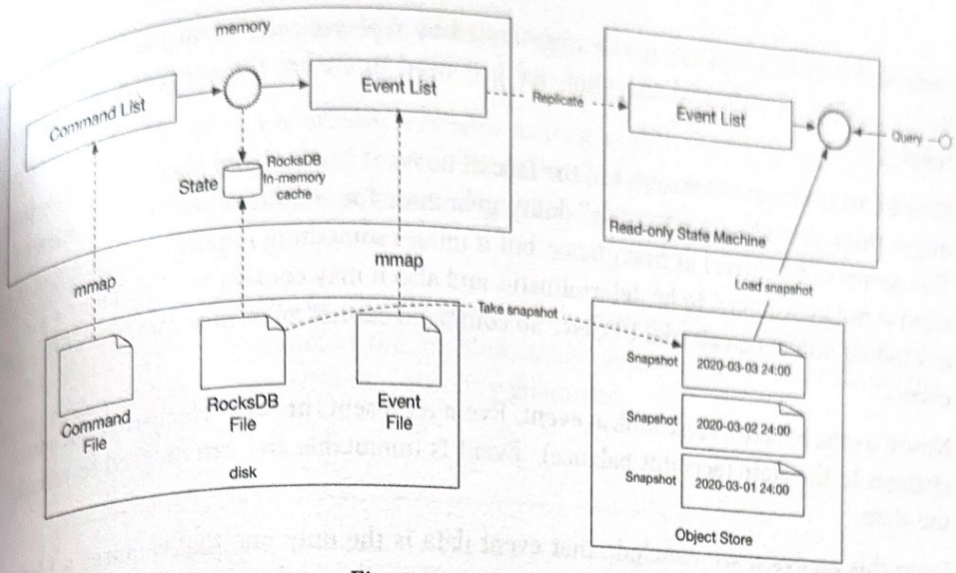


Figure 12.22: Snapshot

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:

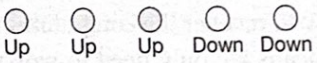


Figure 12.23: Raft

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.

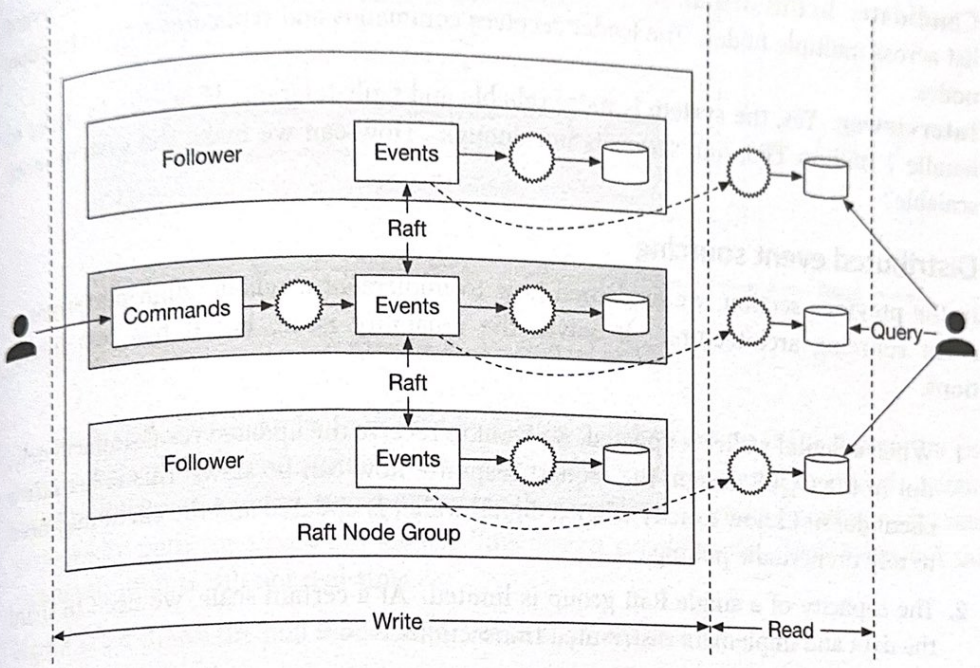


Figure 12.24: Raft node group

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.

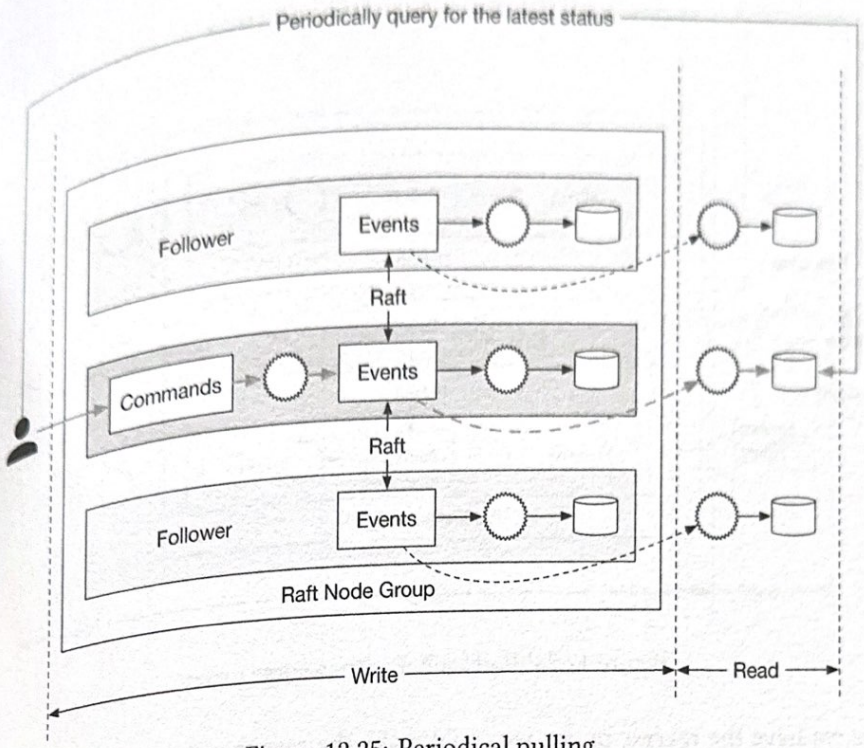


Figure 12.25: Periodical pulling

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.

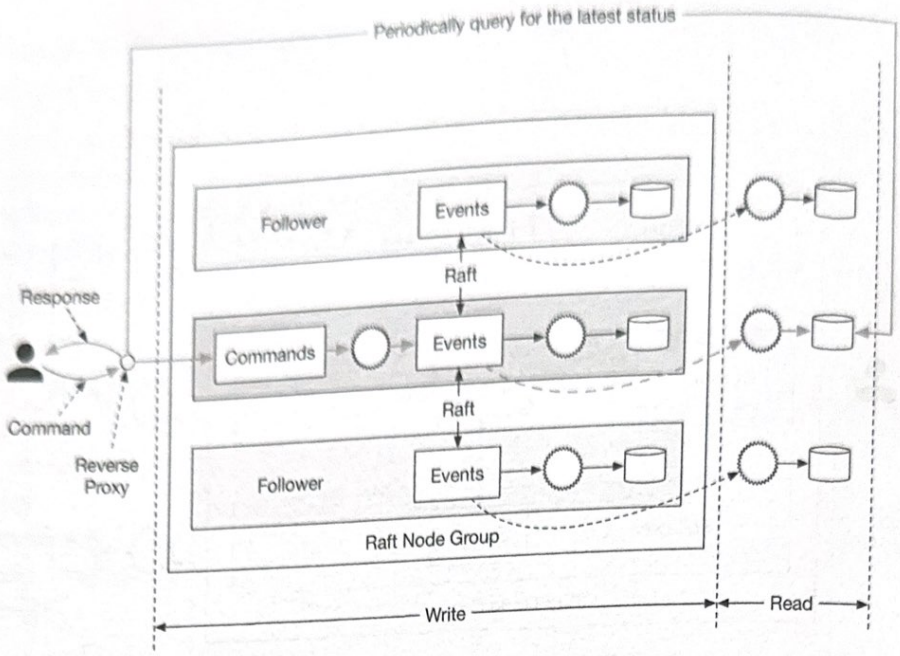


Figure 12.26: Pull model with reverse proxy

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.

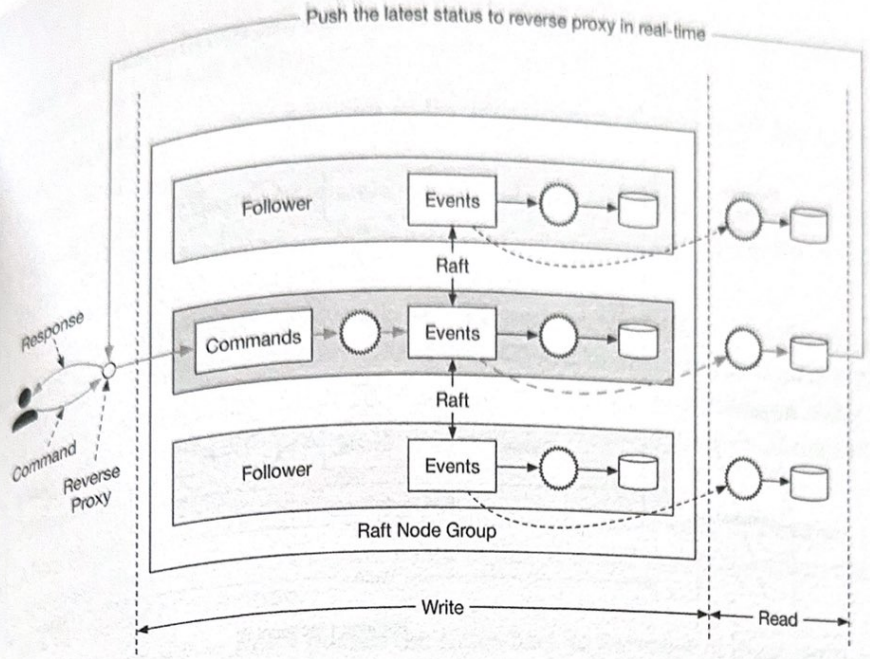


Figure 12.27: Push model

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.

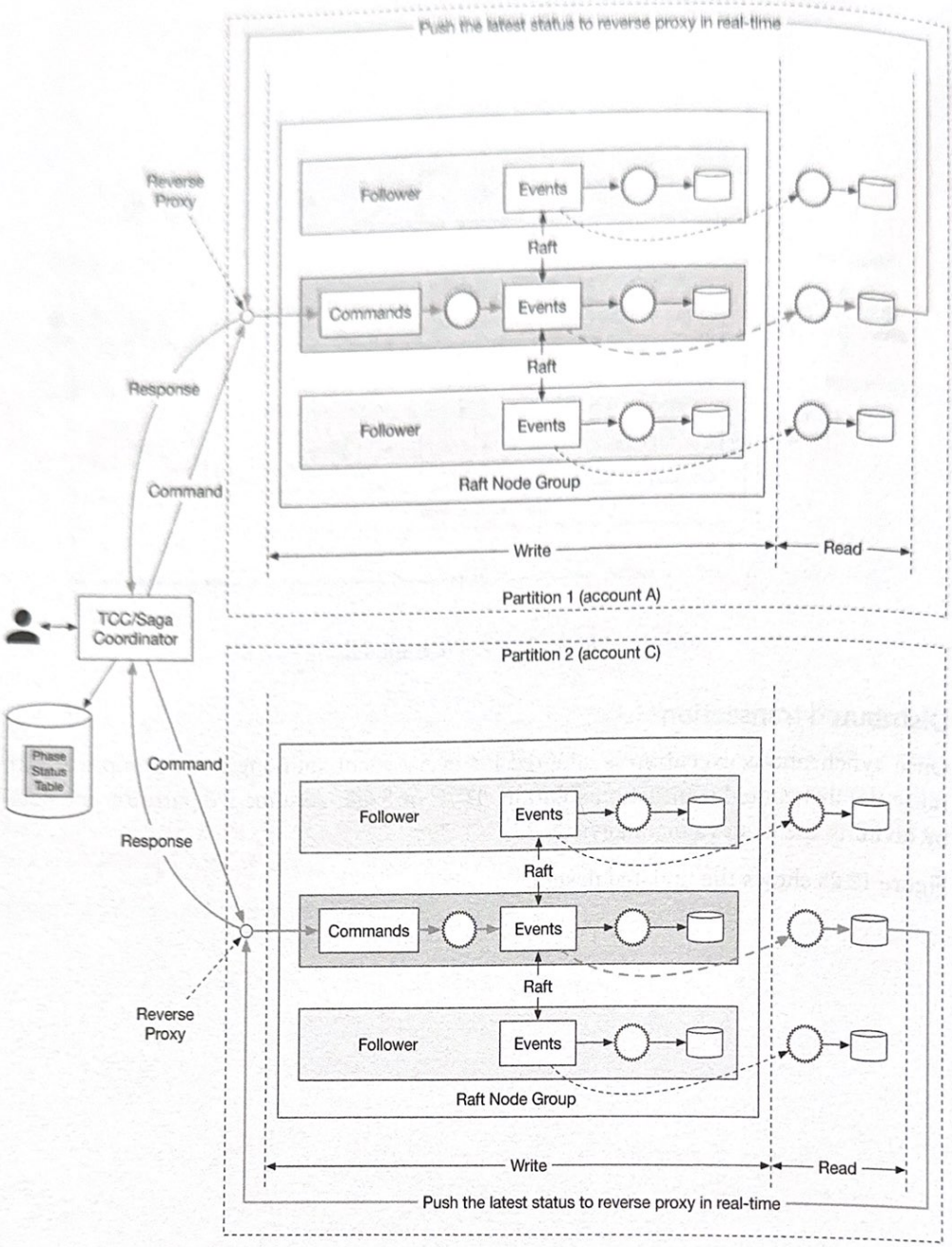


Figure 12.28: Final design

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.

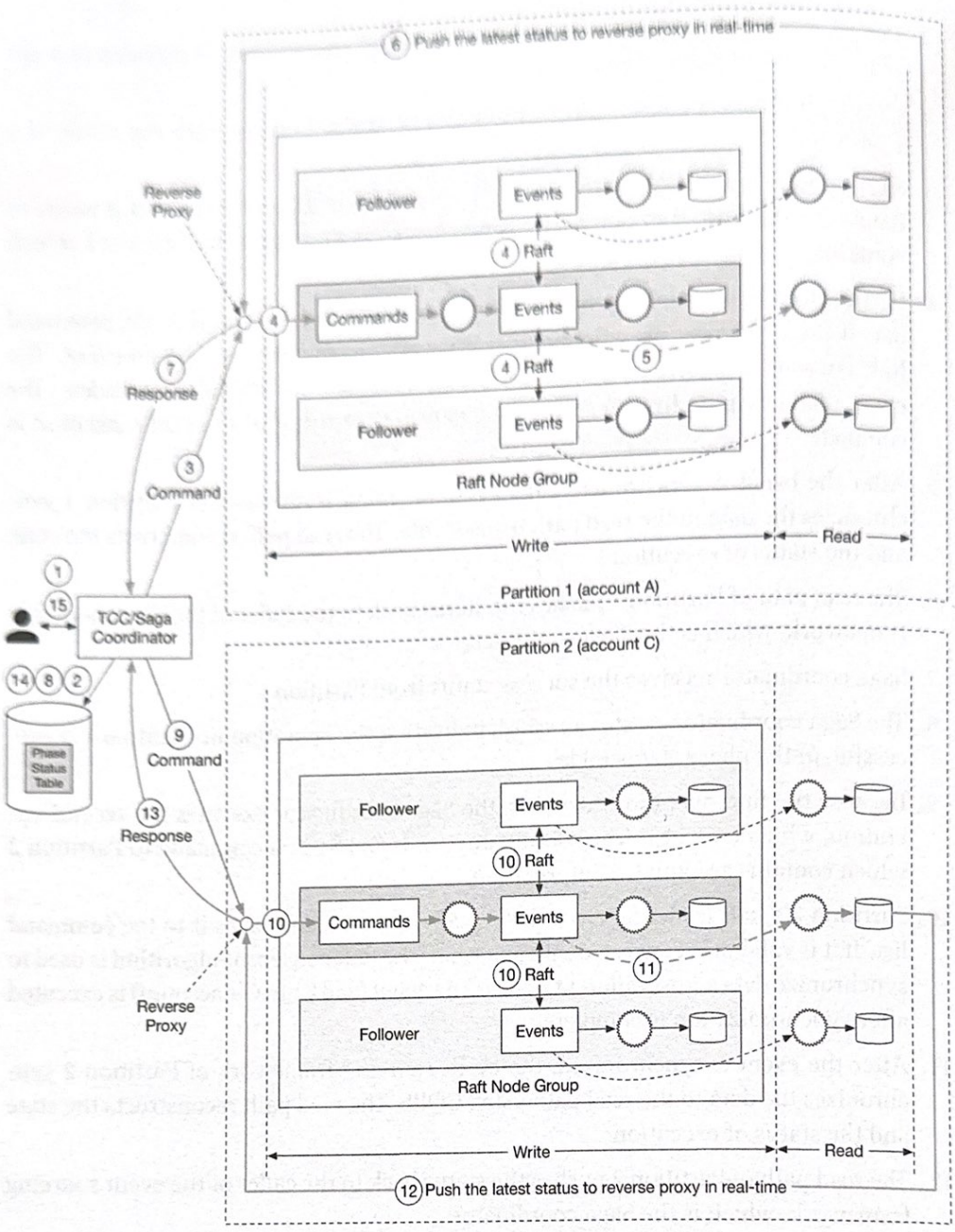


Figure 12.29: Final design in a numbered sequence

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.

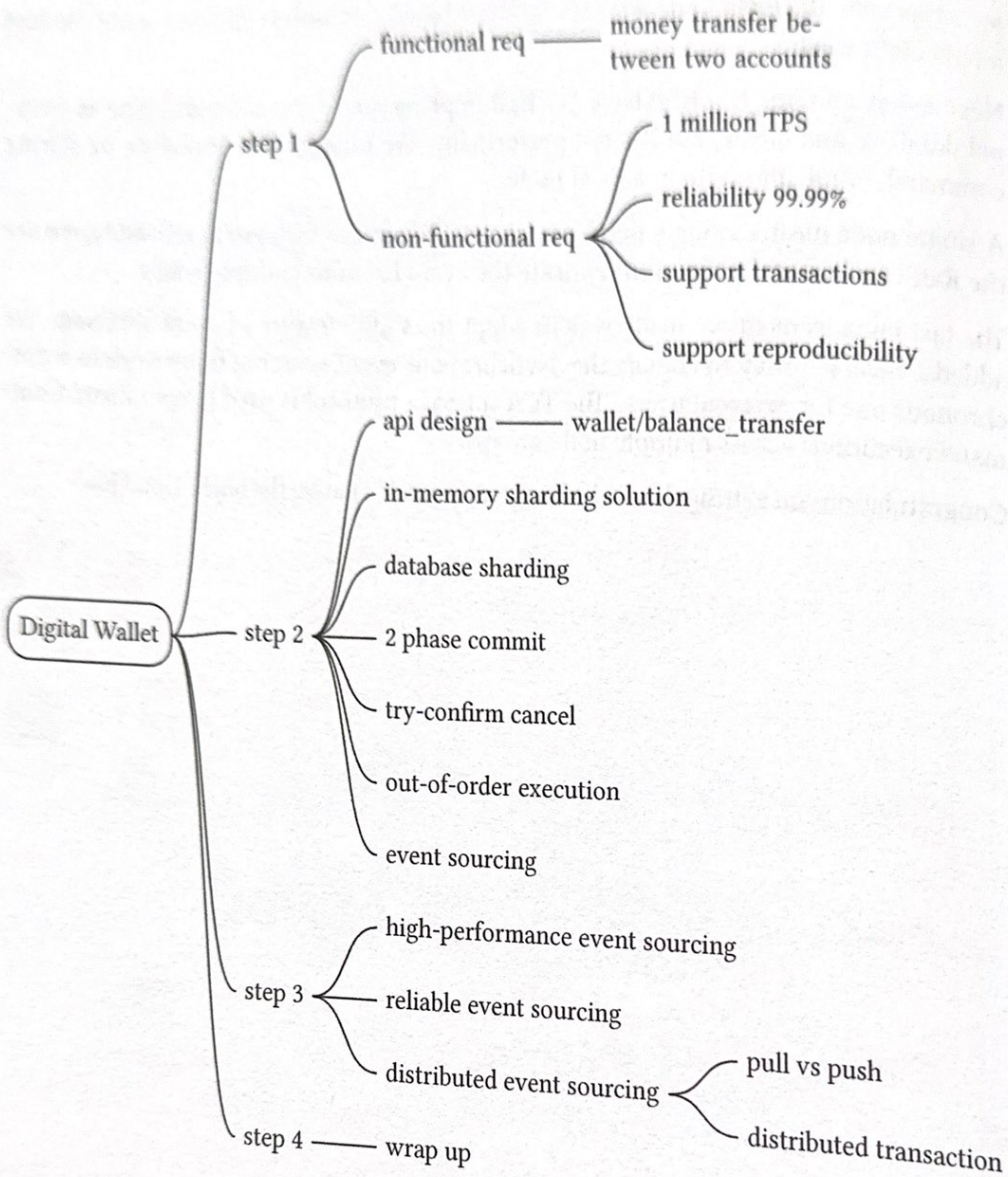
Next, event sourcing is introduced. We first implemented event sourcing using an external database and queue, but it's not performant. We improved performance by storing command, event, and state in a local node.

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



[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.