HDS Serenity Ledger - A Highly Dependable Ledger

André Torres
andre.torres@tecnico.ulisboa.pt

Gonçalo Nunes
goncaloinunes@tecnico.ulisboa.pt

Pedro Lobo
pedro.lobo@tecnico.ulisboa.pt

Abstract

This paper presents the HDS Serenity Ledger, a simplified permissioned (closed membership) blockchain system with high dependability guarantees. It provides a cryptocurrency component that allows clients to perform cryptocurrency transfers between them. It uses the IBFT byzantine consensus protocol [1] to achieve consensus under a byzantine model.

1 System Design

The system is composed of three main components: the Client, Library and Node modules.

1.1 Client

The client component is responsible for interacting with the user through a command-line interface, where the user enters the commands that are later parsed and then handled by the client's local library component.

1.2 Library

The library is responsible for translating the client's requests into requests to the service, effectively triggering instances of the consensus protocol. The library acts as a client of the blockchain. It does some validation on the request parameters issued by the client. However, as noted in section 1.5, clients can exhibit byzantine behavior, effectively bypassing the validations made by the library. When the library issues a request to the node's service, it blocks, returning when a sufficient number of responses is collected, as mentioned in section 2.6.2.

1.3 Node

The node component contains the server-side logic responsible for keeping the state of the system and collectively implementing the blockchain system. The nodes collect requests made by the library component into a transaction pool. When the transaction pool has enough transactions to form a block, the block is proposed as a value for a consensus instance. The block is then decided and the component state is updated in accord to recently appended block's transactions.

1.3.1 State

The application state that is held by the node component maps each of the registered public keys (clients and nodes) into an account held by that entity. The account holds information about the owner and the current amount.

1.4 Requests

There are two types of requests that can be issued by the client's library:

- **Balance Requests**: Obtain the balance of the account associated with the key specified.
- **Transfer Requests**: Send a given amount from a `source` account to a `destination` account, if the balance of the `source` account allows it. The transaction is only performed if the client issuing the transaction is the owner of the `source` account.

1.5 Assumptions

The system was developed under the following assumptions, concerning the system's components and environment:

- The network is unreliable. It can drop, delay, duplicate, or corrupt messages, and communication channels are not secured.
- Up to f out of $3f+1$ nodes can exhibit byzantine behavior.
- Client's can exhibit byzantine behavior. This means that the requests coming from the library component may not be valid and a compromised library may try to attack the service.

- Private keys are kept private and attackers can't access encrypted data without the corresponding decryption key, nor use brute force attacks to crack private keys.

1.6 Dependability and Security Guarantees

The application offers the following dependability and security guarantees:

- The balance of each account should be non-negative.
- The state of the accounts cannot be modified by unauthorized users.
- The system should guarantee the non-repudiation of all operations issued on an account.

2 Implementation Aspects

2.1 Transaction Pool

Each node maintains a transaction pool, which holds the requests issued by the clients. When receiving a new transaction, the node checks if there are enough pending transactions in the transaction pool to propose a new block. Note that, although blocks have a configurable fixed size that is specified when starting the node component, some blocks will not be filled to their full capacity.

To avoid executions where a client issues an operation that is waiting for a block in the transaction pool, but there aren't enough transactions to fill a block, the node periodically checks if such transactions exist and will propose a block to consensus that is, exceptionally, shorter than the designated block size. This optimization offers a shorter response time when the system is under lower load, as blocks don't have to be filled up to their capacity to be proposed to consensus.

2.2 Replay Attacks

To avoid replay attacks, each operation issued by the client has a nonce. Upon deciding a block, each node registers that particular nonce, issued by that particular client, checking if it was already used. When validating transactions, as detailed in section 2.3.1, transactions with repeated nonces are invalidated.

2.3 Transaction Validation

Upon deciding a block, each node iterates over the block's transactions performing general validations and specific validations (specific to the type of operation). Transactions which successfully pass the validation stage are then applied.

Transactions which are invalid are marked as unsuccessful. Despite these transactions being appended to the ledger, they have no effect on the application state. Upon performing the validations, a response is sent to the client confirming the success or failure of the transactions.

2.3.1 General Validations

The general validations are applied to every type of transaction. Firstly, the nonce of the transaction is checked. If that nonce has already been used by the client who issues the transaction, it is invalidated. Upon issuing a request, the library signs the request with the client's private key. This signature is verified with the client's public key. If the transaction was not issued by the client, it is invalidated. Note that the nonce is also signed. A malicious agent trying to change the nonce would have to forge the client's signature. For that, the malicious agent would need the client's private key.

2.3.2 Specific Validations

After applying the general validations, depending on the type of the transactions, specific validations are performed.

For balance transactions, the only validation performed is that the specified account must exist in the system.

For transfer transactions, the following validations are performed:

- The amount to be transferred must be positive.
- The source account must exist in the system.
- The destination account must exist in the system.
- The client issuing the transactions must own the source account.
- The source account must have enough balance to perform the transfer. This includes the amount specified in the transaction as well as the fee that must be paid to the block producer, as specified in 2.4.

2.4 Leader Fee

One of the proposed requirements is that all update transactions must pay a fee to the block producer, which is the leader who manages to add the block to the blockchain. As there is the possibility of changing leaders during a consensus instance, it is not clear which node is the block producer, and who must the fee go to.

2.4.1 Naive Implementation

A naive implementation would be for the block proposer to sign the request with its private key, encapsulating the request in a signed message. However, any byzantine node would be able to decipher the request and sign it with his own private key, passing as the block proposer. Furthermore, this approach would only be correct in executions where the block proposer and the block producer are the same.

2.4.2 Block Producer as Consensus Value

Another approach would be to, instead of submitting the block as the consensus value, submit a pair (`block`, `leader`). The value picked for `leader` would be the chosen leader whom the fee is paid to. This solution would not necessarily consider the leader to be the node that produced the block. It would only make all the nodes agree on the block producer.

2.4.3 Block Producer Function

Another approach would be, for block with sequence number λ, to use the deterministic function that maps a pair (λ, r) to a node identifier, considering the output of the function for $(\lambda, 1)$ as the block producer. This approach leads to situations where a crashed node would earn the fee, despite not contributing to the system. Consider that the considered node doesn't propose any block and, suspecting it to have failed, another node starts a consensus instance and, after the round change happens, it is elected leader. If the block is decided, the crashed node would earn the fee, despite the other node producing the block.

This was the implemented approach.

2.5 Ledger

When a new block is decided, it is appended to the ledger. All requests contained in the block, balance or transfer requests, valid or invalid, are kept in the block that is appended to the ledger. This ensures the system guarantees the non-repudiation of all operations issues on an account, as mentioned in 1.6. Only blocks with no transactions are not appended to the ledger.

2.6 Number of Requests and Responses

2.6.1 Number of Requests

A library's client, when issuing a request, sends it to a byzantine quorum of nodes, corresponding to $2f+1$ nodes.

Sending the request only to $f+1$ nodes wouldn't be sufficient as, in the round change protocol specified in the IBFT algorithm [1], correct nodes will only broadcast a `ROUND-CHANGE` message when receiving a valid set of $f+1$ `ROUND-CHANGE` messages. In the worst case, sending the request to $f+1$ nodes could result in delivering the request to f byzantine nodes and a single correct node. If the f byzantine nodes chose to not broadcast a `ROUND-CHANGE` message, the algorithm wouldn't be able to make progress, violating the termination property. Note that the other $2f$ correct process wouldn't broadcast a `ROUND-CHANGE` message as they are not aware that another consensus instance has started.

2.6.2 Number of Responses

Upon sending a request, the client's library blocks waiting for the confirmation for that request, indicating if the transaction was successful or not. The library returns when receiving $f+1$ confirmations that the transaction was successful. Note that this is enough as the agreement property of the IBFT consensus algorithm [1] states that, *if a correct process decides some value v, then no correct process decides a value v' such that $v' \neq v$*. As there are at most f faulty nodes, one of the $f+1$ must come from a correct node.

3 Behavior Under Attack

To test the system, various configuration files were created. Each one specifies a byzantine behavior. Each of the tests parses one of these configuration files. The implemented byzantine behaviors include:

- **None**: The node is correct. Normal execution.
- **Silent**: The node doesn't send messages. It acts as a crashed node.
- **Drop**: The node drops all packets. The node sends messages but doesn't react to any of the received messages.
- **Fake Leader**: The node acts as the leader. When a new round starts, it sends a pre-prepare message.
- **Default Value**: The node sends consensus messages with a default value (an empty block).
- **Replay Leader**: The leader node duplicates an existing transfer transaction in the block being proposed.
- **Greedy Client**: The client swaps the transfer transaction's source and destination accounts, trying to transfer money to its account.
- **Drainer Client**: The client sets the transfer transaction's transfer amount to its symmetric, trying to drain money from other accounts.

The tests were implemented using JUnit testing framework. When running the tests, it is important to note that, sometimes, tests fail as the socket could not be bound to the port specified in the test configuration. This happens because the socket used in the previous test was not released before the start of the following test. To circumvent this problem, each of the tests can be run manually, one by one.

In the next sections, the defense mechanisms that allow the system to behave correctly under attack are specified, for each of the byzantine behaviors.

3.1 None

All nodes are correct. The system behaves as expected.

3.2 Silent

A node with a `silent` byzantine behavior behaves as a crashed node, not receiving nor replying to any messages.

If the crashed node is not the leader, the nodes can still agree on a value as only $2f + 1$ nodes are needed to reach consensus. This is guaranteed by the fault-tolerant nature of the IBFT algorithm, given by its round change protocol.

As a leader, a node needs to propose a block for the system to make progress. If the crashed node is the leader for the current instance, the other nodes would wait indefinitely for the crashed node to propose a block. This problem is solved, as outlined in section 2.1, by implementing a timeout mechanism where a node that has pending transactions in its local transaction pool will propose a block containing those transactions, despite not being the elected leader for that particular consensus instance.

If the node crashes during consensus, the round change protocol of IBFT ensures that another node is elected leader, so the system can make progress.

3.3 Drop

A node with a `drop` byzantine behavior ignores all messages but can still send messages.

The defense mechanisms that protect the system against this type of behavior are the ones mentioned for the silent byzantine behavior in section 3.2.

3.4 Fake Leader

A node with the `fake` leader byzantine behavior will start a consensus instance, despite not being the elected leader for that consensus instance. The node will also try to broadcast a `PRE-PREPARE` message upon receiving a quorum of `ROUND-CHANGE` messages, despite not being the leader.

Upon receiving a `PRE-PREPARE` message, as specified in the IBFT algorithm [1], will check that the message sender is the leader and discard the message if this condition does not hold.

3.5 Default Value

A node with the `default value` byzantine behavior will propose an empty block as the consensus value, despite having pending transactions in its local memory pool.

The defense mechanism outlined in section 2.1, where a node will propose a block with its local pool's pending transactions, after a timeout, ensures that the system will make progress and the transactions in the faulty node's transaction pool will be served, as any client's library sends a transaction request to $2f + 1$ nodes.

3.6 Replay Leader

A node with the `replay leader` byzantine behavior will, when leader, propose a block with a duplicate transfer transaction, effectively trying to convince the other nodes to execute that transaction twice.

The defense mechanism outlined in section 2.3.1, where the nonce of the client is checked, will prevent the transaction from being executed twice. As the nonce of the duplicate transaction was already used, the transaction is invalidated.

3.7 Greedy Client

A client with the `greedy client` byzantine behavior will swap the transfer transaction's source and destination accounts, trying to transfer money to its account.

The specific validation mechanism mentioned in section 2.3.2 will invalidate this particular transaction, as the transaction sender does not own the source account.

3.8 Drainer Client

A client with the `drainer client` byzantine behavior will set the transaction's transfer amount to its symmetric, trying to drain money from other accounts.

The specific validation mechanism mentioned in section 2.3.2 will invalidate this particular transaction, as the transaction's transferred amount has to be positive.

References

[1] Henrique Moniz. The Istanbul BFT Consensus Algorithm. May 2020.