High-Performance Market Data Distribution System: Architecture and Design

Abstract

This write-up presents my findings of a high-performance market data distribution system designed to handle real-time order book updates with deterministic latency guarantees. The system addresses key challenges in market data distribution: maintaining event ordering, supporting client-specific backfill requests, handling burst traffic, and providing reliable historical replay. We describe the core components, data flow algorithms, and trade-offs that enable the system to deliver microsecond-level latency while maintaining correctness guarantees.

1 Introduction

Modern financial markets demand ultra-low latency data distribution systems that can handle thousands of updates per second while maintaining strict ordering guarantees. Coming from a primarily web-based applications system design mindset, my instinct was to think about a micro-service architecture with multiple pods. However, since the SLA of live data could be as low as nano-seconds, such low level latency is physically only possible with in-memory communication. Therefore, the first major switch up was to move to a single-machine architecture and utilize shared memory data structures and metal code.

2 System Requirements

The following are the requirements for the market data ingestion pipeline. All the subsequent discussions and decisions made for components, data structures or design serve to fulfill the following requirements.

2.1 Latency SLAs

The system must meet different latency requirements across three data paths:

1. **Hot Path (Live Data)**: low micro-second latency from DMA feed to client delivery. On average, we get an event every two microseconds. This is the critical path for real-time trading decisions.
2. **Warm Path (Recent Backfill)**: Millisecond-level latency for backfill. Clients catching up from brief disconnections need fast recovery.
3. **Cold Path (Historical Replay)**: Second-level latency acceptable for queries against archived data in cold storage. Used for analytics and historical analysis.

2.2 Backfilling and Replayability

The system must support:

1. **Symbol-Specific Backfill**: Clients can request historical data for arbitrary symbol subsets, starting from any previously seen sequence number.
2. **Deterministic Replay**: Given a starting sequence number and symbol set, replay must produce identical event streams across multiple requests.

3. **Seamless Transition**: Clients must transition from backfill to live data without gaps or duplicate events.

4. **Full Market Replay**: Support for replaying entire market sessions from cold storage for strategy backtesting and forensic analysis.

3 System Assumptions

The design operates under the assumption that the upstream Direct Market Access (DMA) feed never delivers events out of order. Reordering abstraction, if needed, occurs during order book ingestion. This is a very bold assumption I have taken, but a necessary one. Any incoming DMA event will have to be resolved at our in-memory Orderbook level since that is our closes approximation to market Orderbook data, and any downstream clients will be referring to this Orderbook while requiring any audit of past data. Further work has to be done in the merging of DMA and Snapshot vendors to create this Orderbook, but that's currently beyond the scope.

The second assumption is that all requests and orders will be served on the basis of sequence number. The clients will be sequence aware. Although we could create a hash for timestamp to sequnce number mapping, the solution for that is not trivial.

4 Core Components

4.1 OrderBook Service

The OrderBook Service is the heart of the system. It ingests DMA feed data, maintains the current state of all order books, and generates two types of outputs:

1. **Events**: Discrete market actions (trades, order additions, cancellations) assigned a global sequence number

2. **Snapshots**: Periodic captures of the full order book state for each symbol, indexed by sequence number

Each event receives a **monotonically increasing global sequence number**, which serves as the system's primary coordination mechanism.

4.2 Backfill Log (Memory-Mapped Event Store)

The backfill log is an append-only structure backed by memory-mapped files (mmap). This design choice provides:

1. RAM-speed access for recent events

2. Persistence without explicit disk I/O in the critical path

3. Efficient sequential scan for backfill operations

Events are indexed by their global sequence number, allowing $O(1)$ lookup by sequence. The log serves all backfill requests up to a configurable retention period.

4.3 Snapshot Storage and Index

The snapshot index maintains a per-symbol ordered list of snapshots of the inMemory orderbook, each tagged with the sequence number at which it was captured:

```
class SnapshotIndex {
public:
    // Ordered list enabling O(log n) binary search
    std::unordered_map<Symbol,
        std::vector<std::pair<uint64_t, SymbolOrderBook>>> symbol_level;
};
```

This structure allows efficient initialization of new subscriptions by providing a nearby state anchor, minimizing the amount of event replay needed.

4.4 Publisher

The publisher manages live data distribution using lock-free ring buffers. Each buffer slot contains:

```
struct RingEntry {
    uint64_t seq;       // Global sequence number
    Event event;        // The actual market event
};
```

Ring buffers provide bounded-memory fan-out to multiple clients without synchronization overhead. The write path is single-threaded, while each client maintains its own read cursor. This ensure that the queue is a SingleProducer MultiConsumer (SPMC) Queue. We can also horizonatlly scale the number of ring buffers without loss in latency.

4.5 Consumer Clients

Clients can subscribe to arbitrary subsets of symbols and may connect or disconnect at any time. Each client maintains:

```
class ClientState {
public:
    std::unordered_set<Symbol> subscriptions;
    uint64_t cursor_seq;                          // Global position
    std::unordered_map<Symbol, SymbolState> backfill_tasks;
    uint64_t backfill_cursor_seq;
};
```

This state allows the system to track which events each client has received and coordinate transitions between backfill and live data.

4.6 A note on Shared Infrastructure

Four shared components enable coordination across the system:

1. **mmap Event Log**: Fast access to recent historical events
2. **Ring Buffer**: Lock-free live distribution
3. **Snapshot Index**: Efficient client initialization
4. **global_tail_seq**: The current "now" of the system

These components are designed for concurrent access patterns—writes are serialized, but reads can occur in parallel.

5 Cold Storage and Analytics

5.1 Event Archives

After aging out of the mmap log, events are partitioned by time and enriched with metadata:

1. Minimum and maximum sequence numbers
2. Set of symbols present in the partition
3. Time range covered

This metadata enables efficient filtering during replay operations without scanning entire archives.

5.2 Derived Databases

For analytics and querying, the system maintains derived databases optimized for different access patterns:

1. **Parquet files**: Column-oriented format ideal for time-series scans and symbol-level filtering
2. **ClickHouse**: Optional OLAP database for complex aggregations
3. **DuckDB**: Serverless querying for local analysis

The philosophy is simple: optimize cold storage for reading, not writing.

6 Core Algorithms

6.1 Event Flow Pipeline

When a new event arrives from the DMA feed:

1. The OrderBook Service applies the event to the relevant symbol's order book
2. The service generates:
   (a) An event record with a new global sequence number
   (b) Snapshots for affected symbols (periodically)
3. The event is first written to the ring buffer for immediate fan-out
4. Concurrently, it's appended to the mmap backfill log for durability
5. Eventually, the event is archived to cold storage with metadata

```
struct Event {
    uint64_t seq;           // Global sequence
    Timestamp recv_ts;
    uint32_t symbol_id;
    EventType type;
    std::vector<uint8_t> payload;
};
```

This pipeline prioritizes latency (ring buffer) over durability (mmap) over archival (cold storage).

6.2 Live Consumption from Ring Buffer

Clients read from the ring buffer using a simple protocol:

```
uint64_t slot = expected_seq % RING_SIZE;
RingEntry& entry = ring[slot];

if (entry.seq == cursor_seq + 1) {
    consume(entry);
    cursor_seq++;
} else {
    // Overwritten or not yet written
    trigger_backfill();
}
```

This design is completely lock-free. If a client falls behind and its data is overwritten, it transparently falls back to backfill mode.

6.3 Backfill Protocol

Clients request backfill by specifying their last known sequence per symbol:

```
backfillRequest = {
    "symbols": {
        "AAPL": 1000,       // Last seen sequence
        "MSFT": "default", // No prior state
        "GOOGL": 2500
    }
}
```

The backfill algorithm proceeds as follows:

1. For each symbol, locate the nearest snapshot before the requested sequence ($O(\log n)$ per symbol)
2. Send all relevant snapshots to the client
3. Initialize `backfill_cursor_seq` to the minimum snapshot sequence
4. Stream events from the mmap log, filtering for subscribed symbols
5. Continue until `backfill_cursor_seq` $\geq$ `cursor_seq`
6. Transition to live ring buffer consumption

Critically, live data for non-backfilling symbols continues uninterrupted. Only symbols actively in backfill mode are gated.

6.4 Client Initialization

When a new client connects:

1. Client specifies:
   (a) Last known global sequence (or 0 for new clients)
   (b) Set of symbols to subscribe to
2. System queues relevant snapshots for all requested symbols
3. System initiates backfill from the client's sequence to current
4. Once caught up, client transitions to live consumption

This process ensures clients always have a consistent view of the market, regardless of when they connect.

7 Database Design Considerations

7.1 Access Patterns

The system exhibits clear access patterns:

1. **Write-once, read-many**: Events are immutable once written
2. **Sequential writes**: Events arrive in order
3. **Time-range queries**: Most queries filter by time
4. **Symbol filtering**: Queries often focus on specific symbols

7.2 Storage Format

We use Parquet for cold storage due to its:

1. Columnar layout (efficient for selective reads)
2. Built-in compression
3. Rich metadata support
4. Wide ecosystem support

7.3 Partitioning Strategy

Following the principle of "partition on low cardinality, order by high cardinality":

1. **Partition by**: Time (e.g., hourly or daily files)
2. **Order within partition**: (symbol, timestamp, event_seq)
3. **Metadata per partition**: Min/max sequence, symbol set, time range

This design enables efficient pruning at the partition level, followed by fast seeks within files.

8 Handling Burst Traffic

In high-frequency markets, burst traffic is inevitable. Our design explicitly prioritizes latency predictability over data granularity during bursts.

8.1 Invariants Under Load

1. Clients continue receiving data within the same latency SLA
2. Clients are explicitly notified when burst mode activates
3. Clients not subscribed to bursty symbols are unaffected

The system sacrifices granularity, not correctness or latency.

8.2 Data Prioritization

Market data is prioritized by importance to price discovery:

1. **Trades**: Never dropped (highest priority)
2. **Top-of-book updates**: Strongly protected
3. **Depth updates**: May be suppressed

We implement separate ring buffers per data class, each with independent capacity and drop policies.

8.3 Depth Suppression

During bursts, depth updates may be:

1. **Coalesced**: Multiple updates combined into one
2. **Suppressed**: Dropped entirely

When suppression occurs:

1. Clients receive explicit notification of the suppressed sequence ranges
2. Dropped depth is **not** persisted to storage (reducing I/O load)
3. Clients can request a snapshot if they need to resynchronize

Suppression happens *before* the ring buffer write, eliminating I/O overhead in the critical path.

8.4 Recovery Mechanisms

Clients can validate their internal state and request snapshots if:

1. They detect inconsistencies in their local order book
2. They receive an explicit state-invalidation signal
3. They require full depth granularity and cannot tolerate reduced fidelity

Snapshots serve as the ultimate consistency guarantee.

8.5 Design Guarantees

This approach provides:

1. Bounded latency (deterministic performance)
2. Correct price discovery (trades and TOB never dropped)
3. Deterministic behavior under overload
4. Graceful degradation instead of failure

9 Conclusion

This architecture demonstrates that it's possible to build a market data distribution system that simultaneously achieves:

1. Microsecond-level latency for live data
2. Flexible historical backfill
3. Deterministic behavior under load
4. Efficient storage for analytics

The key insights are:

1. Use global sequencing for coordination
2. Separate hot path (ring buffer) from warm path (mmap) from cold path (archives)
3. Make trade-offs explicit (latency vs. granularity during bursts)
4. Design for the common case (live consumption) but handle the exceptional case (backfill) efficiently

Future work could explore adaptive snapshot frequencies, predictive backfill prefetching, and hierarchical caching strategies for frequently accessed symbols.

10 Conclusion

This architecture demonstrates that it's possible to build a market data distribution system that simultaneously achieves:

1. Microsecond-level latency for live data
2. Flexible historical backfill
3. Deterministic behavior under load
4. Efficient storage for analytics

The key insights are:

1. Use global sequencing for coordination
2. Separate hot path (ring buffer) from warm path (mmap) from cold path (archives)
3. Make trade-offs explicit (latency vs. granularity during bursts)
4. Design for the common case (live consumption) but handle the exceptional case (backfill) efficiently

11 Further Study Required

This study was an initial attempt to understand the problems with low-latency market data ingestion systems, and this is still nowhere close to being production grade. The following is a few pointers about what I intend to explore next ...

11.1 Memory-Mapped File Durability Guarantees

The mmap-based backfill log gives excellent read performance, but page faults can introduce unpredictable latency under memory pressure. **Required work:** ensure the append-only log's pages remain resident in memory, via preallocation and explicit locking hints (e.g., mlock), instead of relying on the kernel's default flushing and eviction behavior.

11.2 DMA Feed Reordering and Conoliation

In practice, trading systems consume both snapshot feeds and DMA feeds simultaneously. DMA feeds are typically multicast UDP streams, which provide low latency but do not guarantee ordering or reliability. As a result, downstream consumers must implement sequence-based reordering, deduplication, and gap detection on their end.

When gaps or anomalies are detected, such as packet loss, out-of-order delivery, or prolonged DMA stalls, systems must fall back to the snapshot feed. A robust mechanism is required to switch to snapshot ingestion during DMA faults and to transition back to DMA gracefully, ensuring consistency and continuity between the two feed types.

11.3 Timestamp-to-Sequence Mapping

The system relies entirely on sequence numbers, but regulatory requirements and analytics often demand timestamp-based queries.

Required work: Implement a timestamp index for cold storage, even if hot/warm paths remain sequence-based. Design efficient timestamp-to-sequence lookup structures (B-tree, skip list) for common query patterns. I can consider maintaining coarse-grained timestamp checkpoints (e.g., sequence number at each minute boundary).