13 Stock Exchange

In this chapter, we design an electronic stock exchange system.

The basic function of an exchange is to facilitate the matching of buyers and sellers efficiently. This fundamental function has not changed over time. Before the rise of computing, people exchanged tangible goods by bartering and shouting at each other to get matched. Today, orders are processed silently by supercomputers, and people trade not only for the exchange of products, but also for speculation and arbitrage. Technology has greatly changed the landscape of trading and exponentially boosted electronic market trading volume.

When it comes to stock exchanges, most people think about major market players like The New York Stock exchange (NYSE) or Nasdaq, which have existed for over fifty years. In fact, there are many other types of exchange. Some focus on vertical segmentation of the financial industry and place special focus on technology [1], while others have an emphasis on fairness [2]. Before diving into the design, it is important to check with the interviewer about the scale and the important characteristics of the exchange in question.

Just to get a taste of the kind of problem we are dealing with; NYSE is trading billions of matches per day [3], and HKEX about 200 billion shares per day [4]. Figure 13.1 shows the big exchanges in the "trillion-dollar club" by market capitalization.

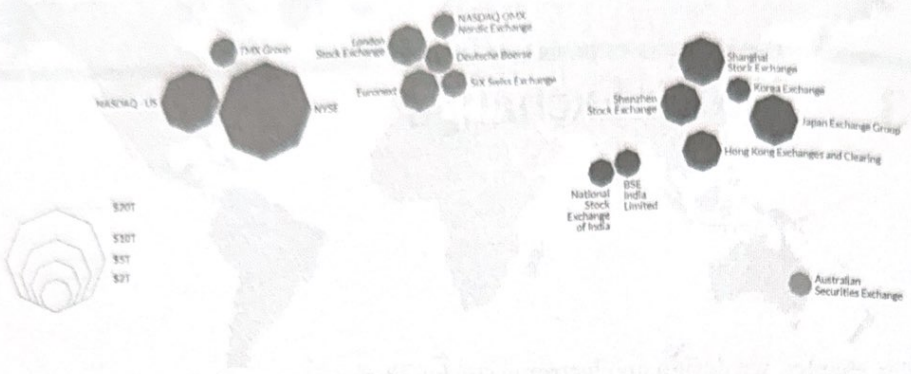

Figure 13.1: Largest stock exchanges (Source: [5])

Step 1 - Understand the Problem and Establish Design Scope

A modern exchange is a complicated system with stringent requirements on latency, throughput, and robustness. Before we start, let's ask the interviewer a few questions to clarify the requirements.

Candidate: Which securities are we going to trade? Stocks, options, or futures?
Interviewer: For simplicity, only stocks.

Candidate: Which types of order operations are supported: placing a new order, canceling an order, or replacing an order? Do we need to support limit order, market order, or conditional order?
Interviewer: We need to support the following: placing a new order and canceling an order. For the order type, we only need to consider the limit order.

Candidate: Does the system need to support after-hours trading?
Interviewer: No, we just need to support the normal trading hours.

Candidate: Could you describe the basic functions of the exchange? And the scale of the exchange, such as how many users, how many symbols, and how many orders?
Interviewer: A client can place new limit orders or cancel them, and receive matched trades in real-time. A client can view the real-time order book (the list of buy and sell orders). The exchange needs to support at least tens of thousands of users trading at the same time, and it needs to support at least 100 symbols. For the trading volume, we should support billions of orders per day. Also, the exchange is a regulated facility, so we need to make sure it runs risk checks.

Candidate: Could you please elaborate on risk checks?
Interviewer: Let's just do simple risk checks. For example, a user can only trade a maximum of 1 million shares of Apple stock in one day.

Candidate: I noticed you didn't mention user wallet management. Is it something we also need to consider?

Interviewer: Good catch! We need to make sure users have sufficient funds when they place orders. If an order is waiting in the order book to be filled, the funds required for the order need to be withheld to prevent overspending.

Non-functional requirements

After checking with the interviewer for the functional requirements, we should determine the non-functional requirements. In fact, requirements like "at least 100 symbols" and "tens of thousands of users" tell us that the interviewer wants us to design a small-to-medium scale exchange. On top of this, we should make sure the design can be extended to support more symbols and users. Many interviewers focus on extensibility as an area for follow-up questions.

Here is a list of non-functional requirements:

- **Availability.** At least 99.99%. Availability is crucial for exchanges. Downtime, even seconds, can harm reputation.
- **Fault tolerance.** Fault tolerance and a fast recovery mechanism are needed to limit the impact of a production incident.
- **Latency.** The round-trip latency should be at the millisecond level, with a particular focus on the 99th percentile latency. The round trip latency is measured from the moment a market order enters the exchange to the point where the market order returns as a filled execution. A persistently high 99th percentile latency causes a terrible user experience for a small number of users.
- **Security.** The exchange should have an account management system. For legal and compliance, the exchange performs a KYC (Know Your Client) check to verify a user's identity before a new account is opened. For public resources, such as web pages containing market data, we should prevent distributed denial-of-service (DDoS) [6] attacks.

Back-of-the-envelope estimation

Let's do some simple back-of-the-envelope calculations to understand the scale of the system:

- 100 symbols
- 1 billion orders per day
- NYSE Stock exchange is open Monday through Friday from 9:30 am to 4:00 pm Eastern Time. That's 6.5 hours in total.
- QPS: $\frac{1 \text{ billion}}{6.5 \times 3{,}600} = \sim 43{,}000$
- Peak QPS: $5 \times \text{QPS} = 215{,}000$. The trading volume is significantly higher when the market first opens in the morning and before it closes in the afternoon.

Step 2 - Propose High-Level Design and Get Buy-In

Before we dive into the high-level design, let's briefly discuss some basic concepts and terminology that are helpful for designing an exchange.

Business Knowledge 101

Broker

Most retail clients trade with an exchange via a broker. Some brokers whom you might be familiar with include Charles Schwab, Robinhood, E*Trade, Fidelity, etc. These brokers provide a friendly user interface for retail users to place trades and view market data.

Institutional client

Institutional clients trade in large volumes using specialized trading software. Different institutional clients operate with different requirements. For example, pension funds aim for a stable income. They trade infrequently, but when they do trade, the volume is large. They need features like order splitting to minimize the market impact [7] of their sizable orders. Some hedge funds specialize in market making and earn income via commission rebates. They need low latency trading abilities, so obviously they cannot simply view market data on a web page or a mobile app, as retail clients do.

Limit order

A limit order is a buy or sell order with a fixed price. It might not find a match immediately, or it might just be partially matched.

Market order

A market order doesn't specify a price. It is executed at the prevailing market price immediately. A market order sacrifices cost in order to guarantee execution. It is useful in certain fast-moving market conditions.

Market data levels

The US stock market has three tiers of price quotes: L1 (level 1), L2, and L3. L1 market data contains the best bid price, ask price, and quantities (Figure 13.2). Bid price refers to the highest price a buyer is willing to pay for a stock. Ask price refers to the lowest price a seller is willing to sell the stock.

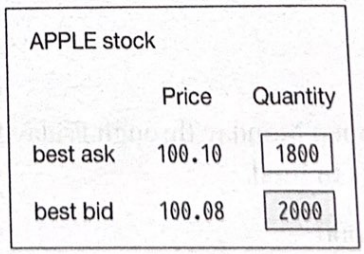

Figure 13.2: Level 1 data

L2 includes more price levels than L1 (Figure 13.3).

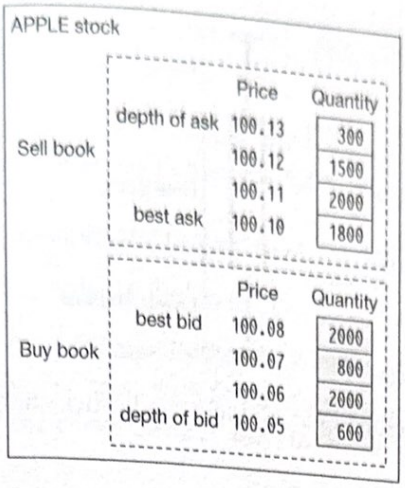

Figure 13.3: Level 2 data

L3 shows price levels and the queued quantity at each price level (Figure 13.4).

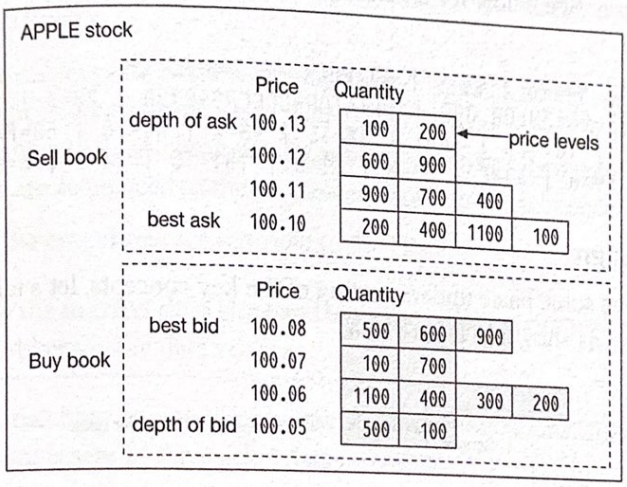

Figure 13.4: Level 3 data

Candlestick chart

A candlestick chart represents the stock price for a certain period of time. A typical candlestick looks like this (Figure 13.5). A candlestick shows the market's open, close, high, and low price for a time interval. The common time intervals are one-minute, five-minute, one-hour, one-day, one-week, and one-month.

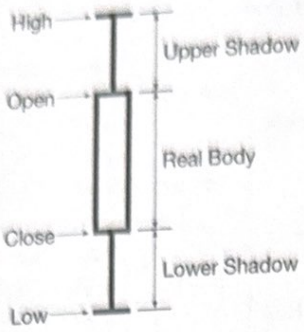

Figure 13.5: A single candlestick chart

FIX

FIX protocol [8], which stands for Financial Information exchange protocol, was created in 1991. It is a vendor-neutral communications protocol for exchanging securities transaction information. See below for an example of a securities transaction encoded in FIX [8].

```
8=FIX.4.2 | 9=176 | 35=8 | 49=PHLX | 56=PERS |
52=20071123-05:30:00.000 | 11=ATOMNOCCC9990900 | 20=3 | 150=E | 39=E
 | 55=MSFT | 167=CS | 54=1 | 38=15 | 40=2 | 44=15 | 58=PHLX EQUITY
TESTING | 59=0 | 47=C | 32=0 | 31=0 | 151=15 | 14=0 | 6=0 | 10=128 |
```

High-level design

Now that we have some basic understanding of the key concepts, let's take a look at the high-level design, as shown in Figure 13.6.

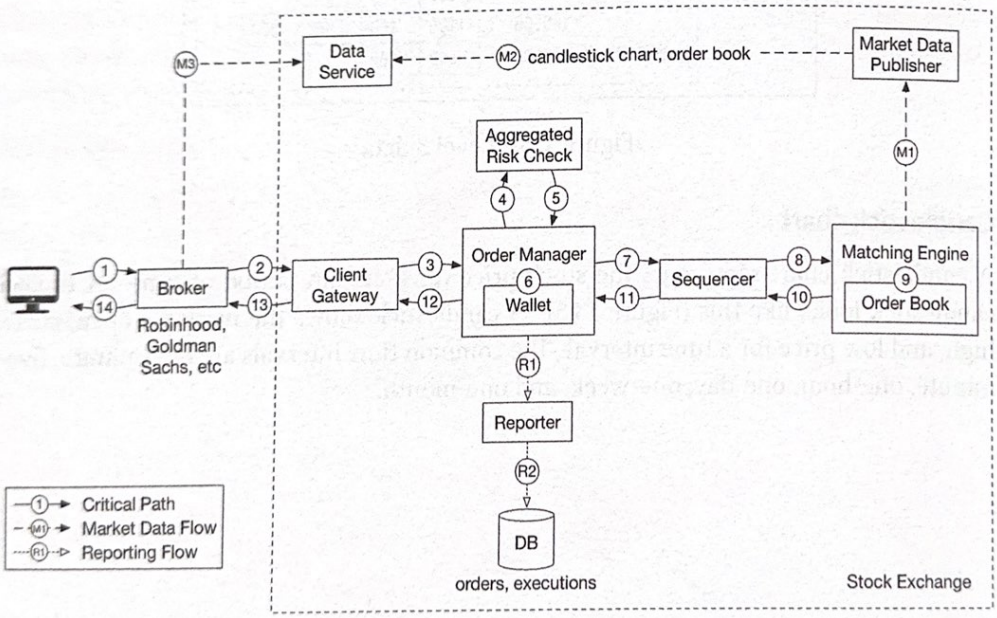

Figure 13.6: High-level design

Let's trace the life of an order through various components in the diagram to see how the pieces fit together.

First, we follow the order through the **trading flow**. This is the critical path with strict latency requirements. Everything has to happen fast in the flow:

Step 1: A client places an order via the broker's web or mobile app.

Step 2: The broker sends the order to the exchange.

Step 3: The order enters the exchange through the client gateway. The client gateway performs basic gatekeeping functions such as input validation, rate limiting, authentication, normalization, etc. The client gateway then forwards the order to the order manager.

Step 4 ~ 5: The order manager performs risk checks based on rules set by the risk manager.

Step 6: After passing risk checks, the order manager verifies there are sufficient funds in the wallet for the order.

Step 7 ~ 9: The order is sent to the matching engine. When a match is found, the matching engine emits two executions (also called fills), with one each for the buy and sell sides. To guarantee that matching results are deterministic when replayed, both orders and executions are sequenced in the sequencer (more on the sequencer later).

Step 10 ~ 14: The executions are returned to the client.

Next, we follow the **market data flow** and trace the order executions from the matching engine to the broker via the data service.

Step M1: The matching engine generates a stream of executions (fills) as matches are made. The stream is sent to the market data publisher.

Step M2: The market data publisher constructs the candlestick charts and the order books as market data from the stream of executions and orders. It then sends market data to the data service.

Step M3: The market data is saved to specialized storage for real-time analytics. Brokers connect to the data service to obtain timely market data. Brokers relay market data to their clients.

Lastly, we examine the **reporting flow**.

Step R1~R2 (reporting flow): The reporter collects all the necessary reporting fields (e.g. `client_id`, `price`, `quantity`, `order_type`, `filled_quantity`, `remaining_quantity`) from orders and executions, and writes the consolidated records to the database.

Note that the trading flow (steps 1 to 14) is on the critical path, while the market data flow and reporting flow are not. They have different latency requirements.

Now let's examine each of the three flows in more detail.

Trading flow

The trading flow is on the critical path of the exchange. Everything must happen fast. The heart of the trading flow is the matching engine. Let's go over that first.

Matching engine

The matching engine is also called the cross engine. Here are the primary responsibilities of the matching engine:

1. Maintain the order book for each symbol. An order book is a list of buy and sell orders for a symbol. We explain the construction of an order book in the Data models section later.
2. Match buy and sell orders. A match results in two executions (one from the buy side and the other from the sell side). The matching function must be fast and accurate.
3. Distribute the execution stream as market data.

A highly available matching engine implementation must be able to produce matches in a deterministic order. That is, given a known sequence of orders as an input, the matching engine must produce the same sequence of executions (fills) as an output when the sequence is replayed. This determinism is a foundation of high availability which we will discuss at length in the deep dive section.

Sequencer

The sequencer is the key component that makes the matching engine deterministic. It stamps every incoming order with a sequence ID before it is processed by the matching engine. It also stamps every pair of executions (fills) completed by the matching engine with sequence IDs. In other words, the sequencer has an inbound and an outbound instance, with each maintaining its own sequences. The sequence generated by each sequencer must be sequential numbers, so that any missing numbers can be easily detected. See Figure 13.7 for details.

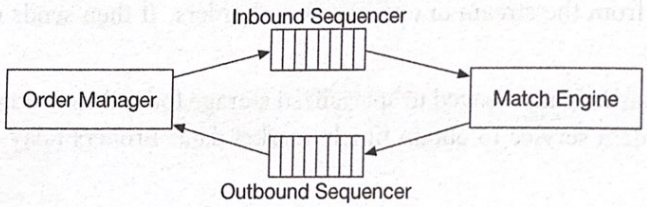

Figure 13.7: Inbound and outbound sequencers

The incoming orders and outgoing executions are stamped with sequence IDs for these reasons:

1. Timeliness and fairness
2. Fast recovery / replay
3. Exactly-once guarantee

The sequencer does not only generate sequence IDs. It also functions as a message queue. There is one to send messages (incoming orders) to the matching engine, and another one to send messages (executions) back to the order manager. It is also an event store for the orders and executions. It is similar to having two Kafka event streams connected to the matching engine, one for incoming orders and the other for outgoing executions. In fact, we could have used Kafka if its latency was lower and more predictable. We discuss how the sequencer is implemented in a low-latency exchange environment in the deep dive section.

Order manager

The order manager receives orders on one end and receives executions on the other. It manages the orders' states. Let's look at it closely.

The order manager receives inbound orders from the client gateway and performs the following:

- It sends the order for risk checks. Our requirements for risk checking are simple. For example, we verify that a user's trade volume is below $1M a day.
- It checks the order against the user's wallet and verifies that there are sufficient funds to cover the trade. The wallet was discussed at length in the "Digital Wallet" chapter on page 341. Refer to that chapter for an implementation that would work in the exchange.
- It sends the order to the sequencer where the order is stamped with a sequence ID. The sequenced order is then processed by the matching engine. There are many attributes in a new order, but there is no need to send all the attributes to the matching engine. To reduce the size of the message in data transmission, the order manager only sends the necessary attributes.

On the other end, the order manager receives executions from the matching engine via the sequencer. The order manager returns the executions for the filled orders to the brokers via the client gateway.

The order manager should be fast, efficient, and accurate. It maintains the current states for the orders. In fact, the challenge of managing the various state transitions is the major source of complexity for the order manager. There can be tens of thousands of cases involved in a real exchange system. Event sourcing [9] is perfect for the design of an order manager. We discuss an event sourcing design in the deep dive section.

Client gateway

The client gateway is the gatekeeper for the exchange. It receives orders placed by clients and routes them to the order manager. The gateway provides the following functions as shown in Figure 13.8.

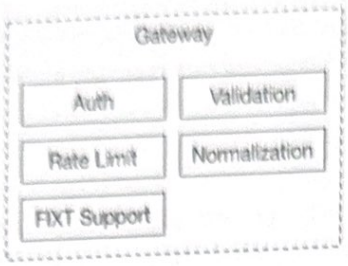

Figure 13.8: Client gateway components

The client gateway is on the critical path and is latency-sensitive. It should stay lightweight. It passes orders to the correct destinations as quickly as possible. The functions above, while critical, must be completed as quickly as possible. It is a design trade-off to decide what functionality to put in the client gateway, and what to leave out. As a general guideline, we should leave complicated functions to the matching engine and risk check.

There are different types of client gateways for retail and institutional clients. The main considerations are latency, transaction volume, and security requirements. For instance, institutions like the market makers provide a large portion of liquidity for the exchange. They require very low latency. Figure 13.9 shows different client gateway connections to an exchange. An extreme example is the colocation (colo) engine. It is the trading engine software running on some servers rented by the broker in the exchange's data center. The latency is literally the time it takes for light to travel from the colocated server to the exchange server [10].

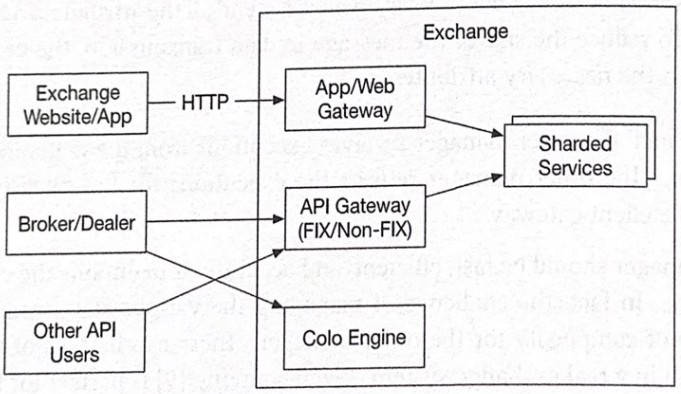

Figure 13.9: Client gateway

Market data flow

The market data publisher (MDP) receives executions from the matching engine and builds the order books and candlestick charts from the stream of executions. The order books and candlestick charts, which we discuss in the Data Models section later, are collectively called market data. The market data is sent to the data service where they are made available to subscribers. Figure 13.10 shows an implementation of MDP and how it fits with the other components in the market data flow.

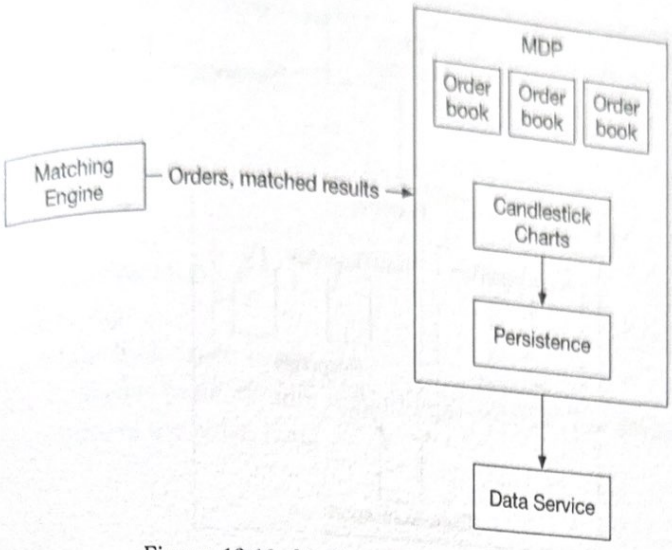

Figure 13.10: Market Data Publisher

Reporting flow

One essential part of the exchange is reporting. The reporter is not on the trading critical path, but it is a critical part of the system. It provides trading history, tax reporting, compliance reporting, settlements, etc. Efficiency and latency are critical for the trading flow, but the reporter is less sensitive to latency. Accuracy and compliance are key factors for the reporter.

It is common practice to piece attributes together from both incoming orders and outgoing executions. An incoming new order contains order details, and outgoing execution usually only contains order ID, price, quantity, and execution status. The reporter merges the attributes from both sources for the reports. Figure 13.11 shows how the components in the reporting flow fit together.

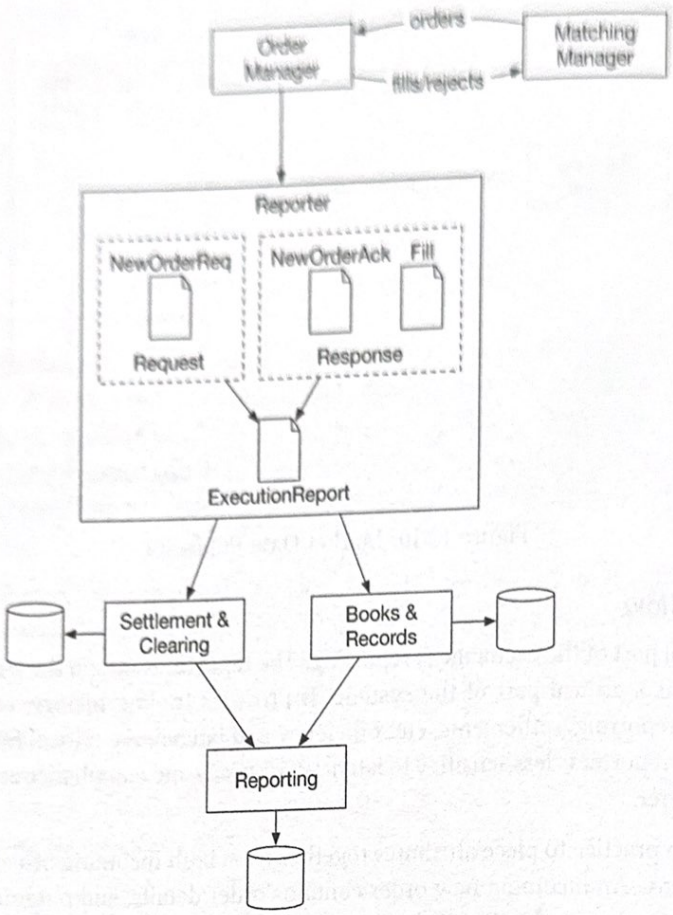

Figure 13.11: Reporter

A sharp reader might notice that the section order of "Step 2 - Propose High-Level Design and Get Buy-In" looks a little different than other chapters. In this chapter, the API design and data models sections come after the high-level design. The sections are arranged this way because these other sections require some concepts that were introduced in the high-level design.

API Design

Now that we understand the high-level design, let's take a look at the API design.

Clients interact with the stock exchange via the brokers to place orders, view executions, view market data, download historical data for analysis, etc. We use the RESTful conventions for the API below to specify the interface between the brokers and the client gateway. Refer to the "Data models" section on page 393 for the resources mentioned below.

Note that the RESTful API might not satisfy the latency requirements of institutional clients like hedge funds. The specialized software built for these institutions likely uses a different protocol, but no matter what it is, the basic functionality mentioned below

needs to be supported.

Order

`POST /v1/order`

This endpoint places an order. It requires authentication.

Parameters

`symbol`: the stock symbol. String
`side`: buy or sell. String
`price`: the price of the limit order. Long
`orderType`: limit or market (note we only support limit orders in our design). String
`quantity`: the quantity of the order. Long

Response

Body:

`id`: the ID of the order. Long
`creationTime`: the system creation time of the order. Long
`filledQuantity`: the quantity that has been successfully executed. Long
`remainingQuantity`: the quantity still to be executed. Long
`status`: `new/canceled/filled`. String
rest of the attributes are the same as the input parameters

Code:

200: successful
40x: parameter error/access denied/unauthorized
500: server error

Execution

`GET /v1/execution?symbol={:symbol}&orderId={:orderId}&startTime={:startTime}&endTime={:endTime}`

This endpoint queries execution info. It requires authentication.

Parameters

`symbol`: the stock symbol. String
`orderId`: the ID of the order. Optional. String
`startTime`: query start time in epoch [11]. Long
`endTime`: query end time in epoch. Long

Response

Body:

executions: array with each execution in scope (see attributes below). Array
id: the ID of the execution. Long
orderId: the ID of the order. Long
symbol: the stock symbol. String
side: buy or sell. String
price: the price of the execution. Long
orderType: limit or market. String
quantity: the filled quantity. Long

Code:

200: successful
40x: parameter error/not found/access denied/unauthorized
500: server error

Order book

`GET /v1/marketdata/orderBook/L2?symbol={:symbol}&depth={:depth}`

This endpoint queries L2 order book information for a symbol with designated depth.

Parameters

`symbol`: the stock symbol. String
`depth`: order book depth per side. Int
`startTime`: query start time in epoch. Long
`endTime`: query end time in epoch. Long

Response

Body:

`bids`: array with price and size. Array
`asks`: array with price and size.Array

Code:

200: successful
40x: parameter error/not found/access denied/unauthorized
500: server error

Historical prices (candlestick charts)

`GET /v1/marketdata/candles?symbol={:symbol}&resolution={:resolution}&startTime {:startTime}&endTime={:endTime}`

This endpoint queries candlestick chart data (see candlestick chart in data models section) for a symbol given a time range and resolution.

Parameters

symbol: the stock symbol. String
resolution: window length of the candlestick chart in seconds. Long
startTime: start time of the window in epoch. Long
endTime: end time of the window in epoch. Long

Response

Body:

candles: array with each candlestick data (attributes listed below). Array
open: open price of each candlestick. Double
close: close price of each candlestick. Double
high: high price of each candlestick. Double
low: low price of each candlestick. Double

Code:

200: successful
40x: parameter error/not found/access denied/unauthorized
500: server error

Data models

There are three main types of data in the stock exchange. Let's explore them one by one.

- Product, order, and execution
- Order book
- Candlestick chart

Product, order, execution

A product describes the attributes of a traded symbol, like product type, trading symbol, UI display symbol, settlement currency, lot size, tick size, etc. This data doesn't change frequently. It is primarily used for UI display. The data can be stored in any database and is highly cacheable.

An order represents the inbound instruction for a buy or sell order. An execution represents the outbound matched result. An execution is also called a fill. Not every order has an execution. The output of the matching engine contains two executions, representing the buy and sell sides of a matched order.

See Figure 13.12 for the logical model diagram that shows the relationships between the three entities. Note it is not a database schema.

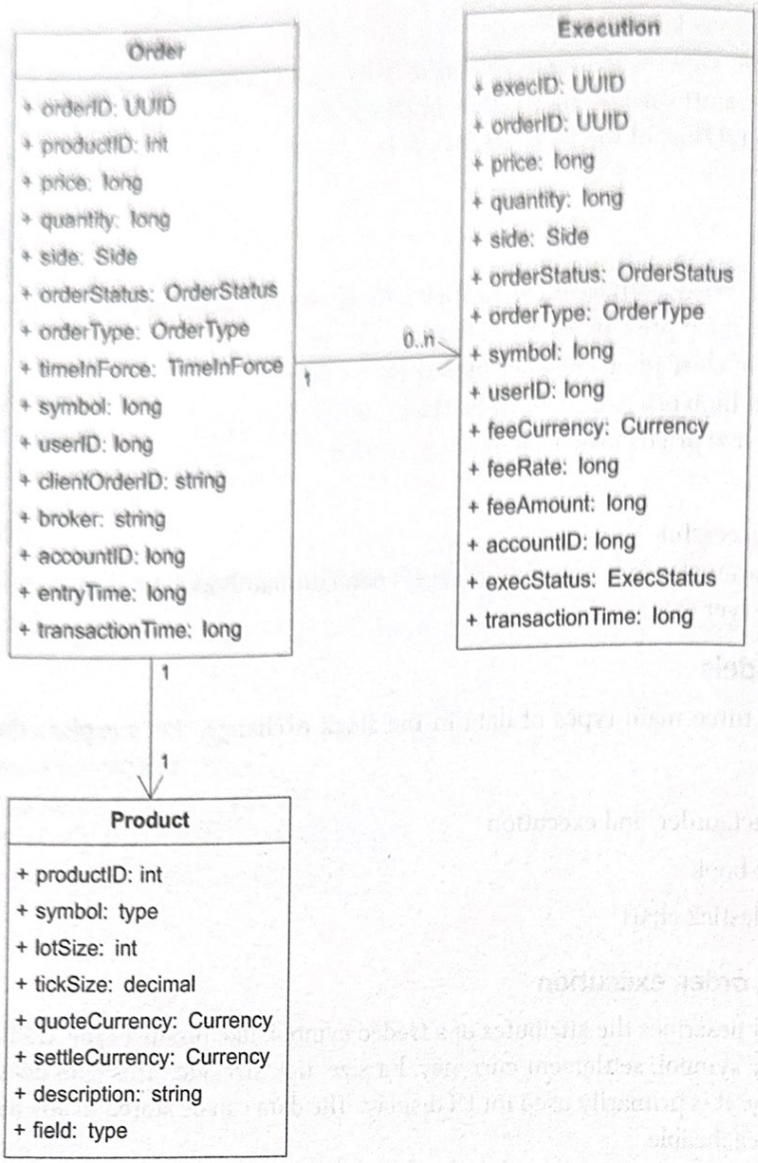

Figure 13.12: Product, order, execution

Orders and executions are the most important data in the exchange. We encounter them in all three flows mentioned in the high-level design, in slightly different forms.

- In the critical trading path, orders and executions are not stored in a database. To achieve high performance, this path executes trades in memory and leverages hard disk or shared memory to persist and share orders and executions. Specifically, orders and executions are stored in the sequencer for fast recovery, and data is archived after the market closes. We discuss an efficient implementation of the sequencer in the deep dive section.
- The reporter writes orders and executions to the database for reporting use cases like reconciliation and tax reporting.

- Executions are forwarded to the market data processor to reconstruct the order book and candlestick chart data. We discuss these data types next.

Order book

An order book is a list of buy and sell orders for a specific security or financial instrument, organized by price level [12] [13]. It is a key data structure in the matching engine for fast order matching. An efficient data structure for an order book must satisfy these requirements:

- Constant lookup time. Operation includes: getting volume at a price level or between price levels.
- Fast add/cancel/execute operations, preferably $O(1)$ time complexity. Operations include: placing a new order, canceling an order, and matching an order.
- Fast update. Operation: replacing an order.
- Query best bid/ask.
- Iterate through price levels.

Let's walk through an example order execution against an order book, as illustrated in Figure 13.13.

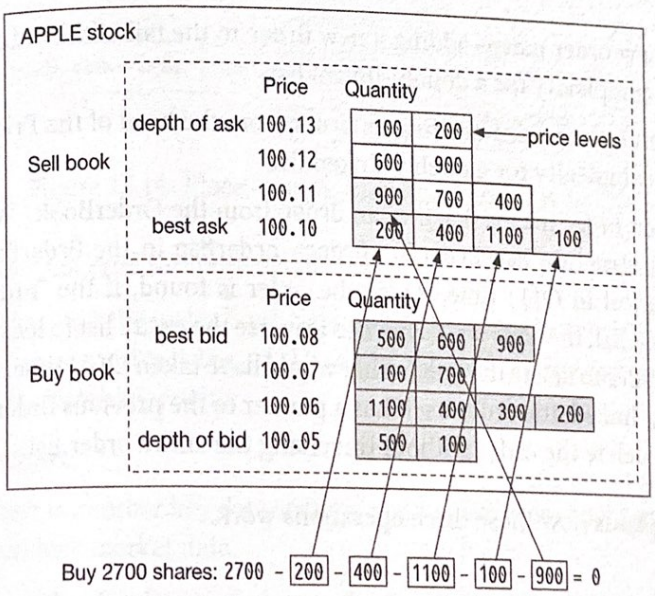

Figure 13.13: Limit order book illustrated

In the example above, there is a large market buy order for 2,700 shares of Apple. The buy order matches all the sell orders in the best ask queue and the first sell order in the 100.11 price queue. After fulfilling this large order, the bid/ask spread widens, and the price increases by one level (best ask is 100.11 now).

The following code snippet shows an implementation of the order book.

```
class PriceLevel{
  private Price limitPrice;
  private long totalVolume;
  private List<Order> orders;
}
class Book<Side> {
  private Side side;
  private Map<Price, PriceLevel> limitMap;
}
class OrderBook {
  private Book<Buy> buyBook;
  private Book<Sell> sellBook;
  private PriceLevel bestBid;
  private PriceLevel bestOffer;
  private Map<OrderID, Order> orderMap;
}
```

Does the code meet all the design requirements stated above? For example, when adding/canceling a limit order, is the time complexity $O(1)$? The answer is no since we are using a plain list here (`private List<Order> orders`). To have a more efficient order book, change the data structure of "orders" to a doubly-linked list so that the deletion type of operation (cancel and match) is also $O(1)$. Let's review how we achieve $O(1)$ time complexity for these operations:

1. Placing a new order means adding a new `Order` to the tail of the `PriceLevel`. This is $O(1)$ time complexity for a doubly-linked list.
2. Matching an order means deleting an `Order` from the head of the PriceLevel. This is $O(1)$ time complexity for a doubly-linked list.
3. Canceling an order means deleting an `Order` from the OrderBook. We leverage the helper data structure `Map<OrderID, Order> orderMap` in the `OrderBook` to find the `Order` to cancel in $O(1)$ time. Once the order is found, if the "orders" list was a singly-linked list, the code would have to traverse the entire list to locate the previous pointer in order to delete the order. That would have taken $O(n)$ time. Since the list is now doubly-linked, the order itself has a pointer to the previous order, which allows the code to delete the order without traversing the entire order list.

Figure 13.14 explains how these three operations work.

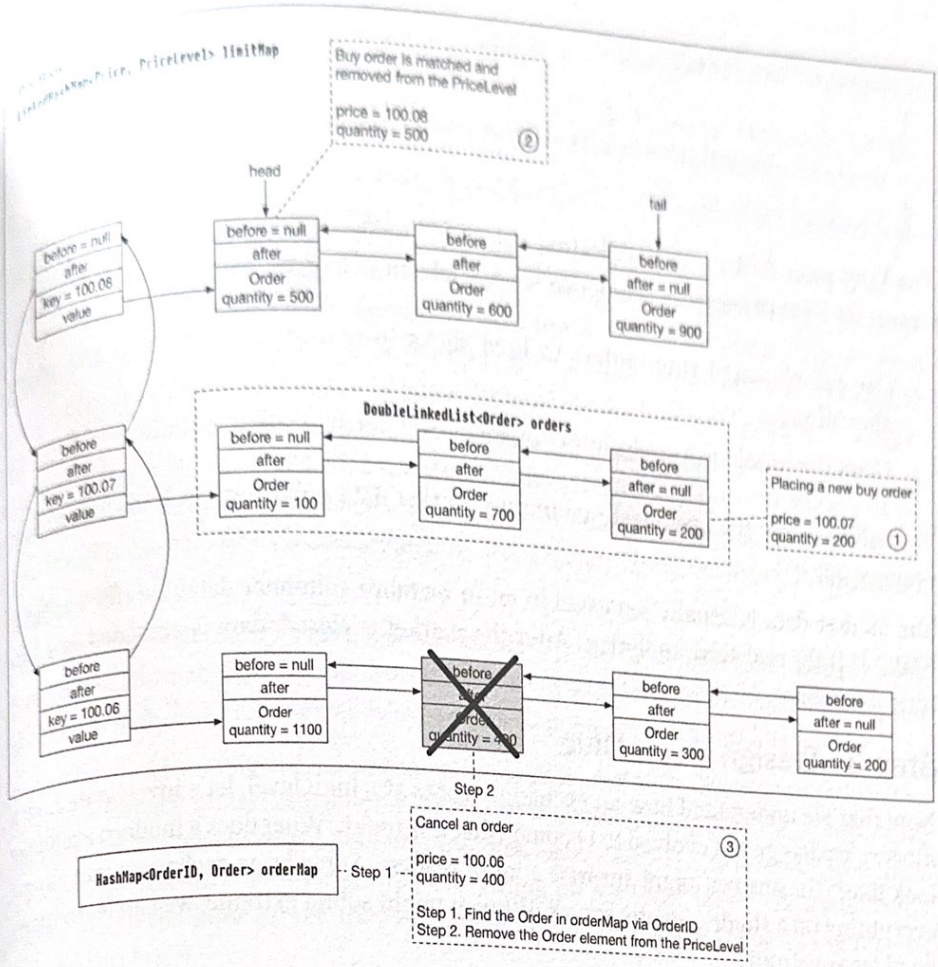

Figure 13.14: Place, match, and cancel an order in $O(1)$

See the reference material for more details [14].

It is worth noting that the order book data structure is also heavily used in the market data processor to reconstruct the L1, L2, and L3 data from the streams of executions generated by the matching engine.

Candlestick chart

Candlestick chart is another key data structure (alongside order book) in the market data processor to produce market data.

We model this with a `Candlestick` class and a `CandlestickChart` class. When the interval for the candlestick has elapsed, a new `Candlestick` class is instantiated for the next interval and added to the linked list in the `CandleStickChart` instance.

```
class Candlestick {
  private long openPrice;
  private long closePrice;
  private long highPrice;
  private long lowPrice;
  private long volume;
  private long timestamp;
```

```
  private int interval;
}
class CandlestickChart {
  private LinkedList<Candlestick> sticks;
}
```

Tracking price history in candlestick charts for many symbols at many time intervals consumes a lot of memory. How can we optimize it? Here are two ways:

1. Use pre-allocated ring buffers to hold sticks to reduce the number of new object allocations.
2. Limit the number of sticks in the memory and persist the rest to disk.

We will examine the optimizations in the "Market data publisher" section in deep dive on page 409.

The market data is usually persisted in an in-memory columnar database (for example, KDB [15]) for real-time analytics. After the market is closed, data is persisted in a historical database.

Step 3 - Design Deep Dive

Now that we understand how an exchange works at a high level, let's investigate how a modern exchange has evolved to become what it is today. What does a modern exchange look like? The answer might surprise a lot of readers. Some large exchanges run almost everything on a single gigantic server. While it might sound extreme, we can learn many good lessons from it.

Let's dive in.

Performance

As discussed in the non-functional requirements, latency is very important for an exchange. Not only does the average latency need to be low, but the overall latency must also be stable. A good measure for the level of stability is the 99th percentile latency.

Latency can be broken down into its components as shown in the formula below:

$$\text{Latency} = \sum \text{executionTimeAlongCriticalPath}$$

There are two ways to reduce latency:

1. Decrease the number of tasks on the critical path.
2. Shorten the time spent on each task:
   a. By reducing or eliminating network and disk usage
   b. By reducing execution time for each task

Let's look at the first point. As shown in the high-level design, the critical trading path includes the following:

gateway → order manager → sequencer → matching engine

The critical path only contains the necessary components, even logging is removed from the critical path to achieve low latency.

Now let's look at the second point. In the high-level design, the components on the critical path run on individual servers connected over the network. The round trip network latency is about 500 microseconds. When there are multiple components all communicating over the network on the critical path, the total network latency adds up to single-digit milliseconds. In addition, the sequencer is an event store that persists events to disk. Even assuming an efficient design that leverages the performance advantage of sequential writes, the latency of disk access still measures in tens of milliseconds. To learn more about network and disk access latency, see "Latency Numbers Every Programmer Should Know" [16].

Accounting for both network and disk access latency, the total end-to-end latency adds up to tens of milliseconds. While this number was respectable in the early days of the exchange, it is no longer sufficient as exchanges compete for ultra-low latency.

To stay ahead of the competition, exchanges over time evolve their design to reduce the end-to-end latency on the critical path to tens of microseconds, primarily by exploring options to reduce or eliminate network and disk access latency. A time-tested design eliminates the network hops by putting everything on the same server. When all components are on the same server, they can communicate via mmap [17] as an event store (more on this later).

Figure 13.15 shows a low-latency design with all the components on a single server:

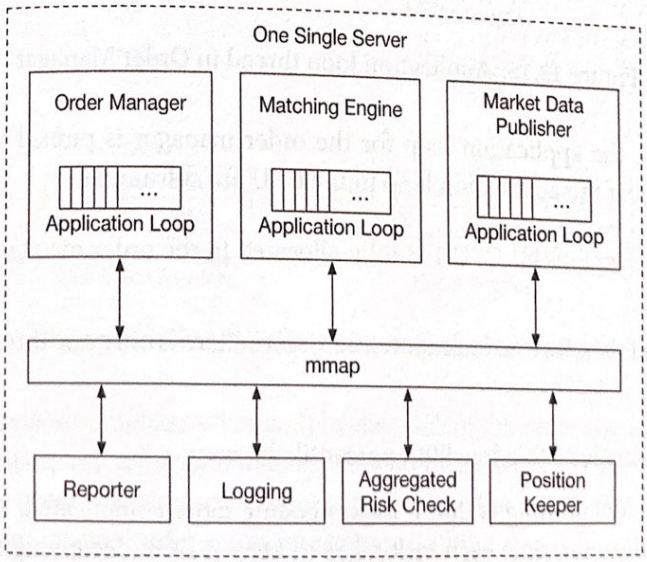

Figure 13.15: A low latency single server exchange design

There are a few interesting design decisions that are worth a closer look at.

Let's first focus on the application loops in the diagram above. An application loop is an interesting concept. It keeps polling for tasks to execute in a while loop and is the primary task execution mechanism. To meet the strict latency budget, only the most mission-critical tasks should be processed by the application loop. Its goal is to reduce the execution time for each component and to guarantee a highly predictable execution time (i.e., a low 99th percentile latency). Each box in the diagram represents a component. A component is a process on the server. To maximize CPU efficiency, each application loop (think of it as the main processing loop) is single-threaded, and the thread is pinned to a fixed CPU core. Using the order manager as an example, it looks like the following diagram (Figure 13.16).

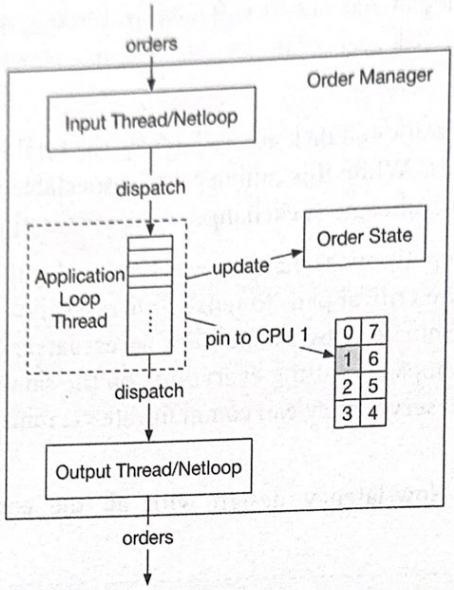

Figure 13.16: Application loop thread in Order Manager

In this diagram, the application loop for the order manager is pinned to CPU 1. The benefits of pinning the application loop to the CPU are substantial:

1. No context switch [18]. CPU 1 is fully allocated to the order manager's application loop.
2. No locks and therefore no lock contention, since there is only one thread that updates states.

Both of these contribute to a low 99th percentile latency.

The tradeoff of CPU pinning is that it makes coding more complicated. Engineers need to carefully analyze the time each task takes to keep it from occupying the application loop thread for too long, as it can potentially block subsequent tasks.

Next, let's focus our attention on the long rectangle labeled "mmap" at the center of

Figure 13.15: "mmap" refers to a POSIX-compliant UNIX system call named `mmap(2)` that maps a file into the memory of a process.

`mmap(2)` provides a mechanism for high-performance sharing of memory between processes. The performance advantage is compounded when the backing file is in `/dev/shm`. `/dev/shm` is a memory-backed file system. When `mmap(2)` is done over a file in `/dev/shm`, the access to the shared memory does not result in any disk access at all.

Modern exchanges take advantage of this to eliminate as much disk access from the critical path as possible. `mmap(2)` is used in the server to implement a message bus over which the components on the critical path communicate. The communication pathway has no network or disk access, and sending a message on this mmap message bus takes sub-microsecond. By leveraging mmap to build an event store, coupled with the event sourcing design paradigm which we will discuss next, modern exchanges can build low-latency microservices inside a server.

Event sourcing

We discussed event sourcing in the "Digital Wallet" chapter on page 341. Please refer to that chapter for an in-depth review of event sourcing.

The concept of event sourcing is not hard to understand. In a traditional application, states are persisted in a database. When something goes wrong, it is hard to trace the source of the issue. The database only keeps the current states, and there are no records of the events that have led to the current states.

In event sourcing, instead of storing the current states, it keeps an immutable log of all state-changing events. These events are the golden source of truth. See Figure 13.17 for a comparison.

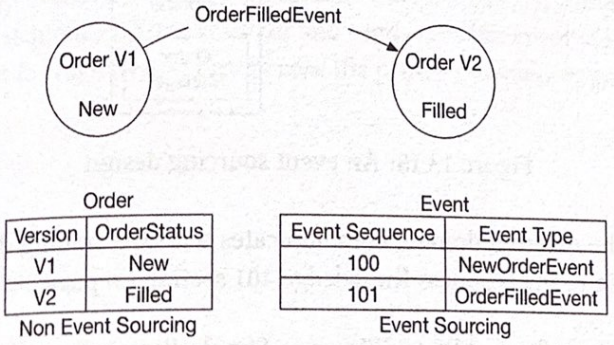

Figure 13.17: Non-event sourcing vs event sourcing

On the left is a classic database schema. It keeps track of the order status for an order, but it does not contain any information about how an order arrives at the current state. On the right is the event sourcing counterpart. It tracks all the events that change the order states, and it can recover order states by replaying all the events in sequence.

Figure 13.18 shows an event sourcing design using the mmap event store as a message bus. This looks very much like the Pub-Sub model in Kafka. In fact, if there is no strict

latency requirement, Kafka could be used.

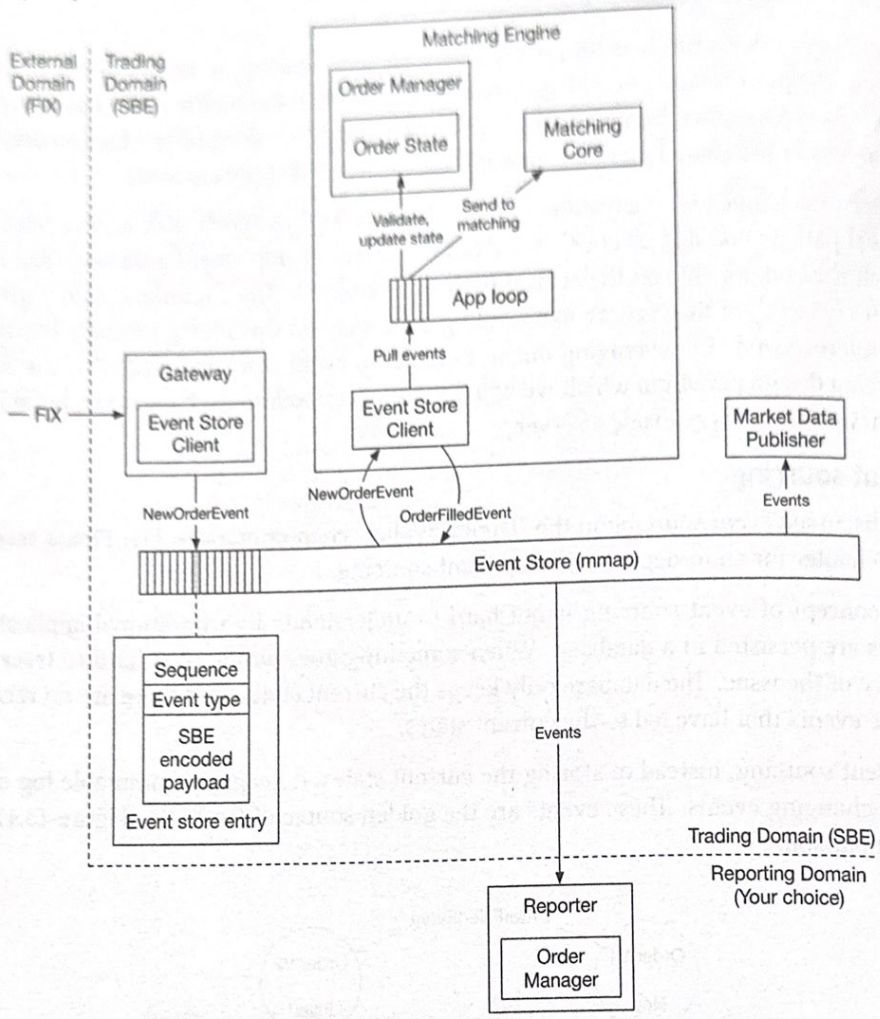

Figure 13.18: An event sourcing design

In the diagram, the external domain communicates with the trading domain using FIX that we introduced in the Business Knowledge 101 section on page 382.

- The gateway transforms FIX to "FIX over Simple Binary Encoding" (SBE) for fast and compact encoding and sends each order as a `NewOrderEvent` via the Event Store Client in a pre-defined format (see event store entry in the diagram).
- The order manager (embedded in the matching engine) receives the `NewOrderEvent` from the event store, validates it, and adds it to its internal order states. The order is then sent to the matching core.
- If the order gets matched, an `OrderFilledEvent` is generated and sent to the event store.