Dissertation Type: enterprise


DEPARTMENT OF COMPUTER SCIENCE

# Architecting and Implementing a Globally Distributed Limit Order Book Financial Exchange for Research and Teaching

---

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

---

Friday 10$^{th}$ May, 2019

# Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

# Contents


**1 Introduction and Motivation** **1**
- 1.1 Algorithmic Trading . . . 1
- 1.2 Why is this study important? . . . 2
  - 1.2.1 Flash Crash . . . 2
  - 1.2.2 Latency Arbitrage . . . 2
- 1.3 Notable Challenges . . . 3
- 1.4 Summary of Aims and Objectives . . . 4

**2 Technical Background** **5**
- 2.1 Auction Types . . . 5
- 2.2 Experimental Economics . . . 6
- 2.3 Limit Order Book . . . 7
- 2.4 Bristol Stock Exchange . . . 8
  - 2.4.1 Exchange . . . 9
  - 2.4.2 Trading Robots . . . 10
  - 2.4.3 Simulation . . . 13
- 2.5 Case Study: Jane Street Exchange (JX) . . . 13
  - 2.5.1 Exchange Components . . . 14
  - 2.5.2 Key Network Technologies . . . 16
- 2.6 Cloud Computing . . . 16
  - 2.6.1 Amazon Web Services . . . 17

**3 Project Execution** **18**
- 3.1 Milestone 1: REST API . . . 18
  - 3.1.1 Language Selection . . . 18
  - 3.1.2 Application Decomposition . . . 18
  - 3.1.3 Exchange Improvements . . . 19
  - 3.1.4 Investigating Python Web Frameworks . . . 20
  - 3.1.5 Building the REST API . . . 20
  - 3.1.6 AWS Elastic Beanstalk . . . 21
- 3.2 Milestone 2: FIX . . . 23
  - 3.2.1 Replacing RESTful HTTP . . . 23
  - 3.2.2 FIX Standards and Messages . . . 24
  - 3.2.3 QuickFIX . . . 25
  - 3.2.4 Implementing FIX . . . 27
  - 3.2.5 Exchange Improvements . . . 30
- 3.3 Milestone 3: Market Data . . . 33
  - 3.3.1 Addressing Methods . . . 33
  - 3.3.2 UDP Multicast . . . 34
  - 3.3.3 Market Data Publisher and Receiver . . . 34
  - 3.3.4 Adding the Simulation . . . 35
  - 3.3.5 Summary of Project Architecture . . . 38
- 3.4 Milestone 4: Cloud Deployment . . . 39
  - 3.4.1 Network and Compute Configuration . . . 39
  - 3.4.2 Deployment Challenges . . . 43

**4 Evaluation** **45**
4.1 Architecture Evaluation . . . 45
4.1.1 Language Selection . . . 45
4.1.2 Micro-service Selection . . . 46
4.1.3 Compromising Exchange Decomposition . . . 46
4.1.4 Omitting Persistent Storage . . . 47
4.2 Technologies Evaluation . . . 47
4.2.1 FIX . . . 47
4.2.2 UDP Unicast . . . 48
4.3 Race-to-Market Experiment . . . 50

**5 Conclusion** **53**
5.1 Contributions and Achievements . . . 53
5.2 Project Status . . . 54
5.3 Future Work . . . 54
5.4 Final Conclusions . . . 55

**A FIX Configuration** **58**
A.1 Example Exchange Configuration . . . 58
A.2 Example Client Configuration . . . 59

**B Example Simulation Configuration** **60**

# List of Figures


2.1 Supply and Demand Curves at Equilibrium, reproduced from De Luca et al. (2011) [12]. . 6
2.2 A graphical representation of a Limit Order Book (LOB), reproduced from Cliff (2018) [6]. 7
2.3 An updated graphical representation of a Limit Order Book (LOB) post trade, reproduced from Cliff (2018) [6]. . . . . 8
2.4 A graphical representation of the Jane Street Exchange. . . . . 14
2.5 AWS Global Infrastructure Map, reproduced from AWS (2019) [27]. . . . . 17

3.1 The Bristol Stock Exchange RESTful API deployed on AWS Elastic Beanstalk. . . . . 22
3.2 Example FIX tagvalue encoding . . . . . 25
3.3 Example FIX New Order Single message . . . . . 25
3.4 The QuickFIX Engine Web Interface displaying the SRVR engine's configured sessions. . . 27
3.5 The QuickFIX Engine Web Interface displaying the CLNT1 engine's configured sessions. . 27
3.6 The five Internet Protocol addressing methods, reproduced from [33]. . . . . 33
3.7 A graphical representation of the Distributed Bristol Stock Exchange. . . . . 38
3.8 A graphical representation of the Distributed Bristol Stock Exchange global network, reproduced from AWS (2019) [27]. . . . . 39
3.9 An example AWS configuration for a VPN Tunnel between two isolated networks, reproduced from AWS (2019) [28]. . . . . 40
3.10 VPC Peering Connection, reproduced from AWS (2019) [29]. . . . . 41
3.11 London VPC Routing Table. . . . . 41
3.12 Exchange Inbound Security Group Configuration. . . . . 42
3.13 Trading Client Inbound Security Group Configuration. . . . . 42
3.14 Distributed Bristol Stock Exchange AWS Configuration. . . . . 43

4.1 Spread of clients latency. . . . . 49
4.2 Graph showing the ratio of total profit per trader type for each client. . . . . 51
4.3 Graph showing the ratio of total profit across clients. . . . . 52

5.1 Distributed Bristol Stock Exchange proposed AWS architecture. . . . . 55

# List of Tables

3.1 API documentation outlining the available endpoints for the BSE RESTful API. . . . . 21

4.1 Table of results for the latency experiment. . . . . 49
4.2 Table showing the spread of the latency experiment. . . . . 50

# List of Listings


2.1 Example Trade Report from Bristol Stock Exchange. . . . . 9
2.2 Example Published LOB from the Bristol Stock Exchange. . . . . 9
2.3 Giveaway (GVWY) BSE Python implementation. . . . . 10
2.4 Zero-Intelligence-Constrained (ZIC) BSE Python implementation. . . . . 11
2.5 Shaver (SHVR) BSE Python implementation. . . . . 11
2.6 Sniper (SNPR) BSE Python implementation. . . . . 12
2.7 Zero-Intelligence-Plus (ZIP) BSE Python implementation. . . . . 12
3.1 Trading Client Place Order Implementation. . . . . 28
3.2 Trading Client Cancel Order Implementation. . . . . 28
3.3 Exchange Execution Report Implementation. . . . . 29
3.4 DBSE's matching function. . . . . 31
3.5 An example of Python typing. . . . . 32
3.6 Example DBSE trader configuration. . . . . 36
3.7 Example DBSE order scheduler configuration. . . . . 37

# Executive Summary

## Abstract

In 2001, a research group at IBM's TJ Watson Labs published results indicating that two types of autonomous, adaptive trading algorithms could consistently outperform human traders in a continuous double auction market [11]. Since then, the world's financial markets have been dominated by trading agents across the globe. However, designing and implementing new trading algorithms is a constantly difficult challenge caused by the financial and regulatory barriers in conducting experimental work on real financial exchanges. Consequently, many trading simulations have been developed to analyse the profitability of new algorithms before they are deployed in the "wild". Despite this, most trading simulations in research and business do not run in real-time and consequently cannot model latency - the speed in which a trader can react to a market event - between clients and exchanges. Modelling latency is an integral part of a trading agents design as well as critical in understanding many modern artefacts of real-world trading including "race-to-market" and "latency arbitrage".

This thesis presents a novel contribution to the field of latency-sensitive financial trading simulations by architecting and implementing a globally distributed limit order book financial exchange for research and teaching. By utilising real world communication technologies, including the Financial Information Exchange (FIX) protocol and the User Datagram Protocol (UDP), this work provides a platform to test the profitability of trading agents in real-time with real latency. Deployable onto global, commercial cloud computing services; exchanges and trading clients can be distributed on a planetary-scale. This work describes the implementation of the Distributed Bristol Stock Exchange (DBSE) and analyses the profitability of four different trading agents under latency-sensitive conditions.

## Summary of Achievements

- I present a novel contribution to the field of latency-sensitive financial trading simulations by redesigning the Bristol Stock Exchange into a real time platform that can be used to run latency-driven experiments for four unique trading agents: Giveaway, Zero-Intelligence Constrained, Shaver and Sniper.
- I spent weeks implementing three distinct Python applications comprising of over 2500 lines of source code: *bristol_stock_exchange*, *dbse_exchange* and *dbse_trading-client*.
- I researched and implemented real world financial communication networking protocols for order placement and market data publication.
- I designed and configured an assortment of Amazon Web Services compute and networking infrastructure to deploy this work across the three global regions: London; Ohio, US and Sydney, Australia.
- I conducted a race-to-market experiment and analysed the effects of latency on the profitability of trading agents. Moreover, this work presents the DBSE as a future latency arbitrage analysis platform, a current focus point in financial trading research.

# Supporting Technologies

Outlined below are the various third-party resources used throughout the project.

- The initial codebase and simulation structure was taken from Dave Cliff's Bristol Stock Exchange.
  `https://github.com/davecliff/BristolStockExchange`

- My investigatory work into a RESTful financial exchange utilised the Python Web Frameworks, Flask and Django.
  `http://flask.pocoo.org/`
  `https://www.djangoproject.com/`

- I used the QuickFix Python library to support my implementation of the Financial Information eXchange Protocol.
  `http://www.quickfixengine.org/`.

- I deployed and configured my application using various Amazon Web Service infrastructure including, Elastic Beanstalk, Elastic Compute Cloud (EC2), Simple Storage Service (S3) and Virtual Private Networks (VPC).
  `https://aws.amazon.com/`

# Chapter 1

# Introduction and Motivation

## 1.1 Algorithmic Trading

Since the dawn of the first financial exchange - the Amsterdam Stock Exchange in 1602 - until the introduction of computer aided trading in the 1980s, the buying and selling of financial products, such as shares and bonds were executed by the verbose shouting of highly-paid individuals on the floors of major financial exchanges. These interactions between people were slow, inefficient and often error prone. Consequently, at the birth of the Internet, the world's financial markets and institutions were one of the first industries to undergo a technological revolution. The buying and selling of financial products became a digital interaction and the traditional trading rooms were slowly closed throughout the world. These financial exchanges are highly sophisticated and complicated distributed systems that enable institutions, such as investment banks, hedge funds, brokers and insurance companies, to trade on the world's open markets. Until the 21st Century, most trades were still executed by humans, but as computer hardware capabilities improved, the financial markets industry underwent a second technological revolution - the introduction of "robot traders".

In the modern financial world, the majority of trades executed on the world's financial markets are by sophisticated autonomous adaptive computational systems. These electrical systems known as "trading agents", "algo traders" or "robot traders" can be responsible - at any one major investment bank - for the cumulative sum of $100Bn in trades in a single financial operating week. Prior to 2001, all research into the profitability of robotic trading had shown to be worse or at best equal to the probability of human traders. In 2001 however, a research group at the IBM's TJ Watson Labs published results from an experiment to test the effectiveness of two adaptive agents, known as ZIP and GD [11]. Using homogeneous populations of humans or agents, the research group discovered that these algorithms could consistently outperform a human trader with respect to efficiency and profitability. In the past decade, the rise of this technology has transformed the financial markets into a world of predominantly robotic traders. These agents are capable of processing vast amounts of data and can react to market changes at millisecond speeds. Consequently, trading agents are seen to be vastly superior to human traders as not only are they more profitable, but they are substantially cheaper. Rather than hiring lots of human traders, robotic agents can be deployed to multiple servers - costing solely the amount to maintain both software and hardware, excluding exchange fees. Many financial institutions and proprietary trading firms now focus solely on the research and development of more intelligent, fast and profitable agents through algorithm design and communication latency mitigation. The reason for this is if two competing, identical agents are listening to market events from the same exchange, the one which can receive and respond the quickest will be the most profitable, known as *race-to-market*. The industry in recent years has been focused on a battle to minimise latency between robotic agents and exchanges. This enables trades to be fulfilled electronically at super-human speeds, a practise known as *High Frequency Trading (HFT)*.

In conjunction to these developments in the real-world, academics have been exploring the profitability of both human and robotic traders with simulations and experiments for decades. In 2002, the Nobel Prize in Economics was awarded to Vernon Smith, for his ground-breaking work in establishing a new field of research now known as *Experimental Economics (ExpEcon)*. He demonstrated that the behaviour of human traders could be researched empirically under controlled and repeatable conditions. In 2012, Dave

Cliff developed the Bristol Stock Exchange (BSE) [5], an open-source minimal simulation tool for teaching and research into how trading algorithms operate within a controlled environment. Cliff's research based on Smith's empirical approach to human trading evaluation was used initially to systematically test the effectiveness of five publicly known trading algorithms: Giveaway, Zero-Intelligence-Constrained (ZIC), Shaver, Sniper and Zero-Intelligence-Plus (ZIP). Despite BSE's success as a teaching aid at the University of Bristol it has significant limitations compared to real-world distributed exchanges. The most notable of which, like many other financial exchange simulations, is that the Bristol Stock Exchange is not real-time and therefore critically assumes absolutely zero latency between trader and exchange. As previously noted, minimising the latency of robotic agents is an integral part of their design and thus simulating them without modelling latency is a serious omission in research. This thesis aims to create a financial exchange simulation that can be used with robotic agents in real-time with real latency. The outcome aims to be an open-source code-base that can be used as a foundation to explore many other key areas of academic research within Experimental Economics.

## 1.2 Why is this study important?

In the 21st Century, the world's financial markets are dominated by "robot" automated trading agents. This has created a substantial demand for computer scientists trained in algorithmic trading and distributed systems architectural design. It is a niche field within computer science that requires expert knowledge in multiple disciplines and as such has become one of the most lucrative industries for Computer Science graduates to enter out of university. Despite this, there are a limited number of UK universities that offer a specific course for computer scientists to learn about the technology used in finance. Moreover, there are limited open-source teaching resources or simulations for learning or testing algorithmic trading agents.

The fundamental challenge with testing robot traders is the question of where can you test them. Running a prototype algorithm on a real-world open exchange would be the most realistic, however at the risk of serious financial consequences if anything were to go wrong. Consequently, it is preferable to test new agents on controlled simulations that imitate the real markets. Many organisations choose to use playback simulations, a process of replaying legacy real market data and recording how agents react to the change in market events. Although real-time, these simulations are fundamentally flawed as the agent is unable to influence the change in market prices with its own activity - regardless of what the agent does, the prices on the market will remain the same over time. The alternative is to build your own market simulator, an example of which is the Bristol Stock Exchange. These simulators enable the user to configure how the price of a stock will change over time and critically enable the agent to influence the market prices. The disadvantage however is the trading agent is operating on substantially less and fabricated market data. Irrespective of playback or market, these simulations are often run on single machine and in the case of the Bristol Stock Exchange within the same application and thread. This results in zero latency between the traders and exchange as the agents continuously have perfect knowledge of the data on the exchange, thus limiting the realism of the simulation. As a result, there is a strong requirement for better simulations, both in the academic and professional world. This is motivated by two key artefacts in financial trading, flash crashes and latency arbitrage.

### 1.2.1 Flash Crash

The Flash Crash on the 6th May 2010 is a strong advocate for realistic testing infrastructure for algorithmic traders. Within a period of just 36 minutes, the Dow Jones Industrial Average saw the greatest one-day decline on record [23]. Following the crash, it was concluded that this chain-reaction of falling points was the direct cause of High Frequency Trading agents reacting to the market. This event caused widespread concern throughout the financial world and understanding and predicting these types of extreme events has since become a key area of research amongst trading firms and academics.

### 1.2.2 Latency Arbitrage

Latency Arbitrage is a profit earning technique, often used by high frequency trading firms, that involves exploiting a time disparity between the public price of a stock and the latest market update. High frequency trading firms pay very large amounts of money for their computer systems to be as physically close

to their target financial exchanges as possible and to have direct access to the market data publishing feeds. Latency arbitrage can be exploited when multiple exchanges are selling the same commodity, for example let's assume both the NASDAQ and the New York Stock Exchange (NYSE) are selling gold. At time, t=0, the price of gold throughout America is constant. This is governed by the Securities Information Processor (SIP), which links all U.S. markets by processing and consolidating all quotes and trades from every trading venue into a single data feed [10]. If a large trade was to occur on the NYSE, thus changing the price of gold on that exchange, market data would be published to both the SIP and along the direct data feeds to high frequency trading firms. Due to the SIP consolidating market data, it is comparatively slow compared to a high frequency trading firm. Within those fractions of a second, the price of gold on the NYSE is different to the NASDAQ. High frequency trading firms can react to this disparity in price and buy or sell gold accordingly between the NYSE and the NASDAQ before the NASDAQ even learns about the changed price from the SIP. Although a highly questioned practise, this is a technique trading firms use to make billions in profits annually. Most simulations, including the Bristol Stock Exchange, has no concept of latency and thus it is impossible to simulate this artefact. Understanding and developing techniques to minimise or exploit latency arbitrage, depending on your stance, is constantly being requested in industry. Despite this, there are limited research tools capable of simulating such a real-world artefact.

In this thesis, I describe the design and construction of a distributed financial exchange that can test the profitability of trading agents in real-time with real-latency. This thesis explores how real-world financial institutions and exchanges build their systems and communicate with each other across a distributed network for order placements, execution reports and market data publications at scale - all whilst minimising latency across the globe. Although originally based on the Bristol Stock Exchange, that code will be rewritten to provide more functionality to the executing agents, removing the limitation of a single order with maximum quantity of one as well as removing the assumption of zero-latency. With suitable network configuration, users of this simulation will be able to remotely host a financial exchange anywhere in the world and connect numerous trading agents in geographically local or remote regions with respect to the exchange. This thesis will demonstrate the capabilities of this simulation deployed on the cloud compute and networking infrastructure provided by Amazon Web Services. At the sole cost of cloud-based compute resources, this work will enable researching and teaching academics the ability to design and test new trading agents on a more realistic simulation that utilises the same communication technologies as real-world financial exchanges. Finally, I aim to ensure the project is designed and implemented with scalability in mind. There are vast opportunities available to extend the project and I aim for the code-base to be accessible to all of those who wish to improve and work with it. The goal is to enhance the research and teaching capabilities of the Bristol Stock Exchange and for it to become the first university based truly distributed financial exchange simulation. Moreover, this research is sort after in industry and provides an opportunity for a potential business proposal. Instead of financial institutions paying for the development of their own bespoke financial trading simulations, this platform could be packaged and sold globally as a profitable business.

## 1.3 Notable Challenges

The first major challenge I faced starting this project was my lack of knowledge in the financial technologies domain. I had not taken the Internet Economics and Financial Technologies unit at the University of Bristol and as such had no background knowledge starting this work. I had to learn the entire academic domain ab initio, including Smith's work on Experimental Economics, Cliff's work on the Bristol Stock Exchange to the design and deployment of complex modern distributed systems. I learnt how the trading algorithms, Giveaway, ZIC, Shaver, Sniper and ZIP operate and how to configure the BSE's order schedulers to manipulate the market when running simulations. This work was required prior to researching what is a highly competitive and secretive industry. It was extremely difficult to find information on the architectural designs of financial exchanges and even more challenging to know how trading firms get their agents to place orders and subscribe to market data publishers from the exchanges. My aim was to build a simulation that utilised real-world technologies and as such required a large amount of research and development that is not present in the final product.

The second major challenge was the testability of the simulation. When processing large quantities of market data, it is close to impossible to validate whether the simulation is accurate or not. This is because when orders are created by the simulation, the side, price and quantity of those order are all

sampled from varying random distributions. This results in no single execution of the simulation being the same; ideal for the requirements but challenging for testing. Throughout development, I continuously published verbose logs of all market activities both on the exchange and connected trading clients. I could have used automated testing for ensuring the exchange matches orders correctly, however this is not possible in determining whether the results of the simulation are as expected, due to the randomness. Throughout the project, I tested the simulation on relatively low quantities of market data and assumed that the exchange continues to operate as expected with increased traffic.

The final prominent challenge was deciding which areas of development to focus on whilst building the distributed financial exchange simulation. There is a notable time limitation for this thesis and it was paramount the project had a suitable conclusion. Financial exchanges alone are vastly complicated interconnected systems with many micro-services that enable their successful operation. For a single person to implement a real-world distributed financial exchange, excluding trading clients, would easily exceed the time available - nevertheless I wanted to build a full end-to-end simulation. As such, a large amount of the exchange's contingency and persistent storage had to be omitted from the work described here. Deciding the appropriate aspects of the exchange and trading clients to work on was constantly re-evaluated through the project and proved to be a continuous challenge.

## 1.4 Summary of Aims and Objectives

Listed below are the aims and objectives for this project:

1. Investigate the existing functionality of Cliff's work on the Bristol Stock Exchange and assess how it can be improved to assist real-time simulations.
2. Extend BSEs functionality to enable multiple orders per trader with no maximum quantity limit.
3. Develop a limit order book financial exchange application that can support multiple trading clients simultaneously.
4. Develop a trading client application that enables users to create different algorithmic trader and order scheduling configurations that can be run against the exchange.
5. Investigate and utilise the best communication protocol for clients to place orders and receive order updates from the exchange.
6. Investigate and implement how an exchange publishes market data quickly and efficiently to interested parties.
7. Deploy the simulation in the cloud with trading clients positioned in geographically disparate areas respective to the exchange, both local and remote.
8. Ensure the project is designed and implemented with scalability in mind so that it can subsequently be used by all who wish to improve and work with it.

# Chapter 2

# Technical Background

## 2.1 Auction Types

For thousands of years, humankind has traded goods by haggling for price. Whether you are trying to buy or sell a certain commodity, it is only natural that both sides aim to get the best possible price - never buying for more or selling for less than they can. In the economics world, this marketplace of buyers and sellers is known as an *auction*. The term auction is defined as the mechanism by which buyers and sellers come together to interact and thereby agree on a price for the exchange of an item or asset for money. Unknown to many, there are numerous types of auction each with different rules and regulations to facilitate trade.

**Posted Offer Auction** is the most common type of auction seen in the retail industry. Simply, the seller of a product or service *"posts"* and advertises their offer-price to potential buyers. If the buyer is interested, they can choose to take it for the advertised price; or leave it, whereby no transaction occurs.

**English Auction:** known in economics as the *ascending-bid* auction. It is the most famous type of auction and is what most people consider when describing a traditional auction house. Buyers gather and compete against each other, gradually increasing their bid-prices until only one buyer remains. Throughout this process, the seller remains silent and the trade occurs at the end of the bidding process with the buyer with the highest bid-price.

**Dutch Auction:** known as *descending-offer* is the reverse of an English auction and unsurprisingly - due to its name - common in the Netherlands. In a Dutch Auction the buyers remains silent at first, and waits for the seller to make an initial high offer-price. The auction proceeds by the seller gradually lowering their offer-price until the deal is taken by the first buyer to speak out.

**Continuous Double Auction:** is best explained as the amalgamation of both the ascending-bid (English) and descending-offer (Dutch) auctions. A Continuous Double Auction (CDA) allows buyers and sellers to place bids and offers respectively at any time. Simultaneously, any buyer can accept an offer from any seller and vice versa.

The Continuous Double Auction is the most prominent auction type in the world's financial exchanges; examples of which include the London Stock Exchange (LSE), the New York Stock Exchange (NYSE) and the National Association of Securities Dealers Automated Quotations (NASDAQ). Its major adoption is due to its asynchronous nature, requiring no centralized auctioneer and enabling buyers and sellers to continuously trade from anywhere in the world. A CDA is vastly superior to other auction types for two distinct reasons. Firstly, its asynchronicity maximises efficiency, as no trader is required to wait for another. Secondly, with just a small number of participants, the price at which transactions occur rapidly reaches stability. Conclusively, modelling and simulating such auctions has been a key area in academic research; founding the field of experimental economics.

## 2.2 Experimental Economics

The first breakthrough into studying the dynamics of Continuous Double Auction markets was published in the Journal of Political Economy by Vernon Smith in 1962 [30]. Smith explored many of the key aspects of real-world CDAs and developed an empirical approach to understanding them using human traders. The basis of his work involved creating an experiment from a room of students. He split the group of students evenly into a subset of buyers and a subset of sellers. Each student was handed a single card by Smith which had written on it a price. The value of this price was known only by the individual and represented the maximum price they were allowed to pay for an arbitrary commodity. Depending on whether the student was a buyer or a seller, they could not buy for more or sell for less than the price on the card, known as the *limit price*. Students were tasked with trying to make a profit; buyers were encouraged to bid for a price lower than their limit, whilst sellers asked for a price higher than their limit. The profit was calculated as the difference between the price the card was sold for and the limit price written on the card. Students could publicise and announce a price in which they would be willing to trade, known as a *quote*, at any point during the experiment. If two students, one buyer and one seller was to agree on this price, then a transaction would occur. Experiments were split into multiple subsections lasting a few minutes, known as *trading days*. At the end of a trading day, all students had to return their card to Smith irrespective of whether a trade had occurred - for each trade-less student their profit would be zero. Cards were then reallocated by Smith; a new trading day would begin and the experiment would continue.

Smith had designed the experiment in such a way that he was controlling the value of the commodity. This is because Smith was controlling the supply and demand schedules of the market by the limit prices he chose to write on the cards. The supply and demand schedules can be represented as functions of price and quantity. The market's supply schedule is a function that represents the total quantity of a commodity available for sale at varying prices. The higher the price, the more supply by the producers, but less demand from the buyers. Similarly, the market's demand schedule is a function that represents the total demand of a commodity desired to be bought at varying prices. Shown in Figure 2.1, the horizontal axis represents the quantity of the market, whilst the vertical axis represents price. The two order schedules, known as curves, represents the entire market. The supply curve is a positive function whilst the demand curve is a negative function; thus there exists an intersection, known as the equilibrium point ($P_0$, $Q_0$), where the price of the commodity would stabilise.

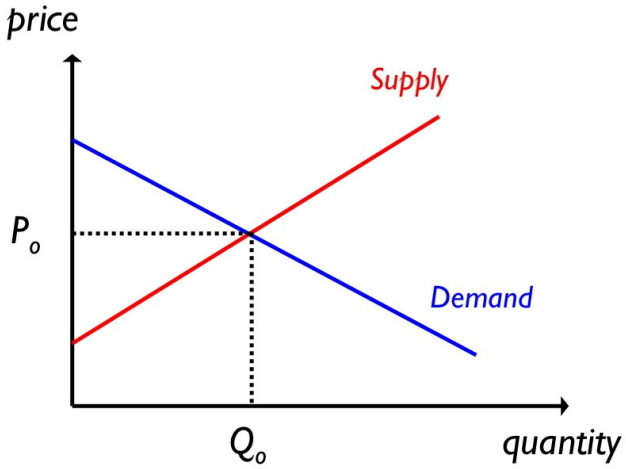



Figure 2.1: Supply and Demand Curves at Equilibrium, reproduced from De Luca et al. (2011) [12].

One key aspect of competitive markets is that they hold the ability to self-equilibrate. The price of a commodity constantly varies over time, caused by the varying supply and demand curves. Regardless of whether you are trying to buy or sell, you aim to get the best possible price but fundamentally desire to secure a deal. Self-equilibrium can be described through the buying and selling of wheat. Simplified, the price of wheat is a balance between the success of the harvest, the supply, and the size of the population, the demand. If we assume that the demand is constant, the price of wheat will predominately be decided

by the success of the harvest. On a good harvest, there will be an influx in supply and thus rivalling farmers will lower their prices to compete and sell their own produce. On a bad harvest, the demand for food respective of the supply will drive prices up. Regardless, for a deal to be made, the buyer and seller will need to consider being flexible with their prices, known as the *bid-price* and *ask-price* respectively. For a trade to occur, buyers and sellers must repeatedly revise their bids and asks, bringing them closer together until a deal can be made. This fundamentally causes the market to self-equilibrate and reach the equilibrium point ($P_0$, $Q_0$) shown in Figure 2.1.

Smith's work demonstrated that market equilibrium could be realised within his human trader CDA experiment. Today, financial exchanges are monolithic digital applications that accept and manage orders in a Continuous Double Auction. These orders can be categorised by two types, *limit orders* and *market orders*. A market order is the conventional type when people describe financial trading; when you place a request to the exchange to buy or sell a commodity, the transaction price is the current live price on the exchange. A limit order on the other hand, is a request to buy or sell a commodity at a limited price. Depending on whether you are buying or selling, this limit price can be lower or greater than the current live price on the exchange respectively. These limit orders, the requests to buy or sell at a given price, builds the supply and demand curves on the exchange. To present this information in a structured format to clients, modern exchanges use a data structure known as the *Limit Order Book (LOB)*.

## 2.3 Limit Order Book

The Limit Order Book (LOB) is a standardised view of all market data for a single commodity within a Continuous Double Auction market. The LOB summarises all the live limit orders currently on the exchange and displays them in a two-sided view - one side for bids and one side for asks. Each side of the LOB is known as a book and hence a LOB has both a bid book and an ask book. The bid book lists the prices and respective quantities available as a key/value pair for all buy orders in descending order, such that the bid with the highest price is first. Conversely, the ask book lists the same price/quantity pair for all sell orders in ascending order, such that the lowest price is first. The best (highest) price for bids and the best (lowest) price for asks is known as the *best bid* and *best ask* respectively.

On the left-hand side of Figure 2.2, you can see an example of the bid and ask books coloured red and blue respectively. On the bid side there are currently 5 different available prices, \$1.50, \$1.28 \$1.10, \$0.75 and \$0.50 with a total quantity of 58. On the ask side there are currently 3 different available prices, \$1.77, \$1.86 and \$2.15 with a total quantity of 43. The current best-bid of \$1.50 and best-ask of \$1.77 are both positioned at the top of their corresponding books with an available quantity of 10 and 13 respectively. It is important to note that the LOB is almost always in equilibrium, as the best-bid is not greater than the best-ask and thus no participant is happy to trade given the current prices on the exchange.


Figure 2.2: A graphical representation of a Limit Order Book (LOB), reproduced from Cliff (2018) [6].

The difference in price between the best bid and the best ask is known as the *bid-ask spread*, or simply *spread*. The spread can be used to calculate two approximations for the value of the market, the *midprice* and the *microprice*. The midprice is calculated by simply taking the arithmetic mean of the best bid and the best ask, calculated as $(\$1.50 + \$1.77)/2 = \$1.635$ in the example above. The microprice is more

sophisticated however, as it considers the opposite quantities of the best bid and the best ask, calculated as

$$\frac{BestBidPrice * BestAskQty + BestAskPrice * BestBidQty}{BestBidQty + BestAskQty} = Microprice \tag{2.1}$$

$$\frac{\$1.50 * 13 + \$1.77 * 10}{10 + 23} = \$1.62 \tag{2.2}$$

and shown in Figure 2.2 as the green text, prefaced with *"M:"*, in the top left corner. The red text, prefaced with *"T:"*, above the microprice is the elapsed time into the market session.

The right-hand side of the LOB displays the bid and ask books as their respective supply and demand curves. Tracing either the bid (red) or ask (blue) curves from left to right; the height represents the price and the width denotes the quantity available at that price point - tailing until the quantity is exhausted. The green cross represents the microprice in the spread. Finally, beneath the bid and ask books on the left-hand side is the *tape*. The tape is a history, a list, of executed transactions since the start of the market session. Currently the tape states that a single execution occurred at time 00:27, with price \$1.50 and quantity 10. Whenever a new trade occurs on the exchange, its transaction data is appended to the tape.

A trade occurs on the exchange when a new order's price exceeds the current best bid or ask price on the LOB, known as *crossing the spread*. Concretely, a trade occurs if a bid order's price is greater than the best ask price or a ask order's price is less than the best bid price. This is irrespective of quantity, if only some of the order can be filled, then the remaining quantity is left on the LOB for future trades to occur. Shown below is an updated view of the LOB above after a new ask order of price \$1.48 and quantity 5 arrived at the exchange at time 00:30, the details of which are shown in the yellow box above the books in Figure 2.3.


Figure 2.3: An updated graphical representation of a Limit Order Book (LOB) post trade, reproduced from Cliff (2018) [6].

When the new order arrived at the exchange a transaction occurs immediately. This is because the order's price of \$1.48 was less than the best bid price of \$1.50, hence had crossed the spread. As a result, the quantity of the best bid on the LOB has decreased by the quantity of the new order, hence from 10 to 5. Hypothetically, if the order quantity had been greater than 10, for example 30, then the remaining order quantity of 20 would be placed on the ask book at a price of \$1.48. As a result of this trade you can also see that new transaction data has been appended to the tape. It is worth noting that the transaction price was \$1.50 and not the order price of \$1.48. This is due to the fact that older orders on the LOB take priority with respect to price. Understanding the order execution flow of a Continuous Double Auction and how the view and structure of a Limit Order Book updates is crucial in understanding how markets operate and lays the foundations for designing a distributed financial exchange.

## 2.4 Bristol Stock Exchange

First released in 2012, the Bristol Stock Exchange [4], downloadable at `https://github.com/davecliff/BristolStockExchange`, is a open-source minimal simulation of a Continuous Double Auction financial

exchange running a Limit Order Book for a single commodity. Written by Dave Cliff as a teaching aid for the Internet Economics and Financial Technologies Unit at the University of Bristol, it now also serves as a research platform for students and academics to understanding algorithmic trading. The simulation is a single-threaded application that incorporates an exchange and an implementation of five different trading algorithms. Written in Python, the simulation is not computationally efficient and has four major simplifications, summarised as follows:

1. only one financial commodity being traded
2. orders have a maximum quantity of 1
3. each trader can have at most one order on the exchange.
4. each order is processed in sequence and republishes LOB to all traders, therefore is not real-time and does not model real-latency.

The Bristol Stock Exchange was written to be as simple as possible and thus the entire code-base lies in a single file named *BSE.py*. It utilises very few external libraries, only *sys*, *math* and *random*, and as such is a lightweight application executable on any Python 2.7 interpreter. It is designed to be run in batch-mode and writes data to .csv files for further analysis post execution. Despite its aim to be readable, due to the complexity of the domain, the application is over 1300 lines long and can be confusing for new readers. This can be contributed to the amalgamation of both the buy side and sell side services being present within the same file. These services can be separated into three distinct categories, the exchange, the trading robots and the simulation.

### 2.4.1 Exchange

The exchange is the heart of a financial market. Accepting, amending and cancelling orders, returning trade notifications and publishing market data are just a few of the many responsibilities that an exchange undertakes. In the Bristol Stock Exchange however, these responsibilities are majorly simplified. The exchange can only trade in one financial instrument and as such the limit order book is ingrained at the forefront of the exchanges code. To add, amend or cancel orders the traders can call functions on the exchange directly without any communication overhead. When a trade occurs, the exchange responds sequentially by returning a Python object, shown below.

```
{
    'type': "Trade",
    'time': 30.0,
    'price': 150,
    'party1': "B01",
    'party2': "S02",
    'qty': 5
}
```

Listing 2.1: Example Trade Report from Bristol Stock Exchange.

This trade report is known as an *execution report* in a real-world financial exchange. In BSE's simplification it contains the time of execution, the price and quantity of the trade as well as the ids of the buyer and the seller. Listing 2.1 gives the example response for the trade described in Figure 2.3.

When a trade occurs, it is the responsibility of the exchange to publish an anoymised version of that trade as market data to all other traders who might be interested in that market. The Bristol Stock Exchange does that by giving a new copy of the LOB whenever a trader is given the opportunity to respond to the market, explained in more detail in Section 2.4.3. Using the previous example in Figure 2.3, the BSE would publish a Python object shown in Listing 2.2.

```
{
    'time' = 30.0,
    'bids' = {
        'best': 150,
        'worst': 50,
```

```
            'n': 5,
            'lob': [(150, 5), (128, 3), (110, 25), (75, 8), (50, 12)]]
        },
        'asks': {
            'best': 177,
            'worst': 215,
            'n': 3,
            'lob': [[(177, 13), (186, 13), (215, 17)]
        }
    }
```

Listing 2.2: Example Published LOB from the Bristol Stock Exchange.

### 2.4.2 Trading Robots

Although the focus of this thesis is not the technical specifics of the different trading algorithms that the Bristol Stock Exchange simulates, it is important to understand them in the context of what data they require, so that the distributed financial exchange fulfils their needs. Furthermore, in the context of latency arbitrage it is important to consider how much compute power they require to determine if one would outperform the other in efficiency and respond to the market changes quicker once latency is a legitimate factor in the simulation. For each of the five trading algorithms the Bristol Stock Exchange currently implements; outlined below is the implementation of the `getorder(self, time, countdown, lob)` function, which dictates what the trader will do with their limit order when given the opportunity to respond to the market.

#### Giveaway (GVWY)

The simplest and least intelligent of the five algorithms, Giveaway does nothing other than the name suggests, it gives away the customer's order at its limit price - irrespective of the state of the market. As such, the Giveaway algorithm requires no current market data and simply generates and places a new order on the exchange equivalent to the customer's limit order, shown below.

```
def getorder(self, time, countdown, lob):
    if len(self.orders) < 1:
        order = None
    else:
        limit_order = self.orders[0]
        quote_price = limit_order.price

        order = Order(
            self.tid,
            limit_order.otype,
            quote_price,
            limit_order.qty,
            time
        )

    return order
```

Listing 2.3: Giveaway (GVWY) BSE Python implementation.

#### Zero-Intelligence-Constrained (ZIC)

The ZIC algorithm, introduced by Gode and Sunder [16], is the first of the algorithms to require the current limit order book market data. As the name suggests, it has no real intelligence and decides the price of new quotes by choosing a random number between the LOB's worst price and the customer's limit order price. If the limit order is a bid, the new price will range between the bid-side's worst price and the order's limit price. If the limit order is an ask, the new price will range between the order's limit price and the ask-side's worst price, shown below. Simply, the limit order acts as a constraint on the new quote ensuring it does not lead to a loss-making deal.

```
def getorder(self, time, countdown, lob):
    if len(self.orders) < 1:
        order = None
    else:
        limit_order = self.orders[0]
        minprice = lob['bids']['worst']
        maxprice = lob['asks']['worst']

        if limit_order.otype == 'Bid':
            quoteprice = random.randint(minprice, limit_order.price)
        else:
            quoteprice = random.randint(limit_order.price, maxprice)

        order = Order(
            self.tid,
            limit_order.otype,
            quoteprice,
            limit_order.qty,
            time
        )

    return order
```

Listing 2.4: Zero-Intelligence-Constrained (ZIC) BSE Python implementation.

### Shaver (SHVR)

The Shaver algorithm aims to constantly have the best bid or ask on the LOB. It does this by "shaving" a penny off the best price - provided it does not exceed the constraints of the customer's limit order. As such it too requires knowledge of the LOB only.

```
def getorder(self, time, countdown, lob):
    if len(self.orders) < 1:
        order = None
    else:
        limit_order = self.orders[0]

        if limit_order.otype == 'Bid':
            if lob['bids']['n'] > 0:
                quote_price = lob['bids']['best'] + 1
                if quote_price > limit_order.price:
                    quote_price = limit_order.price
            else:
                quote_price = lob['bids']['worst']
        else:
            if lob['asks']['n'] > 0:
                quote_price = lob['asks']['best'] - 1
                if quote_price < limit_order.price:
                    quote_price = limit_order.price
            else:
                quote_price = lob['asks']['worst']

        order = Order(
            self.tid,
            limit_order.otype,
            quote_price,
            limit_order.qty,
            time
        )

    return order
```

Listing 2.5: Shaver (SHVR) BSE Python implementation.

### Sniper (SNPR)

The Sniper algorithm is an extension to Shaver that understands the concept of urgency. At the end of each market session, the markets close and thus any orders still on the exchange are unable to be traded. In a famous public contest at the Santa Fe Institute, Todd Kaplan demonstrated an algorithm that lurked in the market until just before the closing, where it became very aggressive [22]. This algorithm, now known as Kaplan's Sniper, demonstrated that it is better to "steal a deal" than trade throughout the session or have outstanding orders on the exchange after closing. In the Bristol Stock Exchange, the simplified Sniper implementation uses the countdown parameter - a percentage of the time remaining in the current market session. As the percentage of time remaining decreases, the algorithm is willing to shave larger and larger margins off the LOB's best price, shown below.

```
def getorder(self, time, countdown, lob):
    lurk_threshold = 0.2
    shavegrowthrate = 3
    shave = 1.0/(0.01 + countdown/(shavegrowthrate*lurk_threshold))

    if (len(self.orders) < 1) or (countdown > lurk_threshold):
        order = None
    else:
        limit_order = self.orders[0]

        if limit_order.otype == 'Bid':
            if lob['bids']['n'] > 0:
                quote_price = lob['bids']['best'] + shave
                if quote_price > limit_order.price:
                    quote_price = limit_order.price
            else:
                quote_price = lob['bids']['worst']
        else:
            if lob['asks']['n'] > 0:
                quote_price = lob['asks']['best'] - shave
                if quote_price < limit_order.price:
                    quote_price = limit_order.price
            else:
                quote_price = lob['asks']['worst']

        order = Order(
            self.tid,
            limit_order.otype,
            quote_price,
            limit_order.qty,
            time
        )

    return order
```

Listing 2.6: Sniper (SNPR) BSE Python implementation.

### Zero-Intelligence-Plus (ZIP)

The Zero-Intelligence-Plus algorithm was first invented by Cliff in 1996 [3]. It is an example of an adaptive trading algorithm that parametrises the current state of the LOB over time by responding to market events and updating its internal variables. The new quote price is calculated my multiplying the customer's limit order price by 1 plus a margin, shown below. For ask orders, the margin is a positive float, while for bid orders the margin is negative. The value of this margin is constantly recalculated by the varying events in the market. Describing the full implementation of the ZIP algorithm is beyond the scope of this thesis, it is worth noting that this algorithm is certainly the most involved and as such has the largest computational requirements.

```
def getorder(self, time, countdown, lob):
```

```
    if len(self.orders) < 1:
        self.active = False
        order = None
    else:
        limit_order = self.orders[0]
        self.active = True
        self.limit = limit_order.price
        self.job = limit_order.otype

        if self.job == 'Bid':
            # currently a buyer (working a bid order)
            self.margin = self.margin_buy
        else:
            # currently a seller (working a sell order)
            self.margin = self.margin_sell

        quote_price = int(self.limit * (1 + self.margin))
        self.price = quote_price
        order = Order(
            self.tid,
            self.job,
            quote_price,
            limit_order.qty,
            time
        )

    return order
```

Listing 2.7: Zero-Intelligence-Plus (ZIP) BSE Python implementation.

### 2.4.3 Simulation

The final aspect of the Bristol Stock Exchange's codebase builds and runs the simulation. It configures the number of each trader by type and defines the rate in which customer orders will be created and distributed to those traders. At this point the market session begins and proceeds sequentially. The simulation is not real-time, instead uses an abstract variable, a float, where the value 1.0 represents one second. The exchange then creates an atomic "time-interval" by subdividing that "second" by the number of traders. For example, if the simulation is configured to run with 40 traders, the time-interval is calculated as $1.0/40 = 0.025$ and used to iterate a while loop. This gives all traders the opportunity to place an order to the exchange within an abstract second. At the end of the second, the exchange responds to all orders placed and publishes market data to all traders. This loop iterates until the end of the market session, defined as a float, usually 300.0 or 600.0 seconds.

At the end of the market session the Bristol Stock Exchange publishes the cumulative sum of the balances of every trader by their type along with the number of trades, as well as the transactions executed during the session. This data is stored in two comma separated list files, *avg_ balance.csv* and *transactions.csv*. With relatively small code changes, market sessions can be repeated multiple times with constant or varying configurations to calculate an overview and summarise the performance of each trader type. This adaptation of the simulation's configuration is up to the user's consideration but requires a knowledge of Python programming to do so.

## 2.5 Case Study: Jane Street Exchange (JX)

The Bristol Stock Exchange is a simulation, it is an abstract and simplified model of a CDA financial exchange. To assist in my understand of real world exchanges it was important to research the ideal approach. On the 2nd February 2017, Jane Street - a global liquidly provider and market making company - gave a tech talk outlying the high-level overview of the design and development of their own financial exchange known as JX [32]. Jane Street develops their own proprietary models and use quantitative analysis to trade over $13 billion in equities worldwide in a single day. The motivation for their development of JX was due to their necessity to test new algorithms and financial models - much

like the motivation for this thesis. The JX exchange is based on the design of the American NASDAQ exchange and is the perfect exemplar for a real-world distributed limit order book financial exchange. It has been designed to achieve many of the underlying requirements of a modern exchange, including high transaction rates, low, deterministic response times, fairness and reliability. It has been reported to handle messages rates in the 500k/second range with latencies in the single-digit microseconds. Outlined below I aim to explain the roles and responsibilities of the individual micro-services on the exchanges private network and outline the key technologies that makes these staggering performance figures possible.

### 2.5.1 Exchange Components

Figure 2.4 outlines the key components of the JX distributed limit order book financial exchange. The network backbone is represented by the thick purple line; any component above this line has no direct communication with the outside world and can be assumed to sit within a private subnet of the network. All components below the network backbone are public facing and can connect to external clients either via the Internet or a paid private connection, at the exchange owner's discretion.

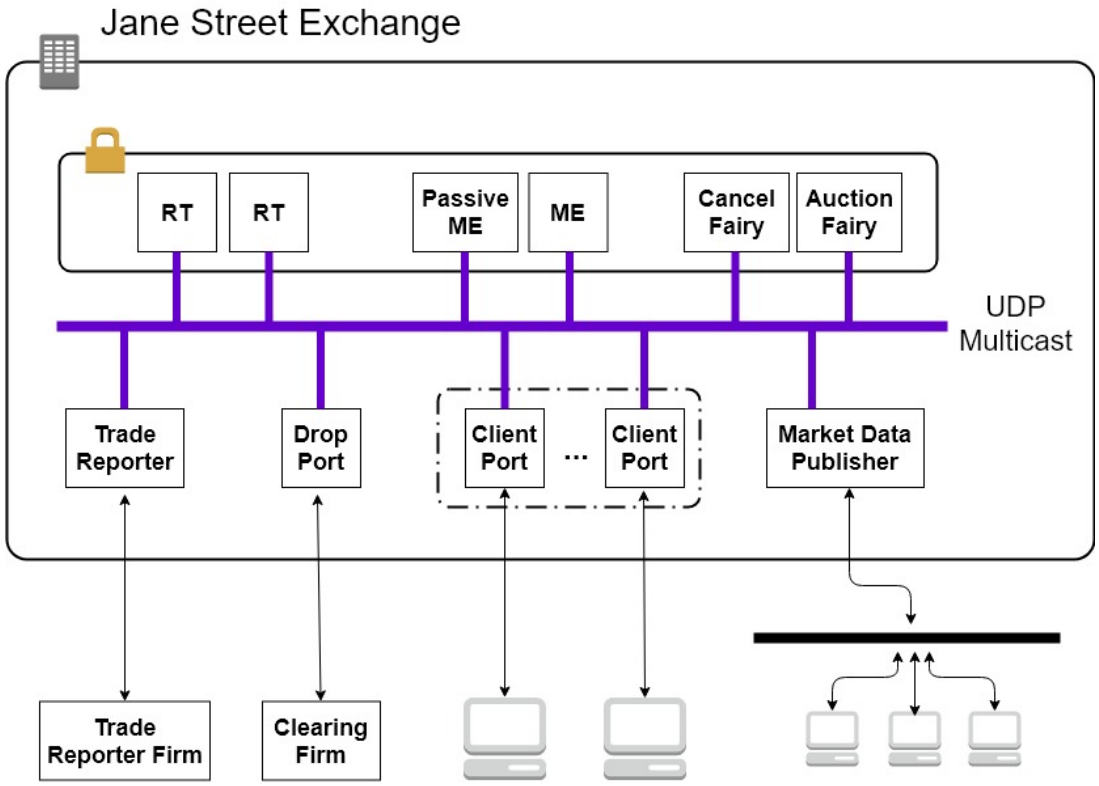



Figure 2.4: A graphical representation of the Jane Street Exchange.

#### Matching Engine (ME)

The *matching engine*, depicted ME is the heart of the exchange. It is a single, monolithic machine that holds all the current orders on the exchange in a Limit Order Book data structure. These orders are kept in memory and the matching engine is responsible for the "matching" of orders and the execution of all transactions on the exchange. The matching engine sits within a strictly managed private network and receives and publishes message to a wide variety of other services on the client side of the network. Adjacent to the matching engine is a passive copy that listens to all output of the active matching engine. The passive ME is used as a fail-safe if the active engine has a system failure and goes offline, at which point the passive engine immediately takes control. In real-world exchanges this process is close to instantaneous to limit impact to clients and the financial world.

#### Cancel & Auction Fairy

As previously stated, the matching engine is the critical component of a financial exchange and must deal with thousands of messages a second. A large proportion of these messages however are not relevant to the live activities of the exchange; an example of which is the cancel order request message. At the end of a market session, it is common for clients to cancel all their live orders. Clients would send cancel order requests throughout the trading day but set a delay so that it is only executed at market close. Historically, it was the responsibility of the matching engine to keep track of all these delayed cancel order requests. This however added noise to the matching engine, and thus modern exchanges offload this responsibility to a independent application, known as the *cancel fairy*. When a delayed cancel order request hits the matching engine, it would be acknowledged by the engine and then picked up the auction fairy. Once it is time for the order to be legitimately cancelled on the exchange, the auction fairy would send a new cancel request that would be executed immediately by the matching engine.

In other non-continuous exchanges, there exists an additional process known as the *auction fairy*. It is used to aggregate many orders, often with overlapping prices, and runs an optimisation to find a price that maximises the shares traded. This process takes time and so is run independently. Once complete the results are returned to the matching engine; another example of how modern distributed exchanges strive to limit the work done by the matching engine.

#### Re-transmitter (RT)

In order to make communication across the network efficient and fair, all components communicate with each other via a technology known as UDP Multicast, explained in detail in Section 2.5.2. Critically this technology does not include guaranteed message delivery or acknowledgement and as such messages can be lost during transmission. To account for this, a series of processes known as *re-transmitters* are added to the network. Their sole purpose it to record all the messages that have been seen on the network and to resend any message that was lost by a micro-service. Multiple re-transmitters are used in the event that one of them did not receive a message and thus they can communicate and reach consensus about the current state of the messages transmitted. In the unlikely event that all of the re-transmitters did not receive a message, they can request it from the matching engine, which maintains an in-memory copy of all messages sent in the current market day.

#### Client Port

The *client port* is the main connection point for external financial institutions, such as brokers and investment banks wanting to trade on the exchange. The client ports accept connections to individual clients and provide a mechanism for them to perform transactions with the matching engine. This includes placing, amending and cancelling orders, requesting quotes and receiving execution reports on any trades that resulted from that client's orders.

#### Drop Port

The *drop port* is very similar to a client port, however accepts connections from institutions known as clearing firms instead of clients. When a trade occurs, it is the responsibility of a third party to "clear" and manage the transmission of money between the trading parties. This is the responsibility of a clearing firm and thus they require information about the activities of both clients. As such, when a trade occurs and the execution reports are sent to the corresponding client ports; both execution reports are sent to and aggregated by the drop port for the clearing firm.

#### Trade Reporter

The *trade reporter* is the public facing data feed for all trade activities on the exchange. It listens to all trades that occurred on the matching engine, anonymises the data and publishes it to an external trade reporting firm.

#### Market Data Publisher

Similar to a trade reporter, a *market data publisher* listens to and anonymises all market data on the matching engine. Instead of broadcasting to an external trade reporting facility however, the market

data publisher uses UDP multicast technology to transmit the data to a network of clients, both human and robotic. Access to these market data publishers are very expensive and is how many high frequency trading firms take advantage of latency arbitrage - knowing the exchange's activity before the public trade reporting firm does.

### 2.5.2 Key Network Technologies

To build a distributed financial exchange, it is essential to have a technical background of how data is sent over a network. Concretely, data is split into bits, known as packets, which are then wrapped in layers of different communication protocols. The network layer implements the Internet Protocol (IP) which provides an end-to-end logical addressing system to route packets to a specified target address. The transport layer builds on top of the network layer and handles the segmentation of data into its corresponding packets. Although there are many transport layer protocols, the Jane Street Exchange uses the two most common; the Transport Control Protocol and the User Datagram Protocol.

#### Transport Control Protocol (TCP)

The Transport Control Protocol or TCP is a connection-oriented transport protocol and the most widely used throughout the Internet and the world. Its popularity comes from its ability to create a bi-direction communication channel between two application programs, which is only released once each end has finished exchanging messages. Furthermore, TCP provides assurance of message delivery by acknowledging every packet via a sequence number and retransmitting any that may have been lost during transport. Many web-based protocols, including HTTP, SMTP and FTP, are all based on the transport control protocol and in the case of JX it is how client ports communicate with connecting clients.

#### User Datagram Protocol (UDP)

The User Datagram Protocol or UDP is the fundamental communication technology that enables the efficient and fair transmission of messages between the different micro-services on the exchange's internal network. Although a renowned communication protocol, it is used very little compared to TCP. This is because UDP is connection-less and as such there is no guarantee that messages will be delivered successfully. The advantage however is that the sending application does not have to wait for an acknowledgement of packet delivery by the receiver - vastly improving network performance. The Market Data Publisher also utilises UDP in order to publish market data to clients, denoted by the thick black line in Figure 2.4. This choice is driven by the need for the exchange to distribute data fairly amongst many clients. Despite this, "fairness", clients such as High Frequency Trading firms pay vast quantities of money to have their servers geographically located close to the market data publisher. By positioning themselves close to the Market Data Publisher, due to the laws of physics, they will receive messages faster than those positioned further away. Consequently, High Frequency Trading firms can respond to market events quicker, leading to latency arbitrage.

## 2.6 Cloud Computing

Cloud Computing can be described as a service for the delivery of software and computing resources over a network, most often the Internet. Cloud Computing can be defined by five fundamental characteristics:

**On-demand Self-service** - a consumer can unilaterally provision resources as needed automatically without requiring human interaction.

**Broad Network Access** - resources are available over the network and accessed through standard mechanisms.

**Resource-pooling** - resources are pooled to serve multiple consumers. Both physical and virtual resources can be dynamically reassigned according to demand.

**Rapid Elasticity** - resources can be elastically provisioned and released, in many cases automatically, to scale with demand.

**Measured Service** - cloud systems automatically control and optimize resources use by measuring their usage.

Cloud Computing is advantageous over traditional on-site hardware provisioning as customers can have access to large-scale computing power without needing large capital expenditure to buy that hardware in advanced. Furthermore, due to the Economics of Scale, Cloud Computing providers can offer their services for fractions of the cost compared to configuring it on-site. In the 21st Century it has revolutionised the way computing resources are provisioned and paid for, with the industry now being reported a staggering $130bn worldwide [31]. Consequently, most the world's big technology firms, including Google, Microsoft, IBM and Oracle now provide Cloud Computing services. Arguably the leader in this field however is Amazon; which was the first to market, launching Amazon Web Services publicly in 2006.

### 2.6.1 Amazon Web Services

Amazon Web Service (AWS) is the longest running commercial cloud platform and is the market leader in Cloud Computing infrastructure. In Q4 2018, Amazon held a 32.3% market share of the worldwide cloud infrastructure and supports many of the world's largest technology firms including Apple and Netflix [1]. AWS provides a plethora of scalable and interoperable services ranging from compute and database hardware to machine learning and big data platform services, available throughout the globe. At the time of writing, the Amazon network, known as *Amazon Cloud*, spans 64 Availability Zones within 21 geographic Regions around the world, with announced plans for 12 more Availability Zones and four more Regions in Bahrain, Cape Town, Jakarta, and Milan [27]. All of these regions, shown in Figure 2.5, are connected via Amazon's high speed network cables enabling a truly global distributed network.


Figure 2.5: AWS Global Infrastructure Map, reproduced from AWS (2019) [27].

# Chapter 3

# Project Execution

## 3.1 Milestone 1: REST API

Prior to discovering the Jane Street Exchange case study, I made the assumption that distributed financial exchanges communicated to clients via the Hypertext Transfer Protocol (HTTP) - the underlying protocol used by the World Wide Web. The HTTP protocol implements TCP and defines how messages are formatted and transmitted between clients and servers across the world. Most web-based services including Amazon Prime, Facebook and GMail all utilise HTTP by implementing *Representational State Transfer* (RESTful) *Application Programming Interfaces* (API). RESTful APIs are web-based applications that define a set of subroutines and enable interoperability of computer systems across the Internet. These subroutines define the business logic of the application and thus could implement the exchange's functionality, such as Publish LOB, New Order and Cancel Order. As such I began understanding the existing Bristol Stock Exchange codebase and started learning how best to implement it as a RESTful API.

### 3.1.1 Language Selection

The first fundamental decision I had to make was that of language selection. The existing Bristol Stock Exchange was implemented in Python; a language renowned for being highly inefficient with regards to memory usage and computational performance. Cliff's original decision in choosing Python was due to the language's readability and simplicity. Python is a popular language for new programmers as it is written much like pseudocode and thus reads similarly to English. Since the Bristol Stock Exchange was designed as a research and teaching platform, it fundamentally did not require the low-level memory management and performance provided by other languages such as C or C++. In the real-world, this performance is critical and thus most modern exchanges are written in the object-orientated and strictly typed language C++. As language design is improving however, new exchanges - including the Jane Street Exchange - are deciding to adopt functional languages such as OCaml for their implementation. Since this thesis's aim is to maintain the BSE as a research and teaching platform, I too concluded that it would be unnecessary to re-write the application in a more performant language. Furthermore, this ensures the thesis focuses on its objective to build a distributed application, rather than adding additional implementation work re-factoring the existing code-base to be more efficient. Consequently, I decided to continue using Python as the predominant language for this implementation but took this as an opportunity to upgrade it to its latest version, Python 3.7.

### 3.1.2 Application Decomposition

As discussed in Section 2.4, the Bristol Stock Exchange can be categorised as having three distinct parts, the exchange, the trading robots, and the simulation. In a distributed system, the exchange is an independent entity from its clients and as such I decided to decompose and restructure the Bristol Stock Exchange's code-base. Focusing solely on the matching engine functionality, I created a new GitHub repository entitled *bristol_stock_exchange*. This repository would form the basis of the HTTP-based distributed exchange and initially housed the matching functionality, including the limit order book data structure, from the existing Bristol Stock Exchange. In the process of decomposing the code-base I re-structured all of the code into independent classes. Whereas previously all data structures and

functionality had existed in a single file, it was now separated into multiples files and folders in a project like structure. Distinct data structures, including Orders and the Limit Order Book, now had their classes in independent files to ensure correct namespacing throughout the project. Rather than these classes being global, they could now be imported specifically into the places they are required. This improves the readability of the application and protects it from implementation error, such as accidental referencing.

### 3.1.3 Exchange Improvements

Once the exchange functionality from the Bristol Stock Exchange had been restructured into my own repository I wanted to take on the task of removing some of the critical simplifications in the application. As discussed in Section 2.4, Cliff's existing simulation could only allow orders with a maximum quantity of 1 and each trader could have at most one order on the exchange at any given time. Choosing to improve the matching engine by removing these restrictions not only makes the simulation more realistic but also gave me an opportunity to demonstrate my understanding of the financial domain.

#### Order Quantity

Using inspiration from a blog post, entitled "Simulating a financial exchange in Scala" [2], I redesigned the matching function of the Bristol Stock Exchange to recursively check whether more than one order on the LOB could be filled when a new order is placed on the exchange. For example, assume that there already exists two ask orders on the LOB, one with price $1.20 and quantity 10, the other with price $1.30 and quantity 5. N.B: For simplicity I will depict these orders as [ASK P=120 Q=10] and [ASK P=130 Q=5] respectively, where the price "P" is represented in cents, and the quantity "Q" an integer value. If a new bid order, [BID P=150 Q=12], was to be added to the LOB then those asks would need to be filled and partially-filled respectively. This is because the bid price exceeds both asks, but the bid quantity does not fill the cumulative quantity of the asks. The former of the asks is processed first as its price takes priority - being the cheapest on the ask book. Its quantity of 10 is completely filled, leaving the bid with a remaining quantity of 2. Since the bid is still unfilled, and there is an outstanding ask with a price that crosses the spread, that too must be processed. However, since the ask has a quantity greater than the remaining bid quantity, [BID P=150, Q=2], the ask is partially-filled and remains on the LOB with quantity 3. Both trades would then need to be added to the tape simultaneously. This process is known as *walking the book*. Previously, the Bristol Stock Exchange did not implement any of this functionality as it never needed to check below the best bid or ask on the LOB. This is because all orders were assumed to be of size 1, and thus any order that crossed the spread would simply fill from the top of the book - deleting that order at index 0. Implementing this functionality required a lot of testing as there are a large number of specific situations to check correct operation in. I had to test with numerous varying orders on the LOB and for each configuration placing new orders with quantities less, greater or equal to the cumulative quantity of the orders on the opposite side of the LOB to the new order. The result however was extremely satisfying as this functionality enabled any single large order to have a substantial influence on the state of the LOB, and thus the mid-price and micro-price of the market.

#### Exchange Responses

At the same time as I was improving the matching functionality, I redesigned the way the Bristol Stock Exchange responds to orders and publishes market data. Previously, the exchange used an inconsistent method of responding to orders, sometimes via Python objects, sometimes via amorphous strings including "Addition" or "Overwrite". My aim was to design structured classes that represented the different types of responses the exchange would want to send to clients. As such, I implemented two abstract classes *OrderBookResponse* and *MarketDataEvent* for transaction responses and market data publication respectively. Each abstract class contained multiple instantiatable subclasses for the varying message types. OrderBookResponse had the messages, *Acknowledged*, *Filled*, *Rejected* and *Cancelled* for the varying possible states of an order request after it had been submitted to the exchange. Meanwhile the MarketDataEvent class had the messages *LastSalePrice* and *BBOChange* to inform clients of the last sale price and when the best bid or offer changes respectively. I incorporated these message classes into the matching functionality, which passed the correct information to two placeholder observer functions, one for transactions and one for market data. The design was that these functions would implement an observer design pattern and transmit those messages to the corresponding clients - at this stage via

an undetermined technology. Regrettably, these generic message types were removed from the final implementation in this thesis, but their development proved to be an insightful exercise in thinking and designing exchange communication.

### 3.1.4 Investigating Python Web Frameworks

As Python was the selected language for this RESTful financial exchange, it was essential to investigate the various web-based frameworks that are compatible with Python. A web framework is a software package designed to support the development of web applications including RESTful APIs. For the Python language, there are two predominant frameworks used in industry, *Flask* and *Django*.

#### Flask

Flask is a microframework that enables the development of web applications varying from a single file to a full project. Flask aims to keep its core simple by requiring no additional tools or libraries to run, but extensible through utilisation of extensions [21]. As such, Flask's core does not provide a database abstraction layer, form validation or use any pre-existing third-party libraries. Consequently, the framework does not enforce a project structure, it makes no decisions for you, and thus is unopinionated - leaving the design of the application entirely in the hands of the developer. Flask is the logical choice for those building lightweight applications or those who wish to have complete control over the application's choice of libraries.

#### Django

In comparison to Flask, Django is a far more comprehensive high-level framework that encourages rapid development and clean, pragmatic design. The framework aims to provide an all-inclusive experience, taking much of the hassle of Web development away from the developer [15]. It follows the model-view-controller architectural pattern and defines structured rules on how to develop applications using its libraries. Out of the box, it provides a directory structure for your apps, database interfaces and an admin panel. Django however has a substantially larger footprint, consisting of approximately 240000 lines compared to Flask's 10000 lines of source code. Conclusively, Django is the logical choice for those working on common simplistic applications such as an e-store or blog and want a single, defined way of implementing them.

Throughout research and development, I learnt how to build web applications utilising both of these frameworks and experimented with a variety of their features. I analysed the advantages and disadvantages of each, before deciding to build the RESTful financial exchange using Flask.

### 3.1.5 Building the REST API

Flask was the logical choice for the financial exchange as the exchange is a non-standard application that did not require a front-end web interface. I wanted to have complete control over its structure and design and only utilise additional libraries when necessary. Furthermore, Flask enabled the exchange to have a lightweight footprint which I knew would be invaluable when attempting to deploy the software in the cloud later in the project.

I implemented a large proportion of the Bristol Stock Exchange's functionality as a RESTful API using the Flask web framework; available for download on GitHub from `https://github.com/bradleymiles17/bristol_stock_exchange`. RESTful APIs are built by defining endpoints; the publicly accessible points for a client to interact with the server. For each endpoint, you define the URL that routes to it and the HTTP methods it accepts. The Hypertext Transfer Protocol derives a variety of different method verbs that are used to define the kind of functionality that the endpoint implements, the most common of which are *GET*, *POST*, *PUT* and *DELETE*. Flask enables developers to create endpoints by adding the `app.route()` decorator to any function with the app's scope. This decorator informs Flask that the function is an endpoint and defines the URL that triggers that function.

During development, in order to simulate the client side of the simulation, I utilised an industry known tool for API development known as *Postman*. Postman allows developers to define and save HTTP requests of varying targets, verbs and parameters in structured collections. The tool vastly improved the

speed of my development, as I was able to define exemplary requests for each of the endpoints that my exchange exposed. As a result, I could quickly and efficiently reuse requests for creating bids and asks on the exchange without the need to constantly redefine them.

| Request | Verb | Endpoint | Description |
|---|---|---|---|
| Publish LOB | GET<br>GET | /<br>/api/lob | Returns the current state of the LOB as a JSON block. |
| Publish Orders | GET | /api/orders | Returns the list of all current orders on the exchange as a JSON block. |
| Get Order | GET | /api/orders/<int:order_id> | Returns the details of the order with order_id=x as a string. |
| Cancel Order | POST | /api/orders/<int:order_id>/cancel | Cancels the order with order_id=x |
| Place Order | POST | /api/orders | Places a new order on the exchange. |

Table 3.1: API documentation outlining the available endpoints for the BSE RESTful API.

My implementation exposes six endpoints, defined in Table 3.1 above. These endpoints allow a user to view the LOB, view all live orders or get the details of a specific order by their ID. Furthermore, a user can place or cancel any number of orders and the matching engine logic defined in Section 3.1.3 would automatically execute trades if any order crossed the spread. These requests use either the GET or POST verb, and their endpoints are categorised by the type of functionality they implement, such that all order related requests are grouped by the `api/orders/` URL. The application was designed in such a way that it could easily be extended; adding new functionality by simply defining new endpoints to the API.

### 3.1.6 AWS Elastic Beanstalk

Since the aim of this thesis was to deploy this application in the cloud, I explored a variety of different options on Amazon Web Services for deploying a Python Flask API. The most widely suggested was utilising AWS's Elastic Beanstalk, an orchestration service for deploying applications on various AWS services, such as Elastic Compute Cloud (EC2), Simple Storage Service (S3) and Simple Notification Service (SNS). Since the RESTful Bristol Stock Exchange was an in-memory application I only required Elastic Beanstalk to deploy the exchange on EC2 - Amazon's compute service. I configured and automated the deployment of the Bristol Stock Exchange through the Elastic Beanstalk Command Line Interface (EB CLI). After making a code change, the application could be redeployed with a single command, `eb deploy`. At this point, the interface would automatically package and upload a new version of the application to S3 and then deploy it to all of the servers I configured on EC2. The process was seamless and a perfect demonstration of continuous integration and continuous delivery in the cloud - a highly sort after aim for professional software development.

Shown in Figure 3.1, is a demonstration of the Bristol Stock Exchange implemented as a RESTful API and deployed in the cloud on AWS Elastic Beanstalk. Viewable within the URL bar of the Google Chrome browser, you can observe the address of the application is an Elastic Beanstalk instance. The browser's body displays the response from the exchange's API after calling the Publish LOB, `/api/lob`, endpoint which returns the latest LOB from the exchange.

The result of this work was a prototype distributed limit order book financial exchange built upon the Bristol Stock Exchange. It utilised the Flask web framework to expose a publicly available RESTful API for viewing and placing orders. For the first time, the exchange acted as an independent entity from the rest of the simulation and could be accessed via a variety of web requests utilising the HyperText Transfer Protocol. The matching engine functionality had been improved to remove some of the major limitations from the legacy BSE. Finally, the application has been successfully deployed to AWS Elastic Beanstalk and could be easily redeployed with a single command. Despite the success of this milestone, none of the above work features in the final implementation. It was at this point that I dived deep into how real-world exchanges operate and communicate to external clients. It was at this point that I discovered

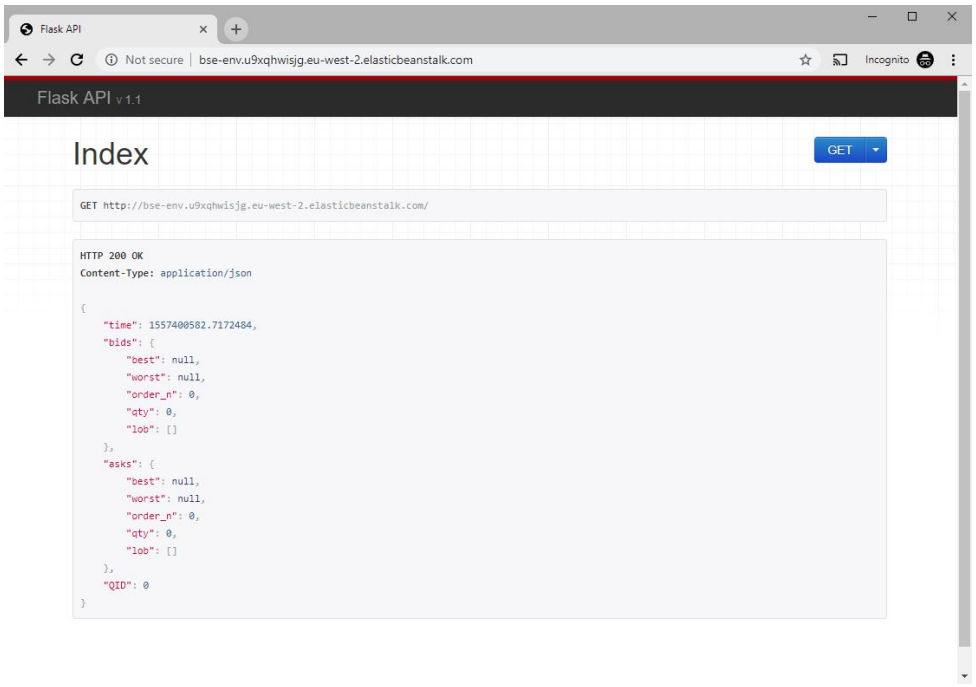



Figure 3.1: The Bristol Stock Exchange RESTful API deployed on AWS Elastic Beanstalk.

the Jane Street Exchange and the Financial Information Exchange (FIX) Protocol.

## 3.2 Milestone 2: FIX

As my knowledge of real-world exchanges and network communication protocols grew, it became more and more transparent that RESTful HTTP was not fit for purpose as the direct communication protocol between the exchange and clients. Firstly, HTTP is uni-directional, meaning a request is always initiated by a client, then processed and a response is returned by the server. Consequently, it would be impossible for the server to initiate the sending of messages and thus the exchange cannot push data to clients. Secondly, HTTP is half-duplex, such that a message can only be sent in one direction at a time; from client to server, followed by a response from server to client. Critically, this limits the overall speed of interoperability as each process must wait for the other to finish communicating before continuing. Thirdly, for each new HTTP request a new TCP connection must be initiated between client and server, this connection is then terminated once the response is returned. This process of constantly initialising and terminating TCP connections is computationally expensive and would be a serious overhead with the vast quantities of messages transmitted in real-world financial exchanges. As a result, it was a clear a new protocol would be required to communicate between traders and exchange, one that solves the limitations of RESTful HTTP.

### 3.2.1 Replacing RESTful HTTP

There were three main characteristics I was aiming for in a replacement protocol for RESTful HTTP. It had to be bi-directional, so both client and server could initialise sending messages. It had to be full-duplex, so messages could be sent in both directions simultaneously without having to wait for the channel to be clear. Thirdly, to avoid computational overhead, the communication channel had to operate over a single constant TCP connection. During my research, I tested two possible communication protocols, the *WebSocket* protocol and the *Financial Information eXchange* protocol.

#### WebSocket Protocol

The WebSocket protocol can be summarised as a bi-directional, full-duplex communication protocol over a single TCP connection. Historically, achieving bidirectional communication required an abuse of the HTTP protocol, constantly polling the server for updates. Standardised in IETF as RFC 6455 in 2011 [14], the WebSocket protocol solves this problem by providing a basic messaging framework, layered over TCP, that allows browser-based applications to communicate with servers without opening multiple HTTP connections. The technology has a variety of different use cases, notably social media feeds, multiplayer games and financial tickers - the latter of which intrigued me for my distributed Bristol Stock Exchange. The financial world moves fast, and as such polling servers using RESTful HTTP to get the latest prices would be far too slow. As a result, many brokerage firm websites implement WebSockets to stream the current exchange prices to their client's browsers. I experimented with this idea by creating a new Django website that implemented the WebSocket protocol to act as a trading client. Whenever the client made a HTTP request to the exchange, the response would then be forwarded to the user's browser via a WebSocket. Despite the technology's success for streaming data to browsers, its use as an application to application communication protocol is limited. As such, I concluded that it was not the ideal solution for this problem.

#### Financial Information eXchange (FIX) Protocol

The Financial Information eXchange protocol [7] or FIX was first created in 1992 to enable electronic communication of equity trading data between Fidelity Investments and Salomon Brothers. Authored by Robert Lamoureux and Chris Morstatt, it was designed to help reduce the large number of human errors that occurred whilst communicating equity data between traders, at the time done over the telephone. It was the first communication standard that used machine-readable data for traders to share, analyse and store financial equity information. At the time, the standard focused on pre-trade and trade communication and defined numerous domain specific messages for equity trading, such as New Order, Cancel Order and Indication of Interest (IOI).

Since its conception in 1992, the FIX protocol has become the language of the global financial markets. It is used by thousands of buy and sell-side firms, including investment banks, exchanges, asset managers and hedge funds to communicate trade information and complete millions of transactions daily. The FIX protocol is standardised by the *FIX Trading Community*. It is a non-profit, industry-driven standards

body that manages the continuous development and promotion of the protocol's messaging language. As such, the FIX protocol is constantly being updated to match the world's financial products and regulations. At the time of writing, there are 290+ financial service companies within the FIX Trading Community organisation, 35% in Europe, the Middle East and Africa (EMEA), 29% in the Americas and 26% in Asia Pacific [9]. Due to the success and adoption of the FIX protocol throughout the financial world, it has become the communication protocol of choice for the equity markets. Furthermore, through recent updates, the FIX protocol has begun expanding into other financial products, such as bonds and futures, where it too has seen considerable uptake.

At the application level, the FIX protocol is a domain specific language for financial information. But moreover, it is a session protocol that provides all of the technological requirements necessary to replace RESTful HTTP. FIX is a bi-directional, full-duplex communication protocol over a single TCP connection similar to WebSockets. FIX however has the added benefit of working seamlessly between application and application as well as understanding the financial domain. As such, I concluded FIX would be the perfect communication protocol for the project. It not only fulfils all of the requirements of a replacement protocol, but it also fulfils one of the aims of the project - to build an exchange using real world technologies. Finally, implementing FIX seemed like the perfect opportunity to provide another teaching utility for my distributed Bristol Stock Exchange. Since FIX is a real-world communication standard, teachers could use my simulation to explain and demonstrate how real trading clients and exchanges construct and transmit messages.

### 3.2.2 FIX Standards and Messages

The Financial Information eXchange protocol has gone through numerous iterations and updates since 1992. Before understanding how messages are constructed you must first select which version of the protocol you are going to support. The FIX Trading Community supports a variety of different versions and encodings which it calls the *Family of Standards* [8].

#### FIX "Family of Standards"

The FIX Trading Community, originally named FIX Protocol Limited (FPL), first published FIX 2.7 in early 1995, followed quickly by FIX 3.0 - yet neither version was widely adopted. FIX 4.0, FIX 4.1 and FIX 4.2 saw the adoption of the FIX protocol more widely in the financial world with each new version iteratively removing some of the ambiguities of its predecessor. FIX 4.2 was released in 2000, and became very popular amongst the equities, foreign exchange and derivative markets. FIX 4.3 added support for fixed income securities and FIX 4.4, released in 2003, added support for post trade processing. FIX 4.4 is the most widely adopted version of FIX in the financial world today, yet not the most recent version. One of the fundamental problems with FIX was that with each new version it was becoming increasingly difficult for financial institutions to upgrade their production systems to support the new standard. This was because the part of the standard that defines the structure of financial messages, known as the *application level standard* and the aspect that defines how messages are communicated, known as the *session protocol standard*, were included within the same FIX version. Consequently, it was impossible to release new versions of the application level standard or the session protocol standard without impacting the other. This inspired the creation of a new session protocol standard called the FIX Transport Session Protocol (FIXT) and a new application level standard, FIX 5.0, in 2006. Unlike previous FIX standards, FIX 5.0 contained no session layer specific messages and focused solely on deriving the structure of the financial messages. As a result, to implement FIX 5.0, you must use it in conjunction with FIXT. This separation helps financial institutions to incrementally support new versions of the FIX standard, because the FIX Transport Session Protocol can support multiple versions of FIX independently.

The most recent FIX standard, released in 2009, is FIX 5.0 Service Pack 2 (SP2), written as FIX.5.0SP2. Comparable to FIX 5.0, it does not contain any session layer specific messages and thus must be implemented in conjunction with FIXT, the most recent of which is FIXT.1.1. In this thesis, I chose to implement FIX.5.0SP2 and FIXT.1.1, for the application level standard and the session protocol standard respectively. As a result, all subsequent explanations of the FIX protocol will use one of these standards depending on whether it resides within the application or session domain.

#### Message Construction

FIX messages are constructed by building delimited strings of tag/value pairs, known as *tagvalue* encoding. Each *tag* is assigned both a descriptive name and a numeric code to represent it. For example, the MsgType tag - which is required in every FIX message - has the unique numeric code 35. N.B: when referencing tags, it is standard to include its corresponding numeric code in the pattern *<descriptive_name>(<numeric_code>)*, thus *MsgType(35)* for the previous example. When constructing messages, multiple pairs, also known as fields, are concatenated together, delimited by the Start of Heading (SOH) character, often depicted in logs as the pipe character "|". Each field uses only the numeric code to represent the tag, followed by the value, delimited by the equals character, "=". Figure 3.2 demonstrates an example field, where the tag is the numeric code, *35*, mapping to *MsgType* and the value *D*, which maps to *New Order Single <D>* in the FIX 5.0 SP2 application standard.

```
35=D
```

Figure 3.2: Example FIX tagvalue encoding

FIX messages consist of three main parts, the header, the body and the trailer. The header, located at the start of the message, defines metadata such as the BeginString(8) tag and the BodyLength(9) tag. The former dictates the beginning of a new message and the protocol version used; the latter states the length of the message in bytes. The trailer, located at the end of the message, contains fields that validate the message, such as the CheckSum(10) tag whose value is a checksum over the message. The body contains all of the application layer data relevant to the message. Figure 3.3 is an example New Order Single message and contains the header fields: BeginString(8) and BodyLength(9); the trailer field: CheckSum(10) and the body fields: MsgType(35), MsgSeqNum(34), SenderCompID(49), SendingTime(52), TargetCompID(56), ClOrdID(11), HandlInst(21), OrderQty(38), OrdType(40), Price(44), Side(54), Symbol(55), TransactTime(60) and ClientID(109) sequentially.

```
8=FIXT.1.1|9=133|35=D|34=4|49=CLNT1|52=20190422-14:37:24.000|
56=SRVR|11=3|21=3|38=1|40=2|44=999|54=2|55=SMBL|60=20190422-14:37:24|
109=S11_SHVR|10=203|
```

Figure 3.3: Example FIX New Order Single message

The construction of these messages is constantly changing with each new version and thus it is paramount to read the documentation provided by the FIX Trading Community.

### 3.2.3 QuickFIX

QuickFix is a free and open source implementation of the FIX protocol [20]; written to support a variety of different languages including C++, Python, Ruby, Java, .NET and Go. At its core, QuickFIX is a series of binaries that manages the session handling between clients and servers, and defines constants and helper methods for constructing tags and messages in the chosen FIX language. The FIX protocol works by implementing a daemon, a program that runs continuously and manages the sending and receiving of requests, known as the FIXEngine. QuickFIX implements this FIXEngine via an application interface that defines seven methods, outlined below, that act as callbacks from your FIX sessions [19].

**onCreate()**: is called when QuickFix creates a new session. A QuickFix session persists throughout the lifetime of the application and exists regardless of whether a counter-party is connected. Once a session is created, messages can be sent to it. If no counter-party is connected, those messages will be stored in a buffer until a connection is established and message transmission can occur.

**onLogon()**: notifies you when a valid logon has been established with a counter-party. Concretely, the FIX login process has completed successfully with both parties exchanging valid logon messages.

**onLogout()**: notifies you when the counter-party has disconnected from the session. This can be as a result of a logout request or more severely because of network failure or termination.

**toAdmin()**: enables you to send administrative level messages from your FIX engine to the counter-party. Admin messages are not designed for normal use but provide a mechanism for the software developer to view additional logging.

**fromAdmin()**: notifies you when an administrative level message has been sent from the counter-party to your FIX engine. It is predominantly used by software developers for additional validation on logon messages such as validating passwords.

**toApp()**: enables you to send application level messages from your FIX engine to the counter-party. Depending on whether you are the trading client or an exchange, this may include new order requests or execution reports.

**fromApp()**: notifies you when an application level message has been sent from the counter-party to your FIX engine. This is the receiving end of the *toApp()* function from the counter-party and is used to receive application messages, such as new order requests and execution reports.

In order to build a distributed simulation, both the exchange and the trading clients within the Bristol Stock Exchange required their own FIX engine. In the language of FIX, the exchanges are known as *acceptors* because they accept connections from clients. Conversely, the trading clients are known as *initiators* as they initialise the connection to exchanges. Both acceptors and initiators require slightly different configuration to communicate with each other correctly. It was at this point that I created two new GitHub repositories, *dbse_ exchange* and *dbse_ trading-client*, available for download from:

- `https://github.com/bradleymiles17/dbse_exchange`
- `https://github.com/bradleymiles17/dbse_trading-client`

The former was created to manage the exchange, matching functionality and the acceptor FIX engine, whereas the latter would eventually be for the robotic trading agents, simulation code-base and the initiator FIX engine. These two repositories enabled me to experiment with configuring and implementing the FIX protocol and became the starting point that resulted in the creation of my globally distributed limit-order book financial exchange, known as the Distributed Bristol Stock Exchange (DBSE).

### Configuring QuickFIX

The QuickFIX library enables you to configure sessions by parsing a settings file to a FIX engine factory on instantiation of your application. The settings file contains two types of headings, a `[DEFAULT]` and a `[SESSION]` heading. The latter tells QuickFIX that a new session is being defined and the former is used to denote constant configuration across all sessions. There are a wide variety of different options outlined in the QuickFIX configuration documentation; below I aim to highlight the key options I used to build my first FIX-enabled trading client and financial exchange.

To configuring any FIX engine, you must first state the session protocol standard you intend to support via the `BeginString` option. Prior to FIX 5.0 and FIXT, this configuration was the FIX version, for example `BeginString=FIX.4.4`. However, since I was supporting FIXT.1.1, that is the value you specify for the `BeginString` option, hence `BeginString=FIXT.1.1`. QuickFIX is configurable for many different FIX versions, as such you are required to specify the `TransportDataDictionary` and `DefaultApplVerID` options. These options are paths to xml encoded files that define all of the tags in that version and how they should be structured to form messages. When you send a message via your QuickFIX engine, the message will be validated against the xml file, throwing an error if the message is malformed or missing any required fields. QuickFIX provides these xml configuration files - downloadable from their website [18]. Since I am using both FIXT.1.1 and FIX.5.0SP2, I downloaded the FIXT11.xml and FIX50SP2.xml files from the QuickFIX website and referenced them as `TransportDataDictionary=./fix/FIXT11.xml` and `DefaultApplVerID=./fix/FIX50SP2.xml` respectively.

Finally you must define the `ConnectionType` of the engine along with the `SenderCompID` and the `TargetCompID`, in a pair-wise fashion, for each session you want to create. These ID's are non-whitespace strings that identify the name of the sending engine and the target engine respectively. For example, if you are an exchange, and hence of connection type acceptor, you could define your `SenderCompID` as `SRVR` and then for each new `[SESSION]`, you would specify a unique TargetCompID, such as `CLNT1` or `CLNT2`. The combination of `BeginString`, `SenderCompID` and `TargetCompID` results in the creation of the *SessionID* in the format `<BeginString>:<SenderCompID>-><TargetCompID>`. This SessionID is used as the unique identifier when sending and receiving messages via the FIX engine. Once configured, QuickFIX provides a mechanism to the view those sessions via a web interface. Figure 3.4 displays the configuration for the `SRVR` engine; with the full configuration readable in Appendix A.1.

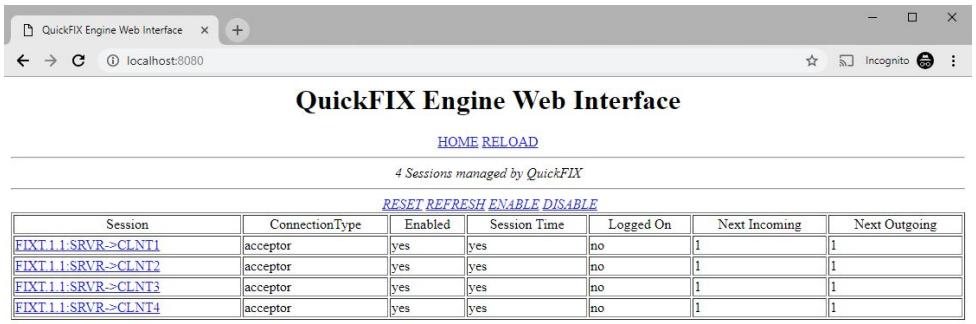



Figure 3.4: The QuickFIX Engine Web Interface displaying the SRVR engine's configured sessions.

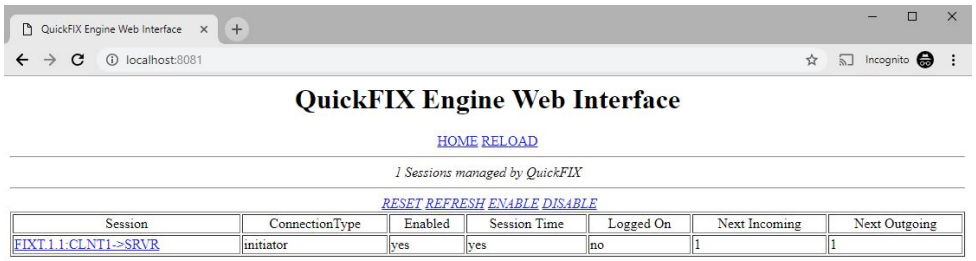



Figure 3.5: The QuickFIX Engine Web Interface displaying the CLNT1 engine's configured sessions.

Conversely, for each session you define on the exchange, you should have an equivalent FIX configuration for the trading agent. Consistent amongst all trading agents, the connection type is an initiator, and since I want them to all connect to the same exchange, their `TargetCompID`'s are all `SRVR`. The varying factor is their `SenderCompID` which are `CLNT1`, `CLNT2`, etc.. so that they are distinguishable from the server. Figure 3.5 displays the configuration for the `CLNT1` engine; with the full configuration readable in Appendix A.2.

### 3.2.4 Implementing FIX

Once I had sessions configured between an exchange acceptor and a trading client initiator, it was time to implement the FIX engine application interface. This was one of the most challenging aspects of the project as I had to decide which FIX messages the Distributed Bristol Stock Exchange would support. Although QuickFIX handles the sessions between initiators and acceptors, it is entirely down to the developer to implement how those messages are constructed and sent, as well as how they are handled once they are received. I spent a long time reading the FIX documentation and determining which of the 118 different messages types would be best to implement for the DBSE. The legacy Bristol Stock Exchange had functionality to place orders, to cancel orders, and an inconsistent and amorphous way of returning order acknowledgement and trade execution data. Consequently, these three aspects were the focus of my research for which I discovered their equivalent FIX counterparts, *New Order Single*, *Order Cancel Request* and *Execution Report*.

#### New Order Single <D>

The New Order Single message type has code D and is used by financial institutions wishing to electronically submit a single equity order onto an exchange for execution. It has several required fields, such as the order type (i.e. Limit Order), the side (i.e. Bid or Ask), the order quantity and the transaction time. The order is identified by the client order id (ClOrdID) - explained in detail in Section 3.2.5 - and the client id (ClientID). I utilised the `ClientID` field to identify which trading agent had created the order on a given trading client. This is not orthodox, as usually you would only have one trading algorithm running per FIX session and thus would not require additional identification regarding the origin of an order. I made this implementation decision however, as it was the best way to integrate the existing simulation structure of the BSE into my DBSE and enabled multiple algorithms to run per FIX session. The New Order Single message also requires the order's symbol, which is a unique identifier, the ticker-symbol, of the stock you are dealing with - in the real world this could be `AMZN` (Amazon.com) or `AAPL` (Apple.com).

The DBSE however still assumes that there is only one tradable commodity on the exchange and thus I implemented the symbol field to contain a placeholder of `SYBL`, which is then ignored by the matching engine. Categorically, the New Order Single message contains all the information required to place an order onto a financial exchange in a structured and consistent manner.

```
def place_order(self, order: LimitOrder):

    trade = fix.Message()
    trade.getHeader().setField(fix.BeginString(fix.BeginString_FIXT11))
    trade.getHeader().setField(fix.MsgType(fix.MsgType_NewOrderSingle))

    trade.setField(fix.ClOrdID(str(order.id)))
    trade.setField(fix.ClientID(str(order.client_id)))

    trade.setField(fix.HandlInst(fix.HandlInst_MANUAL_ORDER_BEST_EXECUTION))
    trade.setField(fix.Symbol(order.symbol))
    trade.setField(fix.Side(side_to_fix(order.side)))
    trade.setField(fix.OrdType(fix.OrdType_LIMIT))
    trade.setField(fix.OrderQty(order.qty))
    trade.setField(fix.Price(order.price))
    trade.setField(fix.TransactTime())

    fix.Session.sendToTarget(trade, self.sessionID)
```

Listing 3.1: Trading Client Place Order Implementation.

Listing 3.1 demonstrates my implementation of the construction of a FIX New Order Single message. Written into the FIX engine of the trading client, it constructs a new empty message and adds all the necessary fields from the LimitOrder class. The message is then sent to the exchange via the `fix.Session.sendToTarget()` method, which accepts both the message and the sessionID as parameters.

**Order Cancel Request <F>**

The Order Cancel Request message type has code F and is used to request the cancellation of all of the remaining quantity of an existing order. It requires many of the same fields as the New Order Single message, notably the Client Order Id (ClOrdID). I implemented the exchange to extract the id of the session and the ClOrdID from the message, find the appropriate order in its limit order book and remove it.

```
def cancel_order(self, order: LimitOrder):

    cancel = fix.Message()
    cancel.getHeader().setField(fix.BeginString(fix.BeginString_FIXT11))
    cancel.getHeader().setField(fix.MsgType(fix.MsgType_OrderCancelRequest))

    cancel.setField(fix.ClOrdID(str(order.id)))
    cancel.setField(fix.OrigClOrdID(str(order.id)))

    cancel.setField(fix.Symbol(order.symbol))
    cancel.setField(fix.Side(side_to_fix(order.side)))
    cancel.setField(fix.TransactTime())

    fix.Session.sendToTarget(cancel, self.sessionID)
```

Listing 3.2: Trading Client Cancel Order Implementation.

Listing 3.2 demonstrates my implementation of the construction of a FIX Order Cancel Request message, which is also written into the FIX engine of the trading client. It constructs a new empty message, adds all of the necessary fields from the LimitOrder class and then sends it to the exchange via the `fix.Session.sendToTarget()` method.

**Execution Report <8>**

The Execution Report message type has code 8 and is used by my exchange for several reporting duties, including:

1. Confirming the receipt of an order.
2. Confirming changes to an existing order (i.e. cancel).
3. Relaying order status information.
4. Relaying fill information on live orders.

In a real world financial exchange it is used for all of the above, plus the reporting of rejected orders and the reporting of post-trade fees. Both of which I omitted from my implementation as the exchange cannot reject orders nor does it have the concept of trading fees. The Execution Report message serves multiple purposes and as such has its own required `ExecType` field which is used to describe the purpose of the execution report. I have implemented two execution types, `ExecType_ORDER_STATUS` and `ExecType_FILL`. The former is used to notify the client of any changes to the order prior to a trade occurring. The latter is used to notify the client if a trade has occurred, either partial or full.

```
def create_execution_report(self, so: SessionOrder, exec_type: fix.ExecType):
    report = fix.Message()
    report.getHeader().setField(fix.MsgType(fix.MsgType_ExecutionReport))

    report.setField(fix.OrderID(str(so.order.id)))
    report.setField(fix.ClOrdID(str(so.order.ClOrdID)))
    report.setField(fix.ClientID(so.order.client_id))
    report.setField(fix.ExecID(self.gen_exec_id()))
    report.setField(fix.ExecType(exec_type))
    report.setField(fix.OrdStatus(order_status_to_fix(so.order.order_state)))

    report.setField(fix.LeavesQty(so.order.remaining))
    report.setField(fix.CumQty(so.order.qty - so.order.remaining))
    report.setField(fix.AvgPx(0))

    report.setField(fix.Symbol(so.order.symbol))
    report.setField(fix.Side(side_to_fix(so.order.side)))
    report.setField(fix.OrderQty(so.order.qty))
    report.setField(fix.Price(so.order.price))
    return report

def send_trade_reports(self, trades: List[Trade]):
    def send_trade_report(so: SessionOrder, price, qty):
        report = self.create_execution_report(so, fix.ExecType_FILL)
        report.setField(fix.LastPx(price))
        report.setField(fix.LastQty(qty))

        fix.Session.sendToTarget(report, so.session_id)

    for t in trades:
        send_trade_report(t.buyer, t.price, t.qty)
        send_trade_report(t.seller, t.price, t.qty)

def send_order_status(self, so: SessionOrder):
    report = self.create_execution_report(so, fix.ExecType_ORDER_STATUS)

    fix.Session.sendToTarget(report, so.session_id)
```

Listing 3.3: Exchange Execution Report Implementation.

Listing 3.3 demonstrates my implementation of the construction of FIX Execution Report messages for both order status updates and trade reports. The `create_exectution_report()` function creates a generic execution report message. It builds the report from an empty FIX message and adds the OrderID

and ClOrdID fields to identify the order appropriately. Furthermore, it includes fields to notify the client of any changes to that order, such as the order status, its remaining quantity, the cumulative quantity (the amount of the order that has been executed), as well as the last price a transaction occurred at.

For all of the above message types, there is undeniably more that could be implemented to improve and perfect the creation and selection of data transmitted within a message. The Execution Report message type alone has over 200 possible fields that could be populated within the message structure, exacerbating the complexity of financial exchanges. The challenge for me for all of the above types was identifying which fields were required and which additional fields could be appropriate for the Distributed Bristol Stock Exchange. I believe I have succeeded at this task as this implementation maintains and improves the original functionality provided by the Bristol Stock Exchange. Orders can now be placed and cancelled via any configured client and the method in which the exchange responds now uses a standardised execution report message.

### 3.2.5 Exchange Improvements

For the trading clients and the exchange to accommodate the distributed nature of the FIX protocol the underlying implementation of the Order class, the matching engine and the trading agents had to change. Outlined below are the key improvements I made to the exchange to support FIX.

#### Session Handling: SessionOrder

For the exchange to know the origin of a given order, it was essential that the session id was also stored with the Order class. This could be implemented simply and naively by adding an additional variable within the existing Order class of the Bristol Stock Exchange. I did not want to do this however, as the session id is not relevant to the order itself and thus would contaminate the purity of the class. As a result, I took this as an opportunity to redesign the entire Order class to handle both market and limit orders in a scalable structure. The result of my restructure was an abstract Order class that contained all the required fields of an order that could handle a quantity greater than one. This included the id, symbol, side, quantity, remaining quantity and the order state. Previously, orders had no concept of state and the side was just the string "Bid" or "Ask". I replaced both fields with classes that defined a list of enumerated types. This vastly improves the quality of the code as whenever those types are referenced or compared, you can use the enumerated values rather than the direct strings. This vastly reduces the chance of software development error by isolating the value of the type in one location. If the developer ever wanted to change its value in the future, it is changed in one place and thus automatically applied throughout the application.

I created two new classes, *LimitOrder* and *MarketOrder* which both extend my new abstract Order class. Both classes are instantiatable and contain an `order_type` variable whose value is another enumerable type (i.e. Limit or Market). Limit orders are distinguishable from market orders by the fact that they also require a limit price. As such, the LimitOrder class requires an additional parameter in its constructor for price. I implemented these two classes as a proof of concept to demonstrate that orders of varying types could be composed in a structured manner; laying the foundation for future work; implementing far more complicated types, such as *Iceberg* orders or *Limit-on-Close* orders.

Finally to add the information relevant to the session, such as the session id and the timestamp for when the order was created, I implemented a *SessionOrder* class. This class accepted an order of any type and became the generic class that was stored in the exchange's limit order book. One notable challenge I had during its creation was overriding the `__deepcopy__` method - a process that recursively copies all values in an object to a new object in memory. This problem was realised when implementing the execution reports for partially filled to filled orders. For example, if orders [ASK P=1.45 Q=5] and [ASK P=1.50 Q=7] were on the exchange, and a new order [BID P=1.60 Q=10] was placed. The expected response to the bid order would be two trade executions; the first a fill with price 1.45 and remaining quantity 0, the second a partial fill with price 1.50 and remaining quantity 2. What was happening however was the execution reports would publicise the final state of the bid order for both reports. This was because the data added to the reports was accessed by reference from the LOB, and thus to solve this, a deep copy of the order in its partial states would be required for each trade report. The `session_id` variable however was of a non-standard, complex type from the QuickFIX library that could not be copied recursively. As

a result, I overrode the `__deepcopy__` method to ignore the session id but proceed with the rest of the class - solving the problem.

### Id Trouble: ClOrdID

One of the unforeseen troubles when creating a distributed financial simulation is that you cannot have a single id to represent an order across the distributed system. When a client creates a new order, it generates a new order id, implemented as an integer value that increments from 0 by 1. Equivalently, when an exchange receives an order, it needs to store it within its limit order book. A naive approach would be to use the order id generated by the client as the unique identifier in the exchange. This does not work however when an exchange accepts orders from multiple clients, as they will have overlapping order ids. The solution is to have the exchange and clients have independent unique identifiers for their orders, known as the order id (OrderID) and the client order id (ClOrdID) respectively. Whereas the client only maintains its order id, the exchange holds both the *OrderID* and the *ClOrdID*. This ensures that when the exchange publishes execution reports with both ids, the receiving client knows which order the report refers to. To implement this I had to add the *ClOrdID* variable to the Order class and rewrite the way the exchange cancels orders. Since, the client has no understanding of the exchange's order id, the client makes order cancel requests using the *ClOrdID*. Consequently, finding and deleting the correct order on the exchange has to be done via the *ClOrdID* in the Distributed Bristol Stock Exchange, where as the legacy BSE only used a single id.

### Real-time

One of the underlying aims outlined in Section 1.4, was the goal of making the Bristol Stock Exchange real-time. In the existing BSE, time was an abstract variable, a float, where the value 1.0 represented one second. This time variable was incremented systematically by iterators and had no correlation with the passing of actual time. To rectify this, I had to rewrite all of the underlying functionality that referenced this float value and replace it with objects from the Python *datatime* module, namely *datatime* and *timedelta*. *datatime* is an object that supports the manipulation of dates and times and *timedelta* is an object that represents duration - the difference between two dates or times. Because of this work, orders stored within the exchange are now timestamped with a datetime that is from a real point in history.

### Improving the Matching Engine

In Section 3.1.3, I described how I updated the legacy order matching functionality to be a recursive function that enabled orders to have a quantity greater than one. Upon re-evaluation of that implementation, and further inspiration from a GitHub repository titled "Financial exchange written in Go, designed for algorithmic trading tests" [13], I concluded that it could be improved and simplified by removing the recursion.

```
def __match_trades(self):
    trades = []
    timestamp = time.time()
    best_bid = self.bids.get_best_order()
    best_ask = self.asks.get_best_order()

    while best_bid is not None and best_ask is not None and
        best_bid.order.price >= best_ask.order.price:

        # calculate resting order price
        if best_bid.timestamp < best_ask.timestamp:
            price = best_bid.order.price
        else:
            price = best_ask.order.price

        # calculate trade qty
        qty = min(best_bid.order.remaining, best_ask.order.remaining)

        # fill bid and ask side
        self.__fill(best_bid.order, qty)
        self.__fill(best_ask.order, qty)
```

```
            # create trade
            trade = Trade(timestamp, deepcopy(best_bid), deepcopy(best_ask), price, qty)

            trades.append(trade)
            self.tape.append(trade)

            # remove any filled orders from LOB
            if best_bid.order.remaining == 0:
                self.delete(best_bid)
                best_bid = self.bids.get_best_order()

            if best_ask.order.remaining == 0:
                self.delete(best_ask)
                best_ask = self.asks.get_best_order()

    return trades
```

Listing 3.4: DBSE's matching function.

Listing 3.4 demonstrates the final implementation of the Distributed Bristol Stock Exchange's matching function. If an order crosses the spread, it calculates the resting order price, such that the older order takes priority and the quantity that will be traded. It then fills both the bid and ask sides, updating the limit order book and creating an object that represents the trade. Finally, if either the best bid or best ask's remaining quantity is zero, it is deleted from the LOB and the best bid or ask is updated. This process repeats until no orders cross the spread.

### Typing

The final change that I made to the Distributed Bristol Stock Exchange was to add typing to a large proportion of the code-base. Python is a type-less language, meaning that when a variable is defined, the type does not need to be specified. In many cases this can be argued as being beneficial to the developer as it is quicker and easier to implement changes. However, type-less languages provide no clarity as to what methods accept or return, often leading to run-time errors. Since I was implementing FIX, a strictly typed protocol, I found it beneficial to specify types within my application where possible. Python allows the ability to specify primitive types for function definitions and provides the *typing* module for more complex types, including *List* and *Optional*. Within a function definition, a parameter type can be specified by including the type after the variable name, separated by the colon character, ":". For a return type, at the end of the function declaration, include an arrow, "->", followed by the type.

```
def side_to_fix(order_side: Side) -> fix.Side:
    map = {
        Side.BID: fix.Side_BUY,
        Side.ASK: fix.Side_SELL
    }
    return map[order_side]
```

Listing 3.5: An example of Python typing.

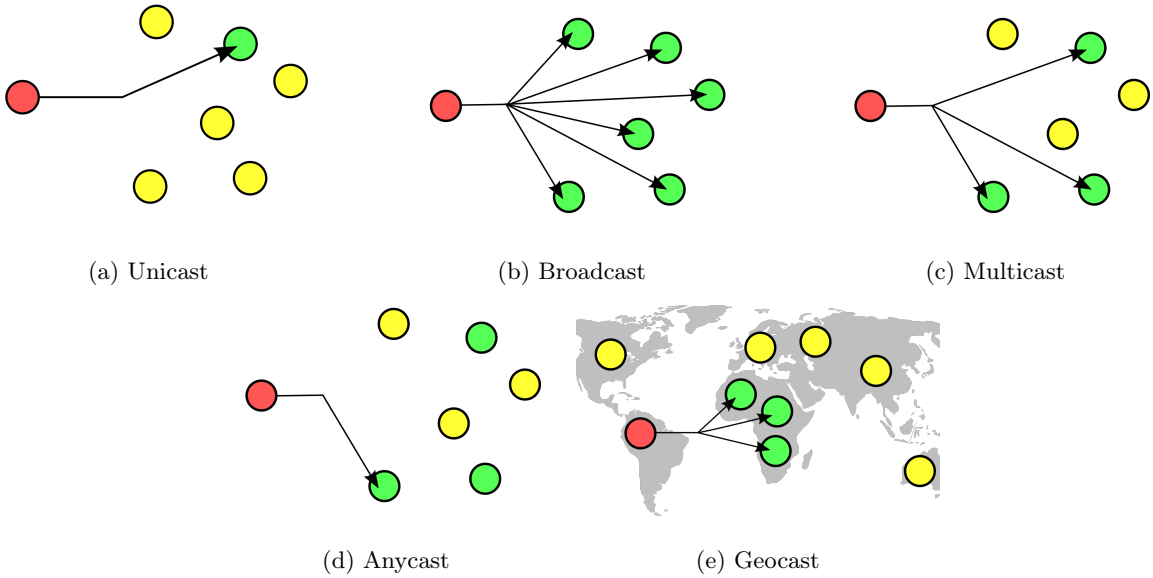
Listing 3.5 demonstrates a good example of typing in Python using a mapping function I implemented. `side_to_fix()` converts my internal representation of an order's side to the equivalent value necessary in a FIX message. Within the function definition, you can observe the `order_side` variable is of type `Side` and the function returns a variable of type `fix.Side`. Typing the application was additional work that I believe has contributed to fulfilling the aim of making the DBSE readable and scalable for subsequent users who wish to improve and work with it.

## 3.3 Milestone 3: Market Data

At this point in the development of the project, clients could place orders onto the exchange and receive updates about those orders via execution reports. One fundamental problem with this however is that many of the trading algorithms described in Section 2.4.2 require market data; they require knowledge of the exchange's LOB. At this point in the project, the Distributed Bristol Stock Exchange currently provided no mechanism to publish any market data to interested counter-parties. Consequently, it would be impossible to implement any other trading algorithm besides Giveaway - since it is the only trading robot that does not require market data. As a result, designing and implementing a market data publication service for the exchange, and a listening service for each client became the next major challenge and milestone in the project.

### 3.3.1 Addressing Methods

Described in the Jane Street Exchange case study in Section 2.5, the underlying technology that supports the efficient and fair transmission of exchange market data to clients is the User Datagram Protocol (UDP). It does this by utilising a routing scheme from the Internet Protocol (IP) known as multicast addressing. The Internet Protocol provides five different addressing methods, and it is useful to understand each of them to justify why multicast is the best option. Diagrammatically shown in Figure 3.6 and explained below are the five addressing methods that IP provides:



(a) Unicast (b) Broadcast (c) Multicast

(d) Anycast (e) Geocast

Figure 3.6: The five Internet Protocol addressing methods, reproduced from [33].

**Unicast** addressing uses a one-to-one mapping from a sender to receiver. During unicast transmission, a packet is sent from a single source to a specified destination endpoint and is the predominant transmission for Local Area Networks and the Internet.

**Broadcast** addressing uses a one-to-all mapping from a sender to all receivers. During broadcast transmission, a single datagram from one sender is routed to all possible receivers on the network. The network automatically replicates datagrams as needed to reach all aspects of the network.

**Multicast** addressing uses a one-to-many-of-many mapping from a sender to many receivers within a subset of the network. During multicast transmission, a single datagram from one sender is routed throughout the network and delivered to specified receivers - if they are subscribed to the multicast group. The internet protocol reserves one-sixteenth of its address space to multicast addressing from the IP address 224.0.0.0 to 239.255.255.255; a multicast group is an IP address within this range. Membership of the group is dynamic and controlled by the receivers, and the routing of datagrams is managed by the network. This is achieved by the routers within a multicast network learning which sub-networks have

active clients, for each multicast group, and routing datagrams to them accordingly. It has significant bandwidth savings compared to unicast - approximately 1/N, for N unicast clients.

**Anycast** addressing uses a one-to-one-of-many mapping from a sender to one receiver within a group. During anycast transmission, the target is selected from a group of receivers based on which is the nearest, according to a distance measure. Transmission then occurs to that selected receiver equivalently to unicast.

**Geocast** addressing uses a one-to-many-of-many mapping from a sender to many receivers within a subset of the network. Geocast addressing is a specialised form of multicast addressing that delivers information in a network to a destination classified by their geographical location.

Out of the five addressing methods that the Internet Protocol provides, multicast addressing is the best option as it provides an efficient method for distributing market data to specific targets. The exchange can configure and publish market data to a multicast group and all clients who pay to have access to the network can subscribe to this group and receive the market data. Multicast transmission ensures that the network is not overloaded with duplicate packets, as only a single packet is added to the network by the exchange and it is then suitably distributed by the network's routing. If unicast addressing was used, packets would need to be duplicated for each receiver, vastly increasing the workload of both the exchange and the network. For IP multicast to work however, additional protocols including the Internet Group Management Protocol (IGMP) is required to support and manage the routing. This is provided by many routers, but it is down to the network manager to configure this protocol accordingly. As a result, I began implementing functionality to provide market data publication for the DBSE via UDP multicast to run within my personal local area network.

### 3.3.2 UDP Multicast

Implementing the UDP protocol is possible through the Python *socket* module. The module provides low-level access to the socket interface provided by the machine's operating system. It translates the underlying Unix system calls and socket interface to a Python object-oriented structure to simplify the configuration of web-based communication protocols.

#### Sending Multicast Messages

To configure sending UDP multicast messages for the exchange, I created a socket object with the internet address family, `AF_INET`, and the datagram socket type, `SOCK_DGRAM` - configuring the socket to use the IP and UDP protocols respectively. Next, to ensure packets do not live within the network indefinitely, it is advisable to configure a maximum life length, known as the *time-to-live*. Using the set socket option method, `socket.setsockopt()`, I set the maximum time-to-live, `IP_MULTICAST_TTL`, for any datagram to one second, for all multicast traffic at the IP level. Once this is configured, UDP multicast traffic can be sent through the network via the `socket.sendto()` method; passing both the encoded message and the multicast group. Any IP address within the multicast group range can be used, for which I chose the arbitrary IP number/port pair, `224.3.30.33:10000`.

#### Receiving Multicast Messages

To configure receiving UDP multicast messages for the trading clients, I again created a socket object configured to use the IP and UDP protocols. This socket must then be bound to an address on the server, via the `socket.bind()` method. Once the socket is created and bound, it must be added to the multicast group that I configured for the exchange. This is achieved, by setting the `IP_ADD_MEMBERSHIP` option on the socket to the 8-byte packed representation of the multicast group address, `224.3.30.33`, followed by the network interface on which the server should listen to traffic. Data can be read from the receiver by requesting a packet using the `sock.recvfrom()` method. This will return both the encoded message as well as the address of the multicast group in which it was received from.

### 3.3.3 Market Data Publisher and Receiver

Once I had a communicating protocol, it was time to use it for publishing and receiving market data. I began by extending the UDP multicast messaging functionality - that I had previously implemented -

to transmit an anonymised version of the exchange's limit order book whenever something happened on the exchange, i.e. an order is placed or cancelled. This functionality was placed into two new classes, one for publishing and one for receiving market data for the exchange and trading clients respectively. One important consideration with transmitting data via UDP is that there is a practical limit - imposed by the IPv4 protocol - on the size of a packet's payload, 65,507 bytes. Although unlikely, if the size of the anonymised LOB was unbounded, there would be a point in which the LOB grew so large it was exceed the capacity for a single packet and throw an exception. I realised however, that sending the entire contents of the LOB is unnecessary, as the current set of DBSE trading algorithms only utilise the best and worst prices to calculate their quotes. For that reason, I restricted the exchange to only publish the best ten bid and ask prices on the LOB, as well as the worst price. This bounded the maximum size of the published LOB and normalised the average size of the UDP packets.

After restricting the payload size, I began investigating ways to optimise the publishing and receiving processes. In real world financial exchanges, market data publication is managed by an isolated micro-service on the exchange's internal network, as shown in Section 2.5. Since my exchange was implemented as a single application however, hardware parallelisation was not feasible; instead I sought ways to optimise the software, via introducing asynchronicity. An asynchronous process is one that can be executed independently of the main application. This is advantageous in the domain of market data transmission as the publication and retrieval of data can be completed independently from the matching engine and trading robots respectively. Asynchronicity can be achieved by a technique called multi-threading. A thread is the smallest sequence of programmed instructions that can be managed independently by a scheduler. By creating multiple threads, one for the market data publisher and one for the receiver, each process can operate independently from their respective applications. Neither the exchange nor the trading clients must wait for the market data to be transmitted before continuing their own execution. For example, by threading the market data publisher, when the exchange wishes to publish market data following a trade execution, the matching engine offloads the data to the publisher thread and thus can continues accepting client orders via FIX. Similarly, whenever a client receives market data, the trading robots do not have to wait for the market data to be processed before continuing, they just operate with the latest market data they possess. Appropriately, neither the matching engine or trading robots must wait for the publisher or receiver respectively to finish executing before continuing.

Threading can be achieved in Python by importing and utilising the *threading* module, specifically the Thread object. The module constructs higher-level threading interfaces on top of the low-level implementation provided by the *_thread* module. When the exchange application is executed, my implementation instantiates a new Thread object that continuously checks whether there is any new market data, publishing it if true. Market data is sent to the publisher through a *Pipe*, provided by the *multiprocessing* Python module. A pipe is a technique for passing information from one process to another asynchronously. Data is placed onto the pipe by the sending process and stored in a first-in-first-out (FIFO) structure. If the receiving process is not ready to accept the data, then the pipe will hold it temporarily in a buffer until it is ready to be read. Applied to market data, the matching engine thread places data onto the pipe and the publication thread reads off the pipe and publishes the data via UDP multicast. The exact reverse process for the market data receiver and trading robots is implemented within the DBSE trading client application.

### 3.3.4 Adding the Simulation

The final component of the Distributed Bristol Stock Exchange was to incorporate the simulation configuration functionality into the trading client application. This was an extensive amount of work that can be summarised by three key components, real-time order scheduling, configuration abstraction and market reaction.

#### Real-time Order Scheduling

For trading algorithms to make decisions and place orders on the exchange, they have to hold customer orders illegible to trade. This is comparable to Smith issuing cards to students and telling them to trade at a price no more than what is stated on the card; the student then decides when best to fulfil that order on the exchange. In an entirely simulated environment, the responsibility of Smith is replaced by an order scheduler, a configurable process that distributes customer orders over time. The Bristol Stock Exchange

had an order scheduler, however its implementation hinged on the abstract variable for time and none of its functionality occurred in sync with the real passing of time. I wanted to implement this equivalent functionality in the Distributed Bristol Stock Exchange but to do so it had to be entirely rewritten for real-time execution. I began by experimenting with a variety of different open-source real-time scheduling libraries available for Python, namely *schedule*, *sched* and *apscheduler*. Out of the three, the latter was significantly more complicated yet provided the most comprehensive functionality. *apscheduler* stands for Advanced Python Scheduler and enables programmers to define jobs that schedule Python code to be executed later, either once or periodically. Jobs are invoked by individually configured *triggers* - a point in time where a condition will become true - and maintained within a pool until that trigger is satisfied. The Advanced Python Scheduler provides three built-in configurable triggers, *date*, *interval* and *cron*:

**date**: triggers will run the job once at a certain point of time in the future.

**interval**: triggers will run the job at fixed time intervals, indefinitely, unless an end-date is specified.

**cron**: triggers will run the job periodically at certain time(s) of day.

In the Bristol Stock Exchange, the order scheduler could be configured by directly changing a Python object within the application's code-base. This object contained a variety of variables to control the scheduling rate, including the *interval* (how often new orders are created), the *timemode* (the rate in which orders are released) and the *ranges* (the price of those orders at different time intervals). I implemented this equivalent functionality for real-time execution in the DBSE using the *apscheduler* library, specifically the date and interval triggers. Using the interval trigger, every order scheduling interval would instantiate additional date triggers for the distribution of each customer order to the appropriate trading agent. These date triggers were timestamps in the future, configurable by the *timemode* option. The result of this work was a configurable real-time order scheduler for each trading client application. Unlike the BSE, it would now possible to have multiple different order schedulers running simultaneously against the exchange. Since each trading client application configured its own order scheduler, it was now be possible to test how individual trading clients can influence the exchange prices. Finally, in addition to order scheduling, I utilised the advanced Python scheduling library to report the time duration of the simulation periodically as well as schedule the trading agent's profitability calculations when the trading day ends.

### Configuration Abstraction

As previously noted, to configure the Bristol Stock Exchange's trading agents and order scheduler you had to directly change the application's Python code. Since my goal was to eventually deploy this software into the cloud, I did not want the DBSE's configuration to be code based. This is because editing code through a command line editor over Secure Shell (SSH) is often error prone and is time consuming for users. As a result, I abstracted and redesigned the configuration into a JSON file that could be imported when the application loads. This enables users to define multiple configurations in advanced and import the one appropriate for the test they are currently trying to run. The configuration can be divided into two main parts, the traders and the order scheduling configuration.

```
"traders": {
    "buyers": {
        "GVWY": 5,
        "ZIC": 5
    },
    "sellers": {
        "SHVR": 5,
        "SNPR": 5
    }
}
```

Listing 3.6: Example DBSE trader configuration.

Listing 3.6 demonstrates an example trader configuration for the Distributed Bristol Stock Exchange. For both the buy-side and the sell-side defined is a map that declares the number of robots of each trading type that you wish to instantiate. For the configuration above, the client application will create five Giveaway and five Zero-Intelligence Constrained traders on the buy side, as well as five Shaver and five Sniper traders on the sell side - a total of twenty agents.

```
"order_schedule": {
    "demand": [
        {
            "from": 0,
            "to": 600,
            "ranges": [{min": 90.0, "max": 120.0}],
            "stepmode": "random"
        }
    ],
    "supply": [
        {
            "from": 0,
            "to": 600,
            "ranges": [{"min": 105.0, "max": 140.0}],
            "stepmode": "random"
        }
    ],
    "interval": 20,
    "timemode": "drip-poisson"
}
```

Listing 3.7: Example DBSE order scheduler configuration.

Listing 3.7 demonstrates an example order scheduler configuration for the Distributed Bristol Stock Exchange. It defines the interval and timemode in which orders will be created, as well as the prices of those orders for both the demand and supply side. For the configuration above, every 20 seconds, between seconds 0 and 600 of the trading day, the order scheduler will create customer orders with a random price between 90p and 120p for demand and between 105p and 140p for supply. This configuration can be customised in a variety of different ways, using different time and step modes. A full example of a DBSE simulation configuration can be found in Appendix B.

**Market Reaction**

The final code implementation of my project execution was to change the way the Distributed Bristol Stock Exchange placed orders onto the exchange. Originally, on the BSE, every "second", a trader decided - of its own accord - whether it wanted to place its customer order onto the exchange. This process is very unrealistic however as the trader is only viewing the exchange periodically, not continuously. As a result, the trader is also only reacting to the changing market in one second intervals. In a real world distributed financial exchange, market data is being published every millisecond, not every second. Consequently, in the current implementation, even if the client received new market data from the exchange, it would not respond until the periodic second.

The significant advantage of the Distributed Bristol Stock Exchange is that it models latency and operates in real time. Therefore, by changing the trading agents to respond whenever they receive new market data, rather than every second, I could model an artefact in financial trading known as *race-to-market*. Race-to-market is a concept by which a trader can "steal the deal" if they learn and respond to a market change before a competitor. My hypothesis was that if I had two equivalent trading clients in geographically different regions, the one that was closer to the financial exchange would receive and respond to market data quicker and thus would be more profitable. I implemented the market reaction functionality by passing a callback function into the market data receiver. As a result, whenever the receiver acquired new market data, it would notify the simulation, which would in turn ask the traders whether they wanted to place an order on the exchange.

One significant challenge I faced because of this implementation was that the simulation could enter a deadlock. Since the exchange only publishes market data when the head of the LOB changes and the trading clients only respond when they receive new market data, the simulation would reach a stale state where no trading occurs. In the real world, due to the scale and the number of trading clients, this problem never occurs as market activity is constantly occurring. To resolve this problem in the Distributed Bristol Stock Exchange, I implemented a market data publisher override using the *apscheduler*; which causes new market data to be published every second even if the LOB is unchanged. This ensures the trading

clients continue to receive market data throughout the duration of the market day and the simulation can continue as desired.

### 3.3.5 Summary of Project Architecture

At this point in the project the Distributed Bristol Stock Exchange's code implementation was complete and ready for deployment. As shown in Figure 3.7, the DBSE encompasses a number of the components that were prevalent within the Jane Street Exchange, including the matching engine, client ports and market data publisher. Although the exchange's micro-services are contained within a single application and the trading clients must be connected to the internal network; the DBSE functions successfully as a distributed simulation. The simulation was configured to accept four different trading clients which communicated to the exchange via the real-world network-based Financial Information Exchange protocol. The chosen number of clients, four, was an arbitrary selection guided by my desire to host clients in at least three geographical regions, with one of those regions hosting a minimum of two clients. Additional clients can be added to the DBSE easily by editing the exchange's FIX configuration and creating an initiator configuration for each new client. Market data is published and received throughout the simulation via the User Datagram Protocol (UDP) and the multicast Internet Protocol addressing method. Presented is a real-time and real latency financial limit order book exchange simulation for research and teaching that was ready to be deployed on a global cloud infrastructure.

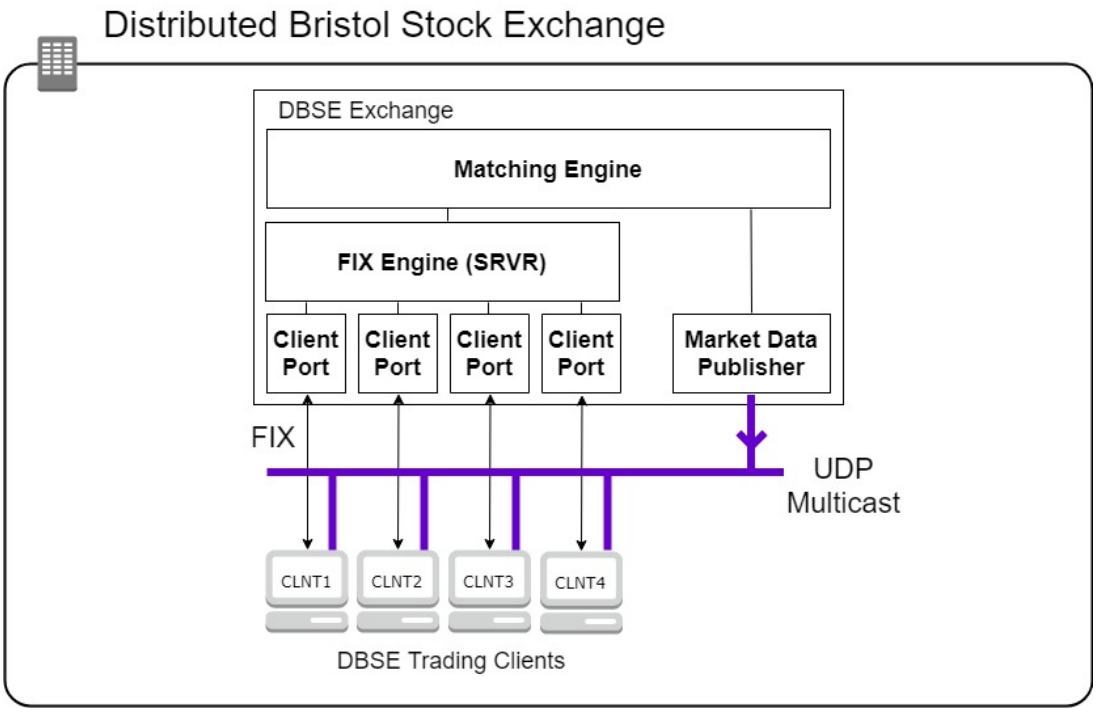



Figure 3.7: A graphical representation of the Distributed Bristol Stock Exchange.

## 3.4 Milestone 4: Cloud Deployment

To best illustrate the full benefits of the Distributed Bristol Stock Exchange, the trading clients need to be in geographically different regions from the exchange. This causes a disparity in the latency for each client and enables the simulation to test the race-to-market artefact. Consequently, the fourth and final milestone of this project was configuring both the network and compute resources to deploy the Distributed Bristol Stock Exchange on a global scale, at disparate locations around the planet. Out of the many cloud computing providers, I chose to use Amazon Web Service's as it currently provides the largest global coverage. Moreover, it offers the largest support community and documentation for its services.

### 3.4.1 Network and Compute Configuration

Before starting to configure a global network within Amazon Web Services, it was important to decide which geographical regions I wanted both the exchange and the trading clients to be located in. Although restricted to the regions and availability zones provided by the Amazon Cloud network, the exchange and trading clients could be positioned in any of the 21 global regions, shown in Figure 2.5. I decided I wanted to position the exchange in the London region as it provides a central location for Amazon's global infrastructure - from the perspective of the UK. For the trading client regions, I wanted to provide a variety of different options. First and foremost, I wanted to be able to run the simulation with multiple clients within the same region as the exchange. As a result, I knew I wanted to have at least two trading clients in London - within the same availability zone as the exchange. Subsequently, the other global trading client locations were arbitrary. I decided I wanted to position one in Ohio, America and one in Sydney, Australia as they were both a substantial and varying distance from the exchange.

#### Virtual Private Cloud (VPC)

The Amazon Virtual Private Cloud (Amazon VPC) is a service that enables consumers to provision a logically isolated section of the AWS Cloud [26]. It enables complete control over the virtual networking environment enabling users to configure many networking properties such as IP address ranges, subnets and routing tables. In order to achieve a fully distributed simulation, I had to configure VPCs in all three of my chosen regions, London, Ohio and Sydney, shown in Figure 3.8.


Figure 3.8: A graphical representation of the Distributed Bristol Stock Exchange global network, reproduced from AWS (2019) [27].

AWS automatically provides a default VPC in each region configured for IPv4 traffic with the Classless Inter-Domain Routing (CIDR) block as *172.31.0.0/16*. CIDR's are used to define the routable IP addresses within a network. The default routing, *172.31.0.0/16*, states that all resources allocated within the network must have an address beginning with IP *172.31*. Consequently, this CIDR block configures

65,536 possible unique private addresses in the VPC between the ranges *172.31.0.0* and *172.31.255.255*. Using these default VPCs would be perfectly fine if the simulation was deployed in isolated regions, however, to connect VPCs across regions, there CIDR blocks must be unique. As a result, I created two new Virtual Private Clouds in the Ohio and Sydney regions with the CIDR blocks *172.32.0.0/16* and *172.33.0.0/16* respectively. By default, new VPCs do not include an Internet Gateway; a component essential for enabling internet-routable traffic. Without an Internet Gateway, it would be impossible to SSH to any compute resource within the network, and thus impossible to interact with the Distributed Bristol Stock Exchange simulation. As such, I configured new internet gateways for each new VPC and attached them accordingly. Once all the VPC's had been created, it was now time to connect them.

### Peering Connections

In the real world, a financial exchange will have clients directly connected to their network, known as exchange hosting. This efficiently provides clients with the exchange's market data traffic at high speeds to accommodate trading. Although connected to the network, they might not necessarily be located within the same sub network or even within the same data centre as the exchange. The question arises then as to how two isolated networks - the exchange network and the client network - communicate in a secure and timely manner. In the real world, the solution is known as an IPsec VPN tunnel. IPSec stands for Internet Protocol Security and is a secure network protocol suite that authenticates and encrypts all packets sent via the Internet Protocol. VPN stands for Virtual Private Network and defines an encrypted link that enables users to send and receive data securely over a public network, such as the Internet. The infrastructure shown in Figure 3.9 demonstrates how it would be possible to configure a IPSec VPN tunnel between an exchange located within a VPC and a trading client located within its own corporate network.

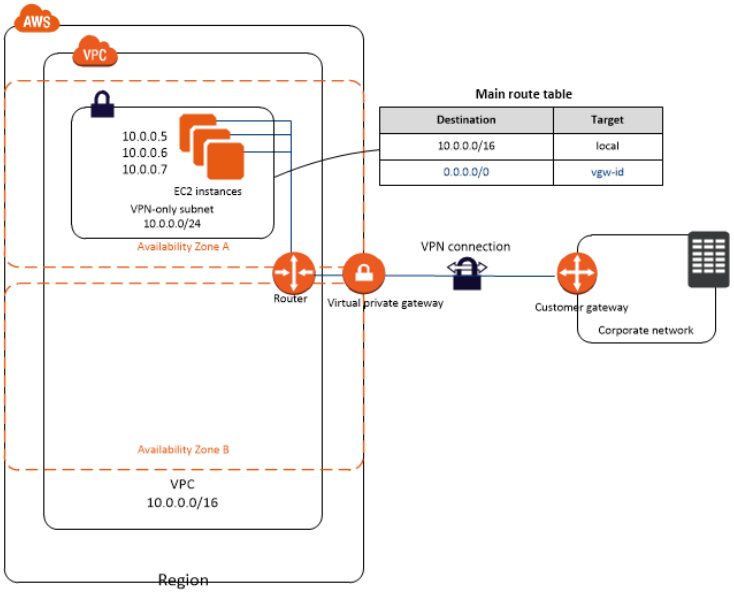



Figure 3.9: An example AWS configuration for a VPN Tunnel between two isolated networks, reproduced from AWS (2019) [28].

The main advantage of utilising the Amazon Cloud however is that trading clients do not need to be located within their own corporate network. The Amazon Cloud is a global network, and as such, neither the exchange nor the trading clients need to utilise the Internet to communicate across the globe. Within Amazon's infrastructure it is possible to configure connections between VPC's directly, using a *peering connection*. A VPC peering connection is a networking connection between two VPC's that enables the routing of traffic via private IPv4 or IPv6 addresses. Instances within either VPC can communicate with each other as if they are within the same network. Moreover, peering connections can be made between different AWS accounts. This is significant for future possibilities of the DBSE, enabling researchers to create new trading clients deployed in their own AWS accounts and to test them on a centralised exchange by requesting access through a peering connection.

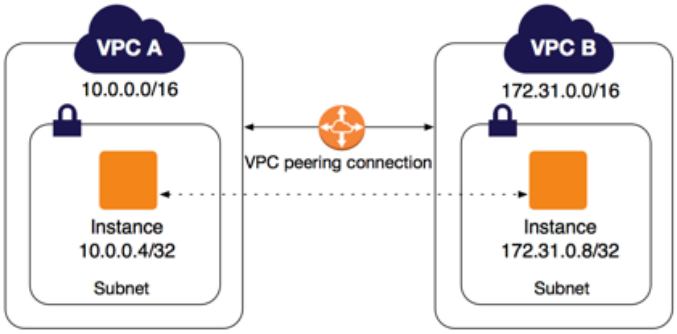



Figure 3.10: VPC Peering Connection, reproduced from AWS (2019) [29].

Figure 3.10 demonstrates the role a VPC peering connection plays in connecting two Virtual Private Cloud networks. The connection is possible because VPC A and B both have different CIDR blocks of *10.0.0.0/16* and *172.31.0.0/16* respectively. Since the Distributed Bristol Stock Exchange utilises three VPC's I had to create two peering connections. One between London and Ohio, the other between London and Sydney. Note that a third connection between Ohio and Sydney is not required as trading clients do not communicate with each other; traffic goes solely between a client and the exchange - located in London.

### Routing Tables

Once the peering connections were configured and active, I had to ensure all network traffic within each VPC would be routed correctly. VPC routing is configured via a *routing table* which defines destination/-target pairs of CIDR blocks and connection ID's, such as a peering connection ID or internet gateway ID. The routing table for the London VPC was the most complex, as it requires routing for its own internet gateway as well as both peering connections to Ohio and Sydney.

| Destination | Target | Status |
|---|---|---|
| 172.31.0.0/16 | local | active |
| 0.0.0.0/0 | igw-edc4fa84 | active |
| 172.32.0.0/16 | pcx-013839b457d3bd303 | active |
| 172.33.0.0/16 | pcx-02753e32c694db734 | active |

Figure 3.11: London VPC Routing Table.

Figure 3.11 displays the AWS routing configuration for the London VPC. As observed, all traffic within the London VPC with the destination IP *172.31.x.x* is classified as local and routed to itself. All traffic with the destination IP *172.32.x.x* or *172.33.x.x* will be routed through the Ohio and Sydney peering connections respectively. Finally, all remaining traffic is assumed to go via the Internet Gateway. This configuration ensures that all traffic is routed to the correct part of the network and to enable successfully communication between sub-networks. Similar routing tables were also configured for the Ohio and Sydney VPC networks.

### Elastic Compute Cloud (EC2)

Once the routing tables are configured, the majority of the networking configuration for the Distributed Bristol Stock Exchange was complete. It was now time to configure and deploy the compute resources for the exchange and trading clients into the appropriate Virtual Private Cloud networks. Amazon Web Service's compute resource is known as the Elastic Compute Cloud or EC2. The service provides an interface to provision virtual machines, known as instances, in the cloud of varying compute and memory performance. For the Distributed Bristol Stock Exchange, I wanted all trading clients to have equally

sized hardware. This was driven by the desire to test race-to-market through latency specifically, as such, I did not want one client to have a computational advantage on the speed in which they could compute a response to a market event. The exchange however, would be required to manage the requests, in this implementation, of up to four clients. As such, I decided to give the exchange both more memory and compute power. AWS is a business however, therefore to hire its services, you must pay. Within EC2, costs are calculated per hour per size of the machine, with the larger hardware costing more per hour. Despite this, to encourage developers to use AWS over competitor cloud providers, Amazon offers a variety of its services within a "free" tier. For EC2, AWS's free tier offers 750 hours per month of one of Amazon's smallest machines, called the *t2.micro*. The T2 tier is Amazon's lowest-cost general purpose instance type offering a balance of compute, memory and networking resources. The micro model has one virtual CPU and 1Gb of memory and as such seemed like the perfect compute resource for each trading client. For the exchange, I chose the *t2.medium* instance, which has two virtual CPUs and 4Gb of memory. Since the Distributed Bristol Stock Exchange is run on demand, these instances do not have to be running and accruing costs all the time. As a result, with moderate use, it is possible to deploy and use the DBSE entirely within Amazon's free tier. Conclusively, I configured a total of five instances across the three VPCs. One *t2.medium* for the exchange and two *t2.micro* for the trading clients in London, one *t2.micro* for the client in Ohio and one *t2.micro* for the client in Sydney.

### Security Groups

The final network configuration to complete for the Distributed Bristol Stock Exchange was *security groups*. A routing table deals with the forwarding of traffic throughout a network, however, it does not apply any security rules to determine whether an instance within the target VPC wants to accept that data or not. A *security group* acts as a virtual firewall and controls the inbound and outbound network traffic of an instance. The Distributed Bristol Stock Exchange communicates via a variety of different networking protocols, namely TCP and UDP. A security group by default rejects all inbound traffic and accepts all outbound, consequently, for the exchange and the trading clients to receive data, their instance's security group must be configured. I created two different security groups, one for the exchange and one for a trading client, and edited their inbound rules, shown in Figure 3.12 and 3.13 respectively.

| Type ⓘ | Protocol ⓘ | Port Range ⓘ | Source ⓘ |
|---|---|---|---|
| Custom TCP Rule | TCP | 8080 | 0.0.0.0/0 |
| SSH | TCP | 22 | 0.0.0.0/0 |
| Custom TCP Rule | TCP | 31001 - 31020 | 0.0.0.0/0 |
| All ICMP - IPv4 | All | N/A | 0.0.0.0/0 |

Figure 3.12: Exchange Inbound Security Group Configuration.

| Type ⓘ | Protocol ⓘ | Port Range ⓘ | Source ⓘ |
|---|---|---|---|
| SSH | TCP | 22 | 0.0.0.0/0 |
| All UDP | UDP | 0 - 65535 | 172.31.19.35/32 |
| All ICMP - IPv4 | All | N/A | 0.0.0.0/0 |

Figure 3.13: Trading Client Inbound Security Group Configuration.

Both security groups have a Secure Shell (SSH) policy to enable the simulation user to access the instances as well as both having an Internet Control Message Protocol (ICMP) for testing pings between instances. The security groups differ with regards to TCP and UDP traffic. On the exchange I have enabled TCP traffic for both port 8080, the configured web interface port for the exchange's FIX engine, and within the range 31001 and 31020, the exchange's accept ports for the trading clients. The trading clients have one differing policy which accepts all UDP traffic from the exchange's IP address, *172.31.19.35/32*, which

was allocated when creating the *t2.medium* instance previously.

With the creation and assignment of the security groups to the different instances, the networking and compute configuration for the Distributed Bristol Stock Exchange was complete. Figure 3.14 summarises all of the above and displays the networking architectural design of the DBSE on Amazon Web Services.

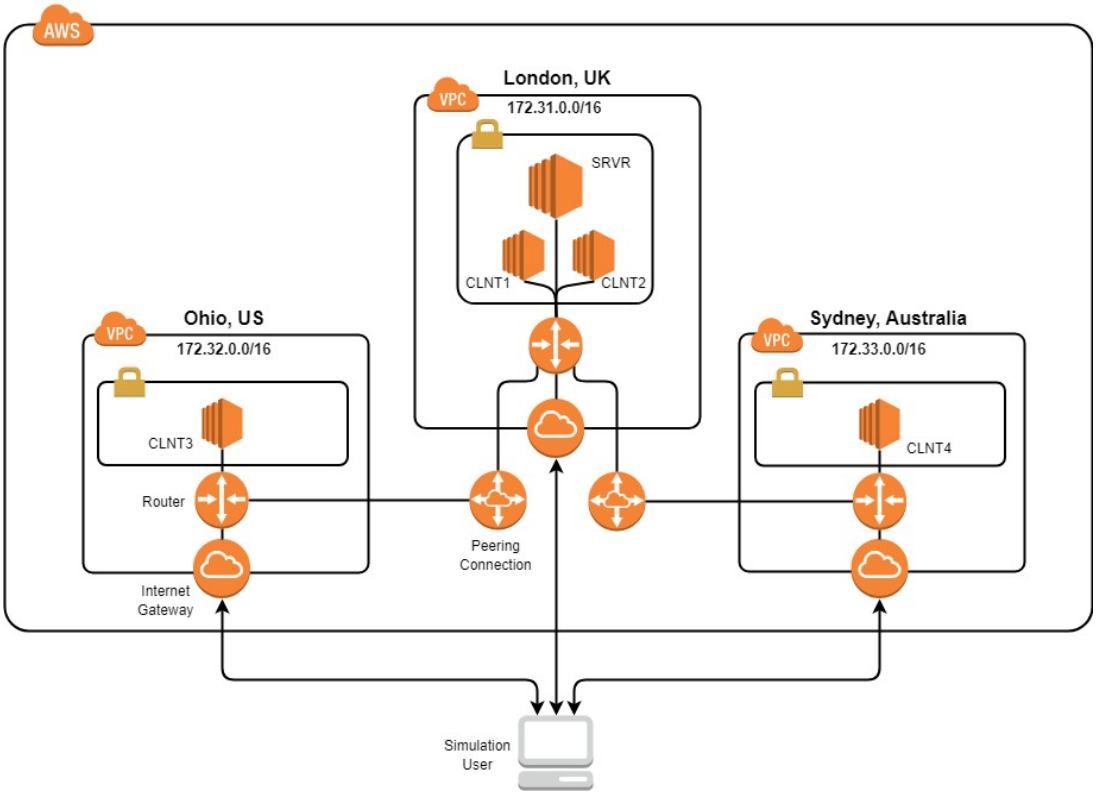



Figure 3.14: Distributed Bristol Stock Exchange AWS Configuration.

### 3.4.2 Deployment Challenges

When deploying the Distributed Bristol Stock Exchange into the cloud, there were a few notable challenges that required additional work, summarised below.

#### QuickFIX Compilation

The QuickFIX implementation of the FIX protocol is quite an extensive library that requires compilation once the source code has been downloaded. When I initially deployed my first trading client onto a *t2.micro* instance, the compilation of the QuickFIX library consistently ran out of memory and would throw an exception. The 1Gb of memory on the *t2.micro* instance was simply not enough to compile the source code required to build the QuickFIX library. To rectify this problem, I deployed the trading client to a larger instance type and compiled the QuickFIX source code on that machine. Once compiled, I created my own snapshot of the instance, known as an Amazon Machine Image (AMI). AMI's are the software loaded onto an instance during instantiation. By creating my own custom AMI, I could simplify the deployment of new trading clients as the AMI contained all of the trading client source code. Furthermore, I could deploy the pre-compiled QuickFIX source code back onto a smaller EC2 instance, specifically the *t2.micro*.

#### Multicast Unsupported

Whilst investigating the networking options of Amazon's Virtual Private Cloud I discovered that unfortunately Amazon does not support the use of the multicast addressing method. As such, my currently implementation for market data publication would not function in the cloud. I investigated the Google,

Microsoft and Oracle cloud platforms and none of them support multicast either. As a result, I reimplement the market data publisher and receiver to use UDP unicast. The exchange now maintains a list of all the client's internal IP addresses on the network, and then sends a UDP packet to each of them sequentially. This is an unfortunate consequence of cloud deployment as the use of unicast will result in the first IP address in the list having a small advantage as it is sent data first. Nonetheless, the multicast functionality remains within the code-base and could be reinstated if Amazon implements multicast routing in the future.

#### Synchronising Clients

One major challenge of having multiple clients is providing a mechanism to synchronise the execution of all of them. Without synchronisation, inconsistency can occur amongst simulation trials as each client will start and finish their trading day slight out of sync to each other. If clients were started out of sync, there would be fewer orders available at the start and end of the day for some clients as not all clients would be live. This can affect the profitability results of some agents, specifically the Sniper algorithm which is only active at the end of the trading day. To stop this, I implemented a mechanism to delay the start of a trading session to the next available minute. Provided the simulation user initialises all their clients within the time before the next minute elapses, all of the client's trading days will begin at the start of the next whole minute. Conclusively, my implementation ensures that each trading client begins and ends in synchronisation and thus has an equivalent opportunity to trade on the exchange throughout the trading day.

#### Amazon Time Sync

The final deployment challenge I faced occurred whilst testing the latency between exchange and trading clients in each region. When measuring time between two independent instances, a problem can arise where their time is out of sync. For example, when market data is published and timestamped by the exchange, if the receiving client instance does not use the exact same time then the latency value will be inaccurate. So much so that the receiving instance might think the market data was published in the future, giving a negative latency value. To solve this problem, I utilised an Amazon service called *Amazon Time Sync* [24]. Amazon Time Sync uses the Network Time Protocol (NTP) along with a fleet of satellites and atomic clocks to perfectly synchronise time in Amazon's data centers. Ubunutu instances do not use Amazon Time Sync by default, and thus to rectify my latency calculations I configured all my instances to use the network time provided by Amazon's atomic clocks.

# Chapter 4

# Evaluation

This thesis focuses on the design and implementation of a distributed financial exchange that aims to operate and communicate as a real-world exchange. As such, its evaluation is best done by scrutinising the different architectural and technological decisions I made throughout the project compared to a real exchange. Reviewed in Section 2.5, the Jane Street Exchange (JX) is a perfect example of how a distributed financial exchange should be architected, and thus will be the baseline for my evaluation. The Jane Street Exchange will represent an ideal approach, however throughout the project there were numerous technical and time limitations that made implementing a replica of that exchange infeasible. Understanding these limitations and evaluating the alternative options and decisions I made throughout the project will be the focus within this evaluation. Fundamentally, despite the enterprise real-world focus, the Distributed Bristol Stock Exchange is designed as a research and teaching tool, not a production-grade fully-implemented financial exchange. Despite this, this evaluation demonstrates the success of the DBSE as more than a simulation. The exchange of the DBSE is not an abstraction, it has been implemented as a deployable and working financial exchange that can accept clients through real-world protocols. In addition, I will reference the project's aims, outlined in Section 1.4, and evaluate their fulfilment. If all aims are satisfied, then I can quantifiably claim that the application is fit for purpose and that the work presented in this thesis has been a success. Finally, I will present the results of a race-to-market experiment that I conducted using DBSE as an example and evidence of what it can and could be used for.

## 4.1 Architecture Evaluation

Emphasised in the sections below are the main successes and limitations of the architectural design and implementation of the DBSE. I evaluate by comparing both the functionality provided and the architectural designs of JX and the BSE to the DBSE, whilst highlighting the overall aims of the project.

### 4.1.1 Language Selection

I began my implementation of the Distributed Bristol Stock Exchange by selecting a programming language best suited for this project. The main influencers in this decision was that Python is a simple to read language for new programmers, thus the DBSE could be understood by non-expert developers. Moreover, I felt it would be a poor use of time to re-write much of the existing functionality provided by the Bristol Stock Exchange in a more performant language. Since the DBSE required far more implementation than was originally expected, the majority of the codebase was rewritten by me from scratch. As a result, in hindsight I would have liked to have written the DBSE in a more suitable, efficient and typed language for trading applications such as C++. Despite my decision to use Python, the DBSE has been written in the latest version, Python 3.7, and most the application's function declarations have been typed. This makes the DBSE arguably easier to read and understand than the BSE, moreover because the application has been reorganised into a project-like structure with the codebase being divided into different files and directories based on functionality. Ensuring the DBSE was designed and implemented with scalability in mind was one of the stated aims of the project. This additional work improving the overall code quality and making the simulation publicly available on GitHub fulfils this aim and ensures that it can be subsequently used by future academics.

### 4.1.2 Micro-service Selection

The architectural designs of JX and the DBSE were illustrated in Figure 2.4 and Figure 3.7 respectively. Although, on first glance, it is noticeable that there are substantially more micro-services within JX than the DBSE; upon further analysis the functionality that both of these exchanges provide is not too dissimilar. At the heart of both exchanges is the matching engine, the service that maintains and processes all of the orders and trades that occur on the exchange. The DBSE's matching functionality has been improved compared to the Bristol Stock Exchange; now enabling orders to have quantities greater than one and enabling multiple trading clients and trading algorithms to have more than one order on the exchange simultaneously. This emphasises my understanding of the financial domain as I have enhanced the underlying matching engine functionality. Furthermore, this supports the DBSE's aim as a more realistic exchange implementation because it removes one of the underlying limitations of the BSE. Conclusively, this work fulfils the second aim of the project - "extend BSEs functionality to enable multiple orders per trader with no maximum quantity limit."

Comparing the client-side of the DBSE exchange to the JX architecture, it is observable that the DBSE successfully supports both the Client Port and Market Data Publisher micro-services. This real-world functionality is critical for a distributed exchange, and a key focus of this thesis, as otherwise it would be impossible for clients to place orders on the exchange or learn how the market is changing over time. Whereas the BSE could only support one trading client, the DBSE can support multiple. Currently, the DBSE is configured to support four independent trading clients that can each have different trader and order scheduling configurations. One major achievement of the DBSE is that it can be configured to support many more client ports; up to the networking and computational limits of the chosen server type running the exchange. These client ports must be configured in advanced and cannot be allocated dynamically - as specified by the QuickFIX documentation. Although this might seem as a criticism, it is in fact equivalent to real world exchanges, which carefully manage their customers and selectively decide and configure in advance which port a given client has access to. Therefore, the DBSE successfully implements an equivalent mechanism to real world exchanges for accepting and configuring clients. For market data publication, the DBSE provides this functionality in the same manner as the JX, however currently limited within Amazon's internal network. In the future, market data could be sent over the internet without requiring any code changes, however the appropriate networking infrastructure - in the form of a Virtual Private Network - would need to be configured within the AWS account hosting the exchange. This additional configuration was unnecessary for this thesis as the exchange and all the trading clients were hosted on Amazon servers. Nonetheless, the DBSE supports this functionality and thus too operates equivalently to a real financial exchange for market data publication. The implementation of the Client Port and Market Data Publisher services within the DBSE fulfils the third aim of the project - "develop a limit order book financial exchange application that can support multiple trading clients simultaneously."

Notably, missing from the client side of the DBSE exchange architecture compared to the JX architecture is the Trade Reporter and the Drop Port. The omittance of these two services from the DBSE is not significant in effecting the success of the project. A trade reporter is used to publish exchange activity to a trade reporting facility and the Drop Port is used to amalgamate client orders to a clearing firm for monetary exchange. Since the DBSE is not a commercially-operated exchange, a Trade Reporter is not required to notify other financial institutions of activity. Furthermore, since the DBSE is a moneyless exchange, drop ports are not required to facilitate financial settlement between buyer and seller. Conclusively, the DBSE achieves its goal as a financial exchange simulation, however I appreciate that without a trade reporter or drop port, the DBSE could not operate as a regulated exchange.

### 4.1.3 Compromising Exchange Decomposition

Despite the success of implementing the matching engine, client ports and market data publisher for the DBSE, in an ideal world they would operate as individual applications, hosted on independent servers within an exchange network. The JX achieves this via the UDP communication protocol and the multicast addressing method. It was an initial target of mine to implement the DBSE using this architecture, however decomposing the exchange is this way would be infeasible for multiple reasons. Unlike market data publication, where it is not essential that packets are delivered, if a packet is lost within an exchange network then several of a client's order requests could be dropped and left unprocessed. The User Datagram Protocol does not support guaranteed transmission, and consequently real-world financial exchanges implement sophisticated repeater services that maintain and manage all of the UDP packets sent within a

trading day via a series of sequence numbers. Implementing this repeater service, although feasible, would require extensive implementation time beyond the limits of the thesis. Regardless of time constraints, at the time of writing, the AWS Virtual Private Cloud does not support the multicast addressing method. Consequently, even if a repeater service had been successfully implemented, it could only be deployed on privately owned networking and compute hardware, not the cloud. The final architecture for the DBSE's exchange is a single application for the matching engine, client ports and market data publisher. Regrettably this does restrict the exchange to vertical scaling as additional client ports cannot be hosted on different machines. Furthermore, it results in more computation being executed on the same machine as the matching engine - a fundamental target to limit in real world financial exchanges. Nonetheless, given the number of trading clients utilising the exchange, as well as the time and monetary restrictions of this thesis, I consider the implemented architecture to be a suitable compromise. The DBSE's exchange still provides all of the functionality expected of an independent exchange at the small cost of performance and scaling capabilities.

### 4.1.4 Omitting Persistent Storage

At the start of the project, I decided to omit persistent storage from the implementation plan of the DBSE. This was driven by the fact that the DBSE remains a simulation and is not designed for high availability deployment - i.e. the simulation only needs to be run on-demand. In a distributed system, like the DBSE, this can cause an inconvenience to users if a trading client is terminated or restarted prematurely of the trading day. If a client is restarted, it will begin allocating new ClOrdID's from 0. This causes an inconsistency between the ID's used to reference orders on the exchange as there will be multiple orders from the same client with the same ClOrdID. If this occurs, the exchange and all clients would have to be restarted and the simulation re-run. This is not a major problem however as with suitable training users will know not to prematurely terminate clients, and even if it does occur, the DBSE can be restarted within a matter of seconds. To improve training and alleviate this problem, I implemented functionality for a client to appropriately cancel all of their live orders from the exchange at the end of the trading day and present a message to the user when the client is safe to terminate. I stand by my decision to not include persistent storage in the scope of this thesis, yet I recommend it as a future works for improving the overall quality of the Distributed Bristol Stock Exchange.

## 4.2 Technologies Evaluation

Two of the aims outlined in Section 1.4 were related to the successfully communication of financial data in a distributed system:

- Investigate and utilise the best communication protocol for clients to place orders and receive order updates from the exchange.
- Investigate and implement how an exchange publishes market data quickly and efficiently to interested parties.

Researching and selecting the best-suited communication technologies for this project was a constant challenge and thus it is essential to evaluate those decisions. For the two aims above, I utilised the Financial Information Exchange (FIX) protocol and UDP unicast respectively.

### 4.2.1 FIX

Deciding to use FIX as my main communication protocol for order placement and execution reports was a major design decision for this project. FIX is unarguably the communication protocol of choice in real-world finance; utilised by thousands of financial institutions and exchanges daily to facilitate trading data exchange. Understandably, for the FIX protocol to handle all aspects of financial trading in the real world, it supports a large and complex language of different messages. This is a noticeable disadvantage for its use within the Distributed Bristol Stock Exchange as FIX's messaging capabilities are far more extensive than what is required. In the latest versions of FIX, the protocol supports messages for all aspects of stock trading as well as other financial services, including bonds and foreign currency exchange. As a result, it could be argued that the protocol provides too much functionality that complicates the development of the DBSE. Instead, a more simplistic protocol could have been used and the messaging language customised for the DBSE's needs. The counterargument is that using FIX enriches the DBSE as

a teaching platform as it presents to students the real-world mechanisms that facilitates financial trading. Moreover, students can observe how the protocol operates, is implemented and provides an opportunity for them to experience creating trading clients of their own as a potential coursework assignment.

From a realism perspective, the decision to utilise the FIX protocol undoubtedly enhances the DBSE. FIX is the global trading protocol and thus the time required to send FIX messages on the DBSE should be close if not equivalent to that of real-world financial institutions - although this data is not publicly available. This supports the DBSE's overarching goal of being real-time and using real world tools wherever possible. Finally, regardless of whether FIX is too extensive or not, the selected protocol for the DBSE had to fulfil three main characteristics. It had to be bi-directional, full-duplex and must communicate over a single constant TCP connection. The FIX protocol supports all three of these characteristics, unsurprisingly as it was designed to support financial communication. As a result, the FIX protocol was definitely the best option as the main communication technology for order requests on the DBSE. Conclusively, the FIX protocol successfully fulfils the former of the two aims, enumerated at the start of Section 4.2 - "Investigate and utilise the best communication protocol for clients to place orders and receive order updates from the exchange."

### 4.2.2 UDP Unicast

The latter of the two aims from Section 4.2 focuses on the successful publication of market data from the exchange to trading clients. Utilised within the final deployment of the DBSE for these purposes are the UDP protocol and the unicast addressing method. This combination is close, but not an exact copy of what is used in the Jane Street Exchange or on other real-world financial exchanges. Ideally, the market data would be published using UDP multicast, to ensure efficient, non-duplicated traffic throughout the network. A compromise unfortunately had to be made because of Amazon Web Services not supporting the multicast addressing method. Since I was forced to use unicast, TCP was a consideration to replace UDP as it would guarantee message delivery. Upon evaluation however, TCP would require the exchange to manage connections between all trading clients, increasing its computational overhead. Moreover, in the event of a lost packet during UDP transmission, it does not cause a major issue to clients as they will just update their market data when a future packet is received. Despite the compromise of using UDP unicast rather than multicast, the DBSE still publishes market data successfully to clients positioned across the globe. The only consequence is that the exchange's publisher must iterate though each client in turn sending them their market data. This does not cause any issues at the current scale of the DBSE, thus far supporting four trading clients, each of which support several robot traders. In any case, the DBSE maintains an implementation of both unicast and multicast transmission if AWS starts to support multicast in the future or a user wishes to buy and maintain their own networking hardware for larger scale tests.

To analyse the success of the UDP unicast market data publisher and the distributed nature of the DBSE, I conducted an investigation into the varying latencies for clients throughout the globe. Specifically, with an exchange hosted in London, I timed how long it takes for trading clients in London, Ohio and Sydney to receive market data. I wanted to ensure that there was a disparity in latency depending on how far from the exchange a client was hosted. This was crucial as without varying latency, it would be impossible to test whether the profitability of a trading agent is dependent on its ability to race-to-market. To test the latency, I ran the DBSE with some additional timing code. When the exchange publishes market data it timestamps the message before sending it through the network. Thus, when each respective trading client receives the message it can perform a comparison between the time of arrival and the timestamp the message was sent - located within the message. By utilising the network atomic clocks provided by the Amazon Time Sync Service I could guarantee that all time on the network would be synchronised and thus the results would be accurate. Presented below is an experiment I ran for ten minutes recoding the latency to transmit market data from an exchange in London to four trading clients under standard simulation conditions.

During the ten-minute experiment, the simulation published market data 491 times. Summarised in Table 4.1, are the minimum, first quartile, median, third quartile and maximum latency timings, in milliseconds, for each of the four clients. CLNT1 and CLNT2 are both located in the London region, CLNT3 is positioned in Ohio, US and CLNT4 is hosted in Sydney, Australia. As expected, the results show that as geographical distance increases from the exchange, so does the latency. Consequently, clients located

| | Latency (ms) | | | |
|---|---|---|---|---|
| | **CLNT1** | **CLNT2** | **CLNT3** | **CLNT4** |
| MIN | 0.4 | 0.5 | 43.7 | 134.9 |
| Q1 | 0.7 | 0.8 | 44.0 | 135.2 |
| MEDIAN | 0.8 | 0.9 | 44.1 | 135.3 |
| Q3 | 1.0 | 1.1 | 44.3 | 135.5 |
| MAX | 2.9 | 1.8 | 55.0 | 138.5 |

Table 4.1: Table of results for the latency experiment.

in Australia receive market data from the exchange in a median time of 135.3ms compared to London's 0.8ms and 0.9ms. Interestingly, market data is received by CLNT2 0.1ms slower than CLNT1, even though they are located within the same region. This is likely because of using the unicast addressing method instead of multicast. The slight delay is likely as a result of the exchange sending each client data sequentially; thus, CLNT2 is usually sent data fractions of a millisecond after CLNT1.

The spread of the latency timings is relatively consistent amongst all four clients, although there are a couple of outliers that result in the high maximum values of 2.9ms and 55ms for CLIN1 and CLNT3 respectively. Shown in Figure 4.1, are the distributions of each client's latency binned into 0.1ms intervals.

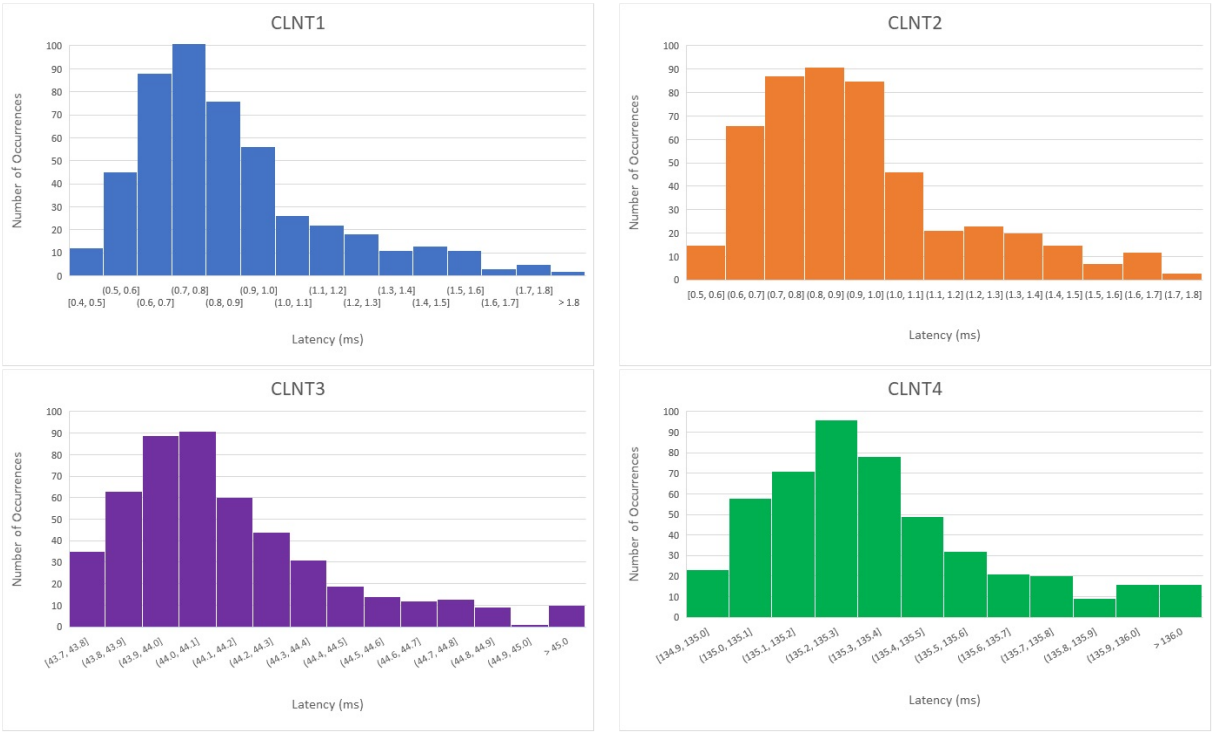



Figure 4.1: Spread of clients latency.

All four graphs have the same shape and the majority of latency is clustered within a 0.5ms spread. Table 4.2 presents the mean, variance and standard deviation of the timing experiment. These results show that CLNT1, CLNT2 CLNT3 and CLNT4 each on average receive market data 0.9ms, 1.0ms, 44.2ms and 135.4ms after the exchange publishes it. This was to be expected, as transmitting messages over increasingly greater distances should take longer amounts of time. However, the values for the variance and standard deviation of CLNT3, positioned in America, were unexpected compared to the other clients. Since all communication traffic was occurring within Amazon's internal network, I would have expected the variance and standard deviation of latency across clients to be consistent. Upon further analysis of the timing data, the larger spread of CLNT3 was caused because of a few outliers, the largest of which was 55ms. This gives insight into the amount of traffic Amazon's internal network is handling between London and Ohio, as these increased latencies suggest that Amazon handles more spikes in traffic between

London and Ohio, resulting in the increased latencies and thus variance.

| | Latency (ms) | | | |
|---|---|---|---|---|
| | **CLNT1** | **CLNT2** | **CLNT3** | **CLNT4** |
| MEAN | 0.9 | 1.0 | 44.2 | 135.4 |
| VARIANCE | 0.1 | 0.1 | 0.4 | 0.1 |
| STANDARD DEVIATION | 0.3 | 0.3 | 0.7 | 0.3 |

Table 4.2: Table showing the spread of the latency experiment.

UDP unicast has shown to be a viable option for transmitting market data within Amazon's network to clients positioned across the globe. UDP was the logical choice, compared to TCP, as it is fast, requires little computational overhead and is the protocol used by real world exchanges. Despite being restricted to the unicast addressing method, the DBSE successfully handles millisecond speed with the current configuration of trading clients. Resultantly, the aim set to investigate and implement how an exchange publishes market data quickly and efficiently to interested parties has been a resounding success and another accomplishment of this work.

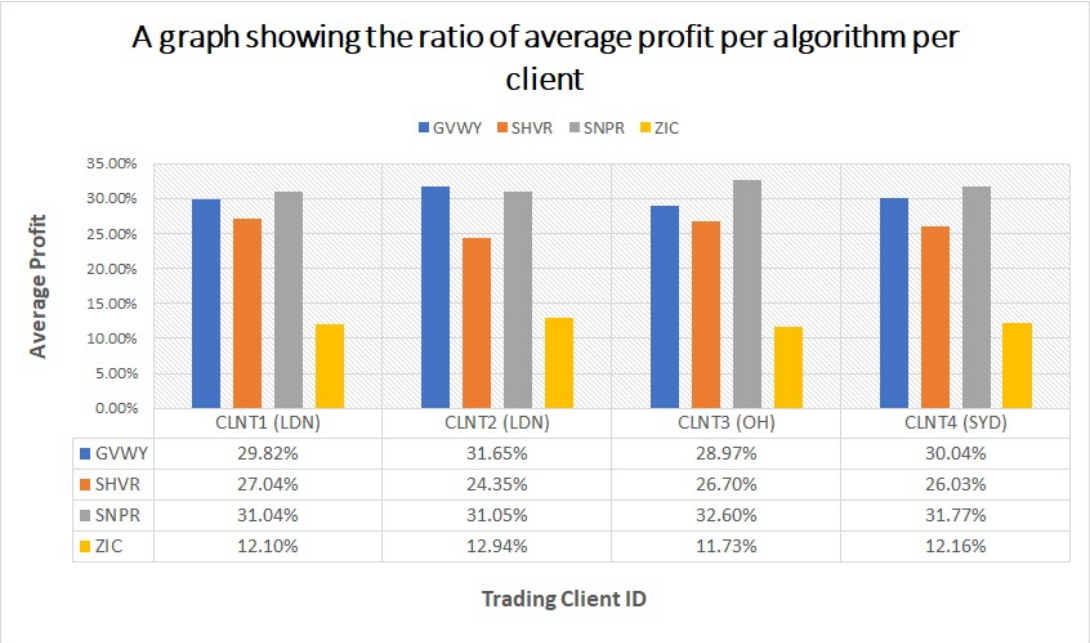
## 4.3 Race-to-Market Experiment

To demonstrate the capability of the Distributed Bristol Stock Exchange as a real-time and real latency simulation I conducted a race-to-market experiment. As previously mentioned, race-to-market is a concept by which a trader can "steal the deal" if they learn and respond to a market change before a competitor. Therefore, in a real-world scenario, if a trading client is positioned further away from the exchange than another trading client, then it takes longer for that client to receive market data. Consequently, the closer of the two clients can react faster to market events and therefore should be more profitable.

I constructed an experiment on the globally deployed DBSE with four configured clients, two in London, one in Ohio and one in Sydney. The experiment would consist of a total of 160 trading agents across the four clients. These trading agents were split 50/50 between supply and demand as well as 25/25/25/25 between the four implemented trading algorithms, Giveaway (GVWY), Shaver (SHVR), Sniper (SNPR) and Zero-Intelligence Constrained (ZIC). As such for each trading client, there were five agents of each robot type on the supply side and five agents of each robot type on the demand side. Totalling 40 agents per client, totalling the 160 agents for the simulation.

Each of the four trading clients were given equivalent order scheduling configuration that ran for a total of three minutes. The order schedulers were configured to distributed new orders in 30 second intervals, with the *drip-poisson* timemode. Drip-poisson is the most realistic distribution method as the inter-arrival times of orders follows a Poisson distribution. Within the three-minute simulation, the range of prices for both the supply and the demand are configured to change every minute. Initially, at time t=0, the supply and demand are configured to sit in the range 100-200 cents. At time t=60, the range increases to 150-250 cents, before returning to the initial range, 100-200 cents, at time t=120. I configured the stepmode of each range to be *fixed*, this results in the DBSE creating an even spread of orders across the price range - resulting in an equilibrium price of 150, 200 and 150 cents for each minute of the simulation respectively. It is common practice in experimental economics to configure simulations in this way; changing the equilibrium price at a set point in time causes a *shock change* in the market prices which is an excellent way to test the reactiveness of trading agents. This is because in the real world, transaction prices are constantly changing depending on the world's events. If the supply and demand curves of the simulation were configured to be constant then so too would be the equilibrium price, thus the simulation would be stale. The full simulation configuration for this experiment can be found in Appendix B.

I repeated the three-minute experiment ten times and for each run recorded the total profit of each trader type. Presented in Figure 4.2 are the average profits for each robot trader type per client over the ten runs. For this specific order scheduling configuration, the results show that the Giveaway, Shaver and Sniper traders all performed roughly equivalently across clients, with the Zero-Intelligence Constrained algorithm performing the poorest. These results show that regardless of distance from the exchange,




| | CLNT1 (LDN) | CLNT2 (LDN) | CLNT3 (OH) | CLNT4 (SYD) |
|---|---|---|---|---|
| GVWY | 29.82% | 31.65% | 28.97% | 30.04% |
| SHVR | 27.04% | 24.35% | 26.70% | 26.03% |
| SNPR | 31.04% | 31.05% | 32.60% | 31.77% |
| ZIC | 12.10% | 12.94% | 11.73% | 12.16% |

Figure 4.2: Graph showing the ratio of total profit per trader type for each client.

each algorithm performs equivalently in each region compared to its counterparts. Figure 4.3 on the other hand compares the total profits of all of the algorithms per client. The results presented here are particularly interesting as they demonstrate that on average CLNT1 and CLNT2 outperformed CLNT3, which outperformed CLNT4 - within standard error. This proves my hypothesis correct, as CLNT1 and CLNT2 are positioned closest to exchange, followed by CLNT3, followed by CLNT4. Although the average profits of each client are close, there is a significant difference with CLNT2 in London earning 25.72% of profit compared to CLNT4 in Sydney, Australia earning 24.10% profit. If latency did not affect the profitability of trading agents and their ability to race-to-market, then we would have expected each client to perform equivalently and each earn 25% of profit across the simulation. These results show that latency is a limiting factor in the profitability of agents. Designing new trading agents is a constant challenge, because as agents are implemented to have more "intelligence", thus more computationally demanding, the delay in which they can race-to-market increases. The trading agents currently available in the DBSE are all relatively computationally undemanding. The DBSE provides an opportunity to test more extensive agents such as ZIP, GDX or AA in the future and to determine whether their computational demand is at the cost of their reactiveness to market events.

The results of this experiment have proven that there is much to learn about algorithmic trading when you have a simulation that can offer real-time and real-latency analysis. The Distributed Bristol Stock Exchange enables such analysis and can be configured to enable researchers to uncover new insights into latency driven simulations. The DBSE has fulfilled all of the aims set in Section 1.4 of this thesis. Based on this evaluation, I think it is reasonable for me to claim that the project has been a resounding success given the time and resources available.

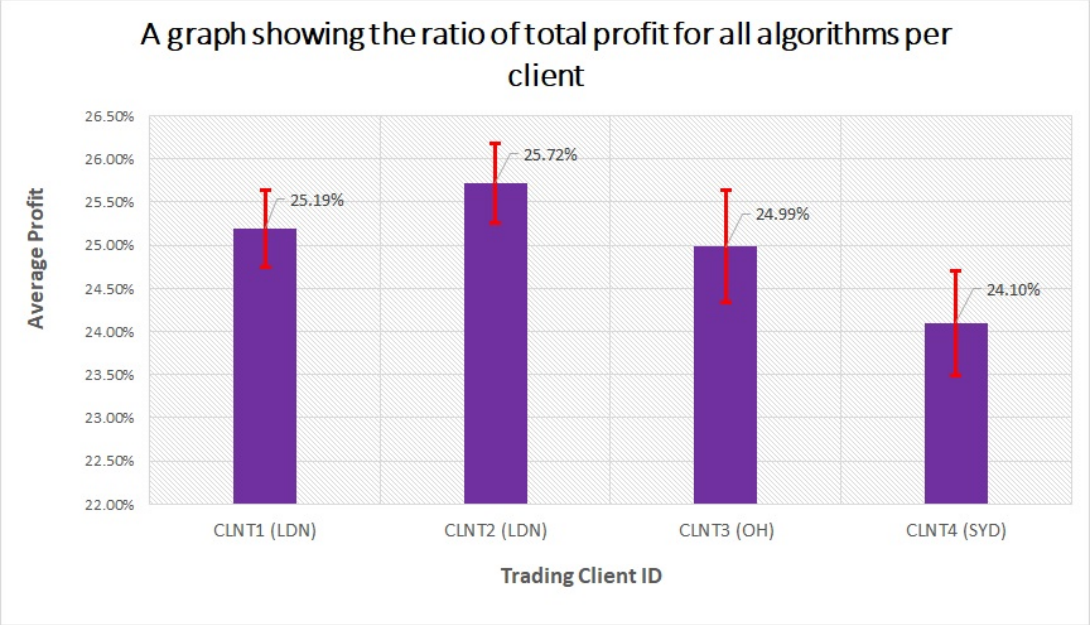



Figure 4.3: Graph showing the ratio of total profit across clients.

# Chapter 5

# Conclusion

The Distributed Bristol Stock Exchange is a globally distributed limit order book financial exchange simulation for research and teaching. Expanded from the original work by Cliff, the DBSE has taken the Bristol Stock Exchange from a single file and single threaded application into a distributed simulation that can be run on widely available commercial cloud-computing services. Trading clients can be configured and positioned around the globe; trading simultaneously on a single stock exchange. Where the Bristol Stock Exchange assumed absolutely zero-latency, the DBSE operates using real-world financial communication protocols that are designed to minimise latency but which do not disregard it. The Distributed Bristol Stock Exchange has already demonstrated its capability in enabling research aimed at understanding race-to-market trading and now can be offered as a platform for further research into latency arbitrage, a heavily under-researched topic in financial trading.

## 5.1 Contributions and Achievements

The overall aim, as described in Section 1.4, was to architect and implement a new more realistic financial trading simulation, using real-world technologies, that could be deployed in the cloud. This aim has been categorically achieved with the implementation and cloud deployment of the exchange and trading client applications that make up the Distributed Bristol Stock Exchange. Previously having no experience in financial trading or financial technologies, the learning curve for this project was extremely challenging. Throughout the project I spent substantial time researching and trialling different web-based architectures and communication protocols that would best suit financial trading. Many of my preliminary assumptions and implementations were unsuccessful, notably my initial Flask web application that utilised RESTful HTTP as the main communication protocol. I identified through extensive research that more sophisticated networking infrastructure and protocols would be required to develop a real-time real-latency simulation. The preceding two applications that form the DBSE are built upon the FIX protocol for order placement and the UDP for market data publication. As evidenced in my research of the Jane Street Exchange, these are the two protocols used every day to facilitate millions of financial trades around the world. The utilisation and implementation of these real-word technologies in this thesis is a conclusive demonstration of my understanding of web-based networking in the financial domain and remains the overarching achievement of this project. In addition to the DBSE implementation, I have designed and configured a globally connected network using the cloud services provided by Amazon Web Services. One exchange and four clients have been deployed within three geographical regions: London; Ohio; and Sydney. They are hosted on five separate instances, within three virtual private clouds that all route messages and communicate via two peering connections. Academics wanting to run and test their own simulations can replicate the network configuration outlined in this thesis on their own AWS account - all within Amazon's free-tier. With appropriate configuration, new trading clients can be created in any of the 21 regions around the world. This process is simplified by the Amazon Machine Image that I created for a DBSE trading client. Over the course of this thesis I have created three independent applications, all available on GitHub, and configured an assortment of networking infrastructure. This project, architecting and implementing a financial exchange, has required an assortment of computer science disciplines including financial technologies, concurrent computing, networking and cloud computing. My work presented here enables researchers in financial trading to simulate exchanges in real-time and is a significant contribution towards the goal of simulating latency arbitrage. I have achieved all of the aims presented in Section 1.4 and with this work plan to submit a paper, based on this thesis,

to the International Multidisciplinary Modeling and Simulation Multiconference to be held in Lisbon in September 2019 [17].

## 5.2 Project Status

The DBSE consists of two independent applications, the *dbse_ exchange* and the *dbse_ trading-client*, available for download from:

- `https://github.com/bradleymiles17/dbse_exchange`
- `https://github.com/bradleymiles17/dbse_trading-client`

The codebase has been written in the latest version of Python, Python 3.7 and function declarations have been typed to assist readability for new users of the DBSE. Both applications use an argument parser when executing, thus when attempting to run the application a user can view the required and optional parameters via the help, *-h*, flag.

The DBSE exchange operates as an independent limit order book financial exchange that can be configured to accept multiple trading clients. The exchange currently supports the ability to place and cancel orders as well as return execution reports via the Financial Information Exchange (FIX) protocol. Market data can be published by either UDP unicast or UDP multicast depending on the user's choice, although limited to unicast if deploying to Amazon Web Services. The exchange now supports orders of quantities greater than one, and the matching engine correctly fills or partially fills those orders according to trade activity. The exchange, equivalently to the BSE, assumes only one tradable commodity, but this could be updated relatively easy given the functional decomposition of the implementation.

The DBSE trading client supports four different trading algorithms, Giveaway, Zero-Intelligence Constrained, Shaver and Sniper. Each trading agent now supports multiple customer orders and appropriately decides whether it wants to place one them onto the exchange whenever it receives new market data. Trading clients can be configured in a variety of ways, with different permutations of trading agents and order schedulers, depending on the user's discretion. Within my own AWS account, I have one exchange deployed in London as well as four clients, two located in London, one in Ohio and one in Sydney. This thesis describes how Amazon's Virtual Private Cloud infrastructure can be configured to recreate the deployment configuration in other AWS accounts. The DBSE is substantially more advanced than the legacy Bristol Stock Exchange and provides a real-time globally distributed platform for academics to run trading simulations. The codebase for the DBSE is ready and publicly available for researchers and teachers to use, and I strongly encourage that they do so.

## 5.3 Future Work

At the start of this project, I spent a large amount of time planning and proposing the potential and long-term scope of this work. The overarching aim would be to develop a platform that could model latency arbitrage. This would require extensive work, requiring multiple exchanges, a trade reporting facility between exchanges and an entirely new trading client that could connect to and place orders on multiple exchanges simultaneously. Included in this work would be an expansion of the FIX messages that the current DBSE exchange supports, such as the Order Replace Request <G>, used to amend orders live on the exchange. Moreover, a persistent storage mechanism, for which I suggest a relational database, would benefit the exchange enabling it to be hosted permanently in the cloud. This work would likely take years to complete for a single developer, but I believe could be a perfect opportunity for PhD study.

As a part of this future work, I propose a new overview AWS architecture, shown in Figure 5.1. This diagram does not include networking infrastructure but gives an overview of the simulation's compute hardware and introduces a new proposed application, the *web client*. Currently, it is inconvenient for users of the DBSE to be required to SSH onto the simulation's hardware to run experiments. The web client would be a web-based application that acts as a simulation controller, hosted permanently in the cloud, that has the permission to orchestrate the instantiation, termination and synchronisation of trading clients across the network. Protected behind a user access control system, provided by AWS

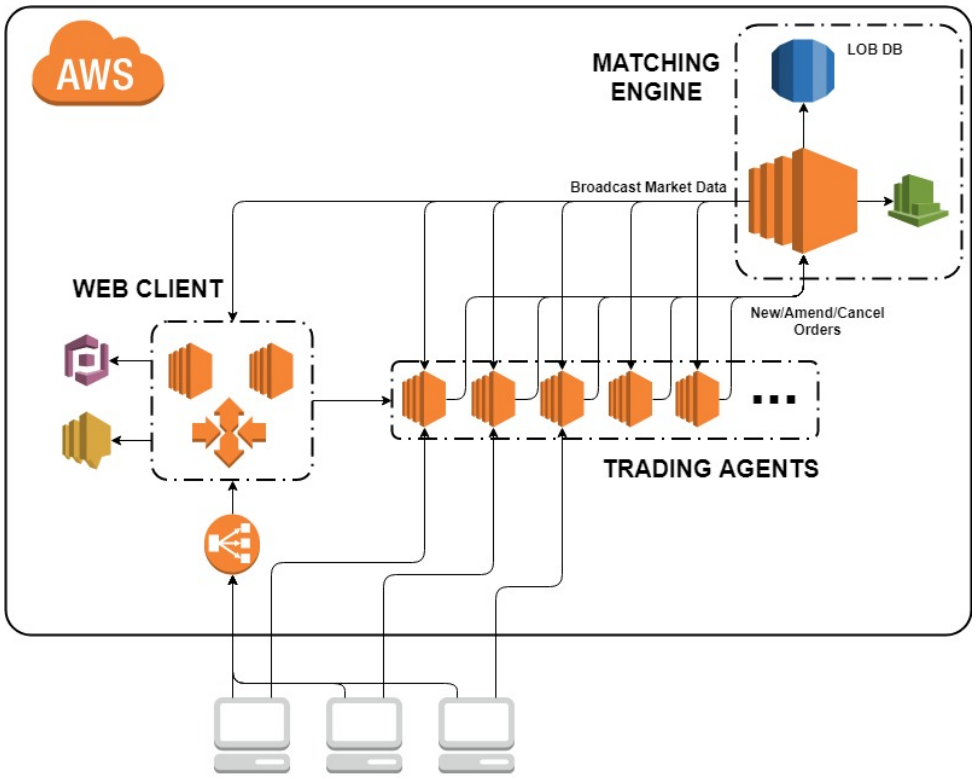



Figure 5.1: Distributed Bristol Stock Exchange proposed AWS architecture.

Cognito [25], the web client would enable easy and efficient configuration of simulation runs in a graphical interface. Upon completion of a simulation, it would amalgamate the results, terminate the unneeded trading clients and provide suitable tools for analysing the results. It is this proposed architecture that would be suitable as a potential enterprise project and business proposal. There are numerous financial institutions that desire the ability to run trading simulations of robotic agents but do not wish to design and implement their own bespoke systems since the development costs are extensive. The DBSE has the potential to be an easy-to-use simulation for non-developers, both in the academic and business worlds, and I am intrigued to see where it is developed in the future.

## 5.4 Final Conclusions

Presented in this work is the design and implementation of a globally distributed limit order book financial exchange for research and teaching. The outcome of this thesis has most definitely been a success with the completion of all of the aims outlined in Section 1.4. The Distributed Bristol Stock Exchange serves as a new platform for academics enabling them to run latency driven simulations and potentially uncover new insights within the field of financial trading. For teaching, the DBSE enables students to learn about trading agents within a controlled and monetary-risk-free environment, but moreover learn about the real-world technologies that facilitate financial trading. This research is an exciting field and has a wide variety of scope for the future. Latency arbitrage in most trading simulations is impossible to simulate due to them not supporting real latency. This thesis would be perfect to expand into a programme of PhD study, extending my implementation to support multiple exchanges and enable fine-grained latency arbitrage analysis. This thesis is a novel contribution to the field of latency-sensitive financial trading simulations, and thus I aim to submit a paper summarising this work to the International Multidisciplinary Modeling and Simulation Multiconference to be held in Lisbon in September 2019.

# Bibliography


[1] Canalys. Cloud market share q4 2018 and full year 2018. https://www.canalys.com/newsroom/cloud-market-share-q4-2018-and-full-year-2018, Feb 2019.

[2] S Chaudhary. Simulating a financial exchange in scala. http://falconair.github.io/2015/01/05/financial-exchange.html, Jan 2015.

[3] D Cliff. Minimal intelligence agents for bargaining behaviours in market-based environments. http://www.hpl.hp.com/techreports/97/HPL-97-91.pdf, 1997.

[4] D Cliff. Bristol stock exchange. https://github.com/davecliff/BristolStockExchange, Oct 2012.

[5] D Cliff. Bse: A minimal simulation of a limit-order-book stock exchange. In Michael Affenzeller, Agostino Bruzzone, Emilio Jimenez, Francesco Longo, Yuri Merkuryev, and Miquel Angel Piera, editors, *30th European Modeling and Simulation Symposium (EMSS 2018)*, pages 194–203, Italy, 10 2018. DIME University of Genoa.

[6] D Cliff. An open-source limit-order-book exchange for teaching and research, Nov 2018.

[7] FIX Trading Community. Financial information exchange protocol. https://www.fixtrading.org, 1992.

[8] FIX Trading Community. Fix family of standards. https://www.fixtrading.org/standards, 2019.

[9] FIX Trading Community. Fix overview. https://www.fixtrading.org/overview, 2019.

[10] CTA. Securities information processor. https://www.ctaplan.com/index, 2018.

[11] Rajarshi Das, James E. Hanson, Jeffrey O. Kephart, and Gerald Tesauro. Agent-human interactions in the continuous double auction. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'01, pages 1169–1176, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[12] M De Luca, CS Szostek, J Cartlidge, and Dave Cliff. *Studies of Interaction Between Human Traders and Algorithmic Trading Systems*. Number DR13 in Foresight Report - The Future of Computer Trading in Financial Markets. UK Government Office for Science, 9 2011. Publisher: UK Government Office for Science.

[13] R Engels. Financial exchange written in go, designed for algorithmic trading tests. https://github.com/robaho/go-trader, Jan 2019.

[14] I Fette and A Melnikov. The websocket protocol. https://tools.ietf.org/html/rfc6455#page-68, Dec 2011.

[15] Django Software Foundation. Django. https://www.djangoproject.com, 2019.

[16] Dhananjay K. Gode and Shyam Sunder. Allocative efficiency of markets with zero-intelligence traders: Market as a partial substitute for individual rationality. *Journal of Political Economy*, 101(1):119–137, 1993.

[17] I3M. International multidisciplinary modeling and simulation multiconference. http://www.msc-les.org/conf/i3m2019/, 2019.

[18] O Miller. Quickfix. `http://www.quickfixengine.org`, 2014.

[19] O Miller. Quickfix: Creating your application. `http://www.quickfixengine.org/quickfix/doc/html/application.html`, 2014.

[20] O Miller. Quickfix c++ fix engine library. `https://github.com/quickfix/quickfix`, Mar 2019.

[21] Armin Ronacher. Flask. `http://flask.pocoo.org`, 2019.

[22] John Rust. Behavior of trading automata in a computerized double auction market. *The Double Auction Market Institutions, Theories, and Evidence*, pages 155–198, 1992.

[23] SEC and CFTC. Findings regarding the market events of may 6, 2010. `https://www.sec.gov/news/studies/2010/marketevents-report.pdf`, Sep 2010.

[24] Amazon Web Services. Introducing the amazon time sync service. `https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-the-amazon-time-sync-service`, 2017.

[25] Amazon Web Services. Amazon cognito. `https://aws.amazon.com/cognito/`, 2019.

[26] Amazon Web Services. Amazon virtual private cloud. `https://aws.amazon.com/vpc`, 2019.

[27] Amazon Web Services. Aws: Global infrastructure. `https://aws.amazon.com/about-aws/global-infrastructure`, 2019.

[28] Amazon Web Services. Vpc with a private subnet only and aws site-to-site vpn access. `https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Scenario4.html`, 2019.

[29] Amazon Web Services. What is vpc peering? `https://docs.aws.amazon.com/vpc/latest/peering/what-is-vpc-peering.html`, 2019.

[30] Vernon L. Smith. An experimental study of competitive market behavior. *Journal of Political Economy*, 70(2):111–137, 1962.

[31] Statistica. Total size of the public cloud computing market from 2008 to 2020 (in billion u.s. dollars). `https://www.statista.com/statistics/510350/worldwide-public-cloud-computing`, 2019.

[32] Jane Street. How to build an exchange. `https://www.janestreet.com/tech-talks/building-an-exchange`, Feb 2017.

[33] Wikipedia. Unicast. `https://en.wikipedia.org/wiki/Unicast`, 2019.

# Appendix A

# FIX Configuration

## A.1 Example Exchange Configuration

```
[DEFAULT]
BeginString=FIXT.1.1
SenderCompID=SRVR
ConnectionType=acceptor
StartTime=00:00:00
EndTime=00:00:00
ResetOnLogon=Y

# Validation
UseDataDictionary=Y
TransportDataDictionary=./fix/FIXT11.xml
DefaultApplVerID=./fix/FIX50SP2.xml
CheckLatency=N

# Logging
FileLogPath=./fix/logs

# Storage
FileStorePath=./fix/sessions

# Misc
HttpAcceptPort=8080


[SESSION]
TargetCompID=CLNT1
SocketAcceptPort=31011

[SESSION]
TargetCompID=CLNT2
SocketAcceptPort=31012

[SESSION]
TargetCompID=CLNT3
SocketAcceptPort=31013

[SESSION]
TargetCompID=CLNT4
SocketAcceptPort=31014
```

## A.2 Example Client Configuration

```
[DEFAULT]
BeginString=FIXT.1.1
TargetCompID=SRVR
ConnectionType=initiator
StartTime=00:00:00
EndTime=00:00:00

# Validation
UseDataDictionary=Y
TransportDataDictionary=./fix/FIXT11.xml
DefaultApplVerID=./fix/FIX50SP2.xml
CheckLatency=N

# Initiator
ReconnectInterval=30
HeartBtInt=30
LogonTimeout=30
ResetOnLogon=Y

# Logging
FileLogPath=./fix/logs

# Storage
FileStorePath=./fix/sessions

# Misc
HttpAcceptPort=8081


[SESSION]
SenderCompID=CLNT1
SocketConnectPort=31011
SocketConnectHost=172.31.19.35
```

# Appendix B

# Example Simulation Configuration

```
{
    "traders": {
        "buyers": {
            "GVWY": 5,
            "ZIC": 5,
            "SHVR": 5,
            "SNPR": 5
        },
        "sellers": {
            "GVWY": 5,
            "ZIC": 5,
            "SHVR": 5,
            "SNPR": 5
        }
    },
    "order_schedule": {
        "demand": [
            {
                "from": 0,
                "to": 60,
                "ranges": [{"min": 100.0, "max": 200.0}],
                "stepmode": "fixed"
            },
            {
                "from": 60,
                "to": 120,
                "ranges": [{"min": 150.0, "max": 250.0}],
                "stepmode": "fixed"
            },
            {
                "from": 120,
                "to": 180,
                "ranges": [{"min": 100.0, "max": 200.0}],
                "stepmode": "fixed"
            }
        ],
        "supply": [
            {
                "from": 0,
                "to": 60,
                "ranges": [{"min": 100.0, "max": 200.0}],
                "stepmode": "fixed"
            },
            {
                "from": 60,
                "to": 120,
                "ranges": [{"min": 150.0, "max": 250.0}],
```

```
                "stepmode": "fixed"
            },
            {
                "from": 120,
                "to": 180,
                "ranges": [{"min": 100.0, "max": 200.0}],
                "stepmode": "fixed"
            }
        ],
        "interval": 30,
        "timemode": "drip-poisson"
    }
}
```