# DepChain

André Gonçalves[1], Miguel Raposo[1], and Vasco Silva[1]

Instituto Superior Técnico

**Abstract.** The report details the stages of development of a permissioned blockchain system that is resistant to Byzantine faults. The report covers the development of a byzantine consensus algorithm and the development of the transaction execution algorithm, alongside the details of the unit tests used to test each component of the system. The report then conclude on a durability analysis of the system, examining what guarantees the system offers thanks to its design.

**Keywords:** Blockchain · Byzantine faults · Consensus · Ethereum Virtual Machine · Smart contracts

## 1 Introduction

The project revolves around the development of a permissioned blockchain system called Dependable Chain (DepChain), which has high dependability guarantees and is resistant to Byzantine failures. The project is divided into two stages: The first focuses on the consensus algorithm and the second will focus on the transaction processing algorithm. This system allows clients to perform transfers between them and check their corresponding balances.

The report is split in three parts: Section 2 details the design of each component of the system and their protection mechanisms against threats, Section 3 explains the tests that were created to validate its correctness, and Section 4 analyzes its dependability guarantees.

## 2 Design

### 2.1 Authenticated Perfect Links

Since this system was developed Byzantine model in mind, some processes may forge messages, making them appear as if they were sent by another process. Consequently, the **no creation** property of perfect links cannot be guaranteed in such models. To mitigate this issue, authenticated perfect links can be employed, ensuring the following properties:

- **APL1 - Reliable Delivery**: If processes $p_i$ and $p_j$ are correct, then every message sent by $p_i$ to $p_j$ is eventually delivered by $p_j$
- **APL2 - No duplication**: No message is delivered (to a correct process) more than once

– **APL3 - Authenticity**: if correct process $p_j$ delivers message $m$ from correct process $p_i$, then $m$ was previously sent from $p_i$ to $p_j$

To enforce these properties on established channels, each message must include an HMAC to ensure its integrity, thereby preventing Byzantine processes from tampering with or forging messages. This approach requires each process to use a key pair: a **public key** to cipher the secret key that will be established between the two processes, and a **private key** to decipher this secret key, enabling them to securely share a common secret.

The implementation of the authenticated perfect links is based from UDP communication. As such, the creation of a new link involves the creation of a DatagramSocket and the creation of a new thread that will read from the same socket and process incoming messages. Objects that wish to receive messages from a Link are represented as an Observer, that receive them through a given **update** function. This removes the need of polling and limits the amount of threads used.

To guarantee that, eventually, the destination process will receive any given message, the sender process will keep sending the same message, less and less frequently with every attempt through exponential back-off. The receiver of the message will notify the sender that it received the message through an acknowledgment, with the same id as the sent message. When the sender receives the acknowledgment, they will store it in a list of acknowledged messages and stop sending.

### 2.2 Consensus

To achieve consensus among processes on the next string to be appended to the blockchain, the **Byzantine Fault Tolerant Consensus** [1] algorithm was implemented. This consensus mechanism was adapted to guarantee **weak validity** instead of **strong validity**, meaning that if all processes are correct and some process decides on a value $v$, then $v$ must have been proposed by at least one process.

To manage the consensus rounds and handle the messages that were decided upon for execution, a **Consensus Broker** was implemented. This class is responsible for overseeing the progression of consensus instances and ensuring that the agreed-upon values are properly executed after the consensus process is complete. To ensure each server is synchronized in terms of the timing of each consensus round, they are started periodically, with each lasting a well-defined finality time. When a new round is started, each server will create another instance of Consensus and create a new block with up to 8 of the oldest received transactions stored in the mempool. At this time, the leader will start the consensus algorithm via the `runAsLeader()` method, while the others get ready to receive messages via the `runAsFollower()` method. The consensus algorithm is divided into two distinct phases:

– **Read**: The leader collects the states from a quorum of processes $(2f + 1)$ to decide on the value to be written. These states consist of a **timestamp-**

**value pair**, which represents the last state that was accepted, and a **write-set**, which includes all the writes made by that process during the current consensus round.
- **Write**: The states collected during the read phase are then sent to all other processes so that they can also decide on the value to write, as the leader's decision cannot be blindly trusted if it is Byzantine. All processes decide on the value to write and send these **Writes** until a quorum of them is reached. Once this quorum is achieved, **Accepts** are sent to indicate which value the processes are ready to decide on. The value is only fully decided when a quorum of **Accepts** for the same value is received.

In the end of the algorithm, the methods `runAsLeader()` and `runAsFollower()` will return the decided block if the consensus round ended correctly, or `null` if it was aborted. If a consensus round must be aborted, due to a timeout, an invalid proposal or the impossibility of consensus, the transactions present inside of the proposed block aren't discarded. Instead, the transactions are placed back in the server's mempool. This ensures the liveness of the algorithm, as submitted transactions eventually will be decided via consensus to be executed at a given round while avoiding unnecessary request cancellation and resubmission by clients. Additionally, the instances of consensus are reused in the case of a round being aborted, ensuring that the transactions previously proposed will be decided in the follow-up round.

Once a block is decided, it is submitted to the blockchain, going through the processes elaborated on section 2.3.

### 2.3 Blockchain

The blockchain data structure will store the DepChain transaction batches in a cryptographically secured, linked-list-like manner using blocks. Each block contains multiple transactions, with a maximum of 8 per block.

To guarantee the integrity and immutability of the chain, each block includes the cryptographic hash of the previous block. This linking mechanism ensures the blocks are interconnected, both as a reference and as part of the current block's hash. As a result, any attempt to alter or insert blocks in the middle of the chain would invalidate the hashes, thus breaking the integrity of the entire chain. To compute the hash of the current block, we first need to obtain the hash of its transactions. This is achieved using a Merkle Tree, where the combined hash of all transactions is represented by the corresponding Merkle Root. Additionally, a timestamp is also included to ensure correct chronological ordering of blocks.

Before appending a block to the blockchain, it must first be validated. This involves the following checks:

- **Validate the parent hash** — compare the parent hash of the block to be validated with the hash of the previous block; they must be identical.
- **Validate the timestamp** — ensure that the block's timestamp is greater than that of the previous block.

- **Validate the transactions** — check that all transaction signatures are valid and in case they have data it matches the 4 bytes.
- **Validate the block hash** — generate a new hash and compare it to the one proposed in the block.

Among these blocks, the first one in the chain is unique and is referred to as the Genesis Block, as it marks the beginning of the state. The entire content of this block is initially loaded from a JSON file, which contains the account balance values and, in case of smart contract accounts, the runtime bytecode as well as initial storage slot keys with the respective values. As this is the first block in the chain, it will have a parent hash of `0x0000`, an empty list of transactions, and its own hash will be equal to the hash of the initial state. This initial state includes three accounts two EOA's and a smart contract: Alice's EOA with the address `0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddead`, Bob's EOA with the address `0xbeefbeefbeefbeefbeefbeefbeefbeefbeefbeef` and finally the ISTCoin smart contract account with the address `0x987654321987654321 98765432198765432198`. Alice starts with 1000 DepCoins, while Bob has a balance of 0.

To support blockchain persistence, a simple JSON-based file storage mechanism was implemented to persist the blockchain upon each new block addition. The bootstrapping logic, initially designed for the `genesis.json` file, has been extended to also support initialization from a previously persisted blockchain file. During this process, the imported blockchain is validated using the block validation steps described earlier. If validation succeeds, the system replays the entire block history to reconstruct the state as it would be at the end of the most recent block.

### 2.4 EVM

To manage the **ISTCoin** token and the accounts on the blacklist, two smart contracts were developed in Solidity and later compiled into **EVM bytecode**. The contract dedicated to blacklist-based access control allows only the owner to add or remove accounts from the blacklist, although anyone can check whether a specific address is listed. To support ownership logic, this contract extends OpenZeppelin's `Ownable.sol` contract, which provides two operations for managing ownership: transferring it to another address or renouncing it entirely.

The second smart contract handles ISTCoin transfers between accounts. It extends the `ERC20` standard, enabling the creation of a new ERC20 token called ISTCoin. This contract supports the following functions:

- `totalSupply()`: Returns the total token supply.
- `balanceOf(address)`: Returns the token balance of a specific address.
- `transfer(address, uint256)`: Transfers a specified amount of tokens to a given address.
- `transferFrom(address, address, uint256)`: Transfers tokens from one address to another, assuming enough allowance granted to the caller.

- `approve(address, uint256)`: Grants permission to another address to spend a specified number of tokens on behalf of the caller.
- `allowance(address, address)`: Returns the remaining number of tokens that one address is allowed to spend on behalf of another.

The minting performed in the contract's constructor immediately assigns the total supply of tokens to the sender which is the contract creator.

Since the `ISTCoin` contract must verify whether an account is blacklisted before allowing transfers, it would typically perform cross-contract calls to the methods of the blacklist contract. However, due to compatibility issues encountered with Hyperledger Besu, and following the professors' recommendations, the `ISTCoin` contract was refactored to inherit directly from the blacklist contract. This approach simplifies deployment by producing a single compiled contract.

To handle deployment, the `genesis.json` file is used to define the initial state of the contract. This includes:

- a balance (initially set to 0),
- the runtime bytecode to be executed by the EVM,
- and the contract's storage, written directly into the `SimpleWorldState`.

This setup emulates the constructor's behavior without executing the deployment bytecode. The predefined storage includes values for key variables such as the total supply, token name, symbol, and owner. It also includes the mapping entry for Alice's address, assigning her the entire token supply—mimicking the minting process that would normally occur in the constructor.

Regarding security, only signed transactions are allowed to reach the execution phase, thereby ensuring integrity. To address concerns related to freshness and potential replay attacks, the system relies on the account state maintained within the EVM. Each account stores a *nonce* value, which is incremented with every executed transaction. Prior to execution, the transaction's nonce is compared against the current nonce stored in the account. A transaction is only accepted if its nonce is strictly greater than the current value, effectively preventing duplicate or replayed executions.

### 2.5 Application

The application provides several operations for clients: they can either do transfers between them or check their balances. For both operations, a client request is created and broadcast to all servers to ensure it reaches the leader. Besides these two operations, there are also two more related to managing a blacklist that prevents clients from making transfers. Only the owner can add or remove clients from this list, and both operations are also wrapped in client requests and sent to the servers.

The client application will create its own authenticated perfect link to send requests to the servers and receive their responses. Details of the requests are sent in the form of a Client Request, with a unique id, a type of request, and a transaction. To ensure that the client always sends this request to a leader, the

command is broadcast to all servers. To verify the validity of the responses, the client will wait until $f + 1$ equal responses before returning the result (where $f$ is the maximum number of faulty processes). If this state can't be reached, the request will be deemed a failure. This prevents Byzantine processes from sending fake responses to the client and gives the client assurance that the response came from correct nodes.

To receive client requests, each server will open a new authenticated perfect link with a different port. The peers listed in this link will be each client, which is currently hard-coded. When these requests are received, they will be processed by a Client Request Broker, who will perform different actions depending on the type of request. If the request is an **off-chain transaction**, it involves a purely read-only operation on the state and therefore doesn't depend on consensus. As a result, these types of requests can be executed off-chain, enabling fast read access. If the request in an **on-chain transaction**, the transaction is stored in the server's mempool. The client has to wait until the servers decide by consensus to execute the transaction before receiving a response informing the client about the transaction's result.

## 3 Testing

To test the correctness and the durability in the system, several unit tests were developed, taking into account correct usages of its features and scenarios where bad actors (byzantine processes) are involved. The collection of tests are divided into 3 categories, each focused on a particular aspect of the system.

### 3.1 Attacker Model

The tests were designed to take into account byzantine processes that try to disturb certain aspects of the system, those being the consensus algorithm and the blockchain. Byzantine servers have the biggest influence on the consensus algorithm. During this testing, it was considered that they could have the following types of behavior that would be relevant for testing:

- **Quiet** - The process may ignore received messages.
- **Client spoofing** - The process may include transactions that were not created or signed by clients.
- **Selectively omitting transactions** - The process may give different information about its collected transactions to its peers, whether them being different transactions or different numbered transactions.

Each of these byzantine behaviors are represented by the enumeration class *ConsensusByzantineMode*.

Unlike the byzantine servers, byzantine clients do not have any impact on the consensus process, but instead can do malicious operations to change the state of the **EVM**. The ones that were considered for testing are the following:

- **Spoofing** - The client may pose as another client to perform operations on their behalf.
- **Replay** - The client may capture the request of another client and resend it, performing a replay attack.
- **Off-chain on-chain transactions** - The client may request on-chain transactions, such as transfer requests, and inform the server that it is an off-chain transaction.

### 3.2 Authenticated Perfect Link

These tests validate the functionality of the links. Through the testing of the perfect link, many tasks in the background are also tested, such as the key exchange protocol, message integrity checking and the acknowledgment mechanism, both explained in section 2.1. These tests examine the following scenarios:

- **Sending to peer** - A process sending a message to its peer. The test seeks to evaluate the correct delivery of the message and its acknowledgment.
- **Sending to self** - A process sending a message to itself. The test seeks to evaluate the correct delivery of the message with no need of acknowledgments.
- **Error checking** - Checking if the link returns errors when it is used incorrectly, such as sending messages to a non-existent peer and sending messages to a closed link.

### 3.3 Consensus

In the testing of the consensus algorithm, it is assumed that there are 4 servers with a possibility of having one of them being a byzantine server. Before each test, the servers are given transactions to simulate a choice of what transactions should be executed. The execution of transactions is not important for these tests, so the EVM component explained in section 2.4 is mocked. In addition to verifying the correct execution of the algorithm under ideal conditions, some tests have servers take on a byzantine behavior referenced in section 3.1.

### 3.4 End-to-end

These tests seek to validate the entirety of the system, from interactions between client-to-server, to the consensus algorithm, to the execution of transactions in the EVM. Each test represents an execution of an operation made client-side, whether being correct, incorrect or malicious.

One of these tests is a long duration, high load test that validates the correctness of the system even processing of a high amount of requests. These requests are random, and can be malicious request made by byzantine clients. During this test, one server is also considered a byzantine server with the intention of proving that the system is capable of resisting attacks made by byzantine servers.

## 4 Durability Analysis

In the Dependable Chain (DepChain) system, the key dependability guarantees ensure that the blockchain remains both reliable and secure despite potential Byzantine faults.

The system provides safety, ensuring that only one value is agreed upon in each consensus round, even in the presence of faulty or malicious processes, thus maintaining the integrity of the blockchain. The system also guarantees liveness, ensuring that progress is made as long as the leader process behaves correctly, and consensus will eventually be reached. Through its Byzantine fault tolerance [1], DepChain can tolerate up to $f$ Byzantine processes (where $f$ is the maximum number of faulty processes), continuing to operate correctly as long as fewer than $2f + 1$ processes are faulty. Additionally, the system ensures message integrity by using authenticated perfect links, which authenticate and protect messages exchanged during consensus. Consistency is maintained as the blockchain records only the agreed-upon values, and availability is ensured as long as a quorum of correct processes is functioning. Lastly, security is reinforced through cryptographic signatures that authenticate messages, ensuring their integrity and preventing forgery.

## References

1. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to reliable and secure distributed programming. Springer Science & Business Media (2011)