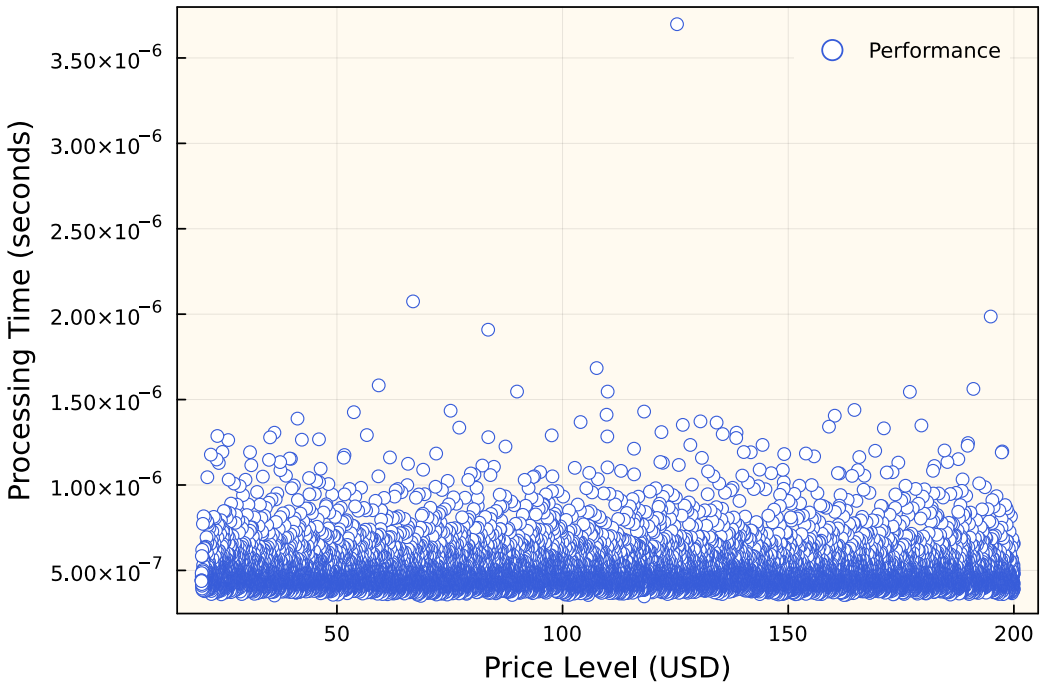

Figure 2.6: The effect of inserting a limit order at different price levels

Depth of Order Book

We also investigated whether different depths of the order book have an impact on construction time. The depth of the order book refers to the price level in the book. For this test, we placed orders sequentially across different price levels, with one million orders in each test. This section is different from the last section, where we explored the effect of inserting a single limit order in a non-empty order book. In contrast, this section explores the impact of inserting the same number of limit orders in an empty order book at various price levels. Results showed the order book construction time is minimized when placing orders at a limited price level (left part) or constantly generating orders at the new price levels (right part) (Fig. 2.7). On the right side of the graph, a notable decrease in processing time was observed. This was attributed to the utilization of an AVL tree data structure to store the price key and a decreased number of

order append at the end of that price queue. With this implementation, appending the higher price level of orders into the tree did not need longer processing time. Conversely, the processing time on the left side of the graph decreased as the number of price levels decreased. This was due to the fact that with very few price levels, inserting a new order is similar to appending an order at the end of a order queue. To further examine this, we also compared inserting the same amount (one million) of data to a queue and AVL set, and the AVL set (0.055 seconds) requires more processing time than a queue (0.005 seconds). This explains why the left part of a graph requires a shorter processing time than the right side. The middle part of the graph experiences a hump shape, indicating that processing time increased as search time on the price key and the need for self-balancing as well as appending the order at the end of each price level, resulting in a longer processing time than the two sides.

Figure 2.7: The relationship between order book construction time and order book depth (price level)

Latency Test for Inter-processes

A latency test was conducted to study the throughput of order submission. We used JSON files for communication and measured the latency of sending and receiving rate of messages between two machines on the local area network, as well as between two processes on the same machine (Figs. 2.8, 2.9, 2.10, 2.11, 2.12, 2.13, 2.14, 2.15). The results showed that transmission between processes in the Linux machine could have the lowest latency in one million transmissions, which is nearly 2.8 * 10^{10} nanoseconds (2.2 million/min). The highest throughput was nearly 5.4 * 10^{10} nanoseconds (1.1 million/min) in which Mac as the server machine and Linux as the client machine (Fig. 2.14). The reason for the gap is that there is a maximum sending buffer limit in the WebSocket transmission, causing the I/O to be blocked and the sender to pause for a few moments. If we deliberately slow down the receiver side to process one message per second, there will be a noticeable pause for the sender to send the message (Figs. 2.8, 2.11).

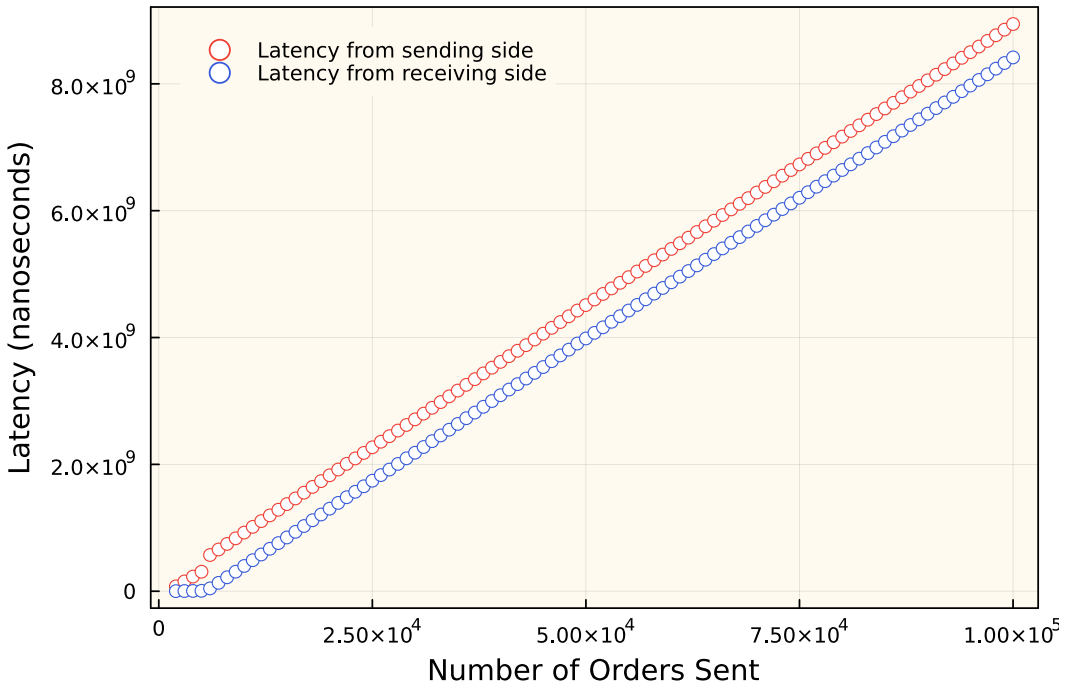

Figure 2.8: Latency of 100 messages transmission between two processes in Mac

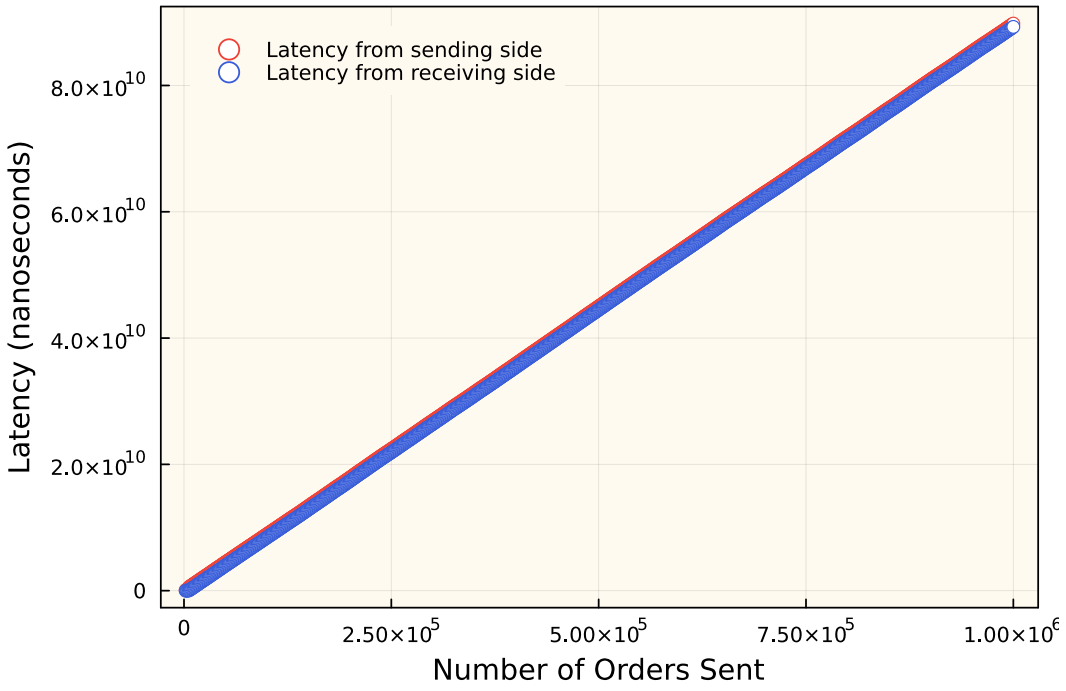

Figure 2.9: Latency of 1000 messages transmission between two processes in Mac

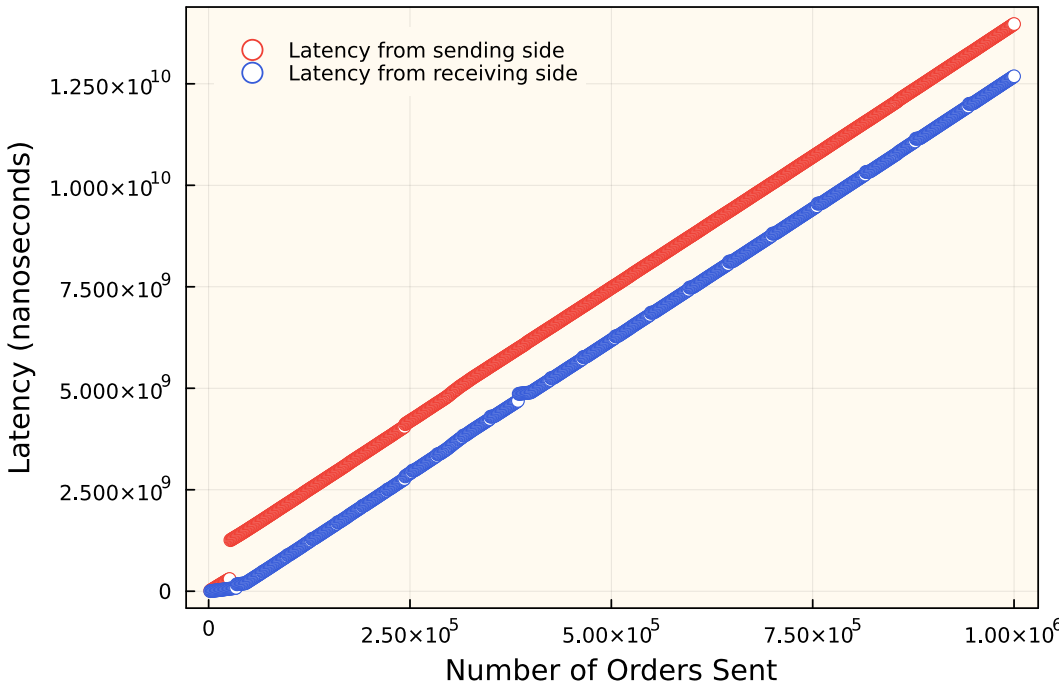

Figure 2.10: Latency of 1000 messages transmission between two processes in Linux

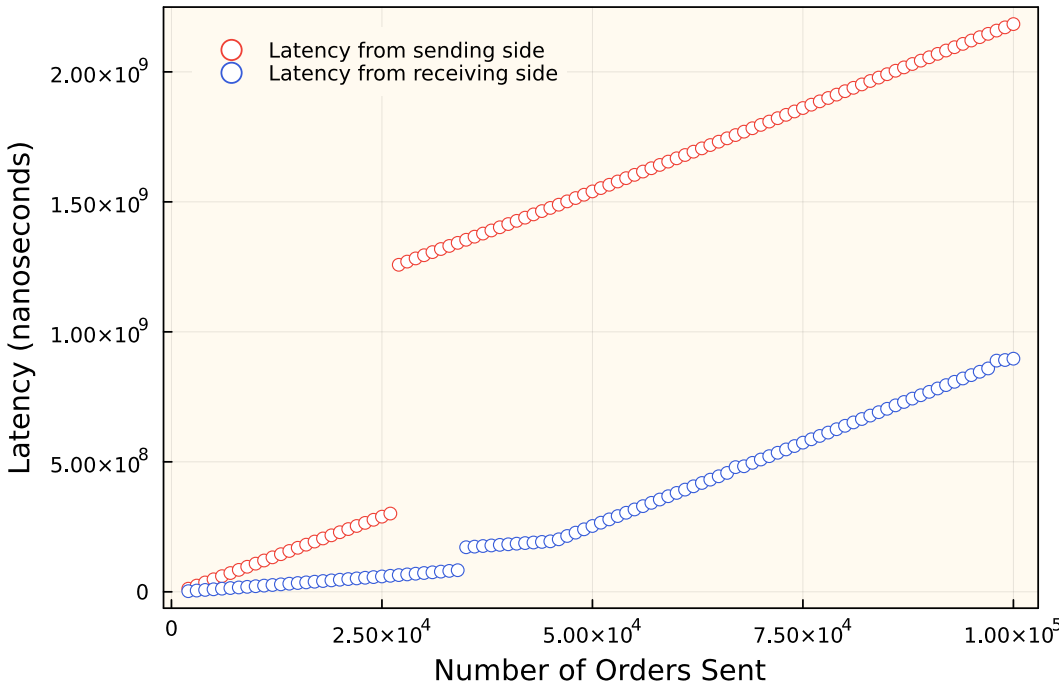

Figure 2.11: Latency of 100 messages transmission between two processes in Linux

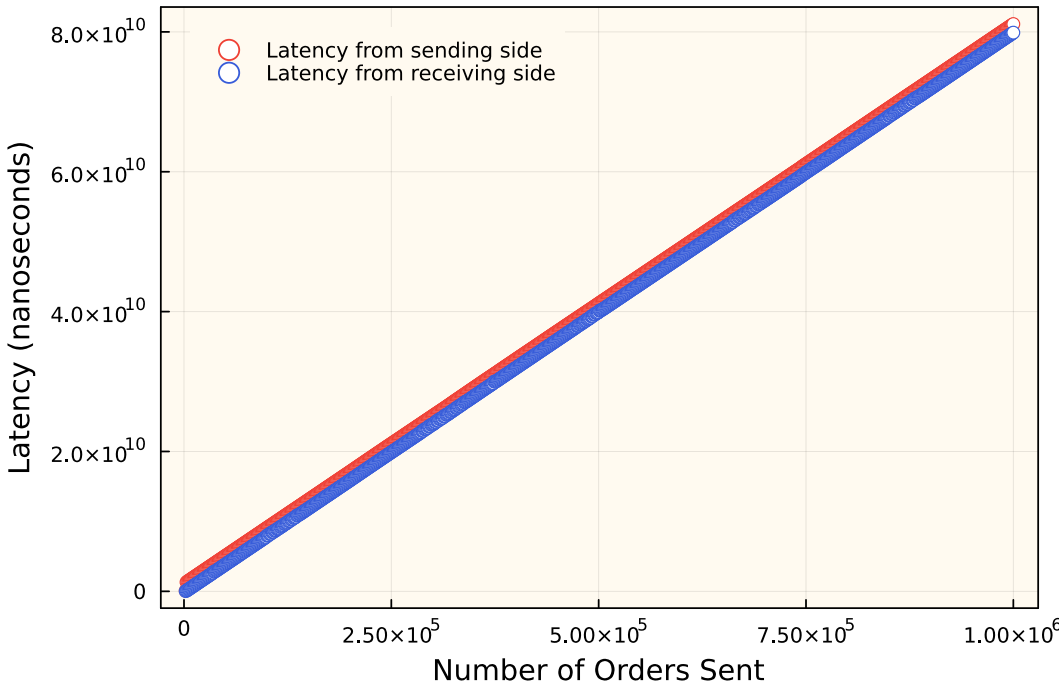

Figure 2.12: Latency of 1000 messages transmission between a server process in Linux and a client process in Mac

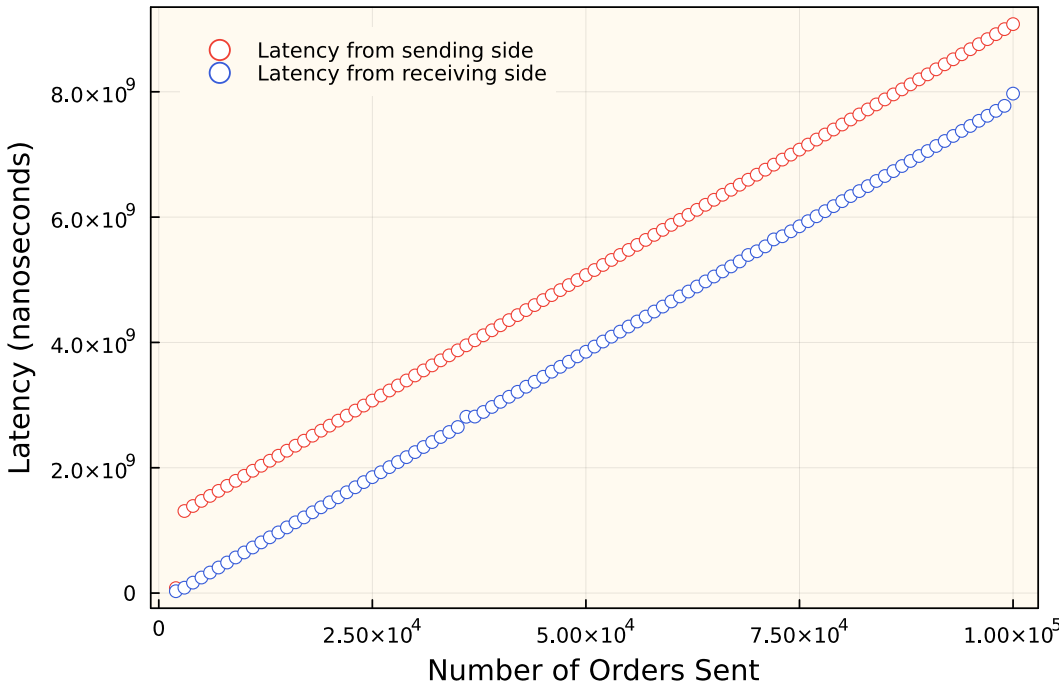

Figure 2.13: Latency of 100 messages transmission between a server process in Linux and a client process in Mac

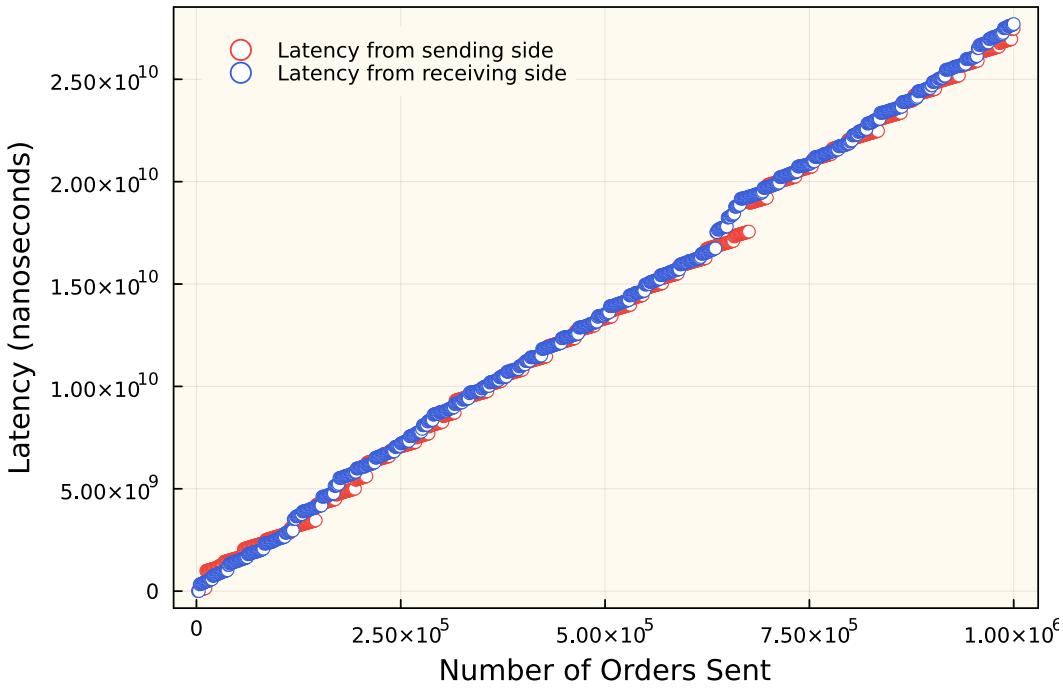

Figure 2.14: Latency of 1000 messages transmission between a server process in Mac and a client process in Linux

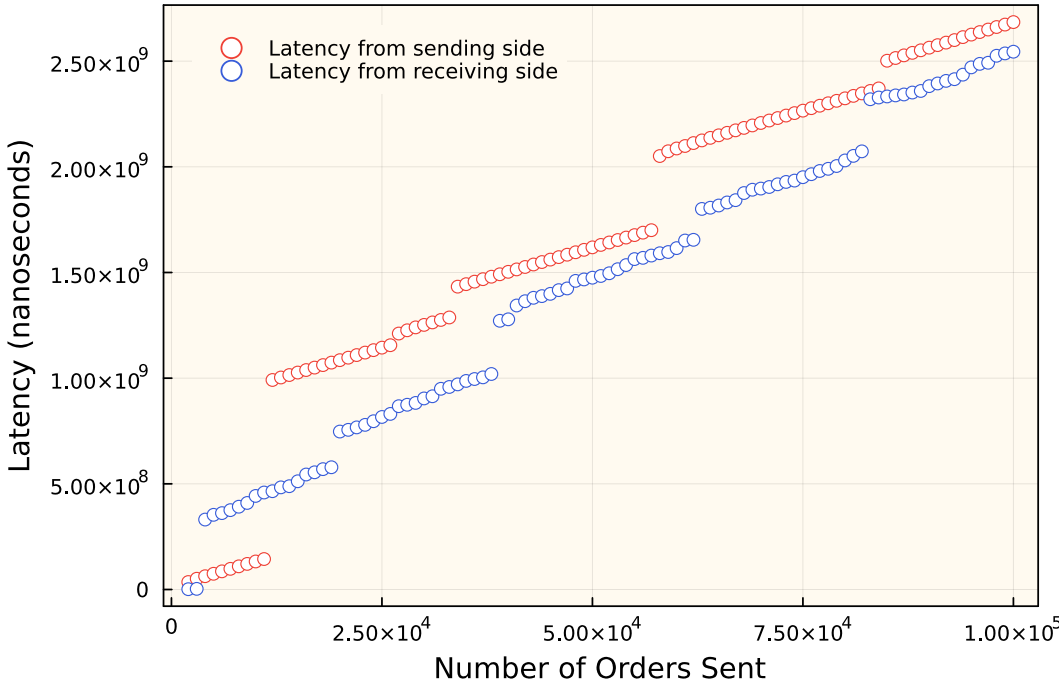

Figure 2.15: Latency of 100 messages transmission between a server process in Mac and a client process in Linux

Concurrency Performance Evaluation

The average processing time required to construct the limit order book was investigated using different numbers of concurrent processes (Fig. 2.16). The results revealed a non-linear relationship between processing time and the number of processes employed. Specifically, processing time decreased linearly as the number of processes increased up to 15-20, after which there was no significant change in gradient.

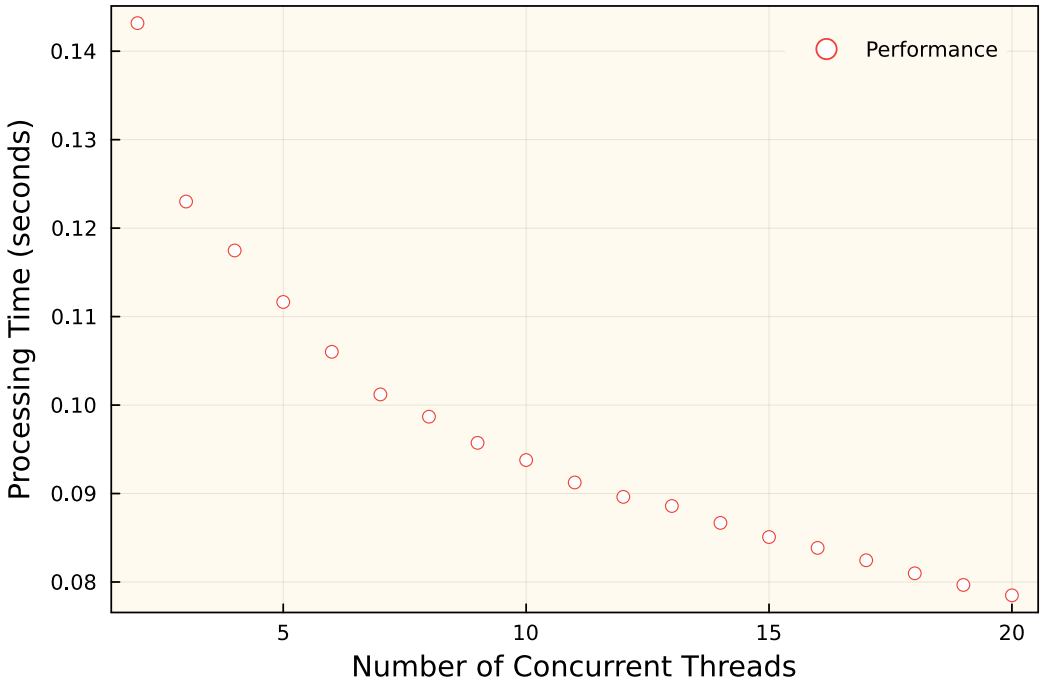

Figure 2.16: Effect of concurrent processes on order book construction time with one million orders

To further evaluate the impact of concurrency, a comparison test was made between the performance of 20 concurrent processes and that of a single process (Figs. 2.18, 2.20, 2.17, 2.19). The findings indicated that the model exhibits strong concurrency capabilities, as opening new processes could substantially reduce processing time. These insights can be valuable for improving simulation speed

in practical applications.

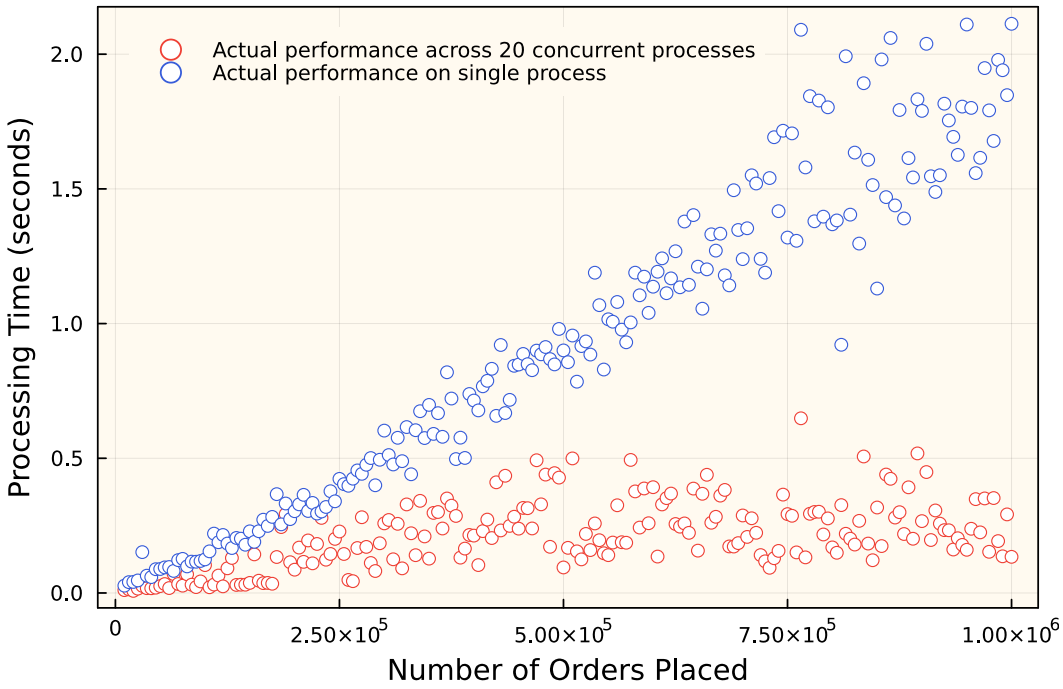

Figure 2.17: Comparison of actual processing time for order placement across 20 processes and single process

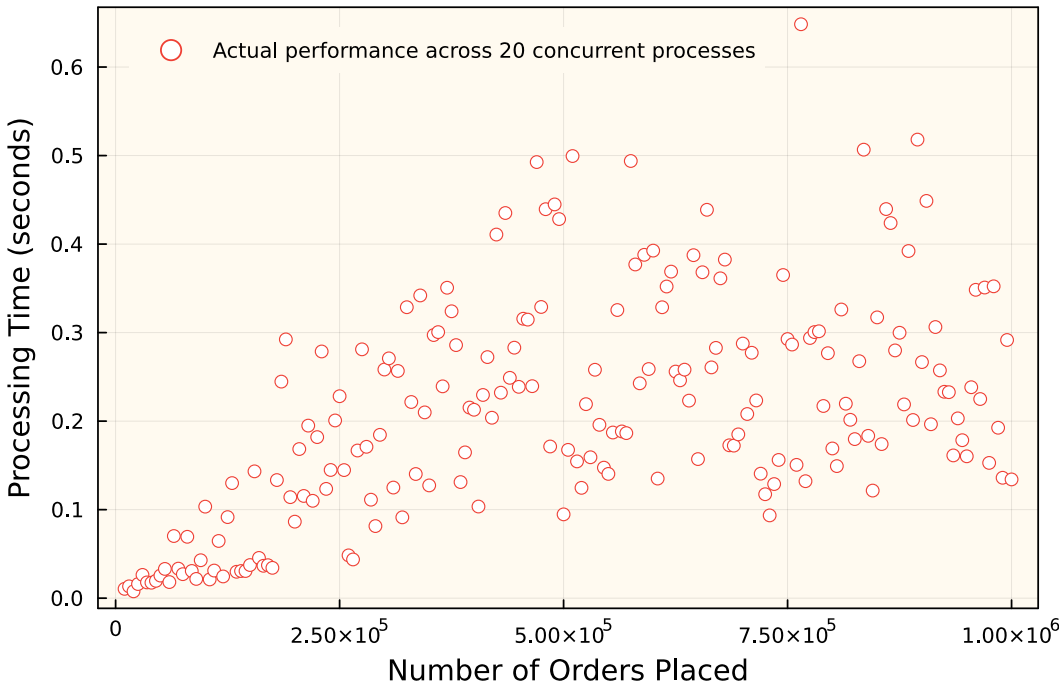

Figure 2.18: Actual processing time for order placement across 20 processes

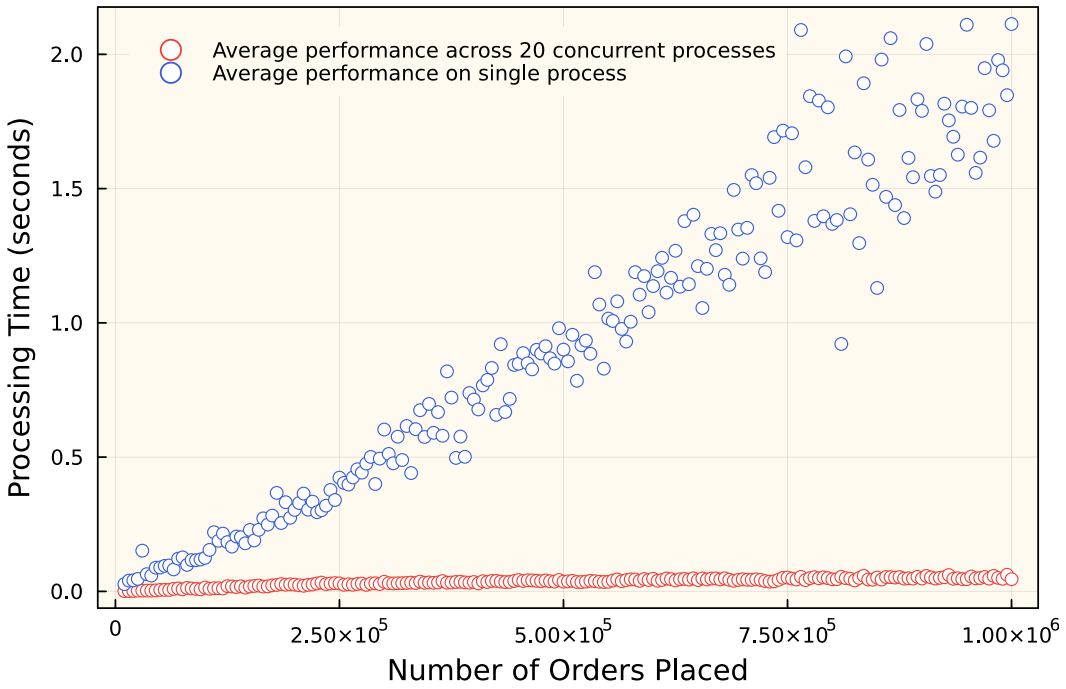

Figure 2.19: Comparison of average processing time for order placement across 20 processes and single process

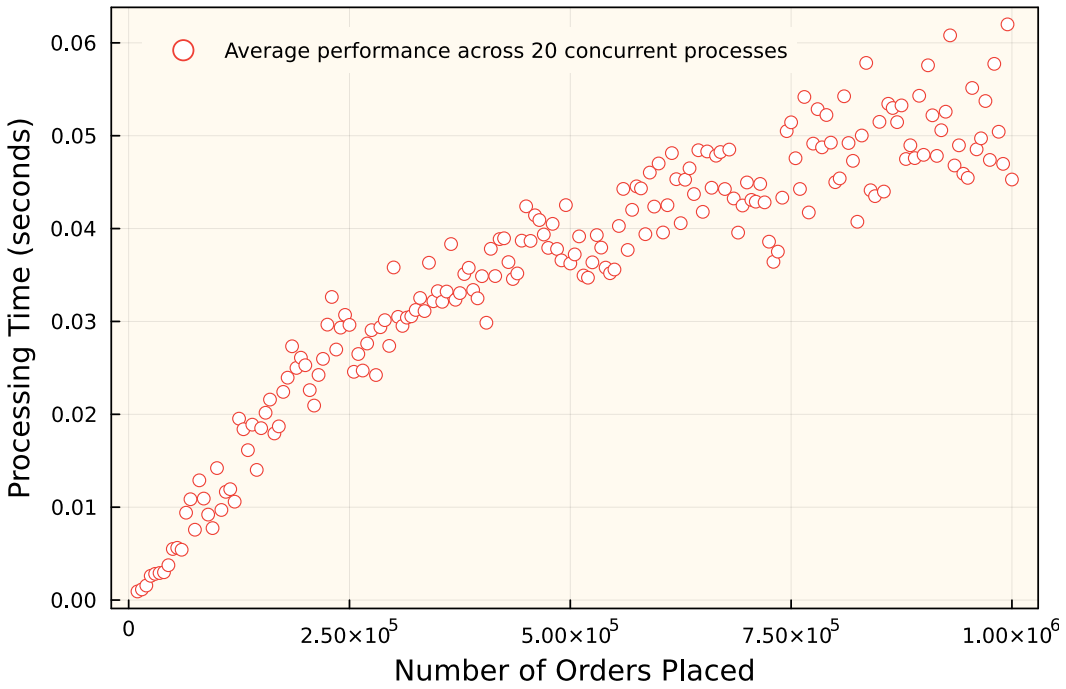

Figure 2.20: Average processing time for order placement across 20 processes

2.4.2 Nasdaq TotalView Data Examination

Time of Simulation

ITCH Book Constructor[7] package was utilized to extract and transform Nasdaq TotalView ITCH data feed to a CSV order message file. To test VLLimitOrderBook.jl package, we picked up some active tickers in the market to verify the correctness and processing time (Table 2.2). Specifically, we chose AMZN, AAPL, QQQ, SPY, TSLA, MSFT, and INTC. The raw data can be accessed via FTP at ftp://emi.nasdaq.com/itch/, we utilized data on Jan 30th, 2020. We could find that INTC has the highest message processing rate, 2.2 million/second, while QQQ has the lowest message processing rate which is 0.995 million/second. The reason is that INTC has 1/3 of the total number of messages for QQQ. So the instance of INTC is small, which results in a short message processing rate. As analyzed in the *Time of Constructing Order Book* section, the larger the total number of LOB instances, the lower the message processing rate. In the meantime, the maximum order book price level for INTC is 1312, while the maximum order book price level for QQQ is 2324. As we discussed in the *Depth of Order Book* section, a larger order book price level (depth) will result in a lower processing rate. Although the unit processing time decreases at the end of the graph, this should not be it since, in reality, we did not always generate orders at new price levels (Fig. 2.7).

Table 2.2: Statistics from selected tickers' simulation including the number of trading messages, average evaluation time, and corresponding single message processing rate.

Name	Symbol	Number of Messages	Average Evaluation Time (ms)	Message Processing Rate (million/sec)
Intel	INTC	1601350	707	2.265
Apple	AAPL	2008467	1059	1.897
Microsoft	MSFT	1854140	948	1.956
SPDR 500 ETF	SPY	4468109	3792	1.179
Invesco QQQ	QQQ	4754517	4777	0.995
Amazon	AMZN	670233	398	1.684
Tesla	TSLA	1030765	624	1.652

Last Trading Price Visualization

To visualize the price changes for the order book, we first validated whether the order book got executed as expected. To verify this, we first used dictionary that only accepts the type ″E″ and type ″C″ in the parsed order messages. Type ″E″ is an order execution at the original price, and type ″C″ is an order execution in a different price. More order types can be seen in the *Order Type Composition* section. We use the timestamp as key and price as the value, this could keep track of the lasted executed price at a certain time stamp. Some order messages might come at the same time, the latter execution price would override the previous order execution price. Then we used another dictionary to keep track of the last market order execution. Our results showed that the two dictionaries were equal for the above tickers, which means our order book get executed as expected. Then we plotted the price changes of those tickers (Figs. 2.21, 2.23, 2.25,

2.27). [1] The historical data for those tickers from Yahoo Finance also provide another way to verify the correctness of the price changes on that date (Table 2.3).

Table 2.3: Historical prices from Yahoo Finance on Jan 30, 2020

Name	**Symbol**	**Open**	**Close**	**High**	**Low**
Invesco QQQ	QQQ	220.38	222.60	222.70	219.69
SPDR 500 ETF	SPY	324.36	327.68	327.91	323.54
Tesla	TSLA	42.16	42.72	43.39	41.20
Apple	AAPL	80.14	80.97	81.02	79.69

We generated error plots for the above-mentioned tickers to evaluate our model's performance before and after modifications to order types and order traits (Figs. 2.22, 2.24, 2.26, 2.28). The errors did not propagate as time goes on since the model would break and stop processing the remaining messages, we have to throw such errors and continue to execute our program. Those scenarios include when canceling an order that does not exist (that order might be treated with higher priority and get executed), or the bid/ask price will cross due to some previous orders at specific price levels not being executed. Our analysis shows that most of the order executions closely resemble the actual last trading prices. Further details of the reasons and methods to eliminate errors can be found in the *Discussion* section.

[1] There is a disparity of TSLA stock prices since one TSLA share bought before August 31st, 2020 would equal to 15 TSLA shares today[16].

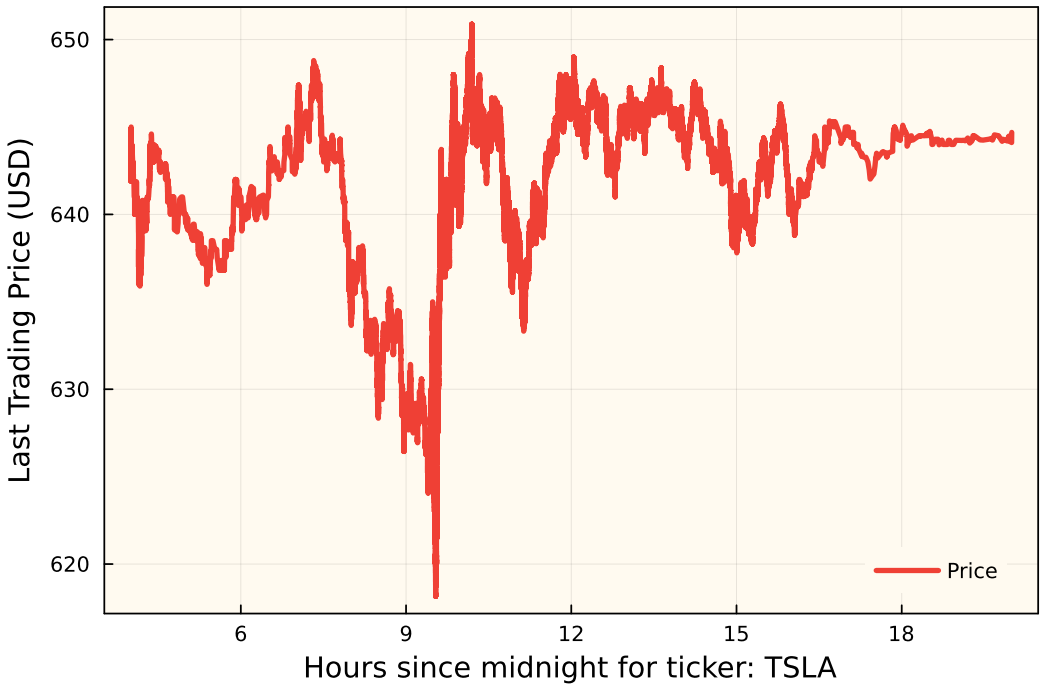

Figure 2.21: The variation of last trade price for ticker TSLA on Jan 30, 2020, on Nasdaq

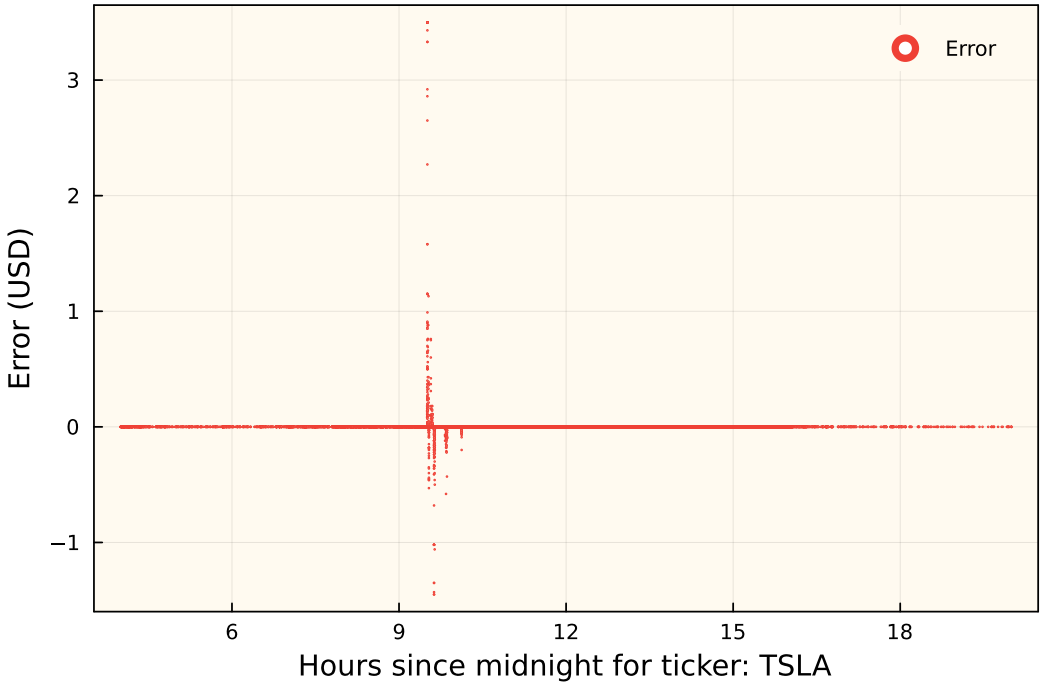

Figure 2.22: The error of last trade price for ticker TSLA on Jan 30, 2020, on Nasdaq

Figure 2.23: The variation of last trade price for ticker AAPL on Jan 30, 2020, on Nasdaq

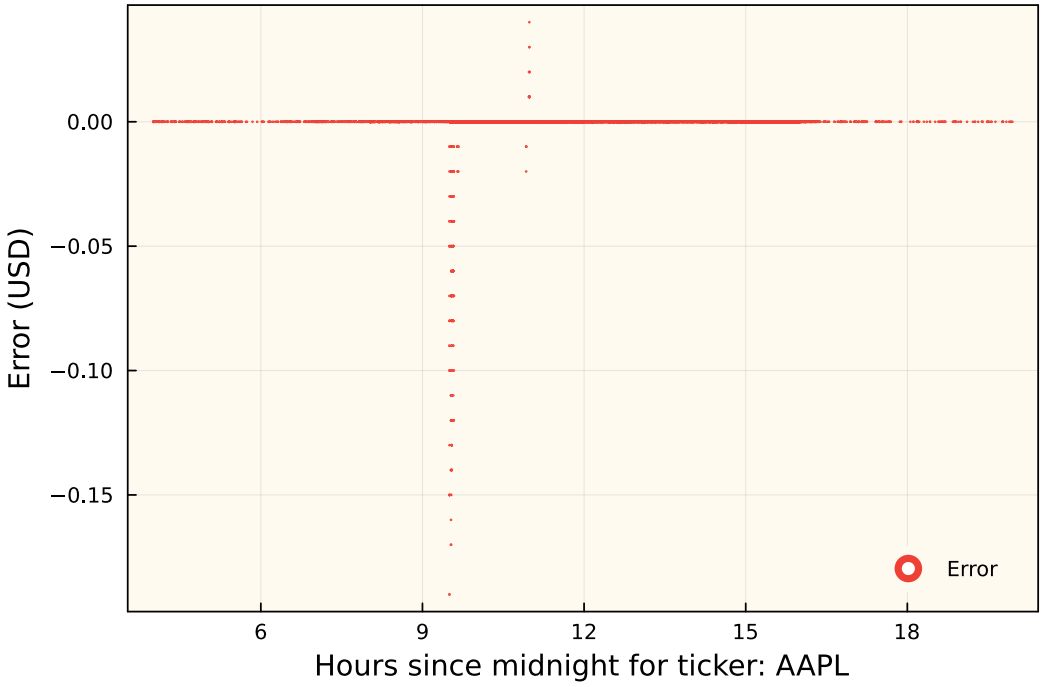

Figure 2.24: The error of last trade price for ticker AAPL on Jan 30, 2020, on Nasdaq

Figure 2.25: The variation of last trade price for ticker QQQ on Jan 30, 2020, on Nasdaq

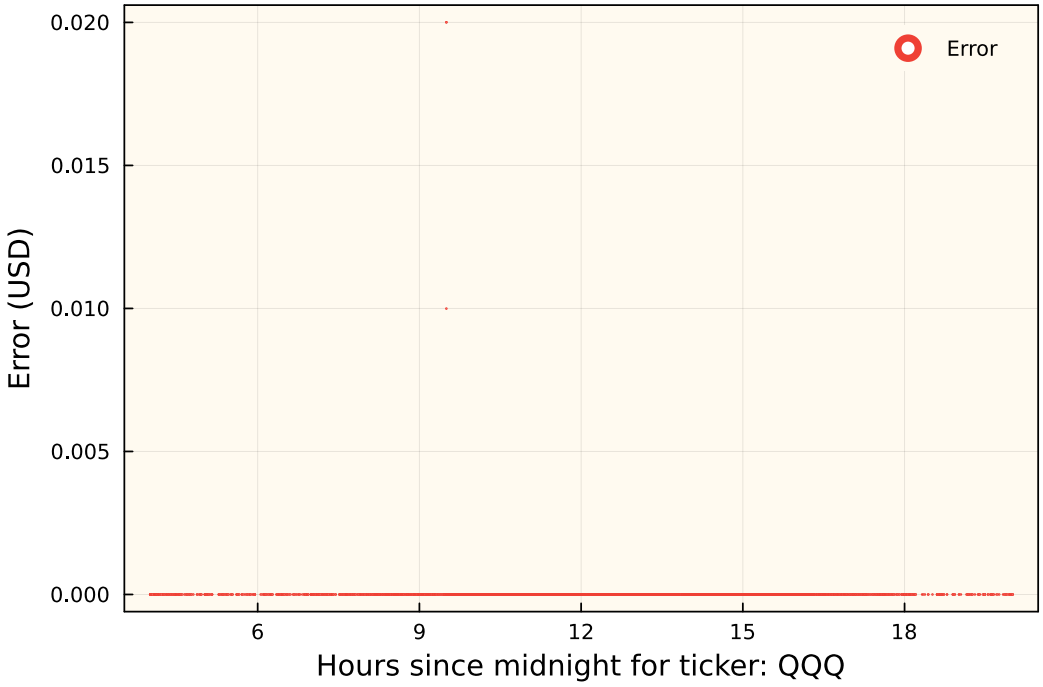

Figure 2.26: The error of last trade price for ticker QQQ on Jan 30, 2020, on Nasdaq

Figure 2.27: The variation of last trade price for ticker SPY on Jan 30, 2020, on Nasdaq

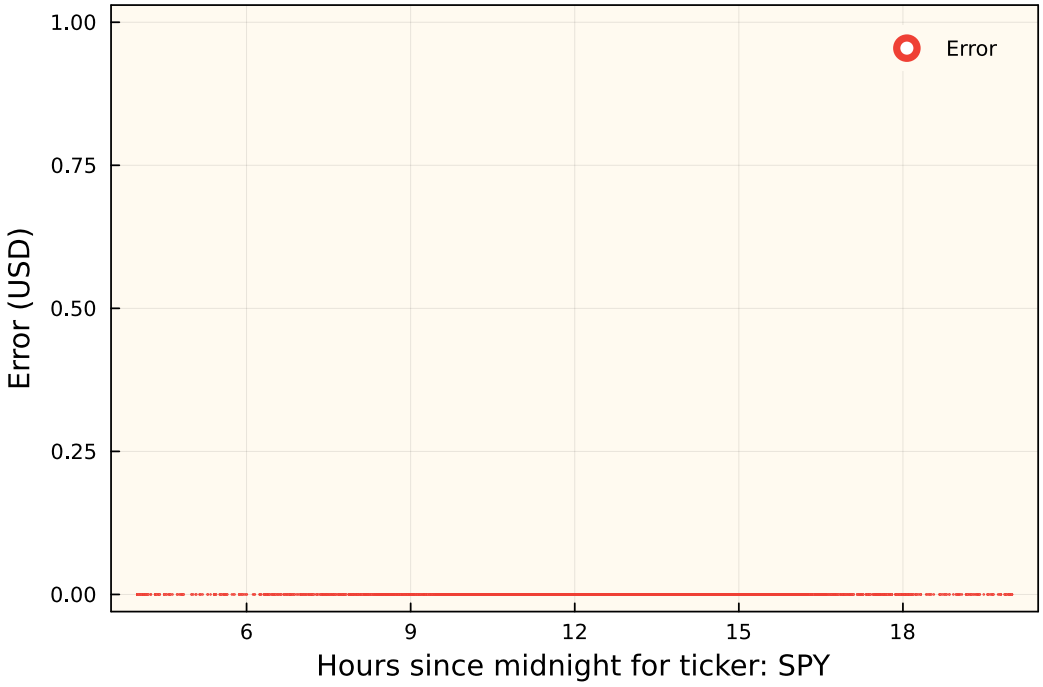

Figure 2.28: The error of last trade price for ticker SPY on Jan 30, 2020, on Nasdaq

2.5 Discussion

2.5.1 Order Type Composition

Understanding the composition of order types can greatly assist people in determining the primary focus of the package. We picked up several tickers and calculated their order message type composition. Results showed that order submissions and cancellations were the primary order activities, which take up to half of all order message types (Table 2.4). Therefore, the discussion and experimentation will be focused on these two behaviors, specifically submitting and canceling limit orders.

Table 2.4: Nasdaq order types statistics (parsed from NASDAQ ITCH50[3] data feed, where A represents adding an order, C represents an order execution with a different price, E represents an order execution with the same price, P represents a hidden order execution, D represents an order cancellation, R represents an order replacement)

Name	Symbol	Number of Messages	A	P	C	D	E	R
Intel	INTC	1601350	746650	4437	935	725148	31462	92717
Apple	AAPL	2008467	907157	15114	227	879475	55168	151325
Microsoft	MSFT	1854140	842425	20431	819	785190	85854	119420
SPDR 500 ETF	SPY	4468109	2068166	11587	1441	2012577	79469	294868
Invesco QQQ	QQQ	4754517	2261270	4425	1255	2229937	44194	213435
Amazon	AMZN	670233	284762	32188	29	261545	36055	55653
Tesla	TSLA	1030765	422317	59971	560	361561	90831	95524

2.5.2 Complexity Analysis

We used an AVL tree data structure to store the order queues, with price serving as the key (Fig. 2.1). In the worst cases, search, query, and delete operations can

achieve up to *O(log N)* time complexity. According to Jenq et al.[38], balanced trees are effective for managing many price levels and high volumes of orders. AVL Tree data structures enable efficient look-up operations, particularly when executed in memory. However, if memory is limited or the transaction is sufficiently large, the use of a B/B+ tree for data storage can be considered.

The complexity of inserting a limit order is *O(log N)*, where N represents the number of price levels in the order book. It takes *O(log N)* time to insert the price key to the order book, followed by a constant time to append the order at the end of the order queue at the corresponding price level. The complexity of canceling a limit order is determined by the number of price levels in the order book (represented by N) and the length of the order queue at the specific price level (represented by n). Specifically, canceling a limit order has a time complexity of *O(log N + n)*. First, it takes *O(log N)* time to locate the price key in the order book. Next, the order with the given order *id* is deleted, which takes linear time proportional to the length of the order queue (represented by n). This applies to both partially canceled and fully canceled order methods in the package. For submitting a market order at one price level, the complexity is *O(n)*. The orders with the top price level are matched. The worst case is to transverse all orders in that order queue at the top price level since the former could be non-displayed orders, which always have lower priority than displayed orders.

2.5.3 Validation with Nasdaq Historical Data

How do we use this data

The ITCH Book Constructor[7] package was employed to parse Nasdaq TotalView ITCH 50 data into input messages. Subsequently, the output data was converted to CSV file, which serves as the order feed for the validation process.

How do we make changes accordingly

To validate the correctness of the model, Nasdaq TotalView ITCH 50 data was utilized to test the model. Additionally, changes were made to enhance the functionality of the model. The *Order* component was modified to include *Display* and *Non-Display* fields. The inclusion of these fields is necessary because orders at the same price level may have different priorities based on their display or non-display features. Orders with a display feature always have a higher execution priority than non-displayed orders, even if this displayed order has joined the order book at a later time. Therefore, some orders can be executed at a higher priority even if they joined the order queue later.

An *allowlocking* boolean field was added to the *OrderTrait* component because some orders, such as pegged orders, allow prices to lock (bid/ask price reached the same level) between the bid/ask spreads as they are executed. It is impossible to know this until specific messages are received from the data feed. Changes are made according to the specific data feed messages to ensure that some orders are executed correctly at different prices.

The *OrderTrait* component was modified to be non-static and mutable since

different orders may have various order traits. It is impossible to distinguish this difference from the order feed of Nasdaq TotalViewITCH 50 data. For example, if a pegged order comes to the order book, it can have price changes in the later execution, which may cause price locking. But in most cases, the best bid/ask price will not be equal to each other.

The *Order* was modified as a mutable component as well. This allows changes to the original order object rather than generating a new object every time. This is useful when orders are partially matched or canceled, as it enhances the performance and efficiency of the order book system.

CHAPTER 3

AN APPLICATION PROGRAMMING INTERFACE (API) FOR STREAMING REAL-TIME MARKET DATA

3.1 Abstract

This project developed a WebSocket API for PQPolygonSDK.jl, a software development kit for Polygon.io financial data warehouse for Julia developers. The WebSocket API can stream trade, aggregate, quote, and limit-up-limit-down messages for all US stocks, and also store data as a txt file in batches. The document also includes data volume and latency analysis, as well as a discussion of the WebSocket structure and its future development.

3.2 Background

Polygon.io provides highly accurate, real-time financial market data and is a leading financial market data platform for market data on stocks, options, futures, and cryptocurrencies. Their aggregated data for stocks has up to one-second resolution. PQPolygonSDK.jl is a software development kit (SDK) for the Polygon.io financial data warehouse for Julia developers [13]. At present, PQpolygonSDK only offers HTTP connection and does not support WebSocket connectivity. However, Polygon.io offers two types of APIs: the first one is an HTTP API for querying the latest data from the server, and the second one is a WebSocket API for streaming real-time data.

HTTP is an application layer protocol in the Internet protocol suite model[5].

It functions as a request-response protocol in the client-server model. A client process submits a HTTP request to a server, which then receives and processes the request and sends a response message back to the client. The response typically contains a status code and response body for the content. If we want to receive the most updated message, one way to achieve this is to constantly make HTTP calls. However, sending the request header every time will consume bandwidth, as only the response body is needed. In the meanwhile, establishing a connection takes time, causing delays in communication. WebSocket resolves these two issues by providing a full-duplex communication on TCP connection [18]. It establishes a connection between the server and the client only once and allows the server to transmit messages back and forth. This enables efficient and real-time data transmission without unnecessary connections.

At present, PQPolygonSDK.jl solely offers a one-time HTTP query connection to Polygon.io. While this provides the most up-to-date data, the response to the query will remain unchanged even as the data updates. It's necessary to develop a WebSocket connection to stream real-time data.

3.3 Contribution

We developed a WebSocket API for PQPolygonSDK.jl for Julia developers. It could stream trade, aggregate, quote, and limit-up-limit-down messages for all US stocks. Additionally, it could store as a txt file in batches. Further details of implementation can be found in the *Discussion* section.

3.4 Results

3.4.1 Estimated storage size

To estimate the data traffic for one day, we picked up different time of the day and streamed data every 10 minutes[14]. We chose some active tickers, SPY and QQQ, to estimate their data volume. We examined quote, aggregate, and trade messages in the stock market. We concluded that no matter which type of data we were streaming, it always reached a peak volume in the time near market opening and market closing. Among all the message types, Quotes messages were the largest, which achieved up to 6.5 * 10^4 kb for 10 minutes at market opening times.

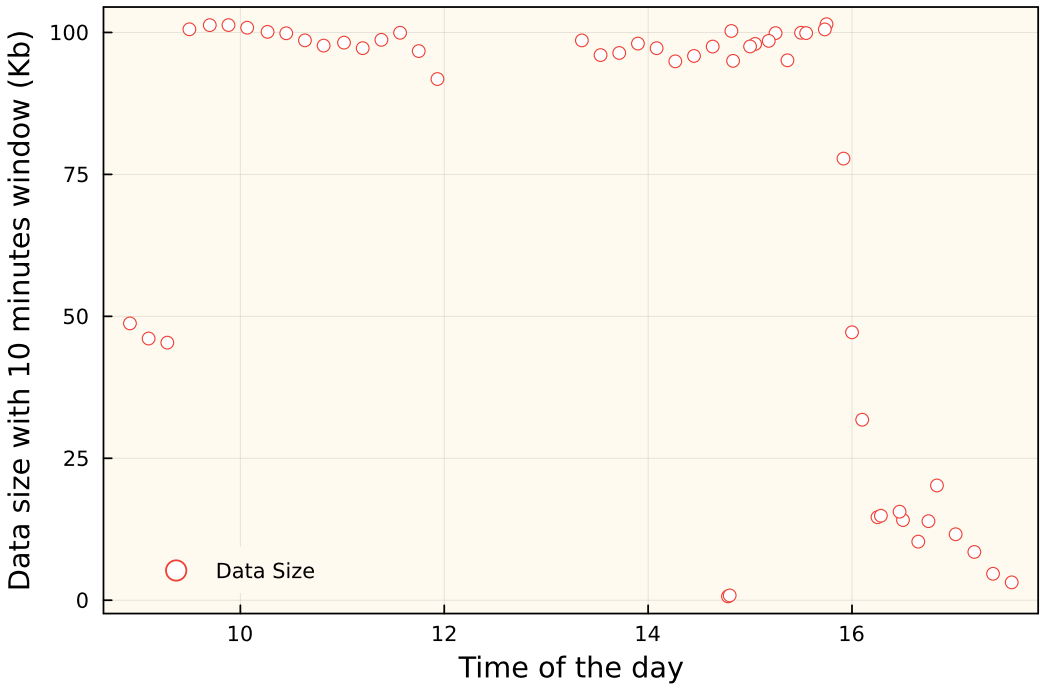

Figure 3.1: Estimated storage size from Polygon.io of aggregates message, ticker QQQ on March 10, 2023

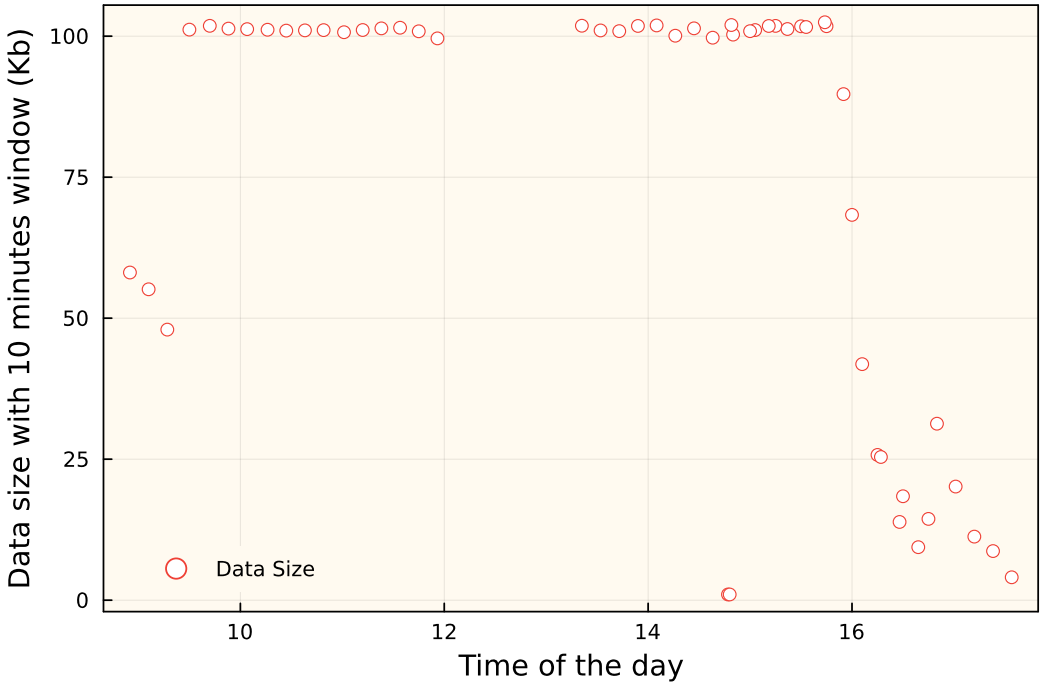

Figure 3.2: Estimated storage size from Polygon.io of aggregates message, ticker SPY on March 10, 2023

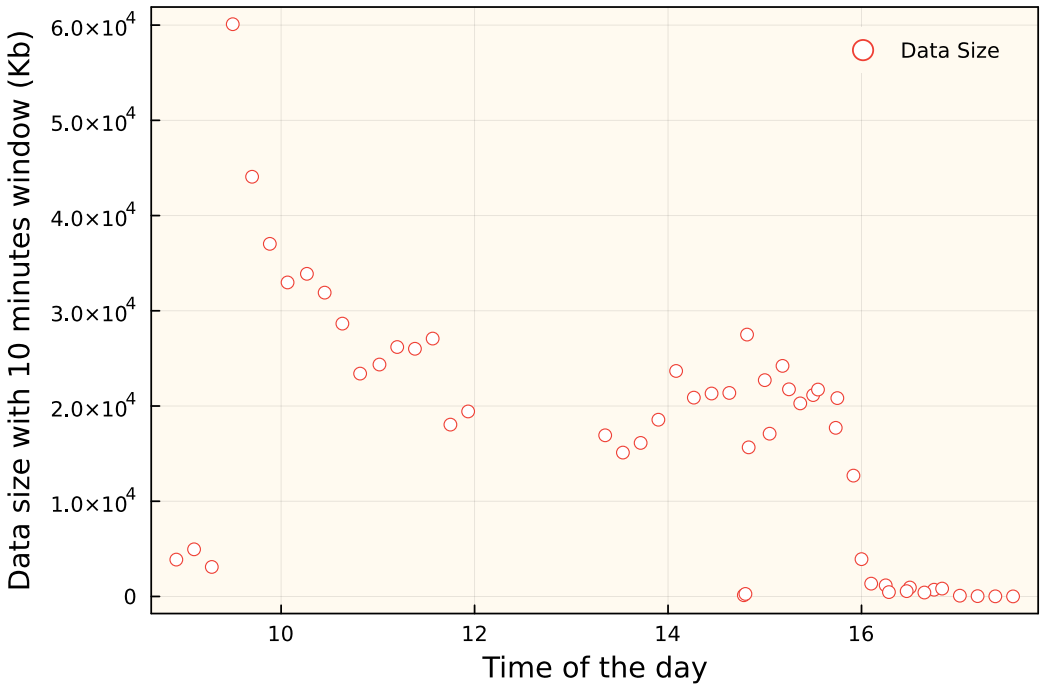

Figure 3.3: Estimated storage size from Polygon.io of quotes message, ticker QQQ on March 10, 2023

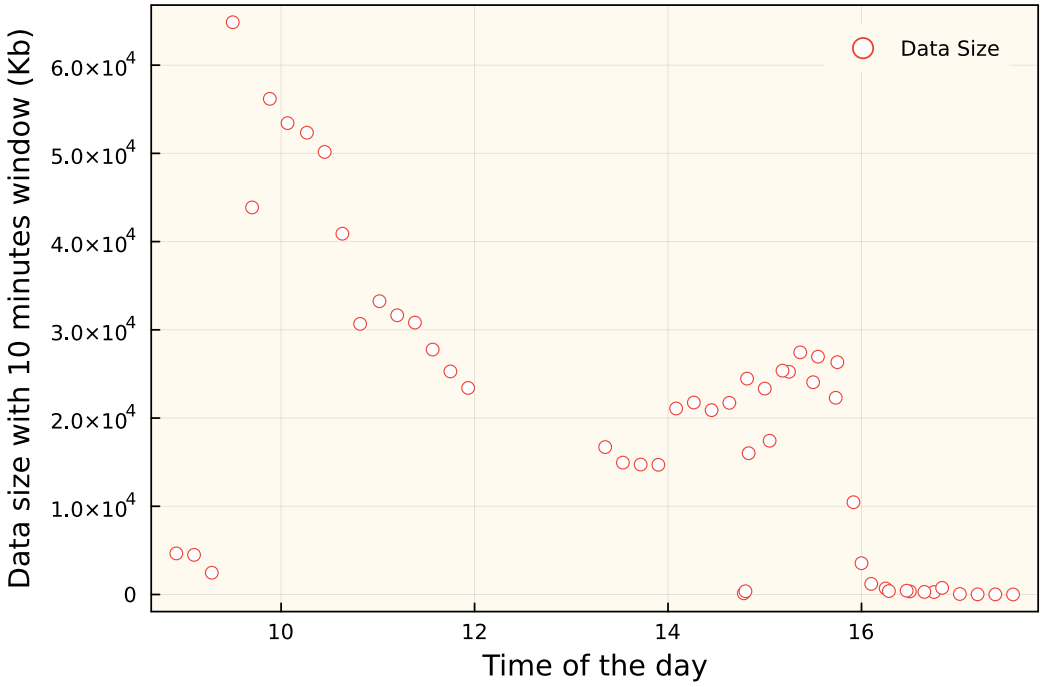

Figure 3.4: Estimated storage size from Polygon.io of quotes message, ticker SPY on March 10, 2023

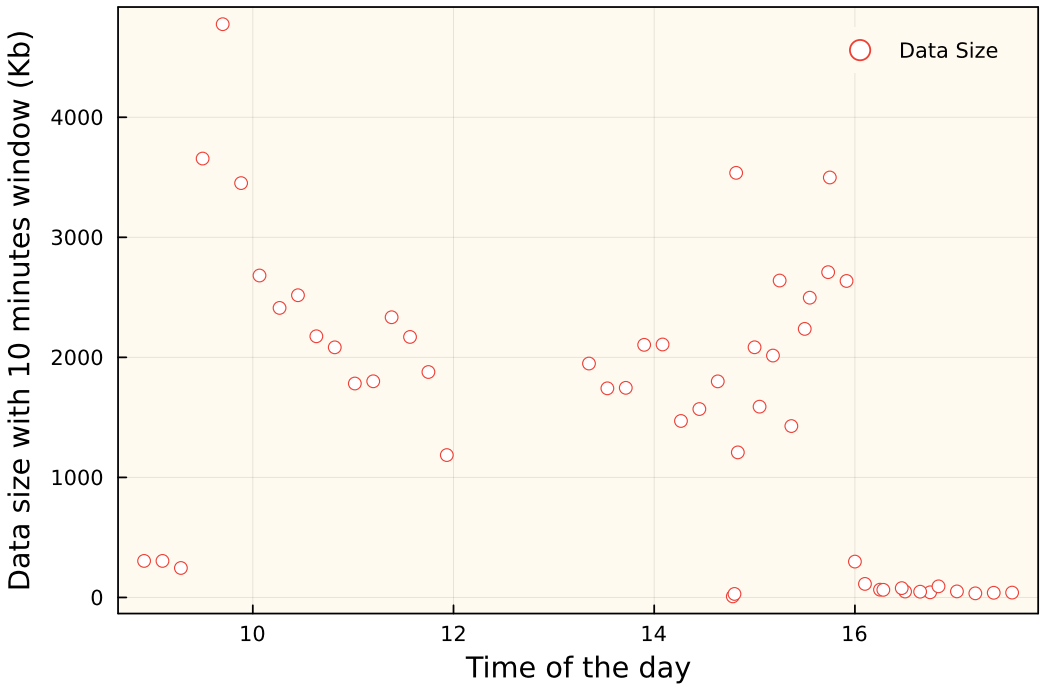

Figure 3.5: Estimated storage size from Polygon.io of trades message, ticker QQQ on March 10, 2023

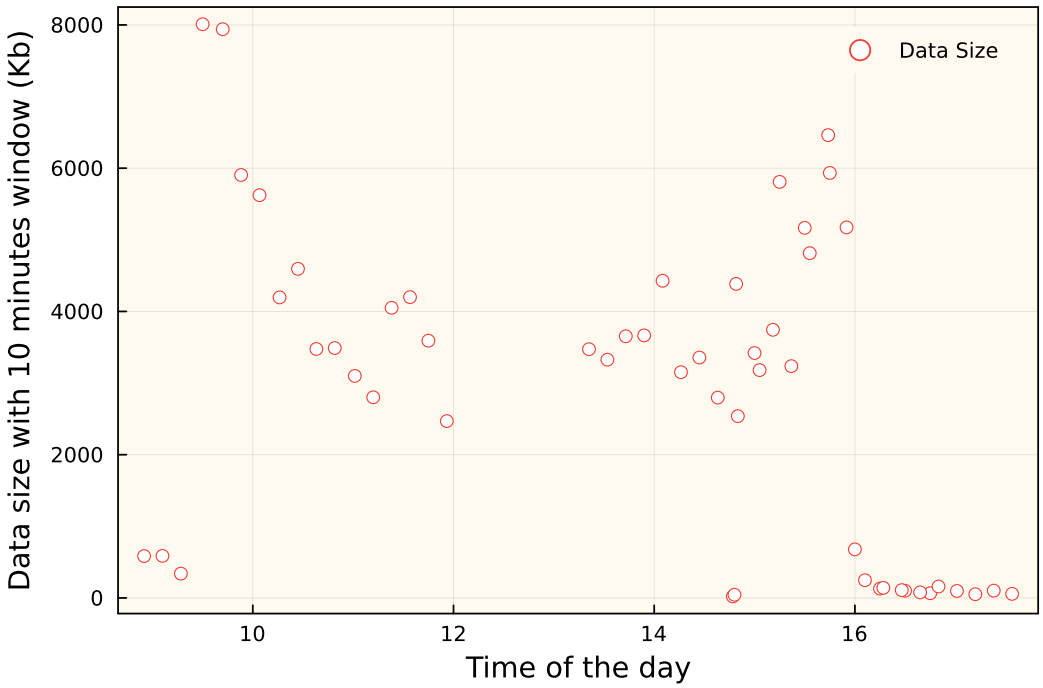

Figure 3.6: Estimated storage size from Polygon.io of trades message, ticker SPY on March 10, 2023

3.4.2 Latency of WebSocket API

We estimated the latency for data processing. As for each group of the experiment, a 30-second window was opened for data streaming (Figs. 3.7, 3.8, 3.9). The latency was assessed by the time required to process each received data point. Although there were occasional spikes in latency for one ticker reaching up to 125 milliseconds, the majority of the measurements were less than 4 milliseconds (on average, trades took 3.23 milliseconds, quotes took 0.29 milliseconds, and aggregates took 0.22 milliseconds). The spike in latency can be attributed to the fact that our program saves the status log at the beginning, which results in a delay in receiving subsequent messages.

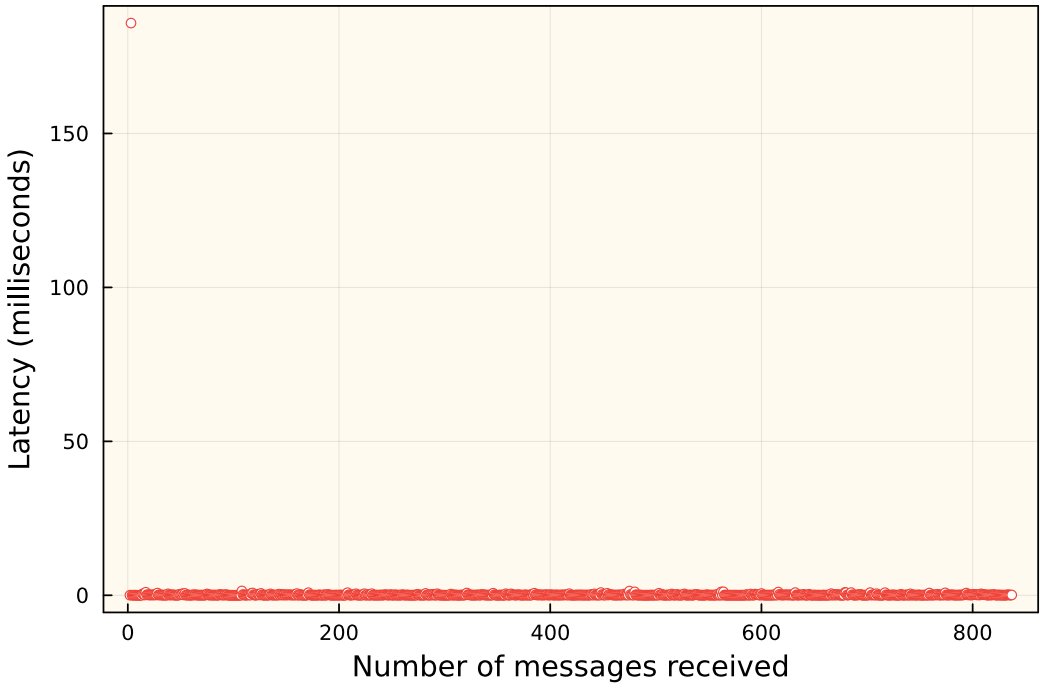

Figure 3.7: API latency for quotes messages in one ticker with a 30-second window

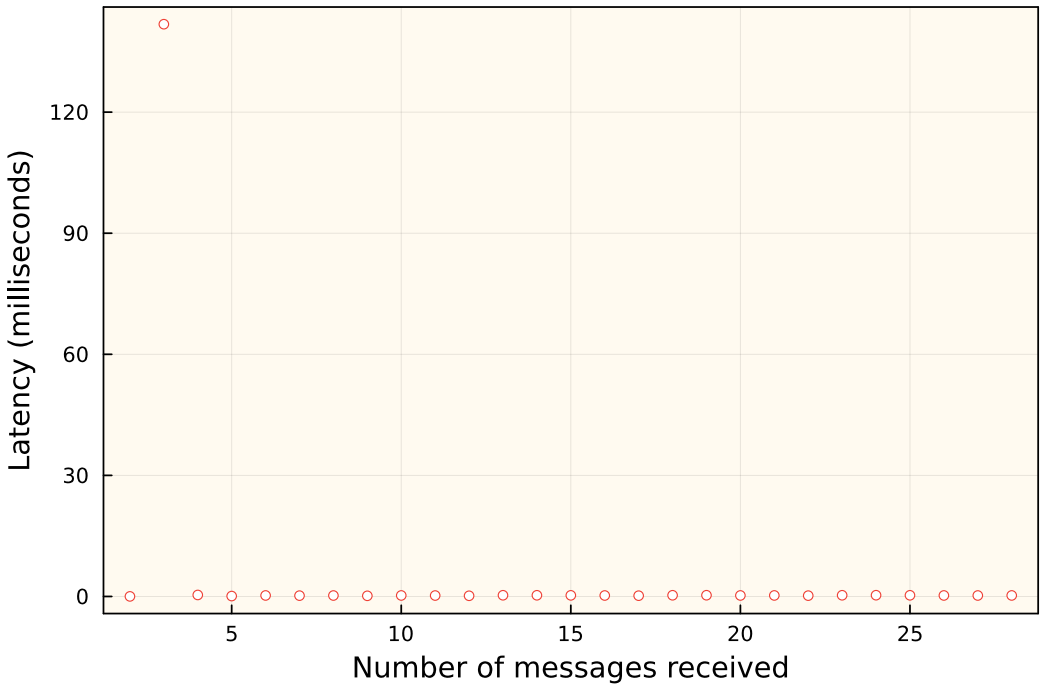

Figure 3.8: API latency for aggregates messages in one ticker with a 30-second window

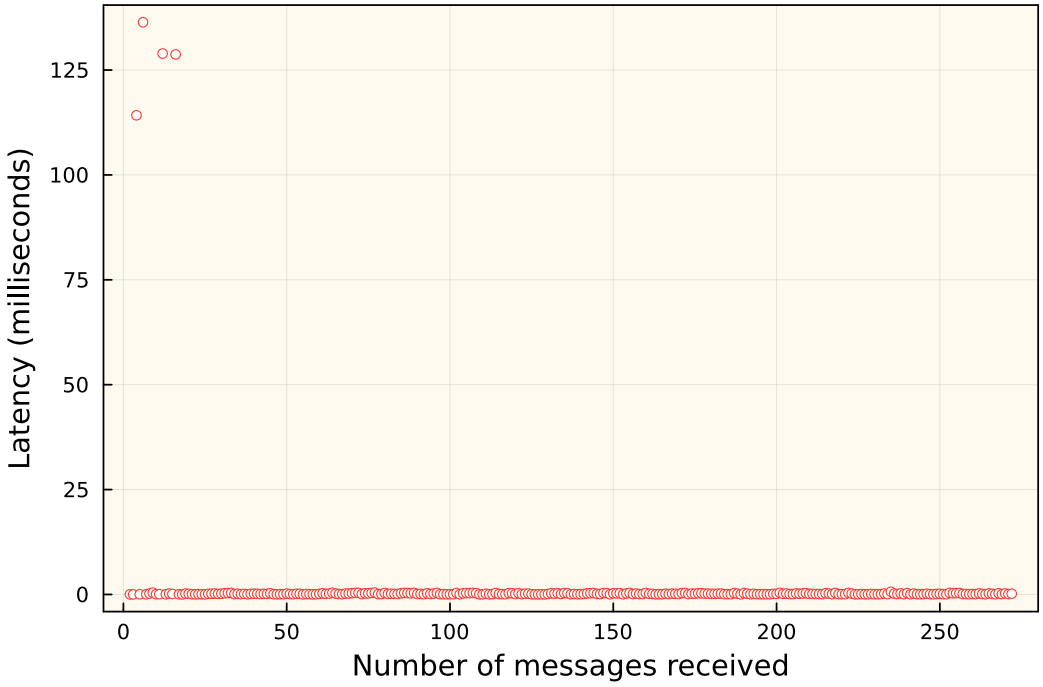

Figure 3.9: API latency for trades messages in one ticker with a 30-second window

3.5 Discussion

The WebSocket function starts an asynchronous task that pushes updates to the model collections when new data arrives. The model collections are built outside the function to ensure that users can access the streamed messages anytime. To use our WebSocket model, clients need to provide an API key. Polygon provides market data for Stocks, Options, Futures, and Cryptocurrencies. Each type of market data has its own WebSocket URL. Depending on what kind of market data they're interested in, clients need to specify the ticker and event type. For example, stock data provides aggregates per minute/second, trades, quotes, and luld. Each event type for specific market data corresponds to a response model, which is used to store real-time data for that event (Fig. 3.10).

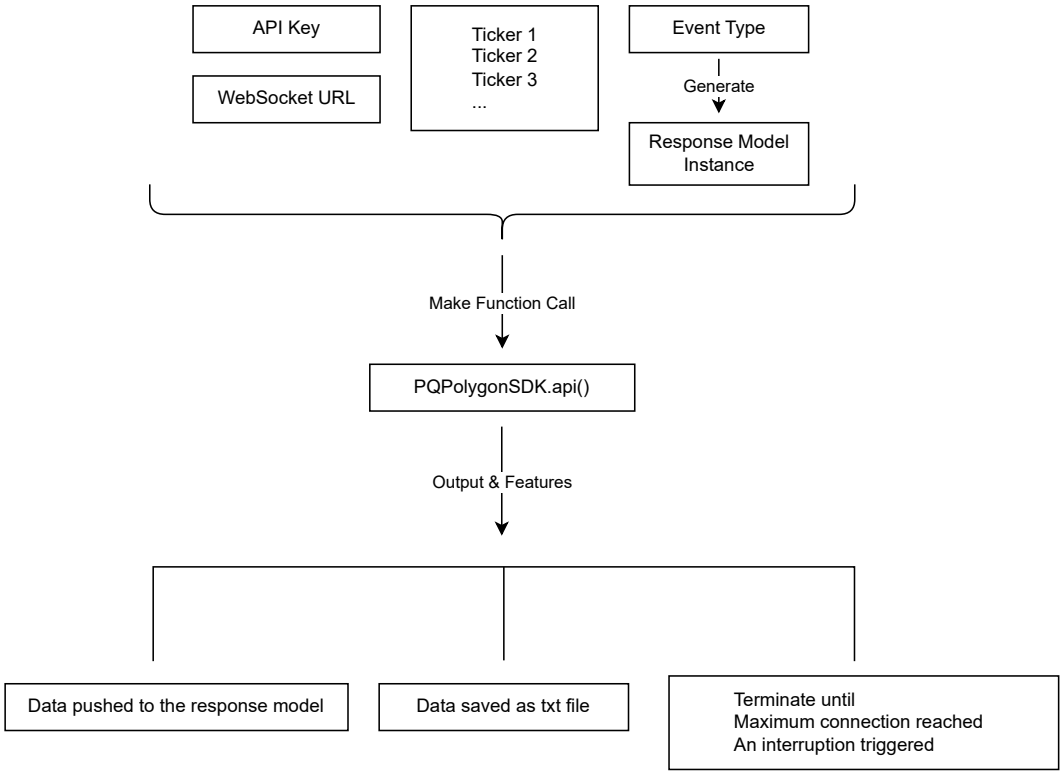

Figure 3.10: Diagram of WebSocket API

Clients also have the option to define the save limit and the number of ”last n” records. When the function is called, it streams the data to the response model and saves it to files. However, since memory is limited, only the ”last n” records will be kept in the response model. Moreover, the data will be stored in batches as text files once it exceeds a certain number of lines. Currently, the file system is used to store the data. However, in the future, we could use relational or non-relational databases. If there is a need to scale up the storage, we could also use cloud computing resources. The WebSocket function is easy to maintain. Currently, we only have fully developed all event types for stocks and one event type for cryptocurrencies. However, if clients prefer to stream other real-time data, they only need to create a corresponding response model.

CHAPTER 4

CONCLUSION AND FUTURE DIRECTIONS

The VLLimitOrderBook.jl package has been developed to simulate order book dynamics using AVL tree data structures. An analysis of our model was conducted in various aspects, including order book validation, construction, and filling of limit orders. Processing time was studied against different insertion positions and depths for the order book. Moreover, a concurrency and communication latency analysis was performed for the throughput of the model.

Results indicated that up to 40 million/min insertions can be achieved in a single process without significant differences between inserting at different price levels. Fewer price levels could be built for fast simulation. The inter-process simulation across different machines achieved a throughput of up to 2.2 million/min and inter-processes across different machines in the local area network achieved a throughput of up to 1.1 million/min. Our model demonstrates superior performance compared to the other two existing models, BSE and CoinTossX. Future research could involve implementing these alternative models in Julia and evaluating their performance.

Additionally, we have developed a WebSocket API to enable WebSocket connections and data streaming from Polygon.io for Julia developers. Our latency can achieve up to 4 milliseconds for processing real-time messages. While the data streaming has not been integrated into the order book system yet, this is a promising direction for future work.

For future improvements, a database system could be incorporated into both packages to allow for more efficient data storage and retrieval, Since the current

file system implementation has limited concurrency and efficiency. Additionally, cloud computing services could be utilized for storing and accessing data as we enlarge the system.

BIBLIOGRAPHY

[1] Auction market: Definition, how it works in trading, and examples. https://www.investopedia.com/terms/a/auctionmarket.asp.

[2] Dealer market: Definition, example, vs. broker or auction market. https://www.investopedia.com/terms/d/dealersmarket.asp.

[3] emi.nasdaq.com - /itch/nasdaq itch/. https://emi.nasdaq.com/ITCH /Nasdaq%20ITCH/.

[4] Exchange-traded fund (etf) explanation with pros and cons. https://www.investopedia.com/terms/e/etf.asp.

[5] Http - wikipedia. https://en.wikipedia.org/wiki/HTTP.

[6] London stock exchange: London stock exchange. https://www.londonstockexchange.com/discover/lseg/our-history.

[7] martinobdl/itch: Nasdaq order book reconstructor. https://github.com/martinobdl/ITCH.

[8] Mutual funds: Different types and how they are priced. https://www.investopedia.com/terms/m/mutualfund.asp.

[9] Nasdaq totalview-itch 5.0. https://www.nasdaqtrader.com/content/ technicalsupport/specifications/dataproducts/NQTVITCHSpecification .pdf.

[10] The nyse and nasdaq: How they work. https://www.investopedia.com/articles/basics/03/103103.asp.

[11] Order matching system - wikipedia. https://en.wikipedia.org/wiki /Order_matching_system.

[12] p-casgrain/limitorderbook.jl: A limit order book matching engine written in julia. https://github.com/p-casgrain/LimitOrderBook.jl.

[13] Paliquant/pqpolygonsdk.jl: Software development kit for polygon.io. https://github.com/Paliquant/PQPolygonSDK.jl.

[14] Renruize12306/dataeva. https://github.com/Renruize12306/dataEva.

[15] Renruize12306/vllimitorderbook.jl at ob-validation-dsp. https://github.com/Renruize12306/VLLimitOrderBook.jl/tree/ob-validation-dsp.

[16] Tesla (tsla) - stock split history. https://companiesmarketcap.com/tesla/stock-splits.

[17] Unit investment trust (uit): Definition and how to invest. https://www.investopedia.com/terms/u/uit.asp.

[18] Websocket - wikipedia. https://en.wikipedia.org/wiki/WebSocket.

[19] Welcome to antwerp. https://visit.antwerpen.be/en.

[20] What are financial securities? examples, types, regulation, and importance. https://www.investopedia.com/terms/s/security.asp.

[21] What is arbitrage? definition, meaning, example, and costs. https://www.investopedia.com/ask/answers/what-is-arbitrage/.

[22] What is nasdaq and how it operates? https://financetrain.com/what-is-nasdaq-and-how-it-operates.

[23] What is the national best bid and offer (nbbo)? how quote works. https://www.investopedia.com/terms/n/nbbo.asp.

[24] Mohiuddin Ahmed, Nazim Choudhury, and Shahadat Uddin. Anomaly detection on big data in financial markets. In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*, pages 998–1001, 2017.

[25] M. Caruso. *Crypto-assets global corporate finance transactions: A comparative and functional analysis of crypto offerings and securities laws*. Law, Business and Finance. INDEPENDENTLY PUBLISHED, 2019.

[26] Younes Chihab, Zineb Bousbaa, Marouane Chihab, Omar Bencharef, and Soumia Ziti. Algo-trading strategy for intraweek foreign exchange speculation based on random forest and probit regression. *Applied Computational Intelligence and Soft Computing*, 2019, 2019.

[27] Dave Cliff. BSE: A minimal simulation of a limit-order-book stock exchange. *CoRR*, abs/1809.06027, 2018.

[28] Rama Cont, Sasha Stoikov, and Rishi Talreja. A stochastic model for order book dynamics. *Operations research*, 58(3):549–563, 2010.

[29] Stewart Denholm, Hiroaki Inoue, Takashi Takenaka, Tobias Becker, and Wayne Luk. Network-level fpga acceleration of low latency market data feed arbitration. *IEICE TRANSACTIONS on Information and Systems*, 98(2):288–297, 2015.

[30] J Doyne Farmer 5, Laszlo Gillemot, Fabrizio Lillo, Szabolcs Mike, and Anindya Sen. What really causes large price changes? *Quantitative finance*, 4(4):383–397, 2004.

[31] Thierry Foucault, Ohad Kadan, and Eugene Kandel. Limit order book as a market for liquidity. *The review of financial studies*, 18(4):1171–1217, 2005.

[32] Koosha Golmohammadi and Osmar R Zaiane. Time series contextual anomaly detection for detecting market manipulation in stock market. In *2015 IEEE international conference on data science and advanced analytics (DSAA)*, pages 1–10. IEEE, 2015.

[33] Martin D. Gould, Mason A. Porter, Stacy Williams, Mark McDonald, Daniel J. Fenn, and Sam D. Howison. Limit order books, 2013.

[34] Gao-Feng Gu, Wei Chen, and Wei-Xing Zhou. Empirical regularities of order placement in the chinese stock market. *Physica A: Statistical Mechanics and its Applications*, 387(13):3173–3182, 2008.

[35] Joel Hasbrouck and Gideon Saar. Low-latency trading. *Journal of Financial Markets*, 16(4):646–679, 2013.

[36] Conghui He, Haohuan Fu, Wayne Luk, Weijia Li, and Guangen Yang. Exploring the potential of reconfigurable platforms for order book update. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2017.

[37] Burton Hollifield, Robert A Miller, and Patrik Sands. Empirical analysis of limit order markets. *The Review of Economic Studies*, 71(4):1027–1063, 2004.

[38] John Jenq and Priscilla Jenq. Order book data structures. In *Proceedings of*

the international conference on foundations of computer science (FCS), pages 49–55. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2018.

[39] Ivan Jericevich, Dharmesh Sing, and Tim Gebbie. Cointossx: An open-source low-latency high-throughput matching engine. *CoRR*, abs/2102.10925, 2021.

[40] Wenjie Lu, Jiazheng Li, Jingyang Wang, and Lele Qin. A cnn-bilstm-am method for stock price prediction. *Neural Computing and Applications*, 33:4741–4753, 2021.

[41] Hugh Luckock. A statistical model of a limit order market. *Sidney University preprint (September 2001)*, 2001.

[42] Reinhold C Mueller. *The Venetian money market: banks, panics, and the public debt, 1200-1500*. JHU Press, 2019.

[43] Amol M Pawar and Manisha S Mahindrakar. A comprehensive survey on online anomaly detection. *International Journal of Computer Applications*, 119(17), 2015.

[44] Ioanid Roşu. A dynamic model of the limit order book. *The Review of Financial Studies*, 22(11):4601–4641, 2009.

[45] Ioanid Roşu. A dynamic model of the limit order book. *The Review of Financial Studies*, 22(11):4601–4641, 2009.

[46] Cyril Schoreels, Brian Logan, and Jonathan M Garibaldi. Agent based genetic algorithm employing financial technical analysis for making trading decisions using historical equity market data. In *Proceedings. IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004.(IAT 2004).*, pages 421–424. IEEE, 2004.

[47] B Mark Smith. *A history of the global stock market: from ancient Rome to Silicon Valley*, pages 16–18. University of Chicago press, 2004.

[48] B Mark Smith. *A history of the global stock market: from ancient Rome to Silicon Valley*, pages 35–37, 39–43, 45. University of Chicago press, 2004.