SoftwareX 19 (2022) 101136

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

CoinTossX: An open-source low-latency high-throughput matching engine

Ivan Jericevich [a,*], Dharmesh Sing [b], Tim Gebbie [a,b]

[a] Department of Statistical Sciences, University of Cape Town, Rondebosch 7700, South Africa
[b] Department of Computer Science and Applied Mathematics, University of the Witwatersrand, Johannesburg 2000, South Africa

ARTICLE INFO

Article history:
Received 24 February 2021
Received in revised form 21 May 2022
Accepted 24 June 2022

Keywords:
Market matching-engine
Open-source application
Microsoft Azure deployment
Java web application
Julia
Python
Agent-based modeling
JSE

ABSTRACT

We deploy and demonstrate the CoinTossX low-latency, high-throughput, open-source matching engine through small-scale desktop and large-scale locally-hosted testing with multiple traded instruments and clients managed concurrently by sending orders using the Julia and Python languages. We demonstrate a cloud-based deployment using Microsoft Azure, with large-scale industrial and simulation research use cases in mind. The system is developed in Java with orders submitted as binary encodings (SBE) via UDP protocols using the Aeron Media Driver as the low-latency, high-throughput message transport. The system separates the order-generation and simulation environments *e.g.* agent-based model simulation, from the matching of orders, data-feeds and various modularized components of the order-book system. This ensures a more natural and realistic asynchronicity between events generating orders, and the events associated with order-book dynamics and market data-feeds.

Code metadata

Current code version	v1.0.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00044
Code Ocean compute capsule	
Legal Code License	MIT License
Code versioning system used	None
Software code languages, tools, and services used	Programming languages: Java, CSS, HTML, Shell, Julia, Python. Tools: Aeron Media Driver, MapDB, HdrHistogram, Apache Wicket, Spring Boot, Java Microbenchmark Harness
Compilation requirements, operating environments & dependencies	Operating environments include Linux, Windows and OSX.
If available Link to developer documentation/manual	https://github.com/dharmeshsing/CoinTossX/blob/master/README.md
Support email for questions	ivan.jericevich@alumni.uct.ac.za

Software metadata

Current software version	1.1.0
Permanent link to executables of this version	https://github.com/dharmeshsing/CoinTossX/archive/refs/tags/v1.1.0.zip
Legal Software License	MIT
Computing platforms/Operating Systems	Linux, OS X, Microsoft Windows
Installation requirements & dependencies	JDK 8
If available, link to user manual - if formally published include a reference to the publication in the reference list	https://github.com/dharmeshsing/CoinTossX/blob/master/README.md
Support email for questions	ivanjericevich96@gmail.com and dharmeshsing@gmail.com

* Corresponding author.
E-mail addresses: ivan.jericevich@alumni.uct.ac.za (Ivan Jericevich), dharmeshsing@gmail.com (Dharmesh Sing), tim.gebbie@uct.ac.za (Tim Gebbie).

https://doi.org/10.1016/j.softx.2022.101136

1. Motivation and significance

A complete study of the market microstructure of the Johannesburg Stock Exchange (JSE) is not possible without access to their matching engine. Studying market microstructure is challenging due to the continual evolution of the market, regulations and technology. Much of the current literature focuses on data generated by existing exchanges and the modeling thereof. The importance of order matching engines in the trading infrastructure makes these systems of interest, not only to computer scientists, but also to computational finance researchers building models, because event-driven processes relating to order matching may provide a calibration boundary for various market models [1–3].

A trade matching engine is the central software and hardware component of an electronic exchange that matches up bids and offers to complete trades. To achieve competitive trading speeds, market clients and other participants need proximity to the matching engine, a system that is resilient to network and power failures, as well as appropriate hardware scaled to accommodate worst-case network traffic scenarios on the busiest trading days [4]. Cloud-based software solutions benefit by being resilient and offering cost-effective, easy scaling. However, the biggest challenge for latency in a cloud-based environment is proximity: that the hardware is not co-located within a data center [4]. That said, a combination of the low latency of traditional matching engines and the resiliency, scalability and availability of cloud-based environments is yet to come about.

The evolution of software and hardware and the increasing need to reduce transaction costs, system failures and transaction errors has led to important changes in market structure and market architecture. These changes have supported the rise of electronic and algorithmic trading. The specifics are unique to each and every market and regulatory environment. CoinTossX is fully configurable. Although implemented and tested here using the publicly available JSE market rules and test-cases, it can be configured for a rich variety of market structures.

The LMAX Exchange [5] is an FX exchange with an ultra low-latency matching engine. Thompson et al. [5] developed the Disruptor ring buffer on the JVM for inter-thread concurrency as an alternative to storing events in queues or linked lists in response to the problem that linked-lists could grow and increase the garbage in the system—causing significant costs to latency and jitter.[1] The Disruptor preallocates memory in a cache-friendly manner which is shared with the consumer allowing the system to process up to 25,000,000 messages per second on a single thread with latencies lower than 50 ns [5].

Addison et al. [4] implement a simple FX trading system and deploy it to cloud environments from multiple cloud providers, recording network latency and overall system latency in order to assess the capability of public cloud infrastructure in performing low-latency trade execution under various configurations and scenarios. They demonstrated sub-500 microsecond roundtrip latencies—concluding that it is feasible to build a production low-latency trading system to support security trading in the cloud [4].

CoinTossX is a simulation environment; so the task of producing realistic simulated market dynamics is left to the user(s) [6]. However, the additional class layer of causation that this introduces in the modeling framework, by having the complex set rules of the environment, the system, and the architecture be separated or independent from the modeling and decision making processes, can place more emphasis on the impact of top-down actions and various environmental constraints on outcomes—this is a potential step towards hierarchical causality [7].

[1] Jitter is the deviation from true periodicity of a presumably periodic signal caused by network congestion, often in relation to a reference clock signal.

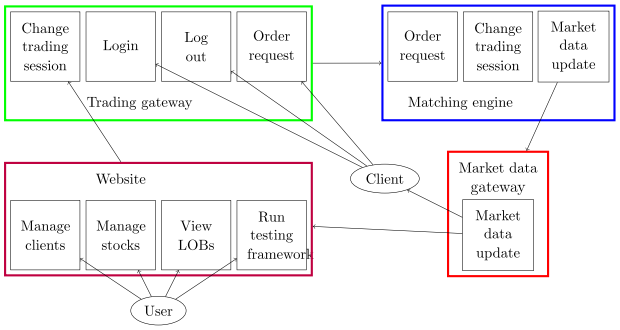


Fig. 1. A High level diagram of the relationship between the components of CoinTossX in terms of end-user functionality [11].

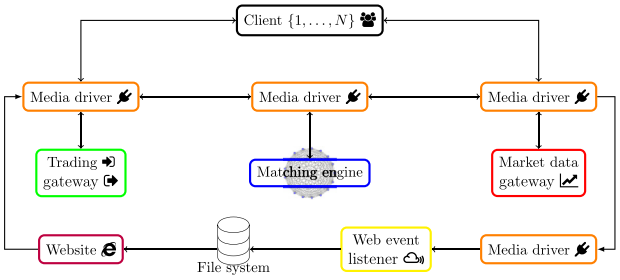


Fig. 2. A High level architecture diagram visualizing the technical relationship between the software components [11].

2. Software description

The system requirements we implemented were obtained from JSE's publicly available technical documentation and test cases [8–10].[2] The eight main components of the simulator which are discussed in the sections that follow are the stocks, clients, trading gateway, matching engine, market-data gateway, website, web-event listener, and database [11] (see Figs. 1 and 2).

2.1. Software architecture

This software is open source and built to run on different operating systems as well as on a single server or on multiple servers. The goal was to build a matching engine that could achieve low-latencies using industry standard technology. Implementation in Java means that CoinTossX can be deployed on different hardware configurations.

A single market event has the following process: A SBE message[3] will be sent to the trading gateway, and be forwarded to the matching engine, which will then process it. It will send a status update back to the trading gateway which will forward the message back to the client. The matching engine will send an update to the market-data gateway if there is a change in the limit order book. The market-data gateway will forward the updates to all connected clients as well as the web-event listener. The web-event listener saves each event to the database. The website then reads and displays the data from the database.

The communication between the website and the file system is done by transmitting data over a network using only User

[2] Technical documentation can be found at https://www.jse.co.za/services/technologies/equity-market-trading-and-information-technology-change.

[3] JSE uses text messages between the clients and their matching engine which is inefficient since characters take up more memory and are slower to transmit across the network [11], the SBE message protocol was found to be the fastest way to encode and decode messages with the greatest throughput.

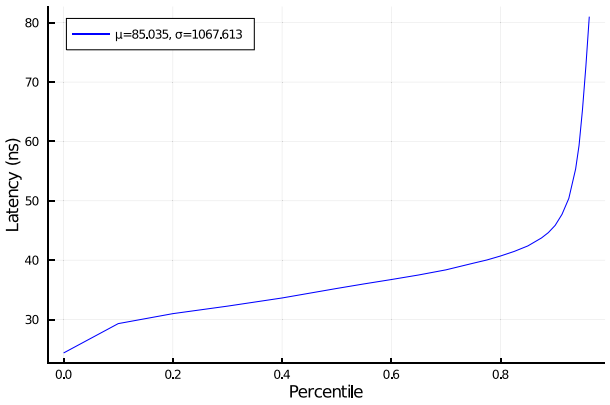


Fig. 3. A High Dynamic Range (HDR) Histogram graph showing the latency percentiles of Aeron's PingPong test. This provides a lower-bound for the achievable latency of CoinTossX as well as for open-source message transport libraries in general [12].

Datagram Protocol (UDP)[4] as opposed to Transmission Control Protocol (TCP)—a heavyweight but reliable protocol. UDP was chosen for being lightweight (although potentially unreliable as it does not check the connection or the order of the messages).

Modularization is achieved by designing each of the above components independently of one other while allowing communication between them to occur only via exposed ports. This functionality was introduced by assigning each component an IP address and port on which to listen. Furthermore, each client/user may submit orders from a remote server as long as the server allows for these types of inbound/outbound port communications.

After extensive comparisons of message transport software, the Aeron media driver library was chosen because it is an efficient and reliable UDP unicast, UDP multicast, and IPC message transport. Each component has its own media driver to shovel events to and from the component. Aeron is designed to be the highest throughput with the lowest latency possible of any messaging system [12] (Fig. 3).

This modular design allows each component and client to be started and run on separate independent servers. It is then possible to have an industrial matching engine by providing each component with its own high-performance server.

The software was designed so that different matching logic algorithms are in separate Java classes—this allows the logic to be changed to test variations of matching logic. The matching engine adopts the price–time priority algorithm during the continuous trading session. Limit and market orders submitted during a call auction are not matched immediately. These orders are matched at the end of the call auction at a single price using a price discovery process—The Volume-Maximizing Auction Algorithm. More specifically, the filter and uncross algorithms run each time the Best Bid or Offer (BBO) changes or every 30 s. The filter algorithm uses the heuristic Hill Climber search/optimization algorithm[5] to find the optimal volume of hidden-limit-orders that can be executed. The search will filter out hidden limit orders with MES constraints that are not eligible. After the filtering, specific rules are used to select the orders and price to executed in the crossed region.

Since storing the active orders on disk would increase the I/O and reduce performance the matching engine stores the active orders in memory. The data structure for storing the LOB is designed to have a low memory overhead (lists, hash tables, trees, B-tree, B+tree) and be efficient in searching, updating and deleting orders [11]. The fast transfer of data from the CPU to main memory is achieved through the efficient use of the CPU cache. The simulator only uses main memory without the use of virtual memory.

The website was developed using Spring Boot and Apache Wicket having only read-permission from the file system. The original design had the event listener and website in the same Java process. When the website was used or paused because of garbage collection, it affected the receiving and saving of events. Therefore this logic was split into a separate component—the web-event listener.

The listener could keep the received events in memory, which would be fast but require an unknown maximum memory setting, or saved to the database. Therefore the data is saved to the database. Using the MapDB library, an off-heap hashmap is used to save the events to memory mapped files. This significantly improves performance. The entire file or a part of the file can be loaded into memory. The values in memory will still be written to the file system even if the JVM crashes. The web-event listener saves events to the file system but receives events faster than it can save it to file. This issue is solved with the Disruptor by having two threads: one to receive and store events, and the other to save events to the database [5].

2.2. Clients

Each client is assigned input and output URLs for both the Native Gateway and Market-Data Gateway specifying the IP addresses (here localhost) to/from which messages will be sent/received as well as the ports on which these components are listening. Therefore each client takes up a number of threads on the machine it runs on.

The number of client–stock pairs that can be supported in total is only limited to the performance capabilities of the server on which the matching engine is running (Section 3). The limits of the number of clients that can be supported by each stock still needs to be measured with respect to different hardware configurations.

2.3. Software functionalities

Once started CoinTossX can be monitored using the website; four screens are available: First, the stock screen showing configured stocks, the active trading sessions, and a button to view the LOBs. Second, the LOB snapshot page to visualizes each stock as a bar chart, here tables of active orders can be exported. Third, the Hawkes process configuration page to changing process parameters (Section 3). Third, the simulation page allowing users to warm-up, start, stop and view the status of trading sessions of the simulation/application. Finally, the client page, showing configured clients and allowing the user to create, update and delete clients (unlisted clients cannot login).

For a client to submit orders a login request, the relevant username and password must be sent to the trading gateway. Similarly, log-out request must be sent when the client leaves. Thereafter the client can submit orders by publishing order messages to the trading gateway via UDP ports. Traders cannot submit orders that are smaller than the LOB tick size. The allowable order types that are not conditioned on time are: market-orders (MO), limit-orders (LO), hidden-orders (HO), stop-orders (SO) and stop-limit-orders (SL) [8].

Some orders can have a time-in-force (TIF) that cannot be amended after the order is placed. Time-in-force options include

[4] JSE uses TCP for their trading gateway and UDP for their market data gateway.

[5] The Hill Climber algorithm is an optimization technique that iteratively searches for the solution to a problem by changing one element in each iteration.

Table 1
Trading sessions currently available.

Session	Trigger
Start of trading	07:00–08:30
Opening auction call	08:30–09:00
Continuous trading	09:00–16:50
Volatility auction call	triggered
Intraday auction call	12:00–12:15
Closing auction call	16:50–17:00
Closing price publication	17:00–17:05
Closing price cross	17:05–17:10
Post close	17:05–18:15
Halt	manually evoked
Halt and close	manually evoked
Pause	manually evoked
Re-opening auction call	manually evoked
Trade reporting	08:00–18:15

(See the supporting documentation [8]): OPG, GFA, GFX, ATC, DAY, IOC, FOK, GTC, GTD, GTT, and CPX. Combinations of time-in-force and order-types are accepted except for the rejection combinations of "hidden orders" (HO), "stop orders" (SO) and "stop limit orders" (LS) with "Good for intraday auction" (GFX), "At the opening" (OPG) and "At the close" (ATC) [8].

Lastly, CoinTossX also allows for multiple trading session types, currently these correspond to those of the JSE [8] and has the sessions described in Table 1.

These trading sessions and the rules adopted by each are configurable via the cron expressions in the `data/tradingSessionsCron.properties` file. The times shown are those of the JSE, provided for context [8] (see Table 2).

3. Testing framework

CoinTossX has been successfully deployed locally as well as to remote servers such as Microsoft Azure, CHPC and TW Kambule Mathematical Sciences Laboratories provided servers (using re-purposed TACC Ranger blades Table 3). Deployment to high-performance compute solutions with a strong dependency on job manager port and worker control *e.g.* using a master–slave approach (such as UCT HPC) will make deployment infeasible.

3.1. Unit testing

Unit tests, based on the test cases made available online from the JSE (with requirements taken from [8]), were implemented to cover the testing of the functional requirements while individual throughput and latency performance tests were implemented in conjunction with the Java Microbenchmark Harness[6] (JMH) to test methods whose performance were critical. Testing of the trading sessions were restricted to the continuous and intraday auction trading sessions using only the DAY TIF.

The outputs of these tests were not compared to the JSE or another exchange as the data is not made available by the industry. The JSE's test environment is also closed and does not provide realistic order-book dynamics. This paper focuses mainly on the results of latency and throughput tests, however, the types/results of the extensive list of tests performed can be found by referring to [11].

[6] JMH is a Java harness for building, running, and analyzing nano/micro/milli/macro benchmarks written in Java and other languages targeting the JVM.

3.2. Order-flow testing

Matching engine integrity was evaluated using unit tests and, instead of an agent-based approach, simulations and tests aim to understand throughput and latency, and were carried out with a 8-variate marked Hawkes process [13,14]. The software and Hawkes client processes were deployed on one server which affected the performance of all components. The mutually exciting processes correspond to 8 different order types in the testing framework, considering only basic aggressive and passive market and limit orders as in [15]. The simulation is conducted using the intensity-based Thinning Algorithm as introduced by [16] and modified by [17] and is used to define the time at which orders arrive.

In the testing framework a single client is associated with each stock and is meant to represent a large group of traders investing in that stock, governed by the Thinning Algorithm submitting large numbers of orders. The prices and volumes are generated based on the order type in a random manner with bids and asks unrealistically crossing each other. Nonetheless this is just to demonstrate the applications ability.

Test scenarios were created to test the performance of the software by evaluating the impact of multiple clients–stock pairs on different operating systems and hardware specifications. Here the performance results are documented by running the application on the two machines listed in Table 3.

Although not presented here, limit order book storage testing was also conducted on the WITS MSS servers and demonstrated the ability to store thousands of orders at each price point.

3.3. Latency tests

The latency was tested and visualized using the HdrHistogram library.[7] With every run, each client submits approximately 110,000 orders. The time it takes to process all these orders is then measured and compared between machines. To reproduce Fig. 4 and Table 4 simply re-run the Hawkes simulations for the system to write latency and throughput results to file.

Fig. 4 demonstrates that the latency, in nanoseconds, increases as the number of clients–stock pairs increase with each run—since each client is associated with multiple threads running processes in parallel. Due to there being only 4 CPUs on the Azure VM, running more than 6 clients simultaneously was found to be infeasible. For the high specification WITS MSS server the minimum and maximum latency (measured up to 10 clients) at the 90th percentile is 106 ns and 248 ns. For the medium spec machine (with a maximum of 6 clients) the minimum and maximum latency at the 90th percentile is 123 ns and 393 ns. On a high end machine one can expect sub 250 ns latency (with 10 clients); with a medium spec machine one can expect sub 400 ns latency (with 6 clients).[8]

Fig. 5 considers the scenario where high volumes of orders (1,000,000) are submitted to the trading gateway from a single client and compared across the two machines. For the WITS MSS Server, the latency is 735 ns at the 90th percentile but maintains a significantly lower latency on average. On the other hand, the Azure VM Server has higher latency's on average with a latency of 964 ns at the 90th percentile.

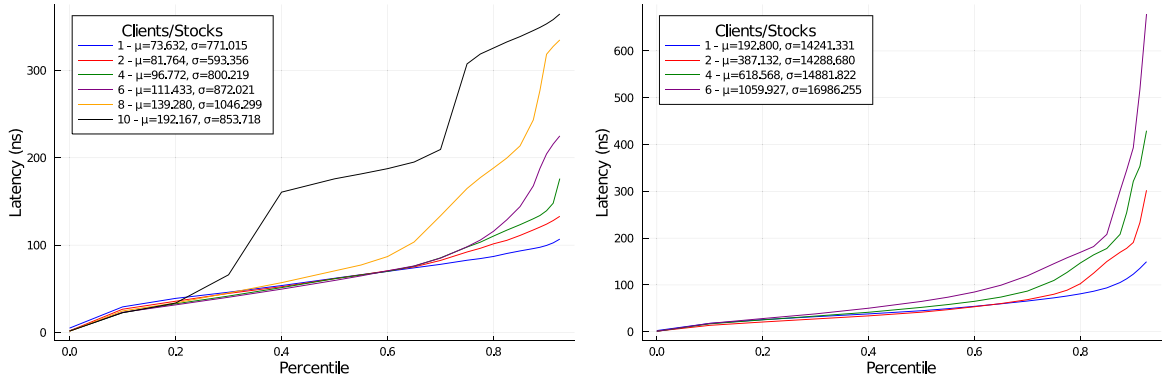
[7] HdrHistogram is designed for recording histograms of value measurements in latency and performance sensitive applications. Measurements show value recording times as low as 3–6 nanoseconds on modern (circa 2012) Intel CPUs.

[8] JSE's average round-trip co-location network latency is sub 100 microseconds and its matching engine has a latency of 50 microseconds [18].

Table 2
Valid trading session and order type/TIF combinations for the submission of new orders to the trading gateway.

Session	Order type/time-in-force											Order type/time-in-force			
	OPG	ATC	IOC	FOK	GTC	GTD	GTT	GFA	GFX	DAY	CPX	MO	LO	SO & SL	HL
Start of trading															
Opening auction call															
Continuous trading															
Volatility auction call															
Intraday auction call															
Closing auction call															
Closing price publication															
Closing price cross session															
Post close															
Halt															
Halt and close															
Pause															
Re-opening auction call															
FCO auction call session															

Accepted | Rejected | Accepted and parked until injected | Accepted and expired immediately if they do not execute upon aggression | Carried forward from the previous day



(a) WITS MSS Server (high specification) (b) Azure VM Server (medium specification)

Fig. 4. A High Dynamic Range (HDR) Histogram graph showing the latency percentiles of the matching engine for increasing numbers of stocks and clients. In the testing framework each client submits approximately 110,000 market/limit orders.

Table 3
Hardware specifications of machines used for throughput and latency testing used to compare the cloud solution (Azure) to the dedicated physical hardware solution (WITS MSS Server blade).

Machine	Operating System	Memory	Processors
WITS MSS Server	64-bit Linux Ubuntu 16.04-LTS	32GB	4× AMD Opteron 8356 @ 2.3 GHz (16 cores)
Standard A4 m V2 Azure VM	64-bit Linux Ubuntu 18.04-LTS	32GB	4× Intel Xeon E5-2673 v3 @ 2.4 GHz (32 cores)

3.4. Throughput tests

The most significant decrease in throughput is when more than 4 client–stock pairs are used. Beyond 6 clients the hardware configuration of the Azure VM was found to be insufficient. In total, the high spec machine was shown to be capable of processing more than 1,000,000 orders in less than a 19 min simulation period with significantly low latencies. On the other hand, the medium spec machine was capable of processing 450,000 and 560,000 orders in approximately 44 min and 2 h simulation periods for 4 and 6 client–stock pairs, respectively (see Fig. 6).

4. Conclusions

CoinTossX is a low-latency, high-throughput, configurable stock exchange allowing users to view the limit order book in real time as well as for multiple clients to connect and send orders to the exchange. It supports multiple stocks and a variety of different trading session. Here Hawkes processes are used to provide a simple but robust simulation of order arrivals for infrastructure testing. The CoinTossX website can be enhanced to analyze the data that is processed. The exchange provides a realistic platform for agent based models exploration.

Table 4
Tabulated measurements of throughput per second with increasing numbers of client–stock pairs. The Hawkes process simulated client submits approximately 110,000 orders. After each simulation the start and end times are published and used to calculate the throughput per second. The hardware configuration on the Azure VM was found to be insufficient using more that 6 clients. All data generated in this table can be acquired at the end of each simulation in the data directory of the start-up folder.

	WITS MSS Server			Microsoft Azure VM		
client–stock pairs	Duration	Number of orders	Throughput per second	Duration	Number of orders	Throughput per second
1	00:00:48.820	111646	2287	00:06:44.917	110825	274
2	00:04:23.981	224562	850	00:19:38.576	222390	189
4	00:12:02.204	448774	621	00:43:55.734	447878	170
6	00:15:12.386	669331	733	02:01:27.494	564070	77
8	00:20:27.010	895080	729	–	–	–
10	00:18:52.289	1120514	989	–	–	–

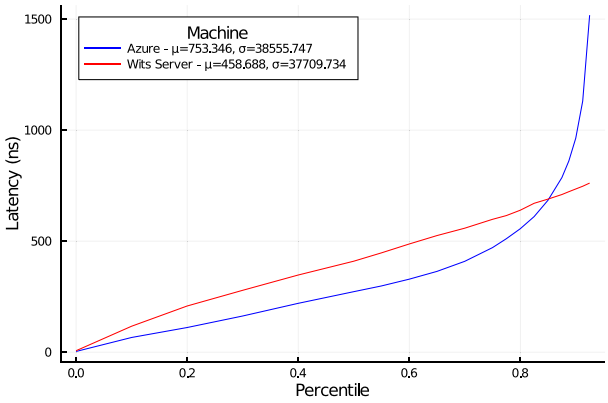


Fig. 5. A comparison of latencies during a high volume scenario. 1,000,000 orders were submitted by a single client on the WITS MSS Server and Azure VM Server.

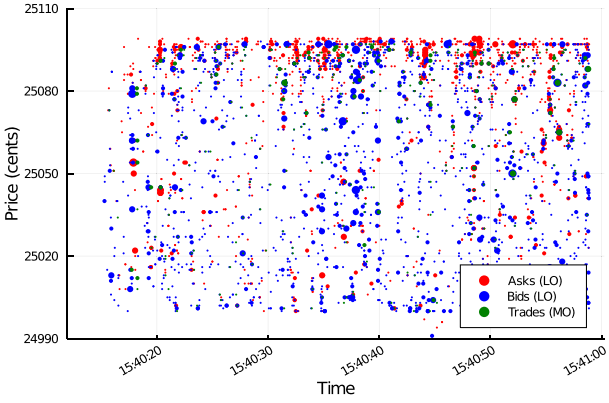


Fig. 6. A Graph of the limit-orders and market-orders submitted to the trading gateway. As a proof of concept, the Hawkes simulation assigns volumes and prices in a unrealistic manner as the bid and ask are allowed to cross (negative spread). The size of the points are proportional to the volume of the order.

The limit-order database components (which include the web-event listener, file systems, and website, and use the LOB snapshot functionality) are independent from the order matching components (the matching engine) and hence can be switched-off for use-cases where the limit order-book (LOB) is maintained and monitored independently, perhaps only using data captured from the market-data gateway feed in the equivalent of an Order Management System (OMS) and where orders are submitted to the trading gateway from an external equivalent of an Execution Management System (EMS).

Additional work can be done to change the matching rules for testing the impact of rules and regulation changes on the limit order book dynamics. Since the components are de-coupled, their implementation language can be continually and easily changed to support the latest frameworks [19]. Future work will be aimed at better understanding the interaction dynamics from increasingly realistic approaches to point-process based simulations *e.g.* with trading clients based on Hawkes processes using models such as those of [20,21] to investigate the interactions of limit-order and market-order trading agents through a realistic order matching process.

5. Impact and ongoing research applications

Aside from the research use-case, fast, robust and realistic open-source matching engine software can be useful for commercial applications *e.g.* by algorithmic trading firms where trading strategy testing outside of the production environment of real markets is either based on low resolution historical data, or may not even be possible. Simulating the entire financial market ecosystem for trade strategy testing and risk management is appealing because of the mechanistic complexity of the market structure and costs and the nonlinear feed-backs and interactions that multiple interacting agents bring to the market ecosystem. This type of simulation can be both financially costly, as well as being computationally expensive; yet, it appears tractable, and subsets of the ecosystem are used for system verification *e.g.* in vendor provided test market venues.

A trading development environment incorporating a simulated matching engine can result in the rapid evolution of trading software and hardware as well as the reduction of transaction costs, system failures and transaction errors. The key problem is that many researchers developing trading and risk mitigation strategies do not have access to vendor provided testing environments, while those that do often do not want to expose their strategies to competitors in a shared testing environment. Sufficiently realistic multi-agent simulation environments remain illusive, particularly for low liquidity and collective behavior based risk-event scenarios. This is because a key component remains the realism of the underlying trading agents and their interactions within test markets.

This brings to the fore two separate areas of research that we think can be practically useful as well as interesting. The first is attempts to replicate known market phenomenology that may be the result of the discrete nature of the order matching process *e.g.* the Epps effect [1]—this can be tackled from at least two distinct perspectives: (1.) point-process models,[9] and (2.) agent-based models.[10] The second is for the realistic simulation and modeling of high-frequency trading and financial

[9] *Simulation and estimation of a point-process market-model with a matching engine*. arXiv:2105.02211 [q-fin.TR].
Reproducible from: https://github.com/IvanJericevich/IJPCTG-HawkesCoinTossX

[10] *Simulation and estimation of an agent-based market-model with a matching engine*. arXiv:2108.07806 [q-fin.TR].
Reproducible from: https://github.com/IvanJericevich/IJPCTG-ABMCoinTossX

market limit order books through the application of event-based and reactive agent-based models. This may allow researchers, strategists and risk managers to more faithfully capture the empirical implications of multi-agent adaption in reactive software systems.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank Turgay Celik and Brian Maistry WITS Mathematical Science Support and the TW Kambule Mathematical Sciences Laboratories for allowing us to use 4 former TACC Ranger blades for some of the simulation work. We thank Patrick Chang, Dieter Hendricks and Diane Wilcox for various discussions and advice with regards to the project. We acknowledge the primary research funder: University of Cape Town, Fund UCT STA549 URC 459282.

References

[1] Chang P, Pienaar E, Gebbie T. The Epps effect under alternative sampling schemes. Physica A 2021;583:126329. http://dx.doi.org/10.1016/j.physa.2021.126329, URL https://www.sciencedirect.com/science/article/pii/S0378437121006026.

[2] Platt D, Gebbie T. Can agent-based models probe market microstructure? Physica A 2018;503:1092–106. http://dx.doi.org/10.1016/j.physa.2018.08.055, URL http://www.sciencedirect.com/science/article/pii/S0378437118309956.

[3] Goosen K, Gebbie T. Calibrating high-frequency trading data to agent-based models using approximate bayesian computation. University of cape town; 2020, http://dx.doi.org/10.25375/uct.12894005.v1, URL https://github.com/KellyGoosen1/hft-abm-smc-abc.

[4] Addison A, Andrews C, Azad N, Bardsley D, Bauman J, Diaz J, et al. Low-latency trading in the cloud environment. In: 2019 IEEE international conference on computational science and engineering (cse) and ieee international conference on embedded and ubiquitous computing. IEEE; 2019, p. 272–82. http://dx.doi.org/10.1109/CSE/EUC.2019.00060, URL https://ieeexplore.ieee.org/document/8919600.

[5] Thompson M, Farley D, Barker M, Gee P, Stewart A. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. Technical report, LMAX; 2011, URL https://lmax-exchange.github.io/disruptor/.

[6] Lussange J, Belianin A, Bourgeois-Gironde S, Gutkin B. A bright future for financial agent-based models. SSRN; 2018, http://dx.doi.org/10.2139/ssrn.3109904, URL https://ssrn.com/abstract=3109904.

[7] Wilcox D, Gebbie T. Hierarchical causality in financial economics. SSRN; 2014, http://dx.doi.org/10.2139/ssrn.2544327, URL https://ssrn.com/abstract=2544327.

[8] Johannesburg Stock Exchange. Equity market trading and information solution JSE specification document volume 00E — Trading and information overview. 4rd ed. 2020, URL https://www.jse.co.za/content/JSEContractSpecificationItems/Volume%2000E%20-%20Trading%20and%20Information%20Overview%20for%20Equity%20Market.pdf.

[9] Johannesburg Stock Exchange. Equity market trading and information solution JSE specification document volume 01 — Native trading gateway. 3.05rd ed. 2018, URL https://www.jse.co.za/content/JSETechnologyDocumentItems/Volume%2001%20-%20Native%20Trading%20Gateway.pdf.

[10] Johannesburg Stock Exchange. Equity market trading and information solution JSE specification document volume 05 — Market data gateway. 3.09rd ed. 2019, URL https://www.jse.co.za/content/JSETechnologyDocumentItems/Volume%2005%20-%20Market%20Data%20Gateway%20(MITCH%20-%20UDP).pdf.

[11] Sing D, Gebbie T. JSE matching engine simulator. University of the Witwatersrand; 2017, URL http://wiredspace.wits.ac.za/handle/10539/25136.

[12] Real Logic. Aeron media driver. Real Logic: GitHub; 2021, URL https://github.com/real-logic/Aeron/wiki.

[13] Hawkes AG. Spectra of some self-exciting and mutually exciting point processes. Biometrika 1971;58(1):83–90. http://dx.doi.org/10.2307/2334319, URL https://www.jstor.org/stable/2334319.

[14] Ogata Y. Statistical models for earthquake occurrences and residual analysis for point processes. J Amer Statist Assoc 1988;83(401):9–27. http://dx.doi.org/10.1080/01621459.1988.10478560, URL https://www.tandfonline.com/doi/abs/10.1080/01621459.1988.10478560.

[15] Large J. Measuring the resiliency of an electronic limit order book. J Financial Mark 2007;10(1):1–25. http://dx.doi.org/10.1016/j.finmar.2006.09.001, URL http://www.sciencedirect.com/science/article/pii/S1386418106000528.

[16] Lewis PW, Shedler GS. Simulation of nonhomogeneous Poisson processes by thinning. Nav Res Logist Q 1979;26(3):403–13. http://dx.doi.org/10.1002/nav.3800260304, URL https://onlinelibrary.wiley.com/doi/10.1002/nav.3800260304.

[17] Ogata Y. On Lewis' simulation method for point processes. IEEE Trans Inform Theory 1981;27(1):23–31. http://dx.doi.org/10.1109/TIT.1981.1056305, URL https://ieeexplore.ieee.org/document/1056305.

[18] Johannesburg Stock Exchange. The lowest-latency connection to JSE markets. 2015, URL https://www.jse.co.za/content/JSETechnologyDocumentItems/3.%20JSE%20Colocation%20Brochure%202015.pdf.

[19] Sing D. CoinTossX software. GitHub; 2017, http://dx.doi.org/10.25375/uct.14069552, URL https://github.com/dharmeshsing/CoinTossX.

[20] Bacry E, Muzy J-F. Hawkes model for price and trades high-frequency dynamics. Quant Finance 2014;14(7):1147–66. http://dx.doi.org/10.1080/14697688.2014.897000.

[21] Zheng B, Roueff F, Abergel F. Modelling bid and ask prices using constrained hawkes processes: Ergodicity and scaling limit. SIAM J Financial Math 2014;5(1):99–136. http://dx.doi.org/10.1137/130912980.