**Logo, company name

Description automatically generated**

**INDIVIDUAL ASSIGNMENT**

**TECHNOLOGY PARK MALAYSIA**

**CT087-3-3-RTS**

**REAL TIME SYSTEMS**

**HANDOUT DATE : 20 MARCH 2024**

**HAND-IN DATE : 10 JUNE 2024**

**WEIGHTAGE : 60%**

**NAME : WONG BIN JIE**

**TP NUMBER : TP060875**

**LECTURER : ASSOC. PROF. DR. IMRAN MEDI**

**INTAKE : APU3F2402CS(DA)**

Table of Contents

[1.0 Abstract 1](#_Toc168873060)

[2.0 Introduction 2](#_Toc168873061)

[3.0 Problem Statement 4](#_Toc168873062)

[4.0 Literature Review 5](#_Toc168873063)

[4.1 Memory Management 5](#_Toc168873064)

[4.2 Error Handling 6](#_Toc168873065)

[4.3 Priority Inversion Handling 8](#_Toc168873066)

[4.4 Synchronous and Asynchronous Programming 8](#_Toc168873067)

[4.5 Zero Cost Abstraction 9](#_Toc168873068)

[4.6 Benchmarking Techniques 9](#_Toc168873069)

[5.0 Research Design and Methodology 10](#_Toc168873070)

[5.1 Stock 11](#_Toc168873071)

[5.2 Trader 11](#_Toc168873072)

[5.3 Stockbroker 11](#_Toc168873073)

[5.4 Code Snippet 12](#_Toc168873074)

[6.0 Results and Discussion (Benchmarking) 19](#_Toc168873075)

[6.1 Stock Generator 19](#_Toc168873076)

[Section 1 – TP060875 19](#_Toc168873077)

[Section 2 – TP061818 20](#_Toc168873078)

[6.2 Simulation Orders per Trader (20 orders) 22](#_Toc168873079)

[Section 1 – TP060875 23](#_Toc168873080)

[Section 2 – TP061818 24](#_Toc168873081)

[6.3 Determine Market News/Condition 25](#_Toc168873082)

[Section 1 – TP060875 25](#_Toc168873083)

[Section 2 – TP061818 26](#_Toc168873084)

[6.4 Affect Price Benchmark 28](#_Toc168873085)

[Section 1 – TP060875 28](#_Toc168873086)

[Section 2 – TP061818 29](#_Toc168873087)

[6.5 Trader’s Buy or Sell Decision Benchmark 31](#_Toc168873088)

[Section 1 – TP060875 31](#_Toc168873089)

[Section 2 – TP061818 32](#_Toc168873090)

[6.6 Broker Process Order Benchmark 34](#_Toc168873091)

[Section 1 – TP060875 34](#_Toc168873092)

[Section 2 – TP061818 35](#_Toc168873093)

[6.7 Overall Simulation Benchmark 37](#_Toc168873094)

[Section 1 – TP060875 37](#_Toc168873095)

[Section 2 – TP061818 39](#_Toc168873096)

[7.0 Conclusion 41](#_Toc168873097)

[8.0 References 42](#_Toc168873098)

# 1.0 Abstract

In the modern world, real time systems are very important in applications that require response time and quick decisions, such as industrial automation, telecommunications, multimedia, etc. In this paper, the author aims to explore the capabilities of Rust in developing the real time stock trading system. The proposed system operates under the data processing and execute instantly for which Rust has the advantages of memory safety and type safety along with features like zero-cost abstraction. The system architecture is divided into two major subsystems which are Trading Processing and Stock Market Processing. These subsystems work concurrently to manage trade lifecycles, implement trading algorithms, and process live market data while being resilient. RabbitMQ is employed for proper management of message flow between traders and brokers in the simulation. Applying Rust in practice, the project also focuses on Rust’s memory ownership, error handling, priority preemption, and ability to support both synchronous and asynchronous programming paradigms. The benchmark reveals the effectiveness of the system performs in various trading situations, highlighting that Rust is well suited for real-time computational tasks. Overall, this study provides a thorough understanding on how Rust can improve the trading business by introducing an environment that supports the execution of actual trading transactions without so much of a delay and minimum glitch, making it an ideal choice for developing real-time stock trading systems.

***Keywords: Rust, real-time system, stock trading, memory ownership, zero-cost abstraction, lifetimes, borrowing, RabbitMQ***

# 2.0 Introduction

According to (Laplante & Ovaska, 2012), the term "real time" is often used in both technical and conventional contexts. Generally, when we say something happens "in real time", it is commonly interpreted as something happening immediately or without any delay. As mentioned by Priyanka (2023), a real-time system (RTS) is an information processing system which is supposed to respond in some given time or within some deadline. These systems are crucial in areas where time factor is highly significant, such as industrial automation, telecommunications, and multimedia. In RTS, accuracy is a measure not only of the logical correctness of the answer, but also the time it takes to obtain the answer. RTS are typically classified in the forms of soft, heard and firm real-time systems, with different levels of acceptance of the possibility of missing the time constraints. Reliability is an important aspect when it comes to developing and implementing the RTS due to the critical nature of timing (Al-Aubidy, 2011). There are various measures that can be adopted by the organizations to monitor and evaluate the performance of the RTS. The major KPIs are hence to minimize the schedule length and response time, as well as increase the system throughput. Other aspects of scheduling changes also entailed the use of scheduling paradigms and scheduling algorithms for the paradigms. Some of the paradigms and algorithms include priority based scheduling or earliest deadline first (EDF) which are used to ensure that the tasks are completed in the right time or within a given deadline. RTS can consistently meet such crucial time limits and satisfy the overall real-time requirements of various applications if the timing constraints are effectively managed and the right scheduling methods are adopted.

Rust is a system programming language which supports multiple paradigms, and it is memory safe, more secure and also the language is enhanced with performance optimization features (Jung et al., 2018). Rust can be seen as being closer to the C++, C and similar languages, but Rust provides more safety guarantees for the programmer while at the same time allowing full control over hardware and fine-grained performance tuning (Qin et al., 2020). Although it is a relatively new language, Rust has aimed at optimizing the efficiency and protecting from the simple mistakes, such as null pointer dereferencing, data races, and buffer overflows. This is done by means of its unique ownership type and borrowing laws, which ensure safe memory management during compilation while not calling for a garbage collector (Khachatryan, 2023). This means that programs cannot access memory owned by other programs. Besides, the zero-cost abstractions and ensured memory safety are targeted at low latency, which makes Go suitable for real- time systems (Jung et al., 2018). It is also statically typed, which, in practice, means that the type of the variable is checked by the compiler rather than at the runtime (Wróbel, 2023). This, in turn, reduces the probability of runtime errors and enhances the resilience of the code by identifying defects at early stages of the development provision. In addition, Rust has adopted asynchronous programming paradigms and supports first-class concurrent programming and thus the developers are able to build clear and efficient code without ending up relying on locks or other synchronization tools. But Rust being a relatively new language, it has issues with the lack of third-party tooling and libraries, and this makes getting started hard.

Thus, this paper will explain how Rust can handle real-time concerns through analyzing its specific characteristics. Moreover, this study will also delve into the application of Rust in the trading sector. The project process flow, simulations with code and explanations, and performance analysis will be covered, including the discussion on micro-benchmarking. Benchmarking outcomes will be discussed, and then a brief conclusion in relation to the project’s overall simulation and benchmarking and final thoughts throughout will be given before marks an end in this report.

# 3.0 Problem Statement

A real-time stock trading system is a software platform which tends to provide support to the fast trading of stocks in real-time or near real-time environment. It involves using advanced algorithms to analyze trading data and execute trades in fractions of seconds (Zuss, 2021). While the classical trading techniques imply that particular transactions can take minutes or even hours to complete, high-frequency trading systems allow traders or investors to submit orders and receive confirmations within mere seconds depending on the rate of changes in the market. However, designing such a system comes with challenges, as even minor delays or inefficiencies can lead to missed opportunities or financial losses.

Real-time systems are ideal for trading due to their ability to process thousands of data and transactions per seconds. Therefore, the design of the system is based on the principles of fast and efficient work in order to reduce latency between data ingestion, strategy adoption, and order completion, ultimately maximizing the profitability. A real-time system is critical for traders to obtain correct and timely market data so that a trader may respond to chance or reduce a loss as soon as possible. Furthermore, the real-time system incorporates feature such as fault tolerance to ensure that any changes are effectively handled to avoid losses due to system failure. In addition, it is built to handle market data and to place trading orders, and at the same time, to utilize resources in order to produce quick results.

There are two subsystems in this real-time stock trading simulation, namely Trading Processing (SS1) and Stock Market Processing (SS2). It is in the SS1 where the functionality of trade orders and traders to generate the buy and sell orders based on market situations as well as the processing of the orders are handled. On the other hand, SS2 is responsible for accepting and processing the trade requests as well as adjusting the stock prices in accordance with the result of the processed orders. The purpose of this project is to develop an environment where trading in stocks is simulated real-time where the efficiency of rust to meet the needs of high performance is well illustrated.

# 4.0 Literature Review

## 4.1 Memory Management

Memory management is one of the fundamental parts of Rust where it ensures both performance and safety without the need for a garbage collector. This approach is attained through a framework called ownership, borrowing, and lifetimes. Ownership assures that every piece of memory is associated with a single owner in Rust. When the owner goes out of the scope, Rust will automatically free up the memory tied to the value. This means no garbage collection is required, which helps to prevent memory leaks and dangling pointers (Jung et al., 2018).

A computer screen shot of text

Description automatically generated

Figure 1 Ownership in Rust

Moreover, Rust provides a complex borrowing and lifetimes mechanism to enforce memory safety at compile time. Borrowing can be either mutable or immutable, depending on whether one is permitted or allowed to alter the value referred to by the references. Borrowed references allow functions to use data and edit it as they require without having to own it, thus allowing the reusing of code and efficient or optimal space organization. Innocuous use of borrows is allowed, but borrowing rules are enforced by the compiler in order to avoid data races and promote memory safety.

A computer code on a black background

Description automatically generated

Figure 2 Borrowing in Rust

On the other hand, lifetimes guarantees that references to memory is valid as long as the reference is required, ensuring that common issues such as use-after-free errors, data races and dangling pointers can be avoided. With this, Rust memory management system allows developers achieve an optimal of efficient and safe memory usage, through a synthesis of the stated mechanisms. This makes Rust especially useful for real-time programming, where stable memory demands are needed for most of the time.

A computer screen with white text

Description automatically generated

Figure 3 Lifetimes in Rust

## 4.2 Error Handling

It is widely recognized that any problem can happen at any time due to invalid user inputs, network problems, or coding mistakes. Error handling is the act of identifying the errors and dealing with them to ensure that the program does not freeze or that data is not destroyed (Ukeje, 2023). It enables the users to know if there is something wrong or likely to go wrong so that steps can be taken towards avoiding similar mistakes in the future (Nathan, 2022). In Rust, there are two types of errors: unrecovered and recovered errors. The errors in which a program ceases its execution are known as unrecoverable errors, in contrast to recoverable errors where programs can resume the execution while taking an action based on the type of error, as pointed out by (Soares, 2023). The unwrap method is used to retrieve value from Result or Option data structure when it exists. In general, it is relevant in cases where the developer believes that operation success is guaranteed, though, it is necessary to recognize that such approach can cause panics in case the result is an error or non-existence.

A computer code with many colorful text

Description automatically generated with medium confidence

Figure 4 Code Snippet of .unwrap() function

On the other hand, Result type represents either success (Ok) or failure (Err). It is commonly used for functions that may fail, allowing the caller to handle the error case directly.

A computer screen with text

Description automatically generated

Figure 5 Code Snippet of result () function

Similar to Result, Option is another Enum type that is often used when a value may or may not be present, providing a mechanism to handle the absence gracefully. Overall, the study highlights the importance of implementing error handling mechanisms to handle various error scenarios, ensuring smoother user experiences and higher software reliability.

A screen shot of a computer code

Description automatically generated

Figure 6 Code Snippet of option () function

## 4.3 Priority Inversion Handling

(Muller et al., 2023) highlighted that priority inversion is a scenario in which a less prioritize activity acquires a resource needed by a more prioritize activity and, therefore, results in unanticipated and potentially fatal delays. his is handled by Rust through synchronized blocks or mechanisms of mutual exclusion, whereby a programmer is able to strictly synchronize the shared resources. To enable synchronization, the standard library offers the Mutex and RwLock primitives. While effective in preventing data races, synchronized blocks can avoid priority inversion if a low priority task grabs a lock that is required by a high priority task. However, traditional mutexes may not adequately implement the notion of real-time system priorities where priority inversion greatly impacts deadlines (Muller et al., 2023). Furthermore, Priority Ceiling Emulation (PCE) assigns a priority ceiling to each shared resource and temporarily boosts the priority of a task to the priority level of the highest priority schedulable object. This prevents lower-priority tasks from preempting higher-priority tasks while holding a shared resource. On the other hand, Priority Inheritance (PI) temporarily inherits the priority of a higher-priority task with which it competes for a resource. This ensures that the task holding the resource does not get preempted by lower-priority tasks, thereby preventing priority inversion. It is worth noting that while PI is simpler to implement than PCE and dynamically adjusts task priorities based on resource contention, it can potentially lead to deadlock situations, especially in scenarios with cyclic dependencies among tasks and resources. Therefore, careful consideration and management are essential when implementing these strategies to avoid such issues.

## 4.4 Synchronous and Asynchronous Programming

Multi-threading and non-blocking I/O operations are well-supported in Rust, making it an ideal language for creating concurrent systems. Synchronous programming runs an activitiy after another, while asynchronous programming grants the freedom to run activities in parallel and not blocking. According to (Cutner & Yoshida, 2021), the language's standard library ‘std::sync::’ has a rich set of synchronization primitive using Rust’s ownership model to manage shared state and prevent data races. For instance, Mutexes ensure that only one thread can access a resource at a time so that concurrent alteration cannot happen. y providing a safe and effective way of passing data over the network, channels enable the exchange of messages between threads within a process. Semaphores make it possible for developers to implement complex access policies on the shared resource through counting mechanisms. Contrary, asynchronous programming can be implemented via ‘async’ and ‘await’, which makes the code look quite clean (Cutner & Yoshida, 2021). The async ecosystem in Rust is mainly composed of two libraries, the async-std and ‘tokio’ libraries that enable asynchronous I/O, task scheduling, and many other utilities for creating high-performance asynchronous applications (Wang et al., 2023). In this approach, the Rust developers use the “async’ and ‘await’ syntax together with such libraries as ‘async-std’ and ‘tokio’, which allow for creating robust applications that can respond to concurrent I/O operations and network requests without halting all other processes. This makes Rust to be a viable solution for server-side applications and other complex systems that require concurrency and fast responses.

## 4.5 Zero Cost Abstraction

Zero-cost abstraction allows developers to create high-level constructs effectively and at the same time do not affect the efficiency of the program at runtime levels (Sequeira, 2023). It allows the compiler to produce code that is optimal for the type used and the way it is used, avoiding any overhead related to abstraction due to static dispatch, generics and traits. This means that the performance characteristics of rust code is on par with low level languages such as C or C++, even when using expressive features. The optimized capability of Rust compiler also elevates this efficiency by making higher level optimizations like inlining and loop unrolling thus producing highly proficient machine code. In addition, Rust provides memory safety at compile time through ownership and borrowing mechanisms; there is no overhead run-time checks, which make the Rust program efficient (Muller et al., 2023). Thus, zero-cost abstraction in Rust not only helps create beautiful and easily readable code but also guarantees that performance will remain at the forefront, and for this reason, Rust is a perfect fit for systems programming and performance-critical applications.

## 4.6 Benchmarking Techniques

According to (Nathan, 2022), Benchmarking is best described as a process of evaluating Rust code at a certain level to understand how much (throughput) can be done or how quickly (latency) that can be accomplished. In Rust, the developers have a great set of benchmarking libraries that address different use cases such as micro-benchmarking and macro-benchmarking. Micro-benchmarking is the analysis of small, specific functions or specific instructions, which can be used to determine performance issues or optimize Rust applications. On the other hand, Macro-benchmarking focuses on the assessment of larger subsystems, hence offering information on system performance and grow-ability. In this context, a few very widely used tools like “bencher,” and “criterion” come into practice. “criterion” is a useful tool for developers that helps to profile Rust code for further optimization and shares information about performance-related features that are essential for applications with high demands on productivity, and stability(Pompeii, 2024). As RabbitMQ is to be used as the asynchronous messaging protocol between the producer and consumer components, using benchmarking tests becomes important in determining the overall performance of the messaging system. By integrating benchmarking tools like "bma\_benchmark" or "criterion" into their workflow, developers can iteratively optimize code and infrastructure to meet the demands of real-time systems effectively, balancing performance, reliability, and efficiency.

# 5.0 Research Design and Methodology

A screenshot of a computer screen

Description automatically generated

Figure 7 Stock Trading System Architecture Design

The figure above shows the system architecture of a real-time stock trading system. The system is divided into two subsystems: SS1: Trade Processing and SS2: Stock Market Processing. There are three entities involved in the stock trading process, which are trader, stockbroker and stock, and each of them play a different role. In this simulation, RabbitMQ will act as a medium channel or intermediary, managing the message flow between the traders and broker, ensuring reliable and efficient communication.

## 5.1 Stock

It represents the actual stocks being traded, with variables such as stock name and current price. The stock price movement will depend on the market news, whether it is good, bad, or neutral news and the number of orders that are being processed by the brokers. Unemployment rate and GDP growth are the 2 conditions that affecting the market news in this simulation.

## 5.2 Trader

A trader is an individual or institution that participates in the financial market by buying and selling the securities on their own behalf or on behalf of clients. The trader acts as a consumer to request stock information based on the market news, specifying the stocks that they are interested in and any specific data they require. This information could include the current price, market trends, and news related to the stock. Based on the information received, the trader will act as the producer, analyses the market and employs predefined logic to determine optimal buy and sell prices for stocks. It will then initiate the trade requests by sending the order to the stockbroker, specifying details such as desire stock, prices, order type, and whether they want to buy or sell.

## 5.3 Stockbroker

The stockbroker is a marketplace where the actual buying and selling of stocks occur. It acts as a consumer, facilitating the matching of buy and sell orders and determines the prices at which trades are executed. Once the stockbroker receives the order request, it also acts as a producer to update stock quote based on the executed trades and other market data in real-time, ensuring everyone have access to the latest information. This process is typically completed within milliseconds on a real-time stock trading system. By passing on trade confirmations and updated prices, the stockbroker ensures that traders are promptly informed about the outcome of their trade requests (success/unsuccess) and the current market conditions, enabling them to make informed decisions and manage their trades effectively.

**Communication Flow**

1. Trader generates trade orders based on the buy/sell logic.
2. Trader sends trade order messages to the RabbitMQ queue “stock\_order”.
3. Stockbroker retrieves the trade order messages from the RabbitMQ queue.
4. The stockbroker processes the trade order and updates the stock prices accordingly.

## 5.4 Code Snippet

A screen shot of a computer code

Description automatically generated

Figure 8 Struct Stock

A computer screen with text

Description automatically generated

Figure 9 Initial Stock in Stocklist

First of all, a vector with 5 stocks is defined with Stock struct (stock\_name and current\_price) in the stock.rs. The vec is protected and managed by RwLock, ensuring that only one trader or broker could update them at a time while others could read their values.

A computer screen with white text

Description automatically generated

Figure 10 Struct Trader

A screen shot of a computer program

Description automatically generated

Figure 11 Struct Broker

There are total of 5 traders, each tasked with generating 20 stock orders randomly. Each trader has an ID, with access to the stock list and latest market news. Meanwhile, the broker is responsible for processing the orders received from the traders correctly and updating the market price accordingly. Arc<Mutex<Vec<Stock>> ensures that the updates or changes made to the Vec< Stock> will be reflected in the original stock list while maintaining the thread safety. It allows multiple threads to safely access and modify the vector concurrently, preventing data races. Contrary, AtomicUsize and AtomicBool are used to ensure thread-safe access to the shared mutable state while providing atomicity.

A screen shot of a computer

Description automatically generated

Figure 12 determine\_market\_news() function

The stock price will be fluctuated based on the market factors such as unemployment rate and GDP growth. These market factors determined the overall market sentiment and were updated periodically to allow traders make decision whether to buy/sell. To determine the market news whether good, bad or neutral, an algorithm was predefined as follow:

* If unemployment rate is below 6.0 and GDP growth is more than 2.0 🡪 Good news
* If unemployment rate is more than 8.0 or GDP growth is less than 0.0 🡪 Bad news
* Else, market news is neutral.

A computer screen shot of a program code

Description automatically generated

Figure 13 Price adjustment based on market news

The stock’s price is adjusted based on the market news. If the market news is good, the price rises by 5%, if the market news is bad, the price drops by 5%, else, the price remains the same.

A screen shot of a computer code

Description automatically generated

Figure 14 generate\_order() function

A counter is initialised to keep track of the number of orders generated by the trader. The block of codes will continue loop until the trader has generated more than the ORDERS\_PER\_TRADER (20 order as mentioned earlier). Random delay is used to simulate time taken for the trader’s to make decision. The market factors will occur with a probability of 40% throughout this simulation. The updated market factors will be sent to the broker via channel and market news (‘Good’, ‘Bad’, or ‘Neutral’ will be printed to the console.

A screen shot of a computer code

Description automatically generated

Figure 15 Random select a stock and price update on market factors

The stock is selected randomly from the vector. The price will be adjusted based on the market news. Based on the random price change, the trader will determine whether to buy or sell.

* let mut stocks = self.stocks.write().unwrap();: This call acquires a mutable write lock on the vector of stocks, so only this thread can modify the contents.
* let stock\_index = rng.gen\_range(0..stocks.len());: Creates an integer represented a random index between 0 and the current length of the stocks vector.
* let stock = &mut stocks[stock\_index];: Conveys a mutable pointer to the desired stock within the vector based on a randomly generated index enabling alteration of the stocks’ attributes.
* price\_change is calculated randomly within the range of -20% to 20%.

A screen shot of a computer program

Description automatically generated

Figure 16 Trader buy/sell logic

If the price\_change is negative, it implies a decrease in price, and the trader will consider buying the stock. After the order is sent to the broker and being processed, the stock current price will increase 5% of the original (previous) price, and vice versa.

A screen shot of a computer program

Description automatically generated

Figure 17 Serialization

The stock order is then serialized to JSON format and sent to the queue for processing. This process will be repeated until the trader has generated the required number of orders. The, the stock price will be adjusted accordingly and prints the transaction details.

A computer code on a black background

Description automatically generated

Figure 18 Increment order count

The order counts are incremented and check if the stop condition is met. If yes, the stop\_signal is set to be true and breaks out of the loop.

A computer screen with text on it

Description automatically generated

Figure 19 start\_traders() function

It first creates an empty vector ‘handles’ to store the thread handles. For each trader, the function creates a new Trader instance with a unique ID and clones of the shared resources, then spawns a new thread to execute the generate\_order() function. The handles for these threads are stored in a vector. After all trader threads are spawned, the function waits for each thread to complete by calling join on each handle, ensuring that the main thread only continues once all traders have finished their tasks.

A computer screen shot of text

Description automatically generated

[1]

[2]

Figure 20 process\_order() fucntion

[1] It retrieves message from the ‘stock\_order’ queue (sent from trader).

[2] It atomically loads the current order count and checks if the total orders meet or exceed the expected total. If the target is reached, it will break the loop and stop generating more orders.

A screen shot of a computer program

Description automatically generated

[1]

Figure 21 Deserialization

[1] The broker deserializes the JSON string into the ‘Stock’ object. It then locks the stock list and searches for a matching stock by name in the current stock list, if found, the stock price will be updated with the new price from the order. The order that has been processed will be logged with current timestamp, stock name and new price.

A screen shot of a computer code

Description automatically generated

Figure 22 Simulation Process (Before)

The simulation initializes the market factors with unemployment rate 6.0% and GDP growth 2.5%. To keep track of the transactions number and to control the market flow, the order\_count and stop\_signal is created. A channel is set up for sending the market updates from trader to the broker. The “MARKET OPENS…..” is printed as an announcement to start the simulation.

A screen shot of a computer code

Description automatically generated

Figure 23 Broker processing thread

The stocks are cloned for broker’s personal use, such as managing the incoming orders. With the help of ‘Mutex’, the broker ensures that only one order was processed at a time. The broker thread was spawned, embarking on its mission to handle stock orders diligently.

A screen shot of a computer

Description automatically generated

Figure 24 Trader thread

The simulation starts trader thread by calling the start\_traders(). Then, the broker listened to the communication channel and processed each order, updating the stock prices as needed to ensure that the market reflected the latest activities and conditions. To ensure the main thread waits for the broker to finish processing all the orders, the .join() function is used. To handle any potential errors that might occur while joining the thread, unwrap() function is used. Once the orders are all executed, the market announced its closure by printing the message “MARKET CLOSED…”.

# 6.0 Results and Discussion (Benchmarking)

## 6.1 Stock Generator

Code Snippet

**A computer screen shot of text

Description automatically generated**

Figure 25 Stock Generator Benchmark

### Section 1 – TP060875

A black screen with yellow and white text

Description automatically generated

Figure 26 Benchmark Result for Stock Generator

The benchmark measures the performance of the stock\_generator function, which simulate the process of generating a specified number of Stock objects. It is tested with an iteration of 100 stocks, and among the 100 measurements, there are only 1% of outlier (high severe), potentially due to external factors like system interruptions or background processes.

A graph and chart with numbers

Description automatically generated with medium confidence

Figure 27 Stock Generator Index Report

The estimated slope time is 729.06 ns. The R2 value is approximately 0.65, indicating moderately strong relationship between the iterations and the total execution time. The density plot shows the distribution of the average execution times centred around 730.84 ns. Overall, the performance of stock\_generator function remains reliable, with most execution times falling within the expected range. This indicates that while the function generally performs consistently, there are occasional anomalies that can cause significant deviations in the execution times.

### Section 2 – TP061818

A screenshot of a computer screen

Description automatically generated

Figure 28 Benchmark - TP061818 (Version 2)

A diagram and graph of a function

Description automatically generated with medium confidence

Figure 29 Index Result - TP061818 (Version2)

**Justification**

The operation in Section 1 is significant faster, with an average time in 714.77ns, whereas Section 2 takes longer time, with 33.906µs. Lesser presence in outliers in Section 1 also indicates that the operation is more consistent that and has less variance in execution time compared to Section 2. In overall, Section 1 operations allows for quicker generation of a large number of Stock objects compared to Section 2.

## 6.2 Simulation Orders per Trader (20 orders)

Code Snippet

A computer screen shot of text

Description automatically generatedA computer code on a black background

Description automatically generatedA black rectangle with white text

Description automatically generated

Figure 30 Simulation Orders per Trader Benchmark

### Section 1 – TP060875

A screen shot of numbers and text

Description automatically generated

Figure 31 Benchmark Result for Simulate Orders per Trader

The benchmark measures the performance of the orders\_per\_trader function, which simulate the trading activity for one trader executing 20 orders. The function exhibits improved performance with an average execution time of 238.227µs per iteration, representing a significant improvement of 11.618%. There are 4% of outliers detected among 100 measurements, including 2 high mild and 2 high severe.

A graph and chart of a graph

Description automatically generated with medium confidence

Figure 32 Simulation Orders per Trader Index Result

The positive slope of 238.27µs indicates the total execution time increases linearly with the number of iterations. The R2 value of 0.54 suggests a moderate fit of the regression line to the data points, indicates there are some other factors that cause variability in the execution time. Despite the presence of a few high-severity outliers, the overall performance is stable and predictable.

### Section 2 – TP061818

A screen shot of numbers

Description automatically generated

Figure 33 Benchmark - TP061818 (Version2)

A graph and chart of a graph

Description automatically generated with medium confidence

Figure 34 Result Index - TP061818 (Version2)

**Justification**

Despite Section 1 implementation having a higher mean execution time, but it demonstrates a positive performance trend and better predictability, showcasing an overall improvement in performance with better predictability (R² value) and fewer outliers. On the other hand, although Section 2 implementation faster on average, but it has regressed in performance and has less predictability.

## 6.3 Determine Market News/Condition

Code Snippet



Figure 35 Determine Market News Benchmark

### Section 1 – TP060875

A black screen with white text

Description automatically generated

Figure 36 Benchmark Result for Determine Market News

The benchmark measures the performance of determine\_market\_news function, which simulates the determination of market news based on randomized unemployment rate and GDP growth rates. The performance has regressed, with an increase in average execution time of approximately 5.33%. The presence of 2 high mild outliers suggests that some measurements deviated from the average but do not significantly affect the overall conclusion.

A diagram and graph with numbers

Description automatically generated with medium confidence

Figure 37 Determine Market News Index Result

The operation has a mean execution time of 39.94ns, with some performance regression observed. The density plot’s peak is around 39.453ns, indicates that most measurements fall within this range. The consistent linear trend over iterations suggests stable performance, however, the moderate R2 value (~0.59) indicates that other factors might be affecting the total sample time.

### Section 2 – TP061818

A screen shot of a computer

Description automatically generated

Figure 38 Benchmark - TP061818 (Version2)

A graph and chart of a market condition

Description automatically generated

Figure 39 Index Result - TP061818 (Version2)

**Justification**

The Section 2 function is more efficient with faster execution times (28.136ns), but it has greater variability and less predictability. The Section 1 function is slower (39.453ns) but demonstrates better consistency and predictability, making it more reliable for scenarios where stable performance is critical. If raw execution speed is the priority, the Section 2 function is preferable. However, for applications requiring consistent and predictable performance, the Section 1 function is more suitable.

## 6.4 Affect Price Benchmark

Code Snippet

A screenshot of a computer program

Description automatically generated

Figure 40 Affect Price Benchmark

### Section 1 – TP060875

A black screen with yellow and green text

Description automatically generated

Figure 41 Benchmark Result for Affect Price

The benchmark measures the performance of affect\_price function, where simulates the stock price fluctuations based on different market news. It shows the operation has a consistent and efficient performance with an average execution time of ~17.083ns, supported by narrow confidence intervals. There are 2 high mild outliers found out of the 100 measurements.

A diagram and graph of a function

Description automatically generated with medium confidence

Figure 42 Affect Price Index Result

The mean (170.54ns) and median (17.003ns) values are close, suggesting that the data is reasonably centered around the mean. The R² coefficient 0.4672347 shows a moderate correlation, indicating that 46.72% if the variance in total sample time can be explained by the number of iterations.

### Section 2 – TP061818

A black background with white text and red and blue letters

Description automatically generated

Figure 43 Benchmark - TP061818 (Version2)

A diagram and graph with numbers

Description automatically generated with medium confidence

Figure 44 Index Result - TP061818 (Version2)

**Justification**

Section 1 has lower execution time, indicating the implementation is faster in simulating the stock adjustment compared to Section 2. The function in Section 1 is also more stable and less variable due to its lower standard deviation. However, Section 2 outperforms if predictability is priority in the simulation as it has higher R2 value (0.59).

## 6.5 Trader’s Buy or Sell Decision Benchmark

Code Snippet

A screen shot of a computer screen

Description automatically generated

Figure 45 Trader Buy/Sell Decision Benchmark

### Section 1 – TP060875

A screen shot of numbers

Description automatically generated

Figure 46 Benchmark Result for trader\_decision

The benchmark measures the performance of a trader’s decision in buy/sell stock. It execution times are consistently between 20.923ns and 22.900ns, with a mean of 21.883ns. With a p-value of 0.37, it shows no significant change detected. Out of 100 measurements, there are three outliers, and all are classified as high mild, meaning they are higher than the typical values but not extremely deviant.

A graph and chart with numbers and text

Description automatically generated with medium confidence

Figure 47 trader\_decision index result

The slope is estimated at 21.883ns, with a lower bound of 20.923ns and an upper bound of 22.900ns. The coefficient of determination is 0.51738328, indicating a moderate fit of the linear regression model to the data with some variability due to external factors, such as transient system load, or background processes.

### Section 2 – TP061818

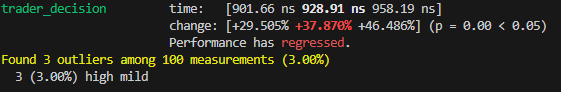


Figure 48 Benchmark - TP061818 (Version2)

A diagram and graph of a function

Description automatically generated with medium confidence

Figure 49 Index Result - TP061818 (Version2)

**Justification**

In short, section 1 function is superior due to its lower execution time (21.283ns faster than 987.37ns), better consistency (standard deviation with estimated 3.6450 ns, lower than 139.87 ns), and higher predictability (R2 value higher than 0.24), making it a better choice for scenarios requiring quick and stable buy/sell decisions.

## 6.6 Broker Process Order Benchmark

Code Snippet

A screen shot of a computer code

Description automatically generated

Figure 50 process\_order benchmark

### Section 1 – TP060875

A screen shot of a computer

Description automatically generated

Figure 51 Benchmark Result for process\_order

The benchmark measures the performance of the process\_order function, simulates how the stock is being processed by the broker. The performance change has an average increase of 4.2259%, and p-value is 0.04, indicating that the observed changes in performance are not due to random noise. There are total of 6 outliers among the 100 measurements, with 1 low mild, 2 high mild, and 3 high severe, showcasing occasional significant deviations from the norm.

A graph and chart with numbers

Description automatically generated with medium confidence

Figure 52 process\_order index result

The density plot shows most values centered around the average time of 16.913µs. The R² value is 0.4315344, indicating that approximately 43.15% of the variability in execution time is explained by the number of iterations. This suggests a moderate fit, meaning there is a significant amount of variability not explained by the linear model. In the scatter plot, we can see that the points closely follow the regression line, showing consistent performance over multiple iterations.

### Section 2 – TP061818

A screen shot of numbers

Description automatically generated

Figure 53 Benchmark - TP061818 (Verison2)

A diagram and graph with numbers

Description automatically generated

Figure 54 Index Result - TP061818 (Version2)

**Justification**

Implementation in Section 1 is faster in processing stock orders, with a lower mean execution time 17.208µs. In terms of predictability and reliability, Section 2 outperforms by having higher R2 value (0.6587873), despite its significant regression in performance.

## 6.7 Overall Simulation Benchmark

Code Snippet

A screen shot of a computer code

Description automatically generated

Figure 55 Overall Simulation Benchmark I

A screenshot of a computer

Description automatically generated

Figure 56 Overall Simulation Benchmark II

### Section 1 – TP060875

A screen shot of a computer

Description automatically generated

Figure 57: Benchmark Result for Overall Simulation

The overall\_simulation function shows a significant improvement in performance, with execution times reduced by an average of 38.845%. This improvement is statistically significant, as indicated by the p-value of 0.00. However, the presence of 10 outliers (10%) suggests some variability in the performance, with occasional significant deviations. Overall, the function demonstrates enhanced efficiency, making it better suited for real-time applications requiring quick and reliable execution.

A graph and chart of a graph

Description automatically generated with medium confidence

Figure 58: Overall Simulation Index Result I

The density graph shows the distribution of execution times, with most values centered around the median time of 1.1255ms. The scatterplot, on the contrary, displays the total sample time against the number of iterations, indicating a general trend of increasing time with more iterations. However, the R² value is quite low, indicates that the linear regression model only weakly explains the variability in execution times. The presence of high severe outliers suggests that while the overall performance has improved, there are occasional significant deviations in execution time.

A graph and diagram of a graph

Description automatically generated

Figure 59 Overall Simulation Index Result II

The performance has significantly improved, as evidenced by the substantial reduction in execution times (approximately 38.845%) and a statistically significant p-value. Based on the right plot, the lower position of the blue regression line and the narrower confidence interval clearly show that the overall\_simulation function now executes more efficiently, with lower total sample times for the same number of iterations.

### Section 2 – TP061818

A screen shot of a computer

Description automatically generated

Figure 60 Benchmark - TP061818 (Version2)

A graph and chart with numbers and text

Description automatically generated with medium confidence

Figure 61 Index Result I - TP061818 (Version2)

A graph and graph chart

Description automatically generated

Figure 62 Index Result III - TP061818 (Version2)

**Justification**

Both implementations have a relatively high number of outliers, but Section 2 function has more severe outliers (12.00% vs. 10.00%). This suggests that Section 1 implementation is slightly more stable. Despite being consistent, Section2 is far less efficient and has shown no significant improvement in performance. Section 2 also showcases its poorer performance with longer execution time, probably due to the usage of channel in benchmarking.

# 7.0 Conclusion

In conclusion, this project achieved its goal of designing a real-time stock trading simulation using Rust and RabbitMQ for the management of high-performance, low-latency trading. The detailed literature review has helped to develop the theoretical framework which, in turn, has supported the research. Moreover, the specification of research design and methodology has contributed to building a strong system design and implementation framework. The benchmarking results have shown the efficiency of the overall system, and it is apparent that Rust is suitable for a real-time system to be designed for stock trading system. It should however be noted that other parameters as system load, network conditions and the nature of operation types preempting and modifying latency jitter. It would be possible for further work to analyze more markets and trading strategies or movements and possible improvements to the system. All in all, the project demonstrates Rust’s strengths for building real-time stock trading platforms and highlights the benefits that Rust can bring to trading applications in terms of speed, concurrency, and low-latency execution of trading operations.

# 8.0 References

Al-Aubidy, K. (2011). *Real-Time Systems Lecture (2) Classification of Real Classification of Real Classification of Real Classification of Real--Time Systems Time Systems Time Systems Time Systems.* Computer Engineering Department.

Cutner, Z., & Yoshida, N. (2021). Safe Session-Based Asynchronous Coordination in Rust. *International Conference on Coordination Languages and Models*, 80-89. https://doi.org/10.1007/978-3-030-78142-2\_5i

Jung, R., Jourdan, J.-H., Krebbers, R., & Dreyer, D. (2018). RustBelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages, 66*(2), 34. https://doi.org/10.1145/3158154

Khachatryan, G. (2023, January 30). *About Rust Programming Language*. Retrieved from DEV Community: https://dev.to/grigorkh/about-rust-programming-language-36ac

Laplante, P., & Ovaska, S. (2012). *REAL-TIME SYSTEMS DESIGN AND ANALYSIS www.it-ebooks.info.* A JOHN WILEY & SONS, INC.

Muller, S., Singer, K., Keeney, D., Neth, A., Argawal, K., Lee, I.-T., & Acar, U. (2023). Responsive Parallelism with Synchronization. *Proceedings of the ACM on Programming Languages, 7*(135), 24. https://doi.org/10.1145/359124

Nathan. (2022, December 15). *How to Handle Errors in Rust: A Comprehensive Guide*. Retrieved from Dev: https://dev.to/nathan20/how-to-handle-errors-in-rust-a-comprehensive-guide-1cco

Pompeii, E. (2024, January 28). *How to benchmark Rust code with Criterion*. Retrieved from Bencher: https://bencher.dev/learn/benchmarking/rust/criterion/

Qin, B., Chen, Y., Yu, Z., Song, L., & Zhang, Y. (2020). Understanding memory and thread safety practices and issues in real-world Rust programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 763-779. https://doi.org/10.1145/3385412.3386036

Sequeira, R. (2023, May 23). *Learning Rust: Understanding Zero-Cost Abstraction with Filter and Map*. https://doi.org/https://ranveersequeira.medium.com/learning-rust-understanding-zero-cost-abstraction-with-filter-and-map-e967d09fff79

Soares, L. (2023, May 24). *Rust Error Handling In Practice*. Retrieved from medium: https://medium.com/coinmonks/rust-error-handling-in-practice-376d86ba12ca

Ukeje. (2023, March 29). *makeuseof.com*. Retrieved from Handling Errors in Rust With Various Approaches: https://www.makeuseof.com/rust-error-handling-approaches/

Wang, T.-Y., Wang, S.-H., Tu, C.-H., & Liang, W.-Y. (2023). CAT: Context Aware Tracing for Rust Asynchronous Programs. *Proceedings of the ACM Symposium on Applied Computing*, 483-492. https://doi.org/10.1145/3555776.3577669

Wróbel, K. (2023, March 27). *Why is Rust programming language so popular?* Retrieved from codilime: https://codilime.com/blog/why-is-rust-programming-language-so-popular/

Zuss, N. (2021, July 2). *PRIMER: high frequency trading*. Retrieved from IFLR: https://www.iflr.com/article/2a646mpixhlq5fir64agw/primer-high-frequency-trading