HDS Serenity Ledger

Simão Silva
Instituto Superior Técnico

Miguel Pato
Instituto Superior Técnico

Marco Castro
Instituto Superior Técnico

Abstract

HDS Serenity Ledger is a permissioned blockchain system, offering high dependability guarantees. This service allows its clients to perform transactions of a virtual coin between them, whilst simultaneously preventing up to f byzantine nodes from making arbitrary changes to the state of the system.

1 Introduction

In distributed systems, Byzantine faults refer to the potential for nodes (or components) to exhibit arbitrary and malicious behavior, posing significant challenges to system reliability and integrity. Coined from the Byzantine Generals' Problem, this fault model assumes that some nodes may act in a Byzantine manner, arbitrarily deviating from protocol specifications, and potentially colluding to disrupt system operations.

Byzantine faults can manifest in various ways, including nodes sending conflicting information, providing incorrect responses, or even intentionally withholding critical data. Such behaviors can undermine the trustworthiness of distributed systems, leading to incorrect outcomes, data corruption, or system failure.

Addressing Byzantine faults is crucial for ensuring the dependability and security of distributed systems, especially in critical applications like blockchain networks and decentralized finance platforms. Robust mechanisms for Byzantine fault tolerance (BFT) are essential to maintain system functionality and consistency, even in the presence of malicious actors.

In the context of the HDS Serenity (HDS2) project, understanding Byzantine faults is paramount due to the inherent risks they pose to the dependability and security of the developed blockchain system. As mentioned earlier, Byzantine faults encompass arbitrary and malicious behaviors that can disrupt system operations and compromise data integrity. This fault model is particularly relevant in distributed systems like blockchain networks, like the one developed in this project, where nodes must cooperate to achieve consensus and maintain a consistent ledger of transactions.

2 Implementation

Regarding the implementation, we have meticulously crafted various components essential for the functionality and security of our distributed ledger system. This section delves into the intricate details of transactions, blocks, and the robust verification mechanisms deployed across the network.

2.1 Transaction

Our transaction structure has the following parameters:

- **senderPublicKey** - Corresponds to the public key of the client that will act as the sender
- **receiverPublicKey** - Corresponds to the public key of the client that will receive the coins
- **nonce** - A 64 bit value, to prevent replay attacks: the first 32 bits correspond to a random number, and the last 32 bits correspond to a message that is incremented each time the client sends a transfer request.
- **amount** - The amount that will be discounted to the sender's account
- **fee** - The amount of coins that the leader of the consensus will receive when the block (in which this transaction is inserted) is accepted. The receiver will be credited **amount** - **fee** coins. This parameter will not be filled by the client when submitting a transfer request, but instead by the leader of the consensus instance before proposing a block.
- **transactionId** - Unique identifier of this transaction, corresponds to *hash*[1]*(senderPublicKey* || *receiverPublicKey* || *amount* || *fee)*. It serves to check the **integrity** of the transaction

[1]The used hashing algorithm was SHA-256

- **signature** - The signature parameter serves to prove that this transfer was submitted by the sender, thus ensuring **authenticity** and **non-repudiation**. The signature corresponds to: *AsymmetricEncrypt*[2]*(transactionId, SenderPrivateKey)*

2.2 Block

Our block structure has the following parameters:

- **index** - Corresponds to the position in the ledger/blockchain where this block will be appended.
- **Array<Transaction>** - Array storing the transactions that constitute this particular block. The size of this array is configurable - a block can have 100 transactions, or just one, depending on the settings of the service.
- **totalFees** - Corresponds to the sum of all fees in *Array<Transaction>*. It will be the quantity that the leader of consensus will receive.
- **blockProducer** - States which node produced the block, i.e., the leader of the consensus protocol when this block was decided.
- **blockHash** - Just like *transactionId*, the hash identifies a particular block, and corresponds to: *hash(index || transaction1 || transaction2 ... || totalFees || blockProducer)*.
- **signature** - Similar to the *signature* in transaction, it is equal to *AsymmetricEncrypt(blockHash, BlockProducerPrivateKey)*.

2.3 Transaction and Block Verification

We will now show the mechanisms we have implemented to prevent byzantine nodes and clients from being able to commit invalid transactions and blocks. We will present several verifications that are done, separated by part of the system where these checks are done.

2.3.1 Link Level Verification

We modified the link class provided in the base implementation so that it acts as an Authenticated Perfect Link. We achieved this by changing the *send()* method in this class to append a digital signature to each message that is sent (this digital signature is created using the sender's private key and the totality of the message data); we also changed the *receive()* method to verify this digital signature in every message that is received. This ensures that a client cannot impersonate another client, or that a node cannot impersonate another node. However, it does not give full guarantees of authenticity in every message(for example, it does not prevent a byzantine node from proposing a transaction that was never sent by a certain client).

[2]We used RSA encryption and keys for this

2.3.2 Transfer Request Verification

Transfer requests are submitted by clients to nodes, in order to transfer coins from the sender's account to another client. Each node needs to perform several validations of the parameters of the submitted transaction structure (that is explained in Chapter 2.1), such as:

- **Valid Public Keys** - The node needs to confirm that both the sender's public key and the receiver's public key correspond to public keys of existing clients in the system, and that the sender of the message corresponds to the sender's public key in the transaction
- **Transaction Authenticity** - Then the node confirms the authenticity of the transaction (i.e., this transfer request was originally created by the sender of the coins), by verifying the **signature** parameter. This is accomplished by doing *AsymmetricDecrypt(signature, senderPublicKey) == transactionId*.
- **Transaction Integrity** - Then we verify this transaction's integrity, by doing *hash(senderPublicKey || receiverPublicKey || amount || nonce) == transactionId*
- **Freshness** - It is also vital to prevent replay attacks - to prevent these, we make use of nonces; as each transfer request is submitted and accepted, each node stores these nonces, and if any client repeats a nonce, the corresponding transfer request is dropped.
- **Business Logic Verification** - Lastly, each node must guarantee that the transaction request doesn't violate business logic, and in our case we must ensure that, if the transaction was to be submitted at that given time, the sender's balance should be greater or equal to zero.

2.3.3 PrePrepare Verification

A pre-prepare message occurs when a leader has gathered enough transactions to form a block, and broadcasts this preprepare message to all nodes. Each node must independently evaluate the proposed block, to ensure its integrity and authenticity (and also the integrity and authenticity of the transactions that this block carries):

- **Transaction Verification** - Upon receiving a block in a preprepare message, each node will iterate over the transactions it carries, and perform the following checks in each transaction:
  - **Transaction Integrity**
  - **Freshness**

  - **Transaction fee** - The fee in the transaction must not be lower or higher than the following value: *TransactionOriginalAmount * FeePercentage*. *FeePercetage* is set in a configuration file, and it is equal to all nodes in the system.
  - **Business Logic Verification** - Besides checking each transaction individually, each node must verify that, if a block contains more than one transaction, if the totality of the transactions is executed, no client has a negative balance.

- **Block Verification** - Now that each transaction was verified, some extra verifications must be performed in the block:
  - **Block TotalFees Parameter** - The sum of the fees in each single transaction must match the value in the *TotalFees* parameter of the block.
  - **Block Integrity** - Each node must verify block integrity by performing: *hash(index || transaction1 || transaction2 ... || totalFees || blockProducer) == blockHash*
  - **Block Authenticity** - Lastly, to verify that this block was originally created by the node that is in the *blockProducer* parameter, we verify if: *AsymmetricDecrypt(blockHash, blockProducerPublicKey) == blockHash*.

2.3.4 Prepare and Commit Verification

It would be redundant and expensive to perform all the verification steps that are done in preprepare, prepare and commit messages; In these, each node checks the following:

- **Transaction Verification**
  - **Transaction Integrity**
  - **Transaction Authenticity**
- **Block Verification**
  - **Block Integrity**
  - **Block Authenticity**

The reason that each node only does integrity and authenticity checks in prepare and commit messages, as opposed to doing all the verification steps done in preprepare messages is because, in order for a block to reach the prepare stage, it must have go through the preprepare stage; therefore, we only need to ensure that it **remains the same unaltered block** (i.e., verifying integrity).

3 Testing and Validation - System Operations Byzantine Faults

To make sure the system implemented is working as expected we've developed a test suite that tests the operations that were implemented (checking the account balance and transferring money to another user), some exceptions that should happen when there is wrongful usage of said operations (like transferring money to a user that doesn't exist) and most importantly the behavior of the Serenity Ledger under byzantine behavior.

3.1 System Testing

Under the umbrella of system testing, we scrutinize the core operations of the system: *check balance* and *transfer*. These tests aim to validate the functionality of these operations under normal conditions, as well as identify and handle any erroneous usage.

- **Operation Validation:** We execute tests to confirm that *check balance* and *transfer* operations perform as expected, ensuring that account balances are accurately maintained and transactions are executed correctly.
- **Exception Handling and Wrongful usage:** Our suite includes scenarios where operations are misused, such as attempting to transfer funds to non-existent accounts or trying to transfer negative amounts of money. These tests verify that the system appropriately handles such exceptional cases, preventing erroneous or malicious actions from impacting the ledger's integrity.

3.2 Byzantine Behavior Tests

We have prepared a total of five tests to test the system under byzantine behavior; in each test, the byzantine node has a different behavior, testing the system's ability to handle/ignore that particular malicious behavior. In all of these tests, there are a total of four nodes and two clients, the amount of transactions in each block is 1 and the transaction fee is 5%. Here are the four behavior types that our tests cover:

3.2.1 Test 1 - Byzantine Leader

In this test, the initial leader is a Byzantine node, and it generates a random block that it then tries to propose in the consensus algorithm; obviously, the transactions in this block have not been performed by the respective clients, therefore this block should never be committed. To verify, we perform a check balance in each of the clients and ensure that it is equal to the initial value of each account (in our case, 1000 coins).

3.2.2 Test 2 - Byzantine Change Transaction

In this test, the Byzantine node is not the leader, and it tries to change the amount (double it) of one of the transactions in the proposed block. The other nodes should ignore the messages from the Byzantine node, but still be able to decide on the original block. The original transaction was a transfer of 50 coins from *client1* to *client2*; in the end of the test we verify if the balance of *client1* is 950 coins, and the balance of *client2* 1050 coins minus the fees(which equals to 1047.5 coins if the fee is 5

3.2.3 Test 3 - Byzantine Change In Fees

In this test, *client1* tries to transfer 50 coins to *client2*, and node 3 tries to change the amount to 100 coins; the decided block should not have this fee (but instead 2.5 coins of fee for the single transaction in the block). To verify this, we ensure that *client2* has a final balance of 1047.5 coins.

3.2.4 Test 4 - Byzantine Leader Tries to Insert Invalid Fee

Similar to test 3, but in this case the Byzantine node is the leader. It creates a valid block with a transaction request of 50 coins from *client1* to *client2*, but inserts an invalid fee amount. The expected behavior is for the block to be rejected; then, there should be a round change (as the Byzantine leader is not able to commit its block). Node 2 should then take over, proposing a correct block, that should be decided. In the end of the test, *client1* should have 950 coins and *client2* 1047.5 coins.

3.2.5 Test 5 - Client Invalid Signature

In this test, we do not test a Byzantine node but rather a Byzantine client. The client sends append requests to the nodes with an invalid signature (using the wrong private key). Nodes should not act upon these requests, and the ledger should remain empty. This test validates if the nodes correctly validate the signature of client requests.