28
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 1 shows this process.

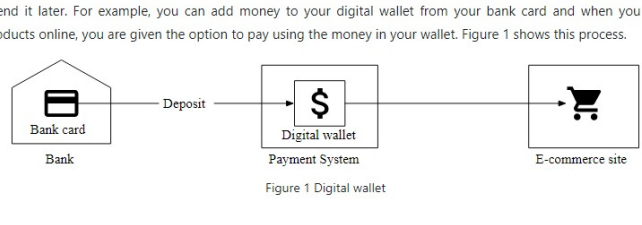


Figure 1 Digital wallet

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 2 shows a cross-wallet balance transfer operation.

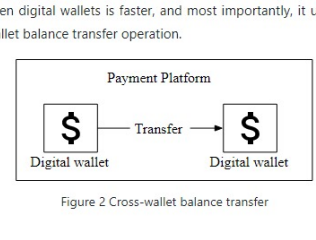


Figure 2 Cross-wallet balance transfer

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 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 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 the Payment System chapter.

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 3 shows the in-memory solution.

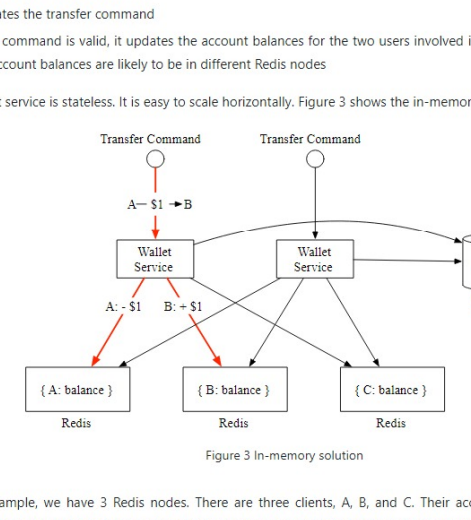


Figure 3 In-memory solution

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 4 shows the architecture. This time, clients A, B, and C are partitioned into 3 relational databases, rather than in 3 Redis nodes.

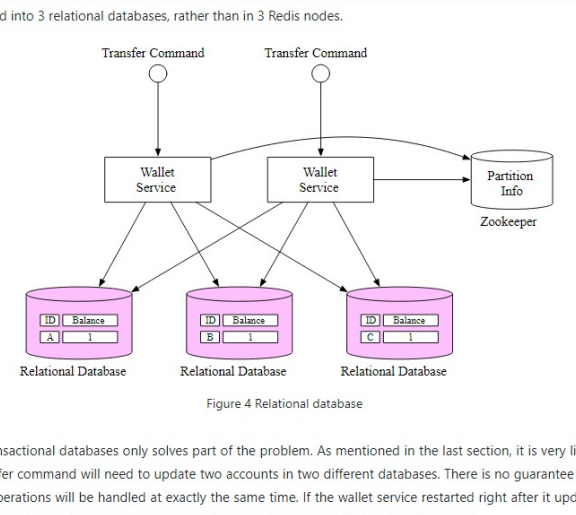


Figure 4 Relational database

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 5.

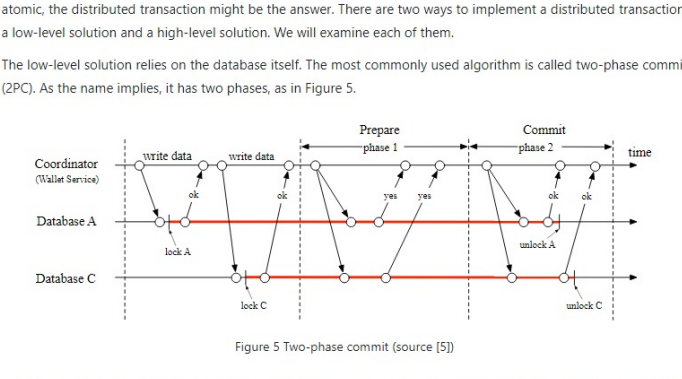


Figure 5 Two-phase commit (source [5])

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 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:
    1. If all databases reply with a "yes", the coordinator asks all databases to commit the transaction they have received.
    2. 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 6.

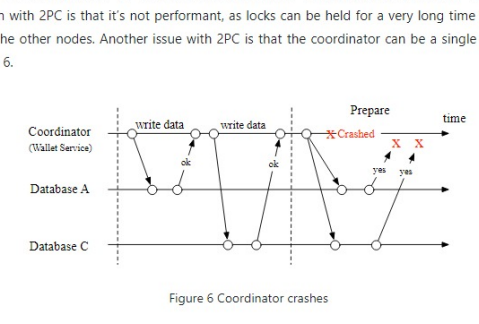


Figure 6 Coordinator crashes

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:
    1. If all databases reply with "yes", the coordinator asks all databases to confirm the operation, which is the Try-Confirm process.
    2. 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 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 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 7. The thick line indicates that a lock is held by the transaction.

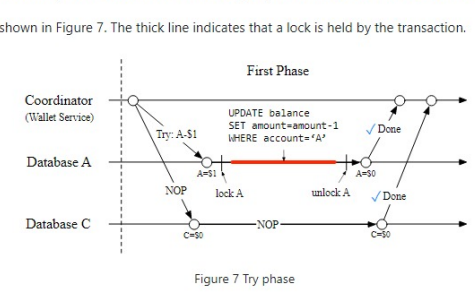


Figure 7 Try phase

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. 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 8.

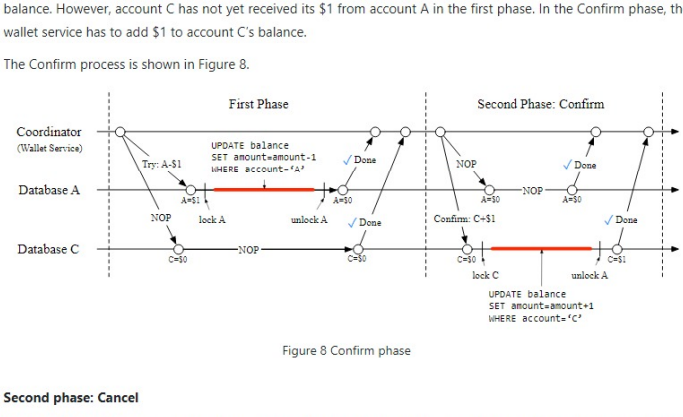


Figure 8 Confirm phase

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 9.

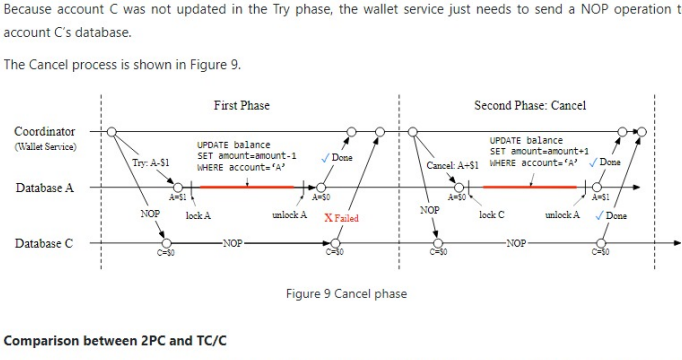


Figure 9 Cancel phase

Comparison between 2PC and TC/C

Table 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	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 3 2PC vs 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 10.

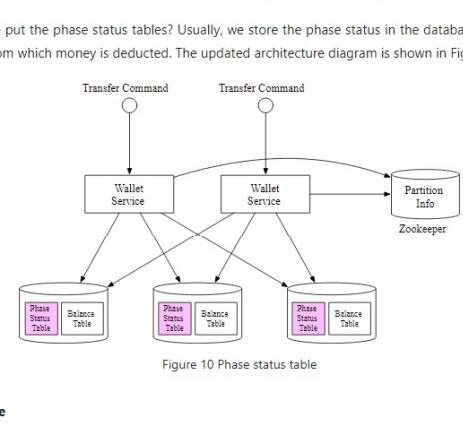


Figure 10 Phase status table

Unbalanced state

Have you noticed that by the end of the Try phase, $1 is missing (Figure 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 11.

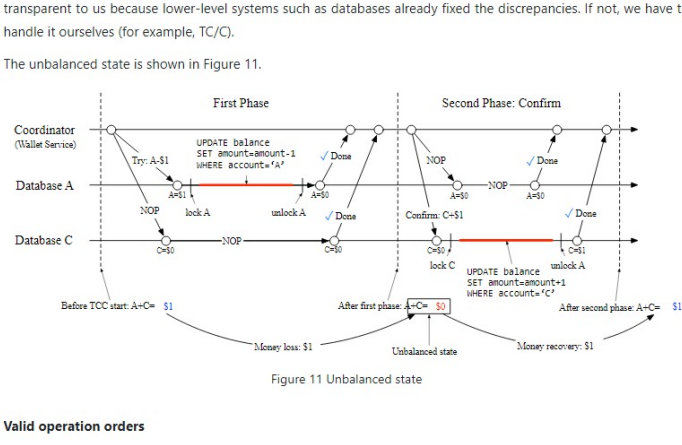


Figure 11 Unbalanced state

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 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 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.

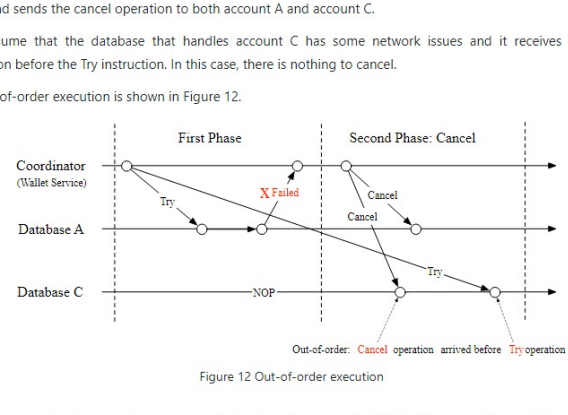


Figure 12 Out-of-order execution

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 2*n* operations: *n* for the normal case and another *n* for the compensating transaction during rollback.

It is easier to understand this by using an example. Figure 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, we have to put the deduction operation before the addition operation.

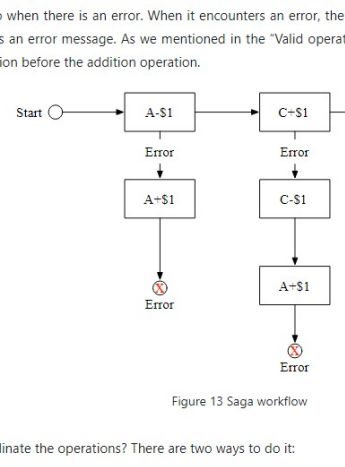


Figure 13 Saga workflow

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 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 5 TC/C vs Saga

Which one should we use in practice? The answer depends on the latency requirement. As Table 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 "transferred $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 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.

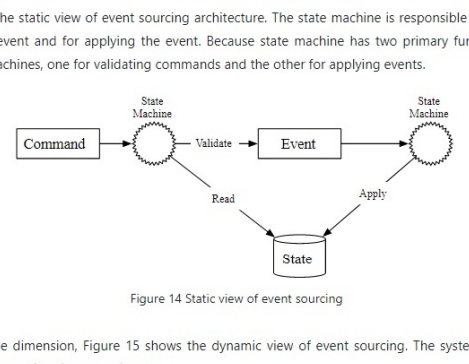


Figure 14 Static view of event sourcing

If we add the time dimension, Figure 15 shows the dynamic view of event sourcing. The system keeps receiving commands and processing them, one by one.

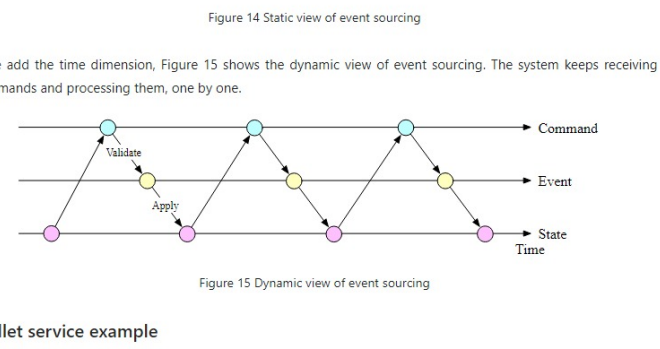


Figure 15 Dynamic view of event sourcing

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 16.

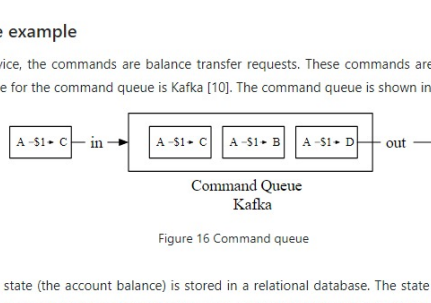


Figure 16 Command queue

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 sufficient balance. If yes, the state machine generates an event for each account. For example, if the command is "A->$1->C", the state machine generates two events: "A:-$1" and "C:+$1".

Figure 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.

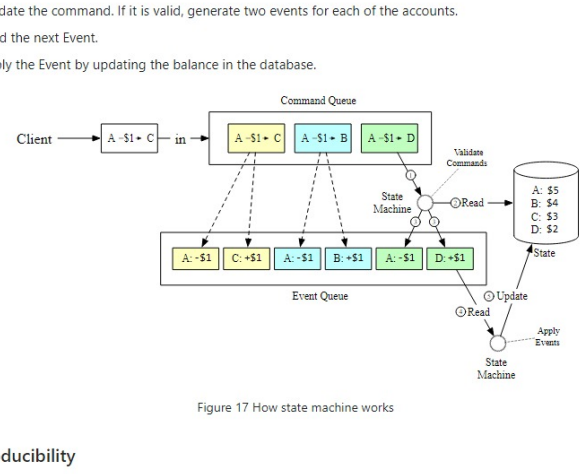


Figure 17 How state machine works

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

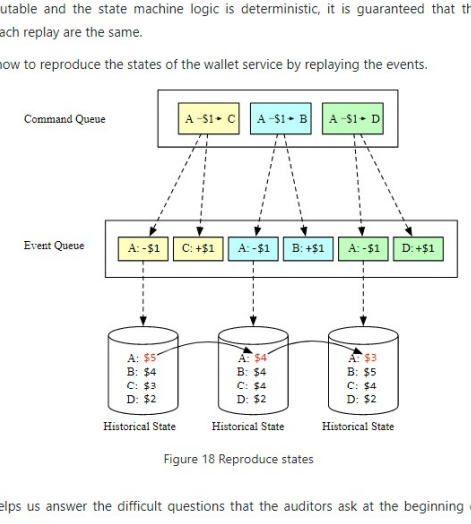


Figure 18 Reproduce states

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 19 shows a classic CQRS architecture.

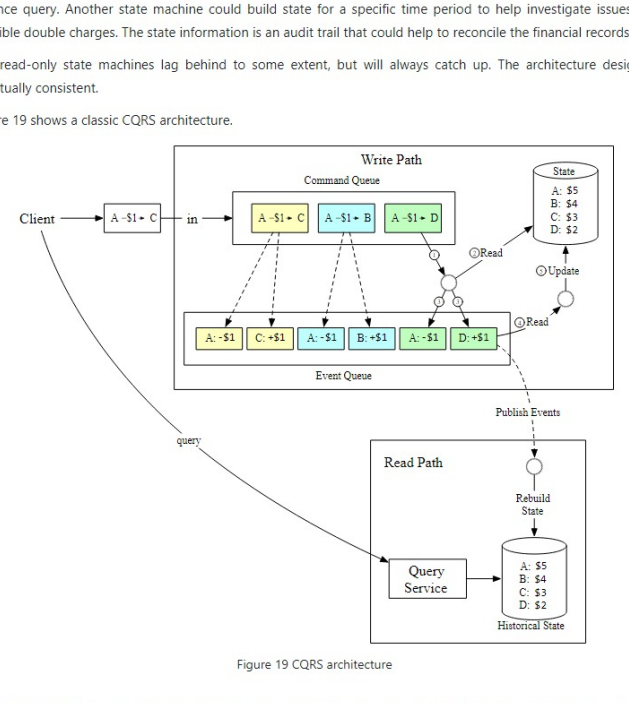


Figure 19 CQRS architecture

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 ACM Queue 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 20 shows the file-based command and event storage.

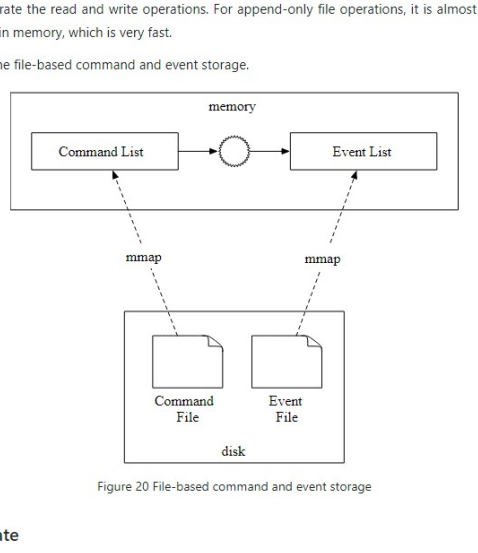


Figure 20 File-based command and event storage

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

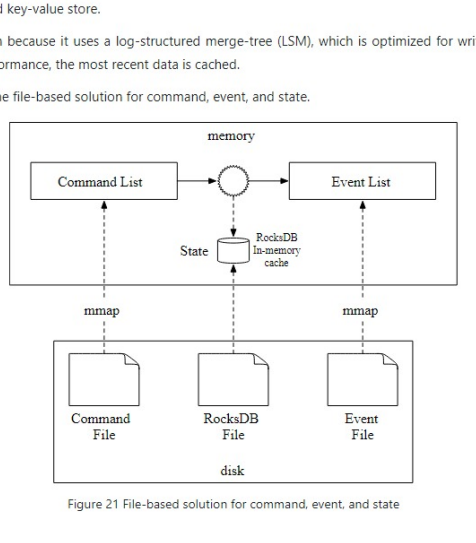


Figure 21 File-based solution for command, event, and state

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 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.

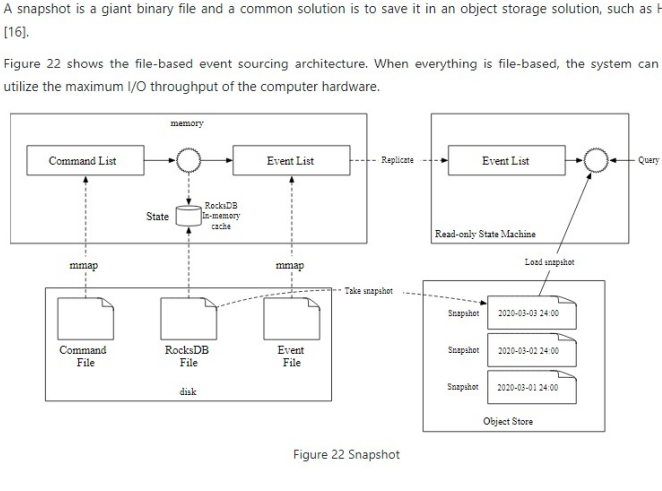


Figure 22 Snapshot

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 half) of the nodes are up as Figure 23 shows, the system can still work properly as a whole:

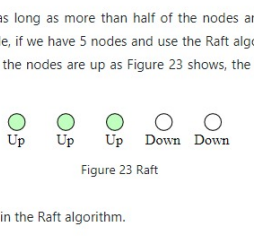


Figure 23 Raft

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 responsible 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 24 shows the event sourcing architecture with the reliability guarantee.

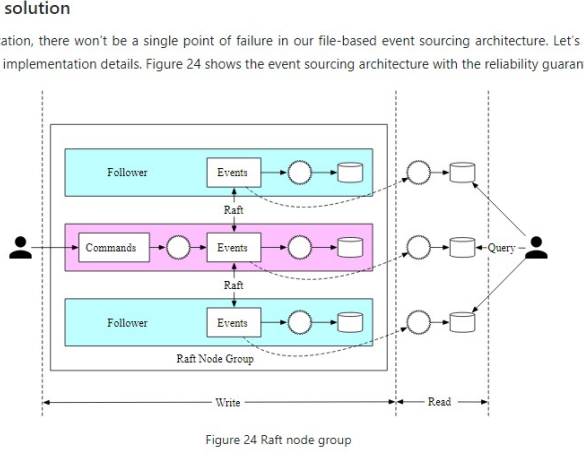


Figure 24 Raft node group

In Figure 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 25 shows the pulling model.

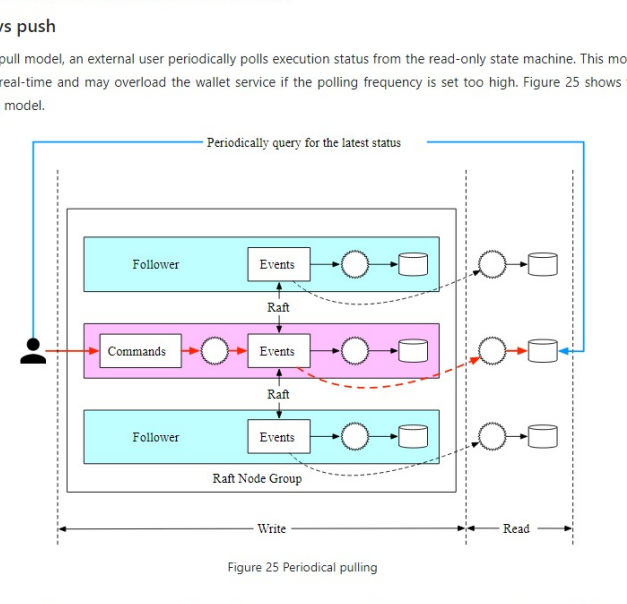


Figure 25 Periodical pulling

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 26 shows the pull model with a reverse proxy added.

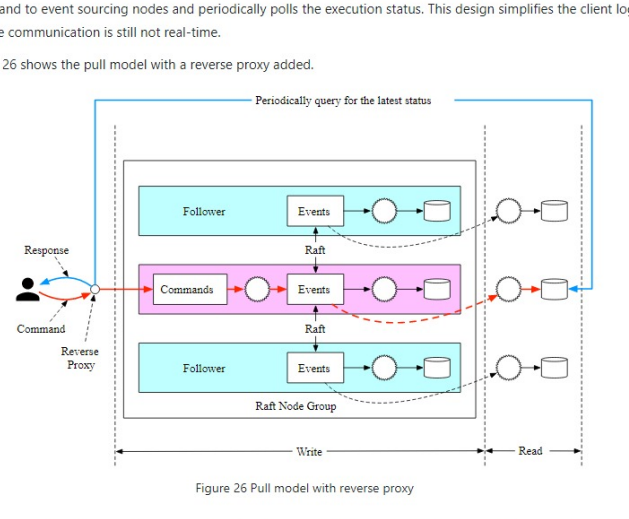


Figure 26 Pull model with reverse proxy

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 27 shows the push-based model.

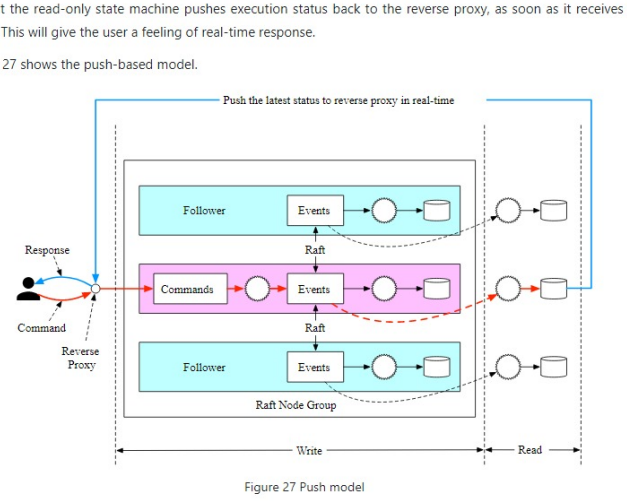


Figure 27 Push model

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 28 shows the updated design.

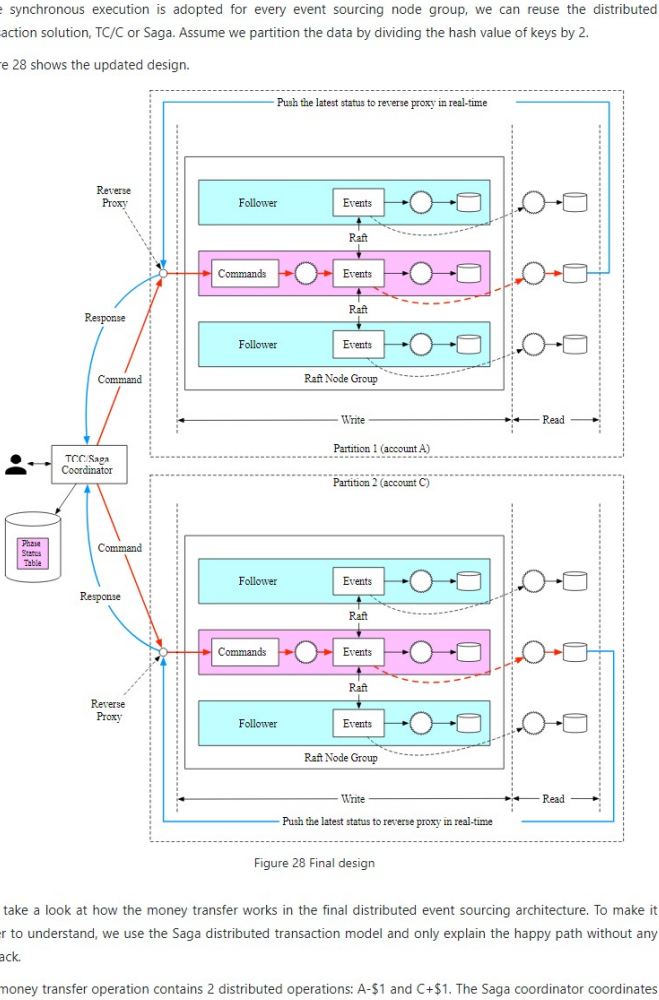


Figure 28 Final design

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 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.

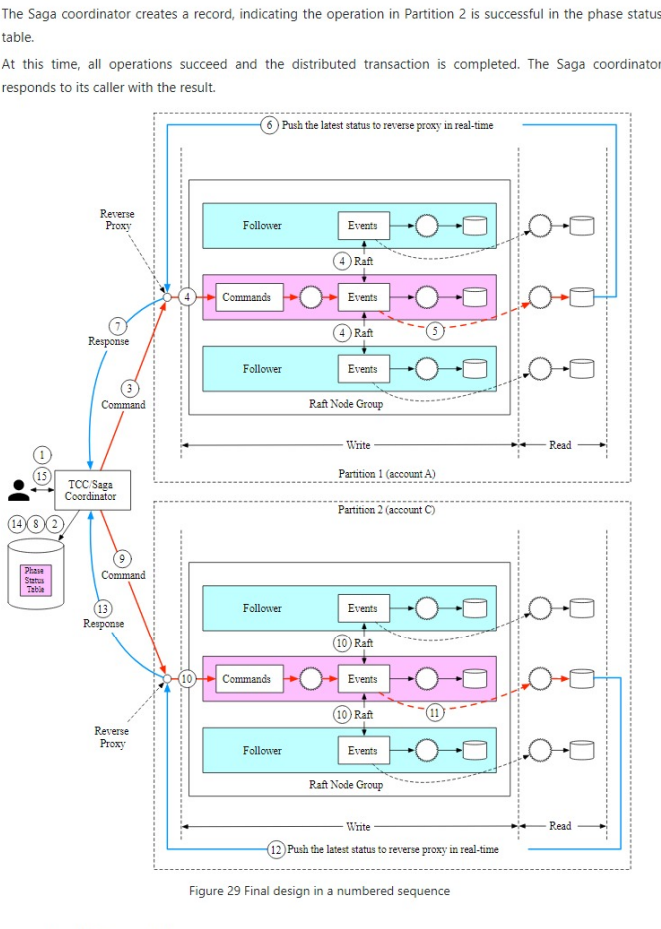


Figure 29 Final design in a numbered sequence

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.

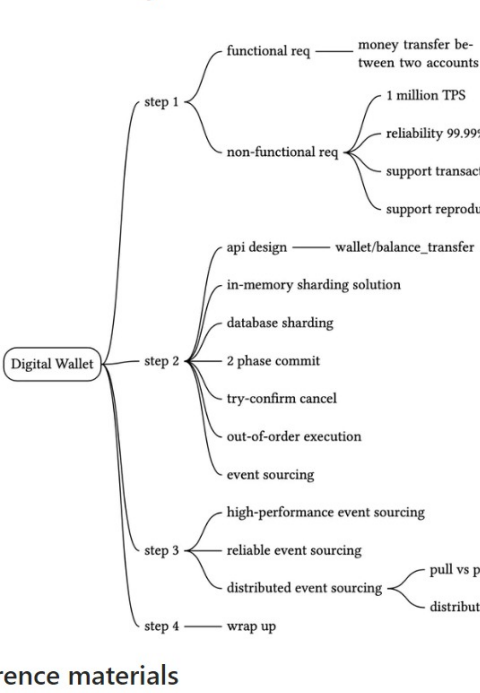
Next, event sourcing is introduced. We first implemented event sourcing using an external database and queue, but it's not performant. We improved performance by storing command, event, and state in a local node.

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 materials

[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 (2017). Designing Data-Intensive Applications. O'Reilly Media.

[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] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional.

[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