* Other components such as the market data processor and the reporter subscribe to the event store and process those events accordingly.

This design follows the high-level design closely, but there are some adjustments to make it work more efficiently in the event sourcing paradigm.

The first difference is the order manager. The order manager becomes a reusable library that is embedded in different components. It makes sense for this design because the states of the orders are important for multiple components. Having a centralized order manager for other components to update or query the order states would hurt latency, especially if those components are not on the critical trading path, as is the case for the reporter in the diagram. Although each component maintains the order states by itself, with event sourcing the states are guaranteed to be identical and replayable.

Another key difference is that the sequencer is nowhere to be seen. What happened to it?

With the event sourcing design, we have one single event store for all messages. Note that the event store entry contains a sequence field. This field is injected by the sequencer.

There is only one sequencer for each event store. It is a bad practice to have multiple sequencers, as they will fight for the right to write to the event store. In a busy system like an exchange, a lot of time would be wasted on lock contention. Therefore, the sequencer is a single writer which sequences the events before sending them to the event store. Unlike the sequencer in the high-level design which also functions as a message store, the sequencer here only does one simple thing and is super fast. Figure 13.19 shows a design for the sequencer in a memory-map (mmap) environment.

The sequencer pulls events from the ring buffer that is local to each component. For each event, it stamps a sequence ID on the event and sends it to the event store. We can have backup sequencers for high availability in case the primary sequencer goes down.

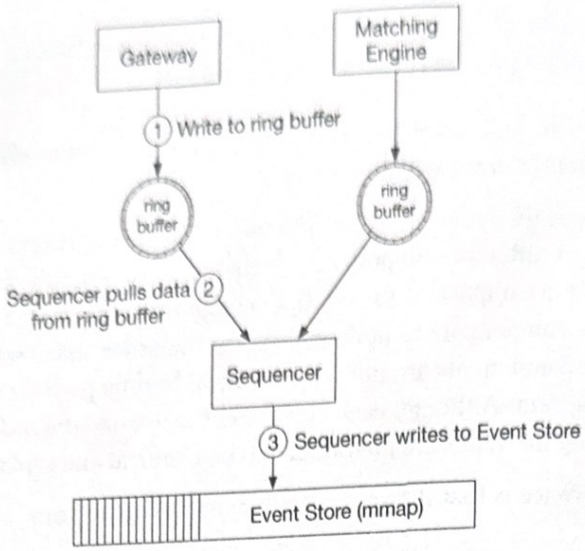


Figure 13.19: Sample design of Sequencer

High availability

For high availability, our design aims for 4 nines (99.99%). This means the exchange can only have 8.64 seconds of downtime per day. It requires almost immediate recovery if a service goes down.

To achieve high availability, consider the following:

- First, identify single-point-of-failures in the exchange architecture. For example, the failure of the matching engine could be a disaster for the exchange. Therefore, we set up redundant instances alongside the primary instance.
- Second, detection of failure and the decision to failover to the backup instance should be fast.

For stateless services such as the client gateway, they could easily be horizontally scaled by adding more servers. For stateful components, such as the order manager and matching engine, we need to be able to copy state data across replicas.

Figure 13.20 shows an example of how to copy data. The hot matching engine works as the primary instance, and the warm engine receives and processes the exact same events but does not send any event out onto the event store. When the primary goes down, the warm instance can immediately take over as the primary and send out events. When the warm secondary instance goes down, upon restart, it can always recover all the states from the event store. Event sourcing is a great fit for the exchange architecture. The inherent determinism makes state recovery easy and accurate.

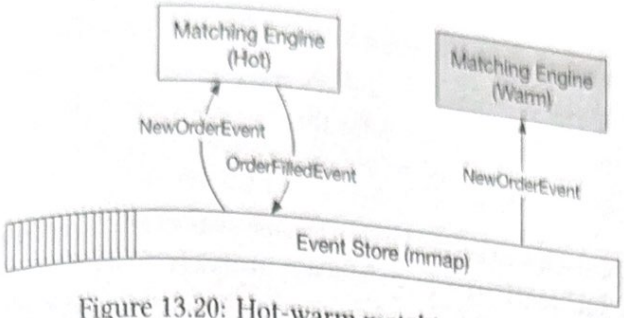


Figure 13.20: Hot-warm matching engine

We need to design a mechanism to detect potential problems in the primary. Besides normal monitoring of hardware and processes, we can also send heartbeats from the matching engine. If a heartbeat is not received in time, the matching engine might be experiencing problems.

The problem with this hot-warm design is that it only works within the boundary of a single server. To achieve high availability, we have to extend this concept across multiple machines or even across data centers. In this setting, an entire server is either hot or warm, and the entire event store is replicated from the hot server to all warm replicas. Replicating the entire event store across machines takes time. We could use reliable UDP [19] to efficiently broadcast the event messages to all warm servers. Refer to the design of Aeron [20] for an example.

In the next section, we discuss an improvement to the hot-warm design to achieve high availability.

Fault tolerance

The hot-warm design above is relatively simple. It works reasonably well, but what happens if the warm instances go down as well? This is a low probability but catastrophic event, so we should prepare for it.

This is a problem large tech companies face. They tackle it by replicating core data to data centers in multiple cities. It mitigates the risk of a natural disaster such as an earthquake or a large-scale power outage. To make the system fault-tolerant, we have to answer many questions:

1. If the primary instance goes down, how and when do we decide to failover to the backup instance?
2. How do we choose the leader among backup instances?
3. What is the recovery time needed (RTO - Recovery Time Objective)?
4. What functionalities need to be recovered (RPO - Recovery Point Objective)? Can our system operate under degraded conditions?

Let's answer these questions one by one.

First, we have to understand what "down" really means. This is not as straightforward as it seems. Consider these situations.

1. The system might send out false alarms, which cause unnecessary failovers.
2. Bugs in the code might cause the primary instance to go down. The same bug could bring down the backup instance after the failover. When all backup instances are knocked out by the bug, the system is no longer available.

These are tough problems to solve. Here are some suggestions. When we first release a new system, we might need to perform failovers manually. Only when we gather enough signals and operational experience and gain more confidence in the system do we automate the failure detection process. Chaos engineering [21] is a good practice to surface edge cases and gain operational experience faster.

Once the decision to failover is correctly made, how do we decide which server takes over? Fortunately, this is a well-understood problem. There are many battle-tested leader-election algorithms. We use Raft [22] as an example.

Figure 13.21 shows a Raft cluster with 5 servers with their own event stores. The current leader sends data to all the other instances (followers). The minimum number of votes required to perform an operation in Raft is $\frac{n}{2} + 1$, where n is the number of members in the cluster. In this example, the minimum is $\frac{5}{2} + 1 = 3$.

The following diagram (Figure 13.21) shows the followers receiving new events from the leader over RPC. The events are saved to the follower's own mmap event store.

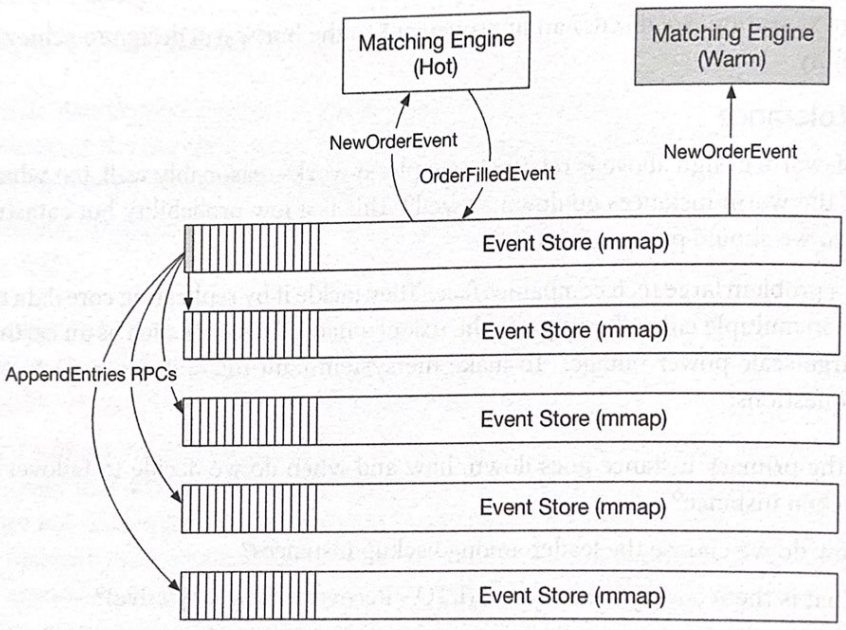


Figure 13.21: Event replication in Raft cluster

Let's briefly examine the leader election process. The leader sends heartbeat messages (`AppendEnties` with no content as shown in Figure 13.21) to its followers. If a follower has not received heartbeat messages for a period of time, it triggers an election timeout that initiates a new election. The first follower that reaches election timeout becomes a

candidate, and it asks the rest of the followers to vote (RequestVote). If the first follower receives a majority of votes, it becomes the new leader. If the first follower has a lower term value than the new node, it cannot be the leader. If multiple followers become candidates at the same time, it is called a "split vote". In this case, the election times out, and a new election is initiated. See Figure 13.22 for the explanation of "term". Time is divided into arbitrary intervals in Raft to represent normal operation and election.

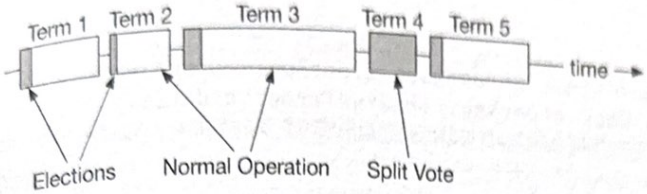


Figure 13.22: Raft terms (Source: [23])

Next, let's take a look at recovery time. Recovery Time Objective (RTO) refers to the amount of time an application can be down without causing significant damage to the business. For a stock exchange, we need to achieve a second-level RTO, which definitely requires automatic failover of services. To do this, we categorize services based on priority and define a degradation strategy to maintain a minimum service level.

Finally, we need to figure out the tolerance for data loss. Recovery Point Objective (RPO) refers to the amount of data that can be lost before significant harm is done to the business, i.e. the loss tolerance. In practice, this means backing up data frequently. For a stock exchange, data loss is not acceptable, so RPO is near zero. With Raft, we have many copies of the data. it guarantees that state consensus is achieved among cluster nodes. If the current leader crashes, the new leader should be able to function immediately.

Matching algorithms

Let's take a slight detour and dive into the matching algorithms. The pseudo-code below explains how matching works at a high level.

```
Context handleOrder(OrderBook orderBook, OrderEvent orderEvent) {
  if (orderEvent.getSequenceId() != nextSequence) {
    return Error(OUT_OF_ORDER, nextSequence);
  }

  if (!validateOrder(symbol, price, quantity)) {
    return ERROR(INVALID_ORDER, orderEvent);
  }

  Order order = createOrderFromEvent(orderEvent);
  switch (msgType):
    case NEW:
      return handleNew(orderBook, order);
    case CANCEL:
      return handleCancel(orderBook, order);
    default:
      return ERROR(INVALID_MSG_TYPE, msgType);
```

```
}

Context handleNew(OrderBook orderBook, Order order) {
  if (BUY.equals(order.side)) {
    return match(orderBook.sellBook, order);
  } else {
    return match(orderBook.buyBook, order);
  }
}

Context handleCancel(OrderBook orderBook, Order order) {
  if (!orderBook.orderMap.contains(order.orderId)) {
    return ERROR(CANNOT_CANCEL_ALREADY_MATCHED, order);
  }
  removeOrder(order);
  setOrderStatus(order, CANCELED);
  return SUCCESS(CANCEL_SUCCESS, order);
}

Context match(OrderBook book, Order order) {
  Quantity leavesQuantity = order.quantity - order.matchedQuantity;
  Iterator<Order> limitIter = book.limitMap.get(order.price).orders;
  while (limitIter.hasNext() && leavesQuantity > 0) {
    Quantity matched = min(limitIter.next.quantity, order.quantity);
    order.matchedQuantity += matched;
    leavesQuantity = order.quantity - order.matchedQuantity;
    remove(limitIter.next);
    generateMatchedFill();
  }
  return SUCCESS(MATCH_SUCCESS, order);
}
```

The pseudocode uses the FIFO (First In First Out) matching algorithm. The order that comes in first at a certain price level gets matched first, and the last one gets matched last.

There are many matching algorithms. These algorithms are commonly used in futures trading. For example, a FIFO with LMM (Lead Market Maker) algorithm allocates a certain quantity to the LMM based on a predefined ratio ahead of the FIFO queue, which the LMM firm negotiates with the exchange for the privilege. See more matching algorithms on the CME website [24]. The matching algorithms are used in many other scenarios. A typical one is a dark pool [25].

Determinism

There is both functional determinism and latency determinism. We have covered functional determinism in previous sections. The design choices we make, such as sequencer and event sourcing, guarantee that if the events are replayed in the same order, the results will be the same.

With functional determinism, the actual time when the event happens does not matter most of the time. What matters is the order of the events. In Figure 13.23, event timestamps from discrete uneven dots in the time dimension are converted to continuous dots,

and the time spent on replay/recovery can be greatly reduced.

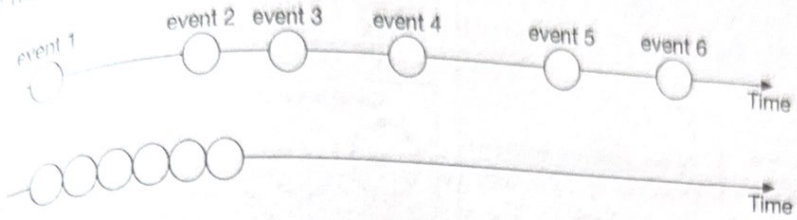


Figure 13.23: Time in event sourcing

Latency determinism means having almost the same latency through the system for each trade. This is key to the business. There is a mathematical way to measure this: the 99th percentile latency, or even more strictly, the 99.99th percentile latency. We can leverage HdrHistogram [26] to calculate latency. If the 99th percentile latency is low, the exchange offers stable performance across almost all the trades.

It is important to investigate large latency fluctuations. For example, in Java, safe points are often the cause. The HotSpot JVM [27] Stop-the-World garbage collection is a well-known example.

This concludes our deep dive on the critical trading path. In the remainder of this chapter, we take a closer look at some of the more interesting aspects of other parts of the exchange.

Market data publisher optimizations

As we can see from the matching algorithm, the L3 order book data gives us a better view of the market. We can get free one-day candlestick data from Google Finance, but it is expensive to get the more detailed L2/L3 order book data. Many hedge funds record the data themselves via the exchange real-time API to build their own candlestick charts and other charts for technical analysis.

The market data publisher (MDP) receives matched results from the matching engine and rebuilds the order book and candlestick charts based on that. It then publishes the data to the subscribers.

The order book rebuild is similar to the pseudocode mentioned in the matching algorithms section above. MDP is a service with many levels. For example, a retail client can only view 5 levels of L2 data by default and needs to pay extra to get 10 levels. MDP's memory cannot expand forever, so we need to have an upper limit on the candlesticks. Refer to the data models section for a review of the candlestick charts. The design of the MDP is in Figure 13.24.

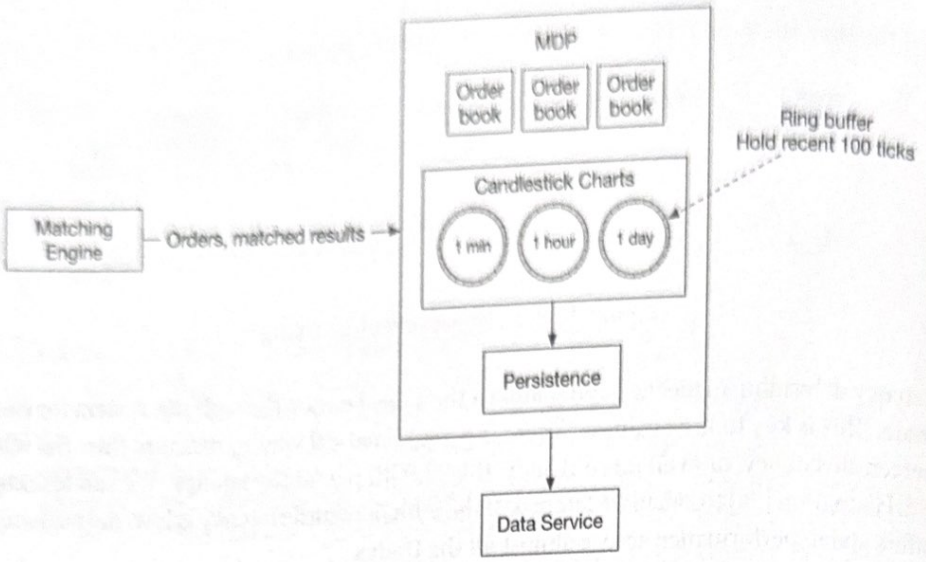


Figure 13.24: Market Data Publisher

This design utilizes ring buffers. A ring buffer, also called a circular buffer, is a fixed-size queue with the head connected to the tail. A producer continuously produces data and one or more consumers pull data off it. The space in a ring buffer is pre-allocated. There is no object creation or deallocation necessary. The data structure is also lock-free. There are other techniques to make the data structure even more efficient. For example, padding ensures that the ring buffer's sequence number is never in a cache line with anything else. Refer to [28] for more detail.

Distribution fairness of market data

In stock trading, having lower latency than others is like having an oracle that can see the future. For a regulated exchange, it is important to guarantee that all the receivers of market data get that data at the same time. Why is this important? For example, the MDP holds a list of data subscribers, and the order of the subscribers is decided by the order in which they connect to the publisher, with the first one always receiving data first. Guess what happens, then? Smart clients will fight to be the first on the list when the market opens.

There are some ways to mitigate this. Multicast using reliable UDP is a good solution to broadcast updates to many participants at once. The MDP could also assign a random order when the subscriber connects to it. We look at multicast in more detail.

Multicast

Data can be transported over the internet by three different types of protocols. Let's take a quick look.

1. Unicast: from one source to one destination.
2. Broadcast: from one source to an entire subnetwork.
3. Multicast: from one source to a set of hosts that can be on different subnetworks.

Multicast is a commonly-used protocol in exchange design. By configuring several receivers in the same multicast group, they will in theory receive data at the same time. However, UDP is an unreliable protocol and the datagram might not reach all the receivers. There are solutions to handle retransmission [29].

Colocation

While we are on the subject of fairness, it is a fact that a lot of exchanges offer colocation services, which put hedge funds or brokers' servers in the same data center as the exchange. The latency in placing an order to the matching engine is essentially proportional to the length of the cable. Colocation does not break the notion of fairness. It can be considered as a paid-for VIP service.

Network security

An exchange usually provides some public interfaces and a DDoS attack is a real challenge. Here are a few techniques to combat DDoS:

1. Isolate public services and data from private services, so DDoS attacks don't impact the most important clients. In case the same data is served, we can have multiple read-only copies to isolate problems.
2. Use a caching layer to store data that is infrequently updated. With good caching, most queries won't hit databases.
3. Harden URLs against DDoS attacks. For example, with an URL like `https://my.website.com/data?from=123&to=456`, an attacker can easily generate many different requests by changing the query string. Instead, URLs like this work better: `https://my.website.com/data/recent`. It can also be cached at the CDN layer.
4. An effective safelist/blocklist mechanism is needed. Many network gateway products provide this type of functionality.
5. Rate limiting is frequently used to defend against DDoS attacks.

Step 4 - Wrap Up

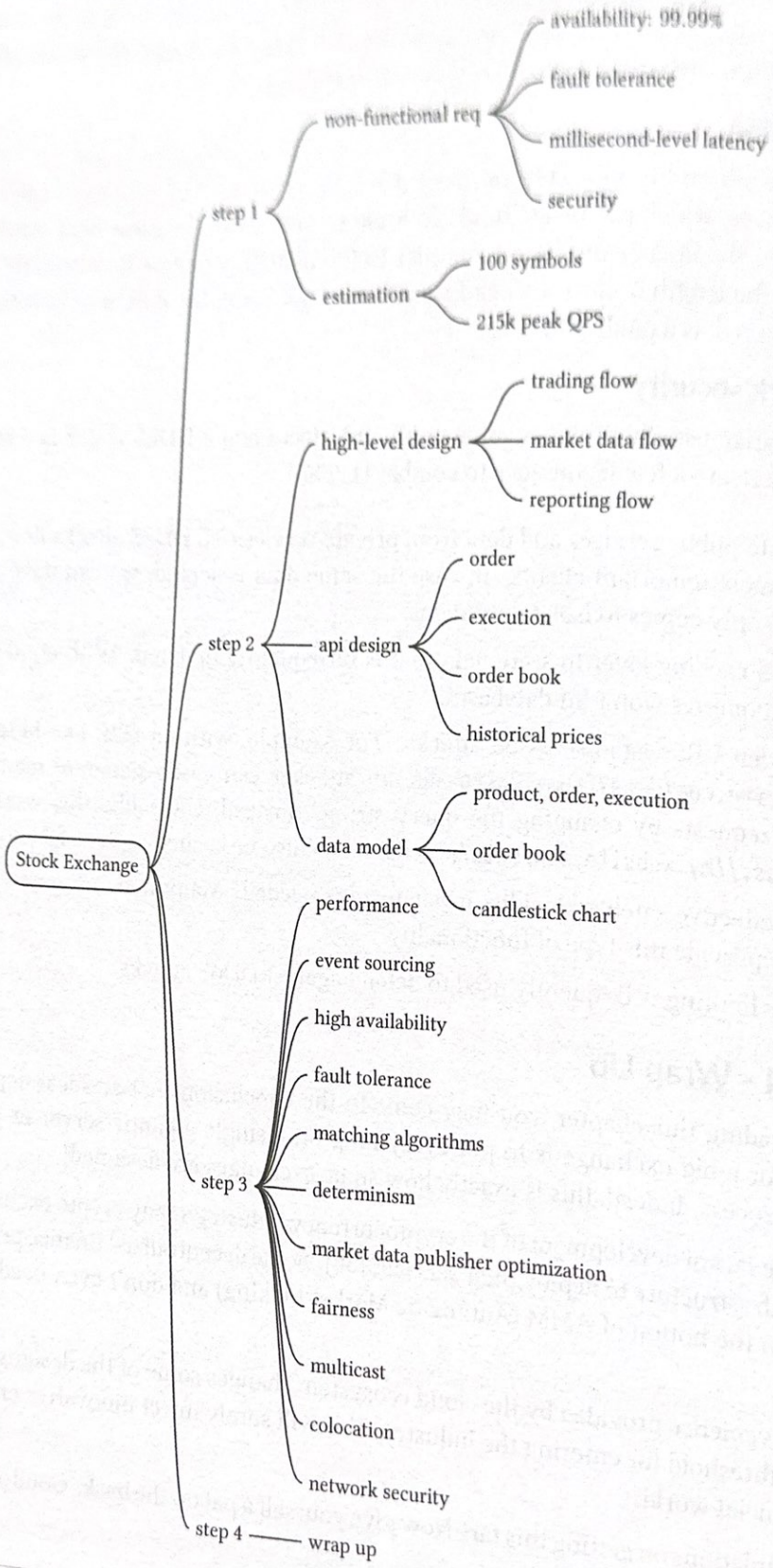
After reading this chapter, you may come to the conclusion that an ideal deployment model for a big exchange is to put everything on a single gigantic server or even one single process. Indeed, this is exactly how some exchanges are designed!

With the recent development of the cryptocurrency industry, many crypto exchanges use cloud infrastructure to deploy their services [30]. Some decentralized finance projects are based on the notion of AMM (Automatic Market Making) and don't even need an order book.

The convenience provided by the cloud ecosystem changes some of the designs and lowers the threshold for entering the industry. This will surely inject innovative energy into the financial world.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] LMAX exchange was famous for its open-source Disruptor. https://www.lmax.com/exchange.

[2] IEX attracts investors by "playing fair", also is the "Flash Boys Exchange". https://en.wikipedia.org/wiki/IEX.

[3] NYSE matched volume. https://www.nyse.com/markets/us-equity-volumes.

[4] HKEX daily trading volume. https://www.hkex.com.hk/Market-Data/Statistics/Consolidated-Reports/Securities-Statistics-Archive/Trading_Value_Volume_And_Number_Of_Deals?sc_lang=en#select1=0.

[5] All of the World's Stock Exchanges by Size. http://money.visualcapitalist.com/all-of-the-worlds-stock-exchanges-by-size/.

[6] Denial of service attack. https://en.wikipedia.org/wiki/Denial-of-service_attack.

[7] Market impact. https://en.wikipedia.org/wiki/Market_impact.

[8] Fix trading. https://www.fixtrading.org/.

[9] Event Sourcing. https://martinfowler.com/eaaDev/EventSourcing.html.

[10] CME Co-Location and Data Center Services. https://www.cmegroup.com/trading/colocation/co-location-services.html.

[11] Epoch. https://www.epoch101.com/.

[12] Order book. https://www.investopedia.com/terms/o/order-book.asp.

[13] Order book. https://en.wikipedia.org/wiki/Order_book.

[14] How to Build a Fast Limit Order Book. https://bit.ly/3ngMtEO.

[15] Developing with kdb+ and the q language. https://code.kx.com/q/.

[16] Latency Numbers Every Programmer Should Know. https://gist.github.com/jboner/2841832.

[17] mmap. https://en.wikipedia.org/wiki/Memory_map.

[18] Context switch. https://bit.ly/3pva7A6.

[19] Reliable User Datagram Protocol. https://en.wikipedia.org/wiki/Reliable_User_Datagram_Protocol.

[20] Aeron. https://github.com/real-logic/aeron/wiki/Design-Overview.

[21] Chaos engineering. https://en.wikipedia.org/wiki/Chaos_engineering.

[22] Raft. https://raft.github.io/.

[23] Designing for Understandability: the Raft Consensus Algorithm. https://raft.github.io/slides/uiuc2016.pdf.

[24] Supported Matching Algorithms. https://bit.ly/3aYoCEo.

[25] Dark pool. https://www.investopedia.com/terms/d/dark-pool.asp.

[26] HdrHistogram: A High Dynamic Range Histogram. http://hdrhistogram.org/.

[27] HotSpot (virtual machine). https://en.wikipedia.org/wiki/HotSpot_(virtual_machine).

[28] Cache line padding. https://bit.ly/3lZTFWz.

[29] NACK-Oriented Reliable Multicast. https://en.wikipedia.org/wiki/NACK-Oriented_Reliable_Multicast.

[30] AWS Coinbase Case Study. https://aws.amazon.com/solutions/case-studies/coinbase/.