SEC Project Report

2022-2023

HDS Ledger (HDL)

Stage 2

MEIC-T, Tagus Park, Group 19

Afonso Bernardo (96834), César Reis (96849), Henrique Vinagre (96869)

I. System Description

The whole project resides on a smart contract concept, implementing a Token Exchange System, where the clients can create and manage their accounts via transactions being made for the system. Besides the validity of the client transactions, this system runs over a blockchain (HDS Ledger) which is over a consensus protocol - the IBFT algorithm [1] discussed in the previous stage, now completed; abstracting these layers.

II. System Design (with implementation)

As the previous stage, our program continues entirely written in JAVA [2] with the MAVEN [3] build tool, using the same cryptographic primitives for signatures and abstractions such as channels and HDL Process information. Moreover, the system stays totally asynchronous, without any temporal bound.

We modify the way of authenticating messages across servers by always using HMAC and when needed, signatures for processing cost reduction purposes.

- Some assumptions:
  - There is a static leader that can be byzantine in our implementation, although we presume the view change phase of the IBFT algorithm is being made, abstracting consensus in this stage.
  - The same instance of a client's API only performs a transaction at a time, meaning that no parallel requests are made for the same client.
- Client API:
  - As the previous stage, the client API has the responsibility of serving client requests. In this case, the client can accomplish following commands (which are related to API services proposed in the assignment):
    - C X - create and API for client X;
    - T C X - client X creates an account on the TES;
    - T T X Y A - client X transfers A tokens to client Y;
    - T B X Y [S/W] - client X checks balance of client Y with **S**trong or **W**eak read guarantees.
  - These commands are restricted in terms of permissions to access an account. That is, the clients can only manipulate their accounts or update others with the *transfer* operation, but nothing stops them from checking the balance of other accounts as soon as it is not a problem.
  - We used the nonces in the transactions to ensure that no responses for the previous transactions are misunderstood as responses for the current transaction. More precisely, the server's response should contain the same nonce as the pending transaction at the moment.
- Client Transactions:
  - All transactions that update the state of an account must be signed with the corresponding private key of that account (the account identifier is the public key).
  - Before a transaction becomes a part of a block, it undergoes **nonce** and syntax checks after signature verification to ensure its validity. Once added to the

blockchain, the block undergoes semantic analysis on the transactions at a specific time to separate syntax from semantics.
    - If a block does not get full of transactions within a fixed time interval after the first transaction comes through, it gets proposed as is. This allows for clients to not be (potentially) waiting indefinitely for a response.
    - Whatever server manages to successfully append a new block to the blockchain gets a reward from the creators of the transactions (clients), proportional to the amount of them in that block.
    - Regarding the read-only operations accomplished by the *check_balance* service, we design optimized reads by bypassing the blockchain layer complexity and the clients are able to choose what level of consistent as follows:
        - Strongly consistent read: simple direct to *consensus*(balance). This process involves performing a strong, consistent read operation on each server to obtain the current balance of the queried account. Through the IBFT consensus protocol, the system determines the most recent timestamp of the balance, ensuring accuracy and consistency across all servers only using one RTT (the COMMIT phase), as we are simply reading from the servers.
        - Weakly consistent read: state of all modified accounts is signed and then propagated by each server (with a unique type of message), in order to have a snapshot of the system every once in a while. This is done every 3-5 (or more) block attachments to minimize the network exchange overhead. When the client requests a weakly consistent read, the server should send the latest of these "snapshots" with *f*+1 signatures for that account. This way, the client's API can verify that there are indeed enough valid signatures to ensure a correct read.

- TES:
    - All the account information of all clients we called the TES state of our system.
    - Each server keeps the TES states (TESState) for each timestamp when it sees a change in the current TES state to allow the strongly consistent read operation after the *timestamp* consensus is reached.

III. Threat analysis

- Transaction authenticity is according to the private key of the creator, ensuring **non-repudiation**, and therefore, no personal attacks are doable for *update* operations;
- TES Account **authorizations** (with the usage of any Public Key to identify accounts) transparent to transaction semantics;
- With the transaction semantic validation approach, a vulnerability arises where clients with null accounts can create transactions infinitely-often, filling up blocks without execution, leading to potential denial of service (**DoS**) attacks. This could be mitigated with timeouts,, for example blocking a client if he sends 3 invalid operations in a short amount of time.
- We ensure replay attack protection by using nonces on client's transactions.

- When performing a weakly consistent read, a server could respond to the client with "404: Account not found" when the account actually exists, and this would be considered by the client as a valid response (as opposed to that, if the account did in fact not exist, this would be the legitimate server response). However, we do think it meets the weakly consistent read specification, quoting: "(...)*allow for reads to provide a correct but stale output(...)*" and this response is indeed correct and **stale**, because sometime in the past the account did in fact not exist. Moreover, the client can verify it because he got the confirmation of the account creation transaction, so he knows the account is there (we don't have forks, the blockchain always moves forward).

IV. Dependability guarantees of the system

- As above referred, our implementation abstracts the consensus layer which implies we trust the supported algorithm and we get its dependability guarantees.

- Blockchain guarantees
    - Once a node (in this case, a TES block) is appended to the blockchain, it cannot be removed or altered in any way.
    - If a correct server decides a new node to append to the blockchain, eventually every correct server will decide the same node to append in the same order to the blockchain.

- Our system (TES) guarantees:
    - Transactions:
        - non-repudiation, for **all** transactions, even for transactions which are invalid.
        - authenticity, each transaction is signed by its author (and includes a nonce).
    - Reads
        - Strongly consistent: always the latest and correct read (after any other client successful transaction).
        - Weakly consistent: provides a correct read of an account, relying on a single server (less demanding on network resources).
    - Accounts do not drop below 0 (zero) tokens (even if he performs more transactions).
    - A client API eventually gets a response (or a quorum (up to 2f + 1) of them, whatever is needed) with the status of any operation he requests.

V. References

[1] Henrique Moniz. The IBFT Consensus Algorithm

[2] Java docs

[3] Maven build tool