Imperial College London

3rd Year Software Engineering Group Project

Imperial College London

Department of Computing

Benchmarking Suite for High-Frequency C++ Trading Systems

Authors:
Kamil Bujel
Dimitri Chamay
Jake Dickie
Oliver Federico
Panagiotis Pachiyannis
Peter Christofides Paton
Aris Zhu Yi Qing

Supervisor:
Dr Paul Bilokon

January 10, 2022

Executive Summary

Low-latency trading, which is often referred to as *High Frequency Trading* (HFT), is estimated to contribute around 60% of total trading volumes in the US [1]. In today's world of open data, it is low-latency software that makes or breaks HFT companies simply because "a microsecond is an eternity" [2].

When developing a low-latency system, it is essential to be able to evaluate it quickly and consistently. The key to creating a reliable and performant system is to have a breakdown of where the code is slowest, so as to iteratively make changes to those parts and subsequently re-evaluate the performance.

It is no different in the world of low-latency trading systems. While some particular design patterns are common in the industry, none of them are well-documented or explored. One of the reasons for this is the high barrier to entry — any researcher would first need to develop a fully-functional trading system and configure it to work well with benchmarking and profiling tools. Another reason is secrecy, which is common in this industry.

This project aims to remove this barrier. We have collaborated with a leading low-latency trading systems provider, qSpark, and present a fully-functional, low-latency trading system. This comes together with a customisable and extensible framework to easily benchmark and profile any part of a trading system, including new trading strategies.

This trading system automatically collects real-time data from exchanges, runs a series of algorithms and places orders onto the cryptocurrency market. It operates on a nanosecond scale, which in itself was quite a challenge to achieve. To substantially improve the speed of the latency-sensitive part of the system (*hotpath*), we had to optimise both the order processing system and the trading algorithms.

This has led to our systems also containing a plethora of low-latency design patterns and algorithms which serve as examples of potential improvements and how to evaluate them. These can be followed to further assess the latency impact of a range of C++ design patterns, the original basis of the project, with rapid comparison being made available through a visual and instructive PDF report.

In addition, the entire system is multi-threaded, which allows a user to add new strategies or exchanges to be processed efficiently and independently, and has also contributed to the challenge of the development.

Our algorithms also maintain a record of all current orders across all exchanges on the given market (*multi-layer order book*), which allows for the buy/sell decisions to be more realistic and based across a variety of signals. This has required further optimisations to ensure efficient use of the data structures.

As the goal of the project is to make it easy for users to benchmark their own trading strategies, we have made every part of the system interfaced. That means it is easily replaceable with user-defined code, if desired, allowing researchers to evaluate the impact of different low-latency design patterns in C++.

The integration of several different benchmarking tools into this trading system offers a solution that can readily be used in the future by developers without the barrier of having to learn the technicalities of each tool or build their own system.

We look forward to this project being used in the future to further analyse C++ design patterns for low latency, as well as for developers to enhance their own systems.

Contents

Acknowledgements

We would like to extend our gratitude to our supervisor, Dr Paul Bilokon, who has guided us throughout this project, providing us with extensive ideas and materials to help us develop the project into what it is now.

We would also like to thank Paul for introducing us to several of his contacts including qSpark, a leading provider of ultra low-latency trading platforms for high-frequency algorithmic trading. Specifically, Anna Arad, Nataly Rasovsky, Nimrod Sapir and Erez Shermer aided us greatly in developing a product attuned to the needs of users which can be used in the future by other developers.

Finally, we would like to to thank Paul for all the time he dedicated to helping us and we look forward to working with him again in the future.

1 Introduction

A low-latency system is a program that is optimised to process a high volume of data with the smallest possible time delay, *latency*. It is used in many daily applications where we want near real-time communication or processing such as online meetings, gaming and trading, the latter of which is the focus of our project.

Low-latency trading, which is often referred to as *High Frequency Trading* (HFT), is estimated to contribute around *60%* of total trading volumes in the US [1]. In this context, there is no latency threshold that needs to be achieved. In today's world of open data, it is low-latency software that makes or breaks HFT companies simply because "a microsecond is an eternity" [2].

While developing low-latency systems, especially those in C++, there are many different language features and design patterns used to optimise the latency. These patterns are scattered around the internet[3] and spoken about at C++ conferences[2], but there is no definitive source of information which explains and compares them. This limits their use because developers struggle to have clear expectations of what improvements a particular change might bring. There is simply no systematic way in which these low-latency design patterns are presented and quantified — our initial goal was to change this.

Early on, we found that there are numerous tools and technologies a developer is required to understand before being able to benchmark their code. This is a high barrier-of-entry because a researcher needs a working trading system as well as knowledge of multiple benchmarking and profiling tools. Through our discussions between our supervisor, the Group Project team and qSpark, our partner and a leading low-latency high-frequency trading solutions provider, we have decided to slightly modify our approach. We have instead developed a *framework to benchmark low-latency design patterns in the context of high-frequency trading*.

To understand the needs of potential users, we built our own fully-functional trading system operating on a nanosecond scale[1]. This gives future researchers a solid base to build upon and provides flexibility in either testing design patterns or the latency of any given code. The ease of setup and the extensive automated latency reports mean that this tool can be used to fill the information void in the world of low-latency systems and encourage more research in this field.

[1] True HFT in cryptocurrencies is not yet realizable because the asset class does not support colocation; network latencies (usually on the scale of milliseconds) dwarf the latencies due to software.

2 Design and Implementation

2.1 Overview

The high level architecture of a trading system is presented in Figure 1.

Market data is received from the multiple exchanges the system subscribes to using WebSocket protocol. There is one thread running per exchange, ensuring that each data stream is processed independently and in parallel.

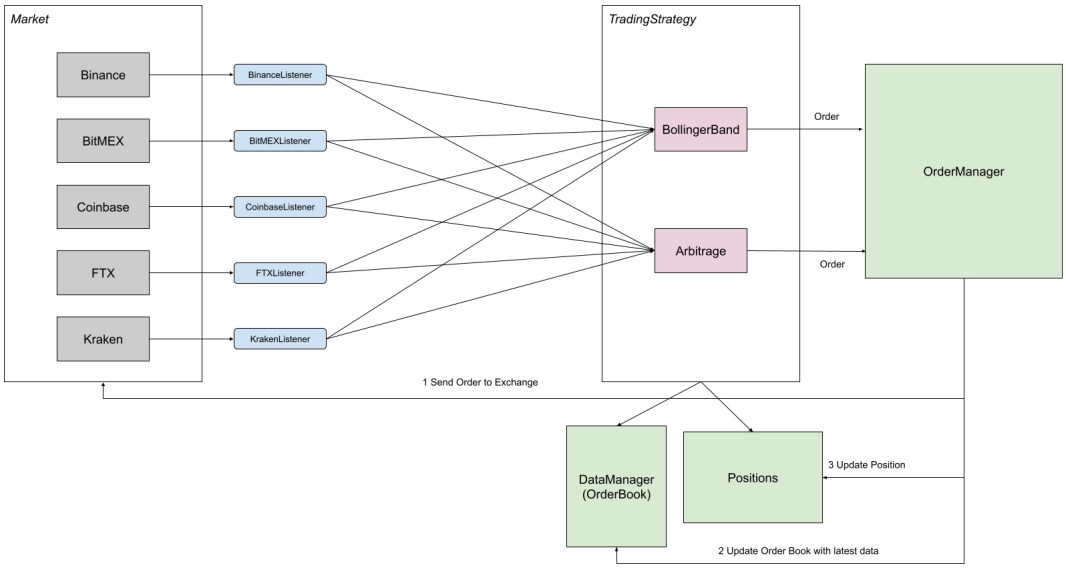


Figure 1: Diagram of the system architecture. On the left we can see listeners to market data from each of the exchanges, with each running on a separate thread. On each update, these pass on the data to all *TradingStrategies*, which decide whether to trade or not. If we would like to trade, we generate an *Order* and pass it on to the *OrderManager*, which sends the *Order* to the respective *Exchange*. We then update our *DataManager* with the latest data we have just received from the exchange and log the newly submitted order in the *Positions* data structure.

The data is then passed to the Trading Strategies, which use their own logic to decide whether or not an order[2] is sent to an exchange. These orders are sent to exchanges via an Order Manager.

After the order is submitted, we update our Position to record the latest trades and also update our Order Book with the latest received data.

Throughout this report, we will refer to the term *hotpath*. It is commonly understood to be the part of the code which is intended to be executed with a latency that is as low as possible. In our case, we measure the hotpath from the moment one of the listeners receives the Market Data to the moment an order is ready to be submitted to one of the exchanges by the Order Manager.

It should be noted that optimising network communications between our server and the exchanges is outside of the scope of this project.

2.2 Low-latency Implementation

We first look at how we chose to implement each section of the trading system to achieve latency on a nanosecond scale. We also look at any design decisions made for the project to be more useable. Although the basis of our project was to create a framework for users to test their own code and design patterns, having a low-latency

[2] e.g., Buy 100BTC @ USD60000

system as a base is essential. This is so that the framework is more realistic and the impact of changes which may only ameliorate the latency by nanoseconds or less are identifiable more easily by the user.

2.2.1 Choice of Language

C++ has sufficient language features for both high-level abstraction and low-level control. As such, we are able to manage such large-scale software with the specified system design, and also perform optimisations for low latency. C++ is an industry standard for low-latency trading system applications and most users are looking to develop their systems with C++. By providing a C++ framework for benchmarking, we address these needs.

For the report-generating script, we chose to work with Python, a language which supports writing script files that can be run in a simple manner. Python offers a wide range of libraries that make complicated tasks, such as generating a PDF file, simpler. This also results in the code being readable and easy to maintain, while also reducing the risk of bugs since most functions come from libraries.

2.2.2 Getting Data from Exchanges in Real-time

Each exchange is associated with a listener whose job is to receive incoming data from the exchange and push it to trading strategies. As such, each listener stays on a single thread acting as a dedicated connection between the exchange and our trading strategies. Once a thread is allocated to a listener, we pin the listener to the thread and do not change the thread which runs the listener, so as to avoid the cost and time of destruction and creation of threads.

2.2.3 Using *WebSocket*

We chose to use *WebSocket* instead of other protocols because *WebSocket* keep a single persistent connection open, which sends updates immediately when they are available. Other protocols such as *HTTP*, for example, would increase latency due to the request/response-based connection.

2.2.4 Trading Strategies

In this trading system, we use the following two trading algorithms, *Bollinger Bands* and *Arbitrage*. We spoke to our supervisor and qSpark and decided that it would be important to have multiple algorithms. These serve as a demonstration of how a user would implement and benchmark their different trading algorithms. More details can be found in Appendix B.

2.2.5 Order Management

The `OrderManager` class manages orders sent to different destinations. It maintains a history of sent orders in the event that orders are to be amended. The `OrderManager` class manages the usage of the `OrderExecutors`. `OrderExecutors` define the submission of an order. This may be to an exchange, a file or *nothing* at all (see Appendix C).

For submitting to exchanges, we have the `ExchangeOrderExecutor` class which implements `OrderExecutor`. Each exchange is associated with an `ExchangeOrderExecutor` implementation, which is responsible for submitting orders.

The rationale behind the use of an `OrderManager` (as opposed to directly submitting orders to executors) is to maintain extensibility.

A trading strategy, such as arbitrage, may rely on several exchanges at once. An architecture that uses a manager class abstracts away the details of ordering from each exchange, and allows support for further exchanges to be added by simply adding the new `OrderExecutor` implementation to the `OrderManager` class (as opposed to having to implement the changes in each trading strategy).

A further advantage of managing executors in such a way is that we can mix and match several types of `OrderExecutors`. For instance, we could define the executor to output to a file for one type of exchange (instead of submitting the order) whilst still submitting trades using another executor.

2.2.6 Order Submission

Order submission is performed by `ExchangeOrderExecutors`, a series of classes that have implementations of order parsing, authentication, signature generation, etc., that are specific to each exchange.

The intention is that if a user would want to add support for another exchange, they would implement the functions in this class.

The project currently has support for *BitMEX*, *Coinbase* and *Binance*, which act as examples in addition to being working `ExchangeOrderExecutor` implementations.

2.2.7 Connecting Back to the Exchanges

In the example implementations we provided, we used HTTP via *libcurl* to submit the serialised order data. A commercialised automated trading program would typically use *WebSocket* to communicate orders to the exchange as it is a faster protocol (compared to alternatives).

There were several reasons we picked *HTTP* over a *WebSocket* implementation. First, not every exchange in our base implementation supported *WebSocket* for order-side requests and instead only accepted *HTTP* requests. In order to manage our time efficiently, we chose to send orders over *HTTP* via *libcurl* as it would work for all exchanges. Second, it is important to note that the mechanism of sending orders is not actually part of the hotpath. As described in §2.3.1, the hotpath exists up to (but not including) the point at which orders are submitted to the exchange over the network. Therefore, hotpath benchmarks are indifferent to the implementation of order submission. As a result, any implementation would suffice regardless of latency.

2.3 Benchmarking and Profiling

The objective of benchmarking was to provide an abstraction that allowed users to:

1. Accurately time their design implementations.
2. Generate a concise representation of the results that could support further analysis.

The following outlines how we met those design criteria.

2.3.1 Generating Results

In terms of accurate timing, there is a requirement for accurate and precise results - hotpath implementations typically feature runtimes with **nanosecond** orders of magnitude.

We implemented function and full hotpath latency benchmarking using the *Google Benchmark* library [4]. We chose *Google Benchmark* over alternative, lightweight options as it provided **nanosecond** precision.

Benchmark	Time	CPU	Iterations
ExchangeOrderExecutor_BinanceOrderExecutor_parseOrder	1075 ns	1074 ns	644977
ExchangeOrderExecutor_BinanceOrderExecutor_authenticate	35084 ns	35060 ns	19549
ExchangeOrderExecutor_BinanceOrderExecutor_generateTimestamp	47.9 ns	47.8 ns	14599636
ExchangeOrderExecutor_BinanceOrderExecutor_generateHeaders	363 ns	363 ns	1960762
ExchangeOrderExecutor_BinanceOrderExecutor_submitOrder	39820 ns	38178 ns	18540

Figure 2: Sample results from our benchmark for the OrderExecutor class.

Another requirement for accurate timing is the ability for users to test specific stretches of code independently, so that calls to the next part of the hotpath are not included in the timed result. A good example of this requirement is benchmarking the *hotpath*.

The system, by default, is a full implementation of an entire automated trading system. This includes fetching data from the exchange via WebSocket, to processing and executing an order over HTTP using *libcurl*. A benchmark over the entire system would include these extra steps in the time it returns. To remain consistent with the fact we are benchmarking solely the *hotpath*, we need to disable/enable stretches of code that are/are not defined as being part of the *hotpath*.

To fulfil this requirement, we initially thought of requiring the user to instantiate classes with a benchmarking flag; the idea being that further function calls would be disabled by these flags.

A problem with this approach is that the resulting code would not actually be representative of the original implementation. Adding additional branches would result in different code, and therefore the subsequent time would be inaccurate.

Ideally, we would modify the code before runtime. This would involve removing elements that the user does not want called during their benchmark instead of disabling it at runtime.

We achieved this by specifying two compiler flags: `BENCHMARK_HOTPATH` and `ENABLE_CPP_BENCHMARKS` (we would have had a conflict with the more appropriately named Google Benchmark's `ENABLE_BENCHMARKS`).

These flags in turn enabled/disabled their corresponding pre-processor definition, which could be placed around function calls (using `#IFDEF` directives) in user implementations to toggle them.

```
void functionToBenchmark() {
  // Some work is done here...
#ifndef ENABLE_CPP_BENCHMARKS
  callToAnotherFunction();
#endif
}
```

Figure 3: Sample usage of pre-processor definitions.

- The first flag, `BENCHMARK_HOTPATH`, is to be used to disable code that lies outside the definition of the hotpath. For instance, calls to *libcurl* in order to submit order data.

- The second flag, `ENABLE_CPP_BENCHMARKS`, is to be used to disable code that would not be considered part of a function call. For instance, disabling the call to submit an order after parsing it (we would not consider the call to be a part of `parseOrder()` but its own benchmarking case `submitOrder()`).

The reasoning behind having two different flags is that the flags are mutually exclusive. Enabling the benchmarking of individual functions would prevent calls to other functions. As a result the hotpath would be incomplete/broken if we had a case where both flags were enabled.

2.3.2 Result Representation

The representation of results is important as the overarching aim of the project is to enable systematic research into the performance of design patterns. Therefore, a good representation should focus on enabling the comparison of different design pattern performance.

The results of the comparison between benchmark_at[2021-12-22_22:31:45] and benchmark_at[2021-12-23_15:35:59].
All the timings are measured in nanoseconds.
The graphs have the y-axis set to logarithmic scale to account for big difference in times amongst the benchmarked functions

These are the results of the functions in the **OrderBook** class

Function Name	Previous Time	New Time	Previous CPU Time	New CPU Time	Time Difference	CPU Time Difference
OrderBookFixture/OrderBook_addTradingStrategy	49.16	84.96	49.06	84.86	35.80	35.80
OrderBookFixture/OrderBook_addEntry	95.58	54.69	95.46	54.61	-40.89	-40.85
OrderBookFixture/OrderBook_getHighestBid	30285.69	17183.23	30221.11	17168.72	-13102.46	-13052.39
OrderBookFixture/OrderBook_getLowestAsk	31170.60	16971.99	31024.22	16961.80	-14198.61	-14062.42

These are the results of the functions in the **TradingStrategy** class

Function Name	Previous Time	New Time	Previous CPU Time	New CPU Time	Time Difference	CPU Time Difference
OrderBookFixture/TradingStrategy_BollingerBand_newData_runStrategy	188.73	154.05	171.45	153.95	-34.67	-17.50
OrderBookFixture/TradingStrategy_insertNewOrderBook	150.74	141.03	149.99	140.87	-9.70	-9.12

These are the results of the functions in the **ExchangeOrderExecutor** class

Function Name	Previous Time	New Time	Previous CPU Time	New CPU Time	Time Difference	CPU Time Difference
ExchangeOrderExecutor_BinanceOrderExecutor_parseOrder	1072.93	656.23	814.17	653.70	-416.70	-160.47
ExchangeOrderExecutor_BinanceOrderExecutor_authenticate	36781.26	28152.89	32405.40	27605.72	-8628.38	-4799.68
ExchangeOrderExecutor_BinanceOrderExecutor_generateTimestamp	132.25	103.81	129.67	103.71	-28.44	-25.95
ExchangeOrderExecutor_BinanceOrderExecutor_generateHeaders	89.36	81.16	88.64	81.13	-8.19	-7.52
ExchangeOrderExecutor_BinanceOrderExecutor_submitOrder	31920.22	31119.01	30804.20	30582.83	-801.21	-221.38

These are the results of the functions in the **OrderDataStore** class

Function Name	Previous Time	New Time	Previous CPU Time	New CPU Time	Time Difference	CPU Time Difference
OrderDataStoreFixture/OrderDataStore_addEntry	67.11	37.05	66.98	37.04	-30.05	-29.94

This is the benchmarked **Hot Path**

Function Name	Previous Time	New Time	Previous CPU Time	New CPU Time	Time Difference	CPU Time Difference
OrderBookFixture/benchmark_hotpath	805.06	370.01	586.65	368.90	-435.05	-217.75

Figure 4: Report tables output example

We represent our results using an automatically generated report. For the report, we compare two JSON files which have been generated by running the benchmark and contain all the functions' executions times and other useful information. In the report, there are two parts; the tables, holding the measurements for each function for each run and the difference between runs, and the graphs, plotting the execution time for each function for each of the two runs. The report-generation script then uses the Python library *fpdf* to generate a PDF file containing the tables and the graphs and automatically opens it for the user. The script also has different behaviour based on the arguments passed and the optional flags that the user sets. We used *argparse*, a Python library, to simplify this process and allow for future

development and extension. There are multiple benchmarking options depending on user needs:

- **No arguments:** the script will run the benchmarker and produce a json file

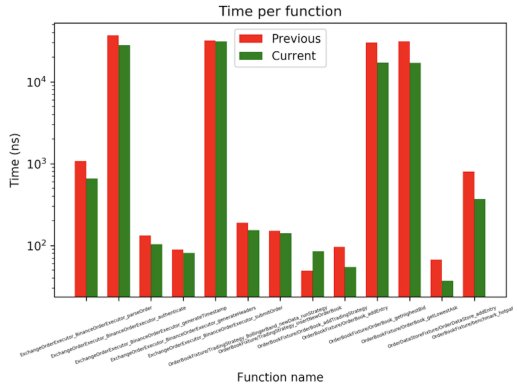
  - If the user runs it with `-l` option then it will run the benchmarker and also compare the results with the most recent run.

- **One argument:** *(a json file name from a previous run)* the script will run the benchmarker and compare the results with the specified file.

- **Two arguments:** *(two json file names from previous runs)* the script will produce a comparison report for the two specified runs.

Note: in order to compare every function and the hotpath benchmark, the script has to recompile the project twice for each report, which is the main drawback because this takes time.

This graph shows the difference in total time between runs:



This graph shows the difference in CPU time between runs:

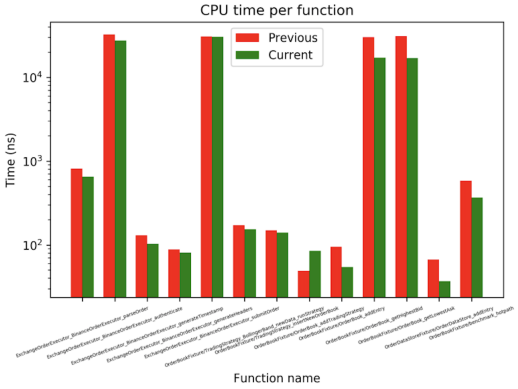


Figure 5: Report graphs output example

2.3.3 Profiling using Perf

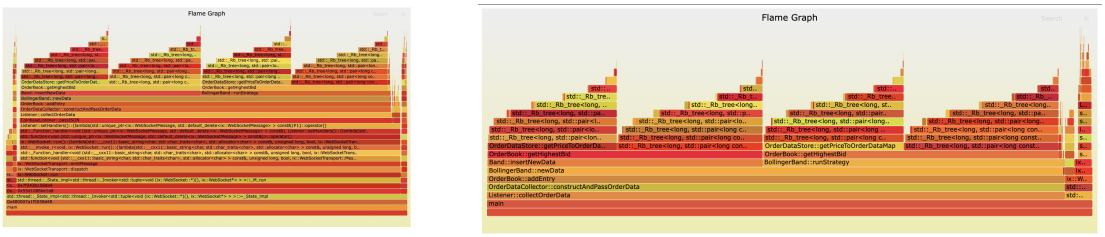
Throughout our development and by talking to our partners, qSpark, we have found that benchmarking is often not enough to identify where exactly the performance bottlenecks are. That is because we can only benchmark functions we have decided to, but we cannot precisely see where our program spends most of its execution (i.e., which parts are the slowest). For that, we need a *profiler* and following our research and talks, we have decided to use *perf* [5], a built-in Linux profiler.

As with most Linux tools, the perf user interface is rough around the edges — it takes a while to understand its results in the textual form. We have come across *FlameGraph* [6], an interactive web-based way to summarise perf's results.

However, we also have found that it is not straightforward to integrate perf with FlameGraphs to provide meaningful outputs. We were required to streamline a lot of parts to deliver clean output. The final result is the *benchmark-perf.sh* script, which automatically runs perf against our trading system and then generates an HTML interactive report using FlameGraph. Figure 6 pictures the initial perf output, as well as the output after our custom improvements.

2.4 Design Patterns and Hotpath Optimisations

As a demonstration of the benchmarking framework we created, we implemented a few low-latency design pattern optimisations and benchmarked their performance.



(a) Raw Perf FlameGraph (b) Perf output FlameGraph after the clean up

Figure 6: Example perf output for the trading system, both before and after the clean up. This chart has allowed us to realise that most of the execution time was spent in *getHighestBid()* method and to find a performance issue there.

More details of the performed optimisations are presented in Appendix A.

2.5 Interfacing

The main objective when creating an extensible framework for benchmarking was for users to be able to integrate their code fluidly into the system. We wanted to remove the barrier to entry of having to develop an entire trading system to be able to test any given part. Therefore, we abstracted each part of the system using C++ interfaces (classes defined in a header file containing virtual functions to be implemented by any subclasses as well as variables common to all subclasses). Assuming a user implements the interface functions required by a part of the system (e.g., buy/sell within a trading strategy), the rest of the system will behave exactly the same. These compartmentalised sections are represented by each of the boxes in the **System Architecture Diagram**. We describe below the reasons for our implementation of each of the interfaces and the process for a user to use them.

We highlight two of the most important interfaces: Trading Strategy and Order Storage System. The rest can be found in Appendix C.

2.5.1 Adding New Trading Strategies

From our discussions with qSpark, this was agreed as being the most likely part of the trading system that users would want to implement. We therefore kept this interface as simple and abstract as possible. The functions to be implemented within a new trading strategy from the interface `TradingStrategy.h` are:

- `newData()`: It is important for users to define behaviour when new order data is passed through since this is the moment at which we test the latency of the hotpath.
- `buy()` and `sell()`: These are fundamental to any trading strategy so we wanted to allow users to change what order data was passed through these methods.
- `runStrategy()`: The body of the trading strategy run on each iteration, used to assess the trading algorithm independently.

The trading strategy interface also contains an unordered map which maps exchanges to order books which can be replaced with a custom map and benchmarked if desired.

2.5.2 Changing the Order Storage System

Order Book

The order book holds a bid and an ask store for each exchange which is built from the exchange data. In our system, we then average these to get an indication of what the 'real' price is in the market within our trading strategies. We decided not to interface the order book methods since users may use this data differently, choosing instead to leave it in the most interpretable form possible through our `OrderData` class. A user could change the order book implementation directly in the `OrderBook.h/cpp` files but since an order book is made up of several data structures rather than methods, we do not have an interface for this.

Order Data

As mentioned above, the data contained within an order is kept as pure as possible to allow custom data manipulation. However, we decided to provide two different ways to change how orders are constructed for as much flexibility as possible.

- Directly changing the data contained within an order in `Order.h` (the type used in order execution) and `OrderData.h` (the type used within the order book and data management). These classes contain a list of fields representing data within an order so any custom fields can be added and removed as desired. This interface does not allow for altering the data in an order on the go.
- An order builder interface `OrderBuilder.h` used to dynamically create and change orders on the go. This may be used for trading strategies/designs where the order data is not fixed (for example, if it is coming from a custom file where a user wants to edit the data after importing it).

2.6 Technical Challenges and Achievements

The primary technical challenge we faced as a group throughout the project was the steep learning curve of C++, since only two members of the group had prior experience. Despite us all being proficient in Java and C, many C++ specific constructs, such as smart pointers or inline functions, were new to us. As low-latency design patterns are a highly technical subset of C++, they often required strong comprehension of the language in order to understand and implement them effectively.

Another challenge we faced was that the project demands us to master many new tools quickly. It is the first time that we use git submodules instead of a package manager to handle libraries. Modern C++ projects use CMake instead of Makefile, therefore we faced the challenge of learning it from scratch and using it proficiently in this large-scale project. We also explored and understood the idiosyncrasies of various libraries we used, such as *libcurl* and *Poco*. Our system required us to learn new communication protocols, such as *WebSocket*. We also had to learn tools for testing and benchmarking, such as *Google Test*, *Google Benchmark* and *perf*.

In addition, we also encountered challenges in understanding relevant financial concepts so that we have sufficient domain knowledge to build a functioning trading system and appreciate common trading strategies. Diving deeper into these, we then understand the technicalities involved behind a competent trading system, and perform necessary optimisations with the help of different tools. For instance, we have identified performance bottlenecks in our solution using *FlameGraph*, such as unnecessary copying of data structures in memory. We fixed these using the benchmarking tools we had set up, thus developing a system able to trade on a scale of nanoseconds.

As our system is multithreaded in C++[7], we inevitably encountered race conditions and memory leaks. That not only caused numerous sleepless nights, but also made us much more aware of C++ best practises, such as using smart pointers whenever

possible. We encountered issues with some of our libraries (such as *libcurl*) not supporting multi-threading well. In order to remedy this, we had to spend additional time making the *CurlManager* class thread safe — a matter of using mutexes to manage access to the *curl* handler pointer.

We also initially struggled in making our system scalable. As an example, we discovered through trial and error the disadvantages of using STL *vectors* over (linked) *lists* — it is easier to get segmentation faults, as the Operating System simply might not have a free contiguous block of memory large enough to store our data [3]. In such a case, using a *list* is preferable, as each element is allocated individually. This would not be an issue in most cases as trading machines tend not to be memory bound. These type of challenges cost us time debugging, but aided us in learning the depths of low-level programming.

Despite these technical struggles, we have successively overcome these challenges and are proud of the produced software.

2.7 Deployment

This system has been designed to work on Linux environments only, as is the industry standard. We ourselves have tested the codebase and deployed the trading system on a server running Ubuntu 20.04 LTS that was kindly made available to us by our supervisor.

2.8 Risk Anticipation and Mitigation

2.8.1 Lack of Low-latency Experience

At the start, none of us had any low-latency experience. Therefore, we expected to initially experience issues with our design or implementation. That is why we aimed for a robust but flexible system open to modifications. That has proven useful when we changed the direction of the project. We realised the need for a multithreaded system based on our conversations with qSpark. Despite that happening in week 3, thanks to our flexible design, we were able to reuse most components in that new multithreaded system and continue with our work.

2.8.2 C++ Code Quality

C++ is a languages where it is easy to quickly write poor-quality code[8] that is hard to understand and untestable. From the start, our goal was to avoid this. That is why we set up a comprehensive CI pipeline that verified code style and compilation before merging. Additionally, we have also aimed for 90% test coverage. We have often worked in pairs and reviewed each other's pull requests to ensure high-quality code. That has ensured that we delivered a C++ codebase that is well-documented and of high quality.

[3] We think this happened likely because our test machine only had 4GB of memory

3 Evaluation

We have delivered a working low-latency trading system that trades using different strategies on multiple exchanges. Together with it, we provide a comprehensive benchmarking suite that removes the need for users to first familiarise themselves with all the benchmarking tools.

Although we had to modify the original goal of the project, we believe the final system we have delivered is of high quality and makes it easier for future researchers to explore these patterns without the need to develop their own system first.

3.1 Software Quality Analysis

Being fully aware of how error-prone C++ is, we have decided to prioritise maintaining good quality, well-styled code with a large test coverage.

3.1.1 Code Style

At the start of the project, we all agreed to adhere to the Google C++ Code Style [9]. In order to enforce this, we have decided to use a *clang-format* [10] tool with a predefined formatting document that was placed in the repository (*.clang-format*).

3.1.2 Unit Testing

```
[==========] Running 62 tests from 11 test suites.
[----------] Global test environment set-up.
[----------] Global test environment tear-down
[==========] 62 tests from 11 test suites ran. (8 ms total)
[  PASSED  ] 62 tests.
```

Figure 7: Result from our unit testing framework.

As our system consists of many independent components, we have decided that unit testing is a good way for us to ensure the correctness of each of the individual APIs. For that, we have used the *Google Test* [11] framework and we present a sample of our test output in Figure 7.

As a way to continuously assess the quality of our unit tests, we have used the *gcov* [12] and *lcov* [13] tools to continuously measure our coverage. The final coverage is 90.1%, with the more detailed results presented in Figure 8.

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level	Lines:	401	445	90.1 %
Test:	main_coverage.info	Functions:	95	103	92.2 %
Date:	2021-12-29 16:14:37				

Directory	Line Coverage		Functions	
dataManager	91.1 %	72 / 79	100.0 %	29 / 29
orderDataCollectors	100.0 %	8 / 8	100.0 %	3 / 3
orderDataCollectors/fileReaderListener	100.0 %	17 / 17	100.0 %	4 / 4
ordering-system	100.0 %	14 / 14	100.0 %	3 / 3
ordering-system/exchangeExecutors	83.3 %	5 / 6	75.0 %	3 / 4
ordering-system/exchangeExecutors/binance	97.9 %	46 / 47	100.0 %	10 / 10
ordering-system/exchangeExecutors/bitmex	96.2 %	50 / 52	100.0 %	10 / 10
ordering-system/exchangeExecutors/coinbase	98.5 %	66 / 67	100.0 %	11 / 11
ordering-system/exchangeExecutors/utils	60.5 %	49 / 81	36.4 %	4 / 11
tradingStrategies	100.0 %	19 / 19	100.0 %	6 / 6
tradingStrategies/bollingerBand	100.0 %	55 / 55	100.0 %	12 / 12

Generated by: LCOV version 1.14

Figure 8: Our unit testing coverage.

3.1.3 Continuous Integration

In order to ensure and maintain the quality of our codebase, we have introduced a number of checks that were required to pass before merging a pull request.

We used Github Actions to execute our pipeline. It first runs the Code Style check to ensure that all the code is properly formatted. That is followed by running the

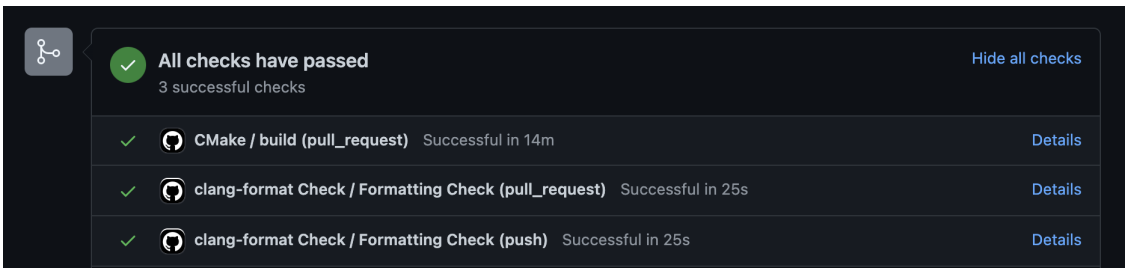


Figure 9: Output of Github Actions checks run on one of the pull requests.

compilation procedure to ensure that our code properly compiles. We then run our unit test suite to ensure the correctness of the proposed changes. A sample output of the pipeline can be found in Figure 9.

Pull requests were also required to be reviewed by one other team member before merging.

3.1.4 Performance

Although our system mainly provides a basis for users to build upon or change parts of it in order to benchmark their implementations, we also strove to create a base framework that operates on a nanosecond scale. As seen in Figure 10, the total duration for the hotpath is approximately 150ns.

```
-------------------------------------------------------------------------------
Benchmark                                Time             CPU   Iterations
-------------------------------------------------------------------------------
OrderBookFixture/benchmark_hotpath     152 ns          152 ns      4100216
DONE
```

Figure 10: Final hotpath benchmark results

3.1.5 Scalability

Our system currently runs on three threads, one per supported exchange. To scale the system further, it is required to add an extra thread per exchange. It is crucial that the number of threads does not exceed the number of available CPUs, as that can lead to resource starvation, slow down the system and lead to unreliable results.

This problem is fairly common in the industry. In order not to compromise on performance, it is common practice to simply allocate new machines if the need to scale to more exchanges arises.

However, it is important to note that there is also the concern of scalability of data. Each of the particular exchanges might in the future increase the rate or volume at which they operate. We have verified that our solutions are flexible and scalable via benchmarking. During benchmarking, many of the benchmarked data structures have been pre-filled to volumes much larger than expected.

3.2 Project Management

We followed two-week sprints to align well with the milestone deadlines and ensure we have working deliverables for each checkpoint. In addition, we have also held group meetings on Mondays (to plan for the week ahead) and Fridays (to sum up what we have achieved). This has ensured that throughout the term, we have constantly made progress on the project, irrespective of other deadlines.

We have additionally met our partners, qSpark, approximately every three weeks to discuss our progress and ask for their feedback.

For task management, we have used JIRA. To allow for remote working, we have usually communicated through Microsoft Teams or WhatsApp and have used individual video calls to resolve any other issues.

3.3 User Evaluation

Throughout the development, we met our partners, qSpark, on numerous occasions to discuss our progress and ask for feedback on our development direction. As an industry leader, their advice, constructive feedback and suggestions have led us to deliver a system that is more suited to modern-age High Frequency Trading. It has led to a pivot to support multiple exchanges and more than one trading algorithm, all running on separate threads. This change meant our system resembles a typical trading software better and that its benchmarks would be more realistic. Following our final feedback sessions, we believe we delivered a system that qSpark are satisfied with and is in line with industry standards.

We have also evaluated our product with our supervisor, Dr Paul Bilokon. Paul was happy to say that indeed this benchmarking framework provides a very good platform to find performance bottlenecks in the trading strategies and benchmark different low-latency C++ design patterns. He believes it will be useful is his future projects, which involve delving deeper into individual low-latency design patterns.

Additionally, we have also demonstrated our system to Dr Robert Chatley, who also gave us his positive feedback and suggested that he would also be interested to see a more visual representation of the benchmarking results. That led us to develop the PDF report generation feature, which sums up and compares the results of previous runs.

4 Ethical Considerations

4.1 Copyright Issues

The main copyright issues we faced came from the use of libraries and APIs.

Library/Software Name	Licence Type
Google Benchmark	Apache-2.0
cURL/libcurl	BSD 0-clause
Google Test	BSD 3-clause
CMake	BSD 3-clause
GCC	GPL-3.0
OpenSSL	Apache-2.0
Poco	Boost Software licence-1.0
Gcov	GPL-3.0
Lcov	GPL-2.0
PyPDF (Python library)	BSD 3-clause
NumPy (Python library)	BSD 3-clause
Matplotlib (Python library)	custom PSF licence
Argparse (Python library)	PSF licence
Datetime (Python library)	BSD 1-clause

Table 1: A table of libraries and their associated licence used in the project

The Apache-2.0 licence allows us "...to use the code for any purpose, to distribute it, to modify it, and to distribute modified versions of the software under the terms of the licence, without concern for royalties". This means that we can allow users to clone our repository and use our software for free.

The BSD licences (no matter how many clauses) allow for free commercial use. The clauses are there to protect redistribution of the source code (Clause 1), the binary code (Clause 2) and the author's name in case of advertisement (Clause 3).

The GPL (GNU General Public licence) is a series of free software licences that guarantee users the freedom to run, study, share and modify the software.

The PSF (Python Software Foundation) licence is a BSD-style, permissive licence which allows for free commercial use and is compatible with the GPL licence.

The Boost Software licence is a very liberal licence that encourages both commercial and non-commercial use.

4.2 Misuse

High Frequency Trading is a somewhat controversial subject with some claiming that it is leading to greater market manipulation and a two-tiered financial system [14].

While there are many potential misuses for trading systems, such as spoofing, layering and quote stuffing [14], our system is unlikely to be used for any of these purposes. Our system is specialised for benchmarking and trading and provides no benefit over other projects directly designed for market manipulation.

4.3 Security

Security is paramount in any system, especially in one that manages real funds. Therefore, trading systems need to be designed with security in mind and not as an afterthought.

The biggest risk in our particular domain is having secret API keys exposed through mismanagement. A secure trading client should include some mechanism to safely store and retrieve API keys, the entirety of this process being integrated into the trading system. In order to mitigate the aforementioned risk, we decided to formulate a simple but robust API key storage system.

Our solution was to store API keys on locally hosted `.env` (environment) files. Additionally, we implemented an interface to import and use the stored keys. Each team member working on a particular system would have to locally own their own copy of the API keys, with the idea that the environment files would be intentionally set to be untracked by version control.

4.4 Regulatory Implications and Compliance

Our product is a combination of an interfaced trading system and a benchmarking suite. The usage is meant to be for benchmarking the user's implementation and comparing it to an older version, clearly outlining the performance increase/decrease. It is not illegal or unethical to create a product for measuring latency of a trading system [15].

Our trading system does not trade any regulated securities, only cryptocurrencies.

5 Conclusion

5.1 Future Steps

One improvement to our system would be to add a more diverse set of trading strategies, exchanges or even generalise the system to run not only for cryptocurrencies but also stocks for example.

There also is potential to further improve our overall latency — this system serves as a base and with more time, we would strive to improve it further to bring down the latency.

From a software engineering perspective, including a performance check as part of the Continuous Integration pipeline would help ensure that code that performs worse is never accidentally merged; having a coverage check also would be useful.

We currently use the exchanges' test networks to place our trades. By using a free version, we can only submit orders via a REST API, which has its own performance implications (throttling and connectivity requirements). Performance could be improved if we can submit these orders over WebSocket using the paid service.

Our suite is intended for benchmarking C++. Given the emerging prominence of FPGAs in high frequency trading, it would be an interesting (but sizeable!) task to extend our framework to enable benchmarking for digital circuits too.

After the addition of more sophisticated trading strategies, it would also be interesting to explore the profit/loss of the system while trading on real exchanges.

We have made our codebase open source[4] in order to encourage future work and usage of the whole system.

5.2 Conclusion

Working with our various collaborators, we have produced a functioning, low-latency trading system that has a customisable and extensible framework, allowing users to benchmark and profile any part of the system. We have implemented a number of low-latency design patterns to serve as examples of potential improvements and how to evaluate them. We have leveraged a multithreaded design to allow users to add new strategies or exchanges, and used an interface-based approach to allow users to implement their own versions of the system. Our trading system allows users to develop a system without having to learn the technicalities of each tool or build their own system from scratch. The system would be of interest to, for example, researchers or users of low-latency, high-frequency systems looking to enhance their knowledge and benchmark their work.

[4] https://github.com/IC3RD/Benchmarking-Suite-for-High-Frequency-CPP-Trading-Systems

References

[1] Tom Polansek. Cftc finalizes plan to boost oversight of fast traders: official. URL https://www.reuters.com/article/us-cftc-trading-oversight-idUSBRE97M0ZH20130823. Last accessed 23 December 2021.

[2] Carl Cook. When a microsecond is an eternity: High performance trading systems in c++. URL https://github.com/CppCon/CppCon2017/blob/master/Presentations/When%20a%20Microsecond%20Is%20an%20Eternity/When%20a%20Microsecond%20Is%20an%20Eternity%20-%20Carl%20Cook%20-%20CppCon%202017.pdf. Last accessed 23 December 2021.

[3] Herb Sutter. High performance low latency c++ introduction and fundamentals. URL https://www.alfasoft.com/files/herb/00-Introduction.pdf. Last accessed 9 January 2022.

[4] Google. Google benchmark, . URL https://github.com/google/benchmark. Last accessed 23 December 2021.

[5] perf. perf: Linux profiling with performance counters. URL https://perf.wiki.kernel.org/index.php/Main_Page. Last accessed 23 December 2021.

[6] Brendan Gregg. Flame graphs. URL https://github.com/brendangregg/FlameGraph. Last accessed 23 December 2021.

[7] Anthony Williams. C++ concurrency in action. Last accessed 9 January 2022.

[8] Dewhurst Stephen C. C++ gotchas: Avoiding common problems in coding and design. Last accessed 9 January 2022.

[9] Google. Google c++ style guide, . URL https://google.github.io/styleguide/cppguide.html. Last accessed 23 December 2021.

[10] LLVM. Clangformat. URL https://clang.llvm.org/docs/ClangFormat.html. Last accessed 23 December 2021.

[11] Google. Googletest, . URL https://google.github.io/googletest/. Last accessed 23 December 2021.

[12] GNU. gcov—a test coverage program. URL https://gcc.gnu.org/onlinedocs/gcc/Gcov.html. Last accessed 23 December 2021.

[13] Linux Test Project. Ltp gcov extension (lcov). URL https://github.com/linux-test-project/lcov. Last accessed 23 December 2021.

[14] Steven McNamara. The law and ethics of high-frequency trading. Last accessed 10 January 2022.

[15] Irene Aldridge. High-frequency trading. Last accessed 9 January 2022.

[16] Fidelity Investments. Bollinger bands. URL https://www.fidelity.com/learning-center/trading-investing/technical-analysis/technical-indicator-guide/bollinger-bands. Last accessed 23 December 2021.

[17] Igor Radovanovic. Arbitrage: What is it and how to find it? URL https://algotrading101.com/learn/crypto-arbitrage-guide/. Last accessed 24 December 2021.

A Design Patterns and Hotpath Optimisations

Design Pattern 1 - Delegation

As presented in our video, we have found that one of the issues with our initial code was that whenever we wanted to fetch the highest bid or lowest ask from our Order Book data structure, we would access the whole Order Book and then call *.first()* or *.last()* on it. That has led to unnecessary copying and moving of data.

To avoid it, we have implemented an Order Book methods called *getFirst()* and *getLast()*, which avoids unnecessarily passing around the whole data structure and ensures we only return the data we care about.

That in itself has improved the performance of *getHighestBid()* and *getLowestAsk()* from *70ms* to *30ns*. The results are presented in Fig. 11.

Figure 11: Benchmark results of OrderBook *getHighestBid()*/*getLowestAsk()* before and after optimisation - the improvement being significant.

Design Pattern 2 - Branch Prediction

Since modern processors have long instruction pipelines, they will try to fetch and decode instructions far beyond where they are currently executing. In the case of a conditional branch, the processor cannot know ahead of time into which branch execution will continue. In order to continue the preemptive instruction pipeline, the processor will attempt to predict which way execution will occur. This is known as branch prediction. Unfortunately, branch prediction can often be wrong and will have to flush the pipeline and roll back to where the branch occurred, losing all of the preemptive work it performed after the branch.

This rollback and loss of work is slow and inefficient. Therefore, for the hotpath to be competitive, we want to minimise the number of branches in our system. If we cannot remove all the branches, we should aim to reduce the number of conditionals in a given branch in order to increase the accuracy of the branch predictor.

An example of this in our code is in updating an internal order book when receiving new market data. As described above, each order book maintains two internal order data stores, one for ask prices and one for bid prices. This requires a switch to determine which data store to insert the new data into. When receiving a high frequency of market data updates, with no way to predict if the updates are bids or asks, the branch predictor performs very badly.

In order to prevent branching in this case, we can store the two data stores in an array in the order book class, and when inserting the new data we can index the array with the type of order the order data is (0 for ask, 1 for bid). The resultant increase in speed on inserting a single entries into an order book with a biased branch predictor can be seen in Figure 12.

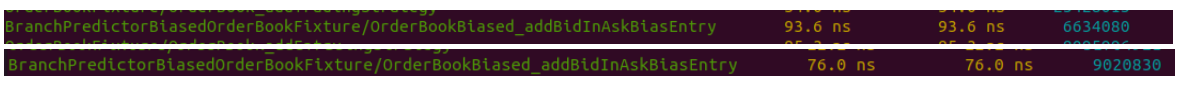


Figure 12: Hotpath speed up before and after removing a conditional branch in order data insertion, respectively, which shows around **17ns** speed up per insertion.

Design Pattern 3 - Hotpath Reformat

After a discussion with our supervisor in our 3rd iteration, we realised that the updates to the order book do not need to occur in the hotpath, but instead can occur after the trading strategy has been run on the new data.

However, due to the modularity of our benchmarking suite, we still can, of course, benchmark the performance of our order book insertion, despite it being outside of our 'hotpath'.

This shows how our benchmarking framework can be used to handle design changes as well as component optimisations.

B Trading Algorithms

Bollinger Bands

The first trading strategy we implemented, as part of our first iteration, was Bollinger Bands. Our supervisor recommended the implementation of Bollinger Bands as an initial strategy because of its simplicity and because, unlike Arbitrage for example, it can operate on data from just a single exchange.

A Bollinger Band is created, for each instrument we want to trade by the moving average and standard deviation of the price over recent market data packets from the exchange (here 'price' refers to the average of the bid and ask prices listed on the exchange). The upper boundary of the band is the recent average price (SMA) plus 2 standard deviations and the lower boundary is the recent average price minus 2 standard deviations. The strategy decides to buy if the current ask price is below the lower boundary and to sell if the current bid price is above the upper boundary. Please see Figure 13 as a visual example.

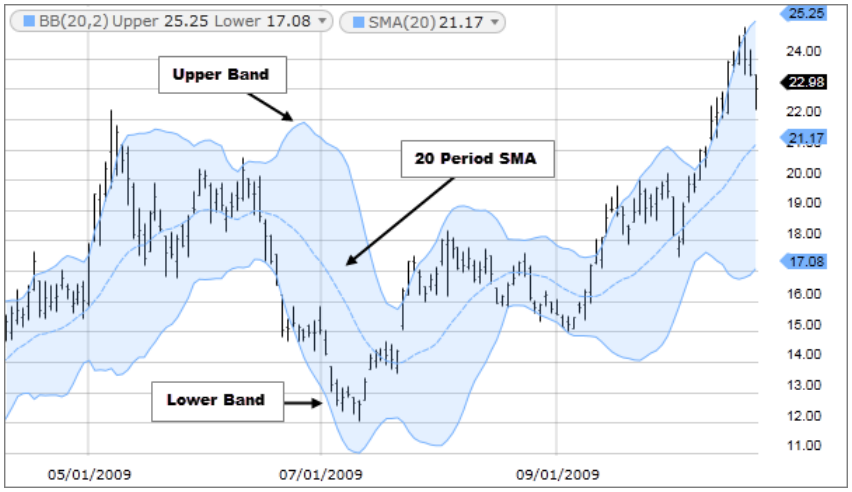


Figure 13: Example Bollinger Band [16]

As the trading strategy is part of the hotpath, we want to perform the moving average and standard deviation recalculations as efficiently as possible. On receiving new market data, we recalculate the mean and variance using the current values as opposed to storing a data set of the recently observed values and recalculating based on the values in the data set. This enables $O(1)$ data insertion time as opposed to an $O(n)$ cost in recalculating the mean and variance over the stored values.

Mathematically, let x_i be the various price points, and say there are n existing price points, then we have $\mu_n = \frac{1}{n}\sum_{i=1}^{n} x_i$ as the current mean price, and upon the

addition of the $(n+1)$th point, we have

$$\mu_{n+1} = \frac{1}{n+1}\sum_{i=1}^{n+1} x_i = \frac{1}{n+1}\left(x_{n+1} + \sum_{i=1}^{n} x_i\right) = \frac{x_{n+1}}{n+1} + \frac{n}{n+1}\mu_n,$$

$$\mathrm{Var}_{n+1} = \frac{(x_{n+1} - \mu_{n+1})^2}{n+1} + \frac{n}{n+1}\mathrm{Var}_{n+1} + \frac{n}{n+1}\delta^2.$$

When we added more listeners to multiple exchanges, on receiving new market data for a given instrument on an exchange, we then updated the Bollinger Band for that exchange and compared the most recent bid and ask price with the upper and lower boundaries of the bands for this instrument across the other exchanges.

Since our implementation extended the `TradingStrategy.h` interface, it was entirely containerised in a later iteration.

Arbitrage

Our second trading strategy is Arbitrage. Arbitrage is the simultaneous purchase and sale of the same cryptocurrency across different exchanges to profit from the small differences in the currency's price. It exploits short-term variations in the price of the cryptocurrency due to market inefficiencies.

For a basic example, assume that the ask price of Cardano (the price someone is trying to sell at) may be £1.44 on Coinbase, and the bid price (the price someone is trying to buy at) may be at £1.46 on Kraken. Therefore, by buying at £1.44 on Coinbase and immediately selling on Kraken at £1.46, the algorithm is guaranteed to make a profit of 1.4% on every Cardano coin bought, assuming the exchange fees for buying on Coinbase and selling on Kraken are less than 1.4%. The algorithm will continue to exploit this arbitrage opportunity until either there is no Cardano left at £1.44 or specialists at the exchanges adjust their prices to wipe out the opportunity. This can also occur by exchanging multiple coins across one or more exchanges as shown in figure 4:

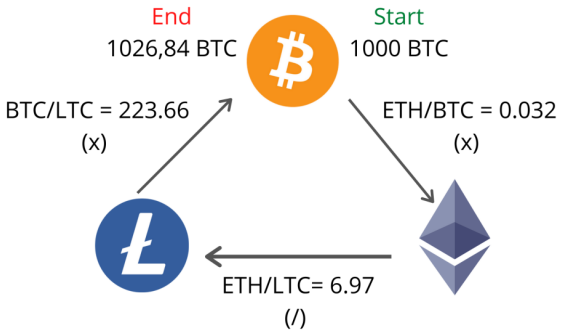


Figure 14: Cryptocurrency arbitrage using exchange rates [17]

We decided to use this strategy since it is the basis of High Frequency Trading (which itself relies on ultra low latency). This means that it provided a good indication of how well our system was working in a low-latency setting. It also uses multiple layers of the order book which is useful in providing a representative example of how to integrate the order book into a new algorithm.

To make it as realistic as possible, we took into account the different exchange fees (to ensure we were making a profit) while looking at all potential price discrepancies simultaneously to ensure that we capitalised on any arbitrage opportunities. Similar

to the Bollinger Band trading strategy, since Arbitrage simply implemented the `TradingStrategy.h` interface, it was entirely containerised.

C Interfacing

Changing Data Input Feed

Getting Data from a Custom Location

The interfacing allows a user to get data from any location such as a text file, database or the internet. This may be used, for example, by developers to test the latency and behaviour of their system deterministically with a fixed data set. It may also be used with historical data to evaluate the sanity of a trading strategy before moving it to production. This change is done as follows:

- Implement methods from `OrderDataCollector.h` in a custom listener
- An example of how to do this is defined in our pre-built `fileReaderListener`

We chose to use `JSON` as the form of data to be transferred from the listeners since it is very lightweight to transfer in HTTP requests and is the standard in the industry.

Adding Additional Exchanges

We decided to have a pre-built multithreaded `Exchange Listener` which implements the `OrderDataCollector.h` interface. This provides some additional websocket specific functionality to make it easier to quickly add new exchanges. Users can therefore use this extra functionality as follows:

- Create a new listener by implementing methods from the interface `Listener.h`
- Update the `Exchange.h` interface to contain the exchange name in the enum
- *Optional:* To change how order data itself is constructed and sent, edit the methods in `Listener.cpp`

Changing How and When a Strategy Is Run:

This involves changing single method `updateData()` within `TradingStrategy.cpp`. We decided to implement this concretely in the '.cpp' file rather than putting it as a virtual function since it is common to all the strategies being run and is independent from the latency of the strategy. We kept this as a single method to make it easier for users to change if required.

Changing How Orders Are Executed

Sending Orders to a Custom Location

We interfaced the order execution system through `OrderExecutor.h` to allow users to output their orders to a custom location if desired. This is because users may want to sanity check their trading strategy and implementation before connecting to a real exchange. It also allows researchers to have an end-to-end trading system to benchmark without having to necessarily place orders into a market.

We chose to interface this since the order placement may not be relevant to the latency of their implementation of other parts of the system. For example, a user testing a new design pattern in their trading algorithm may want to assess the performance implications of it but not want to actually submit orders. Thus, they can dismiss the order once the algorithm makes a buy/sell decision but still get an accurate and consistent benchmark for the algorithm.

Sending Orders to New Exchanges

Similar to receiving data from new exchanges, users may want to place orders on more exchanges. This is why we implemented an interface `ExchangeOrderExecutor.h` which implements the overall interface `OrderExecutor.h`. This interface provides some additional functionality (cURL methods) for sending orders to new exchanges, making it quicker and easier for users to connect to other exchanges. Of course, a user can also directly implement the top-level interface as described above should they not wish to use cURL for data transfer between the system and exchanges.

Using Custom Data Structures

Throughout the project, we mostly used data structures from the standard library because the performance of these was suitable for our performance requirements. However, these can easily be replaced by custom data structures since they are always defined privately within the header files for simplicity when changing them.