Documentation for Deribit Trading system

1. Benchmarking -

Benchmarked the following metrics -

- **Order Placement Latency:** Time taken from placing an order to confirmation.
- **Market Data Processing Latency:** Time taken to process incoming market data updates.
- **WebSocket Message Propagation Delay:** Time for messages to travel between the server and client.
- **End-to-End Trading Loop Latency:** Total time from market data input to order execution.

Methodology used -

1. Order Placement Latency

- **Function:** `BenchmarkPlaceOrder`
- **Process:**
  - Redirects standard input/output to simulate user input for placing an order.
  - Calls `trading_system.placeOrder()` in a loop to measure the average latency of order placement.
- **Purpose:** Evaluate the time it takes for the system to process and confirm an order.

2. Market Data Processing Latency

- **Function:** `BenchmarkMarketData`
- **Process:**
  - Simulates market data subscription using mocked input.
  - Measures the time taken to fetch and process market data for the subscribed channel.
  - Unsubscribes after the benchmark.
- **Purpose:** Identify delays in receiving and processing market data from the exchange.

3. WebSocket Message Propagation Delay

- **Function:** `BenchmarkWebSocketPropagation`
- **Process:**
  - Sends a test message (`public/test`) over WebSocket and measures the time to receive a response.
- **Purpose:** Quantify the latency introduced by WebSocket communication, including transmission and server-side processing.

4. End-to-End Trading Loop Latency

- **Function:** `BenchmarkFullTradingLoop`
- **Process:**
    - Simulates a complete trading loop:
        1. Places a buy order.
        2. Immediately places a sell order in the same iteration.
    - Redirects I/O streams to simulate user inputs for each action.
- **Purpose:** Measure the overall latency from decision-making to order placement and execution.

Results of Benchmarking:

Before Optimization (memory optimization)-

```
Run on (8 X 2095.99 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x4)
  L1 Instruction 64 KiB (x4)
  L2 Unified 512 KiB (x4)
  L3 Unified 4096 KiB (x1)
Load Average: 0.03, 0.04, 0.08
***WARNING*** Library was built as DEBUG. Timings may be affected.
-------------------------------------------------------------------------
Benchmark                               Time             CPU   Iterations
-------------------------------------------------------------------------
BenchmarkPlaceOrder             559718923 ns       657060 ns           10
BenchmarkMarketData             277963331 ns       237530 ns           10
BenchmarkWebSocketPropagation   290554859 ns       115650 ns           10
BenchmarkFullTradingLoop       1176723191 ns      1434330 ns           10
```

After Optimization-

```
Run on (8 X 2095.99 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x4)
  L1 Instruction 64 KiB (x4)
  L2 Unified 512 KiB (x4)
  L3 Unified 4096 KiB (x1)
Load Average: 0.01, 0.03, 0.07
***WARNING*** Library was built as DEBUG. Timings may be affected.
-------------------------------------------------------------------------
Benchmark                               Time             CPU   Iterations
-------------------------------------------------------------------------
BenchmarkPlaceOrder             274594745 ns       323780 ns           10
BenchmarkMarketData              48360925 ns       155782 ns          100
BenchmarkWebSocketPropagation   274128875 ns       115630 ns           10
BenchmarkFullTradingLoop        549760876 ns       556130 ns           10
```

2. Bottlenecks identified -

Thread Management Overhead:

- Creating and managing dedicated threads for each WebSocket connection may introduce significant overhead.
- This could lead to scalability issues as the number of connections grows, resulting in high CPU usage and potential memory exhaustion.

Memory Usage and Management:

- Storing all subscriptions in a centralized data structure with a fixed length could potentially lead to inefficient memory usage if the number of subscriptions exceeds the predefined limit.
- Memory fragmentation and potential memory leaks may occur over time if memory management is not handled carefully.

Market Data Processing Delays:

- While market data processing is handled in a separate thread, depending on the complexity of the data, processing delays can still impact overall performance.
- A large volume of data or slow processing logic could become a bottleneck, affecting real-time responsiveness.

Single WebSocket Connection Per Thread:

- Dedication of one thread per WebSocket connection can lead to inefficiency, particularly in scenarios with a large number of concurrent connections.
- Managing threads for each connection can lead to increased context switching and poor resource utilization.

WebSocket Data Handling Overhead:

- Even though WebSockets reduce the TCP handshake overhead, frequent data exchanges with high-frequency messages could lead to significant memory and CPU load if not managed efficiently.
- Managing large volumes of messages in real time without introducing delays or buffering issues remains a challenge.

3. Optimization Study -

These were the practices implemented for optimising the overall latency of the system and optimise the memory used

Centralized Data Structure with Fixed Length for Subscriptions:

- Using a fixed-length data structure to store subscriptions allows for fast retrieval.
- The approach avoids dynamic memory allocation, leading to improved access time and reduced overhead.

Use of `const` in Functions:

- The `const` qualifier is used in function parameters and return types to indicate that the data should not be modified.
- This practice improves code safety, enables compiler optimizations, and clarifies the intent to avoid unintended side effects.

Asynchronous Network Operations with Boost.Asio:

- Boost.Asio is used for asynchronous network operations, allowing the application to continue other tasks while waiting for network responses.
- This increases responsiveness and scalability by performing non-blocking I/O operations.

Dedicated Threads for WebSocket Connections:

- WebSocket connections are handled by dedicated threads to ensure parallel processing.
- This prevents blocking and ensures that each connection operates independently, enhancing concurrency.

Separate Thread for Processing Market Data:

- Processing of market data is isolated in a dedicated thread to avoid interference with other tasks (e.g., network communication).
- This separation ensures optimized data handling without impacting the responsiveness of other operations.

WebSockets to Reduce TCP Handshake Overhead:

- WebSockets eliminate the need for a TCP handshake on each request by maintaining a persistent connection.
- This reduces latency and communication overhead, making the system more efficient for real-time data transfer.

4. Future Improvements -

- **Thread Pool Implementation**: Instead of dedicating a separate thread for each WebSocket connection, implementing a thread pool can optimize resource utilization. This approach allows for a fixed number of threads to handle multiple connections, reducing the overhead associated with thread creation and destruction.
- **Connection Pooling**: Implementing connection pooling can further reduce the overhead of establishing new connections. By reusing existing connections, the application can handle requests more efficiently, especially in scenarios with high connection turnover.
- **Load Balancing**: Introducing load balancing mechanisms can distribute incoming connections across multiple servers or processes, enhancing scalability and fault tolerance. This strategy ensures that no single server becomes a performance bottleneck.