DepChain: A Dependable Permissioned Blockchain System

Henrique Carrão 100313, João Maçãs 99970, and Duarte São José 103708

Instituto Superior Técnico, Lisbon, Portugal

Abstract. This project presents DepChain, a permissioned blockchain system designed with high dependability guarantees. In the first stage, we implemented the consensus layer using the Byzantine Read/Write Epoch Consensus algorithm. The system assumes a static membership model with a pre-designated leader and relies on a pre-established Public Key Infrastructure. For the second stage, we enhanced the system to support cryptocurrency transfers and smart contract execution with Byzantine fault tolerance against both malicious servers and clients. We implemented an account model based on Ethereum, including support for Externally Owned Accounts and Contract Accounts, along with ERC-20 token and blacklist-based access control smart contracts. The system integrates cryptographic mechanisms for comprehensive security guarantees.

1 Introduction

Our implementation focuses on the Byzantine Read/Write Epoch Consensus algorithm, providing a robust foundation for agreement among blockchain participants. The system operates with a static membership model and relies on a pre-established Public Key Infrastructure (PKI) for node authentication. We've built secure communication channels through authenticated perfect links, ensuring reliable message delivery despite network failures or adversarial behavior. with the blockchain we incorporate transaction processing capabilities, including native cryptocurrency transfers (DepCoin) and smart contract execution. We implemented Ethereum-compatible account types, supporting both Externally Owned Accounts (EOAs) controlled by private keys and Contract Accounts that host smart contract code. The system also includes ERC-20 token functionality and access control mechanisms to improve security.

2 System Design

2.1 Architecture Overview

Our implementation has two main actors, `Client` and `Server`. The `Client` library can interact with the blockchain to check balances, transfer cryptocurrency or execute smart contracts. The `Server` represents the server nodes on the blockchain environment. It maintains a `ClientAplManager` for handling client communication and a `ServerAplManager` to communicate with other `Node`s. Each `Node` runs a consensus thread deciding on blocks to append to the blockchain and appending them, followed by a thread that executes the transactions from the decided blocks. Transactions that only read state are not persisted in blocks in the blockchain, so they are executed immediately without needing consensus.

2.2 Network Layer - Authenticated Perfect Links

For the network layer, we implemented UDP sockets with a resending mechanism to approximate fair loss links. We built higher-level abstractions that enforce Stubborn Link and Authenticated Perfect Link (APL) guarantees.

The APL guarantees include:

- **No Creation**: No message is delivered unless it was sent.
- **No Duplication**: No message is delivered more than once.
- **Reliable Delivery**: If a sender continuously retransmits a message, it will eventually be delivered.
- **Authenticity**: Delivered messages are verified to have been sent by their claimed sender.

We introduced an abstraction layer called `AplManager`, which provides a peer-to-peer interface over a single underlying socket, improving scalability while keeping the interfaace simple. Each `Apl` within the `AplManager` communicates with a counterpart on a different server. During their initial exchange, `Apl`s securely share a secret key, which is then used with HMAC to ensure both integrity and authenticity, as the key is known only to the communicating parties.

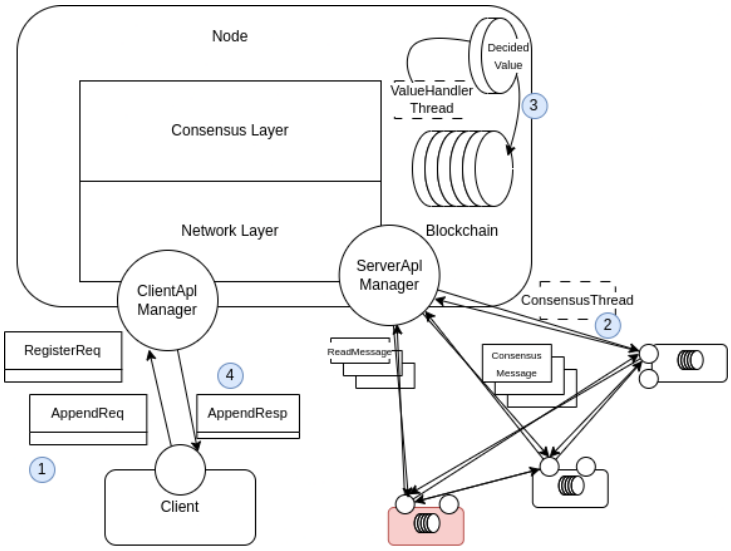

Fig. 1. Client append request process on DepChain

2.3 Consensus Layer - BFT Read/Write Epoch

Algorithm Overview The consensus layer implements a Byzantine Fault Tolerant (BFT) Read/Write Epoch Consensus algorithm. This algorithm operates in epochs, with each epoch having a designated leader. The consensus process consists of two main phases, Read and Write phases. The algorithm is designed to tolerate up to f Byzantine failures in a system with $3f + 1$ total nodes.

Epoch Change Mechanism Our implementation includes an epoch change mechanism that allows the system to progress when the current epoch cannot reach consensus. The `EpochConsensus` class manages this process by:

- Detecting when an epoch should be aborted (using abort counts)
- Broadcasting abort messages to all nodes
- Incrementing the epoch number and resetting relevant state
- Starting a new epoch with a new leader (determined by `epochNumber % NUM_MEMBERS`)
- When a node complains, it stays in a state where it waits for $2f + 1$ equal (write, acccept, abort) messages to proceed

This mechanism ensures liveness by allowing the system to recover from scenarios where consensus cannot be reached in the current epoch. The implementation carefully preserves safety properties during epoch transitions by maintaining consensus state information across epochs.

Implementation Details and Design Decisions Our implementation of the Byzantine Fault Tolerant (BFT) Read/Write Epoch Consensus algorithm is structured to ensure reliability, fault tolerance, and efficiency. It is implemented through several key classes:

- `BFTConsensus`: Orchestrates the overall consensus process
- `EpochConsensus`: Manages the consensus logic for a single epoch
- `ConsensusState`: Maintains the node's view of consensus state
- `EventCounter`: Tracks message events (writes, accepts, aborts) from different nodes

The `EpochConsensus` class implements the core algorithm logic, including:

- Leader election based on epoch number
- Collection and verification of state information from nodes
- Determination of the value to propose based on collected states
- Voting mechanism for reaching agreement on the proposed value
- Abort handling and epoch change procedures

Each node waits for a quorum of responses (at least $2f + 1$ out of $3f + 1$) before proceeding to the next step, ensuring safety despite Byzantine failures. Messages are cryptographically authenticated to prevent forgery and impersonation

Epoch Consensus Management The consensus process operates in a loop, represented in the class `BFTConsensus`, with each iteration representing an epoch. In this loop:

- the main thread waits to be awake either by new consensus message from another member or by new block ready to be added, built from a batch o transactions from clients (both situations mean that a consensus must occur)
- if it doesn't have any block to be added it joins consensus with a null value, but in each epoch it tries to gather a block, to make sure eventually a node will propose one (this is not really a problem because, the consensus was started by someone, but it can be a Byzantine node, so this solves this case)
- after the consensus has been reached and a block is decided, it removes the transactions present in the block decided from the received transactions queue, push each one to the transactions queue to be executed and add the block to the blockchain

`NOTE:` All the transactions already decided are stored in a `Hashset` so we know if a late arrival transaction was already decided.

Each epoch follows a structured approach, incorporating:

- Leader election using the formula `epochNumber % NUM_MEMBERS`, ensuring a round-robin selection.
- State collection and validation from participating nodes before proposing a value. (This is only done once per epoch, an optimization that avoids the read phase if we maintain the leader from the previous consensus)
- Deciding and verifying the value to write
- Write and accept phases, where nodes vote and finalize the consensus decision.
- Abort and epoch change mechanisms to recover from unreachable consensus states in the presence of byzantine processes.

The `EpochConsensus` class encapsulates the logic for a single epoch and manages communication, validation, and state transitions. When an epoch starts, if it receives an `AbortedSignal` exception, it will wait to detect more `2*ALLOWED_FAILURES` messages of an equal event for the same value (accept, write or abort), before proceeding to a new epoch. A epoch is a "infinit" loop, that only stops when a value is decided or `2*ALLOWED_FAILURES+1` processes agree on changing epoch. More in detail, this loop:

- defines two paths:
  1. `LEADER PATH:` starts the read phase if it is needed, broadcasting `READ` and receiving `STATE` messages. After this, creates the `CollectedStates` structure, verifying if every state received is well signed (if not, it is not added). Then, `COLLECTED` messages are broadcasted.
  2. `NON LEADER PATH:` receives the `READ` messages from the leader and sends it the `STATE` message, after signing the state. After this, it receives the `COLLECTED` message, and proceeds to verify the signature for each state.
- the two paths converge. From the collected states, a value to be written is decided and verified (signature verification of each state and each transaction).
- the `WRITE` messages are broadcasted and it waits until it receives a quorum of writes for the same value. In case this doesn't happen, it broadcasts aborts and throws an `AbortedSignal`
- after waiting, it writes the value and timestamp in its state as the most recent quorum written value, and broadcast the `ACCEPT` messages
- finally, it waits until it receives a quorum of accepts for the same value and returns the value itself. In case this doesn't happen, it broadcasts aborts and throws an `AbortedSignal`

`NOTES:`

- We have made an optimization, where we store the epoch number between different consensus so the last leader of the last consensus is the starter leader of the next one.
- During early phases, when receiving messages, we always verify if the member also receives writes, accepts or newepochs, and count them, so that if a quorum of them is received, we jump to the respective phase.
- We ignore all messages from the past or some unsynchronized ones.
- We have two queue of messages for each pair of members, where the exchange messages between them are placed.

Message Handling and Processing The system relies on message-based coordination between nodes. Key message types include: `ReadMessage`, `WriteMessage`, `AcceptMessage`, `NewEpochMessage`

Message passing Managed by the `ConsensusMessageHandler` class, ensuring correct deserialization, storage, and processing of messages received from other nodes.

State Tracking and Verification Each server maintains a `ConsensusState` object, which holds:

- The most recent quorum-written value (i.e., the value confirmed by a majority of nodes).
- A write set tracking values proposed in the past.
- A cryptographic signature ensuring integrity and authenticity.

Before deciding over the collected states, nodes verify state signatures using the pre-established Public Key Infrastructure (PKI) and transaction signatures using the clients' public keys shared when they register in the service. This prevents Byzantine nodes from injecting false states and transactions.

Failure Recovery and Epoch Changes To handle situations where consensus cannot be reached in an epoch, an abort mechanism is implemented:

- Servers track abort counts using an `EventCounter`.
- If aborts exceed a threshold (`ALLOWED_FAILURES`), a `NewEpochMessage` is broadcast.
- Servers increment their epoch counter and attempt a new round of consensus when another threshold is met (`2*ALLOWED_FAILURES`)

This ensures liveness by preventing nodes from remaining in a state where consensus cannot be reached.

Security Enhancements Security is reinforced through cryptographic measures, including:

- Digital signatures: We chose to implement the `Signed Conditional Collect`. So we have digital signatures on some consensus messages to prevent tampering of the states of each member.

2.4 Account Model and Block Structure

`DepChain`'s account model follows `Ethereum`'s approach with two types of accounts:

- **Externally Owned Accounts (EOAs)**: Controlled by private keys, these accounts can initiate transactions and hold `DepCoin` balances.
- **Contract Accounts**: Also contain smart contract code and storage, activated when receiving transactions.

Each account is identified by a unique address and contains:

- **Balance**: The amount of `DepCoin` owned by the account
- **Nonce**: A counter used to prevent replay attacks
- For Contract Accounts: **Code** (EVM bytecode) and **Storage** (key-value pairs)

Each block in `DepChain` consists of:

- **Block header**: Contains metadata including block hash and previous block hash.
- **Transaction list**: Contains all transactions included in the block.
- **State**: The world state after executing all transactions within the block. (this is only present in the persistent block)

The `Genesis Block`, stored in JSON format, contains the initial state of the system, including predefined accounts and pre-deployed `Smart Contract`s.

2.5 Smart Contract Implementation

We implemented two interconnected smart contracts deployed in the Genesis Block:

ERC-20 Token Contract The `ISTCoin` contract extends OpenZeppelin's ERC-20 implementation with the following characteristics:

- Name: IST Coin
- Symbol: IST
- Decimals: 2 (custom implementation)
- Total supply: 100 million tokens minted to the deployer
- Exchange rate: 230,000 IST per ETH

The contract overrides standard transfer functions to integrate blacklist protection and includes a `buy()` function allowing users to purchase tokens directly with DepCoin.

Blacklist Security Contract The `Blacklist` contract extends OpenZeppelin's Ownable contract to provide:

- Owner-controlled address blacklisting functionality
- `addToBlacklist(address)`: Adds accounts to the blacklist
- `removeFromBlacklist(address)`: Removes accounts from the blacklist
- `isBlacklisted(address)`: Verification method used before transfers

The blacklist integration prevents any transactions of IST Coin involving blacklisted addresses, including transfers, approvals, and token purchases, enhancing security and regulatory compliance.

2.6 EVM Integration

We integrated Hyperledger Besu's Ethereum Virtual Machine (EVM) implementation to support smart contract execution by:

- Setting up the execution environment with appropriate context
- Translating between DepChain's internal state representation and EVM's format
- Capturing and processing contract execution results
- Updating the world state based on execution outcomes

The EVM executes the bytecode of smart contracts, supporting both view functions and state-modifying functions.

3 Threat Model and Protection

3.1 Threat Actors

The main threat actors include:

- **Byzantine blockchain members**: Nodes that may deviate from the protocol
- **Malicious clients**: External users who may attempt to exploit the system
- **Network attackers**: Entities that can intercept, modify, or inject messages

3.2 Security Measures

Network Layer Security At the network layer, we implemented a secure communication protocol built on top of UDP with the following security features:

- **Session Key Establishment**: When an Authenticated Perfect Link (APL) initiates communication with a new node, it establishes a secure session by generating a symmetric key, encrypting it with the recipient's public key (leveraging the pre-established PKI), and sending it to the destination node.
- **Message Integrity and Authentication**: All messages transmitted through an established link include an HMAC generated using the session key. This guarantees that messages cannot be tampered with during transmission without detection.
- **Freshness Guarantee**: Each message includes a sequence number that protects against replay attacks by ensuring that messages are processed in the correct order.
- **Receiving Mechanism**: Upon receiving a message, the APL verifies the HMAC and sequence number before delivering the message to the message handler. Then it sends an acknowledgment to confirm the successful receipt and verification.

Consensus Layer Security The consensus layer builds upon the secure network layer and adds additional security measures to ensure correct behavior of the blockchain system:

- **Client Registration**: Before submitting blockchain operations, clients must register with the system by sending a registration request containing their public key. This establishes a trust relationship between clients and the blockchain service.
- **Client Non-repudiation**: After registration, all client requests include a digital signature created with the client's private key. This provides non-repudiation, allowing blockchain members to verify the authenticity of client requests and preventing Byzantine nodes from forging requests.
- **Timestamp-based Freshness**: All application layer requests include timestamps to prevent replay attacks, ensuring that old requests cannot be resubmitted to the system.
- **Byzantine Fault Detection**: The system implements verification mechanisms to detect and isolate Byzantine behavior, such as checking digital signatures and comparing collected values during the consensus protocol.

Transaction Security For transaction processing, we added:

- **Account Ownership Verification**: All transactions require a valid signature from the account owner.
- **Balance Validation**: The system enforces non-negative balances.
- **Transaction Nonces**: Each account maintains a nonce to prevent replay attacks.
- **Smart Contract Security**: Smart contracts execute in a sandboxed environment
- **Access Control**: The blacklist contract prevents potentially malicious accounts from performing operations.

4 Dependability Guarantees

4.1 Safety and Liveness

Safety Guarantees The safety properties ensure that the blockchain maintains consistency despite Byzantine faults:

- **Agreement**: If two correct processes decide on values in the same epoch, they decide on the same value.
- **Validity**: If all correct processes propose the same value v, then no value different from v can be decided.
- **Integrity**: Decided values are tracked to prevent duplicate processing.
- **Authenticity**: Cryptographic signatures prevent tampering.

These properties hold with up to f Byzantine nodes in a system of $N = 3f$ nodes.

Liveness Guarantees DepChain's liveness properties ensure the blockchain progresses despite faults:

- **Termination**: Every correct process eventually decides some value through our epoch change mechanism.
- **Non-blocking progress**: The system continues processing with up to f failed nodes.

Our liveness guarantees rely on the eventual synchrony assumption, meaning the system will progress as long as message delays eventually fall below an unknown but finite bound.

4.2 Byzantine Client Tolerance

Our design tolerates Byzantine client behavior through:

- **Transaction validation**: All transactions undergo rigorous validation before inclusion in a block.
- **Smart contract sandboxing**: EVM execution occurs in a sandboxed environment.
- **State transition validation**: All blockchain state transitions are independently verified by every honest node.

4.3 Blacklist-based Access Control

Our blacklist mechanism provides:

- **Authority restrictions**: Only authorized entities can modify the blacklist.
- **Transaction filtering**: The ERC-20 implementation consults the blacklist before allowing transfers.
- **Consistency**: The blacklist state is subject to the same consensus rules as other blockchain state.

5 Transaction Processing

5.1 Transaction Types and Execution

DepChain supports two primary transaction types:

- **Native cryptocurrency related transactions**: Moving DepCoin between accounts and checking DepCoin balances.
- **Smart contract calls**: Executing functions on deployed smart contracts.

When a transaction is proposed, the following process occurs:

1. The transaction is validated (signature, nonce, balance).
2. The consensus layer determines transaction ordering by deciding on blocks of transactions.
3. Each transaction of the decided block is executed by the EVM in sequence.
4. State changes are applied atomically per transaction.
5. The new state is stored in the block.
6. The block's parameters are computed (blockHash, etc.) and the block is stored in the blockchain.

5.2 Client-Server Protocol

The communication between clients and the blockchain follows a well defined protocol:

Client Registration - When a client wishes to interact with the blockchain:

1. When the client is created, it generates a keypair.
2. When the client decides to interact with the blockchain, it sends a `RegisterReq` message containing its public key and identifier.
3. The blockchain service creates an associated account with an initial balance.
4. After registration, the client can send transaction requests.

Transaction Requests - For transactions (transfers, contract calls, etc.):

1. The client creates a request message containing the necessary details, a sequence number, and a digital signature.
2. The client sends this message to the servers.
3. Blockchain members verify the signature and sequence number, then propose the request for consensus.
4. Once consensus is reached, the transaction is executed and included in the blockchain.
5. The client receives a response message with the result.

6 Conclusion

DepChain demonstrates a highly dependable permissioned blockchain system operating correctly despite Byzantine faults. Our implementation successfully addresses the challenges of both project stages: a resilient consensus layer with authenticated perfect links and Byzantine fault tolerance, extended with cryptocurrency support, smart contract execution, and protection against Byzantine clients.

The layered architecture—from authenticated perfect links to consensus protocol, transaction processing, and smart contract execution—provides a solid foundation for dependable distributed applications. By requiring agreement from a quorum of nodes for all decisions, DepChain maintains consistency while tolerating up to f Byzantine nodes in a system of $3f + 1$ nodes.

The integration of EVM-compatible smart contracts, particularly the ERC-20 token and blacklist-based access control, demonstrates how blockchain applications can benefit from strong Byzantine fault tolerance.

DepChain serves as a practical example of how distributed systems principles can be applied to blockchain technologies, combining the benefits of distributed ledgers with strong dependability guarantees—valuable for enterprise applications where security and reliability are paramount.

References

1. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming. 2nd edn. Springer (2011)
2. Hyperledger Besu Documentation: EVM Implementation. Hyperledger Foundation (2023)
3. OpenZeppelin: ERC20 Implementation. OpenZeppelin Documentation (2023)
4. Ethereum Foundation: Ethereum Yellow Paper - Formal Specification of Ethereum. Ethereum Foundation (2023)