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 details the design, implementation and evaluation of DepChain, a simplified permissioned blockchain system with high dependability guarantees. In stage 1, the core consensus and "blockchain" persistence mechanisms were developed. Stage 2 expands the system by integrating smart contract execution, native cryptocurrency transfers and enhanced Byzantine fault tolerance. We present an in-depth analysis of the design choices, implementation details (including Solidity smart contracts and Java-based blockchain components), 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, designed to withstand malicious behavior from a subset of blockchain members and operate reliably in an unstable network environment. The project was developed in two phases, where the first phase established the core consensus, blockchain persistence mechanisms, and account models, and the second phase refined the application semantics by integrating smart contract execution and native cryptocurrency transfers, along with enhanced fault tolerance. The report outlines the system architecture, which includes the network, client, blockchain, and consensus layers, and describes how the system addresses various Byzantine attack scenarios, ensuring safety and liveness under the assumption of a correct leader.

2 Blockchain Nodes and Consensus

2.1 Network Layer

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

- **Message:** Encapsulates the message content and metadata. Among other things, it contains the type field to identify the message type (e.g., READ, STATE, ACK, etc.). A MAC will be used to ensure the message's aforementioned security properties.
- **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).

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 - TODO: Change this to fit with the new interface

The system's regular workflow is as follows: a user will issue the append `<message>` command via the client's CLI. The client layer will delegate the request to the library layer. Then, the library will construct a CLIENT_REQUEST message and send it to the leader process (known beforehand) using PerfectLink. Next, the library will wait for a CLIENT_REPLY. Since a non-Byzantine leader is assumed, the client does not need to wait for a quorum of replies.

2.3 Blockchain Layer

The blockchain layer represents a node in the permissioned (closed membership) blockchain network. Each member is responsible for participating in the consensus protocol, maintaining a local copy of the blockchain as a list of strings. 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). It also keeps a State object, which stores a sequence of TimestampValuePair objects that represent the history of that member on a particular instance of consensus. The blockchain is updated whenever a consensus decision is reached, ensuring that all non-faulty members have a consistent view of the blockchain.

A message handler loop continuously listens for incoming messages. If a CLIENT_REQUEST is received by the leader, it initiates a new consensus

instance to process the request. 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:

Consensus Instance Each consensus instance keeps track of the current epoch number. The consensus instance is initiated by the leader (process 1) when it receives a CLIENT_REQUEST from a client. The leader coordinates the consensus process, while other members participate by responding to messages and contributing their local states.

Read Phase The consensus process begins with the read phase, where the leader broadcasts a READ message to all members. Each member responds with a STATE message, which contains its local epoch state, including the most recent write and the writeset (a list of all writes).

Collected Phase Once the leader has received STATE messages from a quorum of members, it broadcasts a COLLECTED message to all members. This message contains the collected states from the quorum's members, allowing each member to independently determine the value to be written. Members use the collected states to select the most recent value that appears in the writeset of more than f members (where f is the maximum number of Byzantine faults tolerated). If no such value exists, the value is unbounded. In this case, the leader's most recent write is selected as the candidate value.

Write Phase Every process (members and leader) broadcasts a WRITE message to all members containing the previously determined value. It then waits for a quorum of WRITE messages with the same value proposed. If this condition is not met or if a predefined timeout is reached, then the process aborts the consensus instance; otherwise, it moves on to the next phase.

Accept Phase Every process then broadcasts an ACCEPT message to all members containing the value that was previously agreed upon. It waits for a quorum of ACCEPT messages with the same value back. If this condition is not met or if a predefined timeout is reached, then the process aborts the consensus instance; otherwise, it epoch-decides the value and commits it to the blockchain.

Fault Tolerance and Quorum The consensus protocol is designed to tolerate up to f Byzantine faults, where f is the maximum number of faulty members allowed in the network. A quorum is defined as the minimum number of members required to reach agreement, calculated as $2f + 1$. The protocol ensures that all non-faulty members agree on the same value, even if f members behave maliciously, fail to respond, or crash.

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. Since confidentiality is not a requirement, the system does not encrypt the messages.

The client only communicates with the leader process to simplify the system, a decision that is justified by the assumption that the leader is correct and will not deviate from the protocol. However, the system is designed to handle Byzantine behavior, and if the leader behaves maliciously and deviates from the protocol, the other members are able to detect the misbehavior and abort the consensus instance.

3.1 Smart Contract and Blockchain Enhancements

To extend the capabilities of the basic blockchain, smart contract execution and native cryptocurrency transfers have been integrated. The smart contract `ISTCoin.sol` implements the ERC-20 standard, allowing for token transfers and invoking an access control contract to validate transactions. The contract is compiled to EVM bytecode and is deployed as part of the genesis block, where the deployed bytecode and storage data are stored in a JSON file defining the initial state. An excerpt of this genesis file is as follows:

```
{
  "block_hash": "gs912893421bdajGDSTDAS2108EGDSABkDJHFGDSA",
  "previous_block_hash": null,
  "transactions": [],
  "state": {
    "35bac2533f3d72f58b678e4dbb59c11272e7b75c94dabf0a24d1827890ac69af": {
      "balance": 100000
    },
    "1259fb40b29be1d77a8031d5ee843d281c3f105cfaa2dec04db3711e279825e0": {
      "balance": 200000
    },
    "d98dde5b265c8a68234ffc3460df86bdf411f988a3b9e8c9e8d9b2c01d244376": {
      "balance": 300000
    },
```

```
    "986f0f856dc8a43529c8f5a2145ee619c97ed813dc90f2e7a04c099794d9f243": {
      "balance": 50000000,
      "code": "6080604052348015610000f575f80fd5b50...",
      "storage": {
        "owner": 5,
        "key2": "value2"
      }
    }
  }
}
```

This file initializes externally owned accounts (EOAs) with preset balances and deploys the smart contracts, ensuring that the system starts in a secure, verifiable state.

In addition, the Java implementation has been extended to support these functionalities:

- **EOAccount.java:** Models an externally owned account with methods to query and modify balances, ensuring non-negative balances and encapsulated state modifications.
- **SmartAccount.java:** Handles contract accounts, encapsulating smart contract-specific features such as code storage and owner metadata. An inner `Storage` class manages key-value pairs, which is critical for maintaining contract state.
- **Blockchain.java:** Reads the genesis file and subsequent block files from the specified directory, constructs EOAccounts based on public keys and state information, initializes the smart contract account by extracting the bytecode and storage, and implements methods to create and append new blocks based on incoming transactions.
- **AccessControl.java, Block.java, and BlockState.java:** Support block construction and execution. `AccessControl.java` encapsulates the balance and code attributes used in contract-based access control, while `Block.java` and `BlockState.java` ensure that state transitions resulting from transaction execution are clearly represented.

3.2 Transaction Execution and Byzantine Tolerance

Transactions in DepChain include both native DepCoin transfers and smart contract invocations. The execution process involves:

1. Verifying that the initiating account is authorized (via signature verification and access control checks).
2. Executing smart contract code using Hyperledger Besu's EVM environment.
3. Updating the state in a new block while preserving the non-negativity invariant of account balances.

To address Byzantine behavior, several countermeasures have been implemented:

- **Key Ownership and Non-repudiation:** By relying on public/private key cryptography, the system ensures that only the rightful owner of an account can authorize transactions.
- **Access Control in Smart Contracts:** The access control smart contract restricts modifications to the blacklist, preventing unauthorized transfers.
- **State Integrity:** The genesis file and subsequent block persistence provide a verifiable audit trail of all transactions and state transitions.
- **Detection of Malicious Behavior:** Through extensive testing, the system is shown to withstand various Byzantine attacks, including collusion between clients and servers.

4 Testing and Validation - TODO: Add the behaviors we have integrated with this new functionality

A comprehensive suite of JUnit tests was developed to validate the overall system. The test cases simulate a range of scenarios that include:

- Correct initialization of the blockchain state from the genesis file.
- Accurate execution of DepCoin transfers and smart contract invocations.
- Integrity of the blockchain when faced with Byzantine inputs, such as unauthorized transactions or colluding parties.
- Persistence and recovery of blockchain state across multiple blocks.

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

5 Discussion and Future Work

The integration of enhanced 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 enhance system dependability. Future work could involve:

- Extending the consensus algorithm to support dynamic membership.
- Further refining Byzantine detection mechanisms.
- 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 set forth in the project statement.

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.