**Gellért Peresztegi-Nagy**

# High-availability matching engine of a stock exchange

Computer Science Tripos – Part II

Churchill College

May 14, 2021

# Declaration

I, Gellért Peresztegi-Nagy of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Gellért Peresztegi-Nagy of Churchill College, am content for my dissertation to be made available to the students and staff of the University.

Signed: Gellért Peresztegi-Nagy

Date: May 14, 2021

# Acknowledgements

Writing this dissertation required not only hard work and dedication but also help and encouragement from many individuals.

First of all, I would like to express my gratitude to Prof. John Crowcroft for supervising the project and pointing me to all the right resources. His encouragement and academic excellence have helped greatly whilst writing my dissertation.

I am also grateful to Dr. John Fawcett for providing feedback on drafts of this dissertation and for helping me grow both academically and personally throughout my years at University.

Lastly, I would like to thank my partner, friends and family for reading drafts of this dissertation and for supporting my studies in Cambridge.

# Proforma

| | |
|---|---|
| Candidate Number: | **2383D** |
| Project Title: | **High-availability matching engine of a stock exchange** |
| Examination: | **Computer Science Tripos – May 2021** |
| Word Count: | **11042**[1] |
| Lines of Code: | **4971**[2] |
| Project Originator: | The Author |
| Supervisor: | Prof. Jon Crowcroft |

## Original Aims of the Project

The original aim was to build a highly available matching engine with a simplified interface. The engine has to accept buy, sell and cancel messages that modify the order book at the ask- or bid-price specified in the messages. The matching engine has to reply with acknowledgement messages after the orders are placed. When the engine executes trades, it sends trade confirmation notifications to all parties involved in the trade. The replicated matching engine has to tolerate machine and network failures, including partitions in the network. The plan was to implement this using state-machine replication with the Raft protocol.

## Work Completed

I have achieved all the initial requirements. My replicated matching engine is implemented in Rust using the Raft consensus protocol for state machine replication. It can tolerate $f$ machine failures when run on a cluster of $2f + 1$ machines. It remains consistent and available in the face of partitions, as long as a cluster of $f + 1$ machines can communicate. Its performance is comparable to other replicated state machines implemented using Raft.

## Special Difficulties

None

[2]This word count was computed using TeXcount.

[2]This was computed using loc, and it excludes comments and Jupyter notebooks used for evaluation.

# Contents

# List of Figures

# Chapter 1

# Introduction

This dissertation describes the implementation of a highly available matching engine with a simplified interface. It handles buy, sell and cancel orders and stores outstanding orders in a replicated order book. It acknowledges to clients when their orders have been placed and notifies them of trades occurring through the order book. The replicated matching engine tolerates machine and network failures, including partitions in the network. My implementation of the Raft consensus protocol is used to replicate the matching engine using state-machine replication.

## 1.1 Exchanges

A stock exchange is a venue where people or institutions can buy or sell ownership in public companies. It is the cornerstone of capitalism, allowing profitable companies to acquire more capital for growth, pulling them away from companies with less potential [1].

When the first stock exchanges started their operation around the 17th century, they worked much like marketplaces that people physically attended to exchange money for ownership in a company or vice versa. In the 19th century, however, trading speed increased dramatically with the advent of electronic trading. The National Association of Securities Dealers Automated Quotations, or Nasdaq, as it is called nowadays, began the operation of the world's first electronic exchange on February 8, 1971 [2].

Electronic stock exchanges allow parties to trade the listed stocks at an increasingly fast pace. To support this, they need to be able to handle persistently high loads while having to provide strong consistency, fairness and auditability guarantees. While the market is open, they are expected not to show any sign of failure and allow continuous trading.

The matching engine is the core component of an exchange, responsible for deciding which parties can trade based on those expressing their interest in buying or selling the stock of a company through orders.

As the matching engine is an essential part of the exchange, guaranteeing the failure tolerance of this component helps improve the availability of the stock exchange.

## 1.2 Motivation

It is still common in the financial world to have humans reconfigure systems when failures occur [3]. This has obvious downsides, such as the unavailability of the exchange during the time it takes to notice and initiate failover.

Historically, there have been multiple occasions when this resulted in exchanges becoming unavailable for a considerable time, a recent example of which was in 2015 when the New York Stock Exchange became unavailable for 4 hours due to what was communicated as a configuration error [3]. The New York Times reported that the process was delayed by the 45 minutes it took to reboot a subsystem of the exchange. This resulted in institutions and people losing money by not being able to trade shares solely listed on the NYSE at the appropriate time and – in the long term – the NYSE losing liquidity by a worsened reputation for being a reliable exchange.

This project was motivated by my desire to explore the different tools modern software engineering provides for increasing the reliability of exchanges at the software level and to create a baseline implementation that supports automated failover without the need for human intervention.

Providing such a baseline would be beneficial, not just to stock exchange operators but anyone maintaining a matching engine that needs to tolerate hardware and software failures. Matching engines are operated at almost any larger financial institution that does trading, including market makers, brokerages who provide services similar to exchanges, and trading companies who use matching engines as clearinghouses to eliminate self-trades.

Therefore, solving the failover problem for matching engines would be a contribution valuable to both companies operating a matching engine and their customers.

## 1.3 Related work

In this section, I discuss the relationship of my approach to other work on replication and matching engines.

There are several open-source matching engines available online, however, none of them is failure tolerant. My matching engine was built with failure tolerance in mind from the start, which ensures that it can tolerate both machine and networking failures without limiting the range of operations the engine can perform.

State machine replication has been used repeatedly in the past to provide resilience for large scale systems. Lamport's early work on the Paxos protocol [4] is widely used, however, Raft [5] is more understandable, and therefore gained more traction in the industry recently. Several protocols have been developed to iteratively improve on earlier versions, such as Multi-Paxos [6], which extends Paxos to agree on a sequence of values, Flexible Paxos [7] that relaxes the quorum requirements, Vertical Paxos [8] which shows how to implement configuration changes correctly, or Fast Paxos [9] that reduces the number of message rounds needed for agreeing on a value.

## 1.4 Structure of the dissertation

In the following sections, I outline the work I have done throughout the project and present my contribution to ensuring the failure tolerance of matching engines. In chapter 2, I describe the learning steps that went into preparing for the project and the choices I have made on designing the solution of a failure tolerant matching engine. In chapter 3, I describe the implementation of the project to guide people who would want to build on my solution or learn from the difficulties I have discovered along the way. In chapter 4, I objectively evaluate the implemented solution to demonstrate that I have solved the intended problem and to compare it to related solutions. Finally, in chapter 5, I conclude the project by reflecting on the completed work and highlight possibilities for valuable future work to invite further research in the area.

# Chapter 2

# Preparation

This chapter describes the preparatory work that was undertaken before the implementation started. This included research of distributed algorithms used for state-machine replication, research of matching algorithms used by exchanges, and the preparation of an implementation plan based on the business requirements of the project.

## 2.1 Starting point

The preparation for the project started in the Easter term of Part 1B of the Tripos. I have researched consensus and matching algorithms, distributed systems frameworks and tools to ensure that the implementation phase would go without interruptions.

In terms of background knowledge, I have taken the 1B course on Concurrent and Distributed Systems and the Part II unit of assessment Multicore Semantics and Programming which covered areas related to my project.

I have not used Rust before the start of the project, although the 1B course Concepts in Programming Languages prepared me well to grasp Rust's advanced typing system and ownership model quickly.

## 2.2 Requirements analysis

Electronic stock exchanges are highly regulated and therefore they have many functional requirements that may be non-function requirements in other spaces. For instance, the fairness of the market is required by the Securities Exchange Act of 1934 [10]:

> "SEC. 11A. [78k-1] (a)(1)(C): It is in the public interest and appropriate for the protection of investors and the maintenance of fair and orderly markets to assure [...] (ii) fair competition among brokers and dealers, among exchange markets, and between exchange markets and markets other than exchange markets; [...]"

### 2.2.1 Functional requirements

- **Consistency**: the matching engine should provide a consistent view of the state it stores, nodes should never communicate inconsistent results to clients.

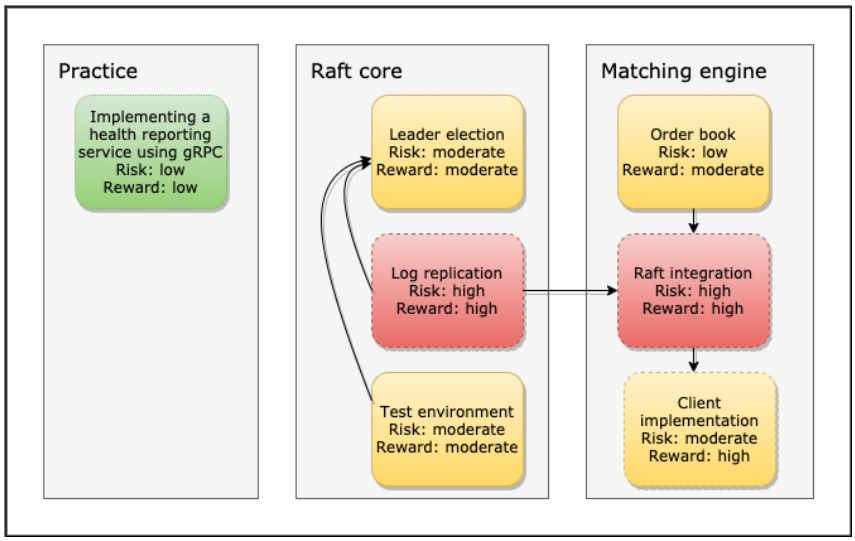



Figure 2.1: An activity diagram of tasks indicating their dependencies, the risk of complications arising during development and their relative reward to the project.

- **Failure-tolerance**: the matching engine should respond to client requests even in the face of failures of a subset of its components.
- **Fairness**: the matching engine should actively ensure that each client has an equal opportunity to trade on the market.
- **Correctness**: stock exchanges are highly regulated and any automated application taking part in trading has to adhere to certain standards. Being able to formally prove correct the application makes the standardisation process easier.
- **Durability**: the matching engine should ensure that orders and trades that are acknowledged to clients are persistently stored for auditability purposes.

### 2.2.2 Quality requirements

- **Availability**: while the CAP theorem states that availability and consistency cannot both be achieved if nodes have to tolerate partitions, the matching engine should strive to make unavailability highly unlikely.
- **Performance**: different exchanges have largely different performance requirements, however, a high-volume market such as the NYSE Amex Options market needs to support an average throughput of over $320,000$ orders per second and a 400 microseconds mean acknowledgement time. These orders are coming from around 150 clients (brokers, market-makers) that are members of the exchange [11].

### 2.2.3 Tasks

Figure 2.1 shows the components of the project with their risks and rewards. The project consists of multiple high-risk, interdependent blocks, therefore, to ensure the success of

the project, I aimed to complete blocking milestones early, and replan when complications arose.

## 2.3 Background research

### 2.3.1 Distributed algorithms

#### State-machine replication

Based on the requirements analysis, I decided to use state-machine replication to ensure the failure-tolerance of the matching engine, which maintains the consistency and fairness guarantees of the system. State-machine replication uses a consensus algorithm to decide the next operation to apply to the state-machine. The consensus algorithm ensures that all of the replicas agree on the order in which they apply the operations, thus the state-machine replicas go through the same states.

State-machine replication has several benefits that made it an appropriate choice for the project. It provides failure-tolerance and consistency and requires minimal adaptation of the business logic, in this case, the matching engine state-machine. It also aligns with the performance requirements of the matching engine, as there are several examples of key-value stores – with comparable business logic to matching engines – performing at the required scale of hundreds of thousands of write requests per second.[1]

Alternatively, I could have used a distributed computing toolkit such as Derecho [12] or the Vsync (earlier ISIS2) library [13] to replicate the matching engine. They provide a simplified interface for implementing replicated state-machines and solve the problem of reconfiguration on failure, atomic multicast, gossip for optimised message delivery. This allows their users to concentrate on the business logic, the core state-machine. On the other hand, they are primarily built for cloud-like environments and make assumptions such as the availability of RDMA[2] hardware to achieve optimal performance. Such a distributed computing toolkit is beneficial for building complex reliable systems, however, for this project, I decided to use standard distributed algorithms to reduce the assumptions I require of the underlying system.

#### Raft

I have researched several consensus algorithms that have been used for state-machine replication. Multi-Paxos [6], Vertical Paxos [8], Flexible Paxos [7] and Raft [5] are algorithms commonly used in industry, and they all optimise for different aspects of state-machine replication.

I have decided to use Raft because of its focus on understandability and simplicity making its implementation appropriate for a Part II project. Furthermore, its paper provides a complete solution to state-machine replication, in the sense that it not only describes how distributed agreement is made on a single value but considers how dis-

[1] `https://etcd.io/docs/v3.4/op-guide/performance/`

[2] Remote direct memory access

| | |
|---|---|
| Order | intent to buy or sell a specific quantity of an instrument |
| Cancel order | intent to cancel a previously placed order |
| Market order | an order to be executed at the best available price |
| Limit order | an order to be executed at a specified price or better |

Table 2.1: Terminology.

tributed agreement interacts with a replicated state-machine requiring the replication of a sequence of state-machine-commands.

The Raft paper describes the core consensus algorithm, but many of the optimisations described in other consensus algorithms apply to it. For example, the optimisation idea of Flexible Paxos [7] can be used to relax the requirement of majority quorums in the leader election and log replication phases to only requiring overlapping quorums between the two phases as shown in [14].

More details on how Raft enables state-machine replication are given in the implementation chapter.

### 2.3.2 Matching algorithms

The field of economics describes security exchanges as double auctions between the instruments traded and "money", the currency of the exchange [15]. A double auction is when two auctions are taking place simultaneously. In the case of a security exchange, on the one side are the buyers, who participate in an auction for the instrument in exchange for money, while on the other side are sellers, who bid for money in exchange for the instrument they hold. Trades occur when there is a price match between the buyer and seller sides.

To represent the current state of the market, a limit order book data structure is used to hold the outstanding orders on the market. A visualisation of the order book is shown on Figure 2.2.

Limit orders are different from market orders (see definitions in Table 2.1), as they are valid until they are cancelled if they cannot be traded immediately with orders already on the market. This is in contrast with market orders, which are cancelled right away if they cannot be filled with trades at the time they are placed. My implementation only considers limit orders, however, implementing market orders would be a simple extension.

Matching algorithms define how the "price match" between buy and sell orders is prioritised if multiple matches are possible where the seller would sell for a lower price than the buyer would buy for.

Many different matching algorithms have been devised since electronic exchanges were first introduced in 1971, and which one of them is used influences how orders get filled, thus influencing how traders react.

The price-time and pro-rata matching algorithms are used most commonly, and they also form the basis of more elaborate matching algorithms [16]. Both of them prioritise orders primarily by price, but they differ in how they fill orders at the same price-level:

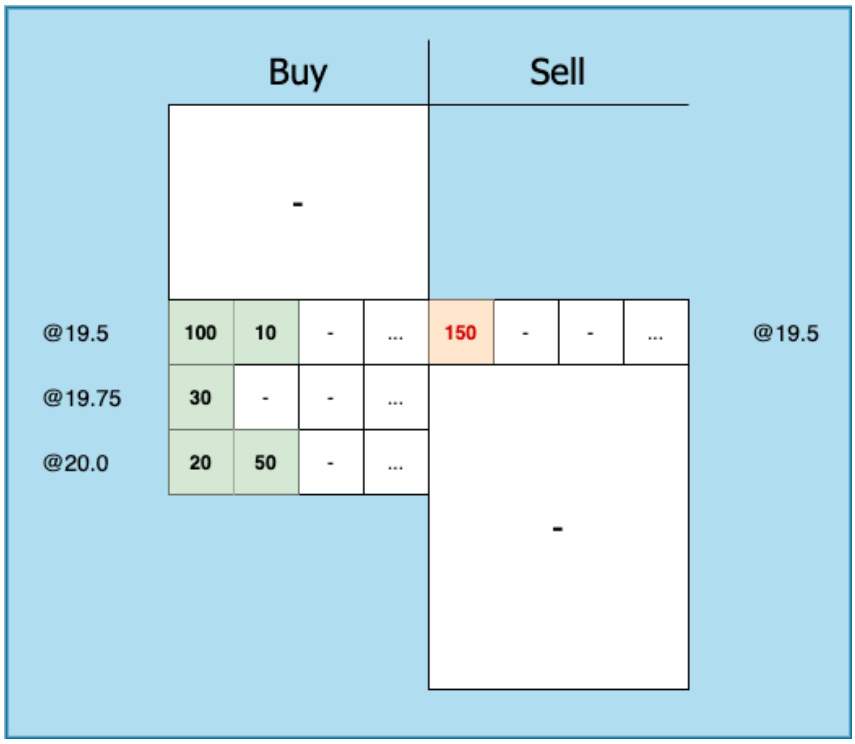



Figure 2.2: A visualisation of an order book with a new tradeable order arriving at a price-point of 19.5 of quantity 150.

- **Price-time or FIFO allocation**: at the best price, an incoming trade is matched with the order that first arrived on the market.

  For example, consider the scenario depicted on Figure 2.2. When the order to sell 150@19.5 arrives on the market, it is first matched with the orders at the best buy-side price (bid) on the market, which are the orders 20@20.0 and 50@20.0. These orders are filled fully, and the matching algorithm has left to allocate the remaining $150 - 20 - 50 = 80$ units of the incoming order. Next, the 30@19.75 is filled, leaving the remaining quantity at 50. Then, the price-time algorithm matches these 50 units with the order that first arrived on the market at the next best price-point, which we may assume was 100@19.5 in this case. Therefore, the incoming order is fully filled with the order 100@19.5, leaving 50@19.5 and the original 10@19.5 on the buy-side.

  More formally, for each incoming order $I$, let $L_1, L_2, \ldots, L_n$ be the limit orders on the market ordered by the time they were added that could be matched with $I$ – these are the orders on the other side of the auction with a better price than $I$, i.e. lower price if $I$ is a buy order, higher price if $I$ is a sell order.

  Then, arrange $L_i$ primarily by price and secondarily by the time they were added, that is, the order with the best price that was added the earliest is at the head of the list.

  The orders are filled in this sorted order sequentially. $L_i$ is filled fully, if it can be, that is, if $quantity(L_i) \geq remaining\_quantity(I)$, otherwise, a new order with $quantity(L_i) - remaining\_quantity(I)$ replaces $L_i$. If there are no more orders that

$I$ could be matched with, $remaining\_quantity(I) \geq 0$, and $I$ is a limit order, then a new order with quantity $remaining\_quantity(I)$ is placed on the market.

Price-time is the most common matching algorithm used, for instance, for the highly-traded E-mini S&P500 future of CME Group.

- **Pro-rata allocation**: at the best price, an incoming trade is matched with all the orders at that price-point, proportionally to their volumes.

  For example, considering the earlier scenario depicted on Figure 2.2. When the order 150@19.5 arrives on the market, it is first matched with the best buy-side price (bid) on the market, similarly filling 20@20.0, 50@20.0 and 30@19.75 completely, leaving $150-20-50-30 = 50$ units of the remaining quantity. Then, the pro-rata algorithm matches these 50 units with all of the orders at that price-point proportionally. The order 100@19.5 is matched with $\lfloor \frac{100}{100+10} \rfloor = 45$ units, while the order 10@19.5 is matched with $\lfloor \frac{10}{100+10} \rfloor = 4$ units. As there is 1 more unit unmatched resulting from the remainders of integer division, this may be allocated using a seconding algorithm, for example, a price-time algorithm matching the remaining 1 unit to the earlier arriving 100@19.5. Therefore, in the last round, 100@19.5 is assigned 46 units leaving 54@19.5 on the market, while 10@19.5 is assigned 4 units leaving 6@19.5 on the market.

  More formally, when an incoming order $I$ arrives with quantity $N = quantity(I)$, all of the limit orders $L_1, L_2, \ldots, L_n$ at the best price are partially filled with proportion $p_i = \frac{quantity(L_i)}{\sum_{j=1}^{n} quantity(L_j)}$. That is, order $L_i$ is filled at quantity $\lfloor p_i \cdot N \rfloor$. As $\sum_i^n \lfloor p_i \cdot N \rfloor \leq \sum_i^n quantity(L_i)$, there may be some remaining quantity of $I$, which is distributed by some secondary algorithm such as FIFO.

  Pro-rata is used, for instance, in the NYSE American Options market.

The matching algorithm influences when and how much of the limit orders get filled, therefore, short-term traders might adapt their trade placing strategy to the matching algorithms used on the market.

The exchange operator has to consider these effects and decide on the ideal matching algorithm for the traded instrument.

Table 2.2 summarises the advantages and disadvantages of the FIFO and pro-rata algorithms.

## 2.4 Tools and dependencies

### 2.4.1 Rust

After I gained a thorough understanding of what I am going to implement, I started thinking about what tools to use. In terms of the programming language, I was considering using either Rust or C++, as they are both flexible enough to support low-level optimisations of concurrent algorithms. They are not garbage collected, thus their performance

| Allocation | Advantages | Disadvantages |
|---|---|---|
| Price-time | Encourages a narrow spread and thus high liquidity, as limit orders close to bid-ask threshold have high price-priority. | Encourages frequent, thus small orders, which increases the computational overhead of the matching algorithm. |
| Pro-rata | Encourages large orders. | Does not motivate a narrow spread, as priority is given to large orders instead of early orders close to the bid-ask threshold. |

Table 2.2: Advantages and disadvantages of the price-time and pro-rata matching algorithms.

is more predictable, and they have a wide selection of libraries that support coding and are both in widespread use in the industry, further justifying their relevance.

I decided that Rust would be the best fit for my project because the strong memory-safety and thread-safety guarantee the compiler provides both aids development and gives confidence in the correctness of the matching engine implementation, a safety-critical system.

Rust ensures memory safety using its ownership model that requires that there can only be either a single mutable or multiple immutable references to a memory location at any given time.

Thread-safety is ensured using the Send and Sync traits that types must implement if they are safe to send to another thread or safe to share between threads, respectively. The precise definition of their relationship is: a type T is Sync if and only if &T is Send.[3] For instance, the Mutex struct is both Send and Sync; the multi-producer single-consumer channel, mpsc::Sender, is Send, but !Sync; and a non-atomic reference-counted reference Rc is both !Send and !Sync.

Furthermore, Rust has an extensive toolchain including a package manager, linting tools and many well-supported packages including the widely used Tokio async IO library.

### 2.4.2 Tokio

Tokio is an asyncIO runtime that was useful to my project for two main reasons.

First, it provides lightweight user-level threads that makes parallel communication between Raft nodes efficient. Rust only has kernel threads by default, and they have a higher latency for creation and context switches between them. As my implementation uses many threads, for example, to send logs to multiple Raft nodes concurrently, it benefits from the better performance of user-level threads.

Second, consensus protocols rely on information exchange between machines that is typically implemented using remote procedure calls, which provide a clean abstraction over network communication. Tokio has an asynchronous gRPC implementation in the Tonic library that I used. I chose gRPC as my remote procedure call framework, as it has an

[3] `https://doc.rust-lang.org/std/marker/trait.Sync.html`

| **Libraries** | |
|---|---|
| tokio | asyncIO framework |
| tonic | gRPC library |
| tracing | logging library |
| criterion | benchmarking library |
| **Tools** | |
| CLion | IDE with autocompletion for Rust |
| clippy | linting and formatting tool |
| GitHub Actions | CI hook for running tests + linting checks |
| Jupyter | Python notebook environment used for evaluation data processing |
| matplotlib | plotting library for evaluation |
| LaTeX | document preparation system |
| BibTeX | bibliography management |
| draw.io | diagram creation tool |

Table 2.3: Main dependencies and tools used for the project with their justification.

efficient binary serialization protocol (protobuf), its procedure calls can be defined easily and flexibly. Moreover, as gRPC is supported by many languages, it could interoperate well with other components of a stock exchange in a commercial context.

### 2.4.3 Toolchain

Table 2.3 gives a breakdown of the libraries and tools I have used during my project with justification for the use of each.

## 2.5 Software Engineering

### 2.5.1 Engineering methodology

Initially, I followed a waterfall development model for implementing the Raft algorithm as per the specification in the Raft paper. After I had a well-tested, complete implementation of the Raft module, I moved over to a spiral development model to iteratively improve the matching engine and to work on optimisation extensions.

I used the detailed timeline and milestones from the project proposal to track whether I am on track with the project. When I was behind, I tried to focus only on the necessary deliverables and tried to complete important milestones early. On the other hand, when I was on track, I allowed myself to explore extension and optimisation options.

### 2.5.2 Testing and documentation

I used both unit and integration tests to detect potential bugs early, and regression tests for later-discovered bugs. I set up a continuous integration system using Github Actions to run the unit and regression tests regularly, on pull requests to the ‘master’ branch. I also

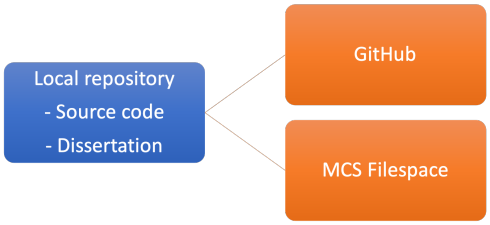



Figure 2.3: The backup policy of the source code and the dissertation files

used experimental tests to explore the libraries I was planning to use. As the integration tests required a controlled environment (at least 3 threads with rare OS context switches), and because I was working on the project on my own, I only ran the integration tests locally.

I used Rust's built-in documentation creation system called `rustdoc` which could generate HTML documentation from documentation-comments prefixed with `///` in the code.

### 2.5.3 Version control and backup strategy

The source code and my dissertation were version controlled with Git and were regularly pushed to both a GitHub repository and my MCS filespace for redundancy.

### 2.5.4 Development environment

I was using my personal machine for development, evaluation and testing. I also used the traffic control[4] utility of Linux to simulate an unreliable environment and introduce network failures for testing and evaluation.

[4] `https://man7.org/linux/man-pages/man8/tc.8.html`

# Chapter 3

# Implementation

This chapter describes the implementation of the replicated matching engine. It starts with a high-level overview of the system, giving an example of how an order is placed on the stock market. Next, it describes the main components and protocols in more detail, including optimisations implemented. Finally, it highlights complex, surprising or otherwise interesting implementation details.

## 3.1 System design

### 3.1.1 Timeline of an order

The system is composed of three main components, a client port that is handling incoming client requests, a matching engine that contains the business logic to find trades between the incoming orders and the Raft module that is providing a consistent log to facilitate the replication of the matching engine component. The interaction of these components is depicted in Figure 3.1.

When a client of the stock exchange (mostly brokerage firms and market-making companies) wants to place an order on the market, it uses the FIX protocol to communicate with the exchange. Both the client and the exchange run an instance of the FIX engine that is used to send and receive orders and order acknowledgements amongst other things.

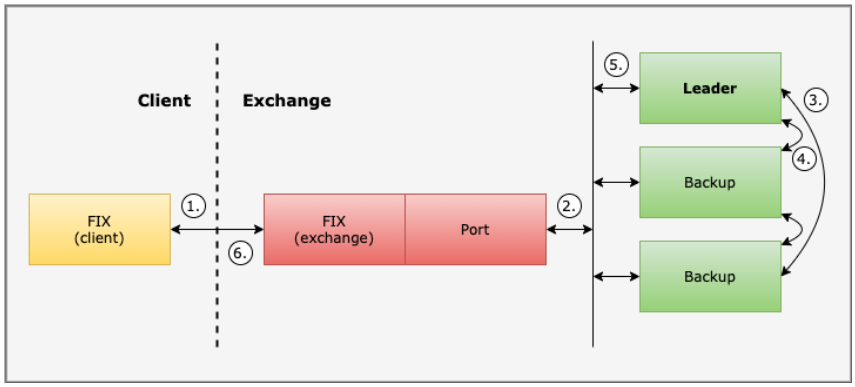



Figure 3.1: Timeline of an order going through the exchange

When the exchange receives an order from a client, its FIX engine passes this request on to a client port that is responsible for handling this request inside the exchange. This port then broadcasts an order request in the internal network of the exchange that is connected to all the live replicas of the matching engine.

All of the matching engine replicas can hear this request, however, only the leader acts on it. When the leader hears a request, it initiates a replication process through the Raft module to ensure that all of the replicas maintain a consistent state of the market.

The replication process is done in two stages. First, the Raft module running at the leader sends an AppendEntries request to the other replicas to replicate the order as the next log entry in their Raft module. Then, when at least a majority of replicas have acknowledged the next order to append (1 backup replica apart from the leader in the depicted case with 3 total replicas), the leader sends another AppendEntries request with an incremented commit index that tells all of the replicas that they can process this order. At this point, the leader also processes this order.

When the order is committed, it means that the Raft node delivers it to the local instance of the matching engine. The matching engine then checks if the order could be filled, even if partially, by other orders on the market, and if it cannot be filled fully, it gets stored in an order book that represents the outstanding orders on the market. To communicate the trades generated by applying this order to the matching engine state machine, and to acknowledge the order back to the clients, the leader replica broadcasts trade and order acknowledgement messages on the internal network of the exchange.

These messages are picked up by the interested client ports and are propagated back to the clients using the FIX engines.

There are quite a few subtleties that are omitted for clarity from the previous timeline, however, it gives an overview of the system in the typical, steady-state case. The following sections describe how the subtleties are handled.

### 3.1.2 Component interfaces

To clarify how exactly the components interact, this section defines the interfaces the internal components use to interact with each other.

The interaction of client ports and the matching engine replicas is done through RPC calls. Client ports can send a stream of limit orders to the engine, and they can also register to receive order acknowledgements and trade notification. The definition of this interface is given below, defined using protobuf, the serialization language of our RPC framework:

```
service MatchingEngineRPC {
  rpc limit_order_queue(stream Order) returns (google.protobuf.Empty);
  rpc register_order_acknowledgements (Client) returns
        (stream OrderAcknowledgement);
  rpc register_trades (Client) returns (stream Trade);
}
```

The matching engine and the Raft module interact using local function calls. The matching engine holds a reference to the local Raft node, and it can call its `start()` method to initiate the replication of new commands. To apply a newly committed command, the Raft module uses its shared reference of type `Arc<Mutex<MENode>>` to deliver the command to the core matching engine struct. The replicated matching engine struct has the following structure to satisfy Rust's ownership model:

```
struct MatchingEngine {
    me_node: Arc<Mutex<MENode>>,
    raft_node: RaftNode,
}
```

The Raft nodes use RPC calls to communicate between themselves. They use the VoteRequest calls for leader election and AppendEntries calls for heartbeats and log replication. This is defined in protobuf as follows:

```
service RaftRPC {
  rpc request_vote (VoteRequest) returns (VoteResponse);
  rpc append_entries (AppendEntriesRequest) returns (AppendEntriesResponse);
}
```

## 3.2 Raft

The Raft component provides a consistent log stream abstraction, which is used to maintain a stream of state-machine commands in a consistent order, enabling state-machine replication.

The implementation of the Raft module is based on the original Raft paper [5].

The Raft implementation is best approached by explaining two things. First, what state each Raft replica holds, and second, the concurrent threads that each Raft node runs to interact with other replicas and the state-machine (in this case, the matching engine).

A key, simplifying idea of Raft is to first elect a leader, and then use the leader to coordinate the replication. The leader is the one that responds to client requests and initiates the replication of new commands.

Raft only applies a new command to the state machine when it has been replicated to at least the majority of the replicas, which guarantees that commands are applied to the state machine in the same order.

The key state that Raft needs to keep track of the current state of the replication is shown below:

```
pub struct Raft {
    me: NodeId,
    current_term: u64,
    voted_for: Option<NodeId>,
    commit_index: u64,
    last_applied: Option<usize>,
```

```
    log: Vec<LogEntry>,
    last_log_index: u64,
    last_log_term: u64,
    current_leader: Option<NodeId>,
    target_state: TargetState,
    reelection_timeout: Option<Instant>,
    next_heartbeat: Option<Instant>,
    state_machine: Arc<Mutex<dyn StateMachine + Send + Sync>>,
    // Initialised for leader only
    next_index: HashMap<NodeId, u64>,
    match_index: HashMap<NodeId, u64>,
    ...
}
```

Raft divides time into discrete terms and we have that in each term, each node can only vote for a single node, therefore, only a single node can receive the majority of votes and become a leader. Terms are monotonically increasing, and they are incremented when the replica has not heard a heartbeat message from the current leader for some time longer than the re-election timeout.

Voting is done using VoteRequest remote procedure calls that include the term of the voting, and the id of the replica requesting the vote. The term is used to discard outdated requests, and to ensure that each node only votes for a single candidate in each term. The RPC call also includes the index and term of the last entry in the candidate's log, which ensures that the elected leader is at least as up-to-date as the log of those voting for it, that is, at least a majority of the replicas. This is to ensure that no already-committed log entries get forgotten during re-election, which could break the assumption that state-machine commands are applied in a consistent order.

Elaborating on how each Raft variable is used:

- `current_term`: the term the local replica thinks is the latest. It is unchanged in the steady-state when the leader does not fail, and it is incremented when a re-election is triggered by a timeout.
- `voted_for`: the id of some other node that the local replica voted for in the current term by replying to a VoteRequest RPC, or `None` if it has not cast a vote yet.
- `commit_index`: the index of the last log entry that is known to be committed to at least a majority of the replicas. This is the last log entry that can be applied to the state-machine safely at the time.
- `last_applied`: the index of the last log entry that has been applied to the local state-machine. This index is incremented until it reaches `commit_index`.
- `log`: the array of log entries that the local replica thinks is the most up-to-date. Each `LogEntry` contains the term when it was appended to the log and a command, which is information on what the state-machine has to do when it applies this log

(eg. `Command { type: Buy, price: 135, quantity: 10, ... }` for a buy order in the case of the matching engine).

The log can both grow and shrink, but it can never shrink to before the `commit_index`. The log grows when new client requests arrive.

When the leader initiates the replication of a new command, it adds a new entry to its local `log` and sends AppendEntries RPC calls to the other replicas to instruct them to append the new entry to their local logs. On receiving an AppendEntries RPC request, the replica first checks to see if it is a valid request (by checking that the `term`, `last_log_index`, `last_log_term` and `entries` fields included in the request are consistent with the local state). If the consistency checks pass, the new log entries are appended to the log of the follower.

The exact consistency checks are succinctly described in Figure 2 of the original Raft paper [5].

- `last_log_index` and `last_log_term`: convenience fields that are equivalent to the index and term of the last entry in `log`.
- `next_index`: a state only maintained by leader nodes. This keeps track of the index of the next log entry that needs to be sent to each of the other replicas.
- `match_index`: a state only maintained by leader nodes. This keeps track of the index of the last log entry that is known to have been replicated at each replica.
- `target_state`: the state the local replica thinks it is in. It can be either follower, candidate or leader.
- `reelection_timeout`: a state only maintained by non-leaders. The local replica should try to become a leader if it does not hear from the leader by this time. This is reset each time a replica receives an up-to-date RPC from the leader.
- `next_heartbeat`: a state only maintained by the leader. The time when the leader has to send the next heartbeat to the followers to ensure that they do not reach their `reelection_timeout`.
- `network`: an abstraction for communicating with other replicas through RPC calls.
- `config`: configuration parameters, such as the length of the re-election timeout and heartbeat period, and the number of replicas in the cluster.
- `state_machine`: a reference to the state machine to which the local replica applies its logs.

Raft uses multiple threads to maintain the core Raft state. To synchronise access to this state locally, it uses an `Arc<RwLock<Raft>>` type. This means that each access to the core Raft state requires taking a read-write lock that is shared using an atomically reference-counted (Arc) reference. The main threads Raft uses are the following:

- **RPC thread:** each Raft replica runs a gRPC server that handles RPC requests coming from other Raft nodes.
- **Heartbeat thread:** each Raft replica runs a thread with a busy-waiting loop that waits until the heartbeat timeout is reached to send a new heartbeat to the other replicas. For non-leaders, this timeout is ignored, they are just waiting in this loop, as only leaders have to send heartbeats.

  It is implemented as a busy-waiting loop with a sleeping period of 20µs, which is a sufficient trade-off between the accuracy of the heartbeat timeout and the impact of busy-waiting on the CPU usage. Using a busy-waiting loop instead of an interrupt or condition variable based approach simplified the code, as it allowed me to decide whether to send a heartbeat or not using a declarative approach by checking the `next_heartbeat` and `target_state` fields of Raft. It also simplified implementing request pipelining and batching, which is detailed later.
- **Re-election thread:** each Raft replica runs a busy-waiting loop that waits for the re-election timeout to trigger. The current leader is just waiting in this loop and ignores the re-election timeout until it again becomes a non-leader. When a non-leader replica reaches the re-election timeout, it converts into a candidate and initiates the re-election process.

  It is implemented as a busy-waiting loop for the same reason as the heartbeat loop.
- **Client thread:** There is one last thread that interacts with Raft replicas, which is the state machine's thread. This is a thread external to the core Raft module, but some of Raft's code runs on this thread. When a new client request arrives at the state machine, the `start()` method of Raft is called to initiate the replication of a command. This method is run on the thread of the caller, and it is synchronized with other threads by taking a lock on the core Raft state.

### 3.2.1 Optimisations

#### Batching append entries

The original Raft paper [5] describes the implementation by sending an AppendEntries RPC for each new log entry that arrives from clients. This aids understandability – the overarching goal of the Raft protocol, but it is not strictly necessary, and it limits performance. Pipelining and batching are left as future work in the paper, however, their implementation is relatively short and simple.

We can implement batching by changing the `next_heartbeat` to be at most 1ms after a new command has been received, which gives a 1ms windows to aggregate commands and sends them all in a single batch.

This leads to a performance improvement, as the overhead of sending a request is distributed across many – usually around 100 – client requests. This is especially beneficial when replicating the matching engine because commands are small in that case – around 32 bytes, which reduces the overhead of sending it using HTTP-based gRPC calls.

## 3.3 Matching engine

The core of the matching engine state-machine works as a database for orders with specialised business logic. It maintains an order book for each security traded on the exchange. The order book structure supports multiple operations, including placing a buy or sell order on the market, cancelling orders and checking for possible trades in the book.[1]

The elegance of state-machine replication is that we can assume that the state-machine is performing sequential operations, therefore, it is easy to extend the order book implementation, as it has no impact on the consistency of the replicated matching engine. This is also favourable in a commercial context, where the project would be divided between several teams, and the team implementing the business logic of the matching engine and the team owning the distributed algorithms could be largely independent, as they would have a clear interface dividing them.

Initially, I implemented an order book with a baseline price-time matching algorithm, then, as an extension, I optimised the price-time algorithm by profiling the code and using the Adaptive Radix Tree data structure for indexing orders by price.

To build a versatile matching engine, further work is needed on the exchange to implement other common matching algorithms described in section 2.3.2. To extend the current implementation, any new algorithm could be used that supports the following standardised order book interface (or trait as it is called in Rust):

```
pub trait Book {
    fn apply(&mut self, order: Order);
    fn check_for_trades(&mut self) -> Vec<Trade>;
    fn cancel(&mut self, order_id: OrderId, side: Side) -> bool;
}
```

### 3.3.1 Baseline matching algorithm

The baseline price-time order book I implemented maintains two priority queues of orders for the buy and sell sides of the order book. The orders in the priority queue are sorted primarily by price (in ascending order for bids and descending for asks), and the secondary sorting order is defined by a timestamp of when the order was added to the book – the earlier it was added, the closer to the front of the queue the order is.

Figure 3.2 shows a visual representation of how the baseline order book stores orders. In this specific example, these are tradeable orders in the book, so the next call to `check_for_trades` would match the orders at the 20.0 price-point and generate two trades.

The standard priority queue implementation in Rust is a binary heap that supports $O(\log n)$ insert, pop and delete operations. This leads to an $O(\log n)$ `apply`, $O(\log n)$ `cancel` and $O(T \log n)$ `check_for_trades` where $T$ is the number of trades generated

[1] As described in table 2.1, there is a difference between market orders and limit orders in financial markets. My order book implementation only handles limit orders, which are kept in the order book if they could not be filled. A market order would be checked for trades and then removed from the market right away.

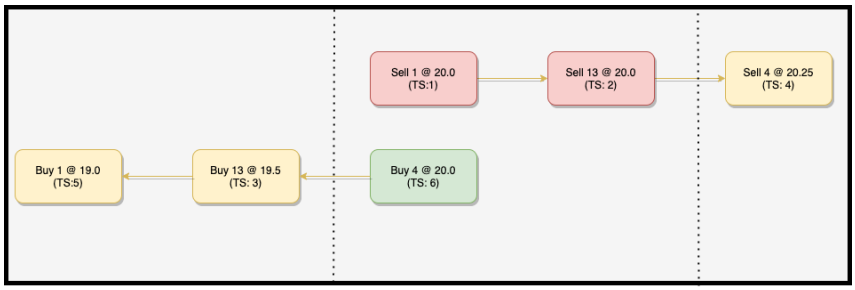



Figure 3.2: Baseline order book implementation using two priority queues and showing the overlapping range of tradable orders.

by the incoming order (usually, we have that $T \ll n$). This performs well as a baseline, however, when $T$ becomes larger – for instance when a large incoming order clears several orders from the book – the matching latency increases to several microseconds.

### 3.3.2 Optimised matching algorithm extensions

In trading, optimising the latency of operations is important, because even microseconds of latency can make the market inefficient and present opportunities for arbitrage between exchanges. For instance, McKay Brothers operates a microwave communication network for financial institutions between exchanges that reduces the propagation delay to 8.5µs from around 11ms to reduce the latency over straight line fiber optic cables [17].

While there are many components of the project whose contribution to the end-to-end latency could be reduced, an attainable and meaningful extension was to reduce the latency of the matching algorithm.

I reduced the latency of the matching algorithm by restructuring the order book to use data structures that support the operations of the book with lower time complexity and better cache efficiency.

I grouped orders by price into buckets, which were implemented as growable circular arrays (VecDeque from Rust's standard library), and indexed buckets using an Adaptive Radix Tree on both the buy and sell sides. The idea to use an Adaptive Radix Tree came from the open source matching engine implementation, exchange-core[2].

The implementation also maintains a hashmap from order id to price to make cancelling orders from the book faster.

#### Adaptive Radix Tree

As there was no Adaptive Radix Tree (ART) implementation in Rust that supported all of the operations the order book needed, I ported a popular and cleanly written C-implementation[3] to Rust. I also adapted the implementation considerably to satisfy Rust's ownership model (which guarantees memory safety for the implementation), while also ensuring that correctness and performance were not sacrificed during this rewrite. As the ART data structure would be beneficial to many others writing code in Rust, I am going

[2] `https://github.com/mzheravin/exchange-core`
[3] `https://github.com/armon/libart`

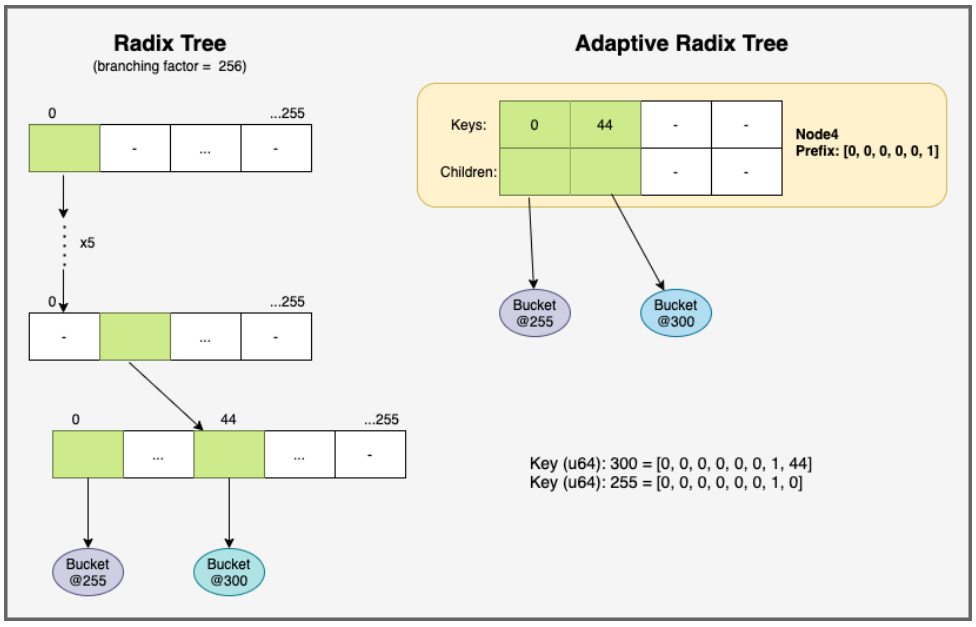



Figure 3.3: Comparison of the memory layout and space consumption of a radix tree and an adaptive radix tree. The adaptive radix tree saves space by compressing prefixes and using adaptively sized nodes.

to open source the implementation on GitHub with a permissive MIT license, and also publish it on Rust's package registry[4] after the project deadline.

A significant property of radix trees is that their height and time complexity depends on the key length (k) rather than the number of keys (n) present in the tree. As key-value pairs are stored in sorted order, radix trees support all of random, minimum and maximum lookups/inserts/deletes with $O(k)$ time complexity, and range scans with $O(kn)$ time complexity. For our scenario of indexing by a price of 8-bytes length, $k = 8$ is a small constant, compared to $n$, the number of price-point buckets with orders, which can be much larger.

However, standard radix trees have a fixed branching factor, which leads to excessive space consumption, which is illustrated on Figure 3.3.

ART solves this problem by adaptively changing the size of internal nodes depending on the number of children it currently has. As the number of children of an internal node increases/decreases, ART will grow/shrink the internal node into one of 4, 16, 48, 256 sizes.

The keys are represented as a sequence of bytes, which means that at each byte-level of the key, there are up to 256 children to go on to, hence the maximal capacity of nodes is 256. Each byte of the key is used to determine the next child of the node, but this is done differently for differently-sized nodes:

- Node4: up to 4 keys and 4 children are stored in this node, with keys stored in sorted order, and the index of corresponding keys and children matching. Looking up the next child is done using a linear search, which, where supported, may be optimised by the compiler into a SIMD instruction.

[4] `https://crates.io`

- Node16: works the same way as a Node4, but holds up to 16 keys and children.
- Node48: has an array of up to 256 positions that map keys to positions in the array of children of size 48. To look up the next child we index into the key array to get a position, and use the position to index into the children array.
- Node256: has an array of 256 children which we can directly index by the key byte to get the corresponding next-child node.

To further save space, key sequences (prefixes) are compressed by storing the prefix in the header of the node if all descendants share that prefix.

### Order book

The optimised implementation has a reduced time complexity for the operations of the order book:

- `apply` – $O(1)$: inserting an order means looking up its price bucket in the ART in constant time, then inserting it to the end of the bucket in amortized constant time. We also have to insert the price of the new order into the order id to price hashmap, which is also done in constant time.
- `check_for_trades` – $O(T)$: checking for trades is done by iterating over the orders on the buy and sell sides simultaneously and merging them into a trade until they are no longer tradeable (the price of the best buy order is lower than the price of the best sell order). Each step of the sorted iteration in ART takes constant time, therefore, the whole operation takes $O(T)$ time where $T$ is the number of trades generated. Usually, we have that $T \ll n$ where $n$ is the number of orders in the book.
- `cancel` – $O(min(i, n - i))$: Using the order id to price-point hashmap, we can look up the price of the order in constant time, then, use the price to look up the price bucket in the ART in constant time. Finally, we can remove the order with a remove operation on the VecDeque data structure, which has $O(min(i, m - i))$ time complexity where $m$ is the number of orders in the bucket and $i$ is the position of the order we are cancelling. Usually, we have that $m \ll n$, where $n$ is the number of orders in the book.

While the time complexity of the operations is not always optimal, they were chosen with cache efficiency in mind. For example, a linked-list could support constant time delete and insert operations, whereas VecDeque only has amortized constant time inserts and $O(min(i, n - i))$ deletes. However, VecDeque stores data as a continuous block on the heap, which has better cache performance for iteration than a linked list with nodes scattered over many places on the heap.

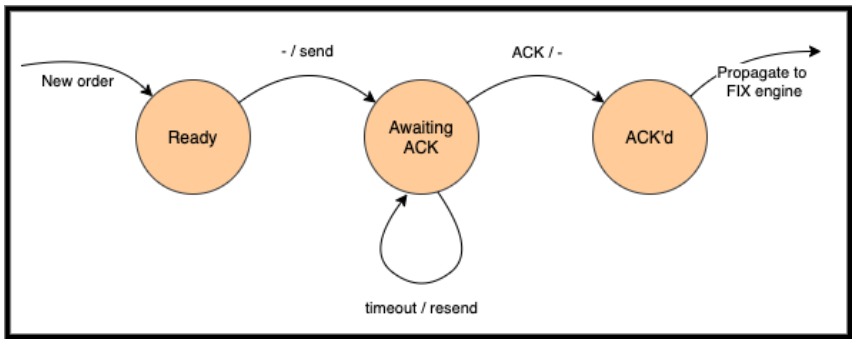



Figure 3.4: State machine of outstanding client orders showing how client orders are retried until an acknowledgement arrives.

## 3.4 RPC calls

As discussed in the preparation chapter, the project is using the gRPC framework for RPC calls. gRPC uses HTTP/2 requests to initiate calls and uses protobuf, a binary protocol, to encode the arguments and return values of the requests.

While the simple RPC calls of gRPC are efficient, gRPC achieves maximal throughput by using streaming calls. As it uses HTTP/2, it can support streaming from both the client and the server.

The original plan was to use unary RPC calls to send orders to the matching engine replicas, however, I observed, by examining the execution traces, that this was limiting throughput. A micro-benchmark of sending 1000, 16-byte-argument RPC calls on the loopback interface showed that the average throughput of streaming was approximately 50 times higher (14925 RPS compared to 289 RPS).

Therefore, I adapted the project to use streaming RPC's to broadcast orders from clients to the matching engine replicas.

## 3.5 Failure tolerance

The Raft protocol can tolerate $f$ node failures when configured to run with $2f + 1$ nodes. However, care has to be taken when integrating the client port and the matching engine components with the Raft protocol to achieve exactly-once semantics of order requests.

This implementation uses sequence numbers and a retry mechanism to ensure exactly-once semantics.

The client port keeps track of the state of all outstanding client orders that can be either ready, awaiting acknowledgement or acknowledged. The transitions between these states are depicted in Figure 3.4.

The retry mechanism ensures at-least-once semantics, while the sequence numbers (with help from the matching engine) ensure at-most-once semantics. The combination of the two leads to exactly-once semantics.

The matching engine has to deduplicate client requests by sequence number, and it does it at two places:

- When a new order arrives at the leader on the internal network of the exchange, it is acknowledged right away (without replication) if the matching engine has already replicated and applied it once.

  This is implemented by maintaining an acknowledged id set, a set of sequence ids of orders for each client that have already been applied. If the sequence id of the incoming order is already in the set corresponding to the order's client, the order is acknowledged right away.

  It is safe to acknowledge these orders early. An order acknowledgement signals that the order has been replicated to a majority of the matching engine replicas, therefore, it is considered to be placed on the market. However, the sequence number of an order can only be present in the acknowledged id set if it has been applied to the matching engine replica after it has been replicated to a majority via Raft. Therefore, a sequence id being in the acknowledge id set for the given client implies that the order has been placed, and as orders cannot be rolled back, it can be acknowledged.

- Furthermore, deduplication has to be implemented in the matching engine replica as well to deduplicate orders that are present in multiple entries of the replicated log.

  Orders can be added to the log multiple times, for instance, when an order is retried by a port before it has been replicated to a majority of the replicas.

  When an order is applied to the matching engine state machine, we first check if its sequence number is present in the acknowledged id set. If it is, the order is simply ignored, because this order has already been processed. Otherwise, the order is processed, and its id is added to the acknowledged ids set.

The above mechanism is used to make the orders idempotent and retrying makes them tolerate the failure of even the entire matching engine replica set. This sacrifices availability for consistency and partition tolerance as per the CAP theorem, which aligns with the requirements of the project as described in the preparation chapter.

## 3.6 Testing

For testing and evaluation of failure-tolerance, I needed a way to simulate node and link failures.

The original plan was to use Mininet [18], a synthetic environment for creating a virtual network of nodes. Mininet allows defining properties of the simulated network including the bandwidth, latency, likelihood of packet loss on a link, and it allows starting and killing processes on the virtual nodes.

The inconvenience with Mininet was that it required me to define the integration tests in Bash or Python, and that made it more complex to synchronize the startup of clients (eg. using barriers) and to assert conditions at runtime or after termination (eg. whether the Raft nodes converge to the same leader).

I would have needed to augment each tested process (eg. the Raft node and the matching engine) and have a separate process to gather the data produced by the tested nodes.

It was easier to start all of the nodes from a single Rust process on separate threads and use Linux's traffic control[5] utility to simulate an unreliable link.

Traffic control helps simulate an unreliable link by adding latency, a bandwidth limit and packet loss to the loopback interface. As all of the nodes are running locally, they all use different ports of the loopback interface to communicate through the "network".

Node failures were harder to simulate because all of the nodes were running in the same process, so stopping nodes meant having to stop threads. This was not supported by the tokio runtime at the time, therefore, I implemented a shutdown mechanism that supported three use-cases: sending a shutdown signal, polling if the shutdown signal has been sent and blocking to await a shutdown signal.

- `fn poll_terminated() -> bool`
- `async fn wait_for_shutdown() -> Future<Item=()>`
- `fn shutdown()`

I implemented the shutdown mechanism using a semaphore. The semaphore is initialised to 0, and it is incremented to 1 once the shutdown signal has been sent. Blocking until the shutdown signal is sent is implemented by waiting on the semaphore. The semaphore also supports non-blocking polling, therefore to see if the signal has been sent, I can poll the semaphore to see if its counter is above 0.

For better performance, on shutdown, the semaphore is incremented to the maximum possible value the semaphore can hold instead of incrementing it to 1 because that allows multiple threads waiting on the semaphore to wake up simultaneously.

The following code snippets show how the interface of the `ShutdownSignal` is implemented:

```
struct ShutdownSignal {
    terminated: Semaphore,
}

fn poll_terminated(&self) -> bool {
    self.terminated.try_acquire().is_ok()
}

async fn wait_for_shutdown(&self) {
    let permit = self.terminated.acquire().await;
}  // NB: permit is dropped here, which signals to the semaphore

fn shutdown(&self) {
    self.terminated.add_permits(MAX_PERMITS);
}
```

[5] https://man7.org/linux/man-pages/man8/tc.8.html

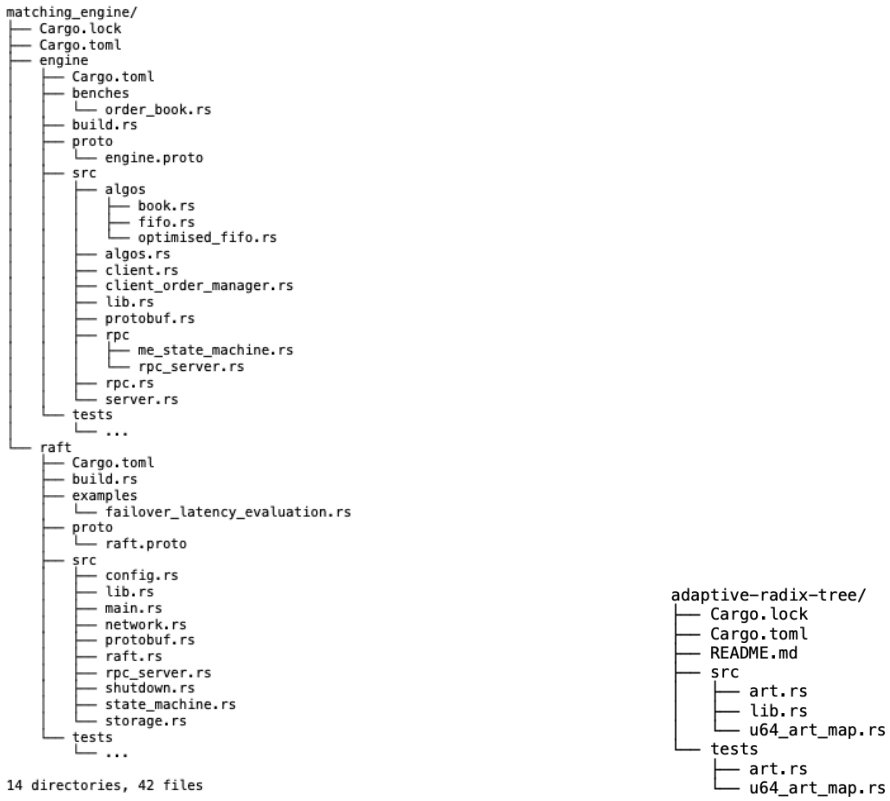



Figure 3.5: The file hierarchy of the source-code directory.

## 3.7 Repository overview

The overview of the source code repository is shown in Figure 3.5.

The code for the matching engine is divided into two modules: `engine` and `raft`. The raft component contains code for state-machine replication using the Raft protocol, while the engine component depends on the raft module to build a replicated matching engine and a client port. Furthermore, the Adaptive Radix Tree data structure is implemented as a separate `adaptive-radix-tree` module.

The code in the submodules is structured according to Rust's default file hierarchy. The main code is in `src`, unit tests are within the source files, integration and exploratory tests are in the `tests` directory. `benches` contains benchmarking code, the `proto` directories contain the protobuf-based gRPC definitions, while the `build.rs` files build the RPC stubs from these definitions. The `Cargo.toml` files contain the dependencies and define the entry points of the applications and libraries.

The raft and matching engine components were written from scratch, while the Adaptive Radix Tree implementation is a port of a popular and cleanly-written C-implementation[6] to Rust. I also adapted the ART implementation considerably to satisfy Rust's ownership model (which guarantees memory safety for the implementation), while also ensuring that correctness and performance was not sacrificed during this rewrite.

[6] `https://github.com/armon/libart`

# Chapter 4

# Evaluation

This chapter evaluates the performance of the Raft module, the matching algorithms and the replicated matching engine to provide evidence that the proposed success criteria have been met.

## 4.1 Failover latency

Failover latency is the time it takes for a new Raft node to be elected as the leader and take over handling client requests after the previous leader failed. During this time the cluster is unavailable to client requests, therefore, it has to be minimized to ensure that too many client requests do not build up in the meantime.

Failover latency is dependent on the re-election timeout used, which is the timeout period follower nodes use to wait for messages from the leader before they try to take over leadership.

### 4.1.1 Experimental setup

This experiment aims to reproduce the experimental setup detailed in the original Raft paper [5] and compare the performance of this project's implementation to the data presented in the paper.

The conditions of this experiment try to simulate a worst-case scenario. To trigger failover, we wait until a leader is elected, and kill the elected leader in a simulated environment. Then, the leader is stopped in the middle of its heartbeat period to force a long re-election timeout before failing over to the next leader. Finally, I also made the first re-election timeout coincide for each candidate, making it take even longer to failover. The same conditions were simulated in the Raft paper.

We are using a three-node Raft cluster for the experiment, and we performed the experiment 100 times for each re-election timeout range measuring the failover latencies.

### 4.1.2 Results

Figure 4.1 shows the cumulative distribution function of the failover latency of Raft as the re-election timeout range varies. It shows that my solution obtains similar failover

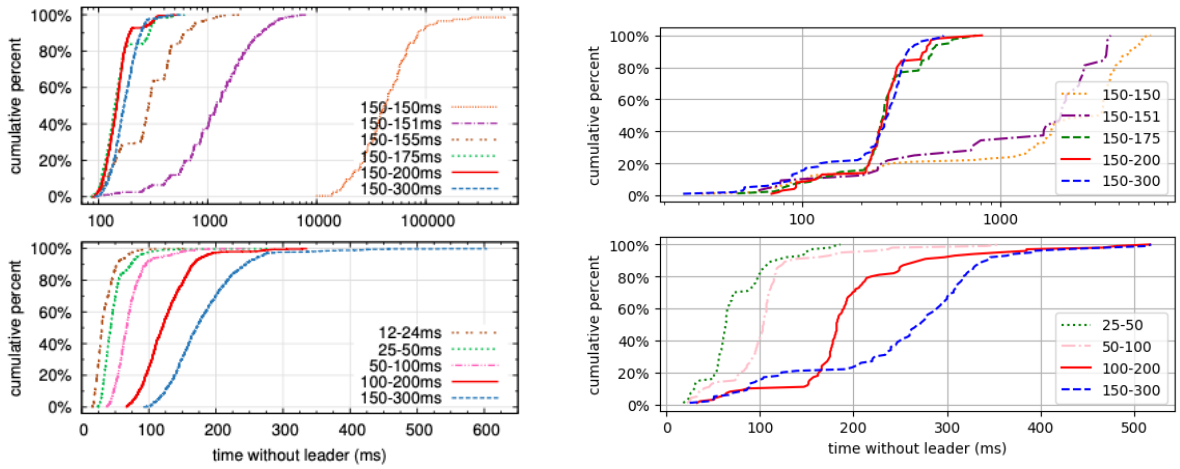



Figure 4.1: Cumulative distribution of the failover latency of Raft as the re-election timeout range varies. The re-election timeout is set uniformly randomly within the range (min-max). The results from the original Raft paper are presented on the left, while my results are presented on the right.

latencies to those presented in the original Raft paper [5].

I omitted the 12-24ms range, as the results were incomparable to those in the Raft paper. This is because our experimental setup is constrained, and, as the several threads of the three Raft nodes are concurrently competing for the limited CPU time of a single, dual-core machine, the 12-24ms timeout range was not long enough for each election thread to be eventually scheduled to the CPU.

Otherwise, the figures are showing the same trend that the higher the bounds of the timeout range are, the longer it takes on average to converge to a new leader. Furthermore, the shorter the range, the more likely it is for elections to fail and take longer than the average case. For example, the timeout ranges of 150-150ms and 150-151ms have approximately 10-100-times worse P99 failover latencies than the rest of the measured timeout ranges.

## 4.2 Order book matching latency

As the number of orders on the market grows, the time it takes to process orders by the local order book becomes a significant part of the end-to-end latency, especially in the common case of orders arriving in bursts – for instance, as a reaction to a recent news event.

This section aims to provide an overview of how the matching latency of my baseline order book implementation compares to an open source order book implementation. Then, it also analyses the matching latency improvement of the optimised order book extension using the Adaptive Radix Tree data structure.

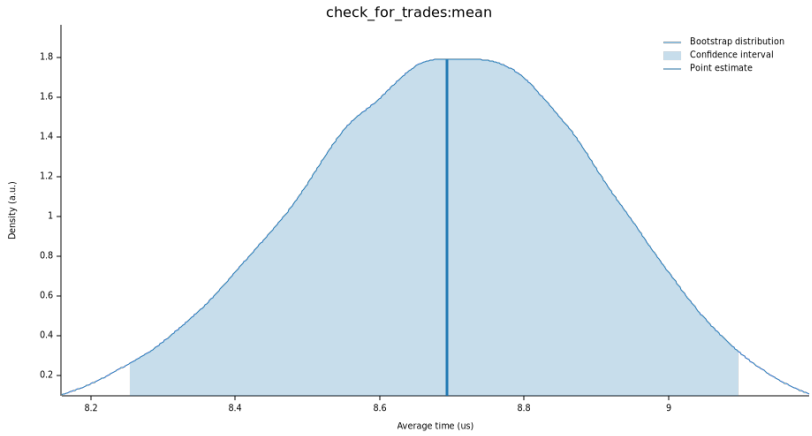



Figure 4.2: Mean latency of checking for trades in the order book of 1000 orders on each side with a single overlapping order-pair. The figure shows the distribution of the mean latency measured over 100 iterations, with marks at the mean and the 95% confidence intervals.

### 4.2.1 Experimental setup

To examine the matching latency of the baseline implementation, I have performed two experiments.

First, to measure the average-case matching latency, I measured the mean latency of checking for trades in a single, local order book with 1000 orders on each side, with a single pair of orders that can be traded. This experiment was repeated 100 times, and the mean latency is reported with a 95% confidence interval.

Next, to aid future performance improvements, I profiled an experiment of processing 1000 pre-generated orders with a high likelihood of having trades between them. This was to see how the latency of adding a new order to the book, and checking for tradable orders compare, and to see which sub-operations are the bottleneck for each.

### 4.2.2 Results

#### Baseline

The distribution of the mean latency of checking for trades is shown on Figure 4.2. It shows that the mean latency for this operation is 8.7µs with 95% confidence intervals at 8.25µs and 9.1µs.

The mean latency of my implementation is higher than the mean latency of 1µs for Exchange Core shown on Figure 4.3. This is due to the matching algorithm being a naive, priority queue-based implementation.

To confirm that it is the data structure choice that was limiting the performance of matching, I profiled the code for matching a sequence of 1000 trades. This is shown on Figure 4.4.

It confirmed that the majority of the time was spent in the data structure operations of

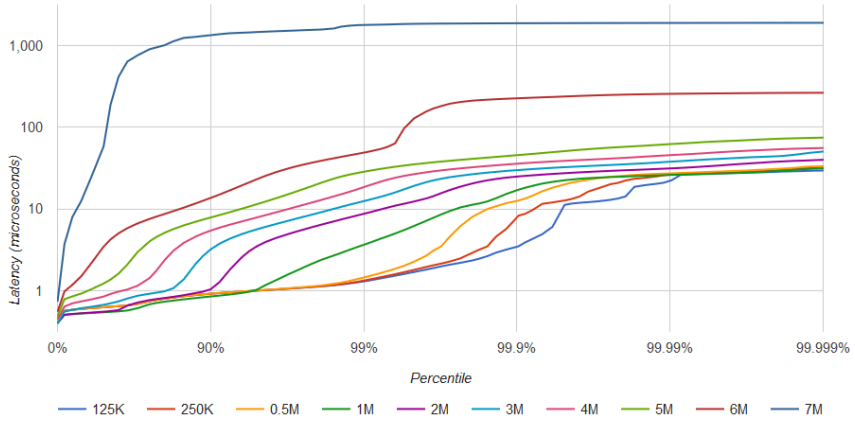



Figure 4.3: Cumulative distribution of the matching latency of the open-source Exchange Core matching engine.

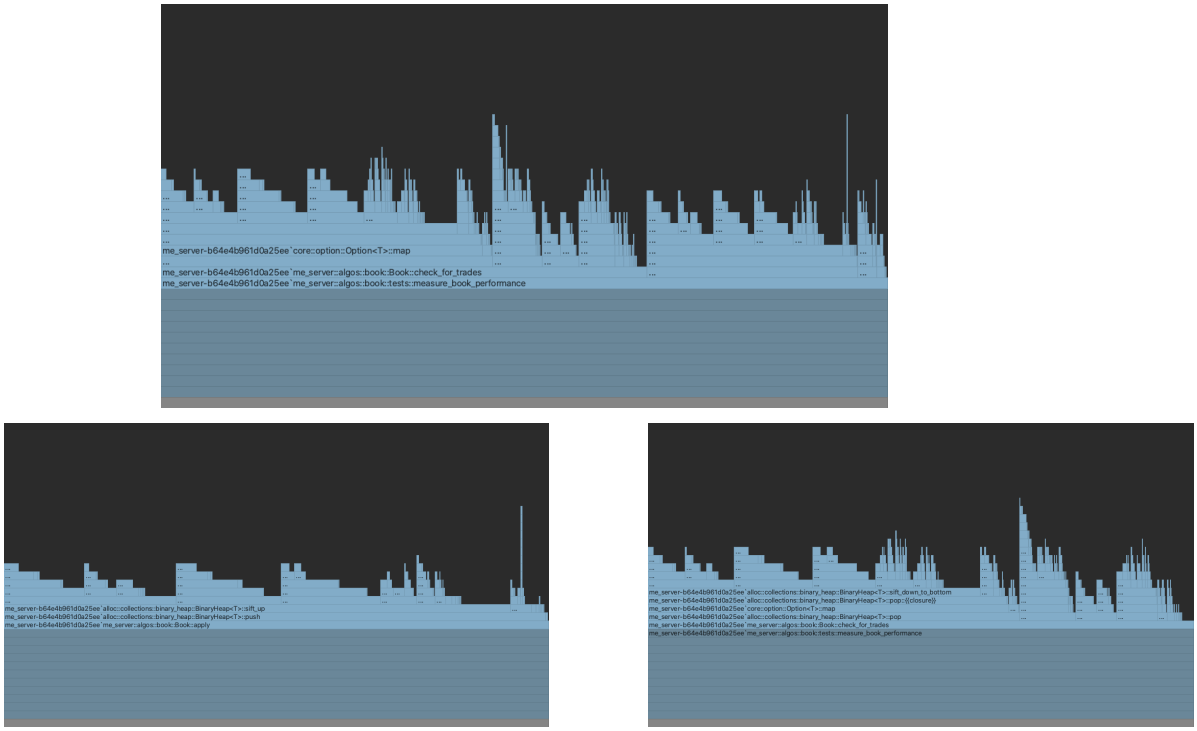


Figure 4.4: Flame graph profiles of matching a sequence of 1000 random orders with a high chance of trades. The top graph shows the time spent in each function showing both inserting the order to the limit book and checking for trades in the order book. The bottom left graph shows the profile of adding a new order to the book, while the bottom right graph shows the profile of checking for trades in the book.

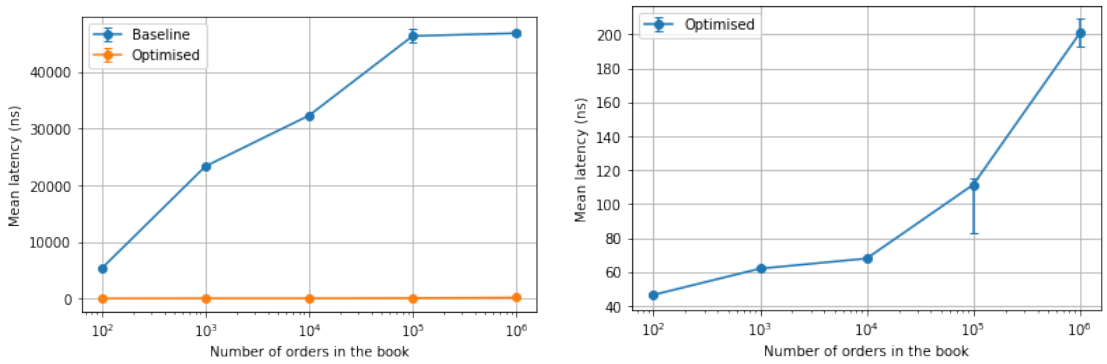



Figure 4.5: Latency of matching a large order using the baseline and the optimised order book implementations as the number of orders in the book varies.

the Binary Heap. The flame graph[1] shows that adding the order to the book contributed to around 65% of the overall order-processing latency, while the rest was due to checking for trades in the book. Adding a new order to the book was implemented by pushing it to a priority queue, which is reflected on the bottom left flame graph. The bottom right flame graph shows that checking for trades in the book is dominated by the binary heap operation, however, it spends a comparable amount of time on reducing the volumes of orders in the book and constructing trade objects.

### Optimisation extension

Given the evaluation data on the baseline matching algorithm implementation, I implemented an optimised version of the algorithm as an extension, and also evaluated its performance.

I performed the following experiment. I inserted a varying number of orders (shown on the x-axis) into a book. For the $N$-order case, I inserted $N$ buy and $N$ sell orders with price uniformly distributed in the $[1000, 1100]$ price-range, and volume uniformly distributed in the $[1, 20]$ units range. Then, I cleared out the tradeable orders in the book. Finally, I measured the matching latency in an event that is expected to trigger a high latency, which is an incoming sell order of size 1000 at price 1000 that matches with a large number of buy orders.

The experiment was run with both the baseline and the optimised order book implementations 100 times. I reported the mean latencies of the operation with error bars at 95% confidence intervals (some of them too small to show). Figure 4.5 shows that the optimised order book performs significantly better, matching new orders with a latency of 40-200 nanoseconds (depending on the order book size) compared to the baseline implementation performing the operation with a latency of 5-50 microseconds.

Figure 4.6 shows the performance of the optimised book in detail, showing the significantly reduced iteration times and the probability density function of the iteration times with 95% confidence intervals.

[1]On the flame graph, each vertical bar represents a function call, its length represents the CPU time spent in the function call, while the bars above it are the functions this one calls into.

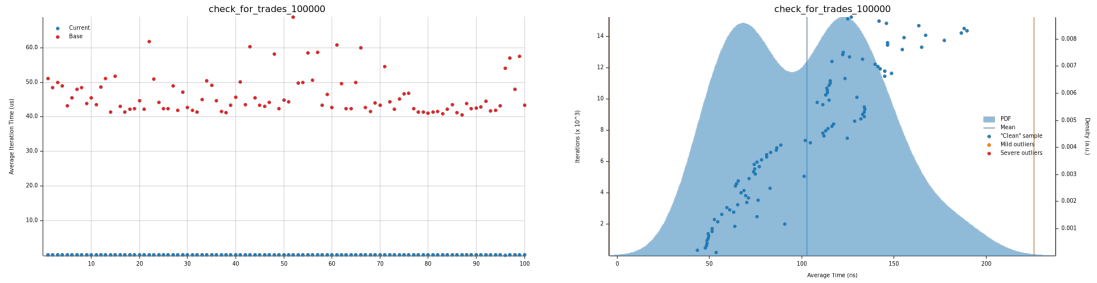



Figure 4.6: Latency of matching a large order with approximately 100000 orders in the book. On the left, the measured latencies are shown for both the baseline (red) and the optimised (blue) order books. On the right, the mean and the PDF of the measured latencies are shown in addition to the individual measurements for the optimised order book.

## 4.3 End-to-end latency and throughput

This experiment aims to reproduce the evaluation results of etcd[2] in a small-scale scenario. Etcd is a similar, Raft-based replicated state machine typically used to store configuration data networked applications. This evaluation shows that the performance results of etcd are comparable to my implementation, and by extrapolating using etcd's performance benchmarks, we can assume that my implementation is capable of satisfying the performance requirements of a stock exchange in a high-performance environment.

### 4.3.1 Experimental setup

I performed two experiments both measuring end-to-end latency and throughput of the replicated matching.

First, I measured the performance of the engine handling 15000 client orders uniformly distributed across a varying number of clients. The engine was running on top of a three-node Raft-cluster in this experiment.

Next, I measured the performance of the engine handling, each sending 7500 requests in a burst. This time, the number of clients was fixed, instead, I varied the cluster size.

For each configuration of both experiments, I ran the experiment in that configuration 10 times, measured the mean latency and throughput, and reported the P10, P50 and P90 latency and throughput of the means over these 10 iterations. On the latency and throughput graphs (figures 4.7 and 4.9), the P10, P50 and P90 performance measurements are shown as a single datapoint with error bars.

### 4.3.2 Results

Figure 4.7 shows the performance of a three-node matching engine cluster as the number of client connections vary. It shows that, when the number of client connections is small, the throughput of requests grows as the number of client connections grow, while the latency

[2] `https://etcd.io/docs/v3.4.0/op-guide/performance/`

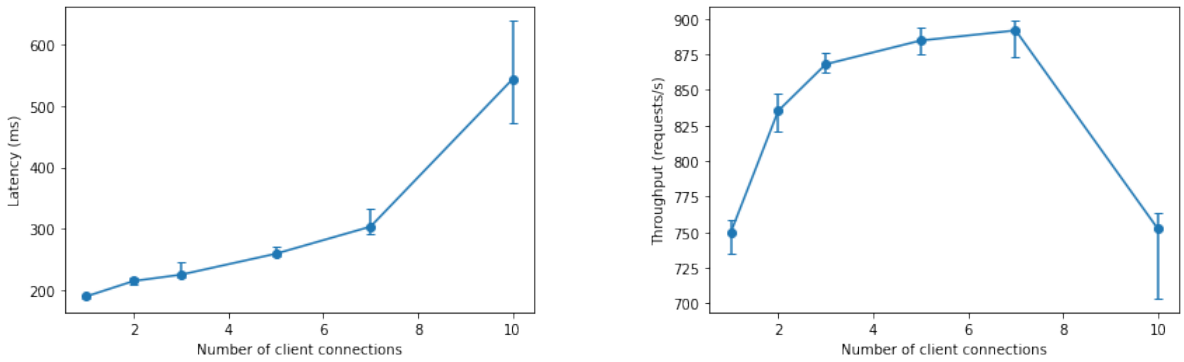



Figure 4.7: Performance of the replicated matching engine using a three node cluster, measured against a varying number of clients. Each node represents the mean of 10 trials with that configuration, with error bars at the P10 and P90 of the means. Some of the error bars are too small to show.

| Number of keys | Key size in bytes | Value size in bytes | Number of connections | Number of clients | Target etcd server | Average write QPS | Average latency per request | Average server RSS |
|---|---|---|---|---|---|---|---|---|
| 10,000 | 8 | 256 | 1 | 1 | leader only | 583 | 1.6ms | 48 MB |
| 100,000 | 8 | 256 | 100 | 1000 | leader only | 44,341 | 22ms | 124MB |
| 100,000 | 8 | 256 | 100 | 1000 | all members | 50,104 | 20ms | 126MB |

Figure 4.8: Performance of a three member etcd cluster with three replica machines of 8 vCPUs + 16GB Memory + 50GB SSD and 1 client machine of 16 vCPUs + 30GB Memory + 50GB SSD.

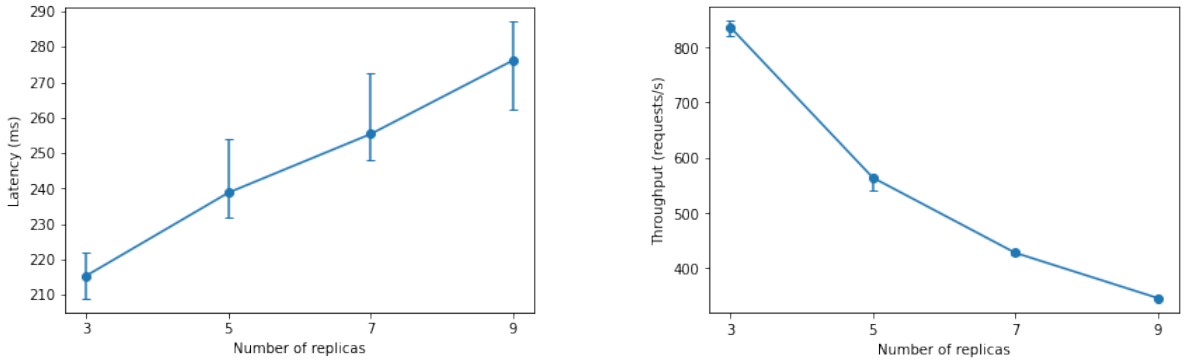



Figure 4.9: Performance of the replicated matching engine as the cluster size varies. Each node represents the mean of 10 trials with that configuration, with error bars at the P10 and P90 of the means. Some of the error bars are too small to show.

is only increasing at a slower rate. However, as the evaluation results were collected on a dual-core machine, the benefit of using more clients slows down, and thrashing reduces performance after about 7 client connections.

The matching engine's throughput (shown on Figure 4.7) is similar to that of etcd in the case of using only a few client connections (shown on Figure 4.8). The throughput is expected to scale similarly to how etcd scales, as they both use Raft as the underlying consensus algorithm. Therefore, I expect the replicated matching engine to be able to scale to the requirements of serving 1000 clients at a throughput of around 100000 requests per second.

The latency of the matching engine is higher than that of etcd. By examining the log traces, this seems to be because my implementation currently waits until the next heartbeat to communicate that new logs have been committed. An example trace from which this is visible is available in Appendix A.

I have also measured the performance of the matching engine as the cluster size increases. Figure 4.9 shows that, as the number of replicas in the cluster increases, performance degrades moderately. For example, to tolerate $f = 2$ failures instead of 1, we need a cluster size of $2f + 1 = 5$ instead of 3, which increases latency by around 10%, while throughput decreases by around 30%. This is expected, as the time it takes to replicate to a higher number of nodes increases the end-to-end latency and reduces throughput.

The fact that the evaluation results were collected on a dual-core machine accelerates the rate at which performance degrades as the number of replicas increases.

## 4.4 Success criteria

The project has achieved all of the success criteria set out in the proposal:

- I implemented the voting and log replication functionality of the Raft protocol as a module as specified in the Raft paper [5].

- I evaluated the throughput and latency of the Raft module comparing it to the results of similar log replication protocols, including the implementation in the original Raft paper [5] and etcd[3].
- I built a replicated matching engine that tolerates node failures. When tested with 5 nodes, it:
  - acknowledges successfully received Buy, Sell and Cancel client requests;
  - notifies clients about executed trades;
  - continues to function properly after 2 nodes have failed.
- I evaluated the average latency and throughput of the matching engine with a small number of nodes under varying network and failure conditions. I also compared the results to an open source matching engine implementation (exchange-core[4]).

Therefore, all of the proposed success criteria were met.

[3] `https://etcd.io`
[4] `https://github.com/mzheravin/exchange-core`

# Chapter 5

# Conclusions

In this dissertation, I described the implementation and evaluation of a software solution to providing consistent automated failover for a simplified exchange. It could help prevent similar exchange outages to the one exhibited in section 1.2 and improve the availability of the stock exchange. While the solution has limitations, it achieved all of the success criteria set out in the proposal and, with further work, it could serve as the core of an exchange.

## 5.1 Future work

Further work could, first of all, improve the business logic of the matching engine by implementing more sophisticated order management, such as prioritising orders by type, adding support for modifying orders, and similar. To guide what types of operations exchanges need to implement, one can refer to the FIX protocol.[1]

Furthermore, the current Raft implementation stores the logs in memory, whereas a real exchange would need to persist these logs for auditing purposes. For this, one could use a lightweight, persistent key-value store such as LMDB[2] or LevelDB[3], or save state to persistently storage as per the Raft paper [5].

Moreover, in my implementation, when client ports send orders to the matching engine, they send it to all of the replicas using gRPC streaming. While this is flexible, a more efficient implementation could exploit the internal structure of the exchange and use a protocol that supports broadcasting. UDP broadcast or multicast with Rust's serde[4] library for serialisation would be a possible choice for implementing this.

Finally, remote direct memory access (RDMA) devices have been used effectively for state machine replication to bring replication latency to under a microsecond using an in-memory backend for storing logs [19]. A replicated matching engine could benefit from this speedup; however, persistence would likely become a bottleneck when using RDMA.

A stock exchange is a complex system with several components interacting; therefore, integrating the matching engine with other exchange components could have its challenges.

[1] `https://www.fixtrading.org/what-is-fix/`
[2] `https://symas.com/lmdb/technical/`
[3] `https://github.com/google/leveldb`
[4] `https://crates.io/crates/serde`

Using formal verification methods [20] to verify the safety and liveness of the integration could help improve the confidence in the correctness of the system design.

## 5.2 Lessons learnt

This project was the first time I used Rust and implemented a complex concurrent and distributed system. While it took some time to get started with Rust, it proved invaluable for eliminating data races and undefined behaviour, common in low-level concurrent code, without sacrificing performance.

I have learnt a lot about implementing and improving consensus protocols, and the difficulties of testing concurrent and distributed code, and successfully integrating complex business requirements with an abstract consensus protocol.

# Bibliography


[1] Adam Smith et al. The wealth of nations. 1776.

[2] Ranald Michie. *The London stock exchange: A history*. OUP Oxford, 2001.

[3] Popper. The Stock Market Bell Rings, Computers Fail, Wall Street Cringes. *The New York Times*.

[4] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[5] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[6] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[7] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*, 2016.

[8] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313, 2009.

[9] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[10] Senate and House of Representatives of the United States of America in Congress assembled. Securities exchange act of 1934, 1934. `https://www.govinfo.gov/content/pkg/COMPS-1885/pdf/COMPS-1885.pdf`.

[11] NYSE Arca and NYSE Amex Options Trading. How NYSE Amex Options NYSE Arca Options Work. `https://www.nyse.com/publicdocs/nyse/markets/american-options/How_NYSE_Amex_Options_NYSE_Arca_Options_Work.pdf`. Accessed: 13-04-2021.

[12] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P Birman. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems (TOCS)*, 36(2):1–49, 2019.

[13] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, 1987.

[14] Ensar Basri Kahveci and Fatma Kahveci. Improved Majority Quorums for Raft. `https://drive.google.com/drive/folders/1nV6ONpO5onwIFsLETfBPiYwAkFBIV-9-`. Accessed: 13-04-2021.

[15] R Preston McAfee. A dominant strategy double auction. *Journal of economic Theory*, 56(2):434–450, 1992.

[16] CME Group. Match algorithms. `http://web.archive.org/web/20120626161034/http://www.cmegroup.com/confluence/display/EPICSANDBOX/Match+Algorithms`. Accessed: 13-04-2021.

[17] McKay Brothers. Products. `https://www.mckay-brothers.com/product-page/#latencies`. Accessed: 28-04-2021.

[18] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[19] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118, 2015.

[20] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

# Appendix A

# Raft trace

This section aims to provide additional context into how Raft (and my implementation in particular) works by presenting and discussing a detailed log trace of replicating ten orders to three Raft replicas.

## A.1 Comments

The log trace starts with a sequence of request-response logs of heartbeat messages, which are implemented as empty AppendEntries (AE) RPC calls.

Then, the client entries successively arrive (first at 12:49:55.565), and they are saved to the local log of the leader. A few milliseconds later an early AE request is triggered, and the first two orders are sent to the other replicas. When they have applied these orders, they respond to the AE requests. When the leader receives the AE responses, it knows that it is safe to commit the orders, therefore it commits them locally and the updated commit index will be propagated to the follower nodes with the next AE request.

The trace shows how the match index and the commit index are incremented, triggering logs to be applied as soon as the logs are safely replicated. For instance, the log entry at index 4 is safely applied at 12:49:55.609 after a single AE response (the other replica only replies to the AE request after the log has been applied). This is safely allowed by Raft.

While the effect of pipelining Raft requests (ie. having multiple outstanding AE requests to the same client) is not exhibited in this minimal example trace, we can see that AE requests are triggered by early timeouts after calls into the start method, which allows us to pipeline requests if the batching time is shorter than the time it takes for replicas to reply to the AE request.

Once the 10 orders have been replicated and applied at all three replicas, the Raft messages shown are only heartbeat messages, which is the steady-state of Raft.

## A.2 Trace

```
...
Apr 19 12:49:55.516  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (1-1)
```

```
Apr 19 12:49:55.516  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (1-1)
Apr 19 12:49:55.516  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.516  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.519  INFO raft::raft: append entries succeeded, match index to 1
    is now 0 from 0
Apr 19 12:49:55.519  INFO raft::raft: Updating leader commit index from 0 to 0
Apr 19 12:49:55.519  INFO raft::raft: append entries succeeded, match index to 2
    is now 0 from 0
Apr 19 12:49:55.519  INFO raft::raft: Updating leader commit index from 0 to 0
Apr 19 12:49:55.565  INFO start{command=Command { r#type: Sell, sequence_id: 1,
    price: 382, client_id: 1, size: 1 }}: raft::raft: Starting appending  new log
    entry: LogEntry { term: 1, index: 1, command: Some(Command { r#type: Sell,
    sequence_id: 1, price: 382, client_id: 1, size: 1 }) }
Apr 19 12:49:55.565  INFO start{command=Command { r#type: Sell, sequence_id: 2,
    price: 674, client_id: 1, size: 1 }}: raft::raft: Starting appending  new log
    entry: LogEntry { term: 1, index: 2, command: Some(Command { r#type: Sell,
    sequence_id: 2, price: 674, client_id: 1, size: 1 }) }
Apr 19 12:49:55.570  INFO start{command=Command { r#type: Sell, sequence_id: 3,
    price: 767, client_id: 1, size: 1 }}: raft::raft: Starting appending  new log
    entry: LogEntry { term: 1, index: 3, command: Some(Command { r#type: Sell,
    sequence_id: 3, price: 767, client_id: 1, size: 1 }) }
Apr 19 12:49:55.571  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (1-3)
Apr 19 12:49:55.572  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (1-3)
Apr 19 12:49:55.572  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.572  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.576  INFO raft::raft: append entries succeeded, match index to 2
    is now 2 from 0
Apr 19 12:49:55.576  INFO raft::raft: Updating leader commit index from 0 to 2
Apr 19 12:49:55.576  INFO raft::raft: Applying command 1: Command { r#type:
    Sell, sequence_id: 1, price: 382, client_id: 1, size: 1 }
Apr 19 12:49:55.576  INFO raft::raft: Applying command 2: Command { r#type:
    Sell, sequence_id: 2, price: 674, client_id: 1, size: 1 }
Apr 19 12:49:55.578  INFO raft::raft: append entries succeeded, match index to 1
    is now 2 from 0
Apr 19 12:49:55.578  INFO raft::raft: Updating leader commit index from 2 to 2
Apr 19 12:49:55.581  INFO start{command=Command { r#type: Buy, sequence_id: 4,
    price: 415, client_id: 1, size: 1 }}: raft::raft: Starting appending  new log
    entry: LogEntry { term: 1, index: 4, command: Some(Command { r#type: Buy,
    sequence_id: 4, price: 415, client_id: 1, size: 1 }) }
Apr 19 12:49:55.583  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (3-4)
Apr 19 12:49:55.584  INFO handle_append_entries: raft::raft: Applying command 1:
    Command { r#type: Sell, sequence_id: 1, price: 382, client_id: 1,  size: 1 }
Apr 19 12:49:55.585  INFO handle_append_entries: raft::raft: Applying command 2:
    Command { r#type: Sell, sequence_id: 2, price: 674, client_id: 1,  size: 1 }
Apr 19 12:49:55.585  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.585  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (3-4)
Apr 19 12:49:55.586  INFO handle_append_entries: raft::raft: Applying command 1:
    Command { r#type: Sell, sequence_id: 1, price: 382, client_id: 1,  size: 1 }
Apr 19 12:49:55.588  INFO handle_append_entries: raft::raft: Applying command 2:
    Command { r#type: Sell, sequence_id: 2, price: 674, client_id: 1,  size: 1 }
Apr 19 12:49:55.588  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.591  INFO raft::raft: append entries succeeded, match index to 2
    is now 3 from 2
Apr 19 12:49:55.592  INFO raft::raft: Updating leader commit index from 2 to 3
Apr 19 12:49:55.592  INFO raft::raft: Applying command 3: Command { r#type:
    Sell, sequence_id: 3, price: 767, client_id: 1, size: 1 }
```

```
Apr 19 12:49:55.596  INFO start{command=Command { r#type: Buy, sequence_id: 5,
    price: 206, client_id: 1, size: 1 }}: raft::raft: Starting appending  new log
    entry: LogEntry { term: 1, index: 5, command: Some(Command { r#type: Buy,
    sequence_id: 5, price: 206, client_id: 1, size: 1 }) }
Apr 19 12:49:55.597  INFO start{command=Command { r#type: Sell, sequence_id: 6,
    price: 261, client_id: 1, size: 1 }}: raft::raft: Starting appending  new log
    entry: LogEntry { term: 1, index: 6, command: Some(Command { r#type: Sell,
    sequence_id: 6, price: 261, client_id: 1, size: 1 }) }
Apr 19 12:49:55.597  INFO raft::raft: append entries succeeded, match index to 1
    is now 3 from 2
Apr 19 12:49:55.597  INFO raft::raft: Updating leader commit index from 3 to 3
Apr 19 12:49:55.601  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (4-5)
Apr 19 12:49:55.601  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.602  INFO start{command=Command { r#type: Buy, sequence_id: 7,
    price: 58, client_id: 1, size: 1 }}: raft::raft: Starting appending new  log
    entry: LogEntry { term: 1, index: 7, command: Some(Command { r#type: Buy,
    sequence_id: 7, price: 58, client_id: 1, size: 1 }) }
Apr 19 12:49:55.603  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (4-5)
Apr 19 12:49:55.604  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.608  INFO raft::raft: append entries succeeded, match index to 1
    is now 4 from 3
Apr 19 12:49:55.609  INFO raft::raft: Updating leader commit index from 3 to 4
Apr 19 12:49:55.609  INFO raft::raft: Applying command 4: Command { r#type: Buy,
    sequence_id: 4, price: 415, client_id: 1, size: 1 }
Apr 19 12:49:55.609  INFO start{command=Command { r#type: Buy, sequence_id: 8,
    price: 291, client_id: 1, size: 1 }}: raft::raft: Starting appending  new log
    entry: LogEntry { term: 1, index: 8, command: Some(Command { r#type: Buy,
    sequence_id: 8, price: 291, client_id: 1, size: 1 }) }
Apr 19 12:49:55.610  INFO raft::raft: append entries succeeded, match index to 2
    is now 4 from 3
Apr 19 12:49:55.610  INFO raft::raft: Updating leader commit index from 4 to 4
Apr 19 12:49:55.614  INFO start{command=Command { r#type: Sell, sequence_id: 9,
    price: 386, client_id: 1, size: 1 }}: raft::raft: Starting appending  new log
    entry: LogEntry { term: 1, index: 9, command: Some(Command { r#type: Sell,
    sequence_id: 9, price: 386, client_id: 1, size: 1 }) }
Apr 19 12:49:55.616  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (5-5)
Apr 19 12:49:55.616  INFO handle_append_entries: raft::raft: Applying command 3:
    Command { r#type: Sell, sequence_id: 3, price: 767, client_id: 1,  size: 1 }
Apr 19 12:49:55.616  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.618  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (5-5)
Apr 19 12:49:55.618  INFO handle_append_entries: raft::raft: Applying command 3:
    Command { r#type: Sell, sequence_id: 3, price: 767, client_id: 1,  size: 1 }
Apr 19 12:49:55.619  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.621  INFO raft::raft: append entries succeeded, match index to 2
    is now 4 from 4
Apr 19 12:49:55.621  INFO raft::raft: Updating leader commit index from 4 to 4
Apr 19 12:49:55.625  INFO raft::raft: append entries succeeded, match index to 1
    is now 4 from 4
Apr 19 12:49:55.625  INFO raft::raft: Updating leader commit index from 4 to 4
Apr 19 12:49:55.629  INFO start{command=Command { r#type: Buy, sequence_id: 10,
    price: 220, client_id: 1, size: 1 }}: raft::raft: Starting appending  new log
    entry: LogEntry { term: 1, index: 10, command: Some(Command { r#type: Buy,
    sequence_id: 10, price: 220, client_id: 1, size: 1 }) }
Apr 19 12:49:55.634  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (5-7)
Apr 19 12:49:55.634  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (5-7)
Apr 19 12:49:55.634  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
```

```
Apr 19 12:49:55.634  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.637  INFO raft::raft: append entries succeeded, match index to 2
    is now 6 from 4
Apr 19 12:49:55.637  INFO raft::raft: Updating leader commit index from 4 to 6
Apr 19 12:49:55.638  INFO raft::raft: Applying command 5: Command { r#type: Buy,
    sequence_id: 5, price: 206, client_id: 1, size: 1 }
Apr 19 12:49:55.638  INFO raft::raft: Applying command 6: Command { r#type:
    Sell, sequence_id: 6, price: 261, client_id: 1, size: 1 }
Apr 19 12:49:55.641  INFO raft::raft: append entries succeeded, match index to 1
    is now 6 from 4
Apr 19 12:49:55.641  INFO raft::raft: Updating leader commit index from 6 to 6
Apr 19 12:49:55.647  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (7-8)
Apr 19 12:49:55.648  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.648  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (7-8)
Apr 19 12:49:55.648  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.650  INFO raft::raft: append entries succeeded, match index to 2
    is now 7 from 6
Apr 19 12:49:55.650  INFO raft::raft: Updating leader commit index from 6 to 7
Apr 19 12:49:55.650  INFO raft::raft: Applying command 7: Command { r#type: Buy,
    sequence_id: 7, price: 58, client_id: 1, size: 1 }
Apr 19 12:49:55.655  INFO raft::raft: append entries succeeded, match index to 1
    is now 7 from 6
Apr 19 12:49:55.655  INFO raft::raft: Updating leader commit index from 7 to 7
Apr 19 12:49:55.656  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (8-10)
Apr 19 12:49:55.657  INFO handle_append_entries: raft::raft: Applying command 4:
    Command { r#type: Buy, sequence_id: 4, price: 415, client_id: 1,  size: 1 }
Apr 19 12:49:55.657  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.663  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (8-10)
Apr 19 12:49:55.663  INFO raft::raft: append entries succeeded, match index to 2
    is now 9 from 7
Apr 19 12:49:55.663  INFO raft::raft: Updating leader commit index from 7 to 9
Apr 19 12:49:55.663  INFO raft::raft: Applying command 8: Command { r#type: Buy,
    sequence_id: 8, price: 291, client_id: 1, size: 1 }
Apr 19 12:49:55.663  INFO raft::raft: Applying command 9: Command { r#type:
    Sell, sequence_id: 9, price: 386, client_id: 1, size: 1 }
Apr 19 12:49:55.663  INFO handle_append_entries: raft::raft: Applying command 4:
    Command { r#type: Buy, sequence_id: 4, price: 415, client_id: 1,  size: 1 }
Apr 19 12:49:55.664  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.670  INFO raft::raft: append entries succeeded, match index to 1
    is now 9 from 7
Apr 19 12:49:55.670  INFO raft::raft: Updating leader commit index from 9 to 9
Apr 19 12:49:55.672  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (10-11)
Apr 19 12:49:55.673  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.677  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (10-11)
Apr 19 12:49:55.678  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.681  INFO raft::raft: append entries succeeded, match index to 1
    is now 10 from 9
Apr 19 12:49:55.682  INFO raft::raft: Updating leader commit index from 9 to 10
Apr 19 12:49:55.682  INFO raft::raft: Applying command 10: Command { r#type:
    Buy, sequence_id: 10, price: 220, client_id: 1, size: 1 }
Apr 19 12:49:55.682  INFO raft::raft: append entries succeeded, match index to 2
    is now 10 from 9
Apr 19 12:49:55.682  INFO raft::raft: Updating leader commit index from 10 to 10
Apr 19 12:49:55.686  INFO raft::rpc_server: AppendEntriesRequest term (1), log
```

```
    indices (11-11)
Apr 19 12:49:55.687  INFO handle_append_entries: raft::raft: Applying command 5:
    Command { r#type: Buy, sequence_id: 5, price: 206, client_id: 1,  size: 1 }
Apr 19 12:49:55.687  INFO handle_append_entries: raft::raft: Applying command 6:
    Command { r#type: Sell, sequence_id: 6, price: 261, client_id: 1,  size: 1 }
Apr 19 12:49:55.687  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.688  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (11-11)
Apr 19 12:49:55.688  INFO handle_append_entries: raft::raft: Applying command 5:
    Command { r#type: Buy, sequence_id: 5, price: 206, client_id: 1,  size: 1 }
Apr 19 12:49:55.689  INFO handle_append_entries: raft::raft: Applying command 6:
    Command { r#type: Sell, sequence_id: 6, price: 261, client_id: 1,  size: 1 }
Apr 19 12:49:55.689  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.691  INFO raft::raft: append entries succeeded, match index to 2
    is now 10 from 10
Apr 19 12:49:55.691  INFO raft::raft: Updating leader commit index from 10 to 10
Apr 19 12:49:55.694  INFO raft::raft: append entries succeeded, match index to 1
    is now 10 from 10
Apr 19 12:49:55.694  INFO raft::raft: Updating leader commit index from 10 to 10
Apr 19 12:49:55.696  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (11-11)
Apr 19 12:49:55.696  INFO handle_append_entries: raft::raft: Applying command 7:
    Command { r#type: Buy, sequence_id: 7, price: 58, client_id: 1, size:  1 }
Apr 19 12:49:55.696  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.698  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (11-11)
Apr 19 12:49:55.698  INFO handle_append_entries: raft::raft: Applying command 7:
    Command { r#type: Buy, sequence_id: 7, price: 58, client_id: 1, size:  1 }
Apr 19 12:49:55.698  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.698  INFO raft::raft: append entries succeeded, match index to 2
    is now 10 from 10
Apr 19 12:49:55.698  INFO raft::raft: Updating leader commit index from 10 to 10
Apr 19 12:49:55.700  INFO raft::raft: append entries succeeded, match index to 1
    is now 10 from 10
Apr 19 12:49:55.700  INFO raft::raft: Updating leader commit index from 10 to 10
Apr 19 12:49:55.702  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (11-11)
Apr 19 12:49:55.702  INFO handle_append_entries: raft::raft: Applying command 8:
    Command { r#type: Buy, sequence_id: 8, price: 291, client_id: 1,  size: 1 }
Apr 19 12:49:55.703  INFO handle_append_entries: raft::raft: Applying command 9:
    Command { r#type: Sell, sequence_id: 9, price: 386, client_id: 1,  size: 1 }
Apr 19 12:49:55.703  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.703  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (11-11)
Apr 19 12:49:55.704  INFO handle_append_entries: raft::raft: Applying command 8:
    Command { r#type: Buy, sequence_id: 8, price: 291, client_id: 1,  size: 1 }
Apr 19 12:49:55.704  INFO handle_append_entries: raft::raft: Applying command 9:
    Command { r#type: Sell, sequence_id: 9, price: 386, client_id: 1,  size: 1 }
Apr 19 12:49:55.704  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.705  INFO raft::raft: append entries succeeded, match index to 2
    is now 10 from 10
Apr 19 12:49:55.705  INFO raft::raft: Updating leader commit index from 10 to 10
Apr 19 12:49:55.707  INFO raft::raft: append entries succeeded, match index to 1
    is now 10 from 10
Apr 19 12:49:55.707  INFO raft::raft: Updating leader commit index from 10 to 10
Apr 19 12:49:55.709  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (11-11)
Apr 19 12:49:55.709  INFO handle_append_entries: raft::raft: Applying command
    10: Command { r#type: Buy, sequence_id: 10, price: 220, client_id: 1,  size: 1 }
Apr 19 12:49:55.709  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
```

```
Apr 19 12:49:55.710  INFO raft::rpc_server: AppendEntriesRequest term (1), log
    indices (11-11)
Apr 19 12:49:55.710  INFO handle_append_entries: raft::raft: Applying command
    10: Command { r#type: Buy, sequence_id: 10, price: 220, client_id: 1,  size: 1 }
Apr 19 12:49:55.711  INFO raft::rpc_server: AppendEntriesResponse:
    Ok(AppendEntriesResponse { term: 1, success: true })
Apr 19 12:49:55.711  INFO raft::raft: append entries succeeded, match index to 2
    is now 10 from 10
Apr 19 12:49:55.712  INFO raft::raft: Updating leader commit index from 10 to 10
Apr 19 12:49:55.713  INFO raft::raft: append entries succeeded, match index to 1
    is now 10 from 10
Apr 19 12:49:55.713  INFO raft::raft: Updating leader commit index from 10 to 10
...
```

# Appendix B

# Project Proposal

A copy of the project proposal is starting on the next page.

# High-availability matching engine of a stock exchange

*Computer Science Tripos Part II Project Proposal*

G. Peresztegi-Nagy (*gp454*), Churchill College

Originator: The Author

October 22, 2020

**Project Supervisor: Prof. J. Crowcroft**

**Director of Studies: Dr. J. K. Fawcett**

**Project Overseers: A. Moore, A. Vlachos**

## Introduction

Electronic stock exchanges allow parties to trade the listed stocks at an increasingly fast pace. To support this, they need to be able to handle persistently high loads while having to provide strong consistency, fairness and auditability guarantees. While the market is open, they are expected not to show any sign of failure and allow continuous trading.

The matching engine is the core component of the stock exchange that maintains an order book of bids and offers and executes trades when orders match up.

State machine replication is an approach that has been successfully used for ensuring the high-availability of applications. Applications defined as deterministic state machines can be replicated straightforwardly by using an algorithm that replicates the log of actions the state machine goes through.

There are many algorithms supporting state-machine replication, each adding a few ideas and improvements to earlier versions. Multi-Paxos [1], Vertical Paxos [2], Flexible Paxos [3] and Raft [4] are algorithms commonly used in industry.

The Multi-Paxos algorithm uses the idea that elected leaders can be reused in later ballot rounds to improve throughput in the case of no failures.

Flexible Paxos is an efficient Paxos implementation that provides strong consistency and failure-tolerance guarantees with the lowest messaging overhead by relaxing the requirements on phase 1 and phase 2 quorum sizes.

Raft was designed with implementation-complexity in mind meaning that the protocol is both easy to understand and easy to use for real-world applications. Therefore, it is a widely used protocol in industry and makes it a suitable choice for my project.

My project will start by specifying the Raft log replication protocol as a module in Rust. Then, it builds on top of this library to implement a highly-available matching engine.

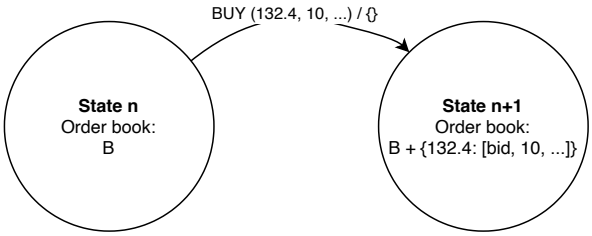
## Work to be done

The final and main goal of the project is to build a highly-available matching engine (ME) with a simplified interface:

- It accepts Buy, Sell, Cancel messages that add or remove orders to the order book at the ask/bid price specified in the messages. Then, the ME replies with acknowledgement messages when the orders are persistently stored.
- When the ME could execute trades, it sends Trade messages to all parties involved in the trade to confirm it.

I will start by implementing the Raft protocol as a module in Rust.

I will implement a non-replicated matching engine that operates as a Mealy state machine. In each state, it can accept a set of possible actions and, in response to the action, move to the desired next state and possibly generate some output. For example, receiving a buy order that does not result in a trade could be represented by the following transition:




Finally, I will integrate the matching engine with the Raft module to replicate it and thus ensure its high availability.

I will also implement mock clients sending requests to the ME to test and evaluate the system. It should support:

- Generating a stream of requests.
- Changing the request throughput by adding multiple clients generating messages.
- Testing for the consumption of expected Trade messages.

Initially, I am going to perform the evaluation in a synthetic environment (eg. Mininet[1]) to simulate an unreliable network. Both the Raft library and the replicated matching engine can be evaluated in such a synthetic environment.

I will evaluate the performance and fault-tolerance of the Raft module by comparing it to similar log replication protocols, such as LibFPaxos[2] evaluated in the Flexible Paxos paper [3, Section 6]. Furthermore, I will evaluate the performance of the replicated matching engine, profile it to see which parts have the largest effect on performance and compare its performance to open source matching engine implementations, such as exchange-core[3].

I will evaluate the Raft module and the replicated matching engine using the following metrics:

[1] `http://mininet.org/`
[2] `https://github.com/fpaxos/fpaxos-lib`
[3] `https://github.com/mzheravin/exchange-core`

- System throughput: average rate of acknowledged client requests, and the average rate of executed trades.
- System latency: average time between request send and acknowledgement.
- Resource usage: memory and CPU usage of both the Raft library and the matching engine.

All of the above metrics can be evaluated with different quorum sizes, varying network conditions, node and link failures.

## Success Criteria

This project will be considered a success if I have done all of the following:

- Implemented the voting and log replication functionality of the Raft protocol as a module as specified in the Raft paper [4].
- Evaluated the throughput and latency of the Raft module comparing it to the results of similar log replication protocols, such as the results of LibFPaxos presented in the Flexible Paxos paper [3, Section 6].
- Built a replicated matching engine that tolerates node failures. When tested with 5 nodes, it must:
  - Acknowledge successfully received Buy, Sell and Cancel client requests.
  - Notify clients about executed trades.
  - Continue to function properly after 2 nodes have failed.
- Evaluated the average latency and throughput of the matching engine with a small number of nodes under varying network and failure conditions. Compared the results to open sources matching engine implementations.

## Extensions

- Formally specify and verify the consensus protocol and/or the matching engine in TLA+ using the TLA model checker.
- Implement optimizations of the Raft protocol, such as log compaction, batching or pipelining requests.
- Evaluate the throughput limit of the system in a high-performance environment using different quorum sizes.
- Implement an advanced trading algorithm, such as bucket trading, which is atomic trading across symbols. If time permits, the algorithm could be made multi-threaded thus distributing the per-symbol work to multiple CPU cores.
- Explore optimization options for the matching algorithm, such as executing multiple non-conflicting trades simultaneously (eg. if they correspond to different single-symbol orders).

## Starting Point

I have not used Rust before the start of the project.

I have previously studied distributed systems in the IB course Concurrent and Distributed Systems.

I have previously used TLA+ for specifying and model checking simple algorithms (eg. Paxos-commit).

## Resources Required

My project will require computing resources in the form of cloud computing credit or compute time at the university's high-performance computing system. This will be needed for the evaluation phase to scale up the matching engine and evaluate its throughput limit.

I will use Rust to implement my project, so I will use its open-source compiler and tools.

I will use the open-source Mininet environment for evaluation. I have already downloaded the Mininet virtual machine for experimentation.

I will use my personal laptop (2014 MacBook Air) for testing and development. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

I will use git for version control, syncing my work to both a private GitHub repository and my MCS filespace. Additionally, I will make weekly backups to an external flash drive.

In case of failure of my machine, I can restore from one of the backups, and continue working from my backup laptop or an MCS machine. This should allow me to continue working with minimal interruption.

## Timetable

### 16 Oct - 29 Oct

Specify the design of the replicated matching engine and the testing framework in a document that can be reused when writing the Implementation chapter of the dissertation.

Start learning Rust through writing small programs, experimenting with its concurrency primitives and networking libraries.

**Milestone:** Detailed system design produced and approved by supervisor.

**Milestone:** Wrote a simple RPC server replying with current system information (eg. CPU load, memory usage), handling client requests concurrently.

### 30 Oct - 12 Nov

Start implementing the Raft module. Implement leader election and start implementing the log replication.

**Milestone:** Leader election implemented and tested.

### 13 Nov - 26 Nov

Finish implementing log replication for the Raft module.

Describe the implementation of the Raft module in a document that can be reused when writing the dissertation.

**Milestone:** All tests passing for the fully functioning Raft module.

**Milestone:** Raft implementation documentation sent to supervisor for feedback.

### 27 Nov - 10 Dec

Implement mock clients for message generation as part of the test and evaluation framework. Moreover, set up the evaluation environment for the Raft module.

Less time due to Part II Unit assessment deadline.

**Milestone:** The test framework works for testing Flexible Multi-Paxos in face of node and link failures.

### 11 Dec - 24 Dec

Evaluate the Raft module and write up the results to be included in the dissertation.

Start implementing the non-replicated ME. Implement acknowledging and storing client orders.

**Milestone:** Evaluation chapter for the Raft module sent to supervisor for feedback.

**Milestone:** The non-replicated matching engine acknowledges and stores client orders.

### 25 Dec - 7 Jan

No work planned for around Christmas time.

### 8 Jan - 21 Jan

Finish implementing the non-replicated ME. Send notifications to clients participating in executed trades.

Integrate the ME with the Raft library by replicating all orders before applying them.

**Milestone:** The replicated ME works in face of node and link failures.

### 22 Jan - 4 Feb

Document the implementation of the replicated ME in as a dissertation chapter.

Write progress report.

Prepare progress report presentation.

**Milestone:** Dissertation section on the implementation of the ME sent to supervision for feedback.

**Milestone:** Submitted progress report (by 12:00 5th Feb).

**Milestone:** Presentation prepared.

### 5 Feb - 18 Feb

Evaluate the replicated matching engine system.

Finish writing the evaluation chapter for the dissertation by extending the evaluation of the Raft module with the evaluation of the ME.

Slack time / work on extensions.

**Milestone:** Presentation presented.

**Milestone:** Draft evaluation chapter sent to supervisor for feedback.

### 19 Feb - 4 Mar

Compile and develop the prepared implementation and evaluation parts into an outline dissertation.

**Milestone:** Outline dissertation prepared.

### 5 Mar - 18 Mar

Finish the draft dissertation by extending it with the introduction, preparation and summary chapters.

Slack time / work on extensions.

**Milestone:** Draft dissertation prepared and sent to supervisor.

**Milestone:** All success criteria met.

### 19 Mar - 1 Apr

Slack time / work on extensions.

### 2 Apr - 15 Apr

Make changes to the dissertation based on feedback.

Slack time / work on extensions.

**Milestone:** Dissertation complete with first round of feedback incorporated.

### 16 Apr - 29 Apr

Make changes to the dissertation based on feedback.

Slack time / work on extensions.

**Milestone:** Dissertation complete with second round of feedback incorporated.

### 30 Apr - 14 May

Slack time.

**Milestone:** Dissertation submitted (by 12:00 14 May).

## References


[1] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[2] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313, 2009.

[3] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*, 2016.

[4] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.