Highly Dependable Systems

HDS Coin Stage 1

André Soares – 82542

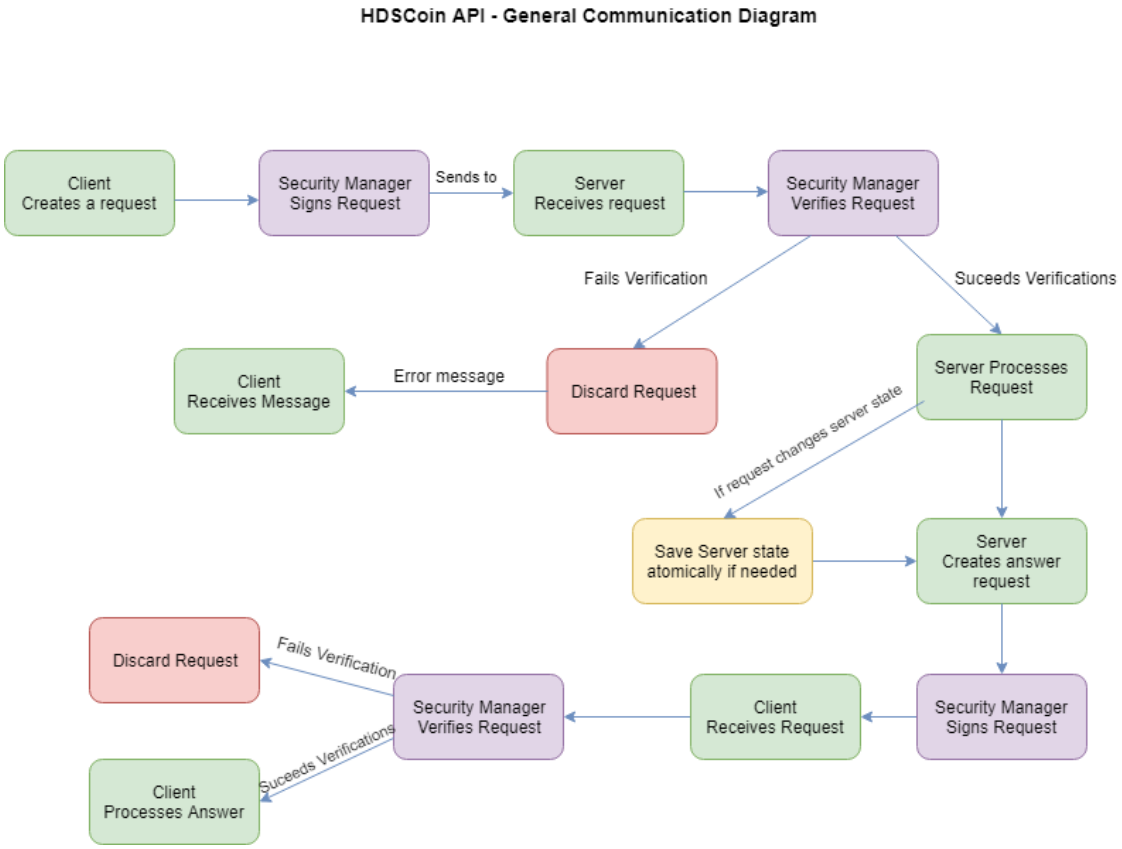
João Freitas – 81950

Rúben Martins – 79532

Considerations

The protocol designed following the requirements assumes that a Public Key Infrastructure is already in place, using RSA with 2048 bits, for simplicity and that the server is non-malicious.

General Communication and avoiding MITM



The communication protocol we implemented guarantees that a message that changes the state of a given account is always validated in order to be processed by including a digital signature of all the information sent in a Request and that a response sent by the server is also verified as being sent from the server.

This ensures that operations that manipulate an account are properly authenticated as coming from the corresponding private key owner and that the user who made a Request cannot declare that they did not do it at a later time as the server would have rejected such a Request.

In order to do this, both client and server possess a SecurityManager, SM, which signs and timestamps a request using SHA256 for the hashing algorithm and RSA for the encryption using the Java Security API, and also validates given signatures against a public key.

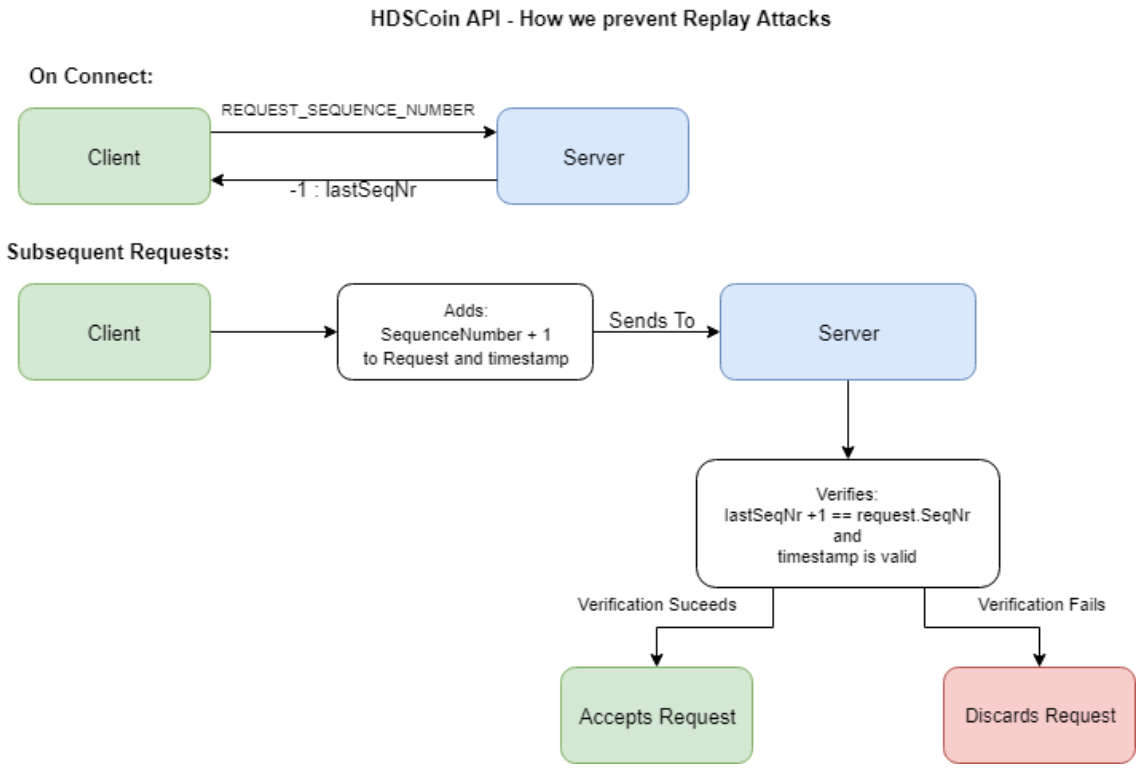
This signing and validation operations done by the SM ensure that a request cannot be tampered with or replayed as such a request would be discarded and also all the information in the request is information that is publicly available and represents no danger if captured. As such, Man In The Middle attacks are useless.

All the Requests done between the client and the server are subject to this protocol.

Avoiding Replay Attacks

In order to avoid and mitigate replay attacks, a Sequence Number is included in each of the requests, which is kept in both the client and the server.

This Sequence Number is negotiated at the start of a connection between client and server and is used in all subsequent requests.



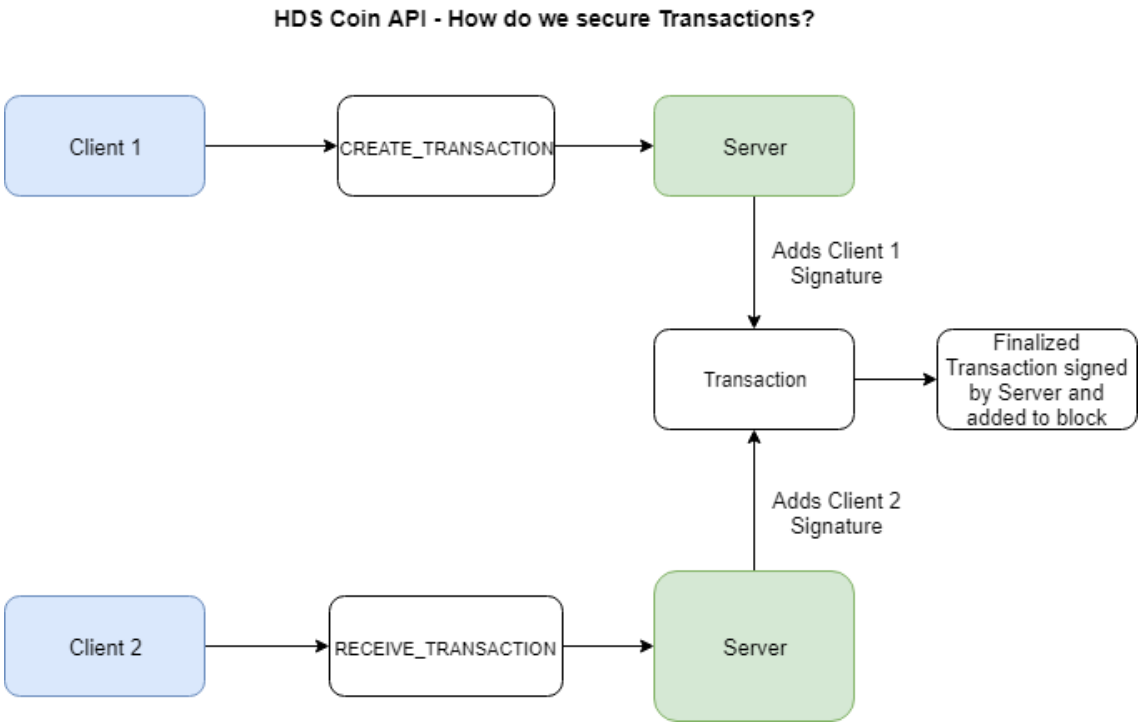
Two timestamps are also included in all Requests, to help reduce the attack window in which attackers could change them.

Securing Transactions

Transactions are probably the main part of the project, in which a user can send or receive coins from other users.

The requests, which we will call CREATE_TRANSACTION and RECEIVE_TRANSACTION, are subjected to the General Communication in order to ensure the non-repudiation and

authentication of the requests.



In this example, Client 1 starts by requesting the server to create a transaction to Client 2, to which the server responds by creating a Transaction with Client1 as the source and Client2 as the destination, and adds the signature of the request made by client 1.

Following this, Client2 can now execute a RECEIVE_TRANSACTION request, giving the Client1's public key as an argument, to which the server will check if there's a pending transaction from Client1 to Client2.

If such a transaction is found and is not processed yet, the transaction is processed and client2 request's signature is also added to the transaction. After finalizing the transaction and ensuring everything was alright (no negative balances are left in the account), the server signs the clients signatures that were added to the transaction and puts it in a block.

The block part is an added bonus that we made for fun that uses a simple proof of work algorithm that would validate blocks and ensure that they cannot be tampered with, that is, saved information cannot be altered as it would break all the blocks, making such information immutable.

Atomic Persistence

In order to achieve persistence, we export the current state of the ledger to a temporary file using JSON to serialize the data, and then, after the writing operation is done, we use a rename function to move the file to the final one, which is an atomic operation under the file system, using NIO.Files API. If the writing fails at any moment, or the process dies, the current backup wont be altered.