

HDS Ledger

Stage 2

Highly Dependable Systems

Alameda

Group 27

Catarina Beirolas
93034

João Luís
95607

14th April 2023

1 Design

The goal of the project’s second stage was to create a simplified Smart Contract application that establishes a Token Exchange System (TES). This application utilizes the functionality already developed in the first stage of the project to build a State Machine Replication system that can effectively support the TES, which holds a set of accounts (that are uniquely identified by a public key from an asymmetric key pair and hold the current balance).

Because it would be expensive to perform consensus for every single transaction individually, to optimize performance, we opted to define a fixed block size that accommodates a specific number of transactions, 4 in our implementation. Only transactions that modify the state of the blockchain are by default executed in block, operations that only query the system run a different protocol we later describe in more detail.

The system membership is static for the entire system lifetime, including a predefined leader and the membership information is known by all participating processes before the start. We assume every node in the system might be Byzantine.

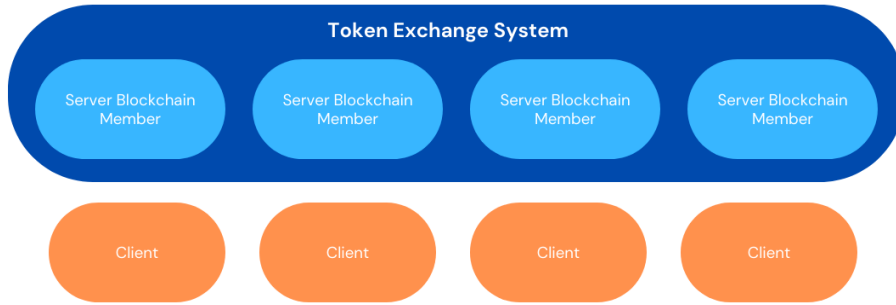


Figure 1: Design Overview

1.1 Client

The Client has a CLI through which it can submit requests to the system. The available operations are explained in detail in Section 3, Available Operations. The Client always has to first create an account before doing any other operation. For every request, a reply is received confirming if and when the request was executed.

1.2 Server

The Servers are responsible for maintaining the set of TES accounts and ensure a set of dependability properties and security guarantees, namely:

- No account has a non-negative balance;

- The state of the accounts can't be modified by unauthorized users;
- All operations issued on an account have a non-repudiation guarantee.

1.2.1 Channels

As required, the basic communication is done using **UDP** which allows the network to approximate the behavior of Fair Loss Links. To simulate Perfect Channels, we implement the following:

- To avoid message loss, we set a 5 second timeout and re-send the message until a reply is received;
- To avoid message duplication, we added a messageId to every message;
- To guarantee Integrity and Authentication, our messages are signed with the sender's private key and the receiver verifies it with the sender's public key;
- To guarantee freshness in the communication, the message's sender adds a nounce and then verifies if the nounce in the reply is the expected one.

1.2.2 Broadcast

We chose to use Best Effort Broadcast in our project. To simulate Perfect Channels and guarantees such as freshness, integrity, non-loss and non-duplication we implemented mechanisms which we explain later in the report.

1.2.3 Command Processing

We use threads to allow the Servers to still receive requests from the Clients while a Consensus instance is happening. When the consensus protocol has started, if a Server receives a blockchain modifying message from the Client, it is added to a queue. When Servers are deciding the requests in the block, they remove the first string in the queue, so that all requests from the Client are only processed once and in order.

1.3 Keys

We assume there is a Public Key Infrastructure in place. Blockchain members and blockchain clients use public/private keys that we previously generated, which are pre-distributed before the start of the system, and this information is also included in the static system membership.

2 Threats and Protection Mechanisms

In order to guarantee the required security and design specifications, we designed a set of demo tests (puppet masters) that create and run 4 servers and 4 clients.

2.1 Byzantine Server sends multiple ACKs/NACKs to the client for a given request

A Client waits for a quorum of ACKs/NACKs from the Servers to know if the request was completed or not. A Byzantine Server could try to send multiple ACKs/NACKs to deceive the Client into thinking a quorum was reached. To avoid this threat, the Client checks, for a given requestId if it has already received an ACK/NACK for a given port. If it has, they ignore the message and don't take into account for the quorum.

2.2 Byzantine Server sends repeated "Prepare" / "Commit" messages during the consensus instance trying to forge a quorum

The approach taken is the same as described before: the Servers also check if for a given consensus' instance, for a given phase, they have already received a message from a given port. If they have, the request is ignored and not taken into account for the consensus quorum.

2.3 Drop Message Attack

All messages exchanged between Clients and Servers (operations or consensus messages) have a timeout: if an answer has not yet been received, it is sent again in an exponential back-off timeout. The ACK/NACK messages are also re-transmitted to reduce this threat.

2.4 Byzantine Server sends in the block a request signed by them instead of the Client

Since all replicas verify the signatures in all the requests sent in the block, they are able to detect if a Byzantine Server introduced a request there trying to pretend it was a Client request. If a Byzantine Leader modifies a Client's request and signs it with its own Private Key, the other Servers detect it and let the Client know the request won't be processed. This happens because, since we have no leader change implemented and only the leader can initiate the consensus, if they are byzantine, when the other Servers detect the wrong signing, they have to reject it and no quorum can be reached.

2.5 Byzantine Server sends a wrong value for a Strong Check Balance

When a Client requests a Strong Check Balance, if one of the Servers is Byzantine and answers with a value different than the other correct Servers, if the Client receives the three correct (and equal) values first, they'll accept that value. Instead, if the value sent by the Byzantine Server is not the last, the Client will request a second phase and the request will be processed in a block.

3 Available Operations

3.1 Create Account

When a Client creates an account, they send their public key in the request, so the Servers verify the Client isn't trying to open an account with a public key which is not theirs. If this rule is not met, the Servers answer the Client with a NACK. The Client also has to send a positive amount to open the account.

3.2 Check Balance

This is an operation that only queries (and does not update) the state of the blockchain, so it is allowed to run a simplified protocol that supports both the following consistency modes.

3.2.1 Strongly consistent read

Allows for reads to execute faster than normal blockchain operations while still obeying the same atomic (linearizable) semantics that are provided by the Istanbul BFT protocol. We implemented by in a first phase, asking all replicas for balance Value of his account and waiting for $2f+1$ responses, if all replicas return the same value it is accepted as the value of the account, if some value differs, we do a second phase that replicas put it in the block as a normal request so we have a atomic value.

3.2.2 Weakly consistent read

Allows for reads to provide a correct but stale output, in a way that such reads can still execute in weak connectivity scenarios, namely when the client can only connect to a single replica. We implement it doing a Merkle Tree, each on each time the replicas create a snapshot (and let every other replica know so that they can check it) of all accounts on a merkle tree, when a replica asks for a Weak Read, they return the state of the account and a proof so the client can verify if the value sent is not wrong, even if is not atomic.

3.3 Transfer

A Client can request a transfer of a positive amount between a pair of accounts provided:

- They own the corresponding private key of the source account where the money is withdrawn from (and so have authority to perform the transfer);
- They have enough balance in their account to cover the amount to be transferred plus the fee paid to the leader (block producer);
- The destination account exists.

If any of these requirements are not met, the Servers answer the Client with a NACK.

4 Dependability Guarantees and Security Attributes

Here we'll discuss some of the essential characteristics related to security and dependability and explain why we included them and how we achieved them.

Confidentiality: based on the project requirements, it was decided that confidentiality wouldn't be necessary for this implementation.

Integrity: since we are using UDP, the network is unreliable and communication channels are not secured, every time any data is transferred from Client to Server and between Servers, we perform integrity checks through the usage of message signatures. This means that any time we send a message, we hash it (using SHA1) and encrypt it with the sender's private key. Then the receiver must decrypt the message with the sender's public key and verify if the sender is correct.

Reliability: we can provide reliability over the period of time that the servers are active and correct, because we still need to reach a quorum.

Freshness: the messages are sent with a message nounce (which is also signed with the sender's private key) and when a reply is received, the sender checks if the message nounce received corresponds to the expected value, guaranteeing it is not an old message being replayed.

Authentication: our project provides integrity and freshness, so it also guarantees authentication.

4.1 References

[1] Henrique Moniz. The Istanbul BFT Consensus Algorithm.

<https://arxiv.org/pdf/2002.03613.pdf>