Field	Description	Type
seller_account	Which seller will receive the money	string
amount	The transaction amount for the order	string
currency	The currency for the order	string (ISO 4217 [4])
payment_order_id	A globally unique ID for this payment	string

Table 11.2: payment_orders

Note that the payment_order_id is globally unique. When the payment executor sends a payment request to a third-party PSP, the payment_order_id is used by the PSP as the deduplication ID, also called the idempotency key.

You may have noticed that the data type of the "amount" field is "string," rather than "double". Double is not a good choice because:

1. Different protocols, software, and hardware may support different numeric precisions in serialization and deserialization. This difference might cause unintended rounding errors.
2. The number could be extremely big (for example, Japan's GDP is around 5×10^{14} yen for the calendar year 2020), or extremely small (for example, a satoshi of Bitcoin is 10^{-8}).

It is recommended to keep numbers in string format during transmission and storage. They are only parsed to numbers when used for display or calculation.

GET /v1/payments/{:id}

This endpoint returns the execution status of a single payment order based on payment_order_id.

The payment API mentioned above is similar to the API of some well-known PSPs. If you are interested in a more comprehensive view of payment APIs, check out Stripe's API documentation [5].

The data model for payment service

We need two tables for the payment service: payment event and payment order. When we select a storage solution for a payment system, performance is usually not the most important factor. Instead, we focus on the following:

1. Proven stability. Whether the storage system has been used by other big financial firms for many years (for example more than 5 years) with positive feedback.
2. The richness of supporting tools, such as monitoring and investigation tools.
3. Maturity of the database administrator (DBA) job market. Whether we can recruit experienced DBAs is a very important factor to consider.

Usually, we prefer a traditional relational database with ACID transaction support over NoSQL/NewSQL.

The payment event table contains detailed payment event information. This is what it looks like:

Name	Type
checkout_id	string **PK**
buyer_info	string
seller_info	string
credit_card_info	depends on the card provider
is_payment_done	boolean

Table 11.3: Payment event

The payment order table stores the execution status of each payment order. This is what it looks like:

Name	Type
payment_order_id	String **PK**
buyer_account	string
amount	string
currency	string
checkout_id	string **FK**
payment_order_status	string
ledger_updated	boolean
wallet_updated	boolean

Table 11.4: Payment order

Before we dive into the tables, let's take a look at some background information.

- The checkout_id is the foreign key. A single checkout creates a payment event that may contain several payment orders.
- When we call a third-party PSP to deduct money from the buyer's credit card, the money is not directly transferred to the seller. Instead, the money is transferred to the e-commerce website's bank account. This process is called pay-in. When the pay-out condition is satisfied, such as when the products are delivered, the seller initiates a pay-out. Only then is the money transferred from the e-commerce website's bank account to the seller's bank account. Therefore, during the pay-in flow, we only need the buyer's card information, not the seller's bank account information.

In the payment order table (Table 11.4), payment_order_status is an enumerated type (enum) that keeps the execution status of the payment order. Execution status includes NOT_STARTED, EXECUTING, SUCCESS, FAILED. The update logic is:

1. The initial status of payment_order_status is NOT_STARTED.

2. When the payment service sends the payment order to the payment executor, the payment_order_status is EXECUTING.
3. The payment service updates the payment_order_status to SUCCESS or FAILED depending on the response of the payment executor.

Once the payment_order_status is SUCCESS, the payment service calls the wallet service to update the seller balance and update the wallet_updated field to TRUE. Here we simplify the design by assuming wallet updates always succeed.

Once it is done, the next step for the payment service is to call the ledger service to update the ledger database by updating the ledger_updated field to TRUE.

When all payment orders under the same checkout_id are processed successfully, the payment service updates the is_payment_done to TRUE in the payment event table. A scheduled job usually runs at a fixed interval to monitor the status of the in-flight payment orders. It sends an alert when a payment order does not finish within a threshold so that engineers can investigate it.

Double-entry ledger system

There is a very important design principle in the ledger system: the double-entry principle (also called double-entry accounting/bookkeeping [6]). Double-entry system is fundamental to any payment system and is key to accurate bookkeeping. It records every payment transaction into two separate ledger accounts with the same amount. One account is debited and the other is credited with the same amount (Table 11.5).

Account	Debit	Credit
buyer	$1	
seller		$1

Table 11.5: Double-entry system

The double-entry system states that the sum of all the transaction entries must be 0. One cent lost means someone else gains a cent. It provides end-to-end traceability and ensures consistency throughout the payment cycle. To find out more about implementing the double-entry system, see Square's engineering blog about immutable double-entry accounting database service [7].

Hosted payment page

Most companies prefer not to store credit card information internally because if they do, they have to deal with complex regulations such as Payment Card Industry Data Security Standard (PCI DSS) [8] in the United States. To avoid handling credit card information, companies use hosted credit card pages provided by PSPs. For websites, it is a widget or an iframe, while for mobile applications, it may be a pre-built page from the payment SDK. Figure 11.3 illustrates an example of the checkout experience with PayPal integration. The key point here is that the PSP provides a hosted payment page that captures the customer card information directly, rather than relying on our payment service.

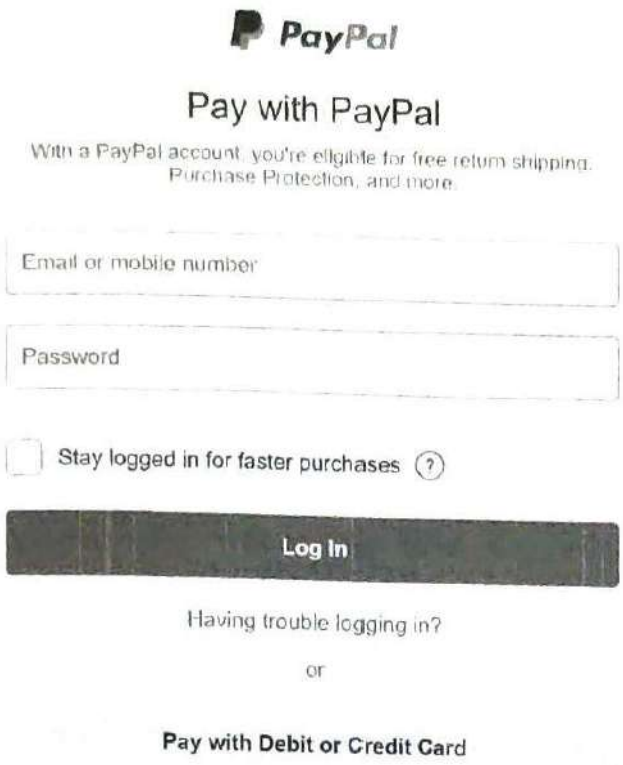


Figure 11.3: Hosted pay with PayPal page

Pay-out flow

The components of the pay-out flow are very similar to the pay-in flow. One difference is that instead of using PSP to move money from the buyer's credit card to the e-commerce website's bank account, the pay-out flow uses a third-party pay-out provider to move money from the e-commerce website's bank account to the seller's bank account.

Usually, the payment system uses third-party account payable providers like Tipalti [9] to handle pay-outs. There are a lot of bookkeeping and regulatory requirements with pay-outs as well.

Step 3 - Design Deep Dive

In this section, we focus on making the system faster, more robust, and secure. In a distributed system, errors and failures are not only inevitable but common. For example, what happens if a customer pressed the "pay" button multiple times? Will they be charged multiple times? How do we handle payment failures caused by poor network connections? In this section, we dive deep into several key topics.

- PSP integration
- Reconciliation
- Handling payment processing delays

- Communication among internal services
- Handling failed payments
- Exact-once delivery
- Consistency
- Security

PSP integration

If the payment system can directly connect to banks or card schemes such as Visa or MasterCard, payment can be made without a PSP. These direct connections are uncommon and highly specialized. They are usually reserved for really large companies that can justify such an investment. For most companies, the payment system integrates with a PSP instead, in one of two ways:

1. If a company can safely store sensitive payment information and chooses to do so, PSP can be integrated using API. The company is responsible for developing the payment web pages, collecting and storing sensitive payment information. PSP is responsible for connecting to banks or card schemes.
2. If a company chooses not to store sensitive payment information due to complex regulations and security concerns, PSP provides a hosted payment page to collect card payment details and securely store them in PSP. This is the approach most companies take.

We use Figure 11.4 to explain how the hosted payment page works in detail.

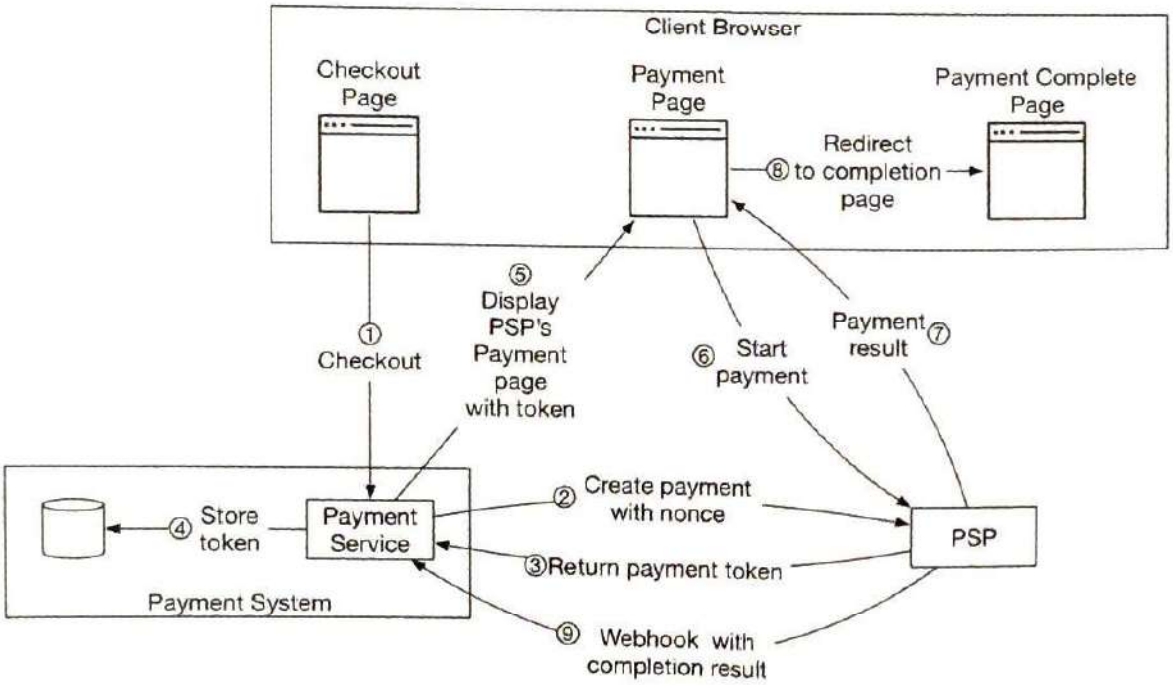


Figure 11.4: Hosted payment flow

We omitted the payment executor, ledger, and wallet in Figure 11.4 for simplicity. The payment service orchestrates the whole payment process.

1. The user clicks the "checkout" button in the client browser. The client calls the payment service with the payment order information.
2. After receiving the payment order information, the payment service sends a payment registration request to the PSP. This registration request contains payment information, such as the amount, currency, expiration date of the payment request, and the redirect URL. Because a payment order should be registered only once, there is a UUID field to ensure the exactly-once registration. This UUID is also called nonce [10]. Usually, this UUID is the ID of the payment order.
3. The PSP returns a token back to the payment service. A token is a UUID on the PSP side that uniquely identifies the payment registration. We can examine the payment registration and the payment execution status later using this token.
4. The payment service stores the token in the database before calling the PSP-hosted payment page.
5. Once the token is persisted, the client displays a PSP-hosted payment page. Mobile applications usually use the PSP's SDK integration for this functionality. Here we use Stripe's web integration as an example (Figure 11.5). Stripe provides a JavaScript library that displays the payment UI, collects sensitive payment information, and calls the PSP directly to complete the payment. Sensitive payment information is collected by Stripe. It never reaches our payment system. The hosted payment page usually needs two pieces of information:
   (a) The token we received in step 4. The PSP's javascript code uses the token to retrieve detailed information about the payment request from the PSP's backend. One important piece of information is how much money to collect.
   (b) Another important piece of information is the redirect URL. This is the web page URL that is called when the payment is complete. When the PSP's JavaScript finishes the payment, it redirects the browser to the redirect URL. Usually, the redirect URL is an e-commerce web page that shows the status of the checkout. Note that the redirect URL is different from the webhook [11] URL in step 9.

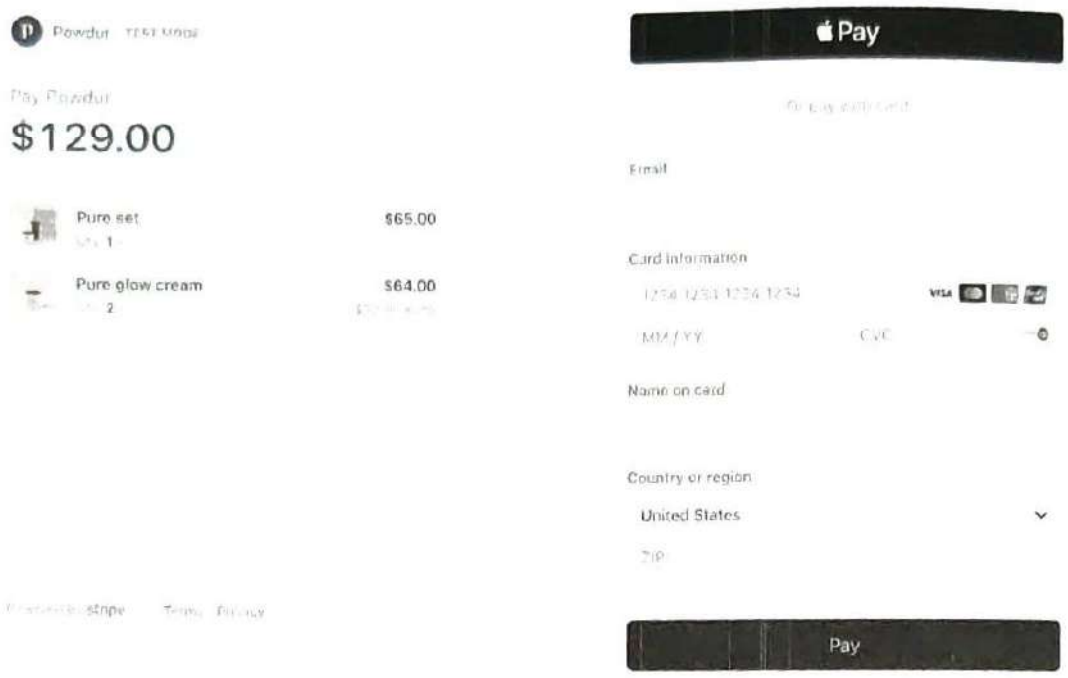


Figure 11.5: Hosted payment page by Stripe

6. The user fills in the payment details on the PSP's web page, such as the credit card number, holder's name, expiration date, etc, then clicks the pay button. The PSP starts the payment processing.
7. The PSP returns the payment status.
8. The web page is now redirected to the redirect URL. The payment status that is received in step 7 is typically appended to the URL. For example, the full redirect URL could be [12]: `https://your-company.com/?tokenID=JIOUIQ123NSF&payResult=X324FSa`
9. Asynchronously, the PSP calls the payment service with the payment status via a webhook. The webhook is an URL on the payment system side that was registered with the PSP during the initial setup with the PSP. When the payment system receives payment events through the webhook, it extracts the payment status and updates the `payment_order_status` field in the Payment Order database table.

So far, we explained the happy path of the hosted payment page. In reality, the network connection could be unreliable and all 9 steps above could fail. Is there any systematic way to handle failure cases? The answer is reconciliation.

Reconciliation

When system components communicate asynchronously, there is no guarantee that a message will be delivered, or a response will be returned. This is very common in the payment business, which often uses asynchronous communication to increase system performance. External systems, such as PSPs or banks, prefer asynchronous communi-

cation as well. So how can we ensure correctness in this case?

The answer is reconciliation. This is a practice that periodically compares the states among related services in order to verify that they are in agreement. It is usually the last line of defense in the payment system.

Every night the PSP or banks send a settlement file to their clients. The settlement file contains the balance of the bank account, together with all the transactions that took place on this bank account during the day. The reconciliation system parses the settlement file and compares the details with the ledger system. Figure 11.6 below shows where the reconciliation process fits in the system.

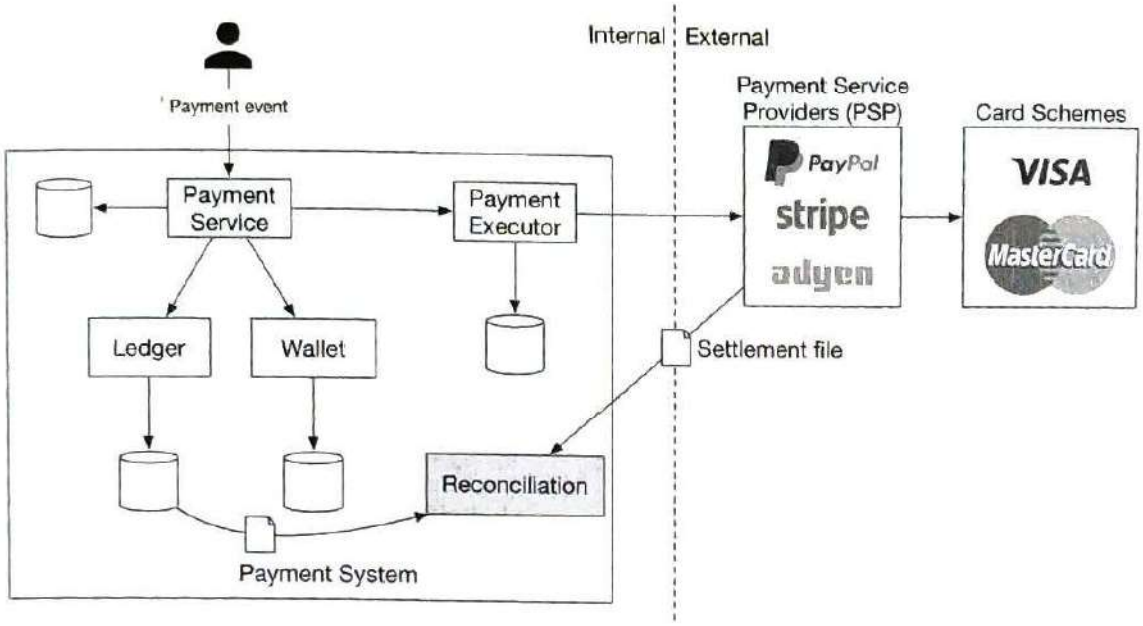


Figure 11.6: Reconciliation

Reconciliation is also used to verify that the payment system is internally consistent. For example, the states in the ledger and wallet might diverge and we could use the reconciliation system to detect any discrepancy.

To fix mismatches found during reconciliation, we usually rely on the finance team to perform manual adjustments. The mismatches and adjustments are usually classified into three categories:

1. The mismatch is classifiable and the adjustment can be automated. In this case, we know the cause of the mismatch, how to fix it, and it is cost-effective to write a program to automate the adjustment. Engineers can automate both the mismatch classification and adjustment.
2. The mismatch is classifiable, but we are unable to automate the adjustment. In this case, we know the cause of the mismatch and how to fix it, but the cost of writing an auto adjustment program is too high. The mismatch is put into a job queue and the finance team fixes the mismatch manually.

3. The mismatch is unclassifiable. In this case, we do not know how the mismatch happens. The mismatch is put into a special job queue. The finance team investigates it manually.

Handling payment processing delays

As discussed previously, an end-to-end payment request flows through many components and involves both internal and external parties. While in most cases a payment request would complete in seconds, there are situations where a payment request would stall and sometimes take hours or days before it is completed or rejected. Here are some examples where a payment request could take longer than usual:

- The PSP deems a payment request high risk and requires a human to review it.
- A credit card requires extra protection like 3D Secure Authentication [13] which requires extra details from a card holder to verify a purchase.

The payment service must be able to handle these payment requests that take a long time to process. If the buy page is hosted by an external PSP, which is quite common these days, the PSP would handle these long-running payment requests in the following ways:

- The PSP would return a pending status to our client. Our client would display that to the user. Our client would also provide a page for the customer to check the current payment status.
- The PSP tracks the pending payment on our behalf, and notifies the payment service of any status update via the webhook the payment service registered with the PSP.

When the payment request is finally completed, the PSP calls the registered webhook mentioned above. The payment service updates its internal system and completes the shipment to the customer.

Alternatively, instead of updating the payment service via a webhook, some PSP would put the burden on the payment service to poll the PSP for status updates on any pending payment requests.

Communication among internal services

There are two types of communication patterns that internal services use to communicate: synchronous vs asynchronous. Both are explained below.

Synchronous communication

Synchronous communication like HTTP works well for small-scale systems, but its shortcomings become obvious as the scale increases. It creates a long request and response cycle that depends on many services. The drawbacks of this approach are:

- Low performance. If any one of the services in the chain doesn't perform well, the whole system is impacted.

- Poor failure isolation. If PSPs or any other services fail, the client will no longer receive a response.
- Tight coupling. The request sender needs to know the recipient.
- Hard to scale. Without using a queue to act as a buffer, it's not easy to scale the system to support a sudden increase in traffic.

Asynchronous communication

Asynchronous communication can be divided into two categories:

- Single receiver: each request (message) is processed by one receiver or service. It's usually implemented via a shared message queue. The message queue can have multiple subscribers, but once a message is processed, it gets removed from the queue. Let's take a look at a concrete example. In Figure 11.7, service A and service B both subscribe to a shared message queue. When m1 and m2 are consumed by service A and service B respectively, both messages are removed from the queue as shown in Figure 11.8.

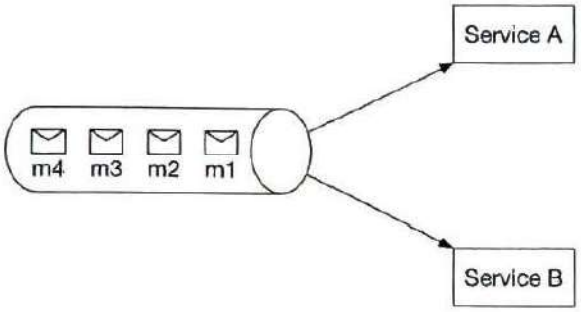


Figure 11.7: Message queue

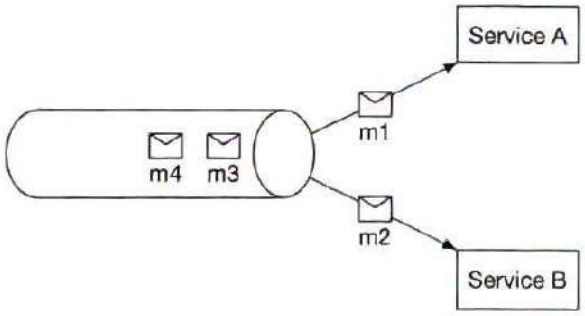


Figure 11.8: Single receiver for each message

- Multiple receivers: each request (message) is processed by multiple receivers or services. Kafka works well here. When consumers receive messages, they are not removed from Kafka. The same message can be processed by different services. This model maps well to the payment system, as the same request might trigger multiple side effects such as sending push notifications, updating financial reporting, ana-

lytics, etc. An example is illustrated in Figure 11.9. Payment events are published to Kafka and consumed by different services such as the payment system, analytics service, and billing service.

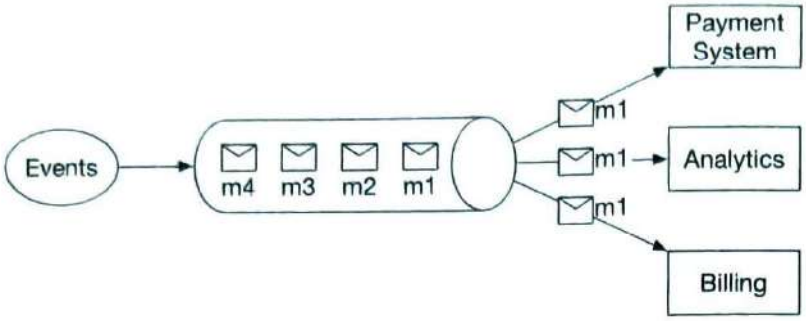


Figure 11.9: Multiple receivers for the same message

Generally speaking, synchronous communication is simpler in design, but it doesn't allow services to be autonomous. As the dependency graph grows, the overall performance suffers. Asynchronous communication trades design simplicity and consistency for scalability and failure resilience. For a large-scale payment system with complex business logic and a large number of third-party dependencies, asynchronous communication is a better choice.

Handling failed payments

Every payment system has to handle failed transactions. Reliability and fault tolerance are key requirements. We review some of the techniques for tackling those challenges.

Tracking payment state

Having a definitive payment state at any stage of the payment cycle is crucial. Whenever a failure happens, we can determine the current state of a payment transaction and decide whether a retry or refund is needed. The payment state can be persisted in an append-only database table.

Retry queue and dead letter queue

To gracefully handle failures, we utilize the retry queue and dead letter queue, as shown in Figure 11.10.

- Retry queue: retryable errors such as transient errors are routed to a retry queue.
- Dead letter queue [14]: if a message fails repeatedly, it eventually lands in the dead letter queue. A dead letter queue is useful for debugging and isolating problematic messages for inspection to determine why they were not processed successfully.

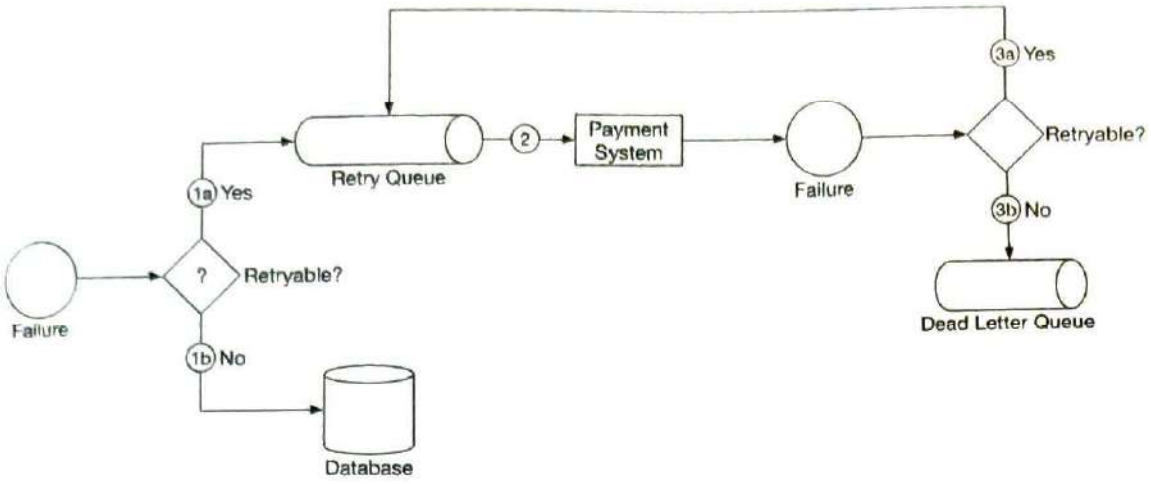


Figure 11.10: Handle failed payments

1. Check whether the failure is retryable.
   (a) Retryable failures are routed to a retry queue.
   (b) For non-retryable failures such as invalid input, errors are stored in a database.
2. The payment system consumes events from the retry queue and retries failed payment transactions.
3. If the payment transaction fails again:
   (a) If the retry count doesn't exceed the threshold, the event is routed to the retry queue.
   (b) If the retry count exceeds the threshold, the event is put in the dead letter queue. Those failed events might need to be investigated.

If you are interested in a real-world example of using those queues, take a look at Uber's payment system that utilizes Kafka to meet the reliability and fault-tolerance requirements [15].

Exactly-once delivery

One of the most serious problems a payment system can have is to double charge a customer. It is important to guarantee in our design that the payment system executes a payment order exactly-once [16].

At first glance, exactly-once delivery seems very hard to tackle, but if we divide the problem into two parts, it is much easier to solve. Mathematically, an operation is executed exactly-once if:

1. It is executed at-least-once.
2. At the same time, it is executed at-most-once.

We will explain how to implement at-least-once using retry, and at-most-once using idempotency check.

Retry

Occasionally, we need to retry a payment transaction due to network errors or timeout. Retry provides the at-least-once guarantee. For example, as shown in Figure 11.11, where the client tries to make a $10 payment, but the payment request keeps failing due to a poor network connection. In this example, the network eventually recovered and the request succeeded at the fourth attempt.

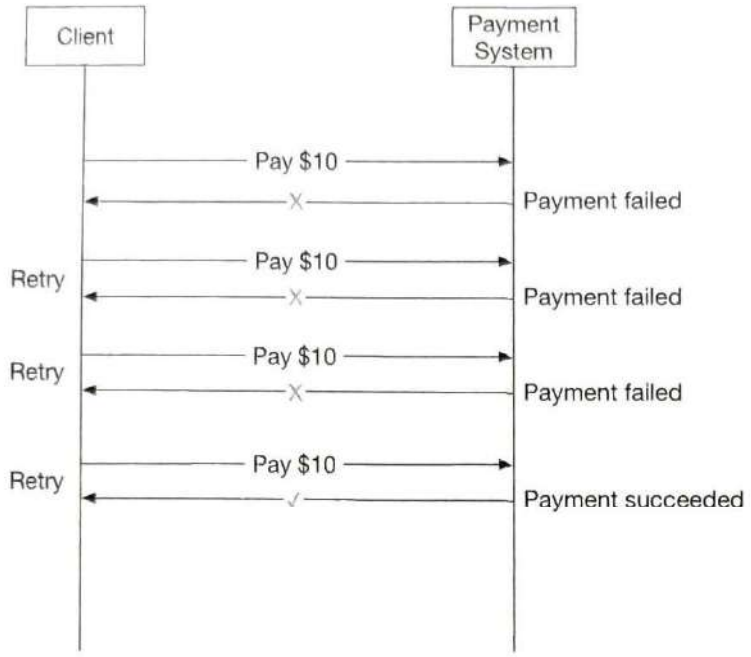


Figure 11.11: Retry

Deciding the appropriate time intervals between retries is important. Here are a few common retry strategies.

- Immediate retry: client immediately resends a request.
- Fixed intervals: wait a fixed amount of time between the time of the failed payment and a new retry attempt.
- Incremental intervals: client waits for a short time for the first retry, and then incrementally increases the time for subsequent retries.
- Exponential backoff [17]: double the waiting time between retries after each failed retry. For example, when a request fails for the first time, we retry after 1 second; if it fails a second time, we wait 2 seconds before the next retry; if it fails a third time, we wait 4 seconds before another retry.
- Cancel: the client can cancel the request. This is a common practice when the failure is permanent or repeated requests are unlikely to be successful.

Determining the appropriate retry strategy is difficult. There is no "one size fits all" solution. As a general guideline, use exponential backoff if the network issue is unlikely to be resolved in a short amount of time. Overly aggressive retry strategies waste computing

resources and can cause service overload. A good practice is to provide an error code with a `Retry-After` header.

A potential problem of retrying is double payments. Let us take a look at two scenarios.

Scenario 1: The payment system integrates with PSP using a hosted payment page, and the client clicks the pay button twice.

Scenario 2: The payment is successfully processed by the PSP, but the response fails to reach our payment system due to network errors. The user clicks the "pay" button again or the client retries the payment.

In order to avoid double payment, the payment has to be executed at-most-once. This at-most-once guarantee is also called idempotency.

Idempotency

Idempotency is key to ensuring the at-most-once guarantee. According to Wikipedia, "idempotence is the property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application" [18]. From an API standpoint, idempotency means clients can make the same call repeatedly and produce the same result.

For communication between clients (web and mobile applications) and servers, an idempotency key is usually a unique value that is generated by the client and expires after a certain period of time. A UUID is commonly used as an idempotency key and it is recommended by many tech companies such as Stripe [19] and PayPal [20]. To perform an idempotent payment request, an idempotency key is added to the HTTP header: `<idempotency-key: key_value>`.

Now that we understand the basics of idempotency, let's take a look at how it helps to solve the double payment issues mentioned above.

Scenario 1: what if a customer clicks the "pay" button quickly twice?

In Figure 11.12, when a user clicks "pay," an idempotency key is sent to the payment system as part of the HTTP request. In an e-commerce website, the idempotency key is usually the ID of the shopping cart right before the checkout.

For the second request, it's treated as a retry because the payment system has already seen the idempotency key. When we include a previously specified idempotency key in the request header, the payment system returns the latest status of the previous request.

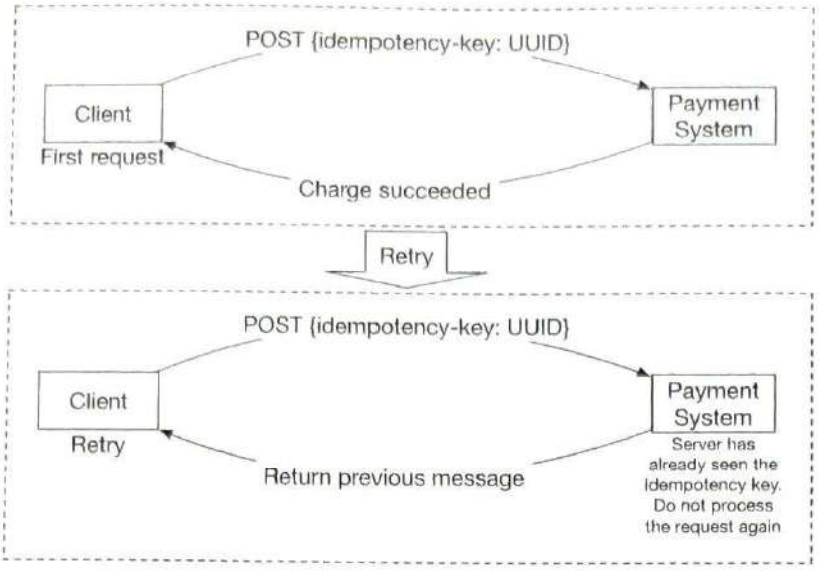


Figure 11.12: Idempotency

If multiple concurrent requests are detected with the same idempotency key, only one request is processed and the others receive the `429 Too Many Requests` status code.

To support idempotency, we can use the database's unique key constraint. For example, the primary key of the database table is served as the idempotency key. Here is how it works:

1. When the payment system receives a payment, it tries to insert a row into the database table.
2. A successful insertion means we have not seen this payment request before.
3. If the insertion fails because the same primary key already exists, it means we have seen this payment request before. The second request will not be processed.

Scenario 2: The payment is successfully processed by the PSP, but the response fails to reach our payment system due to network errors. Then the user clicks the "pay" button again.

As shown in Figure 11.4 (step 2 and step 3), the payment service sends the PSP a nonce and the PSP returns a corresponding token. The nonce uniquely represents the payment order, and the token uniquely maps to the nonce. Therefore, the token uniquely maps to the payment order.

When the user clicks the "pay" button again, the payment order is the same, so the token sent to the PSP is the same. Because the token is used as the idempotency key on the PSP side, it is able to identify the double payment and return the status of the previous execution.

Consistency

Several stateful services are called in a payment execution:

1. The payment service keeps payment-related data such as nonce, token, payment order, execution status, etc.
2. The ledger keeps all accounting data.
3. The wallet keeps the account balance of the merchant.
4. The PSP keeps the payment execution status.
5. Data might be replicated among different database replicas to increase reliability.

In a distributed environment, the communication between any two services can fail, causing data inconsistency. Let's take a look at some techniques to resolve data inconsistency in a payment system.

To maintain data consistency between internal services, ensuring exactly-once processing is very important.

To maintain data consistency between the internal service and external service (PSP), we usually rely on idempotency and reconciliation. If the external service supports idempotency, we should use the same idempotency key for payment retry operations. Even if an external service supports idempotent APIs, reconciliation is still needed because we shouldn't assume the external system is always right.

If data is replicated, replication lag could cause inconsistent data between the primary database and the replicas. There are generally two options to solve this:

1. Serve both reads and writes from the primary database only. This approach is easy to set up, but the obvious drawback is scalability. Replicas are used to ensure data reliability, but they don't serve any traffic, which wastes resources.
2. Ensure all replicas are always in-sync. We could use consensus algorithms such as Paxos [21] and Raft [22], or use consensus-based distributed databases such as YugabyteDB [23] or CockroachDB [24].

Payment security

Payment security is very important. In the final part of this system design, we briefly cover a few techniques for combating cyberattacks and card thefts.

Problem	Solution
Request/response eavesdropping	Use HTTPS
Data tampering	Enforce encryption and integrity monitoring
Man-in-the-middle attack	Use SSL with certificate pinning
Data loss	Database replication across multiple regions and take snapshots of data
Distributed denial-of-service attack (DDoS)	Rate limiting and firewall [25]
Card theft	Tokenization. Instead of using real card numbers, tokens are stored and used for payment
PCI compliance	PCI DSS is an information security standard for organizations that handle branded credit cards
Fraud	Address verification, card verification value (CVV), user behavior analysis, etc. [26] [27]

Table 11.6: Payment security

Step 4 - Wrap Up

In this chapter, we investigated the pay-in flow and pay-out flow. We went into great depth about retry, idempotency, and consistency. Payment error handling and security are also covered at the end of the chapter.

A payment system is extremely complex. Even though we have covered many topics, there are still more worth mentioning. The following is a representative but not an exhaustive list of relevant topics.

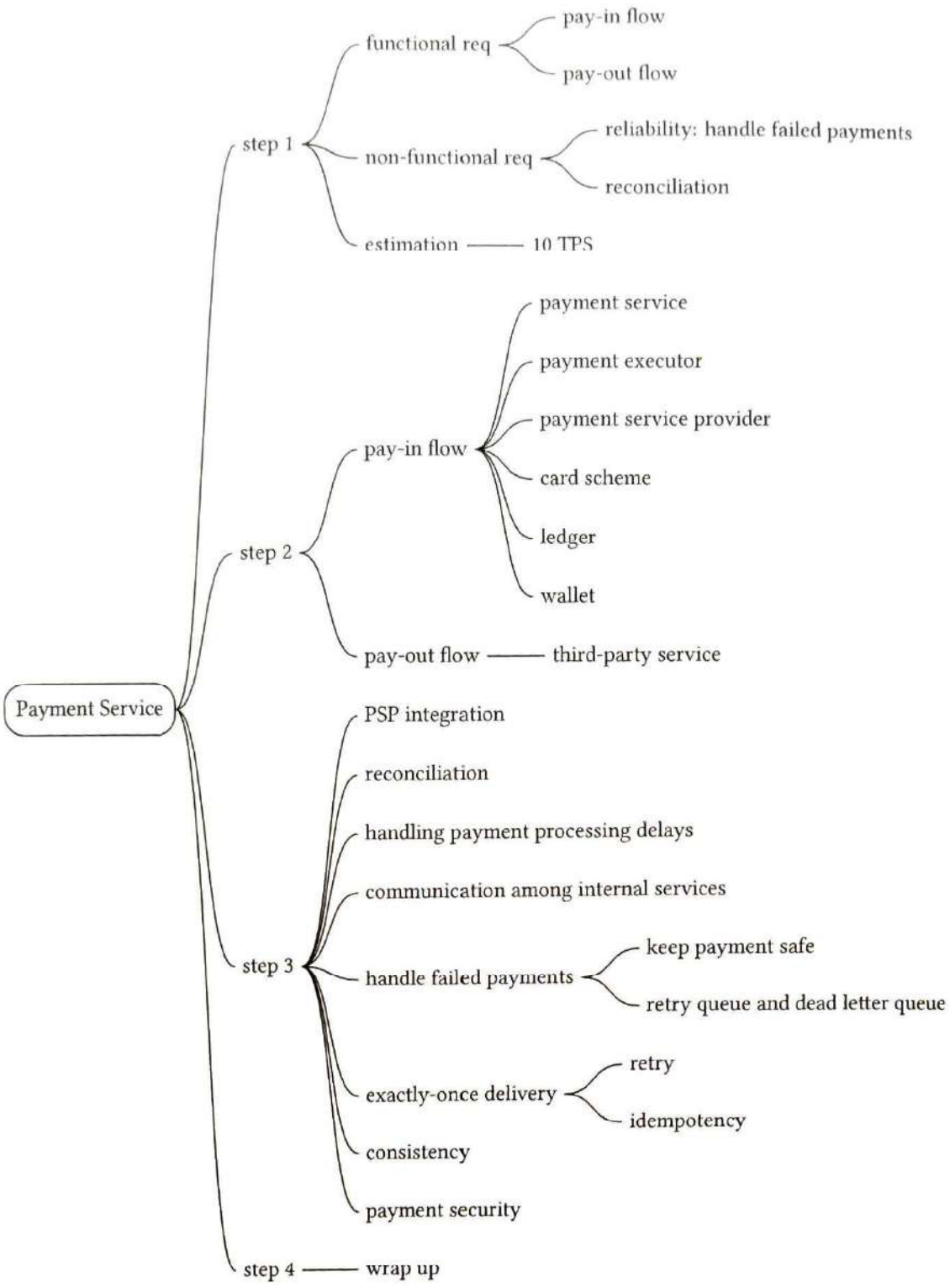
- Monitoring. Monitoring key metrics is a critical part of any modern application. With extensive monitoring, we can answer questions like "What is the average acceptance rate for a specific payment method?", "What is the CPU usage of our servers?", etc. We can create and display those metrics on a dashboard.
- Alerting. When something abnormal occurs, it is important to alert on-call developers so they respond promptly.
- Debugging tools. "Why does a payment fail?" is a common question. To make debugging easier for engineers and customer support, it is important to develop tools that allow staff to review the transaction status, processing server history, PSP records, etc. of a payment transaction.
- Currency exchange. Currency exchange is an important consideration when designing a payment system for an international user base.
- Geography. Different regions might have completely different sets of payment meth-

ods.

- Cash payment. Cash payment is very common in India, Brazil, and some other countries. Uber [28] and Airbnb [29] wrote detailed engineering blogs about how they handled cash-based payment.
- Google/Apple pay integration. Please read [30] if interested.

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

Chapter Summary



Reference Material

[1] Payment system. https://en.wikipedia.org/wiki/Payment_system.

[2] AML/CFT. https://en.wikipedia.org/wiki/Money_laundering.

[3] Card scheme. https://en.wikipedia.org/wiki/Card_scheme.

[4] ISO 4217. https://en.wikipedia.org/wiki/ISO_4217.

[5] Stripe API Reference. https://stripe.com/docs/api.

[6] Double-entry bookkeeping. https://en.wikipedia.org/wiki/Double-entry_bookkeeping.

[7] Books, an immutable double-entry accounting database service. https://developer.squareup.com/blog/books-an-immutable-double-entry-accounting-database-service/.

[8] Payment Card Industry Data Security Standard. https://en.wikipedia.org/wiki/Payment_Card_Industry_Data_Security_Standard.

[9] Tipalti. https://tipalti.com/.

[10] Nonce. https://en.wikipedia.org/wiki/Cryptographic_nonce.

[11] Webhooks. https://stripe.com/docs/webhooks.

[12] Customize your success page. https://stripe.com/docs/payments/checkout/custom-success-page.

[13] 3D Secure. https://en.wikipedia.org/wiki/3-D_Secure.

[14] Kafka Connect Deep Dive – Error Handling and Dead Letter Queues. https://www.confluent.io/blog/kafka-connect-deep-dive-error-handling-dead-letter-queues/.

[15] Reliable Processing in a Streaming Payment System. https://www.youtube.com/watch?v=5TD8m7w1xE0&list=PLLEUtp5eGr7Dz3fWGUpiSiG3d_WgJe-KJ.

[16] Chain Services with Exactly-Once Guarantees. https://www.confluent.io/blog/chain-services-exactly-guarantees/.

[17] Exponential backoff. https://en.wikipedia.org/wiki/Exponential_backoff.

[18] Idempotence. https://en.wikipedia.org/wiki/Idempotence.

[19] Stripe idempotent requests. https://stripe.com/docs/api/idempotent_requests.

[20] Idempotency. https://developer.paypal.com/docs/platforms/develop/idempotency/.

[21] Paxos. https://en.wikipedia.org/wiki/Paxos_(computer_science).

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

[23] YogabyteDB. https://www.yugabyte.com/.

[24] Cockroachdb. https://www.cockroachlabs.com/.

[25] What is DDoS attack. https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/.

[26] How Payment Gateways Can Detect and Prevent Online Fraud. https://www.chargebee.com/blog/optimize-online-billing-stop-online-fraud/.

[27] Advanced Technologies for Detecting and Preventing Fraud at Uber. https://eng.uber.com/advanced-technologies-detecting-preventing-fraud-uber/.

[28] Re-Architecting Cash and Digital Wallet Payments for India with Uber Engineering. https://eng.uber.com/india-payments/.

[29] Scaling Airbnb's Payment Platform. https://medium.com/airbnb-engineering/scaling-airbnbs-payment-platform-43ebfc99b324.

[30] Payments Integration at Uber: A Case Study – Gergely Orosz. https://www.youtube.com/watch?v=yooCE5B0SRA.

12 Digital Wallet

Payment platforms usually provide a digital wallet service to clients, so they can store money in the wallet and spend it later. For example, you can add money to your digital wallet from your bank card and when you buy products online, you are given the option to pay using the money in your wallet. Figure 12.1 shows this process.

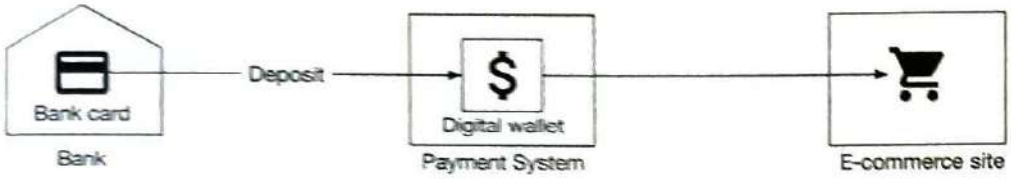


Figure 12.1: Digital wallet

Spending money is not the only feature that the digital wallet provides. For a payment platform like PayPal, we can directly transfer money to somebody else's wallet on the same payment platform. Compared with the bank-to-bank transfer, direct transfer between digital wallets is faster, and most importantly, it usually does not charge an extra fee. Figure 12.2 shows a cross-wallet balance transfer operation.

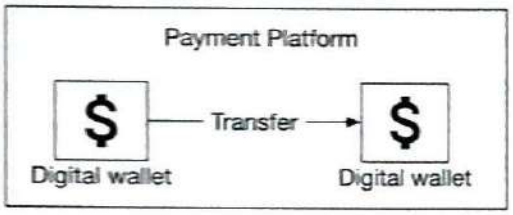


Figure 12.2: Cross-wallet balance transfer

Suppose we are asked to design the backend of a digital wallet application that supports the cross-wallet balance transfer operation. At the beginning of the interview, we will ask clarification questions to nail down the requirements.

Step 1 - Understand the Problem and Establish Design Scope

Candidate: Should we only focus on balance transfer operations between two digital wallets? Do we need to worry about other features?
Interviewer: Let's focus on balance transfer operations only.

Candidate: How many transactions per second (TPS) does the system need to support?
Interviewer: Let's assume 1,000,000 TPS.

Candidate: A digital wallet has strict requirements for correctness. Can we assume transactional guarantees [1] are sufficient?
Interviewer: That sounds good.

Candidate: Do we need to prove correctness?
Interviewer: This is a good question. Correctness is usually only verifiable after a transaction is complete. One way to verify is to compare our internal records with statements from banks. The limitation of reconciliation is that it only shows discrepancies and cannot tell how a difference was generated. Therefore, we would like to design a system with reproducibility, meaning we could always reconstruct historical balance by replaying the data from the very beginning.

Candidate: Can we assume the availability requirement is 99.99%
Interviewer: Sounds good.

Candidate: Do we need to take foreign exchange into consideration?
Interviewer: No, it's out of scope.

In summary, our digital wallet needs to support the following:

- Support balance transfer operation between two digital wallets.
- Support 1,000,000 TPS.
- Reliability is at least 99.99%.
- Support transactions.
- Support reproducibility.

Back-of-the-envelope estimation

When we talk about TPS, we imply a transactional database will be used. Today, a relational database running on a typical data center node can support a few thousand transactions per second. For example, reference [2] contains the performance benchmark of some of the popular transactional database servers. Let's assume a database node can support 1,000 TPS. In order to reach 1 million TPS, we need 1,000 database nodes.

However, this calculation is slightly inaccurate. Each transfer command requires two operations: deducting money from one account and depositing money to the other account. To support 1 million transfers per second, the system actually needs to handle up to 2 million TPS, which means we need 2,000 nodes.

Table 12.1 shows the total number of nodes required when the "per-node TPS" (the TPS a single node can handle) changes. Assuming hardware remains the same, the more

transactions a single node can handle per second, the lower the total number of nodes required, indicating lower hardware cost. So one of our design goals is to increase the number of transactions a single node can handle.

Per-node TPS	Node Number
100	20,000
1,000	2,000
10,000	200

Table 12.1: Mapping between pre-node TPS and node number

Step 2 - Propose High-level Design and Get Buy-in

In this section, we will discuss the following:

- API design
- Three high-level designs
  1. Simple in-memory solution
  2. Database-based distributed transaction solution
  3. Event sourcing solution with reproducibility

API design

We will use the RESTful API convention. For this interview, we only need to support one API:

API	Detail
`POST /v1/wallet/balance_transfer`	Transfer balance from one wallet to another

Request parameters are:

Field	Description	Type
`from_account`	The debit account	string
`to_account`	The credit account	string
`amount`	The amount of money	string
`currency`	The currency type	string (ISO 4217 [3])
`transaction_id`	ID used for deduplication	uuid

Sample response body:

```
{
  "Status": "success"
  "Transaction_id": "01589980-2664-11ec-9621-0242ac130002"
}
```

One thing worth mentioning is that the data type of the "amount" field is "string,"

rather than "double". We explained the reasoning in Chapter 11 Payment System on page 320.

In practice, many people still choose float or double representation of numbers because it is supported by almost every programming language and database. It is a proper choice as long as we understand the potential risk of losing precision.

In-memory sharding solution

The wallet application maintains an account balance for every user account. A good data structure to represent this <user,balance> relationship is a map, which is also called a hash table (map) or key-value store.

For in-memory stores, one popular choice is Redis. One Redis node is not enough to handle 1 million TPS. We need to set up a cluster of Redis nodes and evenly distribute user accounts among them. This process is called partitioning or sharding.

To distribute the key-value data among n partitions, we could calculate the hash value of the key and divide it by n. The remainder is the destination of the partition. The pseudocode below shows the sharding process:

```
String accountID = "A";
Int partitionNumber = 7;
Int myPartition = accountID.hashCode() % partitionNumber;
```

The number of partitions and addresses of all Redis nodes can be stored in a centralized place. We could use ZooKeeper [4] as a highly-available configuration storage solution.

The final component of this solution is a service that handles the transfer commands. We call it the wallet service and it has several key responsibilities.

1. Receives the transfer command
2. Validates the transfer command
3. If the command is valid, it updates the account balances for the two users involved in the transfer. In a cluster, the account balances are likely to be in different Redis nodes

The wallet service is stateless. It is easy to scale horizontally. Figure 12.3 shows the in-memory solution.

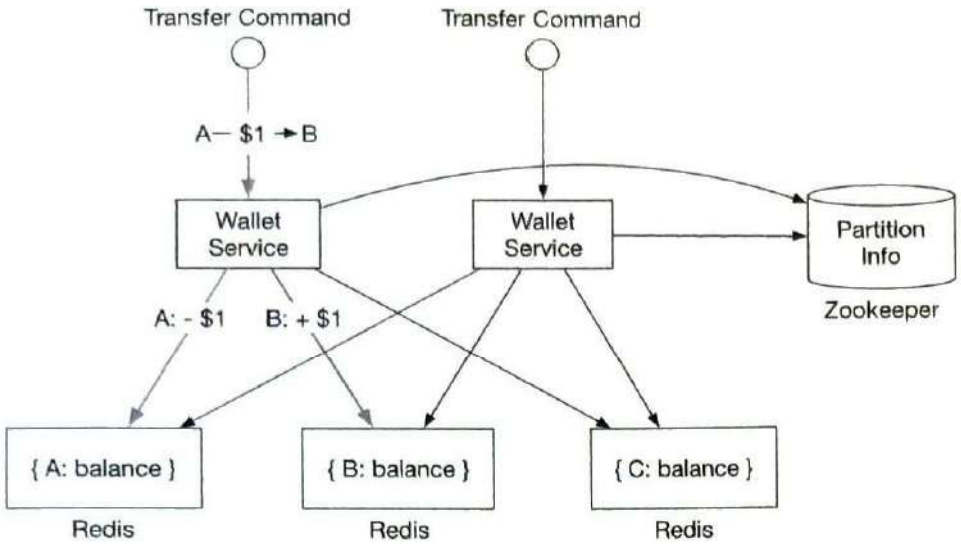


Figure 12.3: In-memory solution

In this example, we have 3 Redis nodes. There are three clients, A, B, and C. Their account balances are evenly spread across these three Redis nodes. There are two wallet service nodes in this example that handle the balance transfer requests. When one of the wallet service nodes receives the transfer command which is to move $1 from client A to client B, it issues two commands to two Redis nodes. For the Redis node that contains client A's account, the wallet service deducts $1 from the account. For client B, the wallet service adds $1 to the account.

Candidate: In this design, account balances are spread across multiple Redis nodes. ZooKeeper is used to maintain the sharding information. The stateless wallet service uses the sharding information to locate the Redis nodes for the clients and updates the account balances accordingly.

Interviewer: This design works, but it does not meet our correctness requirement. The wallet service updates two Redis nodes for each transfer. There is no guarantee that both updates would succeed. If, for example, the wallet service node crashes after the first update has gone through but before the second update is done, it would result in an incomplete transfer. The two updates need to be in a single atomic transaction.

Distributed transactions

Database sharding

How do we make the updates to two different storage nodes atomic? The first step is to replace each Redis node with a transactional relational database node. Figure 12.4 shows the architecture. This time, clients A, B, and C are partitioned into 3 relational databases, rather than in 3 Redis nodes.

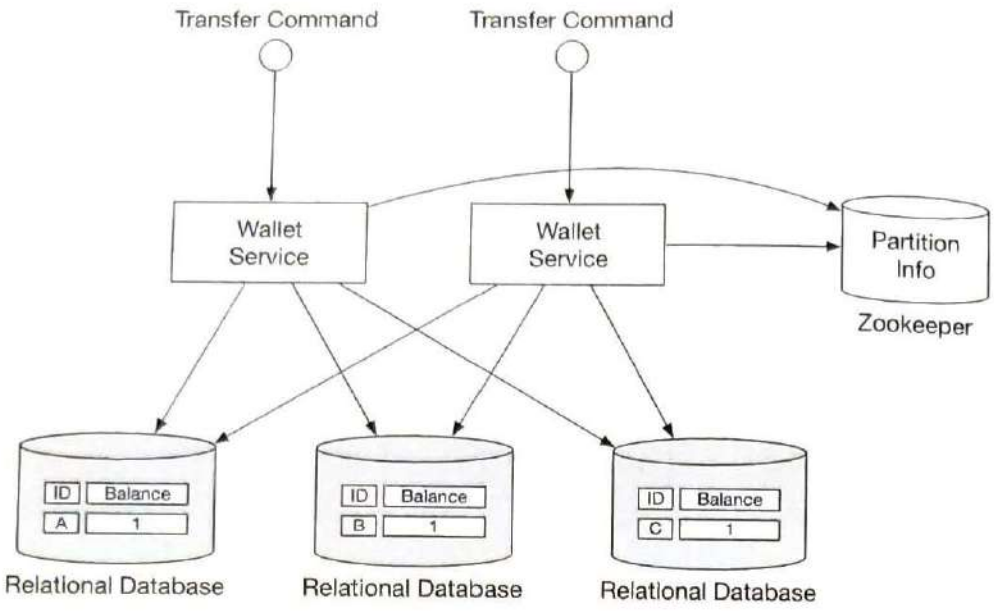


Figure 12.4: Relational database

Using transactional databases only solves part of the problem. As mentioned in the last section, it is very likely that one transfer command will need to update two accounts in two different databases. There is no guarantee that two update operations will be handled at exactly the same time. If the wallet service restarted right after it updated the first account balance, how can we make sure the second account will be updated as well?

Distributed transaction: Two-phase commit

In a distributed system, a transaction may involve multiple processes on multiple nodes. To make a transaction atomic, the distributed transaction might be the answer. There are two ways to implement a distributed transaction: a low-level solution and a high-level solution. We will examine each of them.

The low-level solution relies on the database itself. The most commonly used algorithm is called two-phase commit (2PC). As the name implies, it has two phases, as in Figure 12.5.

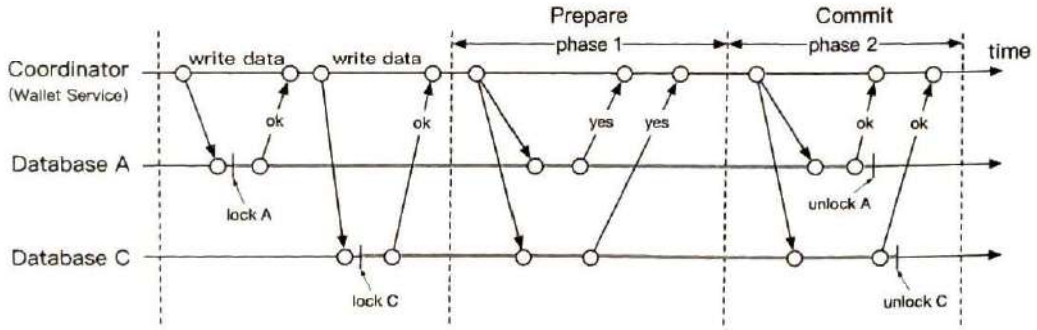


Figure 12.5: Two-phase commit (source [5])

1. The coordinator, which in our case is the wallet service, performs read and write operations on multiple databases as normal. As shown in Figure 12.5, both databases A and C are locked.
2. When the application is about to commit the transaction, the coordinator asks all databases to prepare the transaction.
3. In the second phase, the coordinator collects replies from all databases and performs the following:
   (a) If all databases reply with a yes, the coordinator asks all databases to commit the transaction they have received.
   (b) If any database replies with a no, the coordinator asks all databases to abort the transaction.

It is a low-level solution because the prepare step requires a special modification to the database transaction. For example, there is an X/Open XA [6] standard that coordinates heterogeneous databases to achieve 2PC. The biggest problem with 2PC is that it's not performant, as locks can be held for a very long time while waiting for a message from the other nodes. Another issue with 2PC is that the coordinator can be a single point of failure, as shown in Figure 12.6.

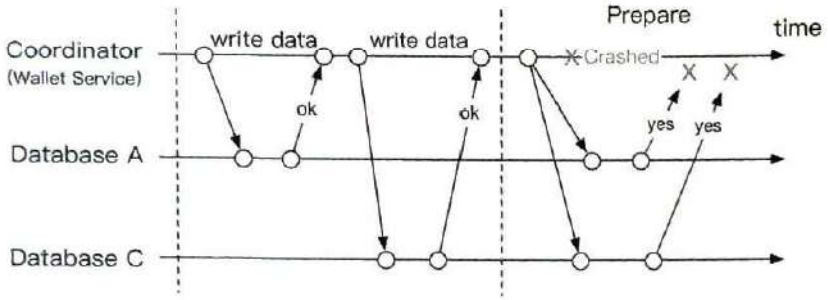


Figure 12.6: Coordinator crashes

Distributed transaction: Try-Confirm/Cancel (TC/C)

TC/C is a type of compensating transaction [7] that has two steps:

1. In the first phase, the coordinator asks all databases to reserve resources for the transaction.
2. In the second phase, the coordinator collects replies from all databases:
   (a) If all databases reply with yes, the coordinator asks all databases to confirm the operation, which is the Try-Confirm process.
   (b) If any database replies with no, the coordinator asks all databases to cancel the operation, which is the Try-Cancel process.

It's important to note that the two phases in 2PC are wrapped in the same transaction, but in TC/C each phase is a separate transaction.

TC/C example

It would be much easier to explain how TC/C works with a real-world example. Suppose we want to transfer $1 from account A to account C. Table 12.2 gives a summary of how TC/C is executed in each phase.

Phase	Operation	A	C
1	Try	Balance change: −$1	Do nothing
2	Confirm	Do nothing	Balance change: +$1
	Cancel	Balance change: +$1	Do Nothing

Table 12.2: TC/C example

Let's assume the wallet service is the coordinator of the TC/C. At the beginning of the distributed transaction, account A has $1 in its balance, and account C has $0.

First phase: Try In the Try phase, the wallet service, which acts as the coordinator, sends two transaction commands to two databases:

1. For the database that contains account A, the coordinator starts a local transaction that reduces the balance of A by $1.
2. For the database that contains account C, the coordinator gives it a `NOP` (no operation). To make the example adaptable for other scenarios, let's assume the coordinator sends to this database a `NOP` command. The database does nothing for `NOP` commands and always replies to the coordinator with a success message.

The Try phase is shown in Figure 12.7. The thick line indicates that a lock is held by the transaction.

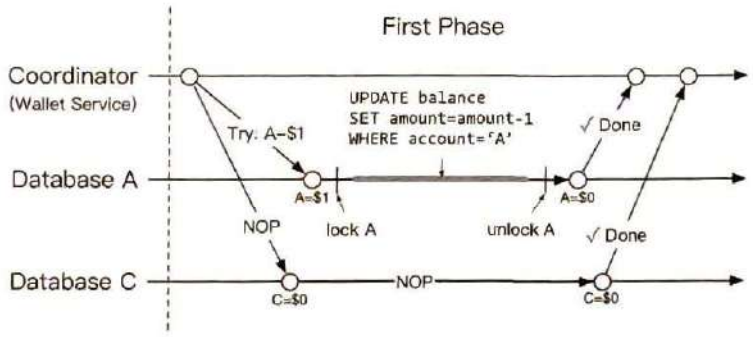


Figure 12.7: Try phase

Second phase: Confirm If both databases reply yes, the wallet service starts the next Confirm phase.

Account A's balance has already been updated in the first phase. The wallet service does not need to change its balance here. However, account C has not yet received its $1 from account A in the first phase. In the Confirm phase, the wallet service has to add $1 to account C's balance.

The Confirm process is shown in Figure 12.8.

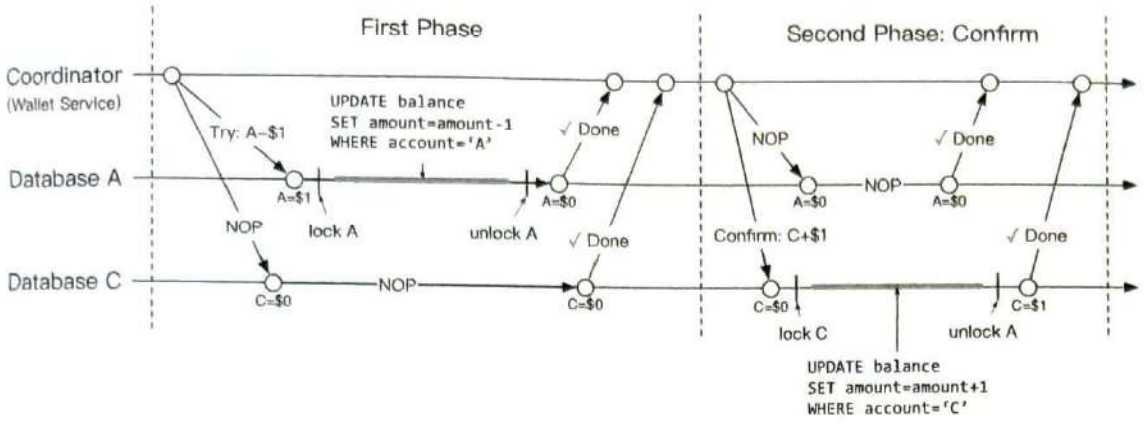


Figure 12.8: Confirm phase

Second phase: Cancel What if the first Try phase fails? In the example above we have assumed the NOP operation on account C always succeeds, although in practice it may fail. For example, account C might be an illegal account, and the regulator has mandated that no money can flow into or out of this account. In this case, the distributed transaction must be canceled and we have to clean up.

Because the balance of account A has already been updated in the transaction in the Try phase, it is impossible for the wallet service to cancel a completed transaction. What it can do is to start another transaction that reverts the effect of the transaction in the Try phase, which is to add $1 back to account A.

Because account C was not updated in the Try phase, the wallet service just needs to send a NOP operation to account C's database.

The Cancel process is shown in Figure 12.9.

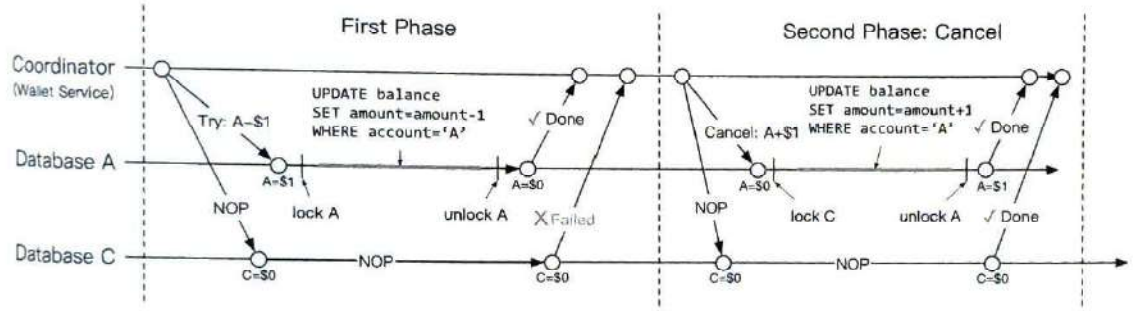


Figure 12.9: Cancel phase

Comparison between 2PC and TC/C

Table 12.3 shows that there are many similarities between 2PC and TC/C, but there are also differences. In 2PC, all local transactions are not done (still locked) when the second phase starts, while in TC/C, all local transactions are done (unlocked) when the second phase starts. In other words, the second phase of 2PC is about completing an unfinished

transaction, such as an abort or commit, while in TC/C, the second phase is about using a reverse operation to offset the previous transaction result when an error occurs. The following table summarizes their differences.

	First Phase	Second Phase: success	Second Phase: fail
2PC	Local transactions are not done yet	Commit all local transactions	Cancel all local transactions
TC/C	All local transactions are completed, either committed or canceled	Execute new local transactions if needed	Reverse the side effect of the already committed transaction, or called "undo"

Table 12.3: 2PC v.s. TC/C

TC/C is also called a distributed transaction by compensation. It is a high-level solution because the compensation, also called the "undo," is implemented in the business logic. The advantage of this approach is that it is database-agnostic. As long as a database supports transactions, TC/C will work. The disadvantage is that we have to manage the details and handle the complexity of the distributed transactions in the business logic at the application layer.

Phase status table

We still have not yet answered the question asked earlier; what if the wallet service restarts in the middle of TC/C? When it restarts, all previous operation history might be lost, and the system may not know how to recover.

The solution is simple. We can store the progress of a TC/C as phase status in a transactional database. The phase status includes at least the following information.

- The ID and content of a distributed transaction.
- The status of the Try phase for each database. The status could be `not sent yet`, `has been sent`, and `response received`.
- The name of the second phase. It could be `Confirm` or `Cancel`. It could be calculated using the result of the Try phase.
- The status of the second phase.
- An out-of-order flag (explained soon in the section "out-of-order Execution").

Where should we put the phase status tables? Usually, we store the phase status in the database that contains the wallet account from which money is deducted. The updated architecture diagram is shown in Figure 12.10.

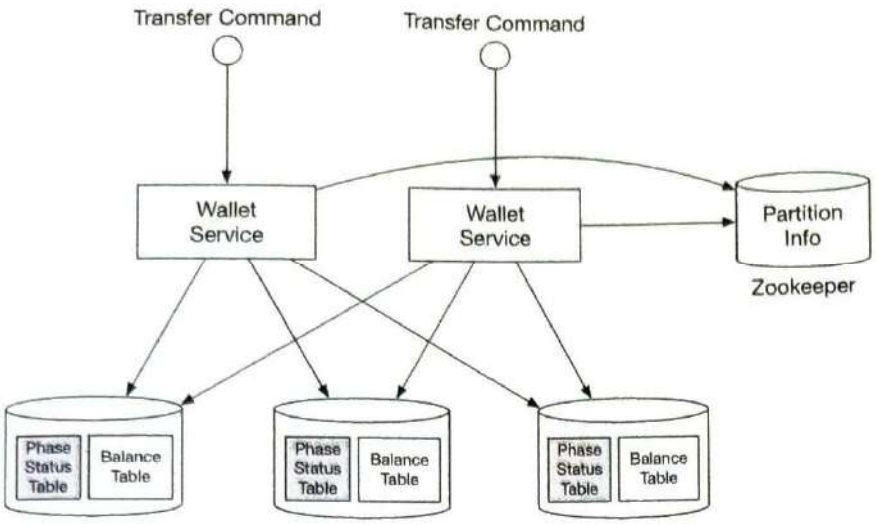


Figure 12.10: Phase status table

Unbalanced state

Have you noticed that by the end of the Try phase, $1 is missing (Figure 12.11)?

Assuming everything goes well, by the end of the Try phase, $1 is deducted from account A and account C remains unchanged. The sum of account balances in A and C will be $0, which is less than at the beginning of the TC/C. It violates a fundamental rule of accounting that the sum should remain the same after a transaction.

The good news is that the transactional guarantee is still maintained by TC/C. TC/C comprises several independent local transactions. Because TC/C is driven by application, the application itself is able to see the intermediate result between these local transactions. On the other hand, the database transaction or 2PC version of the distributed transaction was maintained by databases that are invisible to high-level applications.

There are always data discrepancies during the execution of distributed transactions. The discrepancies might be transparent to us because lower-level systems such as databases already fixed the discrepancies. If not, we have to handle it ourselves (for example, TC/C).

The unbalanced state is shown in Figure 12.11.

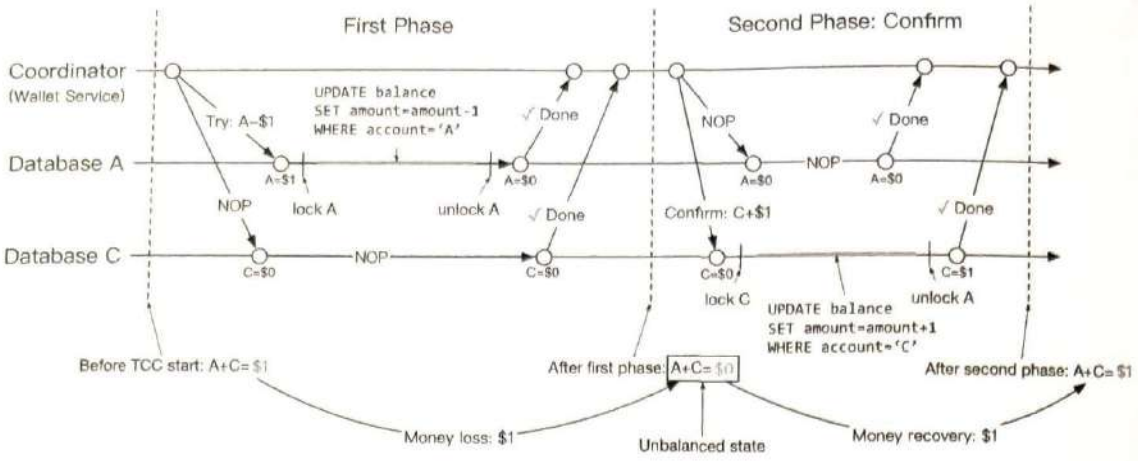


Figure 12.11: Unbalanced state

Valid operation orders

There are three choices for the Try phase:

Try phase choices	Account A	Account C
Choice 1	−$1	NOP
Choice 2	NOP	+$1
Choice 3	−$1	+$1

Table 12.4: Try phase choices

All three choices look plausible, but some are not valid.

For choice 2, if the Try phase on account C is successful, but has failed on account A (NOP), the wallet service needs to enter the Cancel phase. There is a chance that somebody else may jump in and move the $1 away from account C. Later when the wallet service tries to deduct $1 from account C, it finds nothing is left, which violates the transactional guarantee of a distributed transaction.

For choice 3, if $1 is deducted from account A and added to account C concurrently, it introduces lots of complications. For example, $1 is added to account C, but it fails to deduct the money from account A. What should we do in this case?

Therefore, choice 2 and choice 3 are flawed choices and only choice 1 is valid.

Out-of-order execution

One side effect of TC/C is the out-of-order execution. It will be much easier to explain using an example.

We reuse the above example which transfers $1 from account A to account C. As Figure 12.12 shows, in the Try phase, the operation against account A fails and it returns a failure to the wallet service, which then enters the Cancel phase and sends the cancel operation to both account A and account C.

Let's assume that the database that handles account C has some network issues and it

receives the Cancel instruction before the Try instruction. In this case, there is nothing to cancel.

The out-of-order execution is shown in Figure 12.12.

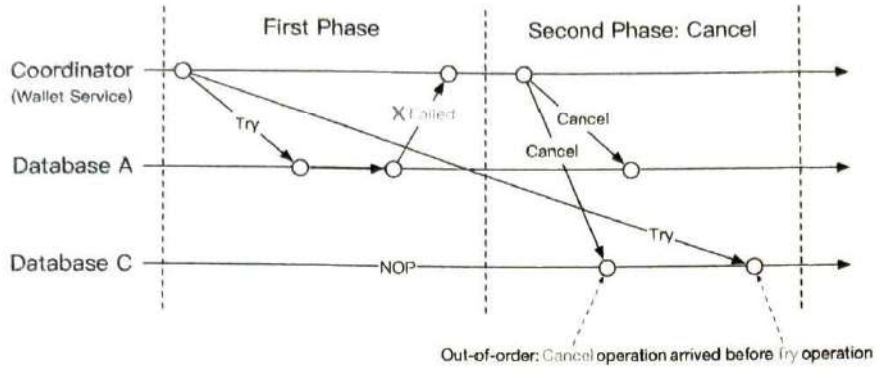


Figure 12.12: Out-of-order execution

To handle out-of-order operations, each node is allowed to Cancel a TC/C without receiving a Try instruction, by enhancing the existing logic with the following updates:

- The out-of-order Cancel operation leaves a flag in the database indicating that it has seen a Cancel operation, but it has not seen a Try operation yet.
- The Try operation is enhanced so it always checks whether there is an out-of-order flag, and it returns a failure if there is.

This is why we added an out-of-order flag to the phase status table in the "Phase Status Table" section.

Distributed transaction: Saga

Linear order execution

There is another popular distributed transaction solution called Saga [8]. Saga is the de-facto standard in a microservice architecture. The idea of Saga is simple:

1. All operations are ordered in a sequence. Each operation is an independent transaction on its own database.
2. Operations are executed from the first to the last. When one operation has finished, the next operation is triggered.
3. When an operation has failed, the entire process starts to roll back from the current operation to the first operation in reverse order, using compensating transactions. So if a distributed transaction has n operations, we need to prepare $2n$ operations: n operations for the normal case and another n for the compensating transaction during rollback.

It is easier to understand this by using an example. Figure 12.13 shows the Saga workflow to transfer $1 from account A to account C. The top horizontal line shows the normal

order of execution. The two vertical lines show what the system should do when there is an error. When it encounters an error, the transfer operations are rolled back and the client receives an error message. As we mentioned in the "Valid operation orders" section on page 352, we have to put the deduction operation before the addition operation.

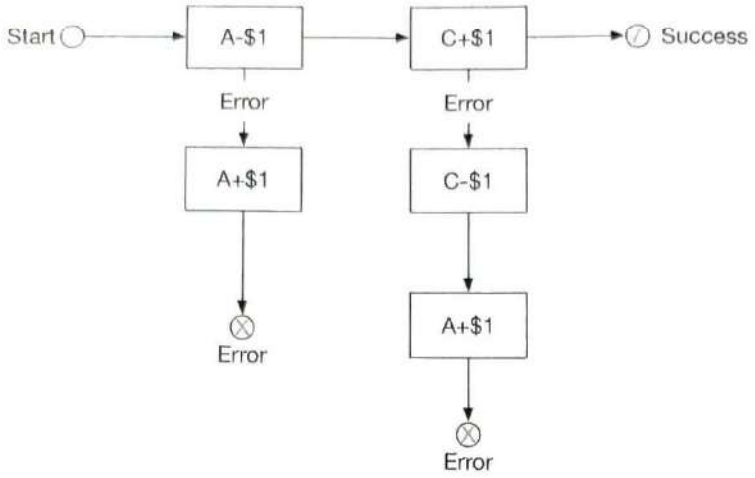


Figure 12.13: Saga workflow

How do we coordinate the operations? There are two ways to do it:

1. Choreography. In a microservice architecture, all the services involved in the Saga distributed transaction do their jobs by subscribing to other services' events. So it is fully decentralized coordination.
2. Orchestration. A single coordinator instructs all services to do their jobs in the correct order.

The choice of which coordination model to use is determined by the business needs and goals. The challenge of the choreography solution is that services communicate in a fully asynchronous way, so each service has to maintain an internal state machine in order to understand what to do when other services emit an event. It can become hard to manage when there are many services. The orchestration solution handles complexity well, so it is usually the preferred solution in a digital wallet system.

Comparison between TC/C and Saga

TC/C and Saga are both application-level distributed transactions. Table 12.5 summarizes their similarities and differences.

	TC/C	Saga
Compensating action	In Cancel phase	In rollback phase
Central coordination	Yes	Yes (orchestration mode)
Operation execution order	any	linear
Parallel execution possibility	Yes	No (linear execution)
Could see the partial inconsistent status	Yes	Yes
Application or database logic	Application	Application

Table 12.5: TC/C vs Saga

Which one should we use in practice? The answer depends on the latency requirement. As Table 12.5 shows, operations in Saga have to be executed in linear order, but it is possible to execute them in parallel in TC/C. So the decision depends on a few factors:

1. If there is no latency requirement, or there are very few services, such as our money transfer example, we can choose either of them. If we want to go with the trend in microservice architecture, choose Saga.
2. If the system is latency-sensitive and contains many services/operations, TC/C might be a better option.

Candidate: To make the balance transfer transactional, we replace Redis with a relational database, and use TC/C or Saga to implement distributed transactions.
Interviewer: Great work! The distributed transaction solution works, but there might be cases where it doesn't work well. For example, users might enter the wrong operations at the application level. In this case, the money we specified might be incorrect. We need a way to trace back the root cause of the issue and audit all account operations. How can we do this?

Event sourcing

Background

In real life, a digital wallet provider may be audited. These external auditors might ask some challenging questions, for example:

1. Do we know the account balance at any given time?
2. How do we know the historical and current account balances are correct?
3. How do we prove that the system logic is correct after a code change?

One design philosophy that systematically answers those questions is event sourcing, which is a technique developed in Domain-Driven Design (DDD) [9].

Definition

There are four important terms in event sourcing.

1. Command
2. Event
3. State
4. State machine

Command

A command is the intended action from the outside world. For example, if we want to transfer $1 from client A to client C, this money transfer request is a command.

In event sourcing, it is very important that everything has an order. So commands are usually put into a FIFO (first in, first out) queue.

Event

Command is an intention and not a fact because some commands may be invalid and cannot be fulfilled. For example, the transfer operation will fail if the account balance becomes negative after the transfer.

A command must be validated before we do anything about it. Once the command passes the validation, it is valid and must be fulfilled. The result of the fulfillment is called an event.

There are two major differences between command and event.

1. Events must be executed because they represent a validated fact. In practice, we usually use the past tense for an event. If the command is "transfer $1 from A to C", the corresponding event would be "transfer**red** $1 from A to C".
2. Commands may contain randomness or I/O, but events must be deterministic. Events represent historical facts.

There are two important properties of the event generation process.

1. One command may generate any number of events. It could generate zero or more events.
2. Event generation may contain randomness, meaning it is not guaranteed that a command always generates the same event(s). The event generation may contain external I/O or random numbers. We will revisit this property in more detail near the end of the chapter.

The order of events must follow the order of commands. So events are stored in a FIFO queue, as well.

State

State is what will be changed when an event is applied. In the wallet system, state is the balances of all client accounts, which can be represented with a map data structure. The key is the account name or ID, and the value is the account balance. Key-value stores are usually used to store the map data structure. The relational database can also be viewed as a key-value store, where keys are primary keys and values are table rows.

State machine

A state machine drives the event sourcing process. It has two major functions.

1. Validate commands and generate events.
2. Apply event to update state.

Event sourcing requires the behavior of the state machine to be deterministic. Therefore, the state machine itself should never contain any randomness. For example, it should never read anything random from the outside using I/O, or use any random numbers. When it applies an event to a state, it should always generate the same result.

Figure 12.14 shows the static view of event sourcing architecture. The state machine is responsible for converting the command to an event and for applying the event. Because state machine has two primary functions, we usually draw two state machines, one for validating commands and the other for applying events.

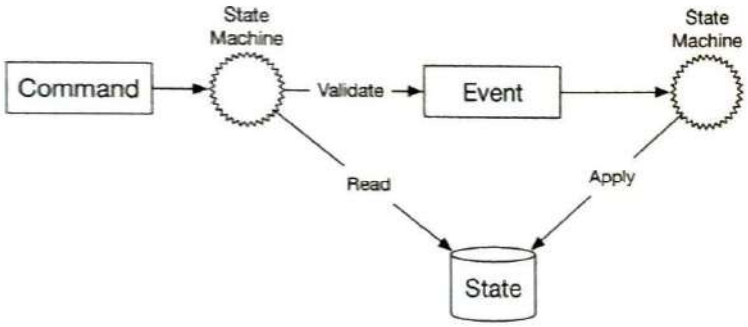


Figure 12.14: Static view of event sourcing

If we add the time dimension, Figure 12.15 shows the dynamic view of event sourcing. The system keeps receiving commands and processing them, one by one.

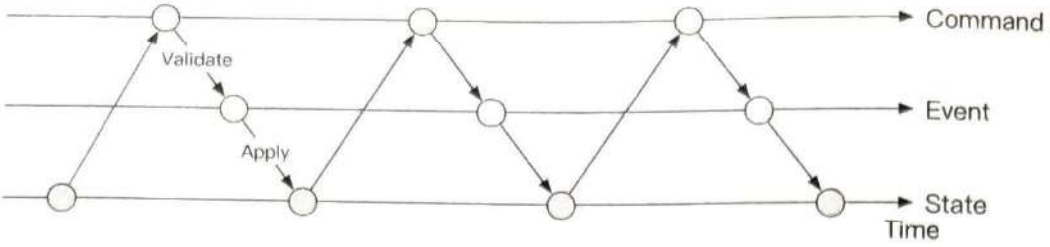


Figure 12.15: Dynamic view of event sourcing

Wallet service example

For the wallet service, the commands are balance transfer requests. These commands are put into a FIFO queue. One popular choice for the command queue is Kafka [10]. The command queue is shown in Figure 12.16.

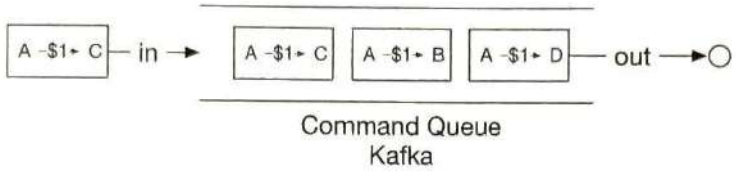


Figure 12.16: Command queue

Let us assume the state (the account balance) is stored in a relational database. The state machine examines each command one by one in FIFO order. For each command, it checks whether the account has a sufficient balance. If yes, the state machine generates an event for each account. For example, if the command is "A $\rightarrow$ \$1 $\rightarrow$ C", the state machine generates two events: "A:−\$1" and "C:+\$1".

Figure 12.17 shows how the state machine works in 5 steps.

1. Read commands from the command queue.
2. Read balance state from the database.
3. Validate the command. If it is valid, generate two events for each of the accounts.
4. Read the next event.
5. Apply the event by updating the balance in the database.

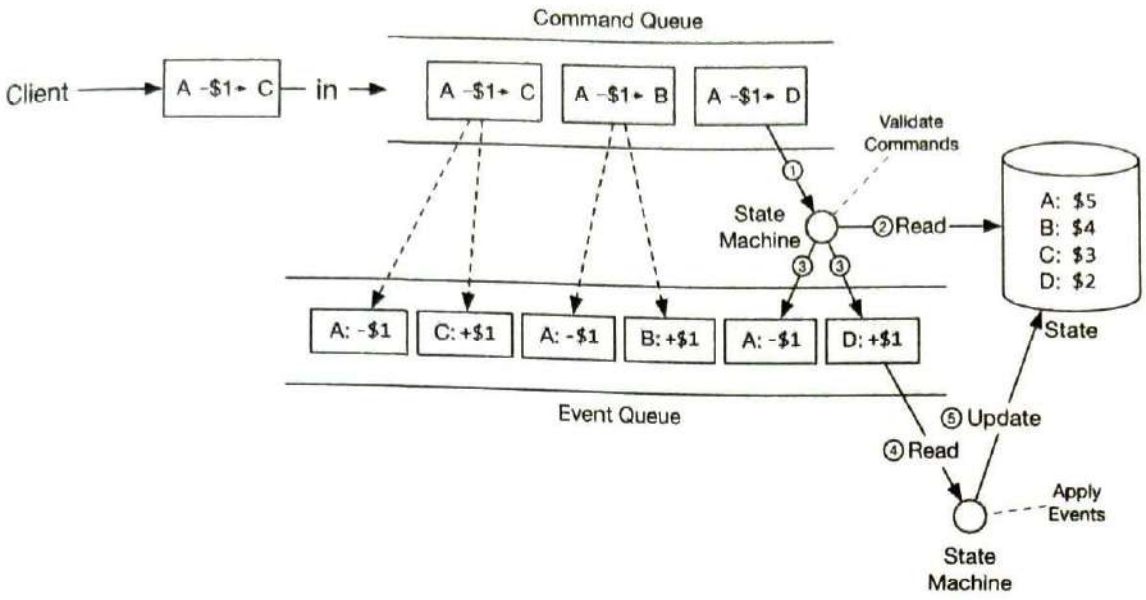


Figure 12.17: How state machine works

Reproducibility

The most important advantage that event sourcing has over other architectures is reproducibility.

In the distributed transaction solutions mentioned earlier, a wallet service saves the updated account balance (the state) into the database. It is difficult to know why the account balance was changed. Meanwhile, historical balance information is lost during the update operation. In the event sourcing design, all changes are saved first as immutable history. The database is only used as an updated view of what balance looks like at any given point in time.

We could always reconstruct historical balance states by replaying the events from the very beginning. Because the event list is immutable and the state machine logic is deterministic, it is guaranteed that the historical states generated from each replay are the same.

Figure 12.18 shows how to reproduce the states of the wallet service by replaying the events.

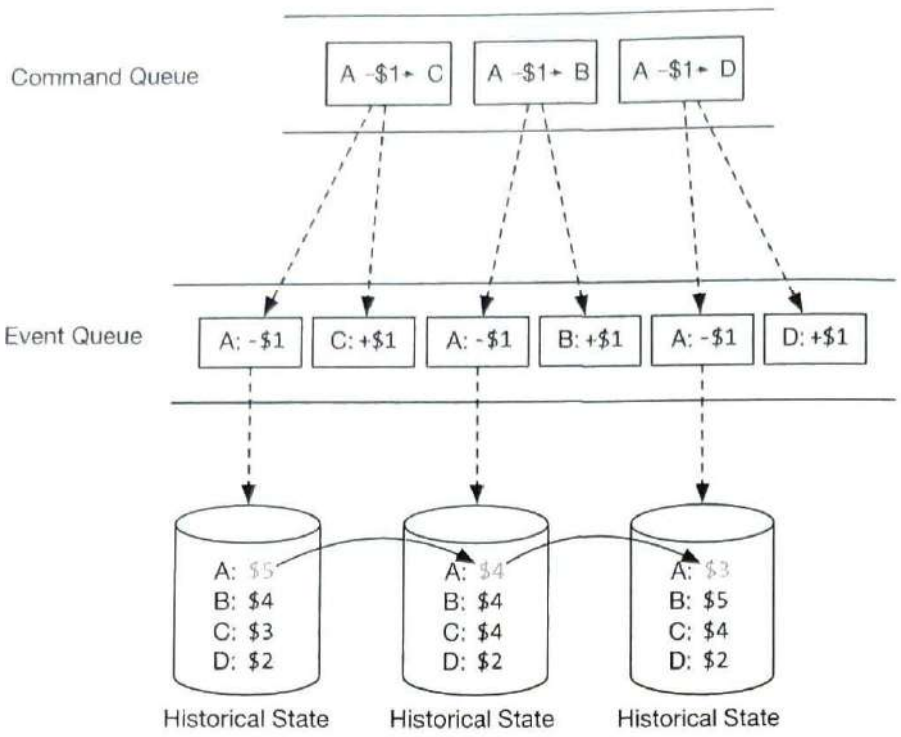


Figure 12.18: Reproduce states

Reproducibility helps us answer the difficult questions that the auditors ask at the beginning of the section. We repeat the questions here.

1. Do we know the account balance at any given time?
2. How do we know the historical and current account balances are correct?
3. How do we prove the system logic is correct after a code change?

For the first question, we could answer it by replaying events from the start, up to the point in time where we would like to know the account balance.

For the second question, we could verify the correctness of the account balance by recalculating it from the event list.

For the third question, we can run different versions of the code against the events and verify that their results are identical.

Because of the audit capability, event sourcing is often chosen as the de facto solution for the wallet service.

Command-query responsibility segregation (CQRS)

So far, we have designed the wallet service to move money from one account to another efficiently. However, the client still does not know what the account balance is. There needs to be a way to publish state (balance information) so the client, which is outside of the event sourcing framework, can know what the state is.

Intuitively, we can create a read-only copy of the database (historical state) and share

it with the outside world. Event sourcing answers this question in a slightly different way.

Rather than publishing the state (balance information), event sourcing publishes all the events. The external world could rebuild any customized state itself. This design philosophy is called CQRS [11].

In CQRS, there is one state machine responsible for the write part of the state, but there can be many read-only state machines, which are responsible for building views of the states. Those views could be used for queries.

These read-only state machines can derive different state representations from the event queue. For example, clients may want to know their balances and a read-only state machine could save state in a database to serve the balance query. Another state machine could build state for a specific time period to help investigate issues like possible double charges. The state information is an audit trail that could help to reconcile the financial records.

The read-only state machines lag behind to some extent, but will always catch up. The architecture design is eventually consistent.

Figure 12.19 shows a classic CQRS architecture.

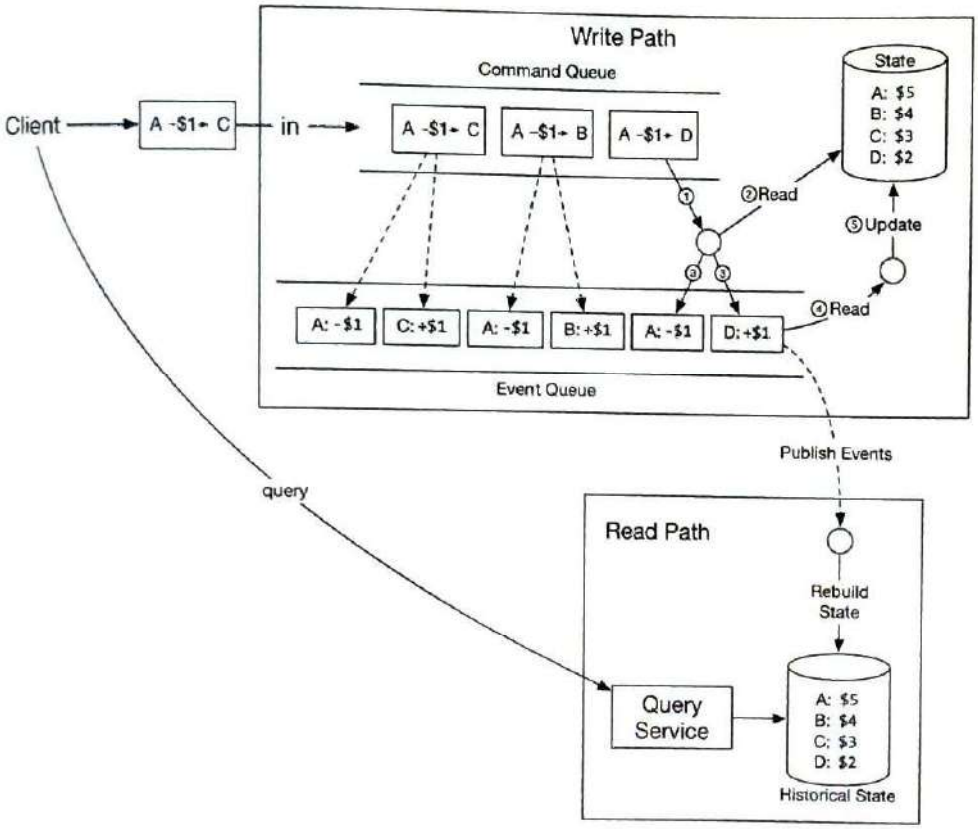


Figure 12.19: CQRS architecture

Candidate: In this design, we use event sourcing architecture to make the whole system reproducible. All valid business records are saved in an immutable event queue which could be used for correctness verification.

Interviewer: That's great. But the event sourcing architecture you proposed only handles one event at a time and it needs to communicate with several external systems. Can we make it faster?

Step 3 - Design Deep Dive

In this section, we dive deep into techniques for achieving high performance, reliability, and scalability.

High-performance event sourcing

In the earlier example, we used Kafka as the command and event store, and the database as a state store. Let's explore some optimizations.

File-based command and event list

The first optimization is to save commands and events to a local disk, rather than to a remote store like Kafka. This avoids transit time across the network. The event list uses an append-only data structure. Appending is a sequential write operation, which is generally very fast. It works well even for magnetic hard drives because the operating system is heavily optimized for sequential reads and writes. According to this article [12], sequential disk access can be faster than random memory access in some cases.

The second optimization is to cache recent commands and events in memory. As we explained before, we process commands and events right after they are persisted. We may cache them in memory to save the time of loading them back from the local disk.

We are going to explore some implementation details. A technique called mmap [13] is great for implementing the optimizations mentioned previously. Mmap can write to a local disk and cache recent content in memory at the same time. It maps a disk file to memory as an array. The operating system caches certain sections of the file in memory to accelerate the read and write operations. For append-only file operations, it is almost guaranteed that all data are saved in memory, which is very fast.

Figure 12.20 shows the file-based command and event storage.

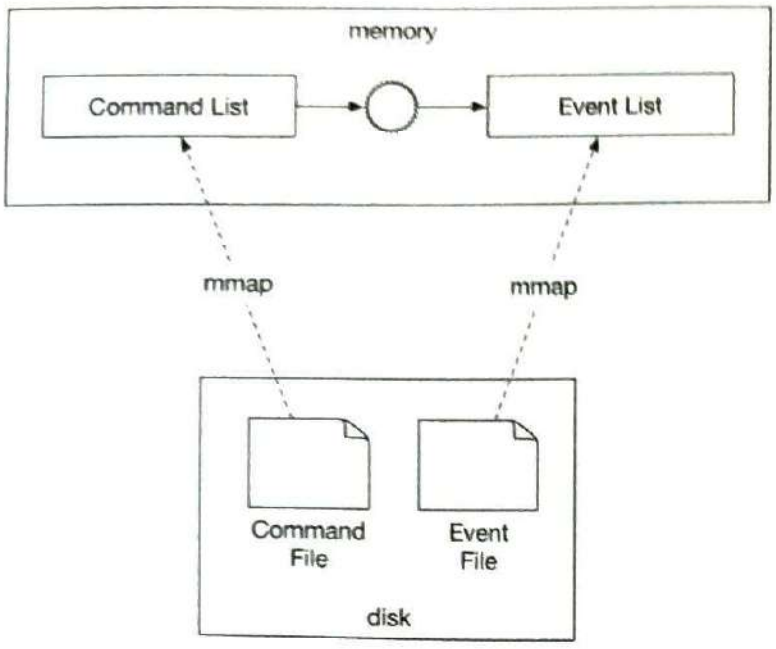


Figure 12.20: File-based command and event storage

File-based state

In the previous design, state (balance information) is stored in a relational database. In a production environment, a database usually runs in a stand-alone server that can only be accessed through networks. Similar to the optimizations we did for command and event, state information can be saved to the local disk, as well.

More specifically, we can use SQLite [14], which is a file-based local relational database or use RocksDB [15], which is a local file-based key-value store.

RocksDB is chosen because it uses a log-structured merge-tree (LSM), which is optimized for write operations. To improve read performance, the most recent data is cached.

Figure 12.21 shows the file-based solution for command, event, and state.

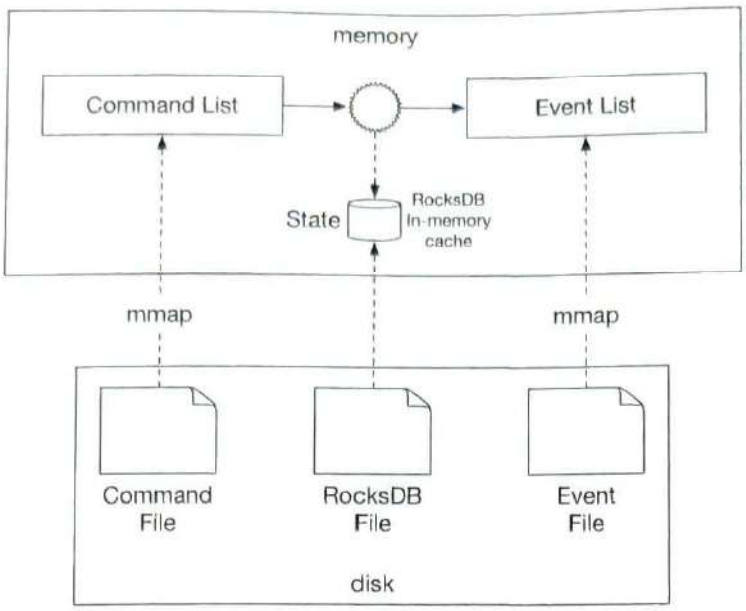


Figure 12.21: File-based solution for command, event, and state

Snapshot

Once everything is file-based, let us consider how to accelerate the reproducibility process. When we first introduced reproducibility, the state machine had to process events from the very beginning, every time. What we could optimize is to periodically stop the state machine and save the current state into a file. This is called a snapshot.

A snapshot is an immutable view of a historical state. Once a snapshot is saved, the state machine does not have to restart from the very beginning anymore. It can read data from a snapshot, verify where it left off, and resume processing from there.

For financial applications such as wallet service, the finance team often requires a snapshot to be taken at 00:00 so they can verify all transactions that happened during that day. When we first introduced CQRS of event sourcing, the solution was to set up a read-only state machine that reads from the beginning until the specified time is met. With snapshots, a read-only state machine only needs to load one snapshot that contains the data.

A snapshot is a giant binary file and a common solution is to save it in an object storage solution, such as HDFS [16].

Figure 12.22 shows the file-based event sourcing architecture. When everything is file-based, the system can fully utilize the maximum I/O throughput of the computer hardware.

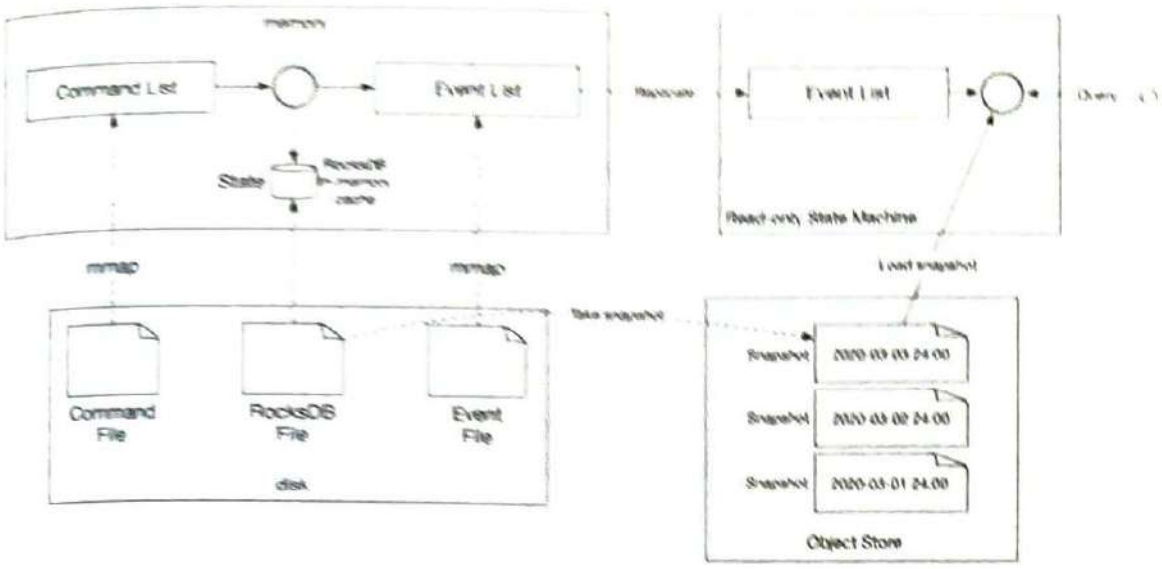


Figure 12.22: Snapshot

Candidate: We could refactor the design of event sourcing so the command list, event list, state, and snapshot are all saved in files. Event sourcing architecture processes the event list in a linear manner, which fits well into the design of hard disks and operating system cache.

Interviewer: The performance of the local file-based solution is better than the system that requires accessing data from remote Kafka and databases. However, there is another problem: because data is saved on a local disk, a server is now stateful and becomes a single point of failure. How do we improve the reliability of the system?

Reliable high-performance event sourcing

Before we explain the solution, let's examine the parts of the system that need the reliability guarantee.

Reliability analysis

Conceptually, everything a node does is around two concepts; data and computation. As long as data is durable, it's easy to recover the computational result by running the same code on another node. This means we only need to worry about the reliability of data because if data is lost, it is lost forever. The reliability of the system is mostly about the reliability of the data.

There are four types of data in our system.

1. File-based command
2. File-based event
3. File-based state
4. State snapshot

Let us take a close look at how to ensure the reliability of each type of data.

State and snapshot can always be regenerated by replaying the event list. To improve the reliability of state and snapshot, we just need to ensure the event list has strong reliability.

Now let us examine command. On the face of it, event is generated from command. We might think providing a strong reliability guarantee for command should be sufficient. This seems to be correct at first glance, but it misses something important. Event generation is not guaranteed to be deterministic, and also it may contain random factors such as random numbers, external I/O, etc. So command cannot guarantee reproducibility of events.

Now it's time to take a close look at event. Event represents historical facts that introduce changes to the state (account balance). Event is immutable and can be used to rebuild the state.

From this analysis, we conclude that event data is the only one that requires a high-reliability guarantee. We will explain how to achieve this in the next section.

Consensus

To provide high reliability, we need to replicate the event list across multiple nodes. During the replication process, we have to guarantee the following properties.

1. No data loss.
2. The relative order of data within a log file remains the same across nodes.

To achieve those guarantees, consensus-based replication is a good fit. The consensus algorithm makes sure that multiple nodes reach a consensus on what the event list is. Let's use the Raft [17] consensus algorithm as an example.

The Raft algorithm guarantees that as long as more than half of the nodes are online, the append-only lists on them have the same data. For example, if we have 5 nodes and use the Raft algorithm to synchronize their data, as long as at least 3 (more than $\frac{1}{2}$) of the nodes are up as Figure 12.23 shows, the system can still work properly as a whole:

Figure 12.23: Raft

A node can have three different roles in the Raft algorithm.

1. Leader
2. Candidate
3. Follower

We can find the implementation of the Raft algorithm in the Raft paper. We will only cover the high level concepts here and not go into detail. In Raft, at most one node is the leader of the cluster and the remaining nodes are followers. The leader is respon-

sible for receiving external commands and replicating data reliably across nodes in the cluster.

With the Raft algorithm, the system is reliable as long as the majority of the nodes are operational. For example, if there are 3 nodes in the cluster, it could tolerate the failure of 1 node, and if there are 5 nodes, it can tolerate the failure of 2 nodes.

Reliable solution

With replication, there won't be a single point of failure in our file-based event sourcing architecture. Let's take a look at the implementation details. Figure 12.24 shows the event sourcing architecture with the reliability guarantee.

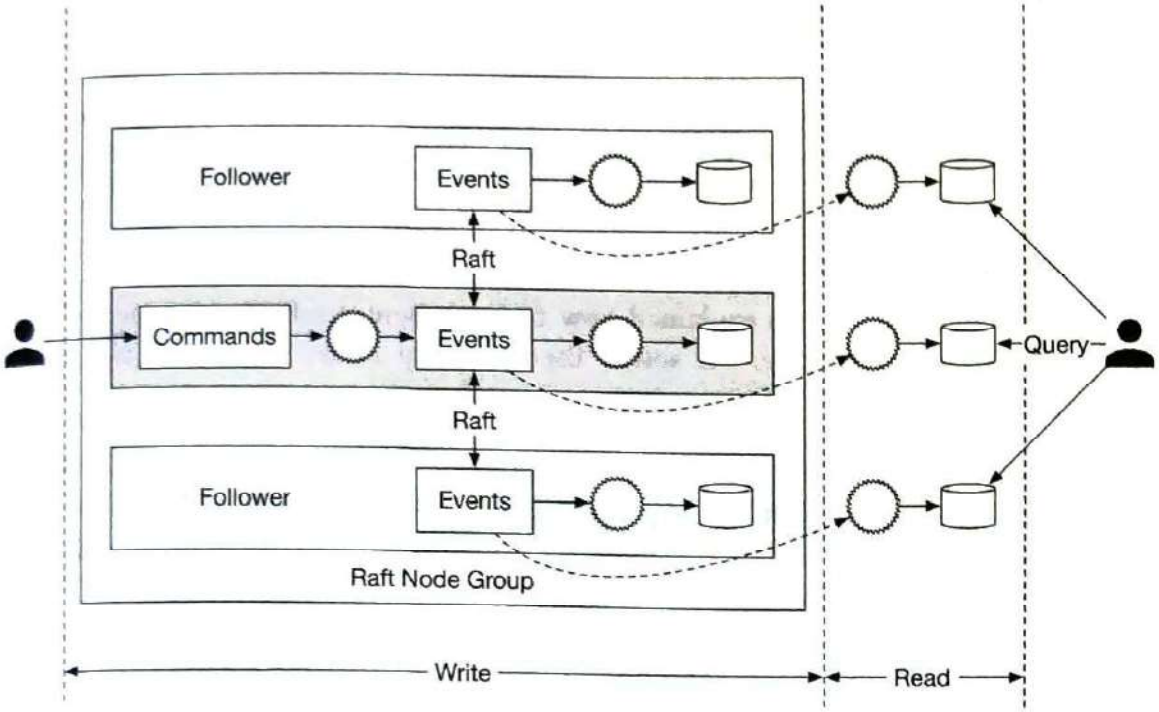


Figure 12.24: Raft node group

In Figure 12.24, we set up 3 event sourcing nodes. These nodes use the Raft algorithm to synchronize the event list reliably.

The leader takes incoming command requests from external users, converts them into events, and appends events into the local event list. The Raft algorithm replicates newly added events to the followers.

All nodes, including the followers, process the event list and update the state. The Raft algorithm ensures the leader and followers have the same event lists, while event sourcing guarantees all states are the same, as long as the event lists are the same.

A reliable system needs to handle failures gracefully, so let's explore how node crashes are handled.

If the leader crashes, the Raft algorithm automatically selects a new leader from the remaining healthy nodes. This newly elected leader takes responsibility for accepting

commands from external users. It is guaranteed that the cluster as a whole can provide continued service when a node goes down.

When the leader crashes, it is possible that the crash happens before the command list is converted to events. In this case, the client would notice the issue either by a timeout or by receiving an error response. The client needs to resend the same command to the newly elected leader.

In contrast, follower crashes are much easier to handle. If a follower crashes, requests sent to it will fail. Raft handles failures by retrying indefinitely until the crashed node is restarted or a new one replaces it.

Candidate: In this design, we use the Raft consensus algorithm to replicate the event list across multiple nodes. The leader receives commands and replicates events to other nodes.

Interviewer: Yes, the system is more reliable and fault-tolerant. However, in order to handle 1 million TPS, one server is not enough. How can we make the system more scalable?

Distributed event sourcing

In the previous section, we explained how to implement a reliable high-performance event sourcing architecture. It solves the reliability issue, but it has two limitations.

1. When a digital wallet is updated, we want to receive the updated result immediately. But in the CQRS design, the request/response flow can be slow. This is because a client doesn't know exactly when a digital wallet is updated and the client may need to rely on periodic polling.
2. The capacity of a single Raft group is limited. At a certain scale, we need to shard the data and implement distributed transactions.

Let's take a look at how those two problems are solved.

Pull vs push

In the pull model, an external user periodically polls execution status from the read-only state machine. This model is not real-time and may overload the wallet service if the polling frequency is set too high. Figure 12.25 shows the pulling model.

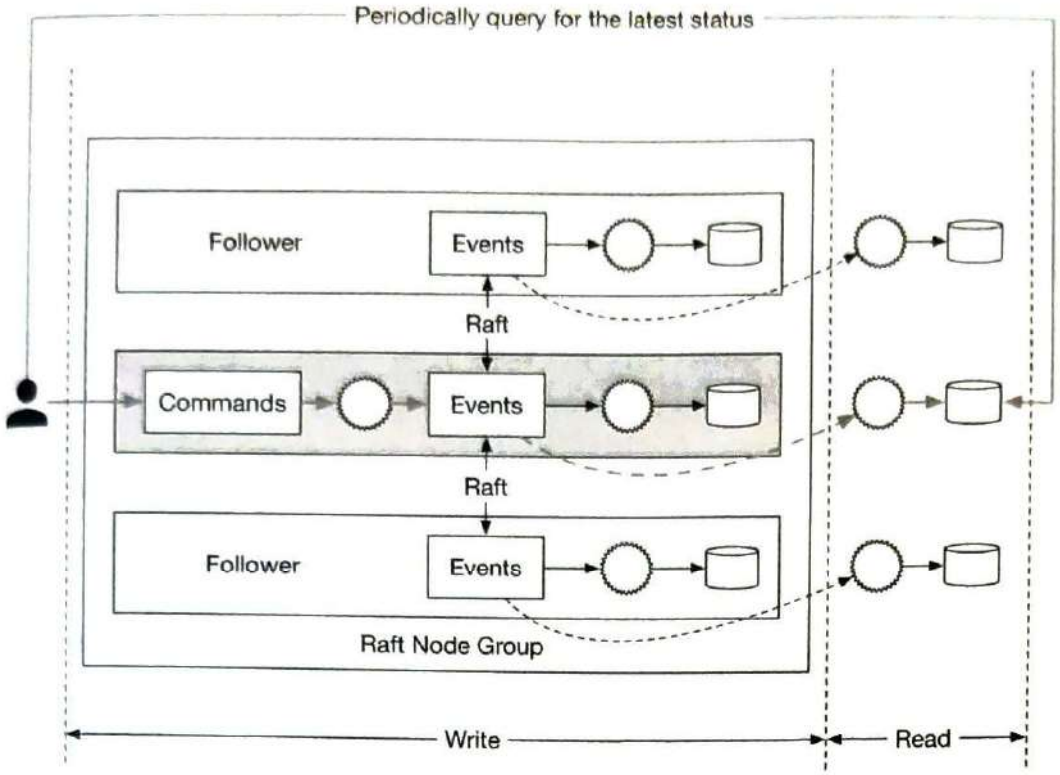


Figure 12.25: Periodical pulling

The naive pull model can be improved by adding a reverse proxy [18] between the external user and the event sourcing node. In this design, the external user sends a command to the reverse proxy, which forwards the command to event sourcing nodes and periodically polls the execution status. This design simplifies the client logic, but the communication is still not real-time.

Figure 12.26 shows the pull model with a reverse proxy added.

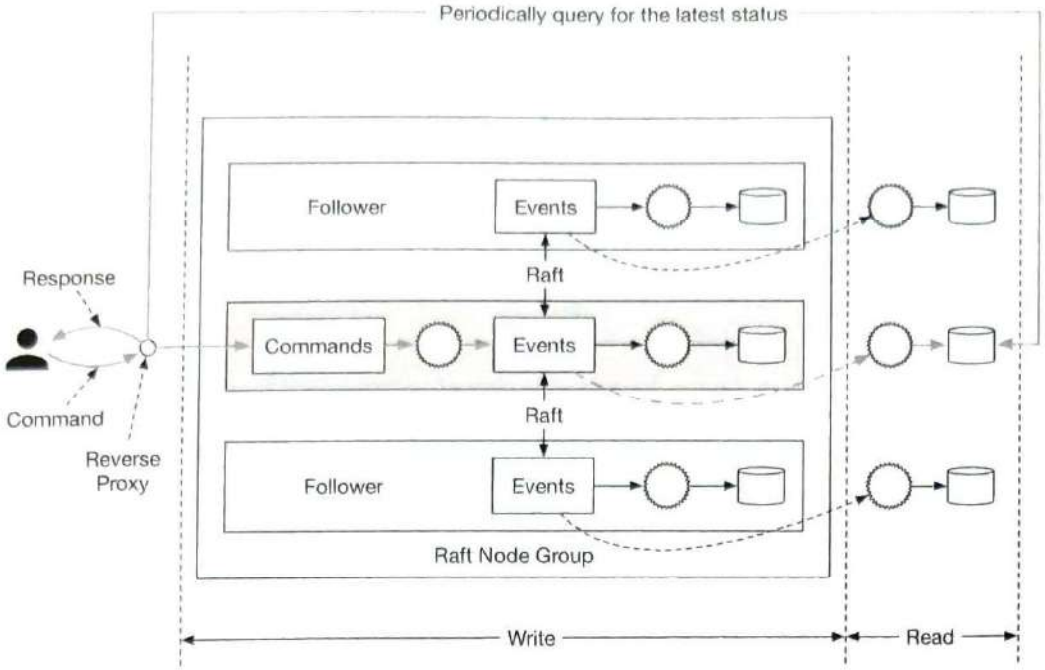


Figure 12.26: Pull model with reverse proxy

Once we have the reverse proxy, we could make the response faster by modifying the read-only state machine. As we mentioned earlier, the read-only state machine could have its own behavior. For example, one behavior could be that the read-only state machine pushes execution status back to the reverse proxy, as soon as it receives the event. This will give the user a feeling of real-time response.

Figure 12.27 shows the push-based model.

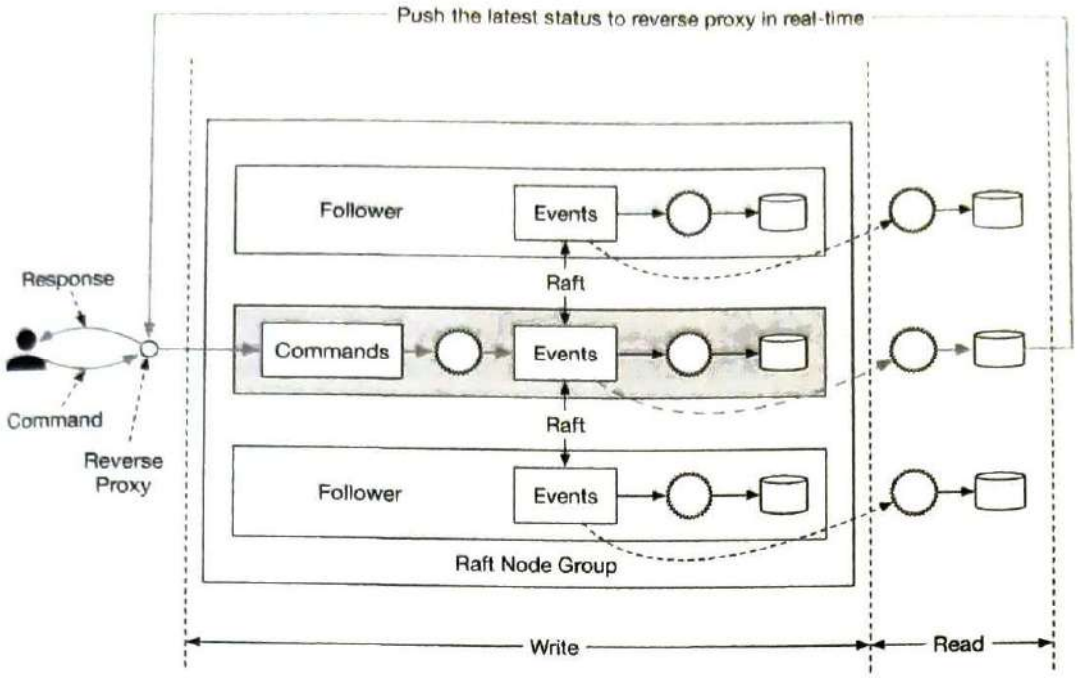


Figure 12.27: Push model

Distributed transaction

Once synchronous execution is adopted for every event sourcing node group, we can reuse the distributed transaction solution, TC/C or Saga. Assume we partition the data by dividing the hash value of keys by 2.

Figure 12.28 shows the updated design.

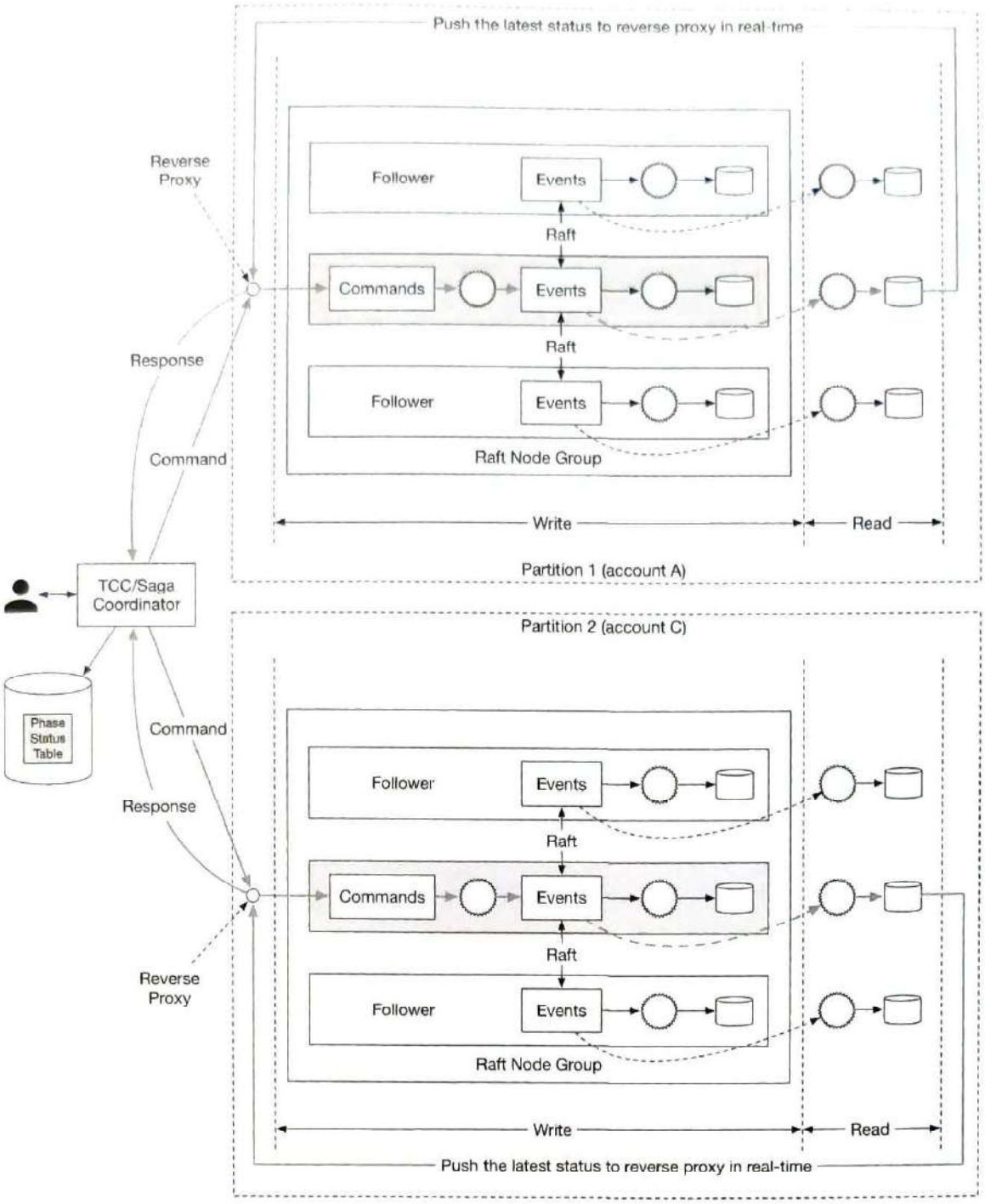


Figure 12.28: Final design

Let's take a look at how the money transfer works in the final distributed event sourcing architecture. To make it easier to understand, we use the Saga distributed transaction model and only explain the happy path without any rollback.

The money transfer operation contains 2 distributed operations: A:−$1 and C:+$1. The Saga coordinator coordinates the execution as shown in Figure 12.29:

1. User A sends a distributed transaction to the Saga coordinator. It contains two operations: A:−$1 and C:+$1.
2. Saga coordinator creates a record in the phase status table to trace the status of a transaction.
3. Saga coordinator examines the order of operations and determines that it needs to handle A:−$1 first. The coordinator sends A:−$1 as a command to Partition 1, which contains account A's information.
4. Partition 1's Raft leader receives the A−$1 command and stores it in the command list. It then validates the command. If it is valid, it is converted into an event. The Raft consensus algorithm is used to synchronize data across different nodes. The event (deducting $1 from A's account balance) is executed after synchronization is complete.
5. After the event is synchronized, the event sourcing framework of Partition 1 synchronizes the data to the read path using CQRS. The read path reconstructs the state and the status of execution.
6. The read path of Partition 1 pushes the status back to the caller of the event sourcing framework, which is the Saga coordinator.
7. Saga coordinator receives the success status from Partition 1.
8. The Saga coordinator creates a record, indicating the operation in Partition 1 is successful, in the phase status table.
9. Because the first operation succeeds, the Saga coordinator executes the second operation, which is C:+$1. The coordinator sends C:+$1 as a command to Partition 2 which contains account C's information.
10. Partition 2's Raft leader receives the C:+$1 command and saves it to the command list. If it is valid, it is converted into an event. The Raft consensus algorithm is used to synchronize data across different nodes. The event (add $1 to C's account) is executed after synchronization is complete.
11. After the event is synchronized, the event sourcing framework of Partition 2 synchronizes the data to the read path using CQRS. The read path reconstructs the state and the status of execution.
12. The read path of Partition 2 pushes the status back to the caller of the event sourcing framework, which is the Saga coordinator.
13. The Saga coordinator receives the success status from Partition 2.
14. The Saga coordinator creates a record, indicating the operation in Partition 2 is successful in the phase status table.
15. At this time, all operations succeed and the distributed transaction is completed. The Saga coordinator responds to its caller with the result.

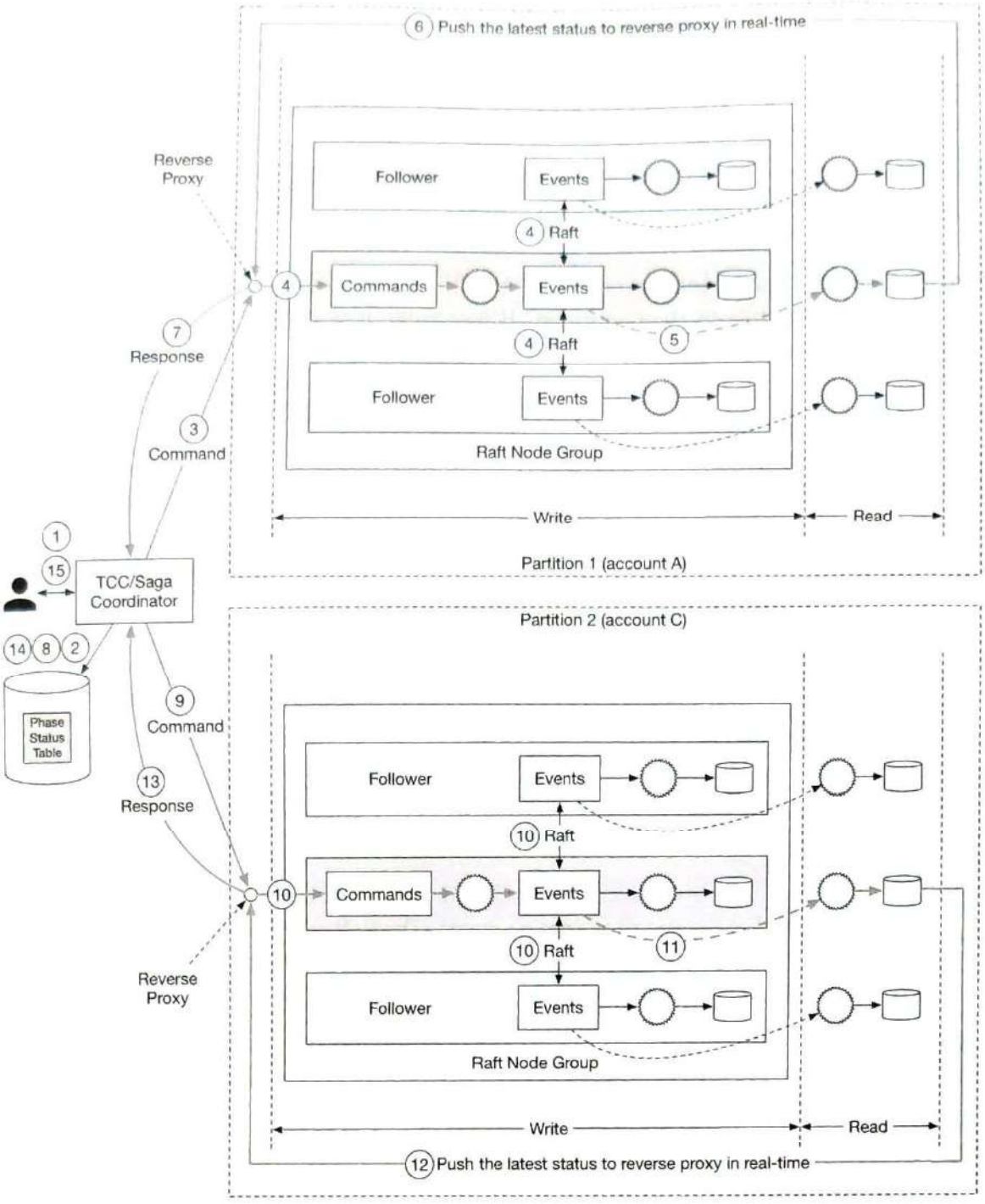


Figure 12.29: Final design in a numbered sequence

Step 4 - Wrap Up

In this chapter, we designed a wallet service that is capable of processing over 1 million payment commands per second. After a back-of-the-envelope estimation, we concluded that a few thousand nodes are required to support such a load.

In the first design, a solution using in-memory key-value stores like Redis is proposed. The problem with this design is that data isn't durable.

In the second design, the in-memory cache is replaced by transactional databases. To support multiple nodes, different transactional protocols such as 2PC, TC/C, and Saga are proposed. The main issue with transaction-based solutions is that we cannot conduct a data audit easily.

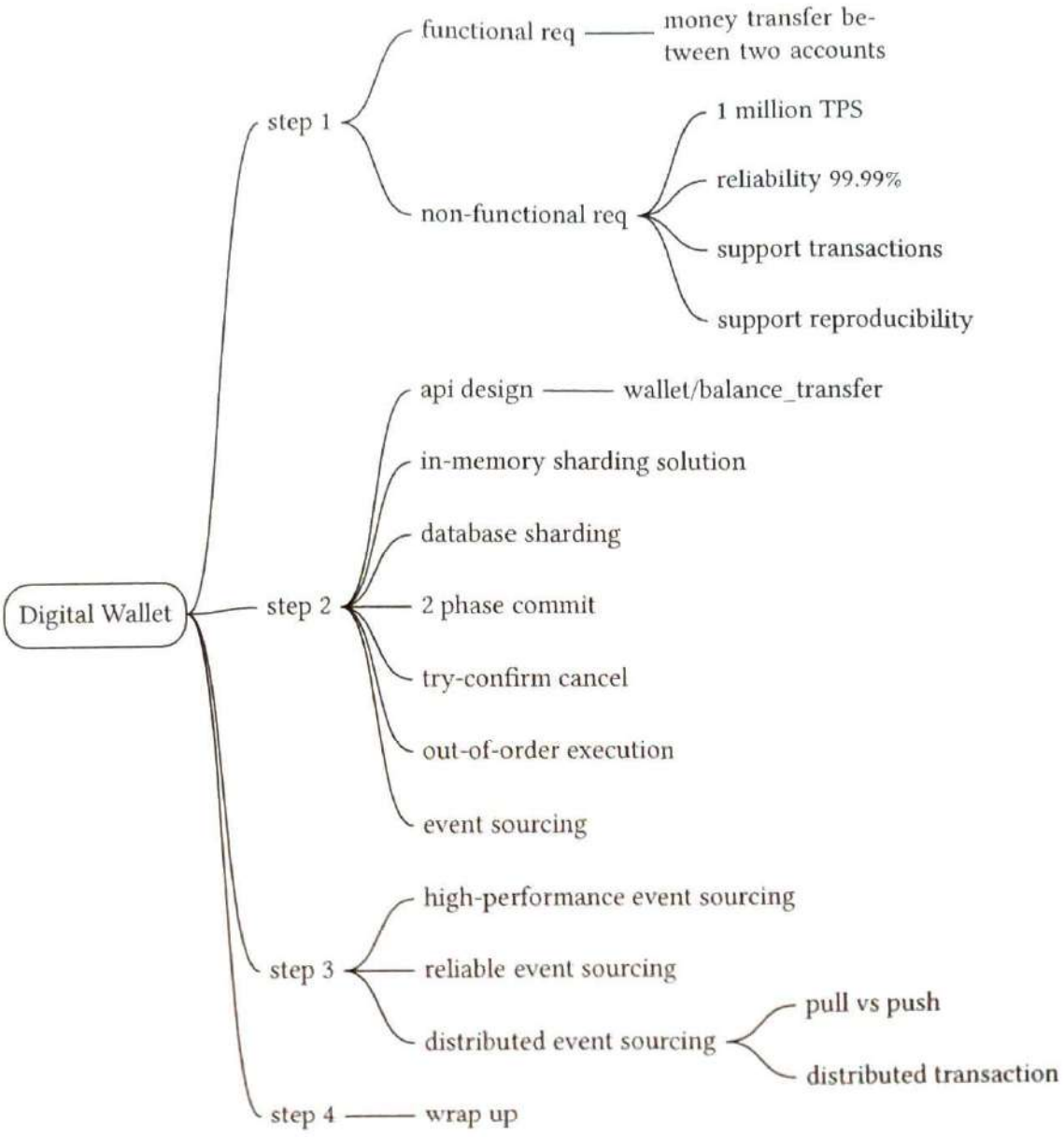
Next, event sourcing is introduced. We first implemented event sourcing using an external database and queue, but it's not performant. We improved performance by storing command, event, and state in a local node.

A single node means a single point of failure. To increase the system reliability, we use the Raft consensus algorithm to replicate the event list onto multiple nodes.

The last enhancement we made was to adopt the CQRS feature of event sourcing. We added a reverse proxy to change the asynchronous event sourcing framework to a synchronous one for external users. The TC/C or Saga protocol is used to coordinate Command executions across multiple node groups.

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

Chapter Summary



Reference Material

[1] Transactional guarantees. https://docs.oracle.com/cd/E17275_01/html/programmer_reference/rep_trans.html.

[2] TPC-E Top Price/Performance Results. http://tpc.org/tpce/results/tpce_price_perf_results5.asp?resulttype=all.

[3] ISO 4217 CURRENCY CODES. https://en.wikipedia.org/wiki/ISO_4217.

[4] Apache ZooKeeper. https://zookeeper.apache.org/.

[5] Martin Kleppmann. *Designing Data-Intensive Applications.* O'Reilly Media, 2017.

[6] X/Open XA. https://en.wikipedia.org/wiki/X/Open_XA.

[7] Compensating transaction. https://en.wikipedia.org/wiki/Compensating_transaction.

[8] SAGAS, HectorGarcia-Molina. https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf.

[9] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2003.

[10] Apache Kafka. https://kafka.apache.org/.

[11] CQRS. https://martinfowler.com/bliki/CQRS.html.

[12] Comparing Random and Sequential Access in Disk and Memory. https://deliveryimages.acm.org/10.1145/1570000/1563874/jacobs3.jpg.

[13] mmap. https://man7.org/linux/man-pages/man2/mmap.2.html.

[14] SQLite. https://www.sqlite.org/index.html.

[15] RocksDB. https://rocksdb.org/.

[16] Apache Hadoop. https://hadoop.apache.org/.

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

[18] Reverse proxy. https://en.wikipedia.org/wiki/Reverse_proxy.

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.

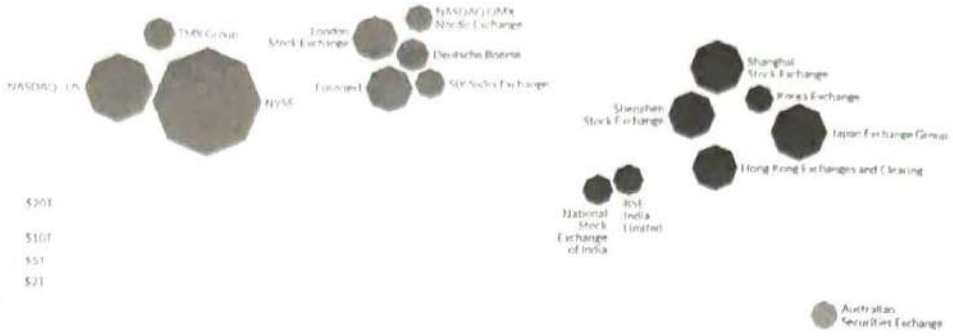


Figure 13.1: Largest stock exchanges (Source: [5])

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.

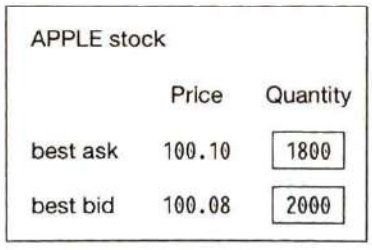


Figure 13.2: Level 1 data

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

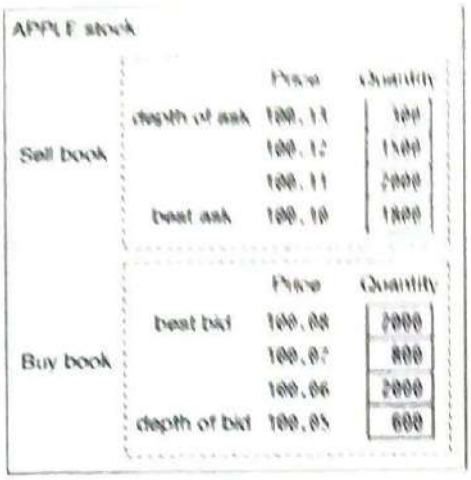


Figure 13.3: Level 2 data

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

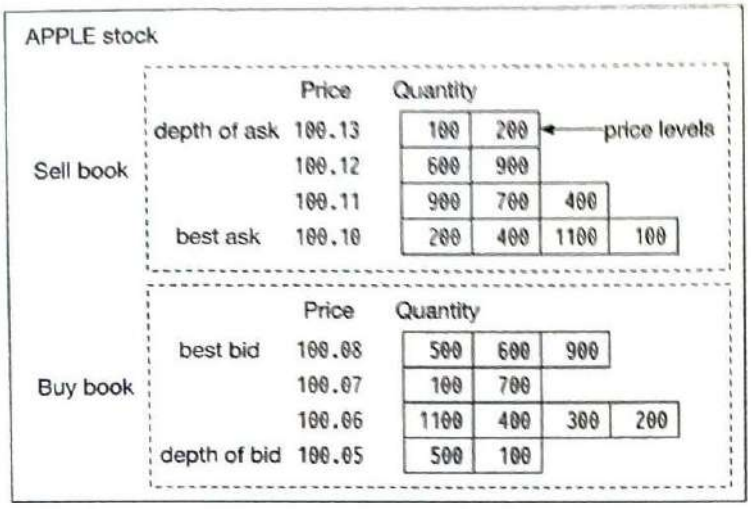


Figure 13.4: Level 3 data

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.

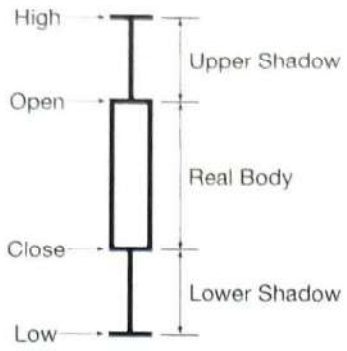


Figure 13.5: A single candlestick chart

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.

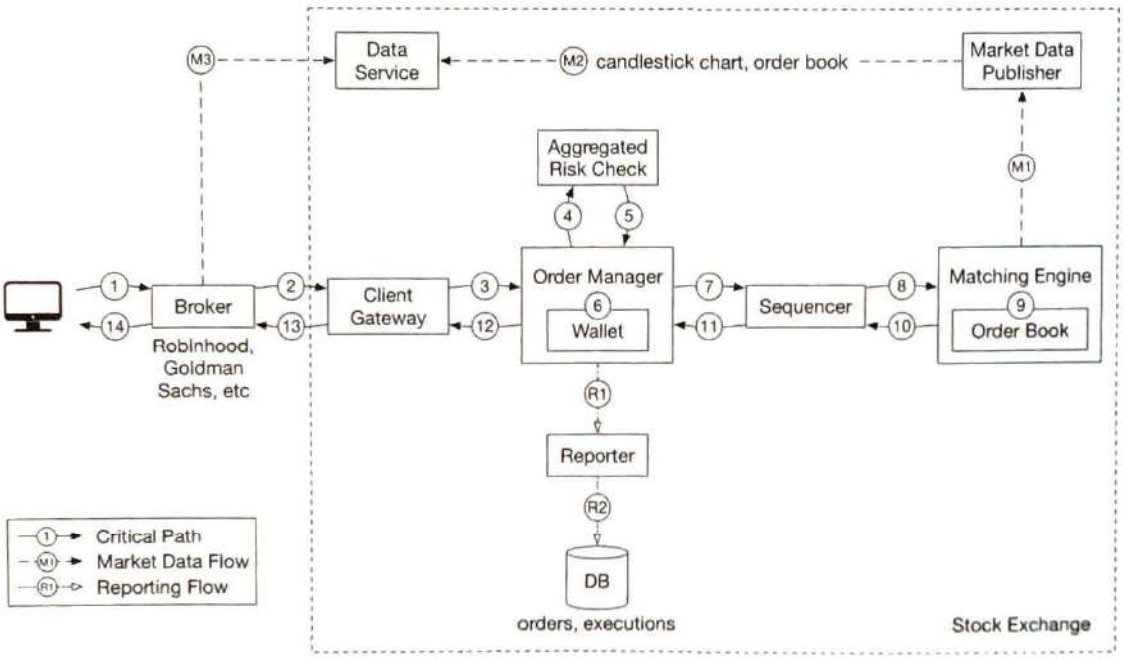


Figure 13.6: High-level design

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 $\sim$ 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 $\sim$ 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 $\sim$ 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$\sim$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.

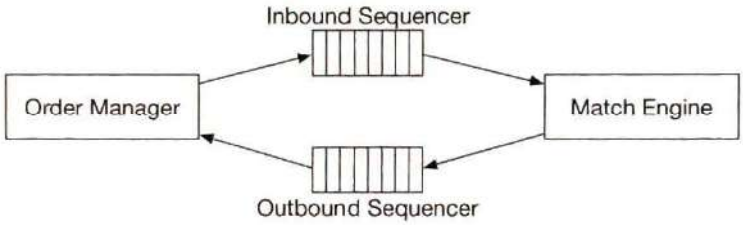


Figure 13.7: Inbound and outbound sequencers

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.

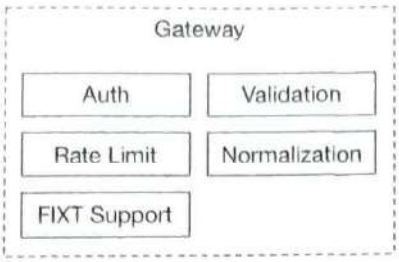


Figure 13.8: Client gateway components

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].

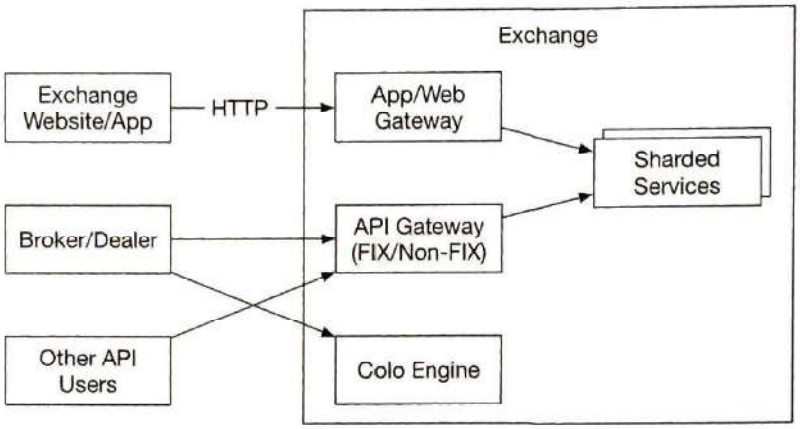


Figure 13.9: Client gateway

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.

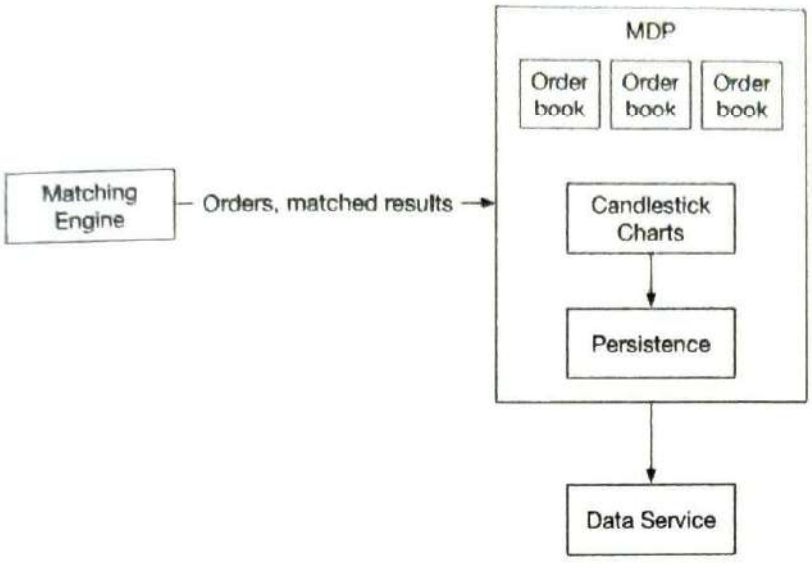


Figure 13.10: Market Data Publisher

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.

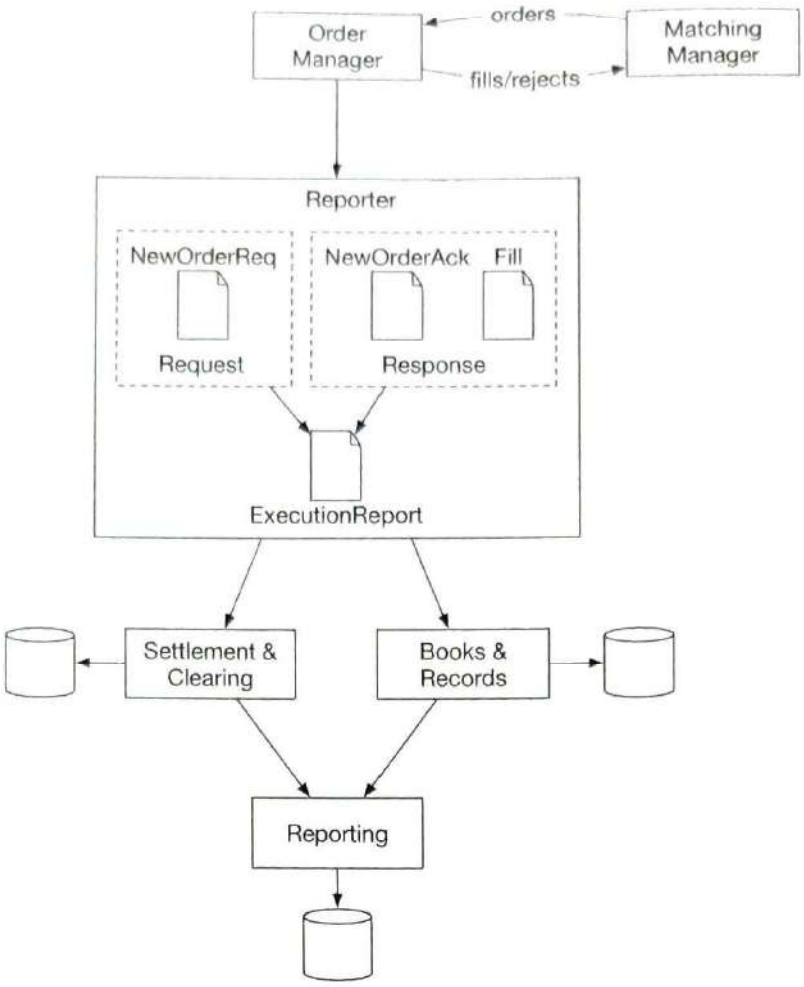


Figure 13.11: Reporter

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.

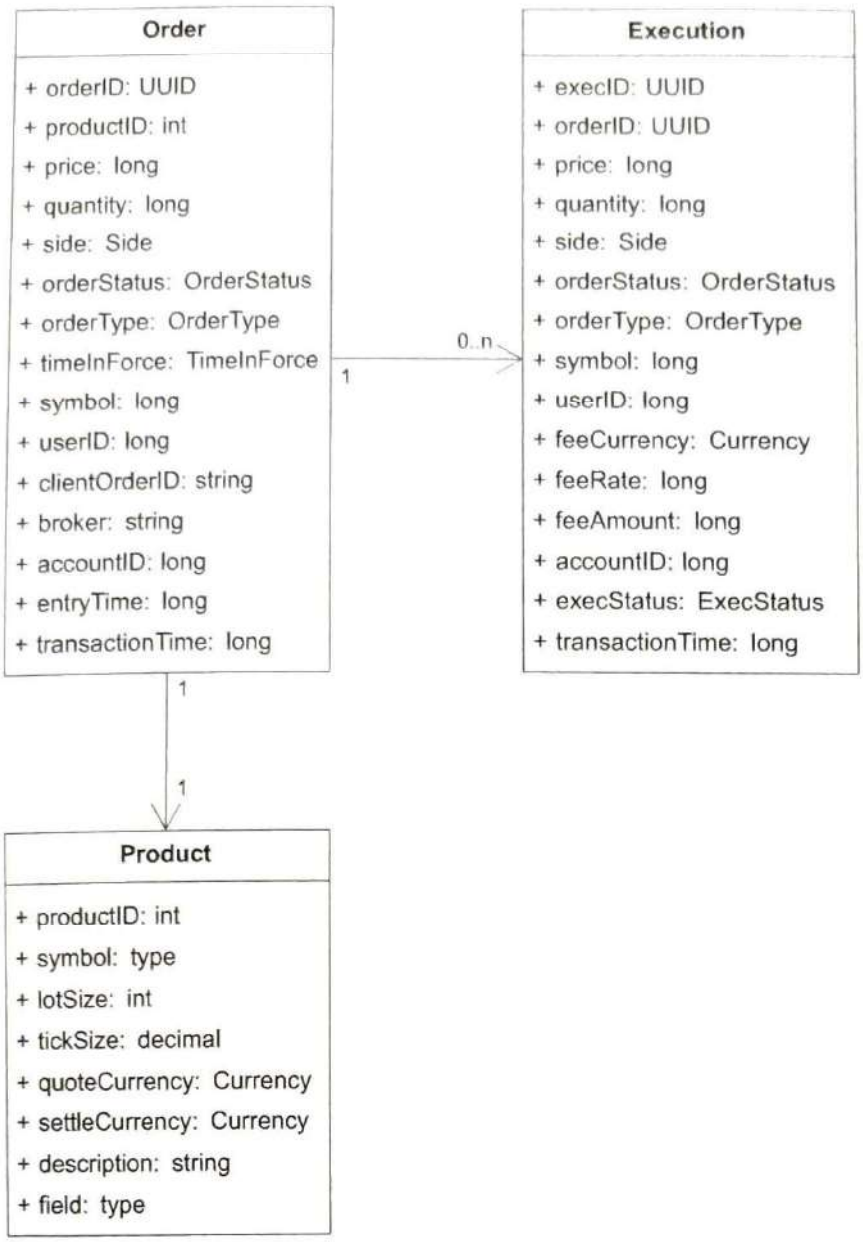


Figure 13.12: Product, order, execution

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.

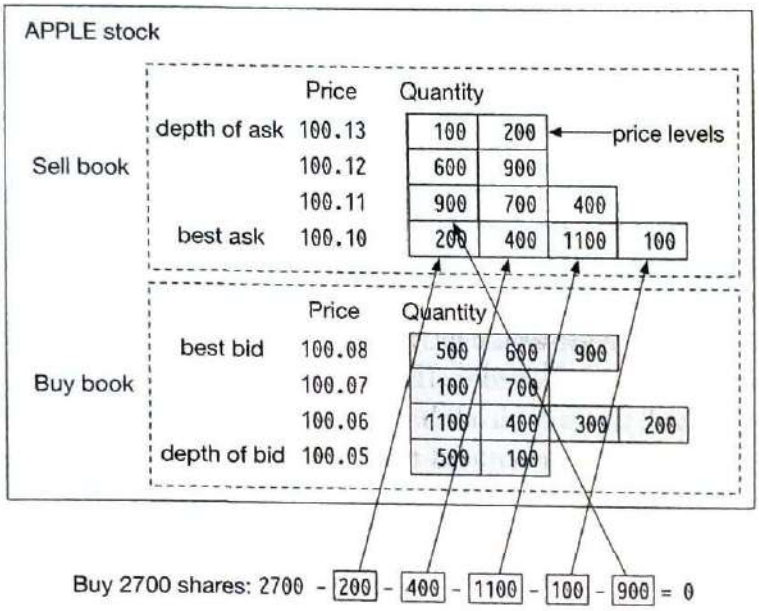


Figure 13.13: Limit order book illustrated

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.

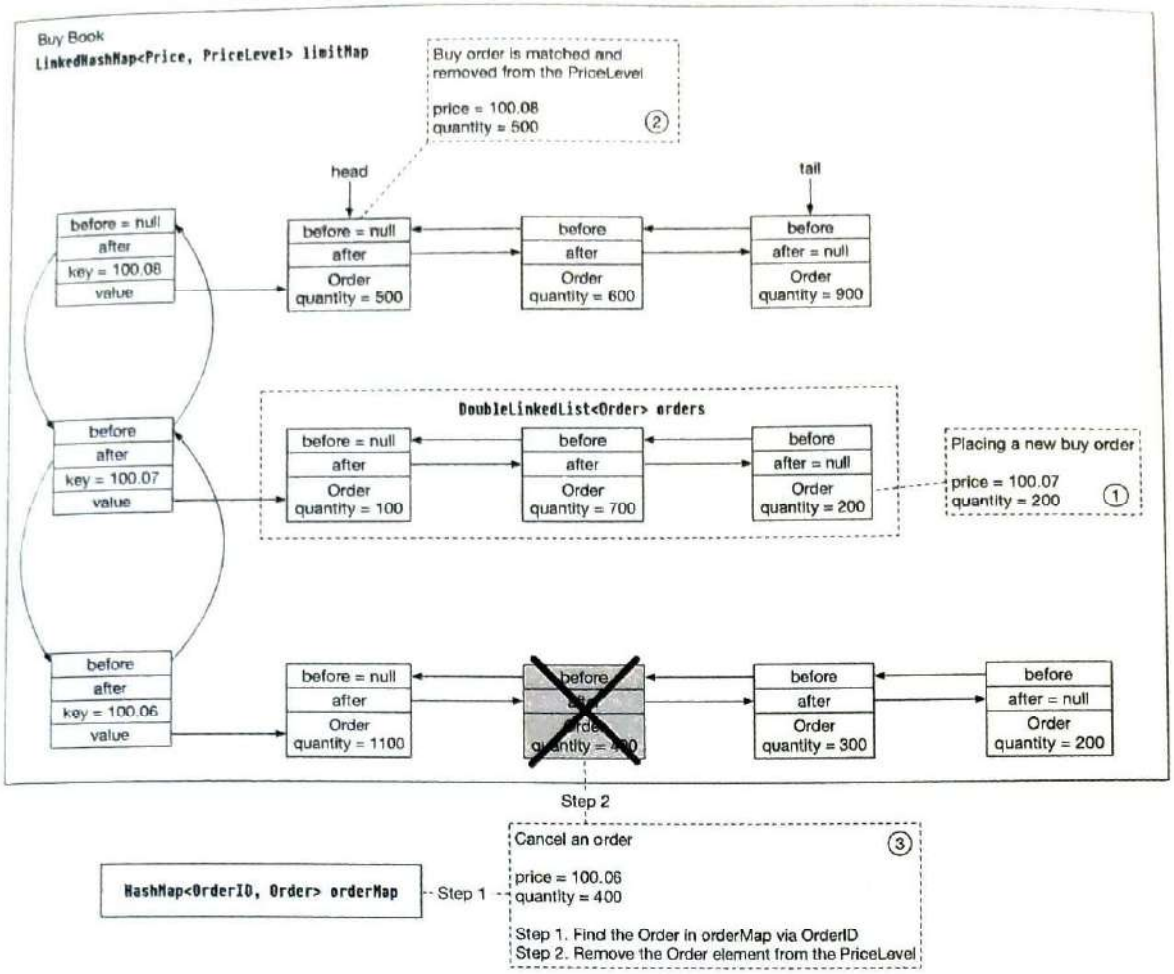


Figure 13.14: Place, match, and cancel an order in $O(1)$

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 review the first point. As shown in the high-level design, the critical trading path includes the following:

$$gateway \rightarrow order\ manager \rightarrow sequencer \rightarrow matching\ engine$$

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:

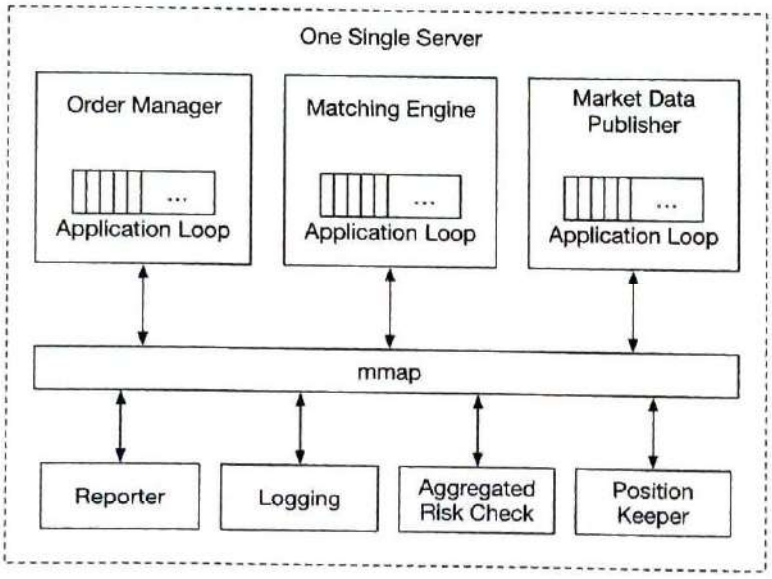


Figure 13.15: A low latency single server exchange design

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).

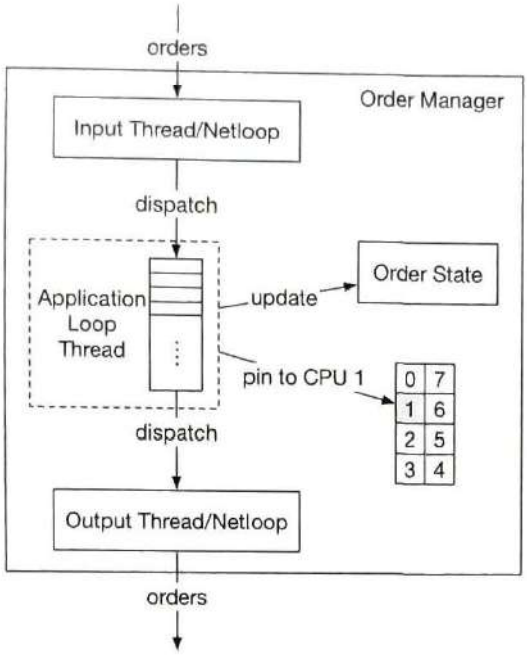


Figure 13.16: Application loop thread in Order Manager

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.

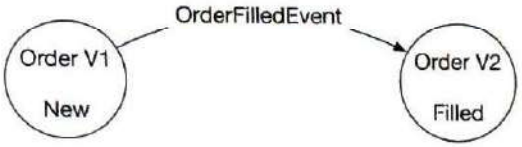
Modern exchanges take advantage of this to eliminate as much disk access from the critical path as possible. `mmap(2)` is used in the server to implement a message bus over which the components on the critical path communicate. The communication pathway has no network or disk access, and sending a message 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.



Order

Version	OrderStatus
V1	New
V2	Filled

Non Event Sourcing

Event

Event Sequence	Event Type
100	NewOrderEvent
101	OrderFilledEvent

Event Sourcing

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.

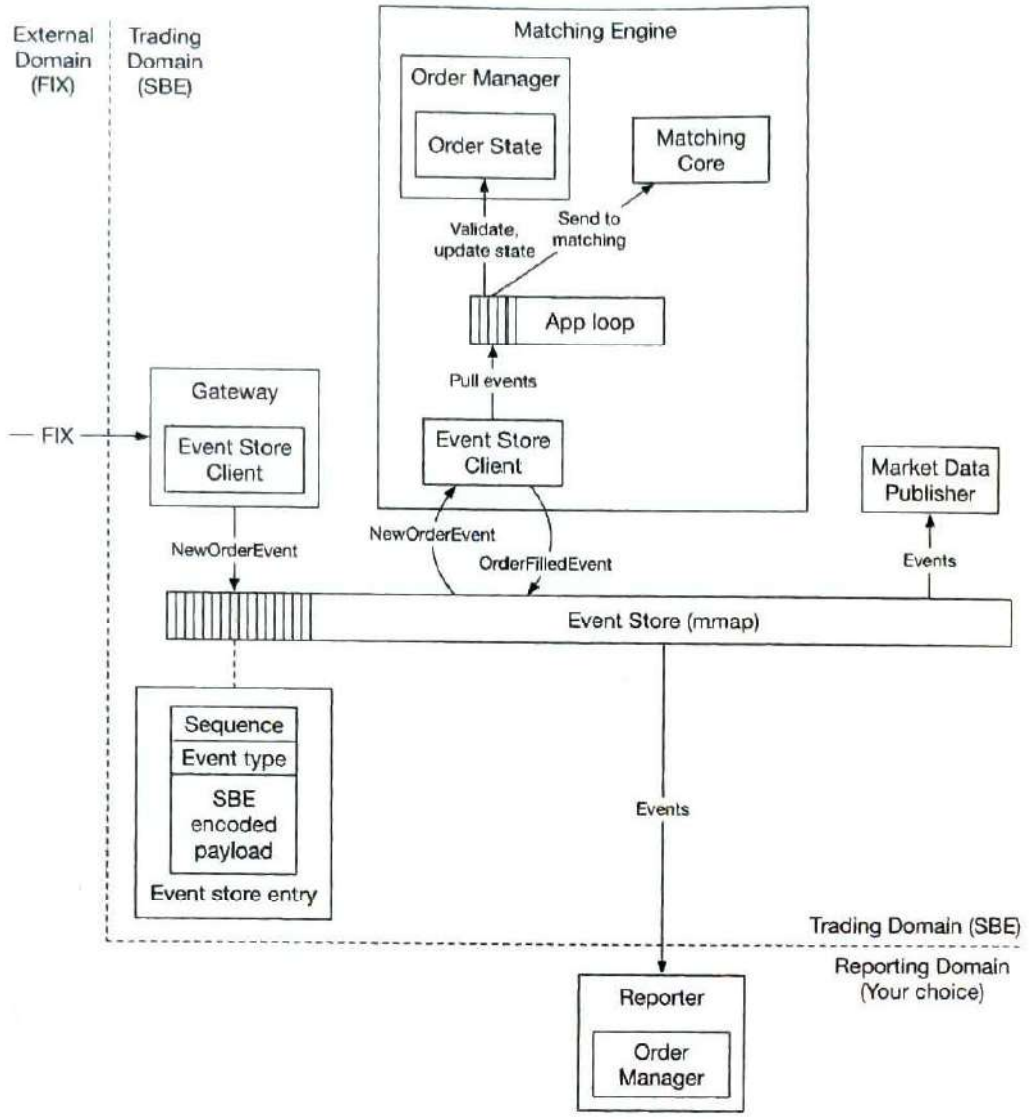


Figure 13.18: An event sourcing design

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.

- 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.

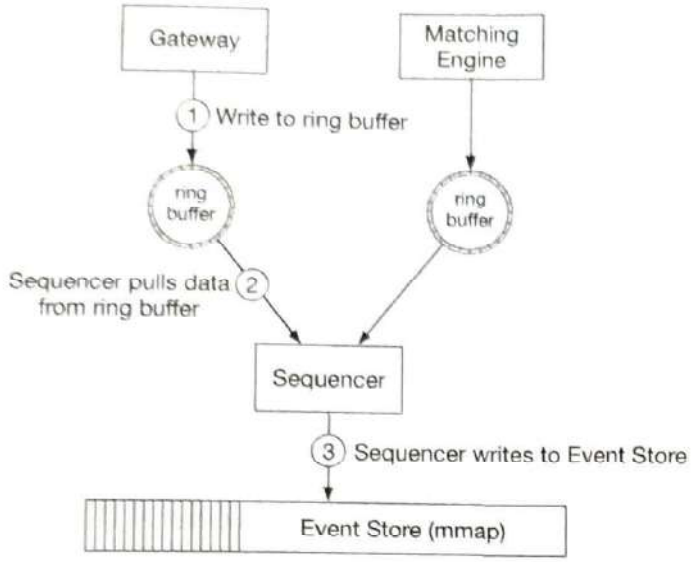


Figure 13.19: Sample design of Sequencer

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.

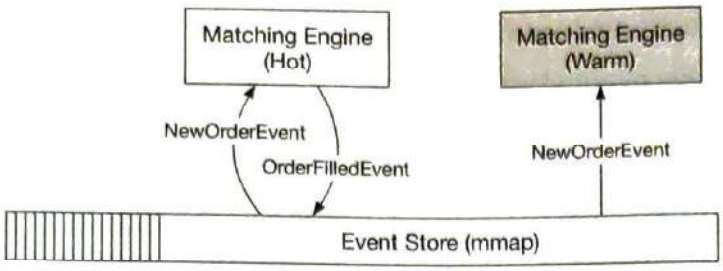


Figure 13.20: Hot-warm matching engine

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.

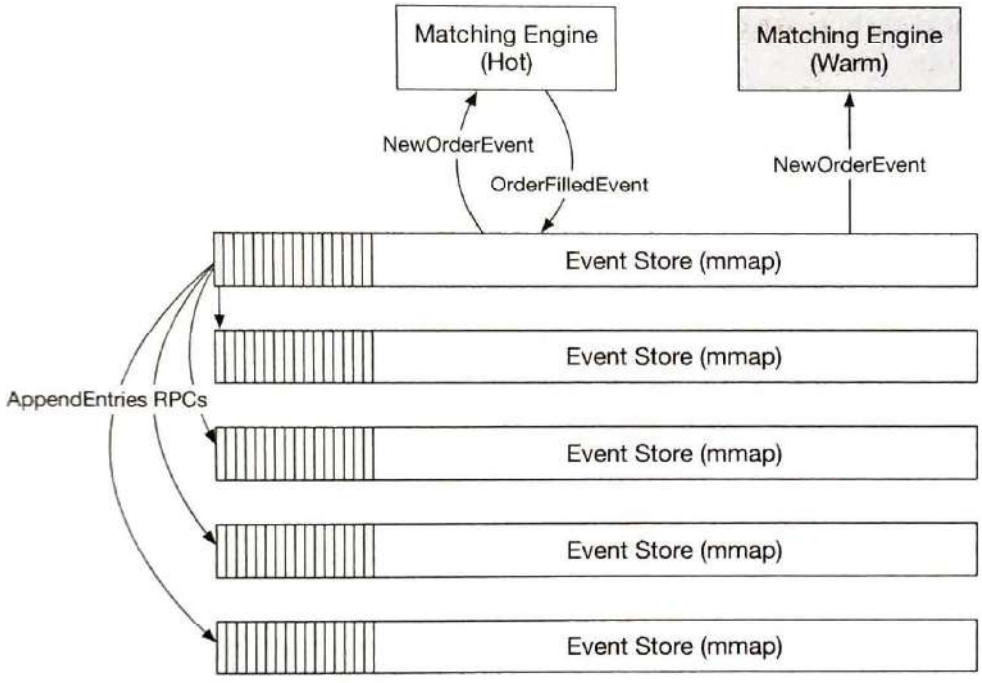


Figure 13.21: Event replication in Raft cluster

Let's briefly examine the leader election process. The leader sends heartbeat messages (`AppendEntries` 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.

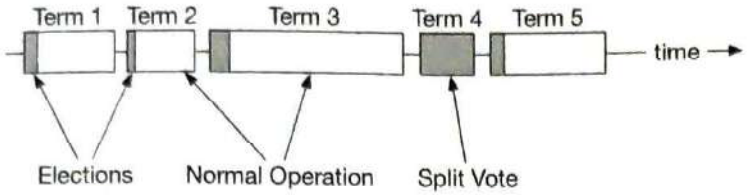


Figure 13.22: Raft terms (Source: [23])

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.

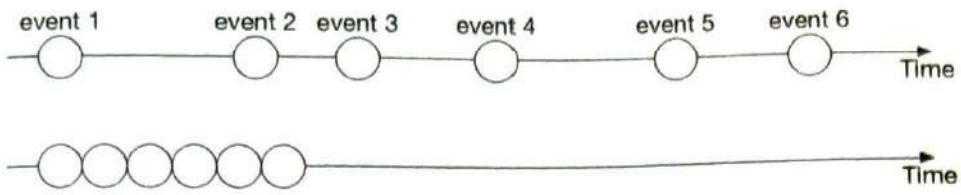


Figure 13.23: Time in event sourcing

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.

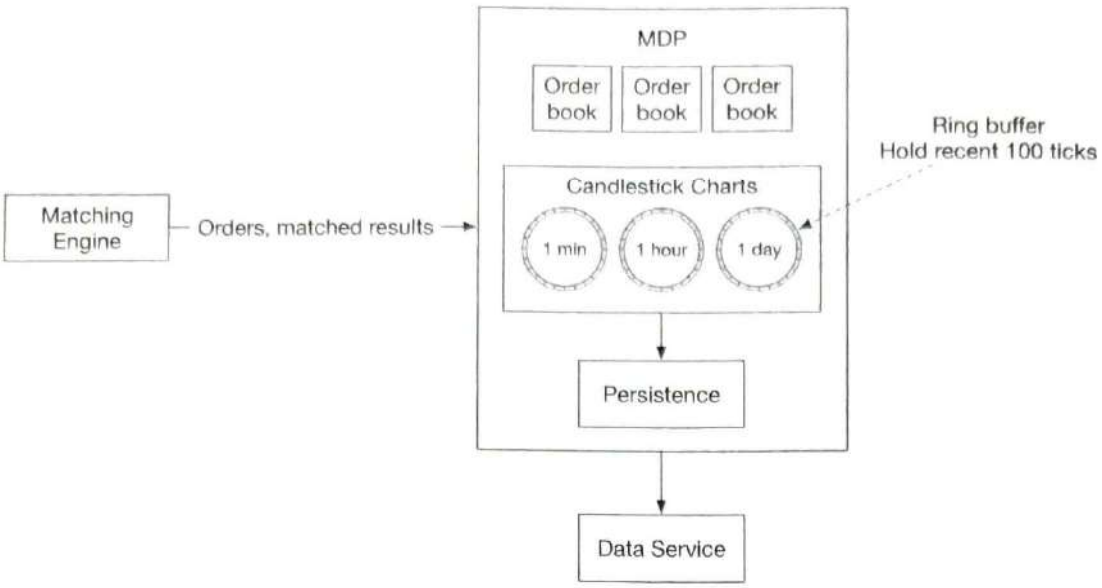


Figure 13.24: Market Data Publisher

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

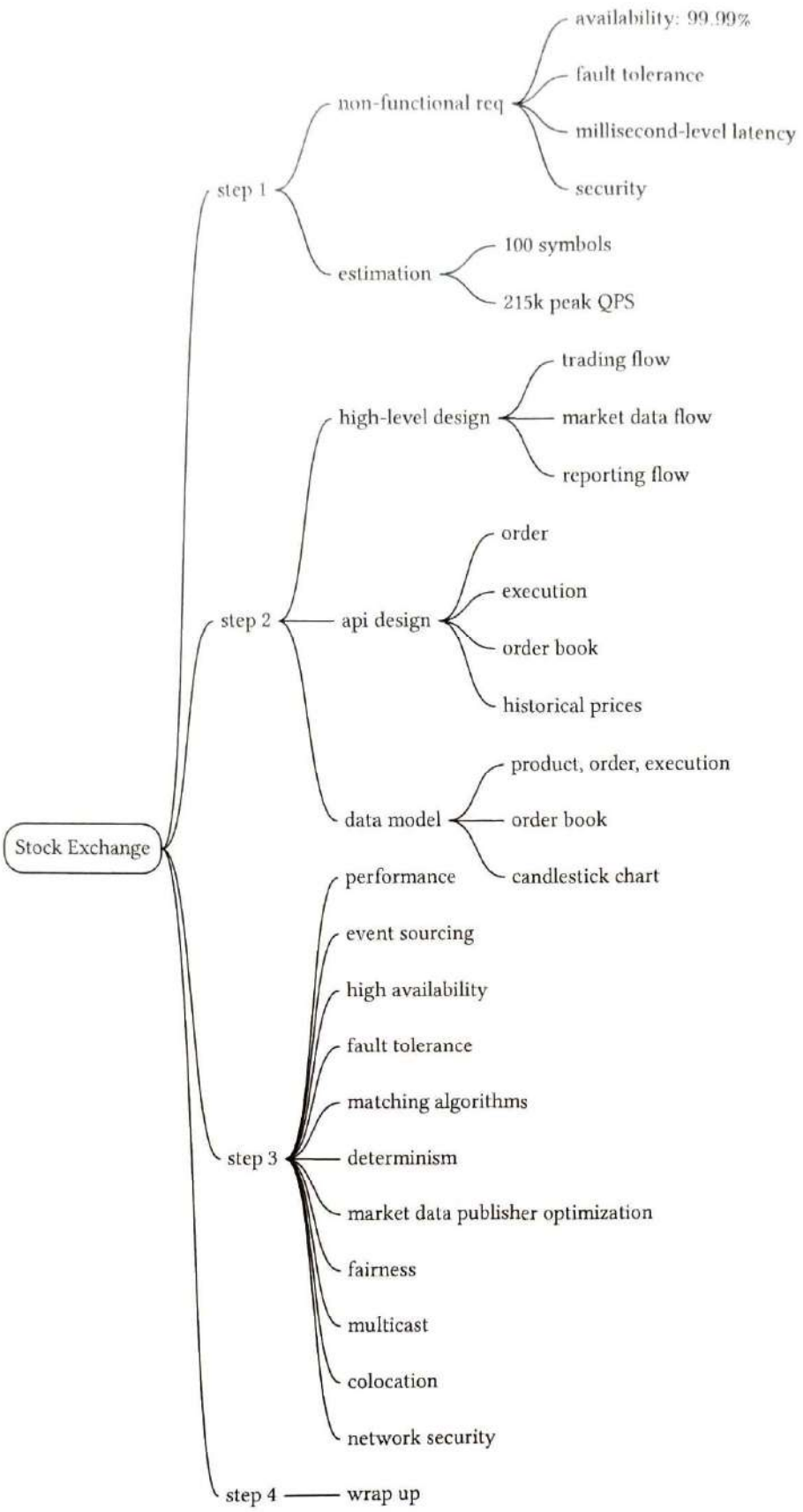
After reading this chapter, you may come to the conclusion that an ideal deployment model for a big exchange is to put everything on a single gigantic server or even one single process. Indeed, this is exactly how some exchanges are designed!

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/.

Afterword

Congratulations! You have completed this interview guide. You have accumulated skills and knowledge with which to design complex systems. Not everyone has the discipline to do what you have done, to learn what you have learned. Take a moment to pat yourself on the back. Your hard work will pay off.

Landing your dream job is a long journey and requires lots of time and effort. Practice makes perfect. Best of luck!

Thank you for buying and reading this book. Without readers like you, our work would not exist. We hope you have enjoyed the book!

If you have comments or questions about this book, feel free to send us an email at hi@bytebytego.com. If you notice any errors, please let us know so we can make corrections for the next edition. Thank you!

Join the community

We created a members-only Discord group. It is designed for community discussions on the following topics:

- System design fundamentals.
- Showcasing design diagrams and getting feedback.
- Finding mock interview buddies.
- General chat with community members.

Come join us and introduce yourself to the community, today! Use the link below or scan the QR code.

http://bit.ly/systemdiscord

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Made in United States
North Haven, CT
20 September 2022

This book is fantastic. It is a great continuation of the first book. I strongly recommend it to anyone who is studying for system design interviews.

– Sunny Patel, Software Engineering Manager at *Microsoft*

I was a Tech Lead at FAANG, but needed help in getting up to speed with unfamiliar domains. If you put in the work, this book will help you acquire the breadth and depth of knowledge you need for talking about bottlenecks and alternatives, as expected of a Tech Lead.

– Herbert Degano, Staff Software Engineer at *Coinbase*

What's inside?

- An insider's take on what interviewers really look for and why.
- A 4-step framework for solving any system design interview question.
- 13 real system design interview questions with detailed solutions.
- 303 diagrams to visually explain how different systems work.

About the Authors

Alex Xu is a software engineer and author. His book 'System Design Interview - An Insider's Guide (Volume 1)' is an Amazon bestseller, which has been translated into six languages. He has worked at Twitter, Apple, and Zynga.

Sahn Lam is a software engineer with decades of experience in building scalable systems at high-growth companies like Discord, Zynga, and NetApp. He has designed, built, operated, and optimized many interesting distributed systems used by millions of users.

You can connect with Alex on social media, where he shares system design interview tips every week.

twitter.com/alexxubyte

bit.ly/linkedinaxu

ISBN 9781736049112