**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 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 of the addresses. This means shard will be the designated handler of requests from addresses in the range .

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.

Diagram

Description automatically generated

Additionally, in order to support server crashes 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) through the use of a consensus and then sent with atomic broadcast.

Zookeeper

In order to achieve synchronization between our servers between the shards and within them, we used zookeeper. This allowed us to register each server to a predetermined (at startup) shard using membership. Using membership, we can sync the state of the followers in each shard as well as determine the leader of the shard by selecting the server with the lowest node number according to zookeeper (marked as green in the figure below). That means that if the leader falls, it will be replaced by a server with the next lowest node number in the shard. The implementation of this can be found in the method in the 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). Used at InitPath.kt, lines 70-80.

Diagram

Description automatically generated

Since we are expected to handle crashes 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 server.

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).