Question: Should we focus only on balance transfrom operation between 2 digital wallet?
Ans: Yes

Question : Transaction Per second
Ans : 1M TPS

Question: Do we need to prove correctness?
Ans: we could always reconstruct historical balance by replaying the data from the very beginning (KAFKA)

Question: Can we assume availability requirement as 99.99%
Ans: Sound good.

Question: Foreign exchanges are in scope?
Ans: No.

Summary:
1. Support balance transfer between 2 digital wallets
2. Support 1M TPS
3. Reliability : 99.99%
4. Support transactions.
5. Support reproducibility.

Back-of the envelop calculation
Assuming each node is capable to handling 1K transaction so in order to handle 1M transaction
We may need 1000 nodes. Quite high, we have another case here each transaction is composed of 2 operation:
1. Moving from 1 account
2. putting into another.
So in total we have 2M transactions, so do we need 2K nodes.
Another non Functional requirement is to increase transaction per node
say 10K so that 200 nodes can work or less.

Step 2: Propose High leve design and get Buy-in.
1. API Design.
2. Three High level designs.
1. Single in-memory solution
2. Database-based distributed transaction solution
3. Event sourcing solution with reproducibility.

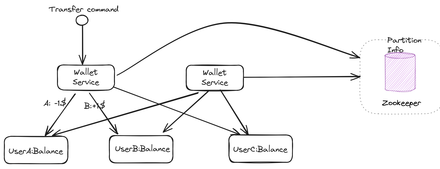
API.
Post /v1/wallet/balance_transfer

Transfer balance from wallet to another.

Request Parameter:
1. from_account
2. to_account
3. amount
4. currency
5. transaction_id(UUID)

Response:
Status
Transaction_id(uuid)

In Memory sharding solution:
A good data structure to maintain account balance for every user is to represent this relationship(user/balance) -> a map or key-value store.
Redis cluster with zookeeper is a good choice of data store in this case, zookeeper will hold number of partitions and addresses of redis nodes.
Final component is service that handles the transfer commands, Wallet service.
1. Receive the transfer command.
2. Validates the transfer command.
3. If command is valid, it updates the account balances for the 2 users involved in the transfer



In this design, account balance are spread accorss 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 balance accordingly.

Problem: What if request A success and request B Fails. The two updates need to be in a single atomic transaction.

The problem can be partially solved by replacing Redis node with DB nodes.

Distributed Transactions: Two-Phase commit.
1. The coordinator, which in our case is wallet service, performs read and write operations on multiple databases as normal.
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 recived.
2. If any database replies with no, the coordinator asks all the databses to abort the transaction.

Problems with 2phase commit.
1. It's not performant, as locks can be held for a very long time while waiting for a message from the other nodes.
2. The coordinator can be a single point of failure.

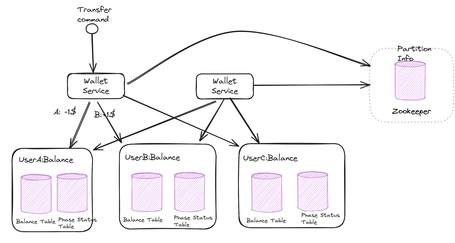
Distributed Transaction: Try-confirm/cancel(TC/C)
1. In first phase, coordinator asks all databases to reserve resources for the transaction.
2. In second phase, the coordinator collects replies from all databases:
If all replies yes the coordinator ask all database to confirm the operation which is Try-confirm process.
If any database replies with no, coordinator asks all databases to cancel the operations, which is try-cancel process.

	First Phase	Second Phase: success	Second Phase: fail
2PC	Local Transaction are not done yet	Commit all local tranactions	Cancel all local tranactions
TC/C	All local transactions are completed, either commited or cancelled	Execute new local tranactions if needed	Reverse the side effect of the already commited transactions or call 'undo'

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 business logic. The advantage of this approach is that it is database agnostic. As long as DB supports transactions, TC/C will work. The disadvantage is the we have to manage the details and handle the complexity of the distributed transactions in the business logic at the application layer.

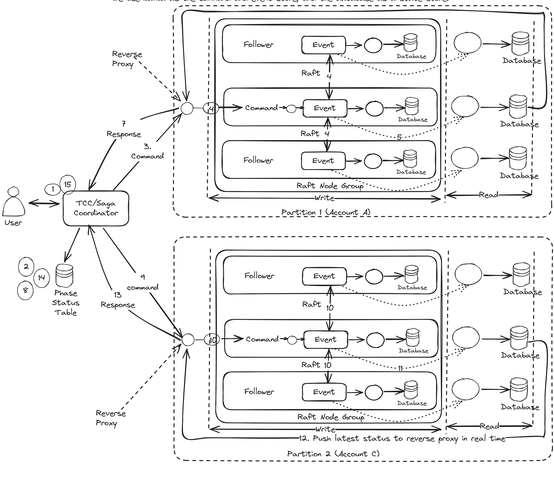
What if wallet service restarts in middle of TC/C?
Phase Status Table
We can store the progress of TC/C as phase status in a transactional database. The phase status includes at least the following:
1. The ID and content of distributed transactions.
2. The status of the try phase for each database.
3. The name of the second phase.
4. The status of the second phase.
5. An out of order flag.

we store the phase status in the database that contains the wallet account from which the money is deducted.



Event Sourcing
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.

High performance
We use kafka as the command and event store, and the database as a state store.



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 A-1 command and stores it in the command list. If then validate the command. If it is valid, it is coverted into an event. the RAFT consensus algorithm is use to synchronize data accross different nodes. The event is executed after sync is complete.
5. After the event is sync, the event sourcing framework of partition 1 syn the data to the read path using CQRS. The read path reconstructs the stat 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 sucess in the phase status table.
9. Because the first operation succeed, 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 recieve the C+1 command and saves it to the command list. If it is valid, it is converted into and event. The RAFT consensus algorithm is used to sync data accross different nodes. The event is executed after sync is complete.
11. After the event is sync, the event sourcing framework of Partition 2 syn the data to read path using CQRS. The read path reconstructs the state and the status of execution.
12. The read path of Partitions 2 pushes the status back to the caller of the event sourcing framework which is saga coordinator.
13. The Saga coordinator receives the success stauts from Partition 2.
14. The saga coordinator creates a record, indicating the operation in Partition 2 is sucess in 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.