DepChain – A Dependable Blockchain System

Simão de Melo Rocha Frias Sanguinho[1[102082]], José Augusto Alves Pereira[1[103252]], and Guilherme Silvério de Carvalho Romeiro Leitão[1[99951]]

Instituto Superior Técnico, Lisboa, Portugal

Abstract. This report presents *DepChain*, a dependable permissioned blockchain system, focusing on the final stage of its development. Building on the core consensus and blockchain service established in *Stage 1*, the system is extended with a transaction processing layer that supports smart contract execution (implemented in Solidity), native cryptocurrency transfers, and improved Byzantine fault tolerance (BFT). We present an in-depth analysis of the design choices, implementation details, potential attack vectors and countermeasures implemented to secure the system.

Keywords: blockchain · byzantine fault tolerance · consensus · dependability · java · smart contract

1 Introduction

This report presents the design and implementation of a Byzantine Fault Tolerant (BFT) blockchain service capable of maintaining correct operation despite malicious behavior from a subset of participants and unstable network conditions. Developed in two phases, the system builds upon the core consensus and blockchain mechanisms established in the initial phase by adding a comprehensive transaction processing layer. The extended architecture introduces external owned accounts, smart contract accounts, native cryptocurrency and token transfers, and enhanced fault tolerance capabilities.

The report details the system architecture organized into five layers: account, network, client, blockchain, and consensus. It examines the system's defenses against Byzantine attacks and proves its ability to maintain safety and liveness even when operating under a non-faulty leader.

2 Architecture

2.1 Network Layer

The network layer is responsible for managing communication between two processes (nodes or clients) in the system. By replicating authenticated perfect links, the network layer ensures that messages are guaranteed to eventually be delivered to the intended recipient, with its integrity and authenticity preserved. There are three main components in the network layer:

- **Message:** The `Message` class encapsulates all communication content and metadata within the network. Its design centers around a `Type` field (e.g., `READ`, `STATE`, `ACK`) to identify the purpose of the message when executing consensus instances.
  To support diverse transaction types (each requiring distinct execution logic) the message includes a `RequestType` field. This field specifies the operation to perform. The reply will hold the response data, which may be a full `Block` (for transaction that are appended on the blockchain) or a simple value read from the blockchain.
  Security is enforced through a `signature` computed over the message's critical components, which includes a `nonce` to prevent replay-attacks.
- **PerfectLink:** Implements the core communication logics, including sending, receiving and managing sessions. To simulate the unreliable network, UDP sockets are used.
- **Session:** Represents a communication session between two processes. The session contains information like the destination process ID, address, session key, and counters for tracking sent and acknowledged messages. The session key is a symmetric key that is used for signing messages (because using the private key for signing every message would be too expensive). Replay attacks are prevented by using `nonces` (counters) that are sequentially incremented for each message sent within that session.

Session Establishment Before any communication can occur, a session must be established between two processes. The session establishment process is as follows: a process initiates a session by sending a START_SESSION message. Then, the recipient responds with an ACK_SESSION message, containing a symmetric session key that is encrypted using the recipient's public key (thus ensuring confidentiality). Once the session is established, all subsequent messages are signed and verified using the session key to ensure authenticity and integrity.

2.2 Client and Library Layer

A user will issue one of the available commands via the client CLI. The client will then delegate the request to the library layer. Then, the library will construct a CLIENT_REQUEST message sign it with its private key to ensure non-repudiation and send it to every process so that each of them may check it independently. Next, the library will wait for 2f+1 replies, and check if there are at least f+1 equal replies, in order to avoid interferences from byzantine processes.

There are several commands that are available to the client:

- `help`: Displays the list of available commands.
- `addBlackList <targetAddress>`: Adds the specified address to the blacklist.
- `transferFrom <senderAddress> <targetAddress> <amount>`: Transfers the specified amount from the sender address to the target address.

- `getISTBal <targetAddress>`: Retrieves the ISTCoin balance of the specified address.
- `isBlackListed <targetAddress>`: Checks if the specified address is blacklisted.
- `approve <targetAddress> <amount>`: The command executor approves the specified address to spend a certain amount on their behalf.
- `transferDep <targetAddress> <amount>`: Transfers the specified amount of DepCoin to the specified address.
- `getDepBal <targetAddress>`: Retrieves the DepCoin balance of the specified address.
- `allowance <sourceAddress> <spenderAddress>`: Checks the remaining allowance for a spender from a source account.
- `removeBlackList <targetAddress>`: Removes the specified address from the blacklist.
- `transferIST <targetAddress> <amount>`: Transfers the specified amount of ISTCoin to the targetAddress.

There are two types of requests: `Block` and `Value`. The `Block` request is used to alter the blockchain, while the `Value` request is used to retrieve information from the blockchain.

The client replies will be different for those two types of requests, as in for the `Block` request, the client will receive a `Block` object that was appended on the blockchain resulting from the requested modifications (that could have been CONFIRMED or REJECTED), while for the `Value` requests, the client will receive a value that contains the requested information.

It is important to note that each block is mandatorily composed of 2 transactions (although this value can be changed in the `.env` file). This way, the leader awaits for 2 transactions to be received before starting the consensus protocol.

2.3 Blockchain Layer

It is assumed that the blockchain is permissioned (closed membership). Each member is responsible for participating in the consensus protocol, maintaining a local copy of the blockchain as a list of blocks, that are being persisted as a JSON file as they are being appended. The system is designed to ensure that all members agree on the state of the blockchain, even in the presence of faulty nodes, by leveraging a Byzantine fault-tolerant consensus protocol (Byzantine Read/Write Epoch Consensus).

For consensus-related messages, the member delegates the message to the current consensus instance for processing.

2.4 Consensus Layer

The consensus mechanism is designed to ensure that all non-faulty members of the distributed system agree on the value to be appended to the blockchain, even in the presence of Byzantine faults. The consensus protocol is implemented in the `ConsensusInstance` class and involves multiple phases:

- **READ Phase:** The consensus begins with the leader broadcasting a READ message to all members, prompting each to reply with a STATE message that includes its current epoch state, recent write, and writeset.
- **COLLECTED Phase:** After receiving a quorum of STATE messages, the leader sends a COLLECTED message containing these states so that each member can determine the most recent value, selecting the candidate from the writeset that appears in more than f members, or defaulting to the leader's latest write if none exists.
- **WRITE Phase:** Following this, every process broadcasts a WRITE message with the chosen value and waits for a quorum of matching WRITE messages before proceeding; failing to achieve this within a timeout results in aborting the instance.
- **ACCEPT Phase:** Next, an ACCEPT message is broadcast by each process, and upon receiving a quorum of consistent ACCEPT messages, the value is epoch-decided and committed to the blockchain.

The protocol is robust against up to f Byzantine faults, with a quorum defined as $2f + 1$ members, ensuring that non-faulty members ultimately agree on the same value despite potential malicious behavior or failures of up to f byzantine members.

3 Implementation Details

The system relies on a Public Key Infrastructure (PKI) to ensure the identity of members. Due to the performance limitations of public key cryptography, the system derives a shared symmetric key for each session, which is used to sign the messages. However, since it's our goal to provide non-repudiation to the transactions requested by the clients, the client will sign the message with its private key, and each node will verify the signature with the public key of the client when processing the transaction.

The client broadcasts the request to all members, allowing for each member to independently check that the final decided block consists of valid client-made transactions. This way, when the leader is byzantine, the other members will be able to detect its misbehavior, aborting the consensus in that situation.

3.1 Smart Contract and Blockchain Enhancements

The blockchain system implements Externally Owned Accounts (EOAs) for native DepCoin transfers and Smart Contract Accounts for ISTCoin token operations. The ISTCoin contract, implementing the ERC-20 standard, is deployed through the genesis block configuration which defines the initial state of all accounts. This configuration specifies both the initial token allocations and the contract's deployment bytecode along with its owner address. An excerpt of this genesis file is as follows:

```
{
  "block_hash" : "gs912893421bdajGDSTDAS2108EGDSABkDJHFGDSA",
  "previous_block_hash" : null,
  "transactions" : [ ],
  "state" : {
      "35bac2533f3d72f58b678e4dbb59c112" : {
          "balance" : 100000
      },
      "1259fb40b29be1d77a8031d5ee843d28" : {
          "balance" : 200000
      },
      "d98dde5b265c8a68234ffc3460df86bd" : {
          "balance" : 300000
      },
      "977398615948d60fd5717800b6bd2f5e" : {
          "balance" : 50000000,
          "code" :  "60806040523480...",
          "storage" : {
              "owner" : "35bac2533f3d72f58b678e4dbb59c112"
          }
      }
  }
}
```


3.2 Transaction Execution

The system processes transactions through distinct pathways based on their type. Native DepCoin transfers undergo direct validation and state updates, while ISTCoin operations require EVM execution. The execution flow for smart contract operations involves:

- 1. Transaction validation including signature verification
- 2. EVM bytecode preparation based on the requested operation
- 3. Execution through Hyperledger Besu's EVM

3.3 Smart Contract: ISTCoin and AccessControl

The ISTCoin smart contract implements a comprehensive token management system combining ERC-20 functionality with robust access control mechanisms. The contract's architecture revolves around two principal components:

- **Token Operations** The contract implements standard ERC-20 token transfers with built-in balance verification, ensuring transactions fail when senders lack sufficient funds. It supports delegated transfers through an allowance system, where token owners can authorize third parties to spend specific amounts on their behalf. The contract also includes all required view functions (balanceOf, totalSupply, etc.) that provide transparent access to token information without modifying the state.

– **Access Control** A permission system restricts operations to unauthorized accounts. The contract owner maintains exclusive rights to modify the blacklist - a security list that prevents specified accounts from sending or receiving tokens. Every token transfer automatically checks against this blacklist, rejecting any transactions involving prohibited addresses.

4 Testing and Validation

A comprehensive suite of testing configurations was developed to validate the overall system, leveraging several behvaviors. Below we include the said behaviors and how our system handles them:

– **Correct**: The system behaves correctly and with no byzantine processes.
– **Byzantine Leader**: A leader process is byzantine and favors a specific client by changing every DepCoin transfer to be from other clients to himself. This is prevented since the original transfer is sent to every node, giving them the ability to check if the transaction is valid after the block is determined from the collected phase.
– **Byzantine State**: A node process sends a state message with a higher epoch than the one it has in order to try to favor its own state. This is inherently prevented by the consensus protocol, since the determined block has to appear in more than f members' writesets in order to be chosen (it not, then the leader's latest write is selected).
– **Byzantine Client**: A client sends a burst of messages to attempt a DoS effect on the blockchain's nodes. Instead of just crashing and halting the operation, the system orderly processes every message as it arrives.
– **Ignore Messages**: A node ignores messages from other processes. This is handled by the consensus protocol, which requires a quorum of $2f + 1$ members to reach a decision, instead of the full $3f + 1$ members.
– **Impersonate**: A node impersonates another node by sending messages with the other node's ID. This is prevented by the use of symmetric keys unique to each session, which are used to produce the MAC of the messages, ensuring the authenticity of the sender.
– **Spam**: A node sends a large number of consensus messages to try to disrupt the consensus. Processes avoid this by associating messages with the node that sent them, thus making sure that a process only participates once in any given consensus phase.

These tests verify the dependability guarantees and ensure that the system meets its design goals under both normal and adversarial conditions.

Furthermore, there is also a JUnit test simulating the interaction with the system's Smart Contract through a generic Smart Account in order to test its functionalities (token transfers and blacklist manipulations).

5 Discussion and Future Work

The integration of smart contract execution and native cryptocurrency transfers has expanded the DepChain system from a simple blockchain to a full-fledged platform capable of handling sophisticated transactions and maintaining a robust audit trail. The design decisions, such as leveraging Hyperledger Besu's EVM for smart contract execution and using a rigorous genesis block format, significantly enhances system dependability. Future work could involve:

- Extending the consensus algorithm to support dynamic membership.
- Implement leader change at consensus abort.
- Optimizing transaction throughput and state synchronization.

6 Conclusion

This report presents a detailed overview of DepChain, covering both the core functionalities and the advanced features integrated into the system. The system now supports secure DepCoin transfers, smart contract execution, and robust countermeasures against Byzantine attacks, meeting the high dependability and security requirements defined in the project statement.

Through the implementation of this system, we have consolidated our understanding of blockchain technology, consensus algorithms, and smart contract execution. The project has provided valuable insights into the complexities of building a dependable distributed system and the challenges associated with ensuring security and reliability in the face of potential adversarial behavior.

References

1. Hyperledger Besu Ethereum Client. `https://github.com/hyperledger/besu/tree/main`.
2. ERC-20 Token Standard. `https://ethereum.org/en/developers/docs/standards/tokens/erc-20`.
3. OpenZeppelin - ERC20 Implementation. `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20`.
4. Springer LNCS Guidelines. `https://www.springer.com/gp/computer-science/lncs/conference-proceedings-guidelines`.