The Backbone of Modern Finance Part II:

Laying The Foundation For A High Performance Financial Exchange Simulator Sandbox – MiniExchange

Peter Torpis
ptorpis@gmail.com

November 6, 2025

Abstract

This document showcases the design of a simplified financial exchange simulator built from scratch in C++. It builds on the prototype outlined in Part I, now with more features and higher performance. These features include asynchronous epoll-based networking with a custom binary message protocol, support for basic market operations, like placing, cancelling, and modifying orders, a high-speed matching engine, client session state tracking, a terminal-based client interface and a configurable client running harness to produce traffic loads, asynchronous event logging and timing instrumentation and more. In basic testing the system achieved 1-2μs internal message processing latencies at over 200,000 requests per second with 1,000 concurrent clients. This system aims to provide a sandbox environment for testing and researching market microstructure dynamics and simulating market behavior scenarios, which is what future work is going to focus on. This paper aims to document the main architectural and design lessons learned from this project.

Contents

List of Figures

List of Tables

Listings

1 Architecture

1.1 Architecture Overview

The overall philosophy of the design is that layers have authority over the layers below them and no layer has authority over the one above it. Each layer fills a different role, each with a set list of responsibilities, no two layers are responsible for achieving the same function. Each operation that needs to be performed is relatively simple, so the main flow of execution happens on a single thread.

Why single-threaded? Multi-threading the matching engine would require synchronization mechanisms that compromise either latency (locks) or complexity (lock-free structures). More critically, FIFO matching engines must maintain strict order determinism for regulatory compliance, auditability, and predictability. A single-threaded design guarantees these properties trivially while achieving low single-digit-microsecond internal latency.

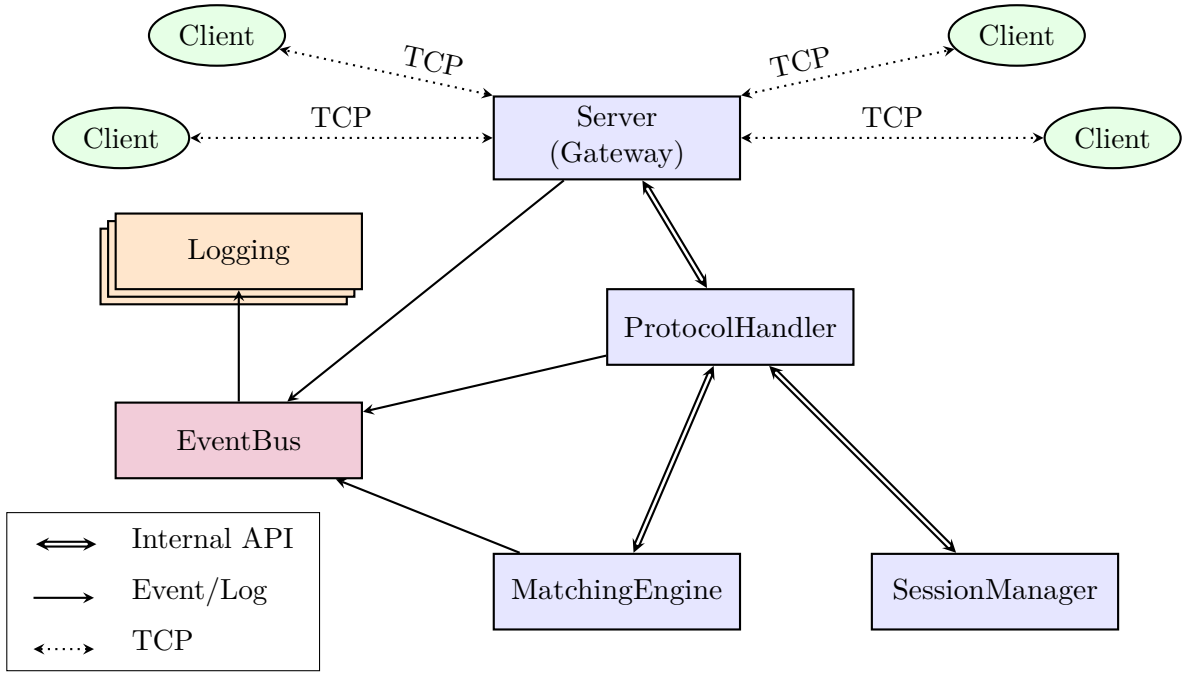

Figure 1: System Architecture Diagram

1.2 Component Responsibilities

Server (Gateway)

The `Server` component acts as the network gateway, managing all TCP connections using an epoll-based event loop. Its responsibilities are accepting new client connections, managing the lifecycle of sockets by creating, monitoring, and cleanly closing them. It handles network read and write operations from per-client data buffers, it detects inactive clients and lastly, it notifies the `ProtocolHandler` when a message is ready to be processed.

Critically, the Server operates entirely on raw bytes – it has no knowledge of message types, order matching, or business logic. This separation allows the network layer to be optimized independently (e.g., switching to io_uring in the future) without affecting application logic.

Protocol Handler

The `ProtocolHandler` sits between the network and business logic layers, responsible for message framing and deserialization from the binary protocol (endianness conversion), validating and routing the messages to appropriate components, serializing the responses and maintaining message sequence numbers per client.

The handler maintains no persistent state – it's purely a transformation layer. This makes it easy to reason about and test: given input bytes and current system state, it deterministically produces output bytes.

Session Manager

The `SessionManager` tracks client state and connection metadata, maps file descriptors to client sessions, maintains client authentication state, tracks activity for timeout detection, stores client-specific data buffers, and assigns server-side clientIDs.

Unlike the Server, which only knows about sockets, the SessionManager understands client identity and lifecycle.

Matching Engine

The `MatchingEngine` contains the core business logic. Its job is to maintain the order book, execute order matching with price-time priority and to do so deterministically, generate trade events, manage the orders' lifecycle, and enforce trading rules (cancelling unfilled market orders, self-trade prevention, etc.).

The engine operates on validated `OrderRequest` objects, making it completely isolated from network concerns and transport layer agnostic.

Event Bus

The `EventBus` provides asynchronous, non-blocking event logging by capturing system events, allowing decoupling between event generation and event processing, and enable multiple subscribers to listen for specific event types.

This design ensures logging overhead doesn't impact latency measurements or matching performance.

2 Implementation Details

The following chapter showcases some of the implementation details of the main components.The entire source code base can be found on the project's GitHub repository.

The C++ design patterns paper by Bilokon & Gunduz [1] describes a lot of the concepts that were applied or considered while building this project, most notably compile-time dispatch, branch reduction and cache awareness. It's a great resource for anyone looking to know more about high performance design.

2.1 Network Layer

Protocol Design

A large portion of network messaging happens through text-based protocols. In high-performance systems, the overhead of string manipulation heavily outweighs the usability benefits a text-based protocol offers. For this exchange a custom binary messaging protocol was designed. In the prototype a JSON based protocol was implemented, which quickly became a bottleneck in the C++ version.

The main goal was to reduce message parsing overhead as much as possible. To do this, the message fields and lengths are fixed for every type. Here is an example format, here is the layout for `NEW_ORDER` messages:

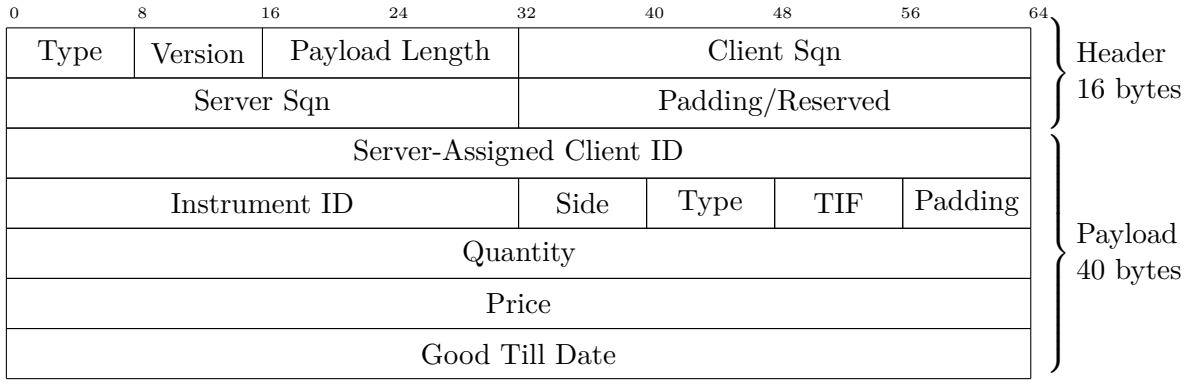

Figure 2: `NEW_ORDER` message format (56 bytes, 8-byte aligned) [2]

The data is aligned to 8 bytes in all cases to allow for efficient data fetching [2]. The protocol uses `#pragma pack(1)` to eliminate compiler-inserted padding, which is crucial to guarantee alignment and consistency by having explicit control over memory layout [3].

To keep consistency across machines, the protocol requires the fields to be in big-endian byte order for fields that are larger than 1 byte, with arrays being treated as series of 1 byte characters. The protocol handler was written in a way so that the messages get serialized in the correct host-endian representation, regardless of which one it uses.

The C standard library provides functions to make this conversion [4], but there is no standard function for 64-bit types. The C++23 standard introduced `std::byteswap`, which is perfect for this use case when paired with `std::endian::native` to convert any integral type [5].

```cpp
template <typename T> inline T swapEndian(T val) {
    if constexpr (std::endian::native == std::endian::little) {
        return std::byteswap(val);
    }
    return val;
}
```

Listing 1: Conversion to host-endian from network byte order

This template-based approach works for all integer types and compiles to a single CPU instruction (`bswap`) on x86-64, making it zero-overhead A.1.

Message Types

The protocol defines a list of messages types that are separated into two parts: messages sent by clients and messages sent by the server. Generally, all requests are acknowledge, so (almost) all client message types have a corresponding "ack" (acknowledgement). There are certain rules to when messages are ack-ed and when are they ignored. The complete set of rules can be found in the specification of the protocol, found in the `docs/` in the project's repository A.4.

Main client message types are: `HELLO`, `LOGOUT`, `HEARTBEAT`, `NEW_ORDER`, `CANCEL_ORDER`, and `MODIFY_ORDER`.

Server-side messages are the corresponding "ack"s to these, with the exception of `HEARTBEAT`, which is not an ack-ed type. In addition, there are the trade notification messages (`TRADE`) that are sent by the server.

Epoll-based Networking

The server uses a single-threaded even loop based on Linux epoll [6] in edge triggered mode (`EPOLLET`), to reduce overhead relative to level-triggered mode. This requires read events to completely drain the socket to avoid hanging data.

Scaling to multiple instruments Production exchanges handle thousands of instruments using thread-per-instrument parallelism. A shared I/O thread (gateway) handles network operations and routes orders to dedicated matching threads based on `InstrumentID`. Since order books are naturally independent, this scales linearly with core count without synchronization overhead.

The protocol and architecture support this extension: `NEW_ORDER` messages already include an `InstrumentID` field, and the modular design separates network and matching concerns. Migration to a multi-instrument architecture would involve extracting the matching engine into worker threads while maintaining the single-threaded epoll event loop for I/O.

Handling requests: Upon reading from a socket, the request is immediately executed by the matching engine 2.2 or the `SessionManager` 2.3. Any responses that may be generated are then buffered into the respective client(s) send data buffers, and a write event is scheduled for them.

When the system processes a request, it generates a response message (for messages that

require responses). There can be cases when a request needs a outgoing message in response to more than one client, for example upon an order request that fills an order, the trade notification has to go out to both (or any number of) parties. When the system returns the result from an operation, it fills a pre-allocated vector of all the file descriptors that have messages ready to be sent.

An example flor for an incoming order request:

1. Client A and client B have resting sell orders active in the market
2. Client C sends a buy order request that fills both A and B's orders
3. The server reads from C's socket into C's read data buffer
4. The server dispatches the message to the `ProtocolHandler` with C's file descriptor
5. The `ProtocolHandler` looks up the session based on this file descriptor and reads the request from the session's read data buffer
6. The `ProtocolHandler` processes the message, dispatches it to the engine
7. The engine fills A's and B's orders and returns the result of this operation to the `ProtocolHandler`
8. A responses are generated for all 3, an `ORDER_ACK` and a `TRADE` for C, and `TRADE`s for both A and B, messages are written into the respective sessions' send data buffers
9. The handler fills the pre-allocated vector with all 3 clients' file descriptors
10. The server schedules write events for all 3, then on the next iteration, sends the messages out.

Connection Handling: New connections are accepted in a loop until no more are pending (`EAGAIN`). Each socket is configured with `TCP_NODELAY` [7] to disable Nagle's Algorithm [8], eliminating the latency penalty at the cost of slightly more packets and more TCP overhead relative to "useful" data being sent over the network.

Example Network Traffic

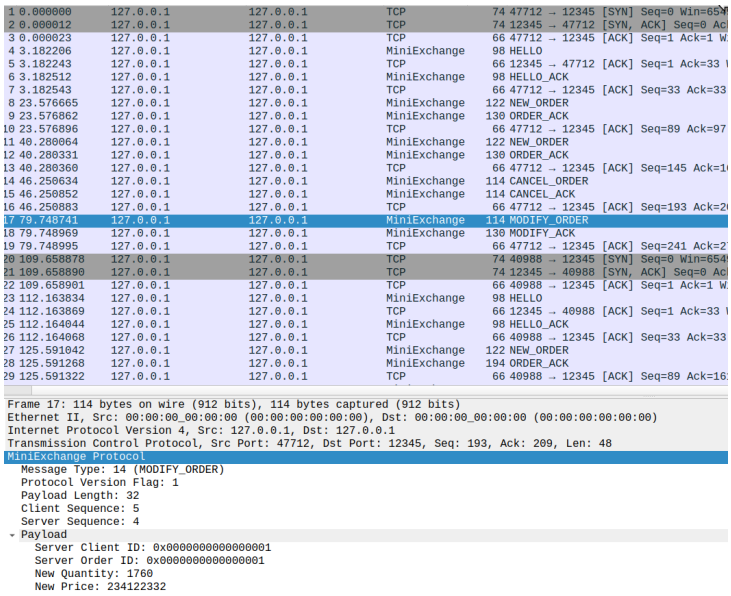

Figure 3: Wireshark packet capture showing exchange protocol with Lua dissector plugin annotations A.4

2.2 Matching Engine and the Order Book

Data Structures

The matching engine is the core part to any exchange. It holds all the currently active orders – sometimes referred to as "resting orders" – in the central data structure called the order book, and it performs operations on the order book based on the matching policy. It is essential to choose the right data structures for this part, as the manipulation of it is the most common execution paths in the system.

The order book must hold all orders in memory, while keeping price-time priority and be able to remove any arbitrary order, all while doing so efficiently.

A first thought of anyone implementing this would probably be to use `std::map` to keep the price levels ordered. Naturally, 2 maps are used, one for bids and the other for asks, the only difference being the comparator passed in to keep highest bid and lowest ask as the first level. The price can be used as the key and the values are the price levels.

Within the price levels, time-price priority must be maintained, also, after accessing the first order in the queue, accessing the one that is next in line is common, because if a resting order is filled and there is still some unmatched quantity left in the incoming order, the order next in line should start matching right away. For this reason it is important to also keep spacial locality in mind. One might think to use `std::vector`, it allows us to efficiently add new elements to the end of the queue (amortized constant time), but a very common operation is to pop the first element – eg. when an resting order is filled – which would cause all the other elements in the vector to shift, which is done in linear time. A better container for this use case is `std::deque`, because it offers similar benefits of spacial locality – storing its elements in contiguous blocks

(unlike `std::list`) – while also allowing for pushing new elements to the back and popping them from the front in constant time.

There is one more requirement needed to be met, and that is to be able to remove any order from any arbitrary position in the order book. To achieve this an unordered map is kept that maps the orders' unique IDs to a pointer to that order. When an order is cancelled, it first looks in this map to find the price (accesses the price field through the pointer), then uses the price together with the ID to find it in the queue. This way only one hash-map lookup, an ordered map lookup, and a traversal of one price level is needed to remove any order.

```
    using OrderQueue = std::deque<std::unique_ptr<Order>>;
    using Price = int64;

    std::map<Price, OrderQueue, std::less<Price>> asks;
    std::map<Price, OrderQueue, std::greater<Price>> bids;
    std::unordered_map<OrderID, Order*> orderMap;
```

Listing 2: Order book data structure organization

Memory Ownership Strategy

The implementation deliberately uses raw pointers in `orderMap` while `OrderQueue` holds `unique_ptr`s. This breaks conventional C++ wisdom that "raw pointers shouldn't coexist with smart pointers," but the trade-off is worthwhile:

- **Memory savings**: Using `shared_ptr` would add overhead in two places: (1) 16 bytes per `shared_ptr` instance in the map (8-byte object pointer + 8-byte control block pointer), and (2) 24 bytes for the control block itself (reference counts and virtual table). In contrast, a raw pointer is just 8 bytes with no control block. This 32-byte-per-order difference is a significant saving in a system designed to handle large order books.
- **Performance**: No atomic reference counting operations on the hot path
- **Safety**: Maintained through a simple invariant – the queue owns the order, and `orderMap` is updated atomically with queue modifications, so the queue is exclusively responsible for managing the order, the `orderMap` can be thought of as a non-owning view.

Matching Algorithm

The matching engine implements a policy-based design with compile-time dispatch. Rather than checking order side and type with runtime conditionals multiple times on every match, the engine uses a dispatch table initialized at construction:

```
using MatchFn = MatchResult (MatchingEngine::*)(std::unique_ptr<Order>);
MatchFn dispatchTable_[2][2];
dispatchTable_[0][0] = &MatchingEngine::matchOrder_<BuySide, LimitOrderPolicy>;
dispatchTable_[0][1] = &MatchingEngine::matchOrder_<BuySide, MarketOrderPolicy>;
dispatchTable_[1][0] = &MatchingEngine::matchOrder_<SellSide, LimitOrderPolicy>;
dispatchTable_[1][1] = &MatchingEngine::matchOrder_<SellSide, MarketOrderPolicy>;
```

Listing 3: Dispatch table for compile-time policy selection

At runtime, when an order comes in, the only time the side and the type is checked is before entering the specialized matching function:

```
MatchResult MatchingEngine::processOrder(const OrderRequest& req) {
    std::unique_ptr<Order> order = service_.orderFromRequest(req);
    int sideIdx = (order->side == OrderSide::BUY ? 0 : 1);
    int typeIdx = (order->type == OrderType::LIMIT ? 0 : 1);

    return (this->*dispatchTable_[sideIdx][typeIdx])(std::move(order));
};
```

Listing 4: Runtime checks for incoming orders

This approach eliminates repeated branching in the hot path. Each template instantiation generates specialized code for its specific scenario, allowing the compiler to optimize away unnecessary checks. For example, `MarketOrderPolicy` has `needsPriceCheck = false`, which causes the price comparison to be completely eliminated via `if constexpr`:

```
if constexpr (OrderTypePolicy::needsPriceCheck) {
    if (!SidePolicy::pricePasses(order->price, bestPrice)) {
        break;
    }
}
```

Listing 5: Compile-time conditional eliminates unnecessary checks

For market orders, this branch doesn't exist in the compiled code – the template instantiation simply omits it.

Core Matching Loop The matching algorithm follows a straightforward greedy approach:

1. Retrieve the best price level from the opposite side of the book
2. For limit orders, verify the incoming order's price can match at this level
3. Iterate through the price level's queue in FIFO order
4. For each resting order, match the maximum possible quantity
5. Update both orders' quantities and statuses
6. Remove fully filled orders from the book
7. Continue until the incoming order is filled or no more matches are possible
8. Residual unfilled quantity for a limit order is added to the book (unfilled market orders are not added, the unfilled quantity is considered cancelled according to the policy of the matching engine)

Self-Trade Prevention The engine includes self-trade prevention – orders from the same client cannot match against each other. While this adds a branch in the inner loop, it's necessary for correctness in real exchanges where market makers post both bids and asks.

Policy-Based Design Benefits The policy-based approach provides several advantages:

- **Zero-cost abstraction**: Template instantiation happens at compile time with no runtime overhead
- **Code reuse**: The core matching logic is shared across all four order type combinations
- **Extensibility**: New order types (e.g., stop orders, iceberg orders) can be added by implementing new policy classes
- **Type safety**: Mismatched policies are caught at compile time

Side policies (`BuySide`, `SellSide`) encapsulate the differences between bid and ask matching:

```
struct BuySide {
    static auto& book(MatchingEngine& eng) { return eng.asks_; }
    static bool pricePasses(Price orderPrice, Price bestPrice) {
        return orderPrice >= bestPrice;  // Buy order must meet ask price
    }
    // Trade event construction...
};

struct SellSide {
    static auto& book(MatchingEngine& eng) { return eng.bids_; }
    static bool pricePasses(Price orderPrice, Price bestPrice) {
        return orderPrice <= bestPrice;  // Sell order must meet bid price
    }
    // Trade event construction...
};
```

Listing 6: Side policy example

Order type policies control matching behavior and finalization:

```
struct LimitOrderPolicy {
    static constexpr bool needsPriceCheck = true;
    static statusCodes::OrderStatus finalize(
        std::unique_ptr<Order> order, Qty remaining,
        Qty original, MatchingEngine& eng) {
        if (remaining > 0) {
            order->qty = remaining;
            eng.addToBook_(std::move(order));
        }
        return /* appropriate status */;
    }
};
struct MarketOrderPolicy {
    static constexpr bool needsPriceCheck = false;
    static statusCodes::OrderStatus finalize(/*...*/) {
        // Market orders never rest in the book
        // Unfilled quantity is cancelled
        return /* appropriate status */;
    }
};
```

Listing 7: Order type policy interface

This design takes advantage of modern C++'s ability to achieve runtime performance equivalent to hand-rolled specialized functions while maintaining clean, maintainable code. The template mechanism serves as a code generator, producing optimal machine code for each scenario without manual duplication.

2.3 Client Session State Management

Overview

The exchange has to keep track of a few pieces of information for every client that is connected. This was achieved by keeping a `Session` object for each client, which stores their file descriptor of their connection, their server assigned ID, a flag to indicate if they are authenticated or not, their network receiving and sending data buffers, and a trade execution counter.

To avoid the network and application protocol layer devolving into chaos, the only class that is allowed to manipulate these `Session` objects is the `SessionManager`.

The `SessionManager` keeps a record of all clients and their sessions, and also does extra bookkeeping to be able to detect inactive connections. If a client suddenly crashes or its internal state otherwise becomes corrupted without the TCP connection terminated, there needs to be a mechanism in place to detect such clients and remove them gracefully.

The exchange protocol supports keepalive messages, which are called "heartbeats", and the server can require connecting clients to send these short messages at an agreed interval, and if the client fails to do so, they are assumed to be dead and their session is terminated and they are disconnected from the server.

Checking the most recent heartbeats for all connected sessions is an iteration that checks the last heartbeat timestamp and if the time between the current time and the timestamp exceeds a certain limit, the client gets marked as inactive, then they get disconnected as described above. Since this is an iteration that happens frequently, storing data about the clients' activity is best done in cache friendly data structures, so neither `std::list` and keeping a "last active timestamp" field in the `Session` that is then stored in some map are ideal.

How this system solves this is by keeping a vector of a simple `Heartbeat` struct:

```
1     struct Heartbeat {
2         int fd;
3         std::chrono::steady_clock::time_point lastHeartbeat;
4     };
5
6     std::vector<Heartbeat> heartbeats;
```

Listing 8: Heartbeats are stored together with the file descriptor of the client's connection

This approach assumes that the heartbeats are checked fairly regularly, and they are touched more frequently than other fields of the `Session`, and that most of the time, there are not going to be many inactive timestamps.

What happens when a timestamp is found to be inactive? The naive solution would be to just call `std::vector::erase` to get rid of that entry, but that starts to run into issues with

there are more than just a few entries. This is because `std::vector` maintains strict contiguity with its elements, so erasing an element that is not at the end must involve shifting all elements after it one place over. This is a similar problem to storing the orders in the order book 2.2, the difference here is that the order of the elements is not important.

A common trick to solve this limitation is to swap the element with the last one, and then pop the last element (now, the inactive one), which always happens at constant time. This works, because the entries do not have to be in any particular order.

Large exchanges that may have hundreds of thousands of active sessions usually also keep the heartbeats in buckets, instead of just one vector, and staggering checks to avoid causing spikes in latency.

A downside to this approach is there must be extra bookkeeping done to ensure that no two sessions' data get switched up or lost.

```
1     std::unordered_map<int, Session> sessions;      // FD -> Session
2     std::unordered_map<ClientID, int> clientIDToFD; // ClientID -> FD
3     std::vector<Heartbeat> heartbeats;              // all heartbeats
4     std::unordered_map<int, size_t> fdToHbIndex;    // FD -> heartbeat index
```

Listing 9: Data structures maintained by `SessionManager`

The `SessionManager` maintains records in these 4 structures. This may seem a lot at first, but notice that the only one that is relatively large in size is the `Sessions` map. The rest are mapping client IDs to file descriptors, and file descriptors to their index in the `heartbeats` vector. This is also the reason that any modification or manipulation to the sessions is done through the `SessionManager`, so as to not lose track of which structure should be updated in what way. There is an API exposed from it, for functions like `removeSession(int fd)` that cleanly take care of that operation.

2.4 Logging and Timing Instrumentation

Events inside the system are handled in a publisher-subscriber model. There is an event bus that connects all the components that would like to publish events and the components that want to perform actions based on those events. This model allows the publishers and subscribers to be decoupled, no publisher cares about what happens to the events once they have been passed to the event bus, and conversely, the subscribers do not need to know how these events were produced.

In a high performance system it is important for the logging to not block other actions, as they are not part of the main functionality, but it is important that the necessary events are recorded for auditability and replay purposes.

Different components produce different types of events, these events are currently being recorded into separate CSV files (more sophisticated database solution might be implemented in the future).

Since writing to disk is orders of magnitude slower than referencing main memory [1], the event bus only blocks the main thread until the data for the event is copied into the buffer of the logger, then the writer thread is woken up with a condition variable notification, and the

execution is handed back to the main thread, while letting the writer thread to record the event to disk in the background.

Instead of using C++'s usual file handling API (`std::fstream`), the loggers use C's faster `fprintf`. In the case that the main thread is outpacing the logger, there is overflow protection built into the logic with modulo-wrapping in a ring buffer, in the case that this happens, events are dropped until the writer can catch up.

2.5 Order Modifications

According to the policy of the matching engine, orders may be modified, but several rules determine how modifications are handled. These rules affect queue position, priority, and whether an order is updated in place or replaced entirely.

Two fields may change when modifying an order: quantity and price. The direction and combination of these changes determine the engine's behavior:

- **Quantity reduction at the same price:** The order is modified in-place and retains its priority within the price level.
- **Any price change or quantity increase:** The original order is cancelled and a new order is created. The modified order is therefore placed at the back of the queue for its new price level. This behavior is standard in real exchanges to prevent users from "jumping the queue" with misleading orders under a price-time priority rule set.

2.6 Reflection in C++

Reflection is a language feature that allows programs to introspect their own structure at runtime. Many high-level languages provide built-in reflection capabilities. For example, Python's `getattr()` and `__dict__` allow accessing object fields by name:

```python
class Order:
    def __init__(self):
        self.order_id = 123
        self.quantity = 100

order = Order()
print(getattr(order, "order_id"))

for field_name, value in order.__dict__.items():
    print(f"{field_name}: {value}")

# Output:
# 123
# order_id: 123
# quantity: 100
```

Listing 10: Python's built-in reflection

C++ lacks native reflection in the standard library (though C++26 may introduce it via `std::meta`). For this project, field name introspection was necessary for logging and CSV

serialization – each event type needed to write its fields as named columns.

Manual Reflection via Template Functions

Rather than using a heavy reflection library or code generation tool, I implemented a lightweight solution using template member functions:

```
struct ReceiveMessageEvent {
    int fd;
    ClientID clientID;
    uint8_t type;
    uint32_t ref;

    template <typename F> void iterateElements(F&& func) const {
        func("fd", fd);
        func("clientID", clientID);
        func("type", type);
        func("ref", ref);
    }
};
```

Listing 11: Manual reflection through visitor pattern

The `iterateElements()` template accepts any callable object and invokes it for each field with both the field name and value. This enables generic operations:

```
template <typename EventT> struct ServerEvent {
    uint64_t timestamp;
    EventT event;
};

// Then to write a header for a CSV log file:
    // ...
    fprintf(f, "ts");
    EventT dummy{};
    dummy.iterateElements([&](const char* name, auto&) { fprintf(f, ",%s", name); });
    fprintf(f, "\n");
    // ...
```

Listing 12: Using reflection for CSV serialization

This pattern provides reflection-like capabilities without runtime overhead – the template is instantiated at compile time, and the compiler can inline the lambda, generating code equivalent to manually writing each field. The trade-off is that each struct must manually list its fields in `iterateElements()`, but this is a one-time cost that maintains type safety and zero-cost abstraction.

There are external libraries that support reflection features, for this project, however, this simple manual solution was enough, and worked without adding dependencies.

3 Performance Analysis

There are a few limitations to the load testing that was performed on the system so far, it was only tested with a physical ethernet connection on the same subnet, sustained load tests were not performed, the maximum throughput achieved was ultimately capped by the client side application, not the server.

The following results are still encouraging, achieving sub-microsecond internal latency and end-to-end latency of around 20-30 microseconds without aggressive optimizations, like fully zero-copy design or kernel bypass networking.

3.1 Testing Environment

Server Configuration

- **Hardware**: Dell Latitude 7490 laptop
- **CPU**: Intel Core i5-8350U (4 cores, 8 threads) @ 3.6 GHz
- **RAM**: 16 GB DDR4
- **Network**: Gigabit Ethernet (Intel I219-LM)
- **OS**: Ubuntu 24.04.3 LTS (64-bit)
- **Kernel**: Linux 6.14.0-33-generic
- **Compiler**: GCC 13.3.0

Build Configuration

- **Compiler**: GCC 13.3.0
- **C++ Standard**: C++23 (`-std=c++23`)
- **Optimization Level**: `-O3`
- **Link-Time Optimization**: Enabled (`-flto`)
- **Build Type**: Release

Client Configuration

Client machines ran on a separate laptop connected via Ethernet, operating in WSL2 (Windows Subsystem for Linux) with Ubuntu 24.04.

Network Topology

Both machines were connected to the same gigabit Ethernet switch on the same subnet (192.168.x.x), eliminating the loopback interface to provide realistic network latency measurements.

3.2 Test Setup

The test was designed to measure throughput and latency with high traffic loads. The clients were not acting intelligently, rather their actions were randomly generated based on a seeded random number generator that placed events into a priority queue and performed those actions based on the schedule. The client runner application spawned 1000 clients and connected them to the exchange, send a login message for all, waited for them to be authenticated (`HELLO_ACK` response from the server) and started scheduling events at a set interval with random jitter.

The server measured latency by using low overhead timing. The sampling rate was set at 1/10, meaning that every 10th event's latency was captured. The timestamps were taken from the CPU's internal timestamp counter, which is ideal for this use case, because it is guaranteed to be monotonic – unlike `std::chrono::system_clock`, which can be subject to small adjustments to keep in sync with real-world time – and sampling it is lower overhead than calling `std::chrono::steady_clock::now()`. At server startup this timestamp counter is calibrated and the tick rate is saved to a run metadata file, which was then later used to calculate real time intervals from the timestamps.

Tests were conducted on a 2018 Dell Latitude 7490 laptop (Intel i5-8350U), demonstrating that sub-microsecond latencies are achievable on consumer hardware through careful software design rather than specialized infrastructure.

3.3 Test Results

Timestamps are taken internally at the moment the message has been received (beginning of Stage 1), after the moment the message has been deserialized (end of Stage 1, beginning of Stage 2), and after the moment the response to the message has been buffered into the send data buffer of the respective client(s) (end of Stage 2). Total time is measured from the beginning of Stage 1 to the end of Stage 2.

The following table shows a detailed breakdown of a 60 second load test with 1000 clients running at 220-230k requests per second. The most significant findings are that both the mean and the median internal latencies were sub-microsecond, indicating very efficient message handling an order book updates. Another statistic worth taking a look at is the significant jump from the 99.99th percentile latencies to the absolute highest recorded ones, indicating that there is still some room for improvement in reducing the tail latency of the system. Linux perf counter statistics showed 2.9% branch miss rate, hinting at the fact that the branch reduction techniques are working.

Message Type	Metric	Stage 1 (µs)	Stage 2 (µs)	Total (µs)
NEW_ORDER	Mean	0.482	1.039	1.521
	Median	0.173	0.533	0.876
	Std	1.179	1.897	2.268
	Min	0.046	0.087	0.142
	Max	135.417	375.568	375.648
	p5	0.080	0.113	0.207
	p10	0.088	0.137	0.283
	p25	0.125	0.264	0.463
	p50	0.173	0.533	0.876
	p75	0.278	1.178	1.941
	p90	1.409	2.578	3.355
	p95	1.737	3.598	4.605
	p99	4.543	6.196	7.670
	p99.9	12.416	21.222	25.275
	p99.99	37.989	54.616	63.263
CANCEL_ORDER	Mean	0.473	0.548	1.021
	Median	0.167	0.180	0.434
	Std	1.210	1.317	1.815
	Min	0.049	0.044	0.101
	Max	274.611	395.656	395.800
	p5	0.080	0.069	0.159
	p10	0.088	0.081	0.184
	p25	0.123	0.125	0.282
	p50	0.167	0.180	0.434
	p75	0.270	0.627	1.419
	p90	1.405	1.462	2.257
	p95	1.724	1.980	3.066
	p99	4.478	3.941	6.332
	p99.9	12.096	8.342	20.081
	p99.99	38.230	40.210	48.331
MODIFY_ORDER	Mean	0.488	0.926	1.414
	Median	0.179	0.315	0.624
	Std	1.210	1.841	2.234
	Min	0.052	0.072	0.136
	Max	134.942	264.471	264.624
	p5	0.091	0.109	0.216
	p10	0.101	0.127	0.254
	p25	0.138	0.212	0.399
	p50	0.179	0.315	0.624
	p75	0.284	0.867	1.899
	p90	1.415	2.556	3.289
	p95	1.736	3.498	4.587
	p99	4.505	6.655	8.012
	p99.9	14.747	14.079	23.177
	p99.99	33.256	43.974	64.766

Table 1: Full latency breakdown for NEW_ORDER, CANCEL_ORDER, and MODIFY_ORDER message types.[2]

[2]Note that the latency columns are *not additive*. Each column is derived from statistics computed independently for its respective stage. For example, the maximum value in "Stage 1" and the maximum value in "Stage 2" come from different samples, so they do not sum to the "total" maximum.

Outlier Analysis Maximum latencies reach the 300 μs range, significantly higher than the p99.9 values. These outliers likely result from:

- Linux scheduler preemption (kernel is not real-time configured)
- Cache line evictions during client connection/disconnection
- Occasional page faults or TLB misses

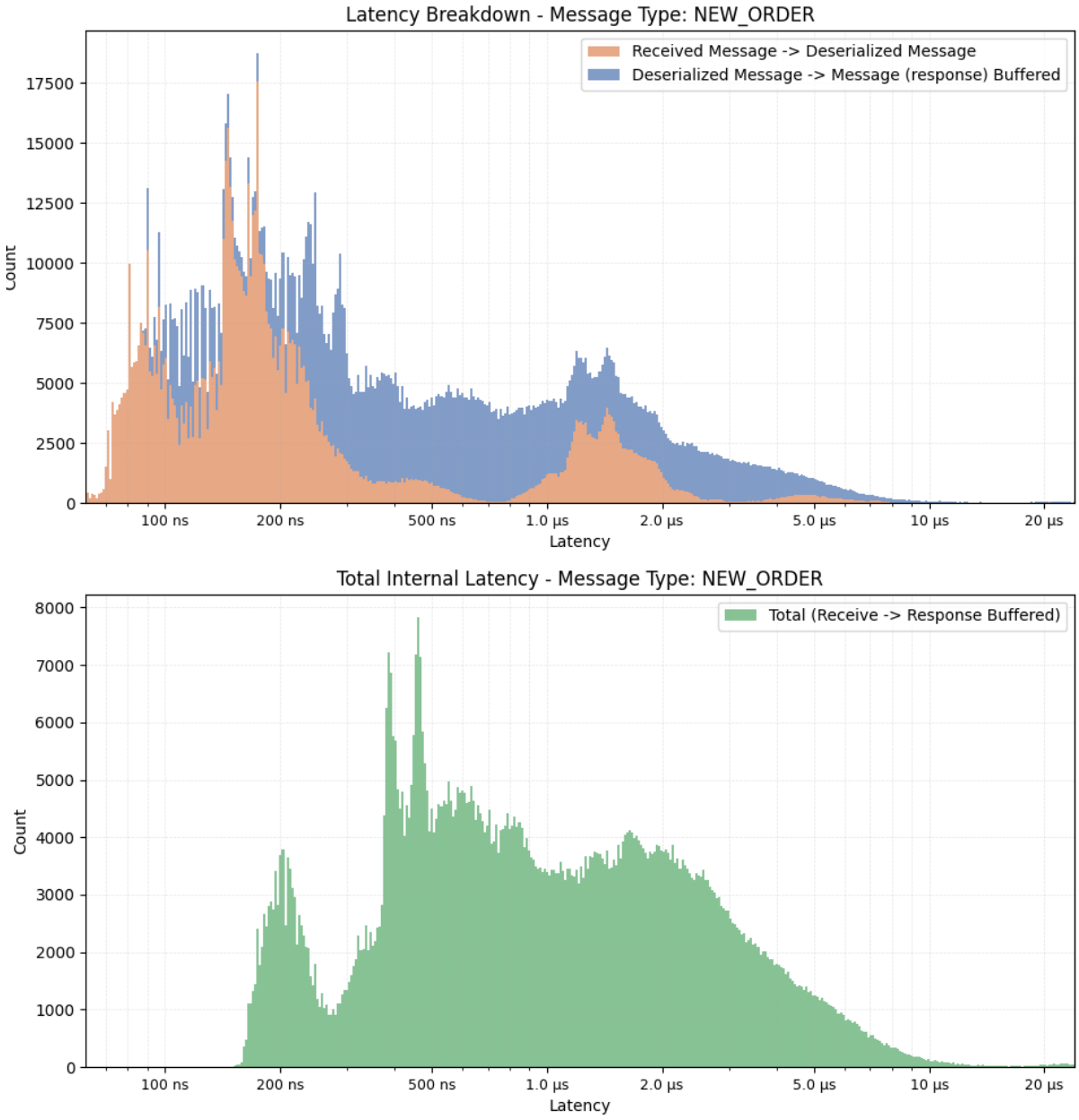

Figure 4: Internal latency of new order requests

A closer look at the latency distribution plotted on a histogram reveals an interesting pattern. The values have more than large peak. The peak around 1-2μs shows the general processing speed. The peaks at around 200-500 nanosecond show the processing speed when there are large number of cache hits in the execution path. This plot demonstrates the power of cache friendly

design, resulting in a 2-5 times speed-up in cases when data is able to be retrieved from cache memory.

Using message reference numbers in the log files, the full end-to-end message latency can be measured (from the moment a message has been read from the network, to the point after the response has been sent). The mean latency for a new order message was 35 μs, which is significantly higher than the internal latency of the system, showing that the majority of the latency is coming from TCP communication. This can be further confirmed by taking a look at the runtime flame graph of the system 7.

There are ways to improve networking latency, but the techniques required to implement are very complicated or require specialized hardware (kernel bypass, faster network interface cards (NIC), etc.) and are beyond the scope of this project at its current state.

Some issues arose when trying to measure latency including the network overhead due to challenges with separating message boundaries in bulk sends, therefore the following figures were produced in a much lower throughput test, which mostly guaranteed single message per write, this allowed for much simpler latency measurement in the gateway.

Message Type	**Count**	**Mean (µs)**	**Median (µs)**	**p95 (µs)**
HELLO	50	926.39	1043.07	1211.25
NEW_ORDER	1,175	35.32	28.56	72.39
CANCEL_ORDER	257	25.70	22.99	47.07
MODIFY_ORDER	76	28.78	24.93	58.79

Table 2: Full end-to-end latency breakdown by message type

3.4 System Call Overhead Analysis

A significant finding from performance profiling is the proportion of time spent in kernel space. Performance counter statistics revealed:

Table 3: CPU time distribution

Category	**Time (seconds)**	**Percentage**
User space (application code)	68.3	53.5%
System space (kernel)	59.5	46.5%
Total CPU time	127.8	100%

The substantial system time (46.5%) is attributable to networking system calls:

- `epoll_wait()`: Monitoring file descriptors for events
- `read()`: Reading incoming messages from sockets
- `write()`: Transmitting responses to clients
- TCP/IP stack processing in the kernel

This overhead is inherent to socket-based networking. Each message requires multiple system calls: one to read the request and typically one to write the response. At 200,000 messages/second, this translates to approximately 300,000-400,000 system calls per second, each requiring a context switch between user and kernel space.

Impact of Power Management Initial tests were conducted on battery power, where the laptop's power management limited CPU frequency to conserve energy. When connected to AC power, performance improved measurably:

Table 4: Performance improvement with AC power

Metric	Battery	AC Power	Improvement
Effective Clock Speed	2.13 GHz	2.93 GHz	+38%
NEW_ORDER mean latency	1.72 µs	1.52 µs	-12%
CANCEL_ORDER mean latency	1.14 µs	1.02 µs	-11%
MODIFY_ORDER mean latency	1.55 µs	1.41 µs	-9%

The 38% increase in sustained clock speed directly translated to 9-12% latency improvements, demonstrating the impact of thermal and power constraints on laptop hardware. Server-grade hardware with proper cooling and no power management would eliminate these limitations entirely.

Kernel Bypass Potential The 46% system time overhead represents a significant optimization opportunity. Technologies like DPDK (Data Plane Development Kit) [9] or io_uring [10] to bypass the kernel networking stack, enabling user-space packet processing. This approach could potentially:

- Eliminate majority of system time
- Reduce per-message overhead by 40-60%
- Lower tail latency by avoiding unpredictable kernel scheduling
- Improve consistency by eliminating context switch jitter

However, kernel bypass adds complexity (custom drivers, loss of standard socket APIs) and reduces portability. For a research platform prioritizing accessibility over absolute performance, the current socket-based approach provides a good balance. Future work may explore kernel bypass as an optional high-performance mode.

4 Future Work

The long-term vision for this project extends beyond a standalone matching engine. The goal is to develop an open-source platform for market microstructure research, making sophisticated trading system experimentation accessible to researchers, students, and practitioners without access to proprietary systems.

4.1 Research Platform Vision

Current market microstructure research often relies on historical data analysis or simplified models. This platform aims to provide a realistic, controllable environment where researchers can:

- Test trading strategies in a production-like matching engine
- Replay historical order flow with configurable modifications
- Study market dynamics under different rule sets (e.g., tick sizes, order types, market maker obligations)
- Analyze the impact of latency on market quality and price discovery
- Experiment with novel market designs before real-world deployment

The current implementation serves as a late-stage prototype with future work aiming to build well integrated infrastructure around the existing features.

4.2 Potential Planned Features

Market Data Infrastructure

A real-time market data feed with:

- Level 2 (full order book depth) and Level 3 (order-by-order) data
- Historical data recording and replay capabilities
- Query interface for research (e.g., "order book state at timestamp X")

Visualization and Analysis Tools

- Real-time order book heatmap visualization
- Trade flow and liquidity analysis dashboards
- Latency distribution monitoring
- Market quality metrics (spread, depth, volatility)

Configurable Market Participants

A framework for defining client behavior:

- Market makers with configurable quoting strategies
- Informed traders with signal-based behavior
- Noise traders for realistic order flow
- High-frequency traders with latency-sensitive strategies
- Policy-based order generation

Currently, clients use randomized priority queues with fixed message type distributions. The next phase will implement realistic trading strategies based on market conditions.

Scenario Testing and Replay

- Load historical order flow from real exchanges
- Replay scenarios
- Inject synthetic events (e.g., flash crash conditions, circuit breakers)
- A/B testing of market design changes

Multi-Instrument Support

Extend the single-instrument design to support:

- Thread-per-instrument parallelism for scalability
- Cross-instrument orders (spreads, baskets)
- Correlated instruments for realistic multi-asset scenarios

4.3 Immediate Next Steps

The near-term roadmap prioritizes:

1. **Improved client implementation**: Replace random message generation with realistic trading strategies to properly stress-test the server
2. **Market data feed**: Real-time updates
3. **Basic visualization**: Web-based order book display
4. **Multi-instrument support**: Validate thread-per-instrument scaling assumptions
5. **Documentation**: API documentation and contribution guidelines

These foundational features will enable the transition from prototype to research platform.

5 Discussion

The system in its current state is still relatively simple. This marks the end of the initial prototyping and exploration phase of the project with more useful and user-friendly features planned in the future, as discussed in the section above.

The message protocol is very lean, production systems would have to include more fields to satisfy more complex requirements of real markets. The simplicity of the protocol helped the system achieve the level of performance it did.

The biggest immediate challenges are message boundary tracking for more detailed network latency measurements, reducing tail latency in message responses, and creating a sensible API for client specification and configuration that would enable more sophisticated testing.

Overall, these initial results show promising potential in the viability of the platform for research and market dynamics exploration.

A Appendix

A.1 Byteswap Compiler Output

Output taken from Compiler Explorer [11].

```cpp
#include <bit>
#include <cstdint>
template <typename T> inline T swapEndian(T val) {
    if constexpr (std::endian::native == std::endian::little) {
        return std::byteswap(val);
    }
    return val; // if the condition is false, the whole expression is optimized away
}


uint64_t convert(uint64_t val) {
    return swapEndian(val);
}
```

With GCC 15.2, with options `-std=c++23 -O2`, it compiles into:

```
convert(unsigned long):
     mov rax, rdi   ; load parameter into return register
     bswap rax      ; byte swap
     ret            ; return
```

A.2 Full Latency Distribution Plots

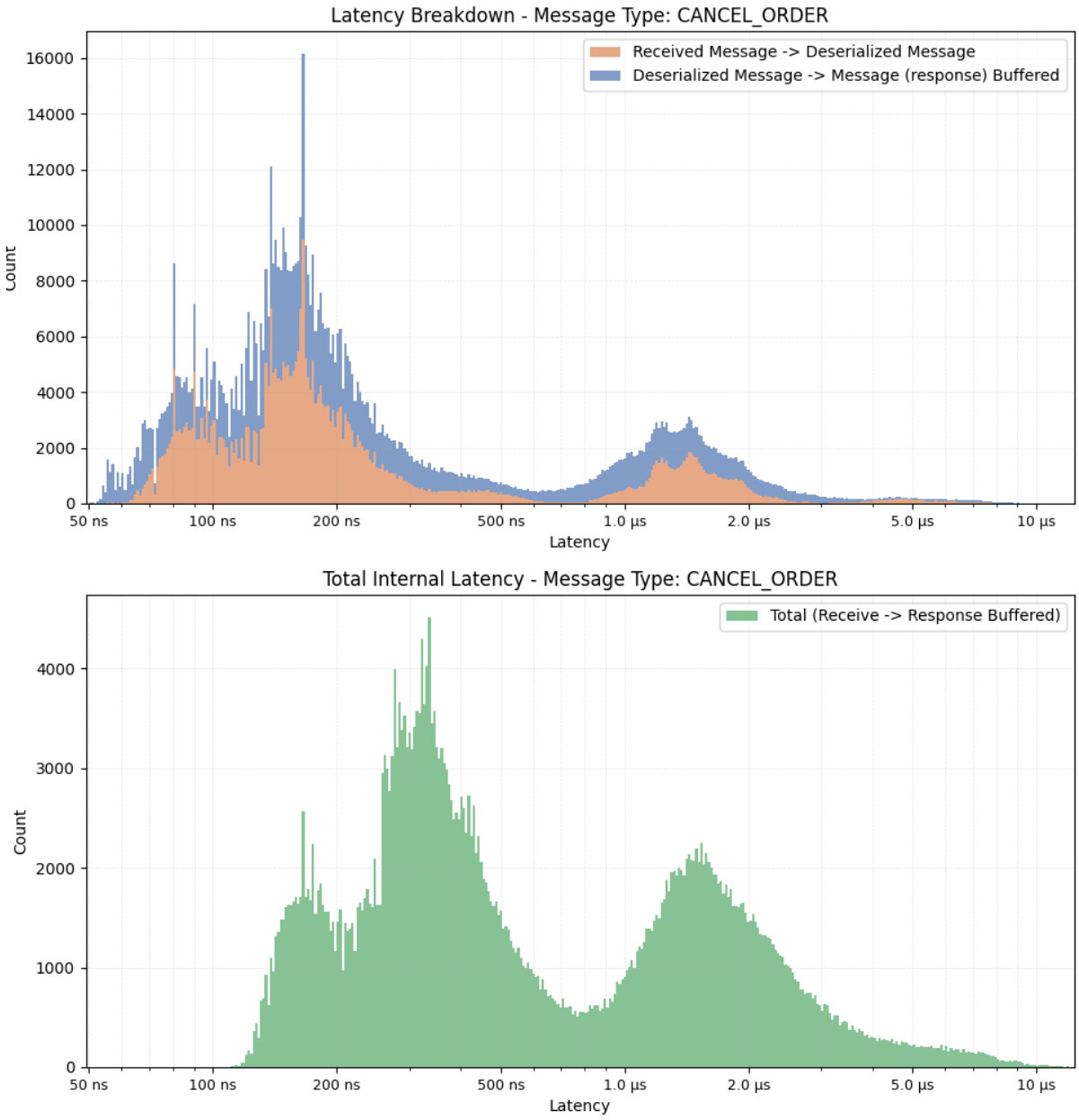

Figure 5: Internal latency for order cancellations

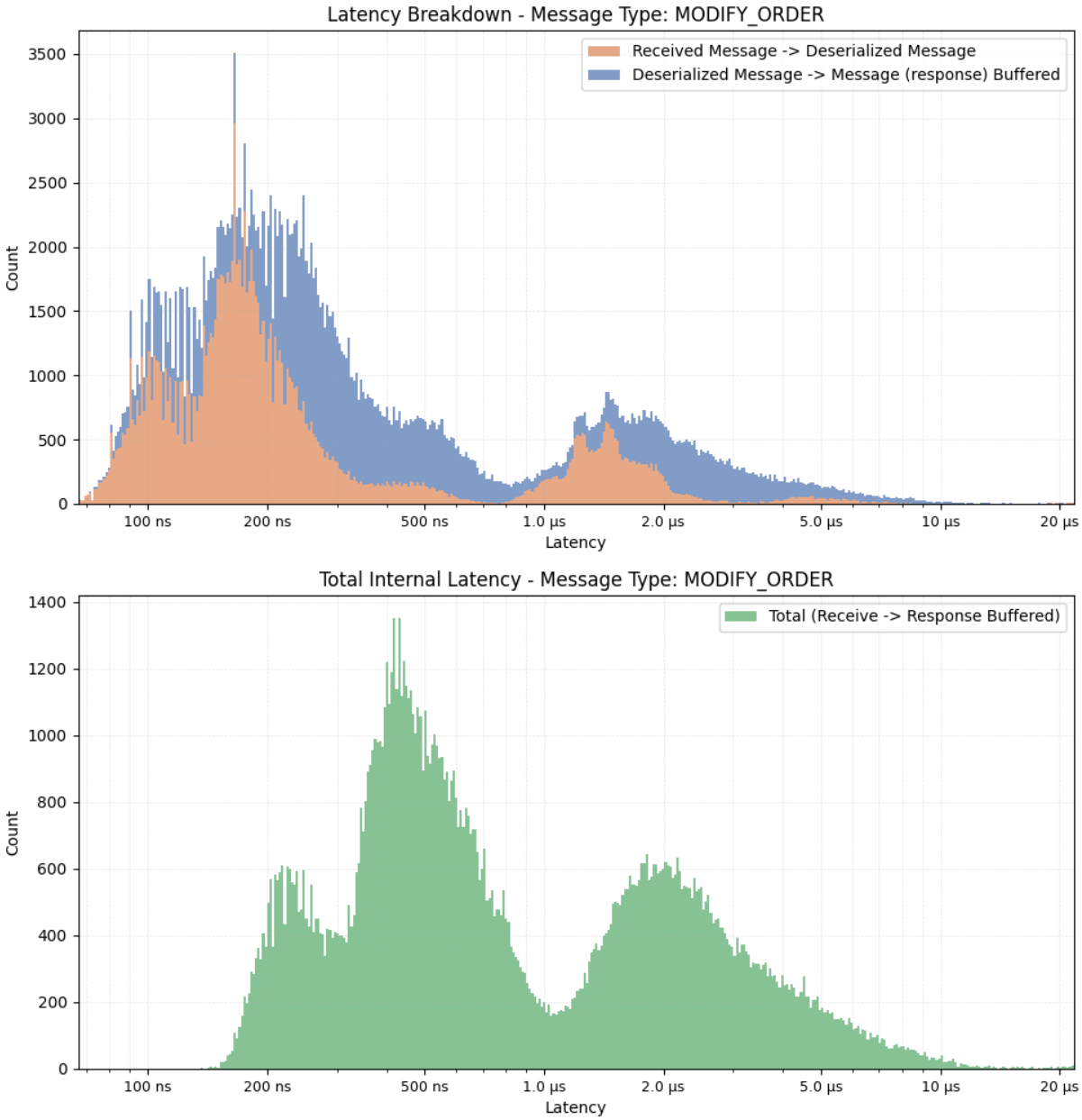

Figure 6: Internal latency for order modifications

A.3 Flame Graph

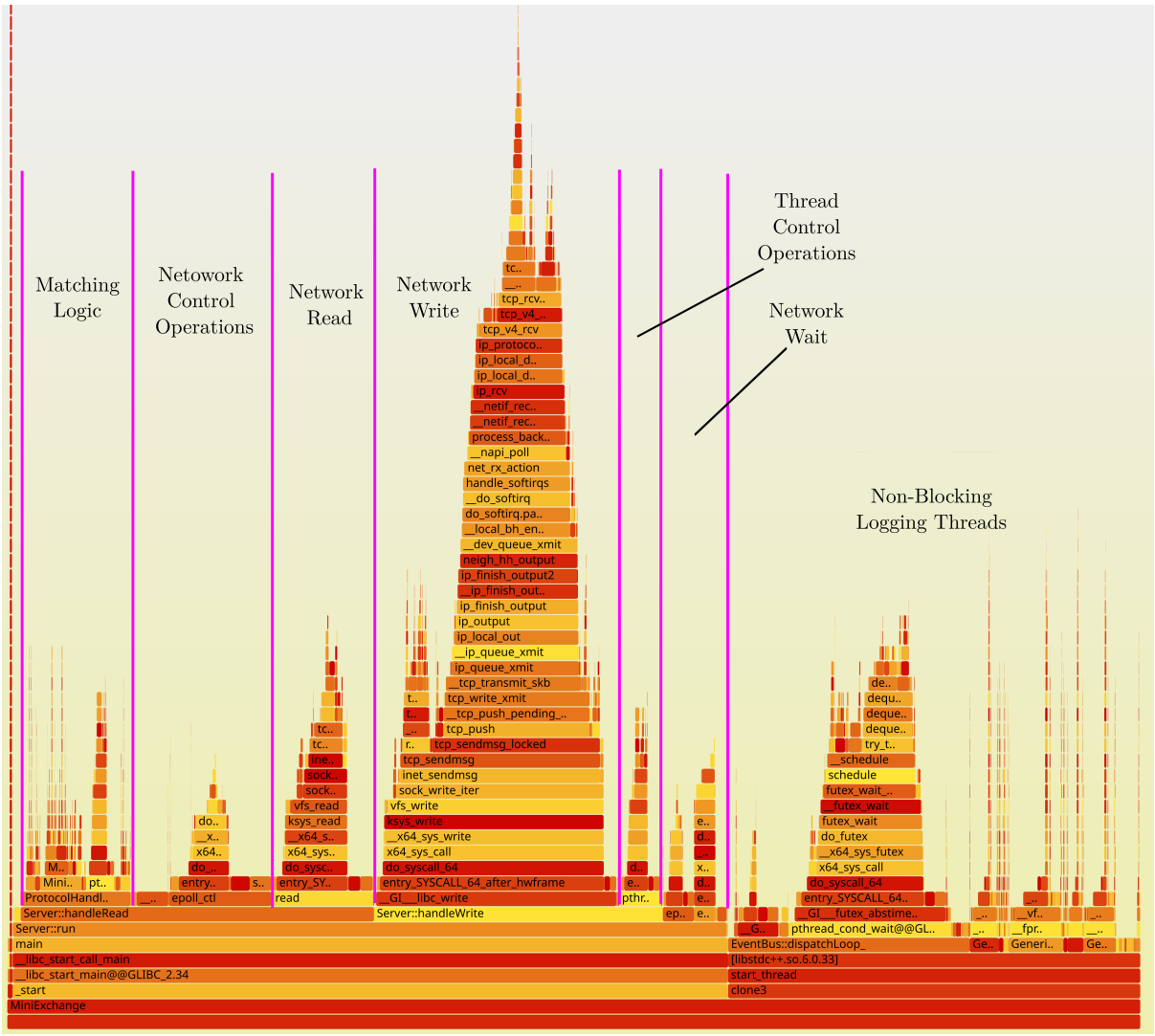

Figure 7: Runtime flame graph @ 100k requests per second with 1000 clients [12]

A.4 New Order Message Fields

Fields in message headers The full message fields specification for the protocol can be found in the docs.

B References

[1] Paul Bilokon and Burak Gunduz. *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*. 2023. arXiv: 2309.04259 [cs.PF]. URL: https://arxiv.org/abs/2309.04259.

[2] Intel Corporation. *Intel®64 and IA-32 Architectures Optimization Reference Manual, Volume 1*. Revision 050. Order Number 248966-042b. Intel Corporation. Santa Clara, CA, 2019. URL: https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html.

[3] GCC Development Team. *Structure-Packing Pragmas*. GCC 4.4.4 Manual. 2010. URL: https://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Structure_002dPacking-Pragmas.html.

[4] The Open Group. *htons - convert values between host and network byte order*. POSIX.1-2017. 2018. URL: https://man7.org/linux/man-pages/man3/htons.3p.html.

[5] cppreference.com. *std::byteswap*. C++23 feature. 2024. URL: https://en.cppreference.com/w/cpp/numeric/byteswap.html.

[6] Linux man-pages project. *epoll - I/O event notification facility*. Linux Programmer's Manual. 2024. URL: https://man7.org/linux/man-pages/man7/epoll.7.html.

[7] Linux man-pages project. *tcp - TCP protocol*. See TCP_NODELAY socket option. 2024. URL: https://man7.org/linux/man-pages/man7/tcp.7.html.

[8] John Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896. Internet Engineering Task Force, Jan. 1984. URL: https://www.rfc-editor.org/rfc/rfc896.html.

[9] *Data Plane Development Kit (DPDK)*. https://www.dpdk.org/. Accessed: 2025-1105. LF Projects / DPDK Project, 2025.

[10] *Linux man-pages project*. The Linux Programmer's Manual / man7.org. Online, 2025. URL: https://man7.org/linux/man-pages/man7/io_uring.7.html.

[11] Matt Godbolt. *Compiler Explorer*. Interactive compiler output viewer. 2024. URL: https://godbolt.org/.

[12] Brendan Gregg. *FlameGraph*. GitHub repository. 2011. URL: https://github.com/brendangregg/FlameGraph.