Project Documentation: High-Performance Order Matching Engine

1. Introduction

This document presents a high-performance, multi-threaded Order Matching Engine designed for electronic trading systems. The engine implements a price-time priority matching algorithm with sophisticated concurrency control, enabling low-latency order processing across multiple financial instruments.

2. System Architecture

The Order Matching Engine employs a unique thread confinement model to eliminate internal locking within the core matching logic, significantly boosting throughput and predictability. Key components, including the **OrderBook**, **OrderTracker**, and a multi-stage **Scheduler** hierarchy. It is built upon a decoupled, multi-threaded architecture designed to handle concurrent order flow efficiently. The system separates the concerns of order ingestion, parsing, and scheduling from the core matching logic.

3. Core Design Principles and Justifications

The OME's performance is underpinned by several critical architectural decisions, many of which are explicitly justified within the codebase through comments.

3.1. Thread Confinement for Order Book

The architecture design decision is to keep one order book per stock symbol. Each order book is confined to a single worker thread. This design eliminates the need for internal locking within the OrderBook for matching, insertion, or updates as there will now be no race conditions in matching logic. This simplifies concurrency management and improves throughput. It also makes sure that the order that was received first will be processed first.

A single thread can handle one or multiple order books. A mechanism can be developed to change ownership of the order book from one thread to another, depending on real-time trade volume stats.

3.2. Order Ingestion and Load Balancing

The OrderInjectorScheduler (OIS) handles incoming raw order messages. Orders are assigned to OIS worker threads using a round-robin mechanism. This design ensures an even distribution of the order parsing and object creation workload across available injector threads, preventing bottlenecks before orders reach the matching engine.

3.3. Lazy Initialization

Order books are created on demand when the first order arrives. It is done to reduce the memory footprint for inactive symbols.

3.4. Order Book Registry: Multiton Pattern

Access to OrderBook instances is managed by a Registry which implements a Multiton pattern (a map from Symbol to an OrderBook instance). This registry uses a *shared_mutex* to safely manage concurrent access: a fast, read-only path is used for retrieval, and a slow, write-locked path is used for creation (*getOrCreateOrderBook*). This ensures safe, reference-counted lifetime management and prevents duplicate order books for the same symbol.

3.5. Separation of Concern: Layered Responsibility Model

The Order Matching Engine implements a strict separation of concerns principle across multiple architectural layers. Each component has a single, well-defined responsibility, making the system maintainable, testable, and scalable.

- Application Layer
  It serves as the entry point for the entire Order Matching Engine. It is responsible for loading all the configurations and initializing the needed schedulers, which in turn launch their worker thread pools

- Scheduler Layer
  It manages worker thread pools and task distribution. There are two types of Scheduler: OrderInjectorScheduler, responsible for managing the threads that will be parsing data from IPC, and OrderBookScheduler, which will be responsible for managing the threads responsible for processing orders.

- Order Validation Layer
  It ensures order integrity before instantiation. The validation chain is configured

once at application startup in the main function and set as the default validator for all Order factory methods.

- Order Book Layer
  Manages all orders for a single symbol and executes matching. The **OrderBookScheduler** ensures that all orders for a given symbol are routed to the same worker thread.

- Order Tracker Layer
  Manages one side (buy or sell) of the order book. By separating the buy and sell sides, each can have its own priority ordering without interfering with the other.

- Price Level Layer
  Manages all orders at a specific price point and is responsible for matching orders at this price point. By isolating the matching logic at the price level, the system can reason about trade execution independently of the overall order book structure. The price level doesn't need to know about price priority or the buy/sell distinction—it simply manages orders at one price and executes trades.

4. **Core Matching Components**

The matching engine is composed of three tightly integrated classes: OrderBook, OrderTracker, and PriceLevel.

Component	**Responsibility**
OrderBook	Manages the entire book for a single symbol. Delegates buy and sell side management.
OrderTracker	Manages one side of the order book (Buy or Sell). Enforces price priority.
PriceLevel	Manages all orders at a specific price point. Enforces time priority.

4.1. Price-Time Priority

The core matching algorithm adheres to the Price-Time Priority rule:

1. **Price Priority:** The OrderTracker uses a map data structure with a custom comparator. For the Buy side, the comparator prioritizes higher prices, and for the Sell side, it prioritizes lower prices. This ensures the best-priced orders are always at the front of the map.

2. **Time Priority:** The PriceLevel uses a linear data structure(doubly linked list) to store orders. New orders are added to the back. When matching, orders are consumed from the front, ensuring that the oldest order at that price level is matched first.

4.2. Matching Logic

The OrderBook::matchOrder function implements the core trade execution loop: It determines the opposite side's OrderTracker (e.g., for a BUY order, it looks at the SELL side). It sets the matching price range (minPrice and maxPrice) based on the order type and price. Then the opposite side’s order tracker is called (from OrderTracker::matchOrder), which iterates through the best-priced price levels, consuming liquidity until the incoming order's quantity is fulfilled or no more matching prices are available. Any remaining, unfulfilled Limit orders are added to the order book. Market orders that fail to match are immediately cancelled.

5. Concurrency Control

The Order Matching Engine employs a carefully considered strategy for lock selection, using different synchronization primitives in different parts of the system based on access patterns and performance requirements.

5.1. Thread Confinement (No Locks)

The primary concurrency strategy in the Order Matching Engine is thread confinement, which eliminates the need for locks.

5.2. Mutex: Protecting Shared Mutable State

It is used in the Worker class to protect the task queue that is accessed by both the scheduler thread (which submits tasks) and the worker thread (which processes tasks). Without the mutex, two threads could simultaneously try to modify the queue's internal pointers, leading to undefined behavior.

5.3. Shared Mutex (Read-Write Lock): Optimizing for Read-Heavy Access

The OrderBook::Registry it to protect access to the registry of order book instances. This is a more sophisticated synchronization primitive than a simple mutex, and it's chosen specifically because the access pattern is read-heavy. A std::shared_mutex allows multiple readers to hold the lock simultaneously, but only one writer can hold it exclusively. This is perfect for the registry because most accesses are reads (checking if an order book already exists) and only occasional accesses are writes (creating a new order book).

6. Future Enhancements

Potential future enhancements for the Order Matching Engine include:

- Integration with advanced order types (e.g., stop orders, iceberg orders).
- Support for multiple exchange protocols.
- Implementation of a low-latency market data feed.
- Enhanced monitoring and alerting capabilities.

For inquiries or support, please contact Vaasu Bisht.