Sharded Transaction Manager

Distributed Systems - 236351

Yarin Bar 204771505
Noam Raveh 313599573

Januaray 27, 2022

Introduction

The following report describes our implementation for a distributed transaction manager which manages a distributed ledger that records all committed transactions. The transaction manager provides the functionalities of submitting pre-built transactions, sending coins between users, getting transaction history and more. In this report we will cover:

1. The **architecture** of the system. This section is split into 2 parts:
   (a) **General Architecture** - here we discuss our considerations and choices when building our system as well as the structure itself.
   (b) **Zookeeper Architecture** - an in-depth explanation of our membership, creation and interactions with the zookeeper.
   (c) **Docker** - we will overview the structure of the network we defined in the docker-compose.
2. The **functionalites** we implemented in the system and how they each work.
3. The **robustness** features we implemented to minimize possible failures and ensure our system will be operational consistently.

Architecture

General Architecture

We chose to implement a Leader-Follower system for its simplicity and robustness. Communication between the end users and the system is handled by a RESTful API we wrote to make our system user-friendly. Internal communications between servers are conducted solely in gRPC.

If a node that is not the leader gets a request that **changes** the state of the system, the request is redirected to the leader. Examples of requests that do not change the state of the system are queries about addresses and history; These can be handled by the followers as all the nodes are in the exact same state as the leader.

Node Attributes

Each node has the following attributes:

1. **Ledger** that holds all the transactions of the addresses that is under this shards' management. We chose to go with *HashMap* for its quick fetching and storing. The key for the HashMap is the address and the value is a *MutableList* of transactions.
2. **UTxOs** another *HashMap* with addresses as keys and a *MutableList* as values. This data structure holds all the utxos available for spending by each address. Each time a transaction is made the input utxos are deleted and the output induced utxos are added.

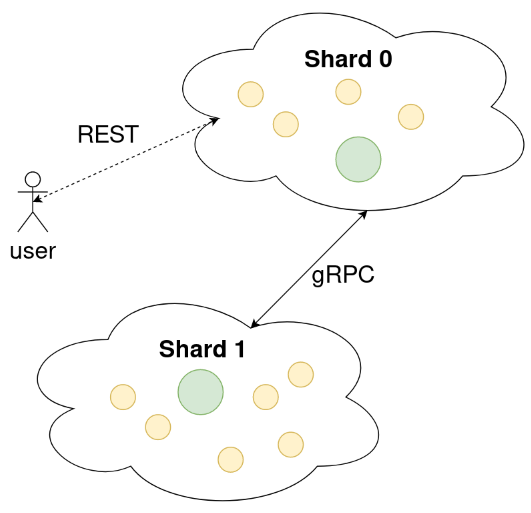

Figure 1: Communication scheme

3. **Zookeeper**
    (a) **Zookeeper Connection** - a channel that is open between each node and the zookeeper.
    (b) **Zookeeper Client** - a client that supplies all the necessary action when interacting with the zookeeper
4. **Shard Info**
    (a) **Num Shards** an integer that indicates the number of shards in the system. It is an environment variable and is imported to the class locally.
    (b) **My Shard** an integer that indicates the affiliation of the node and is used to verify if the node that received the request needs to process it as well. It is an environment variable and is imported to the class locally.
5. **Networking Info**
    (a) **My IP** a string that holds the public IP of the node for creating an **ephemeral** node in the zookeeper.
    (b) **Port** an integer that holds the port on which each node is listening. It is an environment variable and is imported to the class locally.
6. **Atomicing** - a Boolean flag that indicates whether or not the node is currently handling an atomic transaction list. When getting a write request if this flag is on then the request is denied until the flag is off.

Address Handling

Each address is assigned to a shard by computing **address** *mod* **NUM_SHARDS**. Notice that in this way, we gain a few important things:

- We guarantee an even spread of the address space between the different shards.
- Modulo is an accurate, fast and easy computation.
- Each node can determine who is the correct shard without having to maintain a mapping.

Zookeeper

The following are the 3 main components of zookeeper in our project:

Structure

The file structure of the zookeeper adheres to the following simple and intuitive pattern:

```
zookeeper
├─ SHARD_0
│  ├─ 172.100.0.5_00001
│  ├─ 172.100.0.9_00002
│  ├─ 172.100.0.1_00003
│  └─ ...
├─ SHARD_1
│  ├─ 172.100.1.4_00001
│  ├─ 172.100.1.1_00002
│  └─ ...
├─ ...
└─ SHARD_N
   ├─ 172.100.N.15_00001
   └─ ...
```

In each shard the server with the lowest sequential number is chosen to be the leader(which is not necessarily correlated to the IP address as shown in the figure above.

Membership

When a server wakes up it already possesses an IP address and a **shard number** that the node is a part of. If it is the first to wake up in the shard it creates a node named SHARD_${MYSHARD} in the zookeeper and then proceeds to register itself in the newly opened shard. It is created as both a *ephemeral* and *sequential* with the naming convention we discussed earlier. The ephemeral flag ensures that the node is only alive as long as the service that created it is alive, and the sequential flag is responsible for providing an increasing sequence number, which we use for the leader selection.

Since in each operation we elect a new leader, being informed on a node dying is not necessary. We will later see that if the leader had started the operation we are guaranteed to have all followers in complete sync with it.

Interactions

As previously discussed, each node in the system can independently identify the correct shard for each address. After identifying the correct shard, our naming scheme ensures fast retrieval of the correct shard leader's IP.

Docker

First, we will start by presenting a shortened version of the `docker-compose.yml`:

```yaml
version: '3'

services:
  zookeeper:
    image: docker.io/bitnami/zookeeper:3.7
    hostname: zoo1.zk.local
    ports:
      - 2181:2181
    volumes:
      - zookeeper_data:/bitnami
    networks:
      - zookeeper_net
    environment:
      - ZOO_SERVER_ID=1
      - ALLOW_ANONYMOUS_LOGIN=yes
      - ZOO_SERVERS=zoo1.zk.local:2888:3888

  node{SHARD}{ID}:
    environment:
      - SHARD=0
    build: .
    ports:
      - 10000:8080
    networks:
      - zookeeper_net

networks:
  zookeeper_net:

volumes:
  zookeeper_data:
    driver: local
```

As we can see we define 2 kinds of services:

1. `zookeeper` - is the service of the zookeeper (which was discussed earlier) that holds all the data regarding all the servers in the system.
2. `node` - that represent each server in our system. Please refer to the code and see that there are a few settings that come into play here:
   (a) `environment` - holds all the environment variable relevant to the node. In our case this is where we pass the `SHARD` number the server is a part of. That way, as soon as the server wakes up it knows what shard it is part of.
   (b) `build` - that directs to the `Dockerfile` that is present in the same directory.
   (c) `ports` - holds the port each of the servers listens to, which helps up contact each server directly.
   (d) `networks` - here we supply the one and only network that we use in this exercise. This network enables inter-service connection; a server to server or a server to zookeeper.

   In practice, there are many more `node` services to simulate a few shards with a handful of servers in each shard.

Functionalities

1 Submit Transaction

The **submit transaction** operation is the main building block used to add new transactions to the system. Other operations (sending money and submitting an atomic transaction list) rely on this operation, as explained later on. When a user sends a request to submit a transaction, the request is first translated into a gRPC message and given a unique id, and then handed over via gRPC to the leader of the shard responsible for the sender's address. Once the node receives the transaction it goes through the following steps:

1. **Ensure there isn't an atomic transaction taking place**
   If so, the new transaction may be interfering with the current atomic transaction taking place. For this reason, we decided to lock the possibility of conducting another transaction until the atomic list finishes.
2. **Validate the transaction**
   A valid transaction is one where all inputs are associated with a single address s and are all valid UTxOs for that address, and all off its inputs are spent. To validate the transaction, the server checks that the sum of the inputs equals the sum of the outputs. If all checks passed it returns successfully, otherwise an error message is sent.
3. **Add the transaction to the ledger**
   The transaction is then added to the ledger HashMap under the relevant key of the sender's address. If this address does not yet exist in the ledger HashMap, it will be created first.
4. **Remove used UTxOs (inputs) and add UTxOs within the same shard**
   Once the transaction is successfully validated and recorded, any of the used UTxOs must be removed immediately to avoid double spending on the next transactions. The UTxOs are located in the UTxO HashMap under the key for the sender's address and re removed from there. In addition to this, any outputs will be translated into induced UTxOs, and those who belong to addresses which are handled under the same shard are added at this point. Each induced UTxO will get a UTxO ID which is generated based on the TxID and the index of the output in the outputs list.
5. **Gossip the transaction to other servers in the shard**
   After handling the transaction on the leader's node, the transaction is sent out to all other servers within the shard. Upon receiving the transaction, each node checks if it already has this transaction in it's ledger (using the unique transaction ID), and if not it goes throw steps 3-6 of handling and broadcasting the transaction. This will be further discussed in the final section of the report.
6. **Send induced UTxOs to their relevant shards**
   Once one more than one node is aware of the transaction and is handling it, induced UTxOs from the outputs list are sent to their relevant shard. Upon receiving the UTxO, the node checks if it has this UTxO (using the unique UTxO ID) and adds it to it's UTxO list if not. It then begins gossiping about it to the other nodes in the shard. Sending the induced UTxOs outside of the handling shard is done after the

gossiping to avoid a situation where the handling shard's leader crashes after UTxOs were added elsewhere but before the transactions was recorded

Consistency
As the requests are initially handled in each shard by a single server (the leader), the order of the operations is determined by this server and are therefore arranged within themselves. The operations within different shards are also linerizable due to the UTxO model and the assignment's assumption that clients are honest and provide us with UTxOs which were indeed committed. Even if the server doesn't find the needed UTxO, it still uses it as if it had it, and it will be received later on by the shard responsible for the transaction from which it was induced. Since each transaction is handled by a single shard in a linerizable order, a UTxO cannot be used twice.

2 Send Coins

The **send coins** operation is used to send money between two addresses using the available UTxOs. When a user sends a request to send money,specifying the amount of money and the destination, the request is handed over via gRPC to the leader of the shard responsible for the sender's address. The server iterates through the list of available UTxOs for the sender, from the sender's key in the UTxOs hashmap, and adds the UTxOs to a list of UTxOs which will be used for the transaction. The iteration is terminated when either the sum of UTxOs iterated through reaches the amount the user wants to send, or when there are no more UTxOs available. If after going through all the UTxOs the total sum doesn't reach that of the reequest amount, an error is returned and the transfer doesn't happen. Otherwise, the server creates a transaction using the selected UTxOs as inputs, and the desired transfer amount as an output. If need an additional output is added with the sender being the destination - this is the 'change' from the UTxOS that are used. From this point, the newly created transaction is handled using the submit transaction operation.

Consistency
The send money operation is linerizable for the same reason as the submit transaction operation that it uses, which is explained above. As mentioned in the assignment, the server will try to select UTxOs from what it has in it's local memory only, meaning that it is possible that it will fail even though it has UTxOs that it hasn't yet received from other shards.

3 Get Address History/UTxOs

The functions for listing all unspent transaction outputs and entire transaction history work by accessing either the UTxO hashmap or the ledger (respectively) under the key of the given address, and returning the relevant list. To maintain consistency we've decided to also handle these request in the relevant shard leader for the given address. Therefore, any incoming request will be first send to that relevant node. Both operations have the option to add a limit on the number of items returned, as requested.

Consistency
These operations are sequentially consistent as they reflect the program order in which the

requests were sent. Since the requests are handled by a single server in each shard, which also handles all of the write (send money/ submit Tx) requests, these read operations will return the most recent writes. Any additional requests for the history/UTxOs will return the same values as long as no other writes have been performed.

4 Submit Atomic Transaction List

The atomic transaction list also relies on the the submit transaction function, however, additional measures are first taken to ensure the legality of the list and to ensure that no other operation will ruin the legality mid-run. When an atomic Tx list is submitted, it is first sent to the shard leader of one of the senders (the first address in the list). Then, the list is validated by ensuring that no single UTxO is used twice and that each individual transaction is valid. Then, a lock is set for each server that is participating in any of these transactions. The reason for this look is so that no other transaction/coin sending request will come in and use a UTxO that is part of the atomic Tx list, causing it to fail. Any other request for transactions that comes in at this time will wait until the entire atomic transaction list completed processing. Once all the transactions are completed successfully, the locks are released and a success message with the generated tx IDs is returned.

Consistency
Similar to the submit transaction function, the atomic list will be linerizable due to the UTxO model and the fact that a transaction can pass even if the listed UTxO is not found in the list, since it will at some point arrive. Since we validate the request before sending it, and ensure that no other request for a transaction can come in at that time, the request should go through successfully.

5 Get Entire Ledger History

To get the entire ledger history, we reach out to each shard's leader and obtain from them their ledger for all the addresses they handle. We then combine all the results into a single list and order them by their timestamp (generated upon submission to the ledger). This function also provides the option to limit the number of transactions in the output list.

Consistency
The entire ledger history is a linerizable operation, as it reflects real time order between all of the clients - any operation that has taken place *before* (i.e started and concluded) initiating the ledger history request will be included in the ledger history, while any operation that was requested *after* initiating this request will not be part of the results, as their write operations are more recent than this read operation

Robustness

ID Generation

- Tx ID We chose to generate IDs for the transactions using UUIDs. UUIDs are 128-bit hexadecimal numbers that are globally unique. The chances of the same UUID getting generated twice is negligible, which is why we believe this is a good option for our system which requires minimal computational effort. However, if our database size increases, these may cause a problem as they are big in size and don't index well. Nevertheless for the purpose of this assignment and the scale of our system we believe that this is a good and simple choice of ID generation.
- UTxO ID We also decided to provide a unique ID for each UTxO in the system. This enables us to use the gossip mechanism without the fear of a UTxO being added to the system more than once, and thus to avoid double spending of funds. Each UTxO is generated using the Tx ID of its corresponding transaction, and the location of the UTxO in the transaction's input list, meaning that even if multiple servers generate the induced UTxOs from a given transaction, they will have the same IDs and we can ensure that they will only be added to the system once.

Single Server Total Ordering and Fault Tolerance

We used a simple gossip algorithm to ensure fault tolerance in our system. This works well with the way ordering is done in the system (through single server total ordering in each shard), as explained below and in each function in the previous section.

Each transaction is verified and processed **localy** in the leaders' machine.Once the leader completes processing the transaction, it broadcasts the transaction to all of the followers. Once received, each transaction is processed immediately in the followers' machine.

This mechanism ensures fault tolerance. To discuss this, we need to separate into 2 cases:

1. Leader dies before transmitting the transaction - in such case non of the followers have been updated and they are all in agreement of the current state (even if the transaction did not go through).
2. Leader dies after transmitting - in this case it depends whether or not any of the followers that received the transaction are still alive:
   (a) If all the followers that received the message are dead - see 1
   (b) If there exists a follower that got the transaction and is alive, the transaction is processed locally and promptly gossiped to all other alive nodes. This ensures all alive nodes in the shard are in the same state.

An important note is that **each follower sends the induced UTxOs only after gossiping the new transaction to all other nodes in the shard** which prevents a situation where the induced UTxOs had been sent but the transaction was not updated in all the original shards' nodes.