*36th European Modeling & Simulation Symposium, 030*
21st International Multidisciplinary Modeling & Simulation Multiconference

2724-0029 © 2024 The Authors.
doi: 10.46354/i3m.2024.emss.030

# A Distributed Real-Time Multi-Agent Stock Exchange Simulator for Investigating Inter-Market Arbitrage

Artur Varosyan[1] and Dave Cliff[1,*]

[1]Intelligent Systems Laboratories, School of Engineering Mathematics and Technology, University of Bristol, U.K.

[*]Corresponding author; email address: csdtc@bristol.ac.uk

## Abstract

We describe a new multi-agent distributed stock exchange simulation environment (DSXE), built to model the global network of contemporary financial markets. DSXE is an advance on existing state-of-the-art simulation platforms available in the public domain: it is a modular and highly configurable environment which allows researchers to setup and deploy stock exchange agents and trader agents in different geographical locations using commercial cloud-computing services. The efficient implementation in C++ enables the running of large-scale simulations with many simultaneous traders. DSXE has been successfully used to model fragmented markets and to demonstrate price convergence resulting from arbitrageurs operating between two exchange venues trading the same asset. We report here on a series of experiments conducted to measure and quantify the performance and scalability of the system. The implementation successfully achieves the goal of modelling HFT, demonstrating the capability to process up to 355 messages per second. Finally, potential avenues for further research and suggested improvements to the implementation are outlined. The C++ code for DSXE is being made available as open-source code via GitHub.

**Keywords**: Stock Exchange; High-Frequency Trading; Distributed Simulation; Real-Time System; Cloud Computing.

## 1. Introduction

Contemporary financial markets consist of a network of exchanges distributed across different countries and jurisdictions. The same asset, or group of closely related assets, can be traded on different exchanges in multiple countries at the same time, and economic theory known as the *law of one price* suggests that the price across all the markets should converge on the same value. However, in practice, the complex interactions between market participants and varying delays and latencies in trading across different venues give rise to market fragmentation (Claessens (2019)). Studying and understanding the price dynamics of individual assets traded simultaneously on multiple markets is key to making informed decisions regarding financial markets. In this paper, we present first results from a new simulator designed to be remotely configured, launched, and operated via the cloud-computing facilities of Amazon Web Services (AWS). Because AWS is a global network of cloud data-centres, our simulations can be configured to operate anywhere on the planet where AWS has a presence, and our simulator's distributed network of exchanges and trading entities will then run in real-time and be subject to real-world communications and computation latencies: in this sense, our simulator (the Distributed Stock Exchange Simulation Environment, or DSXE) offers a high-fidelity simulation platform for studying issues in inter-exchange trading.

The study of financial markets and the behaviour of market participants has a long history. Chamberlin (1948) showed in 1948 how experiments in laboratory conditions with groups of students can be applied to study market dynamics and explain trader behaviour. And Smith (1962) then built on Chamberlin's work, initiating a long series of experimental studies conducted over decades, for which

he eventually received the Nobel Prize in Economics.

The scientific study of financial market mechanisms and trader behavior began in the mid-20th-Century with the founding of *experimental economics* (in which Chamberlin's and Smith's early papers are often stated to be the seminal works) which involves studying groups of human participants interacting in model markets, but in recent decades this has spawned a complementary approach, involving simulations of agent-based models: sometimes referred to as Agent-Based Computational Economics (ACE): see e.g. Tesfatsion (2023). Modelling economic processes as dynamic systems of interacting agents allows one to simulate and empirically study both complex market structures and individual trader behaviour. One of the first computerised stock exchange simulations was the Santa Fe artificial stock market, as discussed by Palmer et al. (1994), which allowed researchers to study the interactions between adaptive agents in a centralised market and their effect on price dynamics. In the years since the Sante Fe work, simulation models published in the research domain have become increasingly complex, incorporating more sophisticated market models and agent behaviours (e.g. Cliff (2018)), and accounting for factors such as communication latency (e.g. Miles and Cliff (2019)) and execution time of trading algorithms (e.g. Rollins and Cliff (2020)). While many of these implementations excel in modelling individual factors affecting price dynamics in financial markets, there has yet to emerge a model complex enough to account for all the major factors influencing trading in contemporary markets.

With the advent of High-Frequency Trading (HFT: see e.g. Aquilina et al. (2022)), a particular area of concern is the real-time performance of the model and the ability to empirically measure the effects of communication latency and processing delays in real time. Arbitrage strategies employed by HFT companies often rely on instantaneous access to live market data and the ability to co-locate trading systems at exchange venues (see e.g. Arnuk and Saluzzi (2009)). With many market observers and participants voicing opinions that HFT is predatory in nature and negatively impacts market quality (Dalko and Wang (2020)), it is more important than ever to be able to understand and study the impact of these strategies – and for that, a controlled simulation environment like that provided by DXSE is a necessity. To the best of our knowledge, no agent-based financial model capable of simulating HFT in real-time across a distributed environment has been reported in the published literature, and so the primary novel contribution of this paper is the description of DSXE's design and implementation, and the presentation of first results from the simulator. To enable other researchers to replicate and extend our work reported here, the full C++ source-code for DSXE is being made freely available on GitHub.[1]

1 https://github.com/artur-varosyan/distributed-stock-exchange-environment.

Section 2 presents a review and qualitative analysis of existing state-of-the-art financial market simulations available in the public domain. The findings from that review shaped the design and development of DSXE as a novel real-time distributed agent-based stock exchange simulation environment: this is described in Section 3. Then in Section 6 we demonstrate the successful use of DXSE to model price convergence in fragmented markets as a result of inter-market arbitrage, under two different market scenarios. Section 9 then summarises results from performance analysis of DXSE, to measure and quantify the scalability of the platform, and identify the strengths and weaknesses of the implementation: we show that DXSE successfully achieved the aim of handling HFT with a maximum recorded message throughput of 355 messages per second.

## 2. Analysis of Stock-Exchange Simulators

### 2.1. Bristol Stock Exchange (BSE)

One of the earliest open-source stock exchange simulations is the Bristol Stock Exchange (BSE) by Cliff (2018), first released in 2012. BSE is a discrete, event-driven simulation that simulates the trading of a single tradeable instrument on a continuous double auction exchange. At the heart of the simulations sits the limit order book (LOB) which records the unexecuted bids and asks currently present on the exchange. The source code also contains implementations of some of the early well-known automated trading algorithms such as the *Zero-Intelligence-Plus* (ZIP) trader introduced by Cliff (1997) and the *Adaptive-Aggressive* trader by Vytelingum (2006). BSE allows configuring a population of such traders and running simulations for a fixed duration.

#### 2.1.1. *Implementation*

The simulation works as follows: at each timestep $t$, a (pseudo) randomly chosen trader from the population of traders is given a chance to place an order. If the trader chooses to place an order, the order is matched against the resting orders in the LOB. Each trader is restricted to having only one active order in the LOB. If the new order crosses the spread, a trade has occurred and the simulation proceeds to inform all of the traders of the trade. All traders are then given a chance to update their internal state. The internal virtual time is incremented and the cycle repeats, with another random trader being given the chance to place an order. If a trader chooses to place a new order, the previous order is cancelled. Due to the stochastic nature of the selection process and the pseudo-random generation approximately following the uniform distribution, as time $t$ tends to infinity, each trader should be given an approximately equal number of chances to trade.

#### 2.1.2. *Discussion*

While being of value to education and wider academia, the BSE simulation has fundamental design flaws and limi-

tations. The BSE assumes that all transactions are sent and executed with zero latency. In the context of ubiquitous HFT in modern financial markets, where the speed of the trading algorithm and the network latency are key factors in trading profitability, this is a significant limitation of the simulation. Furthermore, the lack of market orders, where the tradeable asset is bought or sold at the best available price currently present in the LOB, and cancel orders, where a previously submitted order is cancelled, restricts the possibility of testing more complex trading strategies which rely on these actions. Finally, the restriction that each trader may only have one active order disqualifies advanced trading strategies such as spoofing or quote stuffing.

### 2.2. Threaded Bristol Stock Exchange (TBSE)

Bristol Stock Exchange served as the basis for many improved financial market simulations. One such improved simulation is the Threaded Bristol Stock Exchange (TBSE), developed by Rollins and Cliff (2020). TBSE aims to address the assumption that each automated trading algorithm takes negligible time to process a market update and submit an order to the exchange. To tackle this assumption, Rollins modified the original codebase by using multithreading. This allowed Rollins to study how profitable different automated trading algorithms are when the execution time is considered.

#### 2.2.1. *Implementation*

At the beginning of each simulation, each trader in the simulation, as well as the exchange, is assigned a unique thread to run on. Then a series of thread-safe message queues are created to facilitate the communication between the traders and the exchange, and the main simulation loop. During the simulation, the traders send orders to the exchange via the exchange message queue. The exchange processes each order one at a time, checking whether the order crossed the spread and a trade is possible against a resting order in the LOB. If a trade occurs, a trade report is sent to each trader via their respective message queue. The order in which traders are updated is fixed throughout the simulation, raising the question of whether fairness is preserved. Similarly to the original BSE, the simulation assumes that each trader may only have one active order on the exchange, and any new orders cancel previously added orders in the LOB.

#### 2.2.2. *Discussion*

The improvements implemented by Rollins certainly mean that TBSE is more reflective of real markets compared to the original BSE. The results of running numerous market simulations on the TBSE have shown that the profitability of some of the well-known trading algorithms changes significantly when their execution time impacts how quickly they can react to market changes.

To critically evaluate this implementation, it is important to identify some of its weaknesses and design flaws. Similarly to its predecessor, TBSE lacks support for market orders or order cancellations, and each trader is only allowed to hold one active order at a time. More importantly, however, as noted by Rollins, since the simulation is implemented in *Python* using the *multithreading* library, it is subject to the Global Interpreter Lock (GIL) Python.org (2024). The GIL ensures that at any one time, only one *Python* thread is allowed to access the *Python* interpreter. This means that the TBSE implementation does not make use of multiple cores of the machine it is running on. In effect, this means that there is no real parallelism and the trader threads are executed sequentially. Rollins argues that while there is no guarantee that each thread will be allocated an equal proportion of the execution time, over the course of the simulation, the total execution time of each thread should be approximately the same. It is difficult to prove this hypothesis, as the allocation of threads to run will depend on system architecture and kernel implementation.

### 2.3. Distributed Bristol Stock Exchange (DBSE)

Another stock exchange simulation derived from the BSE is the Distributed Bristol Stock Exchange (DBSE) described by Miles and Cliff (2019). Similarly to TBSE, DBSE aims to address another design flaw of the original BSE implementation - the assumption of zero latency in communication between the traders and the exchange. Since the BSE is a discrete-event simulation running on a single machine, it is impossible to model or investigate how network latency affects the market dynamics and profitability of each trader. To tackle this issue, Miles developed a distributed simulation deployed on Amazon Web Services (AWS), where trader clients and the exchange server can be deployed to separate virtual instances in different regions of the world. DBSE was built in Python by modifying and extending the source code of the BSE simulation.

#### 2.3.1. *Implementation*

While designing the simulation, Miles chose to implement a subset of the Financial Information eXchange (FIX) protocol (see e.g. Fix Trading Community (2023)). FIX is an industry-standard electronic communications protocol used for the financial markets, most notably in the NASDAQ stock exchange (see e.g. NASDAQ (2023)). The FIX protocol operates on top of the Transmission Control Protocol (TCP) and allows for bi-directional lossless communications between the exchange and the trader. The protocol provides extensive messaging capabilities, beyond what is required for a simulation. On one hand, this makes the simulation highly realistic, however on the other hand, as noted by Miles, complicates the development of the simulation. To implement the support for the FIX protocol, DBSE uses the *quickfix* library (see Miller (2024)). The library allows sending supported messages between the trader clients and the exchange server and provides an

interface to implement handlers for different incoming messages.

In the simulation configuration outlined by Miles and Cliff (2019), the exchange is hosted on an instance based in London with four separate clients: two in London, one in the United States and one in Australia, with every client running a population of traders. On each market update received, the clients shuffle the list of traders randomly, before sequentially iterating through the list and allowing each trader to place and send an order to the exchange. The exchange attempts to match the new order against the resting orders on the LOB and sends an acknowledgement message to the trader. If the new order has resulted in a trade, the exchange broadcasts the LOB data to all clients. DBSE supports limit orders as well as market orders, and order cancellations.

#### 2.3.2. Discussion

By carefully selecting different AWS regions to provision instances in, Miles was able to investigate how increasing the physical distance to the exchange impacts the latency and in turn the profitability of different trading algorithms. Through a series of simulations, Miles has shown that communication latency significantly increases with increasing distance to the exchange and that traders running on clients deployed closer to the exchange achieve better profitability. More specifically, traders running in London outperformed traders running in the United States, which in turn outperformed traders running in Australia.

DBSE is a significant improvement to the original BSE simulation. It supports market orders and cancelling orders. It can effectively model network latency arising from geographical separation. It uses an industry-standard FIX protocol for communications. Finally, it is a real-time simulation with each trader being able to hold multiple active orders on the exchange.

One of the biggest flaws in the design of the DBSE is the decision for each simulation client to host multiple traders, with each trader running sequentially during the simulation. This is not reflective of the real world, where trading companies deploy trading algorithms to machines located physically in the stock exchange building, each one running on independent hardware with a dedicated network connection to the stock exchange system. Furthermore, the manual nature of the deployment of the simulation, requiring a user to SSH into every single instance to run the experiments, limits the potential of the platform as a research tool. The simulation does not produce any output files with data regarding the market session. Instead, all data is displayed on standard output. Finally, the simulation supports running only one exchange venue at a time, meaning that scenarios such as market arbitrage cannot be simulated.

### 2.4. Distributed Threaded Bristol Stock Exchange

The latest iteration of the Bristol Stock Exchange is the Distributed Threaded Bristol Stock Exchange (DTBSE) developed by Jiang (2023). DTBSE aims to synthesise and combine the works of Rollins and Miles into a single unified implementation. More specifically, DTBSE is a real-time simulation deployed in the cloud where each client runs multiple traders, with every trader having a dedicated thread. This allowed Jiang to not only study the effects of real-world network latency as in Miles' paper but also trader execution time as in Rollins' paper.

#### 2.4.1. Implementation

The simulation consists of an exchange instance and several client instances deployed in AWS. This time, however, each client instance is running multiple threads, with one trader per thread. Each trader thread makes use of a dedicated thread-safe queue to send and receive orders and market updates. The FIX application reads incoming messages from the trader queues and sends them to the exchange server via the underlying TCP connection. Similarly, external messages from the exchange, are forwarded to the trader threads via their respective queues.

Jiang added improvements to the implementation making it easier and faster to configure and deploy simulations. DTBSE features a simple TCP server to synchronise the exchange server and the trader clients. This TCP server waits for messages from all simulation clients, before sending out a signal for the simulation to begin. Likewise, at the end of each simulation trial, the TCP server sends a signal to the exchange to reset the exchange's internal LOB ready for the next trial. At the end of each trial, simulation output files are generated and compressed into ZIP archives, before being uploaded to AWS S3 bucket storage Services (2024). This allows for easier retrieval of the simulation data. Lastly, the DTBSE simulation uses a centralised configuration file where simulation parameters can be defined.

#### 2.4.2. Discussion

Jiang was able to reproduce Miles' result and investigate how trader execution times impact trader profitability. The results show that simpler algorithms tend to perform better, particularly as the geographical distance from the exchange increases. Conversely, as the distance from the exchange increased, the performance of complex traders seemed to deteriorate. DTBSE therefore gives the research community the ability to run more complex and realistic simulations, giving new insights into the profitability of automated trading algorithms.

In the future work section of their dissertation Jiang (2023), Jiang outlines some of the possible avenues for further development and expansion of the simulation. Jiang suggests that using containerisation and services such as the Amazon Kubernetes Service (EKS) Services (2024), would allow the users to configure and run simulations more easily. This would eliminate the need for establish-

ing a manual SSH connection to each client as is the case with both DBSE and DTBSE.

While successfully combining the features of DBSE and TBSE into one platform, DTBSE also inherits some of the weaknesses of these implementations. Most notably, each simulation client is running multiple trader threads, but due to the limitations of the Python GIL, only one thread can run at any single time. Moreover, all traders running on the client share the same TCP connection with the exchange. While this should not be an issue for low-frequency trading, to model HFT accurately, each trader should have an interrupted runtime and dedicated network connection to the exchange.

Most of the previous research and financial exchange simulation development at the University of Bristol has focused on extending the functionality of the Bristol Stock Exchange. Miles, Rollins and Jiang, successively worked on the source code of the BSE, adding new features and making the simulation more realistic. The DTBSE is the result of many years of research and its success can be attributed to the contributions of all of these authors. To build an even more sophisticated and realistic stock exchange simulation, however, it was necessary to consider other leading simulation implementations available in the public domain.

## 2.5. Agent Based Interactive Discrete Event Simulation

Agent-Based Interactive Discrete Events Simulation (ABIDES: Byrd et al. (2020)) is an open-source discrete simulation developed by researchers at the Georgia Institute of Technology in collaboration with J.P. Morgan AI Research. Unlike previously considered implementations, this simulation explicitly adopts a completely agent-based approach to modelling the dynamics of trading on a stock exchange. Similarly to the BSE however, ABIDES is implemented as a single-threaded Python application.

### 2.5.1. Implementation

ABIDES is an extensive and highly configurable simulation environment. Among the many configurable parameters, the user is able to specify a computational delay for each trader agent, as well as a simulated network latency. The simulated network latency is configured using a latency matrix - a preconfigured latency in nanoseconds for each agent pair. Furthermore, ABIDES supports simulating specific dates in the past when provided with historical market data. This is achieved using a liquidity injection agent which places orders from historical data. This feature enables studying the potential market impact of different trading strategies when applied to historical prices.

The simulation is centred around a discrete event-based kernel, which acts as the proxy, sending messages between the autonomous agents in the simulation. The kernel is also responsible for maintaining a global virtual time during the simulation. The individual agents communicate with other agents solely via the kernel, with the kernel providing functionality such as sending a message to another agent and scheduling a callback at some point in the future. When the kernel processes messages it applies the previously configured latency when appropriate. The source code contains a base agent class, which all derived agents must inherit from. This base class contains methods that respond to events corresponding to different simulation stages, such as initialisation or termination of the simulation.

The `ExchangeAgent` class contains a LOB which keeps track of orders submitted by traders. When the exchange receives a new incoming order it attempts to match the order against the appropriate side of the LOB. The LOB supports partial execution, meaning that if an order can only be filled in part, the remainder is added to the LOB. The exchange informs the trader agents of the status of the order after it is received and whether it was filled, partially filled or unfilled. The exchange also supports order cancellations. To facilitate the communication of all of these messages, ABIDES uses a custom message protocol inspired by the ITCH and OUCH protocols NASDAQ (2024a)NASDAQ (2024b). These protocols are proprietary protocols developed by the NASDAQ stock exchange and used primarily for ultra-low latency trading. The former is concerned with broadcasting real-time detailed LOB information, while the latter is used for exchange-trader communication and enables submitting orders and receiving confirmations. Unlike the FIX protocol, ITCH and OUCH encode data in binary format for optimised performance and reduced network bandwidth usage. This however does not apply to ABIDES since it is a discrete single-threaded simulation, so no data is sent over the network.

### 2.5.2. Discussion

The highly configurable nature of the simulation together with the completely agent-based approach are undoubtedly the biggest strengths of ABIDES as a simulation platform. The platform can be used for simulating many different market scenarios with a high number of traders and measuring their market impact on historical prices. Moreover, the authors demonstrate that ABIDES can be used as a platform for non-financial applications, such as multiparty artificial intelligence simulations. Among the weaknesses of the simulation, the most notable ones are that latency and computational delay are pre-configured and cannot be empirically measured. A simulation which could test in real-time how the network latency and computational time of algorithms impact their profitability would provide more credible results. Furthermore, similarly to the BSE, the Python-based single-threaded implementation limits the scalability of the platform.

## 2.6. Multi-Agent eXchange Simulator (MAXE)

Thus far, the financial market simulations covered in this section have all been implemented using Python. One public-domain stock exchange simulation implementa-

tion, focused on efficiency and written in C++ is the Multi-Agent eXchange Simulation (MAXE: Belcak et al. (2021)) developed by researchers at the Oxford-Man Institute of Quantitive Finance, released as recently as 2020. Similarly to ABIDES, it is a discrete agent-based simulation built on a message-driven architecture.

#### 2.6.1. Implementation

In MAXE, all entities of the simulation are modelled and implemented as agents - this includes traders, exchanges but also possibly other types of agents such as news outlets. All agents remain dormant unless they have received a message. On receiving a new message, an agent is given unlimited execution time to process the message and decide whether (or not) to take an action. Similarly to ABIDES, MAXE allows users to configure a non-negative processing delay. This delay includes both the network latency and the decision time for any agents.

The `ExchangeAgent` implementation is capable of handling many types of messages including limit orders, market orders, order cancellations, order retrievals and subscriptions to different market events such as trades. The `ExchangeAgent` holds a pointer to an LOB which contains a separate queue for asks and bids. The LOB processes new orders against the existing resting orders and supports partial order execution. On each successful trade, the `ExchangeAgent` notifies all trade subscribers.

#### 2.6.2. Discussion

MAXE is a versatile simulation with many potential use cases. Its architecture implicitly allows running simulations with multiple exchanges at the same time, and the presence of the XML configuration file allows users to configure arbitrary simulations with different types of agents. MAXE includes implementations of different matching algorithms including the most common price-time first-in-first-out (FIFO) algorithm as well as pro-rata order matching, where incoming orders are matched to resting orders proportionally to their quantity.

One of the biggest strengths of this simulation is its performance and speed of execution. The efficient implementation means that the simulation is easily scalable and handles a large number of agents without issues. The authors compared the implementation to ABIDES and demonstrated that with an increasing number of agents, the simulation time on MAXE is increasingly lower than that of ABIDES when configured to run the same simulation. Furthermore, the authors noted that while running simulations with 450 agents on ABIDES resulted in a memory error, the MAXE implementation with the same configuration runs efficiently using <100 MiB of memory.

When critically evaluated, however, the MAXE implementation suffers from some of the same limitations as ABIDES, namely being a discrete simulation where real-world network latency and execution times cannot be investigated and must be pre-configured.

### 2.7. MarketSim

The final stock exchange simulation considered in this section is the MarketSim simulation platform developed by Duffin and Cartlidge (2018). This is a discrete-event agent-based simulation with support for multiple trading venues to investigate the effects of latency arbitrage in fragmented markets.

#### 2.7.1. Implementation

The simulation is implemented as an object-oriented application in Java. The source code is built on top of *Desmo-J*, an open-source library for discrete-event simulations developed by the University of Hamburg University of Hamburg (2024). The simulation is inherently agent-based, with each agent represented by a node in a weighted directed graph. The weights between any two given nodes represent the latency that should be applied when messages are sent between them. The `Model` class acts as the proxy, and facilitates this communication, with each message being represented by a packet. Agents can choose to implement handlers for different types of packets such as limit orders and acknowledgements.

Duffin provided two implementations of exchange agents - a continuous double auction exchange, typical of most contemporary financial exchanges, and a discrete-time call auction, where incoming orders are batched and matched at a regular time interval. Both exchange implementations only support limit orders, however, due to the well-structured code and class hierarchy, adding more complex order types would not require significant effort. While only one agent is active at any time during a simulation, the implementation uses multiple threads to run multiple independent experiments in parallel.

#### 2.7.2. Discussion

Duffin's paper was heavily inspired by the works of Wah and Wellman (2013), who used an agent-based model to show that latency arbitrage negatively affects market efficiency in fragmented markets. To replicate their experiments, Duffin built a two-market model with a population of zero-intelligence traders and an arbitrage trader with faster access to the market data. Duffin was then able to show that the presence of latency arbitrage benefits fragmented markets, casting doubt on Wah and Wellman's initial results.

MarketSim and Wah and Wellman's models are the only agent-based models developed for studying arbitrage scenarios published in the literature in recent years. The models inherently support having multiple exchanges in the same simulation. They therefore will serve as an inspiration for designing some of the parts of the DSXE simulator introduced in this paper. However, being discrete-event simulations, both models do not allow one to empirically measure factors such as computational delay or communication latency and therefore have limited potential for simulating HFT.

# 3. Designing for low-latency networking

## 3.1. Network protocols and performance

Given the distributed nature of the simulation and the aim of modelling HFT accurately, a strong emphasis was placed on developing an efficient and highly performant simulation. More specifically, significant effort was put into developing a system with low-latency communication.

To ensure that any computation and processing is as fast as possible, C++ 20 was chosen as the language of the implementation. The low-level nature of C++ and the compilation of source code to machine code allows C++ code to make better use of the underlying hardware and achieve higher performance when compared to programs written in other languages such as Python or Java.

To facilitate the communication between any two agents in the simulation, the simulation platform supports two transport-layer protocols: Transport Control Protocol (TCP) and User Datagram Protocol (UDP) Kumar and Rai (2012). The former is a loss-less connection-oriented protocol, while the latter is a simple protocol that does not guarantee the order of delivery of packets.

TCP requires two entities to establish a connection and perform a handshake before exchanging further messages. On each packet received, the receiver sends an acknowledgement (ACK) with a sequence number back to the sender. If the sequence number does not match or the acknowledgement is not received within a time interval, the sender retransmits the packets. Each segment contains a checksum that ensures the integrity of any messages. These two features guarantee reliable delivery and make TCP the protocol of choice for electronic communication between traders and exchange venues.

UDP on the other hand is a much faster protocol used for applications where the speed of delivery takes priority over reliable delivery. It does not require a handshake and does not maintain a connection between two communicating parties. It is most commonly used in real-time streaming of data and media, and in the context of financial markets, it is used for publishing market data to all market participants at extremely high frequency. If a packet with market update information is lost, little to no damage occurs as another packet immediately follows.

## 3.2. Implementation

In the simulation, networking has been implemented using the `boost::asio` library Boost.org (2024), an open-source low-level IO and networking library for modern C++. The `NetworkEntity` class encompasses the functionality to send and receive messages using both protocols. For the purposes of the DSXE simulation, a *message* denotes a piece of information sent via a previously established TCP connection and a *broadcast* denotes a piece of information sent via a UDP connection. The `NetworkEntity` class provides the interface for agents to communicate through either of two means. Going back to our graph analogy, you may now consider the simulation system to be a multi-graph - that is a graph with parallel edges between any two given nodes. More specifically any pair of nodes may have two parallel edges, one representing the TCP communication and one representing the UDP communication.

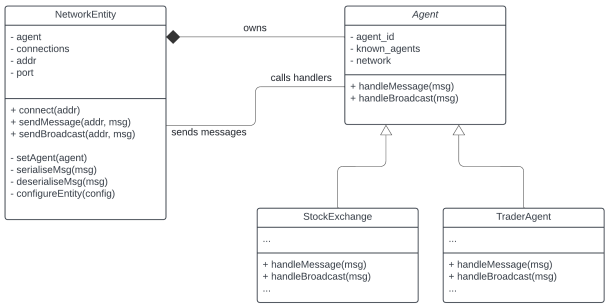


**Figure 1.** UML architecture diagram of `Agent` and `NetworkEntity` classes

The relationship between a `NetworkEntity` and an `Agent` can be described as a composition. The lifecycle of an Agent is wholly owned by the `NetworkEntity`. The `NetworkEntity` may own only one `Agent` at a time. Similarly, a `NetworkEntity` may remove the Agent and replace it with another one. When a new message or broadcast is received, the `NetworkEntity` calls a handler function of the underlying `Agent`. The `Agent` holds a non-owning reference via a pointer to the `NetworkEntity`. This allows it to use the interface to initiate communication with and send messages to other `NetworkEntities` (and therefore `Agents`) in the simulation. Intuitively, while the `Agent` can be considered as the brain of the system, the `NetworkEntity` can be considered as the body. While this analogy is useful for conceptualising this relationship, it is not a perfect comparison, as the `Agent` may be replaced at any time. Figure 1 shows the architecture diagram illustrating the two classes and their interactions.

# 4. Harnessing the power of the cloud

## 4.1. Motivation

There are many motivations for a distributed computing approach to developing a financial markets simulation. With the rise of public cloud computing services, it is trivial to deploy applications to servers located in arbitrary geographical locations around the world and empirically measure the latency between them. This allows for studying and investigating the effects of latency due to geographical distance between two distant parties, such as traders trading in exchange venues across the ocean. This was the motivation behind the work by Miles and Cliff (2019), where they showed that increasing latency due to geographical distance negatively affects automated trader profits. Perhaps more importantly, the use of cloud computing platforms such as AWS, allows one to study the effects of co-location on HFT strategies. As noted by Zook and

Grote (2017) contemporary financial exchanges operate co-location facilities or server farms, where HFT companies rent racks of computers to maximise proximity to the matching engine servers. The physical distance between the trading computer and the exchange servers is critical to HFT companies to the extent that major stock exchanges offer standardised cable lengths. Arnuk and Saluzzi (2009) argue that HFT companies are able to achieve almost risk-free profits by engaging in latency arbitrage enabled by this co-location at stock exchange venues.

### 4.2. Cloud infrastructure

To accurately model co-location and physical distance between agents, the simulation is deployed on Amazon Web Services (AWS). To ensure that each trader agent has a dedicated and uninterrupted compute time and network connection, the EC2 service Services (2024) is used with each agent running on an independent instance. EC2 allows the provisioning of virtual compute instances in any of the supported geographical regions. The service offers a range of instance options, with different numbers of virtual CPUs (vCPUs), amounts of RAM and network bandwidth. The highly configurable and scalable nature of the service made it an appropriate choice for running the simulation. Since the source code is written in C++, which is in turn is compiled to architecture-specific machine code, containerisation was used to ensure cross-platform compatibility. Containers are lightweight, isolated environments that contain an executable package that can be run on any platform supporting the containerisation engine. This was one of the improvements suggested by Jiang (2023) for DTBSE. For this project, all source code was compiled and built into a Docker (2024a) image before being deployed to the server instances in the cloud.

### 4.3. Provisioning simulation nodes

The deployment and orchestration of the simulation distributed across tens and hundreds of virtual servers in the cloud has proved to be a challenge. It was essential to come up with effective ways of deploying the code and synchronising the simulation. A simple shell script was used to automate the deployment and set-up of each simulation node. Below is a step-by-step description of the process of provisioning instances and deploying the executable:

1. A user provisions a set number of EC2 instances in the AWS console.
2. A newly provisioned instance automatically runs the set-up script.
3. The instance installs the Docker engine using the built-in package manager.
4. The latest version of the DSXE image is then pulled from Docker Hub Docker (2024b).
5. The container starts in the background and the server node is ready to be used in the simulation.

This almost entirely automated process means that configuring the simulation to run on a high number of server nodes is trivial.

### 4.4. Orchestrating the simulation

The next step in designing the simulation environment was to define how the simulation is configured and simulation nodes synchronised. The bulk of the configuration is done using an XML file. The configuration file allows to specify the IP addresses of all simulation nodes involved in the current simulation. The user may also specify the types of agents present in the simulation, such as different trading agents and exchange agents, and their properties.

To synchronise all simulation nodes effectively, a new type of `Agent`, the `OrchestratorAgent` was implemented. When the executable is run in the orchestrator mode, the local configuration file is read and an instance of the `OrchestratorAgent` is instantiated. The `OrchestratorAgent` attempts to connect to all server instances listed in the configuration file. Upon a successful connection, the `OrchestratorAgent` sends a configuration message to each server node to assign it an agent identity and configure its properties. The `NetworkEntity` on the other side of the connection reads the configuration message and creates an instance of the specified `Agent` with the provided properties. The node is then considered to be fully configured and actively running an `Agent`. At the end of the simulation, the server nodes remain live and can be reused for the next simulation trial. The user may restart the orchestrator with a different configuration file. The server nodes will then be assigned different agent identities and a new simulation trial will begin.

## 5. Stock exchange implementation

The network infrastructure outlined above forms the basis of a scalable and efficient simulation platform. The subsections below describe in detail how a complex stock exchange environment was built on top of this platform.

### 5.1. Scalable Limit Order Book (LOB)

To begin with, the concept of orders is introduced to the simulation. Fundamentally, an order contains a unique identifier, a symbol identifying the asset traded, a quantity, a side, describing whether the order is a bid or an ask, metadata about its sender and an indication of its current status. Several concrete types of orders have been implemented, namely a limit order and a market order. The former additionally contains a maximum or minimum price at which the asset should be bought or sold. Any concrete order implementation is derived from the generic `Order` class. The full list of supported order types in DSXE and their descriptions can be found in Table 1.

Limit orders which have not been executed immediately, known as *resting orders*, are kept in a LOB. The LOB

| Order type | Description |
|---|---|
| Market order | The order is executed immediately at the best available price. If the order cannot be executed in full, the remainder will be cancelled. |
| Limit order | The order must be executed at price specified or a better price. Any remainder quantity of the order is inserted into the LOB. |
| Immediate-or-Cancel (IoC) | The order must be executed immediately at the price specified or a better price. Any remaining quantity is cancelled. |
| Fill-or-Kill (FoK) | The order must be executed immediately and in-full at the price specified or a better price. Otherwise the order is cancelled. |
| Cancel order | If the order has not been filled in-full cancel any remaining quantity immediately. |

**Table 1.** Supported order types in DSXE

consists of two priority queues - one for the bids and one for the asks. The orders are sorted according to price-time priority. Orders are first ranked according to their price, and in the case of two orders with the same price, according to their time of arrival. The in-built priority queue handles insertions of new orders, and retrievals of best bids and asks efficiently.

### 5.2. Fast matching engine

Unlike in a discrete-event simulation, the performance of the matching engine in a real-time context is key to the successful operation of an exchange. The matching engine must be able to sequentially process a large number of messages in the order of arrival at extremely low latency and zero downtime Aquilina et al. (2022).

This requirement calls for an uninterrupted runtime and efficient implementation of the matching engine logic. In DSXE, the matching engine therefore runs on a standalone thread. A dedicated IO thread handles any communication between the exchange and the traders. The two threads communicate via a custom implementation of a synchronised queue. This is a classic example of the producer-consumer problem. On every new message received, the IO thread, or the *producer*, pushes the message to the synchronised queue. The matching engine, or the *consumer*, continuously consumes from the queue until the queue is empty. If the queue is empty, i.e. no new messages are present, the matching engine thread sleeps until a new message is pushed to the queue. This allows to create a buffer between messages received and messages being processed. Similarly, when the matching engine sends an acknowledgement message or a market data broadcast to market subscribers, it posts the job to the IO thread and proceeds to read the next message from the queue. The `boost::asio` library provides an interface that allows for a smooth handover of tasks from non-IO threads to IO threads. For a diagram illustrating two threads working together in DSXE, see Varosyan (2024).

Unlike some of the other simulation platforms, the DSXE matching engine supports partial order execution, meaning that limit orders submitted by traders can be partially executed and their remainder can be saved in the LOB for later execution. Alternatively, trader agents can choose to submit Fill-or-Kill (FoK) and Immediate-or-Cancel (IoC) orders. The former describes orders which must be fulfilled in full or cancelled entirely, while the latter describes orders which can be partially executed but no remainder is saved in the LOB.

### 5.3. Unified implementation

The `StockExchange` class encompasses both the scalable LOB and the fast matching engine implementation. Being derived from the base `Agent` class, the `StockExchange` class implements handlers for different types of incoming messages. The `StockExchange` maintains a separate LOB for each symbol traded. Trader agents may choose to subscribe to market data related to any of the assets trading on that particular exchange. Furthermore, the `StockExchange` maintains a tape of all trades that have been executed and messages that have been received. This granular data is then written to a CSV file for further processing and data analysis.

The completely agent-based approach enables one to go beyond having one stock exchange in a simulation and to model several exchange venues in a single trial. It is trivial to set an arbitrary number of `StockExchange` agents in a simulation and define the population of automated traders trading in any subset of them. This allows for models and simulation of greater scale, and studying market scenarios involving multiple exchanges, such as market arbitrage. For this reason, DSXE is best thought of as an *environment* or *platform*, rather than a single program or simulator.

## 6. Using DXSE for Experiments

### 6.1. Statistical analysis

The subsections below demonstrate the features of DSXE and describe some of the results obtained from running different experiments. Each experiment ran for a fixed duration of 300 seconds. Each set of results was obtained by replicating the experiment 5 times and taking the median across repetitions to account for anomalies in the data. The data was visualised and analysed, and appropriate conclusions were drawn.

### 6.2. Compute and networking

As described in Section 4, the simulation was deployed on AWS. All EC2 compute nodes used in the simulation were deployed in the `eu-west-2a` availability zone of the `eu-west-2` region. This effectively achieves the effect of co-location, as availability zones are isolated data centres located within the same region with sub-millisecond communication latency between any two given instances. This was consistent with the experiments, as the average measured latency was 0.18 ms. More powerful compute instances were chosen for the exchanges used in the sim-

ulation and less powerful for the traders. The two types of AWS instances used, and their specifications, were as follows: for the exchanges we used AWS `c5.2xlarge` instances which have 8 vCPUs, 16GiB of memory, and up to 10Gbps network bandwidth; for the traders we used AWS `t3.micro` instances which have 2 vCPUs, 1GiB of memory, and up to 5Gbps of network bandwidth. The main consideration behind the instance choice for the exchange was the requirement of high network bandwidth to handle concurrent high-frequency communication from the traders, and a high clock rate to process through orders with low latency. The individual traders in turn require much less computational power and network bandwidth as they communicate solely with the exchange agent.

## 7. Arbitrage with zero-intelligence traders

### 7.1. Market setup

The scenario studied in this subsection consists of two independent exchanges trading the same asset. The trading window for both exchanges is assumed to open simultaneously and lasts for 5 minutes. Each exchange contains a population of zero-intelligence traders of type ZI-C, 20 buyers and 20 sellers. Each trader in the experiment is assigned a private limit price at the start of the experiment. The exchanges have fixed and symmetrical supply and demand curves, with theoretical equilibrium prices of 100 for Market 1 and 70 for Market 2. Every trader is intra-marginal and may engage in a trade. Each trader may only trade at one exchange. The traders have been initialised with a trading interval time of 500ms.

This market setup describes an example of two fragmented markets with complete information asymmetry. The two populations of traders can only access the market data from their respective exchanges and are unaware of the state of the other exchange throughout the simulation. In the second experiment, an arbitrageur is introduced to the markets after 60 seconds of trading. The arbitrage agent has instantaneous access to both markets and may submit orders to either exchange. The arbitrage agent has also been configured with a faster trading interval of 250ms.

### 7.2. Price convergence

Figure 2 shows the median trade price per second for each one of the two experiments, with and without arbitrage. Initially, the trade price oscillates around the respective theoretical equilibrium prices on each exchange as expected. After 60 seconds, when the arbitrage trader is introduced, the prices across the two exchanges move towards convergence instantaneously. From that point onwards, the prices on the two exchanges oscillate in harmony around a new equilibrium. When the trade price decreases in Market 1, the arbitrageur submits buys in Market 1 and sells in Market 2 to make a risk-free profit, causing the trade price in Market 2 to follow. The reverse also

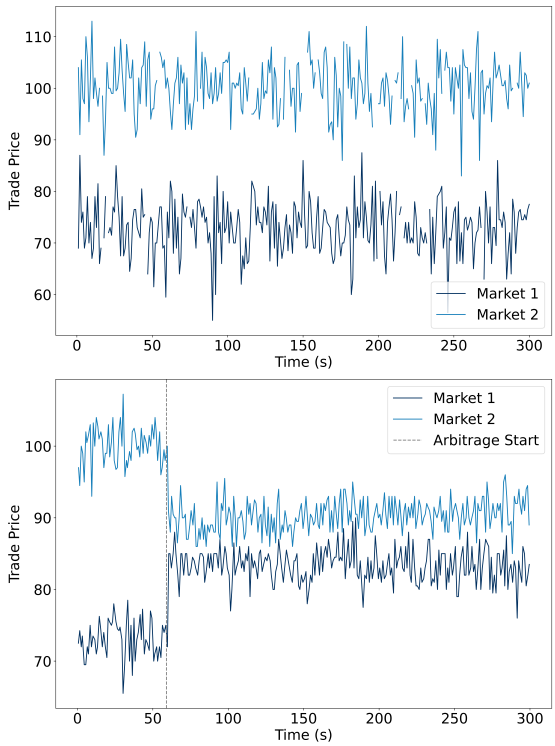


**Figure 2.** Stock price in fragmented markets with ZI-C: upper figure is without arbitrage; lower figure is with a single arbitrage agent enabled at time=60s.

holds. While no full price convergence has been achieved and the prices across the two exchanges continue to differ by some offset, this is likely due to the sparse distribution of order prices around the theoretical cross-exchange equilibrium. As implied by the name, zero-intelligence traders do not respond to market events and therefore continue generating prices according to the uniform discrete distribution after arbitrage begins. The arbitrage trader will in turn continue submitting limit orders whenever there is a sufficient arbitrage opportunity, bringing the prices across the two exchanges closer.

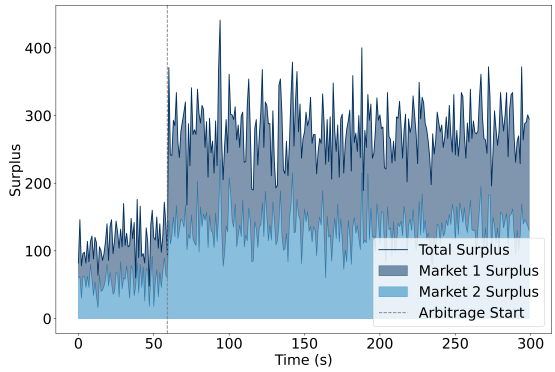


**Figure 3.** Surplus in fragmented markets with ZI-C

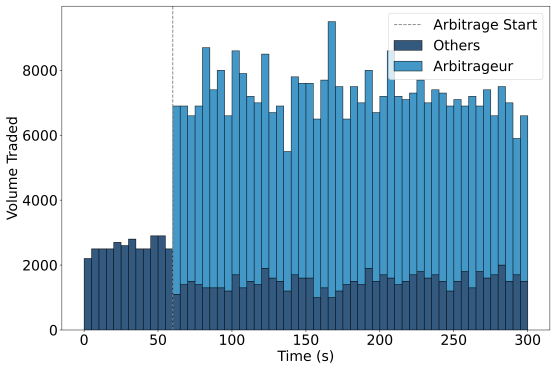


**Figure 4.** Trading volume in fragmented markets with ZI-C

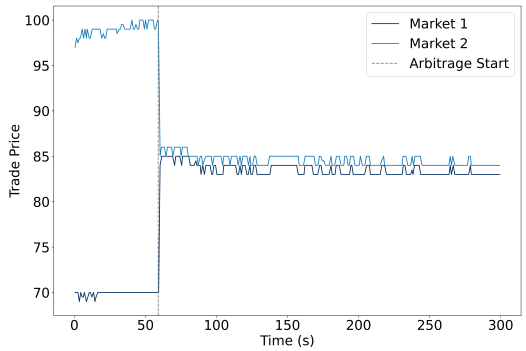


**Figure 5.** Stock price in fragmented markets with ZIP

### 7.3. Market surplus and trading volume

The dynamics of the markets after the introduction of the arbitrage trader were studied in more depth. Figure 3 illustrates the total market surplus, defined as the difference between the trade price and each trader's private limit value. A higher surplus after the introduction of a trader is expected as the arbitrage trader generates profit until full convergence occurs. Since no full convergence is achieved, the two markets continue to exhibit a higher total surplus for the remainder of the simulation.

Similarly, the total trading volume across the two exchanges increases dramatically with the introduction of the arbitrage trader. Since the arbitrage trader is configured to trade faster and is able to continue making a profit on the price discrepancies between the two exchanges indefinitely, a higher trading volume is expected. This is illustrated in Figure 4. The arbitrage trader is responsible for the vast majority of trades on the two exchanges once arbitrage begins.

## 8. Arbitrage with reactive traders

While the market setup used in the previous experiment is a source of valuable insight into how the price dynamics change in the presence of an arbitrageur, it is not indicative of the real world. The population of zero-intelligence traders used in the experiment does not react to market events and continues to submit orders at random quote prices. It would be desirable to investigate the price convergence with a more intelligent and realistic trader that considers the current state of the market when making trading decisions.

### 8.1. Market setup

A similar experiment was therefore conducted with a population of ZIP traders. ZIP traders actively aim to make a profit and adjust their profit margins according to the current trade price in the market and whether they successfully engaged in a trade. The market setup closely resembles the previous experiment. Two exchanges are configured with a fixed population of 20 buyers and 20 sellers each. The theoretical equilibrium price is 70 in Market 1 and 100 in Market 2. As previously, an arbitrage trader is introduced to the markets after 60 seconds.

### 8.2. Stronger price convergence

Figure 5 shows the median trade price per second recorded during the experiments. In line with previously published findings (Cliff (1997)), ZIP traders converge on the theoretical equilibrium price on each exchange. When the arbitrage trader is introduced, the prices across the two exchanges instantaneously move towards the theoretical global equilibrium. Unlike with the population of zero-intelligence traders, the trade prices remain close to the equilibrium for the remaining duration of the simulation. Interestingly, the prices across the two exchanges converge with a constant difference in value of 1. Since the arbitrage trader only submits orders when there is an arbitrage opportunity, and has been configured with a faster trading interval, it provides the majority of the liquidity in the market and can manipulate the markets to converge at a price which guarantees continuous arbitrage opportunity.

### 8.3. Higher market surplus and trading volume

During the first 60 seconds of the simulation, the two markets populated with ZIP traders exhibit much higher surplus and volume traded than the respective zero-intelligence counterparts. The total market surplus for ZIP markets is 3.53x higher. This can be explained by the fact that ZIP traders are better at price discovery and therefore are better able to extract the maximum surplus given the supply and demand in the market. This can be also seen in the trading volume, which is 3.6x higher compared to the ZI-C population during the pre-arbitrage period.

When the arbitrage trader is introduced, both the trading volume and market surplus increase momentarily before levelling off seconds later. This is due to the additional liquidity present in the market while the arbitrage trader brings the two prices closer to the global market equilibrium. Once the prices have converged, the arbitrage

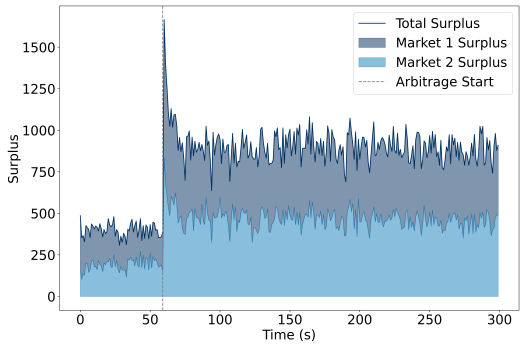


**Figure 6.** Surplus in fragmented markets with ZIP

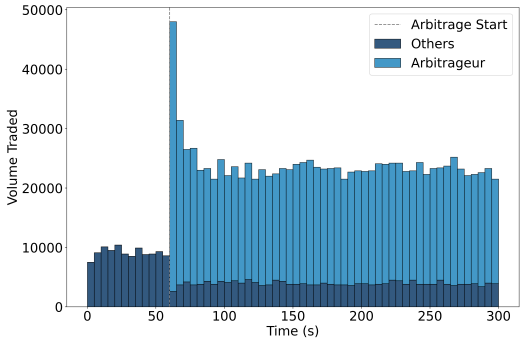


**Figure 7.** Trading volume in fragmented markets with ZIP

continues, albeit generating less surplus.

## 9. DXSE Performance testing and scalability

### 9.1. Experimental setup

To test the performance and scalability of the DSXE implementation, a series of experiments with an increasing number of zero-intelligence traders in one market are performed. The experiments begin with a total population of 20 traders, 10 buyers and 10 sellers. The traders are initialised to give rise to a flat supply and demand curve with a theoretical equilibrium price of 100. The population of traders is scaled by factors of 2, 3, 4, 5, 6, and 7, reaching up to 140 simultaneous traders. Every trader is running on a dedicated server instance, meaning that up to 141 AWS EC2 instances were used during the experiments. Several key metrics such as processing times and throughput are recorded and presented below.

### 9.2. Metrics

- **Message Throughput:** The total number of messages processed per second by the exchange.
- **Message Interarrival Time (ms):** The time between any two given consecutive messages received by the exchange.
- **Total Processing Time (ms):** The time taken for a message to be fully processed by the exchange, including time spent in the IO thread and matching engine.
- **Matching Engine Processing Time (ms):** The time taken for a message to be processed by the matching engine only.
- **Max Memory Usage (MB):** The maximum recorded memory usage of the exchange during the simulation.

### 9.3. Performance analysis

Figure 8 (upper) shows the number of messages per second processed by the exchange during the simulation. The message throughput scales linearly with the number of traders up until 100 simultaneous traders are present. The exchange implementation is able to comfortably accommodate this number of simultaneous traders. During this time, the median total processing time was recorded to be under 5 ms. The peak message throughput achieved across all simulations was 355 messages per second. This corresponds to a message interarrival time of 2.816 ms. The implementation therefore achieves the goal of supporting HFT.

Beyond 100 traders, the performance of the exchange begins to drop rapidly, with the total processing time growing exponentially and the total message throughput decreasing. Figure 9 illustrates the proportion of the processing time spent in the matching engine thread. While the median processing time in the matching engine increases linearly with the increasing number of traders, it surprisingly remains well below 1 ms. Therefore, the majority of the processing delay, as seen in Figure 8 (lower), can be attributed to the IO thread, implying that it is the bottleneck of the system. As described in Section 5.2, the IO thread handles and serialises any incoming messages and adds them to the queue to be read by the matching engine. It also is responsible for deserialising and sending outgoing messages from the exchange. For every additional order message received, the exchange must send a market update to all the market subscribers. Therefore the relationship between incoming messages and outgoing messages can be expressed as follows: $msgs_{in} = n$; $msgs_{out} = n * msgs_{in} = n * n = n^2$ where $n$ is the number of traders, and $msgs_{out}$ and $msgs_{in}$ are the number of messages sent and received by the exchange per second respectively. It follows that with an increasing number of simultaneous traders, the total processing time grows exponentially. Figure 10 shows the maximum memory usage recorded during the simulation for each market configuration tested. The max memory increases in an exponential trend, peaking at just over 100 MB of memory used. This represents a fraction of the 16 GiB of memory available in the compute instance. This furthermore implies that the computation is CPU-bound on the IO thread.

### 9.4. Potential improvements

The analysis of performance metrics recorded during testing provides insight into the bottlenecks of the implementation. The following improvements are believed to in-

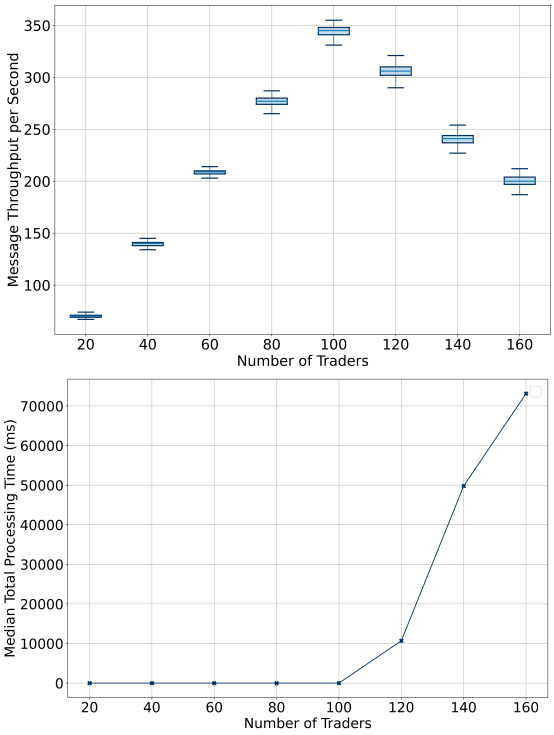


**Figure 8.** Implementation scalability performance: upper graph is message throughput; lower graph is total processing time.

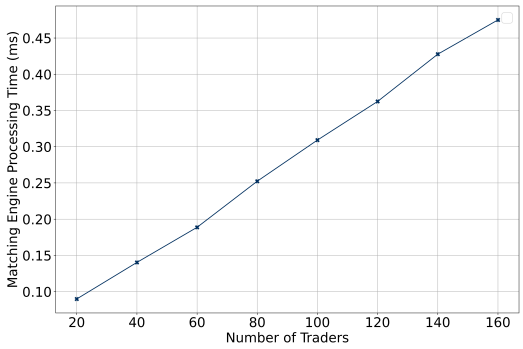


**Figure 9.** Matching engine processing time

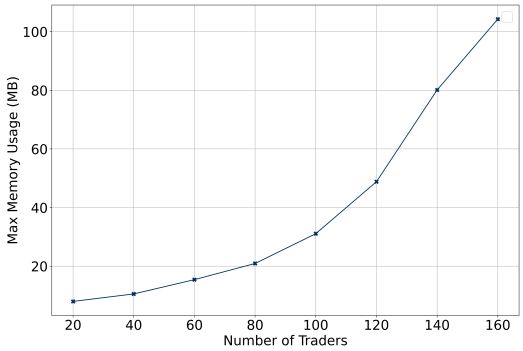


**Figure 10.** Maximum memory usage

crease the message throughput potential of the system:

1. **Separate outgoing and incoming IO threads**
A separation of concerns is desirable in this implementation. The IO thread should be divided into an incoming IO thread handling TCP communication from all the traders and an outgoing IO thread sending TCP messages to the traders and sending UDP broadcasts with market data.
2. **Minimising work done on the IO threads**
The work done by the IO threads should be minimised to allow for greater message throughput. Currently, the IO thread redundantly deserialises all messages into a character stream for every outgoing UDP broadcast message with market data. Since the same market data message is sent to all subscribers of the market, the message could be deserialised once and the same array of characters sent to all market participants.
3. **Thread pool for incoming IO communication**
Lastly, a pool of threads could be used to handle incoming communication from the traders. This would make better use of the multicore processors on the compute instances. Each thread would read from the TCP connection buffer, proceed to serialise the message and push it to the queue. Since the matching engine processes each message in the queue sequentially, this approach maintains fairness while maximising incoming message throughput.

While the potential improvement is difficult to quantify, a severalfold increase in message throughput is predicted.

## 10. Future work

The versatility of the simulation developed in this paper allows for studying price convergence in arbitrary market configurations. This paper is our first publication to describe DSXE and many potential avenues for further investigation remain open, including the following:

1. **Latency arbitrage in fragmented markets**
The simulation platform could be extended to conduct experiments with a high-frequency arbitrageur and a population of traders trading at multiple exchange venues simultaneously, building on top of the findings from previous research by Wah and Wellman (2013) and Duffin and Cartlidge (2018).
2. **The impact of co-location on arbitrageur profits**
Since co-location can be trivially achieved in DSXE by deploying the exchange agents and trader agents in the same availability zone, it allows one to investigate how co-location and the latency between the arbitrageur and the exchange impact price dynamics and arbitrageur profits.
3. **Predatory trading by HFT**
Given the high message throughput of the platform and its measurable latency and processing delay, predatory trading strategies such as spoofing and quote stuffing could be investigated in a controlled simulation environment.

Finally, while the DSXE platform has demonstrated

high performance and ability to handle a large number of simultaneous traders, the analysis of performance metrics provided insights into the bottlenecks of the implementation. The evidence suggests that the potential improvements outlined in Section 9.4 would further increase the scalability and performance of the system.

## 11. Conclusion

In this paper we have introduced DSXE which was designed and implemented from scratch in response to our careful review of existing public-domain financial exchange simulators reported in the literature. We have shown DSXE being used for real-time experimental studies of arbitrage trading that would be difficult or impossible with the previously-available simulators, and we have presented results from performance profiling which show that DSXE is fast, robust, and scalable. The C++ source-code for DSXE has been made freely available as open-source on GitHub, so that other researchers can use DSXE as a platform for further experimentation, and so that other software developers can build upon the foundations that we have established here.

## References


Aquilina, M., Budish, E., and O'neill, P. (2022). Quantifying the high-frequency trading "arms race". *The Quarterly Journal of Economics*, 137(1):493–564.

Arnuk, S. and Saluzzi, J. (2009). Latency arbitrage: The real power behind predatory high frequency trading. *Whitepaper, Themis Trading*.

Belcak, P., Calliess, J.-P., and Zohren, S. (2021). Fast Agent-Based Simulation Framework with Applications to Reinforcement Learning and the Study of Trading Latency Effects. In *Int. W'shop Multi-Agent Systems and Agent-Based Simulation*, pages 42–56. Springer.

Boost.org (2024). Boost C++ Libraries. *www.boost.org*.

Byrd, D., Hybinette, M., and Balch, T. H. (2020). ABIDES: Towards high-fidelity multi-agent market simulation. In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 11–22.

Chamberlin, E. H. (1948). An experimental imperfect market. *J. Political Economy*, 56(2):95–108.

Claessens, S. (2019). Fragmentation in global financial markets: good or bad for stability? *BIS Working paper*.

Cliff, D. (1997). Minimal-intelligence agents for bargaining behaviors in market-based environments. Technical Report HPL-97-91, Hewlett-Packard Labs.

Cliff, D. (2018). BSE: a minimal simulation of a limit-order-book stock exchange. *arXiv preprint arXiv:1809.06027*.

Dalko, V. and Wang, M. H. (2020). High-frequency trading: Order-based innovation or manipulation? *Journal of Banking Regulation*, 21(4):289–298.

Docker (2024a). Overview. *docs.docker.com/get-started/overview*.

Docker (2024b). Overview of Docker Hub. *docs.docker.com/docker-hub*.

Duffin, M. and Cartlidge, J. (2018). Agent-based model exploration of latency arbitrage in fragmented financial markets. In *Proc. SSCI*, pages 2312–2320. IEEE.

Fix Trading Community (2023). FIX Trading Community v1.9. *www.fixtrading.org/what-is-fix/*.

Jiang, A. (2023). Implementation of a Distributed and Threaded Exchange Simulator. *MSc Thesis, Department of Computer Science, University of Bristol*.

Kumar, S. and Rai, S. (2012). Survey on transport layer protocols: TCP & UDP. *International Journal of Computer Applications*, 46(7):20–25.

Miles, B. and Cliff, D. (2019). A cloud-native globally distributed financial exchange simulator for studying real-world trading-latency issues at planetary scale. *arXiv preprint arXiv:1909.12926*.

Miller, O. (2024). Quickfix. *pypi.org/project/quickfix*.

NASDAQ (2023). FIX Protocol. *www.nasdaqtrader.com/Trader.aspx?id=FIX*.

NASDAQ (2024a). Historical TotalView-ITCH. *nasdaqtrader.com/Trader.aspx?id=ITCH*.

NASDAQ (2024b). OUCH Advanced Technology. *nasdaqtrader.com/Trader.aspx?id=OUCH*.

Palmer, R. G., Arthur, W. B., Holland, J. H., LeBaron, B., and Tayler, P. (1994). Artificial economic life: a simple model of a stockmarket. *Physica D*, 75(1-3):264–274.

Python.org (2024). Thread-based parallelism. *docs.python.org/3/library/threading.html*.

Rollins, M. and Cliff, D. (2020). Which trading agent is best? Using a threaded parallel simulation of a financial market changes the pecking-order. *arXiv preprint arXiv:2009.06905*.

Services, A. W. (2024). What is AWS? - Cloud computing with AWS. *aws.amazon.com/what-is-aws/*.

Smith, V. L. (1962). An experimental study of competitive market behavior. *J. Political Economy*, 70(2):111–137.

Tesfatsion, L. (2023). Agent-based computational economics: Overview and brief history. *AI, Learning and Computation in Economics and Finance*, pages 41–58.

University of Hamburg (2024). DESMO-J: A Framework for Discrete-Event Modelling and Simulation. *desmoj.sourceforge.net/home.html*.

Varosyan, A. (2024). Distributed multi-agent stock exchange environment for investigating arbitrage and price convergence. *MSc Thesis, School of Computer Science, University of Bristol; SSRN:4893697*.

Vytelingum, P. (2006). *The structure and behaviour of the continuous double auction*. PhD thesis, University of Southampton.

Wah, E. and Wellman, M. P. (2013). Latency arbitrage, market fragmentation, and efficiency: a two-market model. In *Proceedings of the fourteenth ACM conference on Electronic commerce*, pages 855–872.

Zook, M. and Grote, M. H. (2017). The microgeographies of global finance: High-frequency trading and the construction of information inequality. *Environment and Planning A: Economy and Space*, 49(1):121–140.