INDIVIDUAL ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT087-3-3-RTS

REAL TIME SYSTEM

STUDENT NAME	:	SOO JIUN GUAN
TP NUMBER	:	TP068687
INTAKE CODE	:	APD3F2408CS(DA)
LECTURER NAME	:	ASSOC. PROF. DR. IMRAN MEDI
HAND OUT DATE	:	18TH OCTOBER 2024
HAND IN DATE	:	22ND DECEMBER 2024

Abstract

Real-time systems need to achieve predictable performance under tight timing constraints. Rust provides low latency and robust error handling in such applications by offering its ownership model and memory safety. This paper presents the use of Rust in developing a real-time trading simulation within a context that is performance and safety-intensive. The project will demonstrate that Rust has huge potential to be a trustworthy substitute for traditional languages like C and C++.

Table of Contents

1.0 Introduction

Real-time systems are usually computing systems that react within a strict time constraint after an input or event occurrence, thus guaranteeing dependable and predictable performance. Unlike general-purpose systems, they give priority to the timeliness of responses over throughput (Ravindran et al., 2005). They are indispensable in critical domains, including but not limited to space, health care, and industrial automation.

These are broadly categorized into hard real-time, where failure to meet deadlines is considered a failure, and soft real-time, where delay degradation impacts performance but is not catastrophic (Devi, 2006). Real-time systems fulfill the tight timing requirements by the use of special hardware, operating systems, and scheduling algorithms.

The key points in designing real-time systems involve predictability, dependability, and efficiency. In modern technology, as in the Internet of Things, their incorporation is common in research related to energy efficiency and scalability issues (Tien, 2017). This advancement continues to safely enable very precise technology-to-physical-world interaction, which continues to underpin these types of mission-critical and time-sensitive applications.

2.0 Literature Review

2.1 Introduction

Rust is a modern systems language programming that meets both safety and performance challenges, hence highly suitable for real-time systems. Enforcing memory safety and data race prevention at compile time, Rust prevents many common bugs without sacrificing execution speed. With its unique ownership model, a strong type system, and the ability to write low-level code, Rust is a competitive alternative with C and C++ in performance-critical environments. However, real-world adoption has identified several limitations: unsafe code reliance, usability challenges, and complex formal verification processes are some of the concerns. This review aims to present an overview of Rust's strengths, limitations, and applicability to real-time systems.

2.2 Strengths of Rust in Real-Time Systems

2.2.1 Safety and Performance

Rust's ownership model and borrow checker ensure memory safety at compile time, which solves many problems related to null pointers, buffer overflows, and data races. Unlike garbage-collected languages, Rust does not incur runtime overhead, ensuring low latency and predictable execution critical for real-time systems (Bugden & Alahmar, 2022).

RustBelt is a formal verification project which proves Rust's type system is sound, including the interactions with unsafe code (Jung et al., 2017). These theoretical foundations provide yet another guarantee that undefined behavior is prevented by Rust with no penalty in execution time. Bugden and Alahmar (2022) support it by benchmark studies confirming that Rust reaches execution speeds similar to C and C++ but higher than Go and Python. This combination makes Rust an ideal choice for resource-constrained systems such as embedded devices.

2.2.2 Fault Tolerance and Error Handling

Robust error handling in Rust offers fault tolerance due to its Result and Option types, which explicitly force the developer to handle the errors. As opposed to implicit error handling in other languages, it ensures that all possible errors can be handled at compile time, hence

reducing runtime failures (Astrauskas et al., 2020). Additionally, the "?" operator simplifies error propagation, hence making the code cleaner and more reliable.

Lagaillardie et al. (2022) has presented Affine Multy Party Session Types (AMPST), an extension for Rust that guarantees communication safety in distributed systems. AMPST guarantees the deadlock freedom and the absence of cascading failures due to communications errors of processes, crucial for fault-tolerant real-time systems where recovery mechanisms must be robust.

2.2.3 Controlled Use of Unsafe Code

While Rust enforces safety, it provides the unsafe keyword for activities that require low-level control, such as hardware manipulation or performance optimization. Encapsulation of unsafe code means it will be confined to specific parts of a program; thus, safety guarantees can be retained in other parts. Levy et al. (2015) have shown that even when implementing OS kernels, Rust minimizes the amount of unsafe code the programmer needs to write.

Evans et al. (2020) also notice that only 30% of Rust libraries use unsafe code directly, while over 50% have indirect propagation through dependencies. While the former is an extremely important challenge to Rust safety claims, its ability to restrict and audit unsafe operations gives them a strong advantage compared to other language alternatives such as C.

2.3 Challenges of Rust in Real-Time Systems

2.3.1 Unsafe Code Propagation

Although unsafe code is restricted in Rust, it tends to creep into a project via dependencies. Evans et al. (2020) have shown that more than half of Rust libraries depend indirectly on unsafe operations, mostly unbeknownst to the developer. Qin et al. (2020) have reported bugs that are due to subtle interactions between safe and unsafe code, hence undermining the safety guarantees provided by Rust.

While formal tools like RustBelt have been used to prove the safety of Rust's type system, these proofs focus on simplified cases and do not fully cover complex real-world

programs. This mismatch between theory and practice raises serious questions about the reliability of Rust's guarantees in large-scale systems (Qin et al., 2020).

2.3.2 Steep Learning Curve

While Rust's ownership model is powerful, this also provides a highly burdensome for developers. The borrower checker is a major feature to ensure safety, but it is highly frustrating to many programmers that do not have much experience with such strict rules on ownership and borrowing with lifetime in Rust. Such a steep curve thus slows the pace of adoption because teams may have prior experience only with C and C++ languages (Zhu et al., 2022).

These challenges are especially highlighted for rapid development. For example, Qin et al. (2020) note challenges that developers face in handling thread safety while doing concurrent programming. Many developers find a balance between Rust's strict ownership rules and the requirement of efficiently sharing data access. Workarounds often involve leveraging unsafe code to sidestep such limitations. While Rust enforces safety, these restrictions can slow development and increase complexity in real-world systems.

2.3.3 Usability of Formal Verification

The theoretical soundness of Rust, as demonstrated by RustBelt, holds only for simplified and idealized subsets of the language (Jung et al., 2018). In real-world systems, the need for unsafe code arises when accessing low-level hardware or optimizing performance-critical operations, tasks that cannot always align perfectly with Rust's safety guarantees. Qin et al. (2020) point out that these practical requirements introduce subtle bugs, especially in concurrent programs, due to complex interactions between safe and unsafe code leading to unexpected behaviors.

For instance, one might expect thread synchronization problems or other resource sharing issues to show up in systems in which the developers have used unsafe code to circumvent ownership rules. These bugs are often very hard to find and debug because verification tools typically exclude unsafe code from their analysis (Qin et al., 2020). This creates a gap between Rust's formal guarantees and its real-world applicability: there is a clear need for better tools to improve visibility, enhance auditing, and provide safer integration of unsafe operations that do not compromise the safety of the language.

2.4 Conclusion

Rust combines safety, performance, and reliability in a unique way, making it a real candidate for real-time systems. Its ownership model, borrow checker, and formal safety proofs are a big plus compared to more conventional languages like C and C++. Rust eliminates many common programming errors, enhances fault tolerance due to its robust error handling, and provides performance suitable for resource-constrained systems. However, some challenges remain. Propagation of unsafe code in dependencies, steep learning curve of Rust, and gaps between formal verification and practical use still limit its adoption at large. Future research and improvements in developer ergonomics, visibility of unsafe operations, and safety guarantees in more complex real-world systems could be further extended. With a resolution to these challenges, Rust could easily raise the bar regarding safety, fault tolerance, and performance in real-time systems to then give reason for wide use as a safer alternative to older languages.

3.0 Outline of Simulations

3.1 Concurrency

This simulation will be concerned with concurrency, making use of asynchronous tasks in addition to shared state management-important in modeling a realistic environment of trading. The system is designed to ensure independence in the operations of the traders while maintaining synchronized access to shared resources, such as the stock market data. It also includes a centralized messaging system, RabbitMQ, for decoupled communication among components.

1. Concurrent Trader Tasks

```rust
// Spawn tasks for each trader to simulate concurrent trading
let mut trader_handles: Vec<JoinHandle<()>> = vec![];
for mut trader: Trader in traders {
    let trader_stop_signal: Arc<AtomicBool> = stop_signal.clone();
    trader_handles.push(task::spawn(future: async move {
        trader.start(trader_stop_signal);
    }));
}
```

Figure 1: Concurrent Trader Task Execution

Traders are independent; each of their trading logics runs concurrently, using Tokio's asynchronous runtime. Each trader would execute as a separate task, thereby enabling them to buy or sell without causing others to block. It can be compared, in some ways, to the real world, where many traders act upon the market at one and the same time.

2. Shared State Management

```rust
// Shared state for stocks and order book, protected with Mutex for safe concurrent access
let stocks: Arc<Mutex<Vec<Stock>>> = Arc::new(data: Mutex::new(vec![]));
let order_book: Arc<Mutex<OrderBook>> = Arc::new(data: Mutex::new(OrderBook::new(stocks: vec![], tr
let stop_signal: Arc<AtomicBool> = Arc::new(data: AtomicBool::new(false)); // Shared stop signal fo
```

Figure 2: Handling common resources

Shared resources, for example, the stock market data in stocks, are shared using Arc<Mutex<>>. This means thread-safe operations, no race conditions while traders and the consumer workflow can share and update the market data concurrently.

```rust
let stock_price: Option<(i32, f64, f64)> = {
    let stocks: MutexGuard<'_, Vec<Stock>> = self.stocks.lock().unwrap();
    if let Some(stock: &Stock) = stocks.iter().find(|stock: &&Stock| stock.name == stock_name)
        Some((quantity, stock.price, buy_trigger))
    } else {
        None
    }
};
```

Figure 3: Lock the mutex

The traders will access and update the shared state, acquiring the mutex lock to ensure mutual exclusion in the critical operations.

3. Asynchronous Messaging

```rust
tokio::spawn(future: {
    let channel: Arc<Channel> = self.rabbitmq_channel.clone();
    let order_json: String = order_json.clone();
    async move {
        let _ = channel Arc<Channel>
            .basic_publish(
                exchange: "",
                routing_key: "orders", // Publish to "orders" queue
                options: BasicPublishOptions::default(),
                payload: order_json.as_bytes(),
                properties: BasicProperties::default(),
            ) impl Future<Output = Result<…, …>>
            .await Result<PublisherConfirm, …>
            .expect(msg: "Failed to publish order");
    }
});
```

The simulation uses RabbitMQ for asynchronous communication between the different components. Traders publish 'Buy' or 'Sell' orders to the order queue, decoupling their operation from order processing. This is how the simulation can be extended to increase with any number of additional traders or components.

4. Graceful Shutdown

```rust
while !stop_signal.load(order: Ordering::Relaxed) {
    // Perform trading actions
```

Figure 4: End using AtomicBool

In this simulation, the AtomicBool flag acts as a globally shared signal for all simultaneous tasks to stop gracefully together. Every trader regularly verifies the stop_signal for any expiration of simulation time.

3.2 Scheduling

This simulation encompasses periodic and aperiodic scheduling to deal with the tasks ranging from regular actions of trading to handling unpredictable events.

1. Periodic Scheduling

```
let start_time: Instant = Instant::now();
while start_time.elapsed() < Duration::from_secs(60) {
    if let Some(delivery: Result<Delivery, Error>) = consumer.next().await {
        let delivery: Delivery = delivery.expect(msg: "Error in consumer");

        // Deserialize stock update message
        let stocks_data: Vec<Stock> = serde_json::from_slice(&delivery.data)?;
        println!("📩 Received stock updates");

        // Update shared stock data
        {
            let mut shared_stocks: MutexGuard<'_, Vec<Stock>> = stocks.lock().unwrap();
            *shared_stocks = stocks_data.clone();
        }

        // Acknowledge the message
        channel Arc<Channel>
            .basic_ack(delivery.delivery_tag, options: BasicAckOptions::default()) impl Future<Output = Result<…, …>>
            .await?;
    }
}
```

Figure 5: Routine trading actions by traders

Periodic scheduling ensures that the stock market prices are kept current for traders and continuously take trading actions with every interval. This is through setting up a time-based interval inside a loop, at which period the trader processes new, refreshed, and real-time stock market data from the stock subsystem and updates its decisions. Place the producer, which represents the stock subsystem publishing updates into the queue every 5 seconds, with a flow wherein it ensures that the consumer (trading subsystem) receives such updates timely. This approach maintains a steady cadence for decision-making, ensuring that trading activities occur predictably and remain synchronized with periodic market updates.

2. Sporadic Scheduling

```rust
pub fn update_portfolio(&mut self, stock_name: String, quantity: usize, price: f64, is_buying: bool) {
    let entry: &mut (usize, f64) = self.portfolio.entry(key: stock_name.clone()).or_insert(default: (0, price));

    if is_buying {
        // When buying, add quantity and update price
        let total_quantity: usize = entry.0 + quantity;
        let total_cost: f64 = (entry.0 as f64 * entry.1) + (quantity as f64 * price);
        entry.0 = total_quantity;
        entry.1 = total_cost;
        println!(
            "📈 Portfolio Update - Buy: Trader {} now holds {} shares of {}",
            self.id, total_quantity, stock_name
        );
    } else {
        // When selling, subtract quantity
        if entry.0 <= quantity {
            // If selling all or more than owned, remove from portfolio
            self.portfolio.remove(&stock_name);
            println!(
                "📉 Portfolio Update - Sell: Trader {} sold all shares of {}",
                self.id, stock_name
            );
        } else {
            // Otherwise just reduce quantity, keep same price basis
            entry.0 -= quantity;
            println!(
                "📉 Portfolio Update - Sell: Trader {} now holds {} shares of {} ",
                self.id, entry.0, stock_name
            );
        }
    }
} fn update_portfolio
```

Figure 6: Update portfolio function

This process of reactive portfolio updates due to dynamic order matching within the `OrderBook` (in another subsystem) is scheduled sporadically. For example, immediately after a trader's `Buy` or `Sell` order has been matched against another trader's order, their portfolios are updated. This inherently event-driven behavior depends on the unpredictable arrival and attributes of new orders, such as price, quantity, and stock type. This will be implemented as a simple matching logic working its way through the current state of the `OrderBook` as new orders are added, processing matches where they occur. If it finds a match, then the `update_portfolio` function for both traders is invoked so that their portfolio reflects the outcome of this transaction. The sporadic scheduling here introduces reactivity and dynamism into this simulation whereby updates to the portfolios are tied to market events rather than occurring on any regular schedule.

```rust
if market_price >= 0.98 * purchase_price && market_price <= 1.02 * purchase_price {
    let mut rng: ThreadRng = rand::thread_rng();

    // Randomly decide whether to sell or not
    if rng.gen_bool(0.5) {
        // Decide the quantity to sell: all, half, or quarter
        let sell_fraction: f64 = match rng.gen_range(0..=2) {
            0 => 1.0,  // all shares
            1 => 0.75,  // 3/4
            _ => 0.5, // half
        };

        let sell_quantity: usize = (current_quantity as f64 * sell_fraction).round() as usize;

        println!(
            "💰 Trader {} decided to sell {} shares of {} at ${:.2} each (Market Price Close to Stored Price).",
            self.id, sell_quantity, stock_name, market_price
        );

        // Place the sell order
        self.place_order(
            stock_name: stock_name.to_string(),
            sell_quantity,
            market_price,
            OrderType::Sell,
        );

        return Ok(()); // Successfully placed sell order
    } else {
        // Trader decides not to sell
        println!(
            "💰 Trader {} decided not to sell {} shares of {} at ${:.2} (Market Price Close to Stored Price).",
            self.id, current_quantity, stock_name, market_price
        );
        return Err(TraderError::ProfitIsLow {
            trader_id: self.id,
            stock_name: stock_name.to_string(),
            paid_price: purchase_price,
            current_price: market_price,
        });
    }
}
```

Figure 7: Condition that purchased price last time near to current market price when selling

The decision to sell the stocks is dynamically activated by the condition that the market price has to drop in a range of the purchase price the trader bought last time, making this event irregular and dependent on market fluctuations. The random choice to sell or not adds further to the uncertainty regarding when this action will happen.

3.3 Communication

The trading simulation has been split into two broad spheres of communication: one external via RabbitMQ and another internal between the traders and shared resources. This thereby allows creating a dynamic trading environment in real time, which would support both real-time updates and synchronized operations.

1. External Communications

External communication in the code uses RabbitMQ for doing asynchronous decoupled messaging between components. Two crucial RabbitMQ queues are used in this process:

Stock Price Updates:

```
// Start consuming messages from the "stock_market" queue
let mut consumer: Consumer = consumer_channel Arc<Channel>
    .basic_consume(
        queue: "stock_market",
        consumer_tag: "consumer_tag",
        options: BasicConsumeOptions::default(),
        arguments: FieldTable::default(),
    ) impl Future<Output = Result<…, …>>
    .await?;
```

Figure 8: Get stock market info

For continuous and updated stock prices, there is a queue called "stock_market". The main part of the consumer workflow, which listens for messages from this queue, continuously listens for such updates. In case of an update arrival, it is deserialized to a list of Stock data and written into shared `stocks` list. As the main consumer workflow, it listens continuously to messages on this queue. On reception, the update is deserialized to a list of stock data and written into the shared list of stocks.

Trader Orders:

```
tokio::spawn(future: {
    let channel: Arc<Channel> = self.rabbitmq_channel.clone();
    let order_json: String = order_json.clone();
    async move {
        let _ = channel Arc<Channel>
            .basic_publish(
                exchange: "",
                routing_key: "orders", // Publish to "orders" queue
                options: BasicPublishOptions::default(),
                payload: order_json.as_bytes(),
                properties: BasicProperties::default(),
            ) impl Future<Output = Result<…, …>>
            .await Result<PublisherConfirm, …>
            .expect(msg: "Failed to publish order");
    }
});
```

Figure 9: Send stock subsystem order for waiting to match

Each trader asynchronously sends its sell and buy orders to the Orders Queue. The orders will be serialized into JSON for sending to RabbitMQ to decouple the trader's issuance of orders from the core processing or matching logic of the orders. This permits scalability because one

can scale up the traders or scale up the order processor without being forced to change core simulation logic.

2. Internal Communications

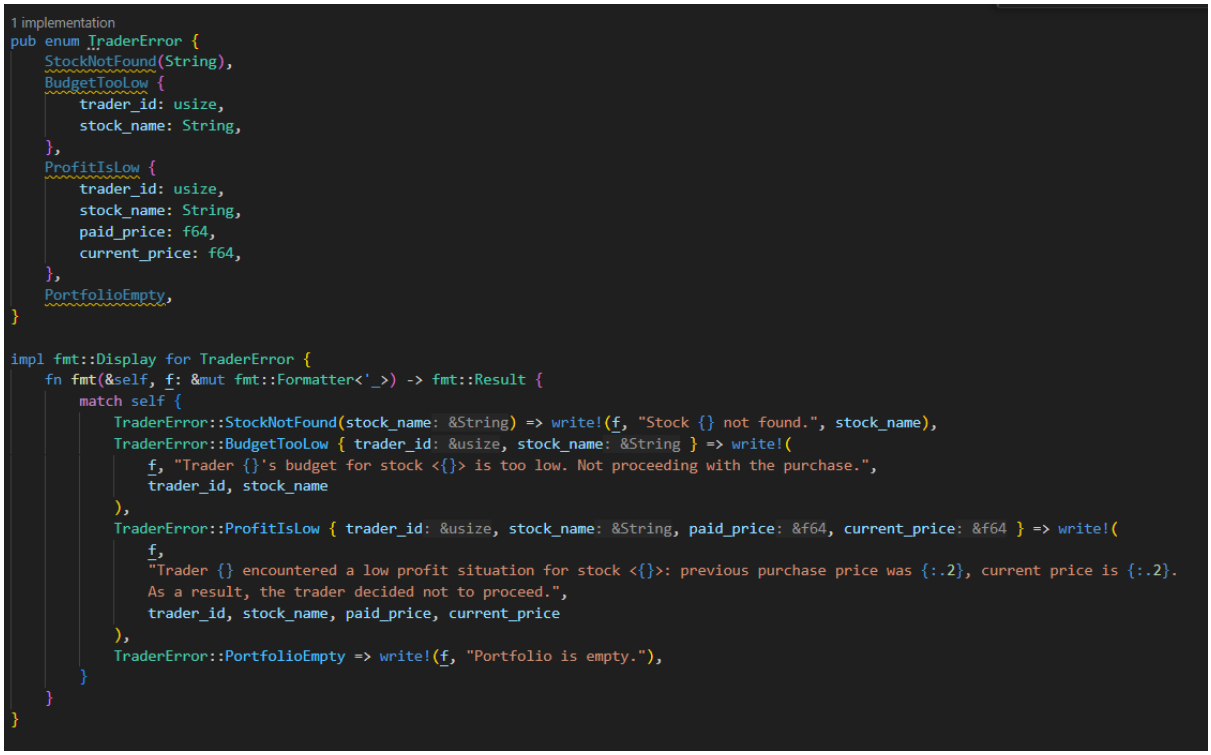
Traders internally depend on the shared stocks list to make trading decisions, and this stock list has to be shared across all traders. It will hold the real-time market data which updates dynamically. This is wrapped in an Arc<Mutex<>> to provide thread safety without race conditions in a concurrent environment. This list is accessed by the traders through a mutex lock to fetch the latest stock information that decides whether to buy or sell. This will ensure synchronized access so that all traders will operate on the same market data, thus enabling them to perform realistic and dynamic trading.

```
let stock_price: Option<(i32, f64, f64)> = {
    let stocks: MutexGuard<'_, Vec<Stock>> = self.stocks.lock().unwrap();
    if let Some(stock: &Stock) = stocks.iter().find(|stock: &&Stock| stock.name == stock_name) {
        Some((quantity, stock.price, buy_trigger))
    } else {
        None
    }
};
```

Figure 10: Accessing the stocks list

3.4 Error and Event Handling

1. Error Handling



```
1 implementation
pub enum TraderError {
    StockNotFound(String),
    BudgetTooLow {
        trader_id: usize,
        stock_name: String,
    },
    ProfitIsLow {
        trader_id: usize,
        stock_name: String,
        paid_price: f64,
        current_price: f64,
    },
    PortfolioEmpty,
}

impl fmt::Display for TraderError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            TraderError::StockNotFound(stock_name: &String) => write!(f, "Stock {} not found.", stock_name),
            TraderError::BudgetTooLow { trader_id: &usize, stock_name: &String } => write!(
                f, "Trader {}'s budget for stock <{}> is too low. Not proceeding with the purchase.",
                trader_id, stock_name
            ),
            TraderError::ProfitIsLow { trader_id: &usize, stock_name: &String, paid_price: &f64, current_price: &f64 } => write!(
                f,
                "Trader {} encountered a low profit situation for stock <{}>: previous purchase price was {:.2}, current price is {:.2}.
                As a result, the trader decided not to proceed.",
                trader_id, stock_name, paid_price, current_price
            ),
            TraderError::PortfolioEmpty => write!(f, "Portfolio is empty."),
        }
    }
}
```

Figure 11: enum TraderError to deal with error during trading

This simulation defines and handles errors with the TraderError enum that categorizes issues such as missing stocks, insufficient budgets, low profits, or an empty portfolio. These errors are propagated as Result types, which allow functions either to handle them locally or propagate them upwards to higher levels of the program. As an example, if a trader tries to sell a stock that current market price close to his purchased price, a TraderError::ProfitIsLow error is returned and logged. It will also log errors to the console for feedback, enabling debugging and monitoring during simulation. This ensures issues that traders encounter do not crash the simulation but gracefully report and handle them, avoiding outrageous orders being stuck in the matching list.

Non-error event handling

The non-error event handling in the simulation mainly involves three two kinds of events: stock updates and trader decisions. These events drive the core functionalities of the system to dynamically respond to real-time data.

```rust
// Wait until the first stock update is received
while stocks.lock().unwrap().is_empty() {
    if let Some(delivery: Result<Delivery, Error>) = consumer.next().await {
        let delivery: Delivery = delivery.expect(msg: "Error in consumer");

        // Deserialize received data into stock structures
        let stocks_data: Vec<Stock> = serde_json::from_slice(&delivery.data)?;
        println!("📥 Received initial stock updates");

        // Update the shared stocks data
        {
            let mut shared_stocks: MutexGuard<'_, Vec<Stock>> = stocks.lock().unwrap();
            *shared_stocks = stocks_data.clone();
        }

        // Acknowledge the message
        channel Arc<Channel>
            .basic_ack(delivery.delivery_tag, options: BasicAckOptions::default()) impl Future<Output = Result<…, …>>
            .await?;
    }
}
```

Figure 12: stock update

Traders periodically evaluate the shared stocks list and their portfolios, making decisions to buy or sell stocks based on their predefined logic. These decision-making events are sporadic and influenced by market data, portfolio state, and random triggers.

```rust
pub fn start(&mut self, stop_signal: Arc<AtomicBool>) {
    while !stop_signal.load(order: Ordering::Relaxed) {
        // Perform trading actions
        let actions: Vec<(Action, String)> = self.select_random_stocks();

        for (action: Action, stock_name: String) in actions {
            if stop_signal.load(order: Ordering::Relaxed) {
                break; // Exit if stop signal is set
            }

            match action {
                Action::Buy => {
                    if let Err(e: TraderError) = self.place_buy_order(&stock_name) {
                        println!("❌ Trader {} failed to buy {}: {}", self.id, stock_name, e);
                    }
                }
                Action::Sell => {
                    if let Err(e: TraderError) = self.place_sell_order(&stock_name) {
                        println!("❌ Trader {} failed to sell {}: {}", self.id, stock_name, e);
                    }
                }
            }
        }
```

Figure 13: Trader decision events

Traders go periodically through the list of the shared stocks and their portfolios. They decide on buying/selling stocks based on his pre-defined logic. These kind of decision-making events will be sporadic and depending on market data, portfolio state and random triggers.

4.0 Performance Analysis

4.1 Concurrency

The benchmark results below have highlighted performance characteristics related to two main types of processes: starting multiple traders (the student's subsystem) and updating a shared list of stocks (the partner's subsystem). Benchmarking those operations is representative for general efficiency and reliability concerns regarding these processes.

```
multiple_traders_start  time:   [5.0034 s 5.0035 s 5.0037 s]
                        change: [-0.0114% -0.0056% +0.0005%] (p = 0.09 > 0.05)
                        No change in performance detected.
Found 2 outliers among 15 measurements (13.33%)
  1 (6.67%) low mild
  1 (6.67%) high mild

stock_update            time:   [666.30 ns 713.85 ns 758.54 ns]
                        change: [+3.2189% +11.557% +23.177%] (p = 0.02 < 0.05)
                        Performance has regressed.
Found 2 outliers among 15 measurements (13.33%)
  1 (6.67%) high mild
  1 (6.67%) high severe
```

Figure 14: Criterion result in CLI

The "multiple_traders_start" benchmark has pretty consistent performance and executes, on average, in about 5.0035 seconds. Changes are negligible, ranging from -0.0114% to -0.0056% and +0.0005%, respectively, which the p-value supports at $p = 0.09 > 0.05$, implying no statistical significance of these changes. It can, therefore, be said that this process of initializing and running several traders at once is quite stable and well-optimized. Nevertheless, execution time appears quite high, meaning that some computationally heavy operations may be present in this workload, like maintaining RabbitMQ communications, initializing a portfolio, or shared resources management. Even though there were two outliers, one of them was rather mild, and the other one slightly high, neither of them influenced the general performance, which means that the system copes with running several tasks concurrently in an effective way.

In contrast, the "stock_update" benchmark measures the efficiency of updating the list of shared stock and is suffering from performance regression: it has a mean execution time of 713.85 nanoseconds and now represents a performance change of +11.557% compared with the baseline. This regression is statistically significant, given that $p = 0.02 < 0.05$. There are

two outliers, one mild and high, and one severe. Inefficiencies in either the logic to update shared resources or contention for the mutex that protects the stock list may thus hint at a few possible sources of inefficiency. Still, it remains reasonably quick to do this operation. The presence of this regression does suggest something that one might investigate further to ensure performance stays consistent. Comparing the two benchmarks, it's clear that "multiple_traders_start" is a much heavier operation; it runs on a larger timescale but is very stable and reliable. The lightweight process of "stock_update", on the other hand, has shown to bring forth inefficiencies in its handling of shared resources, as was shown by the regression and outliers. This incongruity would suggest that the system handles high-level concurrency well but may struggle with fine-grained synchronization and shared resource access.

In the light of the above findings, profiling of a "multiple_traders_start" process seems to be justified, looking at the parts where less execution time is spent-sharpening the edges relating to RabbitMQ communication, for example, or speeding up trader initialization. Then again, optimization regarding synchronization on the shared list on the "stock_update" should aim further at reducing the scope, perhaps, of the mutex or seeking better concurrency control methods, such as "RwLock" (Nadiger, 2023).

4.2 Scheduling and Communication

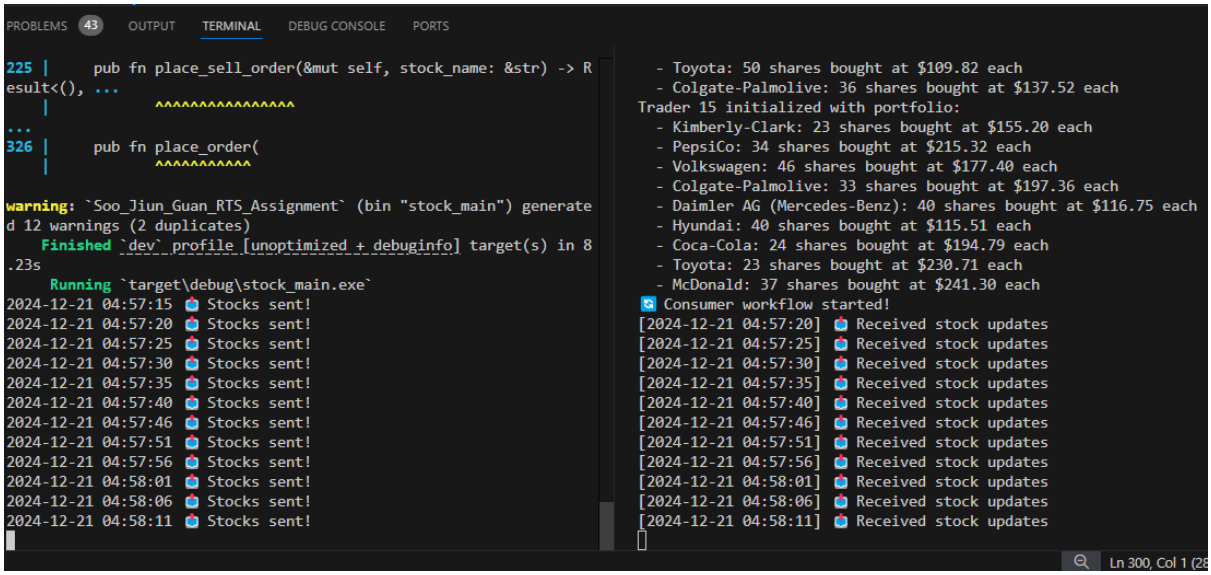


Figure 15: Timestamp for stock updates between publisher (stock subsystem) and consumer (trading subsystem)

Above show the system is a robust and synchronized system wherein the producer publishes stock updates with regularity every 5 seconds, as can be told by the exact timestamps. These

are then processed immediately by the consumer, thus real-time synchronization and responsiveness are guaranteed. After the first stock update, traders get initialized efficiently to operate using the most recent market data, keeping them in sync with the periodic schedule of the producer. Equally impressing is the communication performance, whereby seamless data transfer is reflected between the producer and consumer through RabbitMQ queues. No delay in messaging and no loss of messages reflects a very effective messaging system to ensure that all stock updates are received reliably and processed in real time to support accurate and timely trading decisions. A powerful combination of robust scheduling and efficient communication thus reflects a well-integrated high-performance simulation system.

4.3 Error and Event Handling

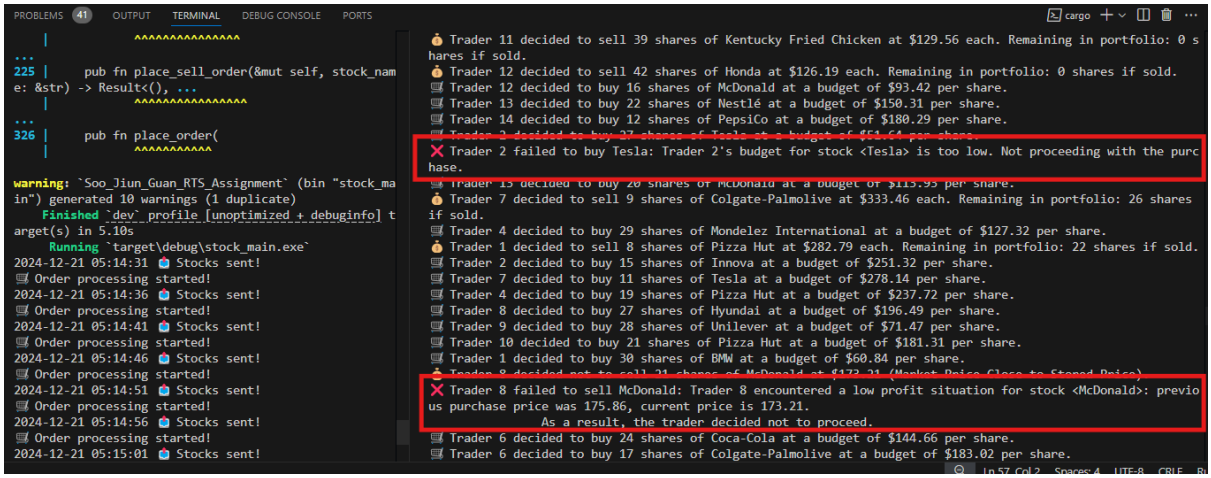


Figure 16: Handling scenarios of insufficient budget and low profit from sales

From above, it can be seen that the system provides an efficient way of error handling in terms of identifying and logging various problems, such as insufficient budget or low profit margin, for realistic and stable trading behaviour. As a matter of fact, error messages are informative, therefore very helpful for debugging and analysis, while the integrity of the trader operations and portfolio will be preserved. In addition, it will prevent the order with an extremely low budget from getting stuck inside the order book by confirming the budget constraints before order placement. It also prevents situations that could have users selling stocks when the profit concern is low by incorporating checks that make sure there is a minimum profit margin before going ahead to sell some stocks. In this way, only realistic and feasible transactions are logged in, maintaining efficiency in real-like trading logic.

Conclusion

This work presents the usability study of Rust for designing and implementing a real-time trading simulation system. Its Ownership Model and Borrow Checker from Rust have been applied throughout the coding to ensure thread-safe code and memory handling. Also, the periodic and aperiodic scheduling integrated smoothly to perform the tasks of a dynamic trading event and market update, respectively. RabbitMQ ensured separation of components in a proper way for communications.

The system successfully coped with the key challenges: managing concurrent tasks of traders, synchronizing shared resources, and preventing errors. Benchmark analysis showed stability for several traders working concurrently and pointed out some lines for further optimization. Descriptive error handling ensured robust and realistic trading behavior, enhancing reliability and usability of the system.

In all, this work shows the promising capability of Rust in constructing real-time systems that have high safety and performance. By mitigating the existing challenges, such as propagating unsafe code and inefficient synchronization, Rust can position itself as a stronger candidate for real-time applications in critical domains. This work lays the foundation for future research and development, particularly in scaling real-time systems and improving developer ergonomics in Rust.

References

Astrauskas, V., Matheja, C., Poli, F., Müller, P., & Summers, A. J. (2020). How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 1–27. https://doi.org/10.1145/3428204

Bugden, W., & Alahmar, A. (2022). Rust: the programming language for safety and performance. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2206.05503

Devi, U. C. (2006). *Soft real-time scheduling on multiprocessors*. The University of North Carolina at Chapel Hill.

Evans, A. N., Campbell, B., & Soffa, M. L. (2020). Is rust used safely by software developers? *arXiv (Cornell University)*, 246–257. https://doi.org/10.1145/3377811.3380413

Jung, R., Jourdan, J., Krebbers, R., & Dreyer, D. (2017). RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, *2*(POPL), 1–34. https://doi.org/10.1145/3158154

Lagaillardie, N., Neykova, R., & Yoshida, N. (2022). Stay Safe under Panic: Affine Rust Programming with Multiparty Session Types. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2204.13464

Levy, A., Andersen, M. P., Campbell, B., Culler, D., Dutta, P., Ghena, B., Levis, P., & Pannuto, P. (2015). Ownership is theft: experiences building an embedded OS in Rust. *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, 21–26. https://doi.org/10.1145/2818302.2818306

Nadiger, A. (2023, June 25). *Mutex, RWLock, Poison Error*. https://www.linkedin.com/pulse/mutex-rwlock-poison-error-amit-nadiger/

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 41st ACM*

SIGPLAN Conference on Programming Language Design and Implementation, 763–779. https://doi.org/10.1145/3385412.3386036

Ravindran, B., Jensen, E., & Li, N. P. (2005). On Recent Advances in Time/Utility Function Real-Time Scheduling and Resource Management. *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, *4*, 55–60. https://doi.org/10.1109/isorc.2005.39

Tien, J. M. (2017). Internet of Things, Real-Time decision making, and artificial intelligence. *Annals of Data Science*, *4*(2), 149–178. https://doi.org/10.1007/s40745-017-0112-5

Zhu, S., Zhang, Z., Qin, B., Xiong, A., & Song, L. (2022). Learning and programming challenges of Rust. *Proceedings of the 44th International Conference on Software Engineering*, 1269–1281. https://doi.org/10.1145/3510003.3510164