Shard Transaction Manager

Nadav Rosenthal - 205429194, Eden Doron – 206336869

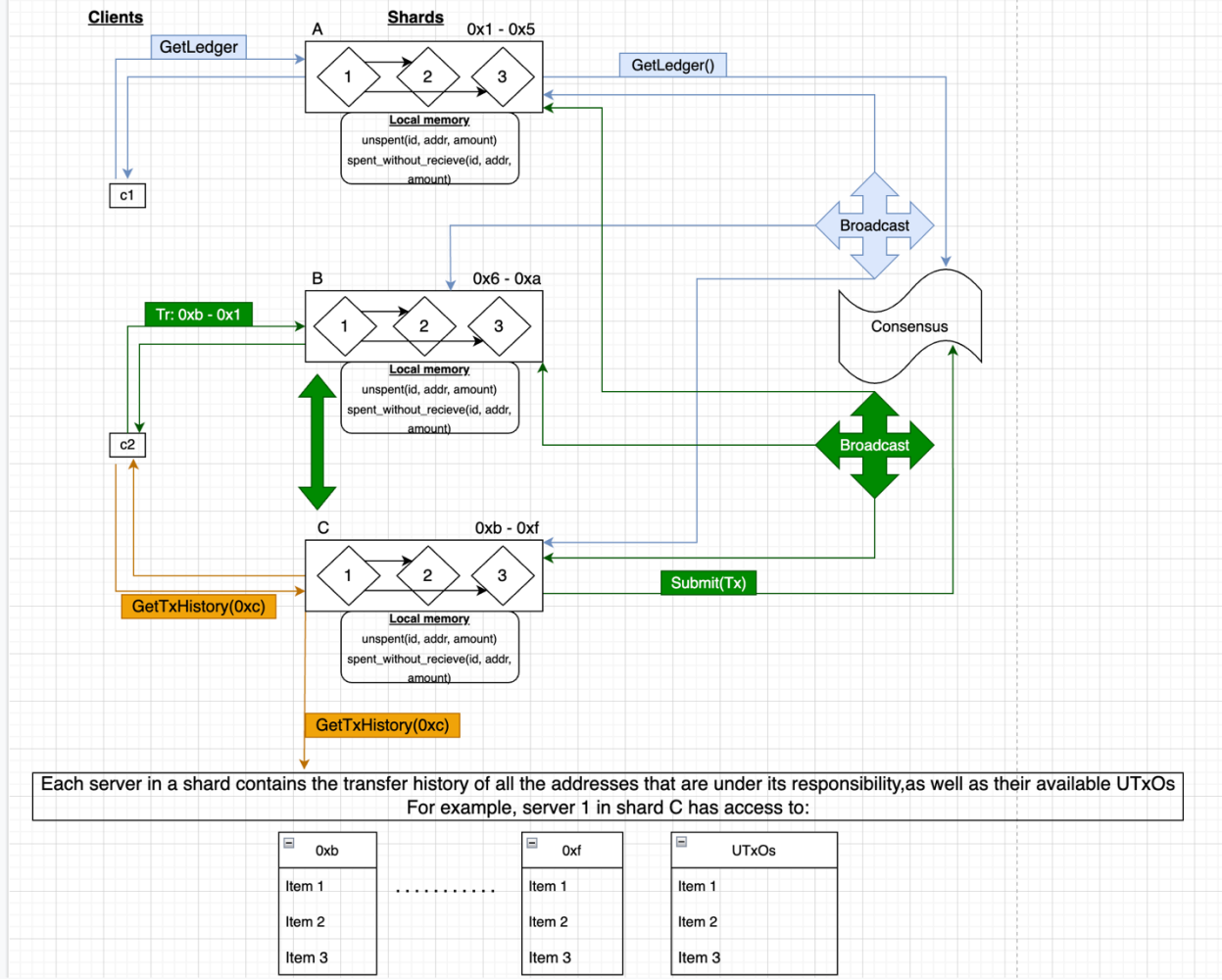
Introduction

For this project, we were tasked with designing, building and implementing a distributed shard transaction manager using tools, technologies and concepts such as kotlin, zookeeper, docker, gRPC, REST API's, replication, membership, consensus, atomic broadcast and total ordering. First, we will explain the design of our system.

Design

Per the instructions of the assignment, we had to handle transaction of a cryptocurrency between different clients of the system. Each client was allocated an address from a total of 2^{128} different addresses. Since we didn't want a single server to handle all the requests from the different addresses, we decided to balance the load and split the address space into 3 segments where each shard handled a $\frac{1}{3}$ of the addresses. This means shard i will be the designated handler of requests from addresses such that $mod_{numOfShards}(address) = i$. Practically speaking, a user may send a request to any of the shards in the system and it will either be handled by the contacted server in the shard or will be redirected to the corresponding shard based on the client's address. Both ways the user will "feel" like he is interacting with a single server.

We decided on a modulo based distribution on the assumption that no single address will submit more requests than any other (uniform distribution).



Each server will hold a local copy of the shard's transaction history (for each of it's designated addresses) and both an available UTxO pool and a missing UTxO pool. The first being all the induced UTxOs generated from the submitted transactions or transfers and the latter containing UTxO which were used but have not yet been received by the shard.

Additionally, in order to support failures, we have decided to implement a primary-backup replication in the form of leader/follower. Each shard contains 3 servers where one of them is the leader of the shard and is the one processing the requests and the rest are updated with the new state of the shard (e.g., update local ledger and utxo pool) by the consensus mechanism which is deployed by atomic broadcast.

Submitting a transaction

One of the main parts of the service is to provide a transaction submission API. Clients can submit a transaction request to any of the servers in the system via an HTTP request. The request is then allocated with a ID (UUID) of 128 bits that is unique with a very high probability (the chance that an ID will repeat itself is almost nonexistent). Additionally, the client has the option to provide a tx-id in the request body.
Next, the server will find the correct (the leader of the shard which is responsible for the client's address) gRPC server, call its submitTransaction method and return the reply to the client.
Once in the gRPC function, the server will check the inputs of the transaction and remove those from its local UTxO pool. If no UTxO was found, we assume that the server is missing them and that they will be received in a later time. In the meantime, the server will save them in a missing UTxO pool (they will be removed once the corresponding transaction will be processed).

In a makeTransfer request, the server will use the available UTxOs in its UTxO pool to construct a transaction (while checking to see if the sender should receive change) and submit it to the atomic broadcast mechanism.

The server will then generate the induced UTxOs from the transaction's outputs once it is received through the atomic broadcast.

*We assume the client will submit valid transaction (inputs sum = outputs sum) and that its inputs exist (or will be received by another shard).

Atomic Transaction List

In order the support an atomic transaction list submission to the system, we assume clients are honest and therefore support "zero transactions list" (all tx-id's are "0"), under the assumption that all input UTxOs are valid and unrelated - meaning can be submitted atomically or "Non-zero transaction list" (all tx-id's are valid uuid). Then we check that the given list is valid. i.e. check if the intersection between the input UTxOs of one transaction and the output of another.

To ensure the list is submitted atomically, we send the entire list as a single message to the atomic broadcast instance. (In order to support this operation, we always use a message type of a list of transactions, even if only one transaction is sent)

Ledger History

In order to get the entire ledger history, each server in the system saves the transactions history of all the addresses in its jurisdiction. When a server receives a request to get the ledger history, it sends a blocking request to the leader of every shard and gets back their respective transaction history. In order to sort the set of transactions, we used Kotlin's SortedSet data structure which sorts the items automatically. Once the server has all the data, it takes the number of transaction defined by the "limit" parameter and returns it.

Atomic Broadcast

We have implemented the omega failure detector in a way that the chosen leader is always the last child of the last shard. If this node fails, a watcher will be notified, and a callback will be invoked so zookeeper will give us a new leader.
As mentioned before, this leader in consistent with the one selected for the consensus proposer.
Additionally, when a proposer receives a message as a bytestring that should be proposed, he adds it to a list of messages to be batched. Every 1 second the proposer will send the batched messages across the system through the "deliver" method. Since we don't actually know the system requirements or average throughput, we thought a 1 second timer would be suitable but could easily be changed accordingly.
When a server wants to broadcast a message to the other servers, it will use the "send" method and every server listens to the atomic broadcast stream.
Once a message is received from the stream, the state of the server (local memory objects) will be updated accordingly.

Failure Handling

In order to ensure our system's normal behavior during server failures we implemented several features:

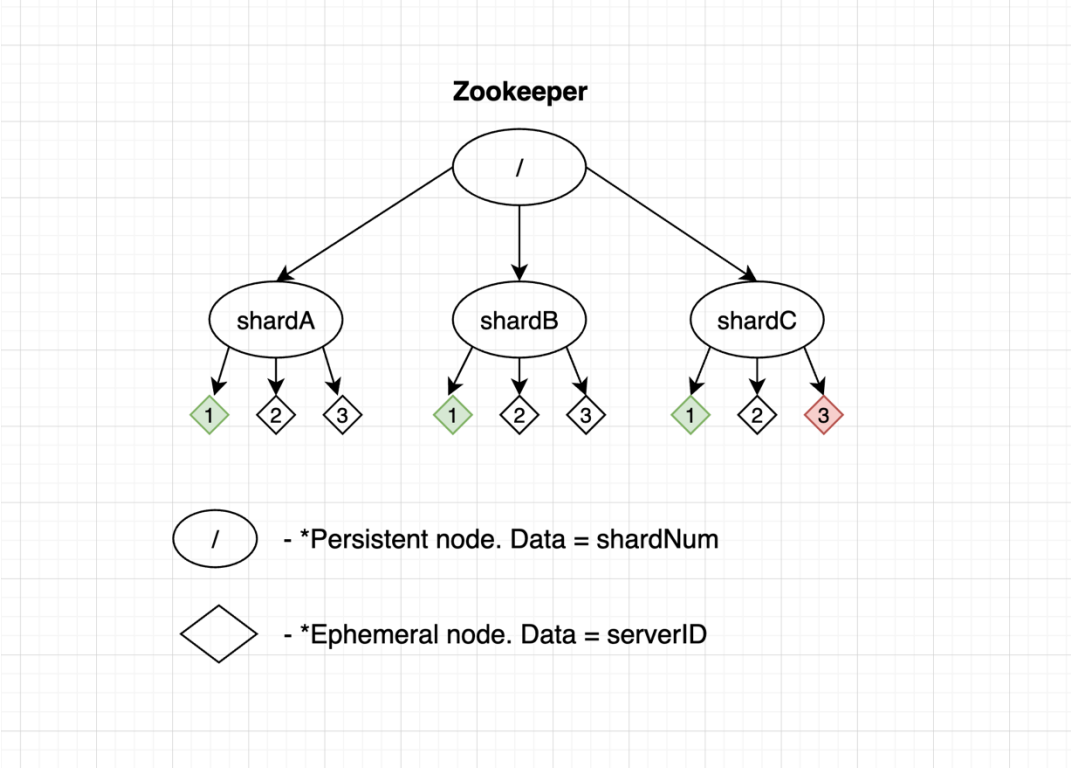
- Each shard consists of three identical servers (nodes) such that only one of them is actually processing the requests and the other 2 are updated frequently with the new state of the shard and ready to take its place upon failure. Since we are expected to support up to $\frac{n}{2}$ failures, 3 nodes in each shard are enough. This is based on the Hot Standby (Leader/Follower) method we saw in class.
- When an HTTP server receives a request, it will try to call the leader of corresponding shard. If the server doesn't return an answer during the specified deadline (4 seconds), the calling server will retry to find the leader (which will be a different server in the shard now) and attempt to invoke the method again. This will happen 3 times (again, due to possible $\frac{n}{2}$ failures). Two retries would also work but we decided to err on the safe side.
- Once a server receives a message from the atomic broadcast stream, it will check if it already processed the transaction to prevent double spending that could happen from server failures.

Zookeeper

In order to achieve synchronization between our servers and between the shards, we used zookeeper. This allowed us to register each server to a predetermined (at startup) shard using membership. Using membership, we can sync the relevant state of the followers in each shard as well as determine the leader of the shard by selecting the first child of said shard according to zookeeper (marked as green in the figure below). That means that if the leader falls, it will be replaced by the next child of the shard. The implementation of this can be found in the $findOwner$ method in the $TransactionManagerRPCservice$ at lines 56-61

This is possible by creating a persistent node for each shard (which serves as a membership indication) and an ephemeral node for each server in the shard. For example, server #3 in shard #2 can be found at /SHARD_2/manager3.zk.local

Additionally, we used zookeeper to select the proposer of the system (the server that transfers messages using atomic broadcast) as the last child of the last shard (marked as red). This server serves as both the leader output of the omega failure detector and as the chosen proposer of the atomic broadcast mechanism. Implementation of both can be found at InitPath.kt.



Since we are expected to handle failures of up to half of the nodes in each shard, we decided that 3 replications would suffice so that we will always have at least 2 nodes running in every shard.

The zookeeper nodes creation can be found at lines 54-60 in the initPath.kt file

Docker

In addition to the provided zookeeper instance which runs in a docker container we have decided to do the same for each of our transaction manager servers.
We built a docker image that is used in all of the containers (since they all use the same code) where each container runs a single instance of a service.
In order to differentiate between the instances, in the docker-compose file we provide each container with an HTTP port (on the local machine) for clients to

connect to and environment variables with the needed information for the server to run (such as the server's ID and what shard does the server belong to).
Each container then runs an independent instance of both an HTTP server for the REST API and a gRPC server (the servers communicate with each other with gRPC calls that are sent between the docker containers).

Conclusion

We have implemented all of project's requirements including all bonuses.
In a personal note, this project was very challenging and overwhelming at first. After a lot of hard work we managed to overcome many of the difficulties and build a project that we are proud of. We believe that many interesting and useful technologies and techniques are presented in this assignment and pleased that we have learned and experience it as we did. We also believe that we have managed to face a lot of the problems and different scenarios in the distributed computing world as required in this project.