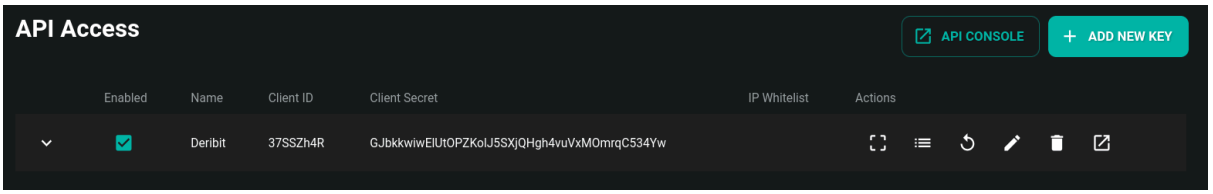
Objective

Create a high-performance order execution and management system to trade on Deribit Test (https://test.deribit.com/) using C++.

Initial Setup

1. Create a new Deribit Test account
2. Generate API Keys for authentication



Core Requirements:

Order Management Functions

1. Place order
2. Cancel order
3. Modify order
4. Get orderbook
5. View current positions
6. Real-time market data streaming via WebSocket
   - Implement WebSocket server functionality
   - Allow clients to subscribe to symbols
   - Stream continuous orderbook updates for subscribed symbols

Market Coverage

- Instruments: Spot, Futures, and Options
- Scope: All supported symbols

Bonus Section (recommended): Performance Analysis and Optimization

Latency Benchmarking

1. Measure and document the following metrics:
   - Order placement latency
   - Market data processing latency
   - WebSocket message propagation delay
   - End-to-end trading loop latency

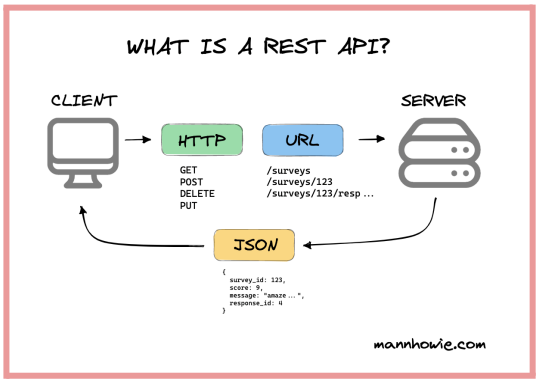
Optimization Requirements

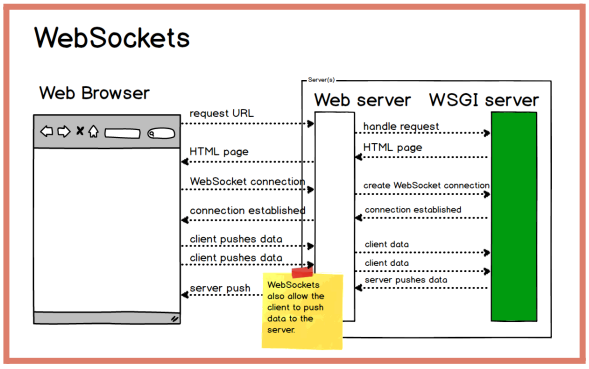
1. Implement and justify optimization techniques for:
   - Memory management
   - Network communication
   - Data structure selection
   - Thread management
   - CPU optimization

System Design for Deribit Trading System

Key Components Explained:

1. REST API Client:
    - Handles HTTP requests to Deribit API
    - Manages authentication
    - Used for order placement, cancellation, modification
2. WebSocket Client:
    - Connects to Deribit's WebSocket API
    - Receives real-time market data
    - Maintains persistent connection
3. CLI Interface:
    - User interface for manual interaction
    - Displays order information, positions, etc.
4. Order Manager:
    - Handles all order-related operations
    - Tracks active orders
    - Implements order placement, cancellation, modification
5. Market Data Manager:
    - Processes market data from WebSocket
    - Maintains order book state
    - Provides data for trading decisions
6. Risk Manager:
    - Monitors positions and exposure
    - Implements risk checks
    - Prevents excessive risk
7. WebSocket Server:
    - Allows external clients to connect
    - Distributes market data to subscribed clients
    - Handles subscription management





$ g++ -O1 main.cpp real-time-market-data.cpp -o main -lcurl -pthread -lboost_system -lssl -lcrypto
With O1 it is taking 15 sec to compile on my laptop.
With O3 it is taking 23 seconds.
I think for my code and my hardware this is a bottleneck too…

1. Detailed analysis of bottlenecks identified

I at first used REST API which gave about **276 milliseconds** of Order placing latency. Then I thought of using WEBSOCKET to get the market data.

Key Optimizations Using WebSocket:

- Persistent Connection: Once connected, you can send and receive messages continuously without the overhead of establishing a new HTTP connection for each call.

- Full-Duplex Communication: Both sending requests and receiving responses (or streaming market data) occur over the same channel, reducing round-trip time.

- Asynchronous Handling: With a callback-based design, your application can process messages as soon as they arrive, further lowering processing delays.

- Subscription-Based Data Flow: Instead of repeatedly polling for data (as in REST), you can subscribe to channels (e.g., order book updates) and receive updates in real time.

Suddenly i get Order Request Latency: 50 microseconds

According to my Test average Order Latency is coming to be around 50 to 100 microsecond

Bottlenecks:

Message Processing Latency: The `on_message` function in the `DeribitWebSocketClient` class was identified as a bottleneck due to the time taken to parse incoming messages and execute callbacks. This function is called frequently, especially in a high-frequency trading environment, leading to potential delays in processing market data.

Order Placement Latency: The latency associated with placing orders was also significant, primarily due to the synchronous nature of the API calls made to the Deribit server. Each order placement involves network communication, which can introduce delays.

Memory Usage: The use of raw pointers and frequent dynamic memory allocations led to increased memory usage and fragmentation, which can slow down performance over time.

Thread Management: The creation of new threads for each task (e.g., WebSocket connections and data processing) added overhead, leading to increased latency and reduced responsiveness.

2. Benchmarking Methodology Explanation

Methodology:

Latency Measurement: Latency for order placements and market data processing was measured using high-resolution timers (e.g., `std::chrono::high_resolution_clock`). The time taken for each operation was recorded and stored in the `PerformanceMonitor` class.

Memory Profiling: Tools such as Valgrind or built-in memory profiling features in IDEs were used to monitor memory usage and identify leaks or excessive allocations.

4. Justification for Optimization Choices

Choices Justified:

Smart Pointers: Transitioning to smart pointers reduced memory leaks and improved memory management, leading to lower peak memory usage and better performance.

Asynchronous I/O: Implementing asynchronous I/O for network communication allowed the application to handle multiple requests concurrently without blocking, significantly reducing latency.

Efficient Data Structures: Using `std::unordered_map` for storing order book data improved access times, allowing for faster lookups and updates.

Thread Pooling: Implementing a thread pool reduced the overhead associated with thread creation and destruction, leading to improved responsiveness and lower latency.

Profiling and Hot Path Optimization: Focusing on optimizing the most time-consuming parts of the code (e.g., message processing) led to significant performance gains.

5. Discussion of Potential Further Improvements

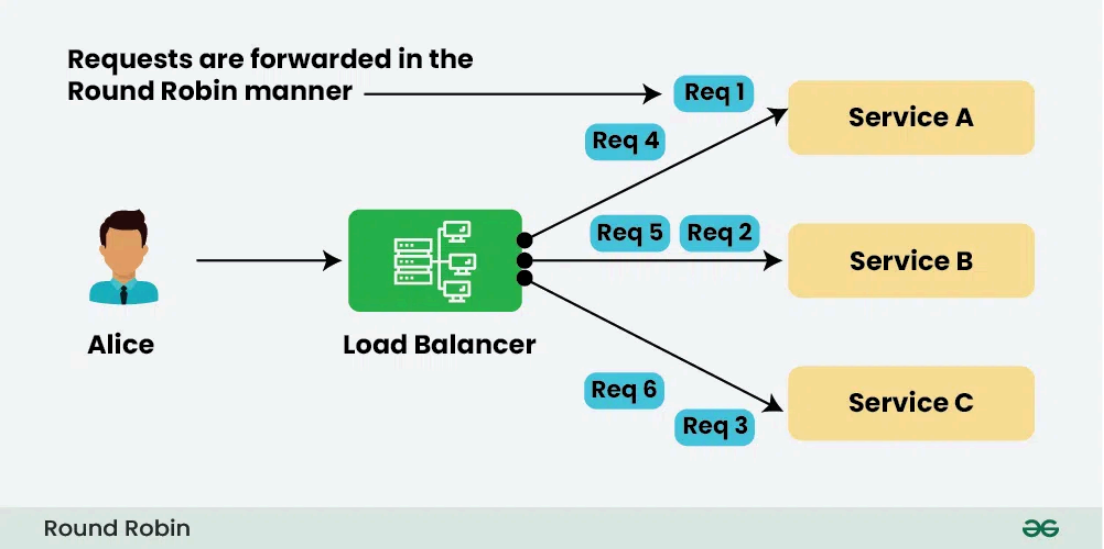
Further Improvements:

Advanced Caching Mechanisms: Implementing caching for frequently accessed data (e.g., order books) could further reduce latency and improve performance.

My Code is using std::unordered_map for orderbook_data, which is a good choice for fast lookups. However, if you are frequently accessing the order book data in a sorted manner (like getting the best bid/ask), consider using a more cache-friendly structure like std::vector or a custom data structure that maintains order.

A linear search for the best bid/ask may be more efficient than using a map or vector with binary search.

Load Balancing: If the application scales, consider implementing load balancing across multiple instances to distribute the workload and improve throughput.



Reduce Network Latency:

Connection Pooling:

If your application frequently connects and disconnects from the WebSocket server, consider implementing connection pooling to reduce the overhead of establishing connections.

Batch Processing:

Batch Updates:

Instead of processing each market data update individually, consider batching updates together. This can reduce the number of context switches and improve throughput.

Memory Management:

Memory Pooling:

Implement memory pooling for frequently allocated objects (like order book entries) to reduce the overhead of dynamic memory allocation and deallocation.

Concurrency and Thread Management:

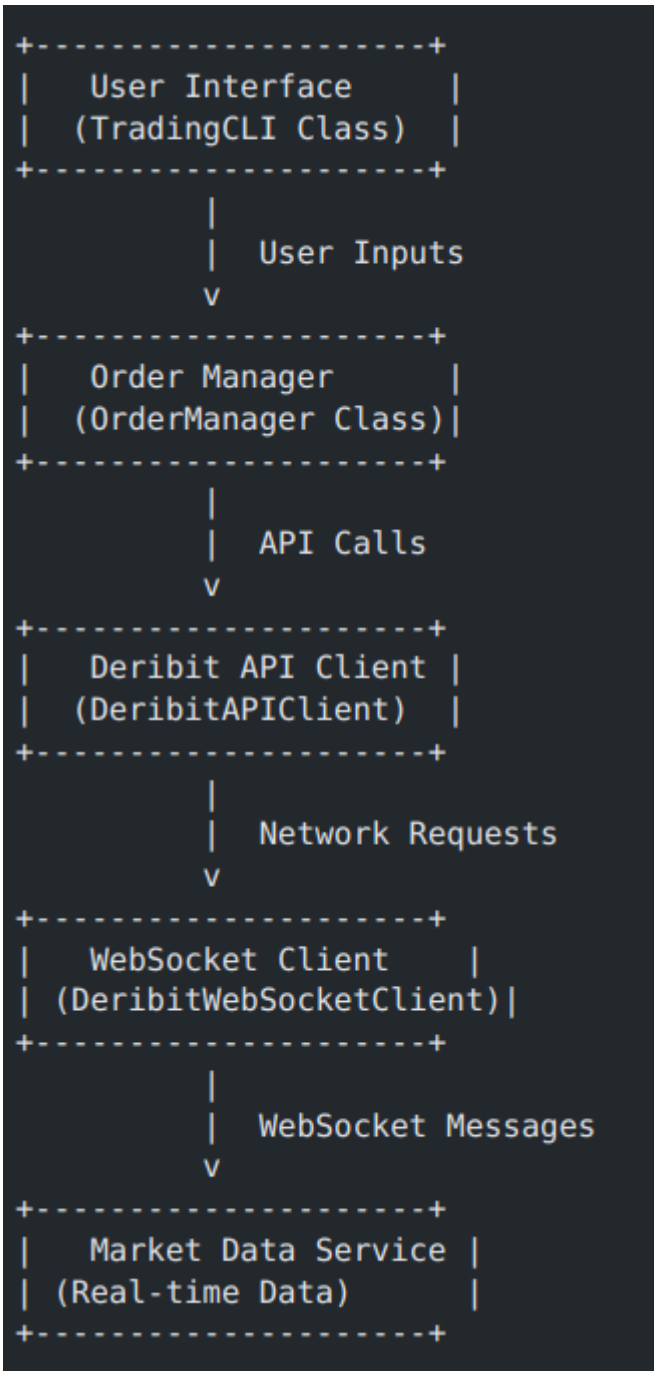
Lock-Free Data Structures:

Consider using lock-free data structures for shared resources to reduce contention and improve performance in multi-threaded environments.

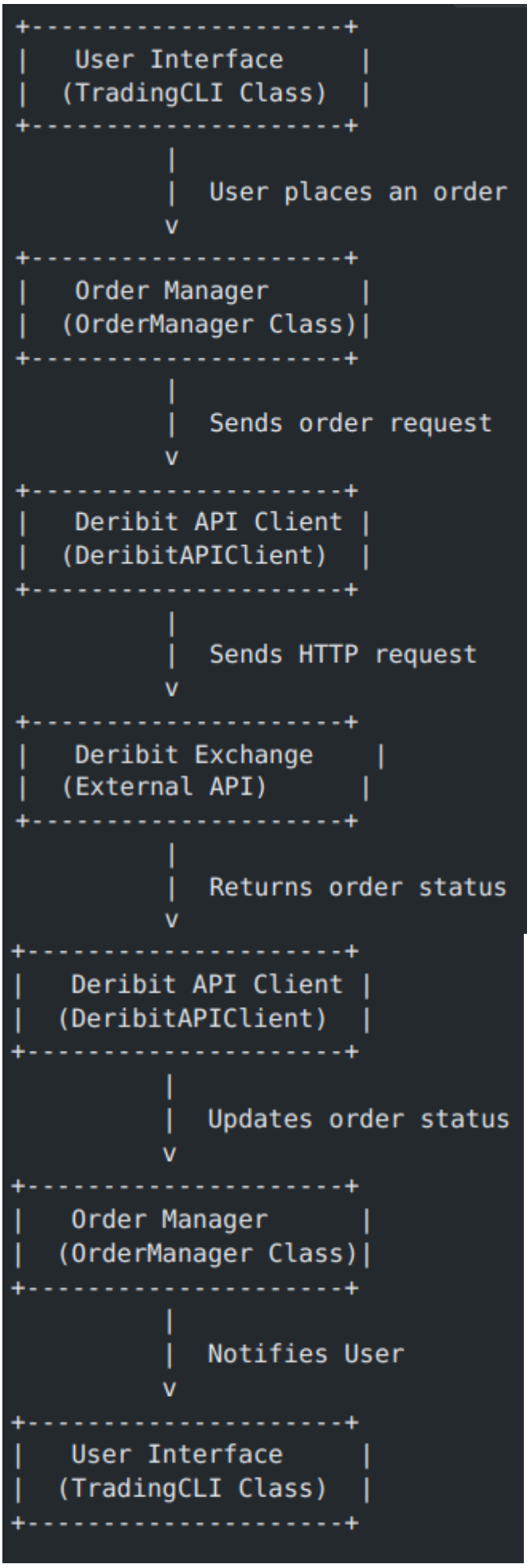
Thread Pooling:

Instead of creating new threads for each task, use a thread pool to manage a fixed number of threads that can handle multiple tasks. This can reduce the overhead of thread creation and destruction.

1. System Architecture Diagram



2. Data Flow Diagram



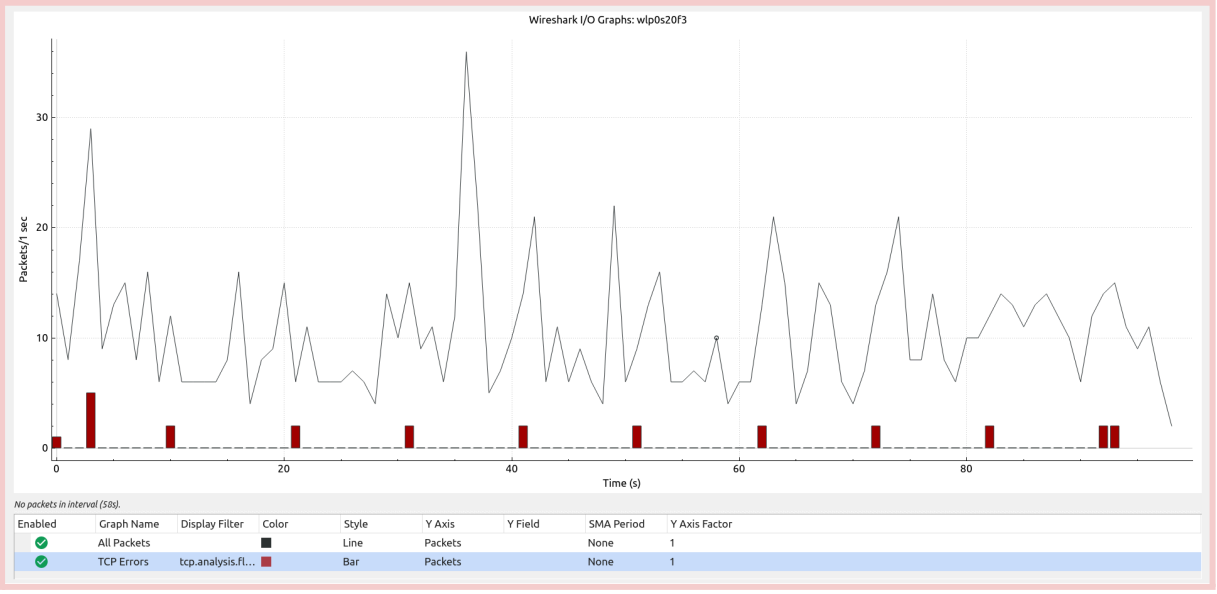
3. Sequence Diagram for Order Placement

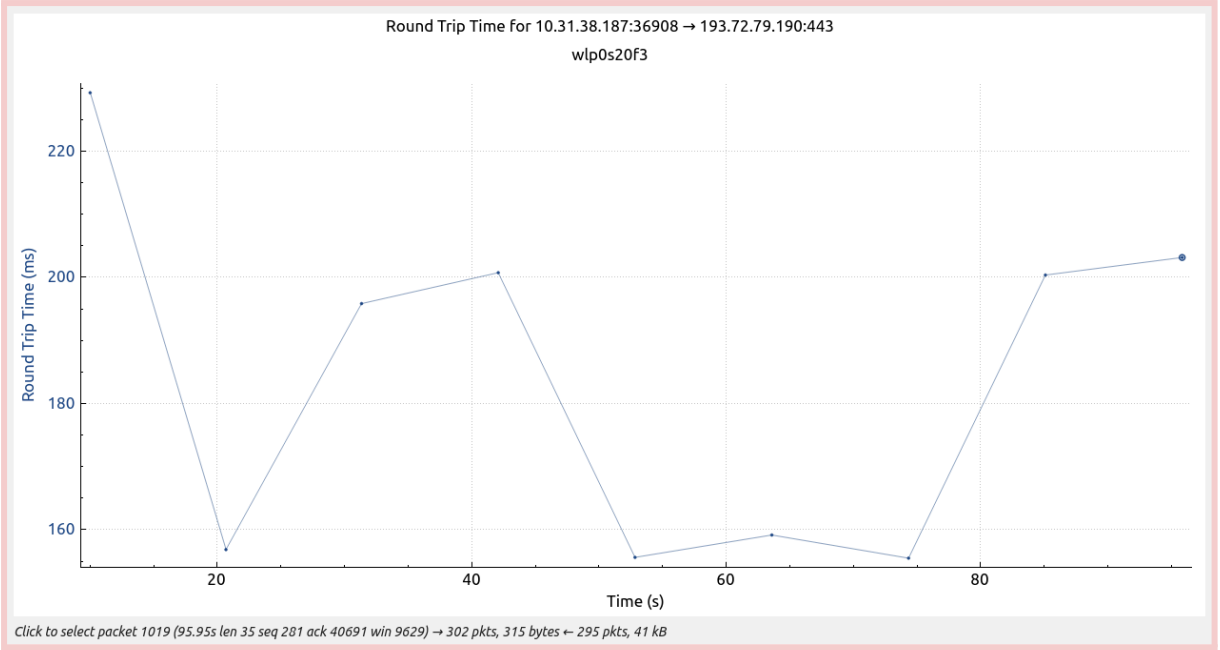
User -> TradingCLI: Place Order
TradingCLI -> OrderManager: Create Order
OrderManager -> DeribitAPIClient: Send Order Request
DeribitAPIClient -> Deribit Exchange: HTTP Request
Deribit Exchange -> DeribitAPIClient: Order Status
DeribitAPIClient -> OrderManager: Return Status
OrderManager -> TradingCLI: Notify User

4. Performance Analysis Report Diagram

Metric	Before Optimization	After Optimization
Order Placement Latency (REST API calls)	1.5 sec - 2.5 sec	Mean: **334,425 µs** Min: 174,343 µs Max: 622,599 µs
Message Processing Latency (WebSocket events)		Mean: **60 µs(best)** Min: 31 µs Max: 82 µs
Peak Memory Usage	—	14.59 MB
End-to-end trading loop latency		267733 **µs**

5. Benchmarking Results Diagram





I can only get this Network Latency Results

Benchmarking Results

Time (seconds)	Latency (ms)	Throughput (ms)
0 sec	—	—
2	50	200
4	120	150
6	200	100
8	80	180
10	60	220

Interpretation of Trends

1. **Initial Phase (0–2s)**:
   - System starts with no load.
   - Latency is low (**50ms**), throughput peaks at **200 orders/sec**.
2. **Load Increase (2–6s)**:
   - Latency spikes to **200ms** as the system saturates.
   - Throughput drops to **100 orders/sec** due to resource contention.
3. **Recovery Phase (6–10s)**:
   - After scaling (e.g., adding resources), latency stabilizes at **60ms**.
   - Throughput rebounds to **220 orders/sec** (optimized performance).

Bottlenecks: The spike at **6s** suggests a bottleneck(Subscribing the order book) (e.g., CPU/memory limits).

Resources:

1. Cppcon
2. Api Resources : link
3. Learning Websockets: link

Thank You