Highly Dependable Systems Final Delivery: HDS Serenity Group - 8

Francisco Sousa
Instituto Superior Técnico

Miguel Porfírio
Instituto Superior Técnico

Sara Aguincha
Instituto Superior Técnico

Abstract

Traditional threats to blockchain systems like unintentional errors or even malicious actors frequently compromise the dependability of permissioned blockchains. Byzantine faults where nodes can, either purposefully or unintentionally, deviate from the norm are especially hard to overcome. This project introduces HDS Serenity Ledger, a system that uses Istanbul BFT [1] to achieve high reliability and dependability.

HDSS is extended with a cryptocurrency application, enabling secure account transfers and readings, while upholding non-negative balances, authorization enforcement, and non-repudiation. We explore and deter any future Byzantine attacks, being it from clients or servers.

Through extensive testing, we reveal our system's robustness against several different types of attacks. HDSS proves itself as a dependable permissioned blockchain, achieving availability, reliability, safety, and integrity.

1 Introduction

Permissioned blockchains offer a secure and dependable method for recording transactions among pre-established clients. However, ensuring such dependability in the presence of malicious parties is still a challenge. Byzantine faults lead to major threats in this context. Through these faults, nodes deviate from the established protocol behaviour, which can range from sending incorrect information to completely halting participation, making it difficult to distinguish faulty nodes from correct ones. This unpredictable behaviour can disrupt the system's ability to reach consensus on the validity of transactions, potentially compromising system integrity and halting progress.

Our project introduces **HDS Serenity** (HDSS), a highly dependable permissioned blockchain system designed to sustain Byzantine faults. We opted to use the Istanbul Byzantine Fault Tolerance (BFT) [1] consensus algorithm, also known as IBFT, to achieve high system reliability, even under malicious behavior. This project supports a secure cryptocurrency application enabling account transfers and balance readings. The system implements crucial security properties such as non-negative balances, authorization enforcement, and non-repudiation of transactions. We also explore potential Byzantine attacks that could be launched by both servers and clients, and implement countermeasures to mitigate these threats.

2 System Design

HDSS Ledger is a simplified blockchain system with high dependability guarantees. The system consists fundamentally of *Clients* and *Nodes*. The client application makes calls to a *Library*, which translates them into requests to a *Client Service* on the nodes' side, that in turn processes those requests and sends them to a *Node Service* to trigger instances of the consensus protocol.

2.1 Communication

The communication between clients and nodes is done **equally** for both sides. It uses **Authenticated Perfect Link**, on top of the **Perfect Link**, the **Stubborn Link** and **Fair Loss Link** layers. We implemented authentication through the use of signatures. Every message is hashed and asymmetrically signed by its author. This way, all messages are authentic and preserve integrity. The hash algorithm used for signing is SHA-512 and the asymmetric encryption is ensured by RSA. Hence, all communications in the system offer non-repudiation.

Clients broadcast their request to every node in the network and each node replies to the client who sent the request after either completing the operation or upon encountering an error with the request or during the operation.

2.2 Ledger and Client Operations

HDSS cryptocurrency application supports 2 different **asynchronous** operations: **Transfer** and **Balance Reading**. A **transfer** consists of sending, through the use of public keys,

another client node in the system a specific amount. A **balance** operation consists of reading, through the use of public keys, a specific client's balance.

Both of these operations go through the *client API*, or *Library*, that serves as a bridge between the client application and the service provided by the nodes. The client simply inputs the desired operation, and the API takes care of creating the requests and handling the responses.

The client application does not block after submitting a transaction, allowing it to send **multiple** transfer operations concurrently. However, **self-transfers** are prohibited in order to prevent invalid transactions. The client eventually receives a server response indicating the outcome of the transaction. Instead of waiting for only a single reply from the nodes regarding a request, the client instead waits for **f+1** responses in order to ensure that at **least one correct** node processed the request properly. This requirement is enough as the leader rotation feature of our implementation will guarantee that this transaction will eventually be processed. Leader rotation will be discussed into more detail next.

Whenever a node receives a **transfer request**, it's queued for processing. The system waits until the queue reaches a minimum size corresponding to the maximum transactions a block can have. Once enough transfers are accumulated, the leader verifies and validates them. If there are enough verified transfers remaining to fill a block, the leader **creates** a new block and **initiates** the consensus process. Once the leader verifies the queued transfers, any invalid transactions are **discarded** and the senders of those transfers are **notified**.

The system does not allow for transactions with:

- The same sender and receiver
- A non-existent receiver
- An amount higher than that of the balance of the sender
- Invalid attributes, such as wrong signatures and repeated nonces

Upon submitting a block to the ledger, the client **pays** a transaction **fee** to the **block creator**. This fee is a fixed, small decimal number. To ensure sufficient funds, the system verifies that the client's balance can cover both the transfer and the associated fee before approving the transaction for consensus. The implementation of transaction fees introduces a **monetary incentive** for nodes to participate actively in the block creation and behave correctly. By receiving fees as rewards, nodes are **encouraged** to **contribute to the network**.

2.3 Consensus

As previously stated the system uses the IBFT [1] algorithm to ensure State Machine Replication (SRM). As the base code had the algorithm implemented, with the exception of prevention against faulty leaders, we will go into more detail on the *Round Changes* of the algorithm.

2.3.1 Round Changes

Our implementation of the *Round Change* messages follows closely the IBFT algorithm. Round Changes are mainly triggered through a timer that each node saves. Upon having the timer expire, the node broadcasts a *Round Change* message to all other nodes.

Following this logic, whenever a node receives a set of *f+1 Round Change* messages that has a *Prepared Round* Pr_j greater than the node saved *Prepared Round* Pr_i, it broadcasts a *Round Change* message with *Round* and *Prepared Round* equal to Pr_j.

Finally, when receiving a quorum of *Round Change* messages, the node searches for the highest *Prepared Round* in the quorum and sends a *Pre-Prepare* message with *value* equal to *Proposed Value* correspondent to *Prepared Round*.

However, for a *Round Change* or a *Pre-Prepare* message to be safe and not prone to be arbitrarily changed by a byzantine process, it must be justified.

2.3.2 Message Justification

Message justification is based on the fact that:

1. All *Round Change* messages that have a *Pr* and *Pv* different than $\perp$ must be justified.
2. All *Pre-Prepare* messages that have *round different than 1* must be justified.

Message justification is ensured by *piggy-backing* a quorum of *Prepare* messages with the message to be justified. We ensure this by having an *attribute justification* on the *Message* class. This attribute is not signed by the creator of the message, as the messages present in this list are already signed by the original senders of the *Prepare*.

3 Implementation Aspects

This section focuses on the significant design decisions and implementation choices that shaped our code.

3.1 Public Key Infrastructure

HDSS uses asymmetric encryption, more precisely, RSA. Thus, we store in a **Key Infrastructure** directory both the clients' and nodes' Public and Private Keys. For simplicity purposes, they are stored together and the files' names have the following format:

Private Key: id<identifier number>.key
Public Key: id<identifier number>.key.pub

Note: - No client can access another client's private key, even though they are stored together. To make the system easier to use, each public key is associated to an identifier, which

corresponds to both the *clientID* and the *nodeID*. Whenever an operation is executed, these identifiers are used for the sake of simplicity.

3.2 Leader Rotation

Our initial approach implements the IBFT [1] consensus algorithm. However, upon further investigation into potential Byzantine faults and attacks, we opted to **extend** IBFT's capabilities by implementing a custom leader rotation mechanism. In a real world scenario, if a **Byzantine leader** starts to selectively deny requests from a specific client (i.e. does not create blocks with any of those transactions), and the consensus occurs but always with transactions from other clients, this will not be noticed by the other nodes in the system. This problem creates a **denial of service** to the client in question.

In order to mitigate this attack, we implement a **leader rotation** every **5** consensus instances. By doing so, even though it might take longer than usual, it is **guaranteed** that client's correct transactions will **eventually** be committed.

Leader rotation is also beneficial to ensure that a node wants to contribute to the blockchain. If the leader is always the same until crashing, nodes don't have much of a monetary incentive to be part of the blockchain. By rotating leaders, we can distribute the monetary incentive across all nodes, instead of being targeted at a single one.

3.3 Message Non Repudiation

All messages sent between nodes and clients ensure non repudiation, as every message is signed before being sent. The method *getSignable()* is present in the *Message class* and every class that inherits from *Message* includes their own implementation of the *getSignable()*.

This method returns a transformation of the class' attributes that should be signed, and, for most cases, all attributes of the class *Class* inheriting *Message* are signed.

The only exception to this rule is *Message Justification*, as the messages inside a justification do not need to be signed (since they are already signed by the sender of the *Prepare* message). With this in mind, *getSignable()* does not return any transformation of the *justification* attribute of any *Class* inheriting *Message*.

3.4 Denial of Service

Our system is designed to prevent **denial-of-service** attacks through invalid transactions. These attacks aim to **overwhelm** the system with invalid transactions, making it difficult or even impossible to create new blocks. Normally, these invalid transactions would be **mixed** with valid ones, requiring blocks to be discarded and recreated. However, our implementation verifies the **validity** of each transaction before including it in the block, this way not needing to ever recreate any block.

4 Behavior Under Attack

4.1 Byzantine Attacks

In a blockchain system, a malicious node might attempt to disrupt operations by impersonating the designated leader or performing unintended actions. Our system has proven to withstand the following byzantine behaviours:

4.1.1 Faulty Leader Crashes

Whenever a leader **crashes**, the system should be able to continue to make progress. The *Round Changes* of the IBFT [1] algorithm ensure that whenever a leader crashes, there is **safety** across rounds and the algorithm progresses, ensuring **liveness**.

4.1.2 Node Impersonating Leader

To prevent this situation, HDSS implements a mechanism that identifies and **disregards *Pre-Prepare* messages** originated from any node other than the rightful leader.

4.1.3 Leader Modifying Transaction Requests Received From Clients

The leader can tamper with the **receiver, amount or other data** attributes in a transaction, but this will be detected by the other nodes when verifying the validity of the **transactions within** the block. Since the signature is based on the original transaction information, any changes will cause a **mismatch.** This failed verification **prevents** the block from **achieving consensus** and being added to the ledger.

4.1.4 Client Trying To Impersonate Another Client

A client could try to impersonate another client and perform a transaction **on their behalf**, but such behavior is also prevented upon verifying the **validity** of the transaction. Since the attacker does not have access to the other client's **private key**, there will be a mismatch when verifying the signature. This prevents the attacker from performing transactions as another client.

4.1.5 Node Modifying The Block In The *PrePrepare* Step

A malicious node might tamper with the block in the *PrePrepare* step, usually by **altering transactions** values and then sending a ***Prepare Message*** with modified transactions. However, this will not be an issue since a **quorum** of honest nodes need to agree on a block in the *Prepare* step for it to be valid.

4.1.6 Node Modifies The Block In The Prepare Step

Similar to the previous case, the block will not be approved in the *Commit* step due to not having a **quorum** of honest nodes agreeing on it. However, the malicious node can still try to commit the modified block to its ledger. Since this is only done locally, it is not a problem.

4.1.7 Replay Attacks From Both Byzantine Node Or Client

Any malicious actor can **intercept** a signed transaction and **resend** it as is to the blockchain. In this test, the attacker reuses a transaction that was previously committed to the blockchain. We mitigate this by enabling all transactions to have a **nonce** (number used once) and storing all nonces of all committed transactions. This way, if any transaction has possesses a previously used nonce, it will be **discarded** and won't be added to a block.

4.1.8 Absence Of Commit Forcing Round Change

In the absence of a valid **quorum** of *Commit* messages for a round *r*, each correct node will eventually have their timer expire. This will trigger a *Round Change* with some already prepared values and prepared rounds. The focus of this test is to evaluate the **justification** properties of the system, as this *Round Change* quorum will require a quorum of *Prepare* messages to be justified.

Upon having the timer expired, each correct node will **broadcast** a *Round Change* message with *Pr* and *Pv*. This will make each correct node check the *justification* of the *Round Change* message.

The *justification* list is checked by verifying the **authenticity** of the *Prepare* messages as well as if the *Pr* and *Pv* of the quorum match the **highest prepared** *Round Change* message's *Pr* and *Pv*. The same logic is applied to the *Pre-Prepare* justification, as the round is different from 1.

4.1.9 Leader Ignoring A Client's Requests

A malicious leader might try to **ignore** a specific client's transaction requests, in order to essentially perform a **denial-of-service**. This is prevented by having a **leader change** every 5 consensus instances, thus ensuring that requests from that client will eventually be processed and guaranteeing **liveness**.

4.1.10 Leader Creating Message With Large Instance

A malicious leader can try to start a consensus by sending a message with a **large instance** number. This is mitigated by having the nodes check the received instance in the ***Pre-Prepare*** message and ensuring that it is the **correct** one.

4.1.11 Node Not Receiving Messages From Other Nodes

A byzantine node might not **receive** messages from other nodes on a consensus instance. Such behavior triggers the node's **timer** and he attempts to send a *Round Change* message. While this is happening, the other nodes have already committed for that instance. The *Round Change* message sent from the byzantine node will have the other nodes reply with a **quorum** of commit messages for that round. After that, the node will commit as well, effectively canceling the round change requested by that node.

5 Conclusion

Through extensive testing, we support our system's robustness against several different types of attacks. HDSS proves itself as a dependable, permissioned blockchain, achieving availability, reliability, safety, and integrity.

While developing the system, we had real world scenarios in mind. We took into account the scalability of the system, the monetary incentives to continue having nodes contribute to the blockchain and the malignant behaviour of byzantine clients that might want to disrupt the safety and liveness of the system. Overall, we've achieved the goals that were set for the development of this system.

References

[1] Henrique Moniz. The istanbul bft consensus algorithm, 2020.