Highly Dependable Systems: DepChain

Luís Marques - IST1110859
Instituto Superior Técnico

1 Introduction

This project focuses on the development of a simplified permissioned blockchain system with high dependability guarantees, named *Dependable Chain* (DepChain). DepChain is designed to operate under a closed membership model, where all participants, both clients and nodes, are known in advance and possess predefined cryptographic identities. DepChain targets two fundamental properties:

- Reliable transaction processing between clients, including native cryptocurrency transfers and the execution of smart contracts.
- Resilience to Byzantine faults, including malicious or arbitrary behavior by clients and nodes.

The system employs a consensus protocol derived from the Byzantine Read/Write Epoch Consensus algorithm, tailored for environments with static membership and a correct leader assumption. While the leader is assumed to be non-faulty, other nodes may exhibit Byzantine behavior. The protocol ensures safety under all conditions and guarantees liveness when the leader behaves correctly.

DepChain's architecture consists of two external clients capable of issuing transactions and interacting with the blockchain, and four internal nodes that coordinate with each other to maintain consensus and process those client requests. Nodes communicate using authenticated perfect links and each node maintains a local copy of the blockchain state, which evolves over time based on consensus decisions.

Each node runs a full stack including consensus logic, transaction validation, state execution and block management. Clients, on the other hand, generate keys, construct transactions and submit them to the system for inclusion in the blockchain.

By combining a lightweight yet robust consensus protocol with native support for smart contracts and access control mechanisms, DepChain provides a modular and extensible foundation for dependable distributed applications.

2 Design and Implementation

2.1 Genesis Block and Node Initialization

The initialization process of *DepChain* involves two critical phases: the generation of the genesis block and the configuration of both clients and nodes participating in the system.

Genesis Block Construction. The genesis block defines the initial state of the blockchain. This includes pre-allocated balances to Externally Owned Accounts (EOAs), the deployment of smart contracts and the system's control account. These accounts are defined in a `genesis.json` file, which is parsed at startup. The file also specifies the contract bytecode (constructor code) to be deployed under a specific `CONTRACT` address.

The `GenesisBuilder` class initializes a custom EVM environment using the Hyperledger Besu execution engine. Accounts are added to a `SimpleWorld` instance, assigning them balances and, if needed, contract bytecode. A dedicated constructor execution phase extracts the contract's runtime bytecode using an EVM tracer and replaces the initialization code with the actual executable logic, emulating a deployment transaction.

Once the EVM world state is initialized, the genesis block is created by instantiating a `Block` object that includes synthetic transactions reflecting initial account balances (in DepCoin) and ABI-encoded calls to retrieve ISTCoin balances from the deployed ERC-20 contract.

Node and Client Configuration. The system is composed of four nodes and two clients. Each participant is configured through a dedicated JSON configuration file, containing metadata such as:

- Network ID and hostname/port information,
- Cryptographic key paths (public and private),
- Ethereum-style address used in the system,
- Behavioral mode (e.g., honest, Byzantine, delayed).

Nodes are responsible for running the consensus protocol, validating transactions and persisting the blockchain state. Clients act as transaction generators and send requests to the nodes.

A Python launcher script is responsible for orchestrating the system's startup. It validates the presence of configuration files, compiles the project via Maven, and then launches each node and client in separate terminal windows using a terminal emulator (e.g., Kitty). Nodes and clients receive their configuration as command-line arguments and use them to load keys, establish communication channels, and bind to their respective ports.

This modular initialization approach enables isolated testing, easy fault injection via configuration and dynamic changes in node behavior without modifying the core implementation.

2.2 Smart Contracts

DepChain includes a native ERC-20 compliant smart contract named `ISTCoin`, which serves as a fungible token with two decimal places and a total supply of 100 million units. The contract was implemented in Solidity and compiled using the Remix IDE, which generated both the deployment bytecode

and the runtime bytecode necessary for execution on the EVM. The Solidity source code of the smart contract is shown below:

```
pragma solidity ^0.8.20;

contract ISTCoin {
    string private _name = "IST␣Coin";
    string private _symbol = "IST";
    uint8 private _decimals = 2;
    uint256 private _totalSupply = 100
        _000_000 * 10**2;
    mapping(address => uint256) private
        _balances;

    constructor() {
        _balances[msg.sender] = _totalSupply
            ;
    }

    function name() public view returns (
        string memory) {
        return _name;
    }

    function symbol() public view returns (
        string memory) {
        return _symbol;
    }

    function decimals() public view returns
        (uint8) {
        return _decimals;
    }

    function totalSupply() public view
        returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address account)
        public view returns (uint256) {
        return _balances[account];
    }

    function transfer(address to, uint256
        value) public returns (bool) {
        require(to != address(0), "Invalid␣
            receiver␣address");
        require(_balances[msg.sender] >=
            value, "Insufficient␣balance");

        _balances[msg.sender] -= value;
        _balances[to] += value;
        return true;
    }
}
```

Listing 1. ISTCoin ERC-20 Smart Contract

Deployment Logic. Since the contract contains a constructor that mints the total token supply to the deployer's address, it was essential to distinguish between the deployment and runtime bytecode. First, the deployment bytecode was executed in the EVM environment, which returned the finalized runtime bytecode. This bytecode was then explicitly set as the contract's code in the blockchain state.

The owner and deployer of the contract is Client 1 (`0x101...01`), who receives the entire token supply upon deployment. The contract's state is stored under a predefined `CONTRACT` address in the genesis block and is accessible to all blockchain participants via EVM method invocations.

Runtime Integration. When executing transactions against the smart contract (e.g., token transfers or balance queries), the EVM executor loads the runtime bytecode using the `.code()` method on the contract account. This ensures that only the persistent, executable logic is interpreted, mimicking the behavior of an Ethereum full node.

Token Properties. The `ISTCoin` contract provides basic ERC-20-like functionality, including:

- Total supply: `100,000,000.00 IST`,
- Minting of tokens to the deployer only (during construction),
- Token transfer between accounts with balance and address validation,
- Public read access to metadata (name, symbol, decimals).

Further access control over token transfers is enforced by a separate smart contract (the Access Control List), described in the next subsection.

2.3 Transaction Execution

DepChain supports two main types of transactions: native cryptocurrency transfers (DepCoin) and smart contract method invocations (e.g., ERC-20 token transfers and balance queries). All transactions follow a unified internal structure and are executed using a custom EVM executor backed by Hyperledger Besu.

Encoding Smart Contract Calls. Smart contract interactions are ABI-encoded using the Web3J library. For instance, to query a token balance or transfer tokens, the corresponding method name and arguments are encoded into a byte array using the Ethereum ABI standard.

To encode the call to `balanceOf(address)`, the following snippet is used:

```
Function function = new Function(
    "balanceOf",
    Arrays.asList(new Address(address)),
    Collections.emptyList()
);
String inputData = FunctionEncoder.encode(
    function);
```

Listing 2. Encoding balanceOf(address) using Web3J

Similarly, to encode `transfer(address, uint256)`:

```
Function function = new Function(
    "transfer",
    Arrays.asList(new Address(to), new
        Uint256(value)),
    Collections.emptyList()
);
String inputData = FunctionEncoder.encode(
    function);
```

Listing 3. Encoding transfer(address, uint256)

The resulting `inputData` is passed to the EVM for execution.

Executing Transactions on the EVM.. Transactions are executed in the EVM through the following sequence:

```
executor.callData(Bytes.fromHexString(
    inputData));
executor.sender(Address.fromHexString(from))
    ;
executor.code(contractsBytecode.get(coinType
    ));
executor.execute();
```

Listing 4. Executing smart contract transaction

In this process:

- `callData` contains the ABI-encoded method invocation.
- `sender` specifies the originator of the transaction.
- code refers to the runtime bytecode of the target smart contract.
- `execute()` runs the EVM and applies state changes.

Integration with Consensus and State. Each transaction undergoes signature and authorization checks before being accepted into a block. Once validated, transactions are grouped and processed in the order decided by the consensus protocol. After successful execution, the resulting changes (account balances, contract storage, etc.) are persisted in the local world state.

This execution model ensures consistency across all nodes and supports both simple and complex operations in a deterministic and secure manner.

2.4 Node Service and Consensus Algorithm

DepChain's fault tolerance is built upon a variant of the *Byzantine Read/Write Epoch Consensus* algorithm. This protocol ensures that even in the presence of Byzantine faults, the system maintains agreement among honest nodes. Consensus progresses in discrete *epochs* and follows the classic pattern: `READ` → `STATE` → `WRITE` → `ACCEPT`.

READ Phase (Leader Initiation). The designated leader for an epoch initiates consensus by proposing a block through a signed `READ` message. All nodes receiving this message reply with a `STATE` message, containing their current block for the epoch and their local write set.

STATE Phase and Conditional Collect. Once the leader collects a quorum of `STATE` messages, it runs the `Conditional Collect` procedure. This logic determines whether to proceed with a value previously seen in the system (`binds`) or with the leader's new proposal (unbound).

The method `binds(states)` checks whether a previously certified value exists in the received states:

```
int highestTs = states.stream()
    .mapToInt(ConsensusMessage::getEpoch)
    .max().orElse(-1);

Block value = states.stream()
    .filter(m -> m.getEpoch() == highestTs)
    .map(ConsensusMessage::getProposedBlock)
    .findFirst().orElse(null);

boolean isQuorumHighest = quorumHighest(
    highestTs, value, states);
boolean isCertifiedValue = certifiedValue(
    highestTs, value, states);

return isQuorumHighest && isCertifiedValue;
```

Listing 5. binds(): determining valid committed value

`quorumHighest()` ensures that the value (ts, v) has the highest timestamp in a quorum, meaning no conflicting higher-timestamped value could override it:

```
int count = 0;
for (ConsensusMessage m : states) {
    int ts = m.getEpoch();
    Block v = m.getProposedBlock();
    if (ts < highestTs || (ts == highestTs
        && Objects.equals(v, value))) {
        count++;
    }
}
return count >= quorumSize;
```

Listing 6. quorumHighest(): ensuring no higher conflicting value

`certifiedValue()` checks whether the value is "certified", i.e., present in the write sets of at least $f + 1$ nodes:

```
int count = 0;
for (ConsensusMessage m : states) {
    Map<Integer, Block> writeset = m.
        getBlockWriteset();
    if (writeset != null) {
        for (Map.Entry<Integer, Block> entry
             : writeset.entrySet()) {
            if (entry.getKey() >= highestTs
                && entry.getValue().equals(
                value)) {
                count++;
                break;
            }
        }
    }
}
return count > f;
```

Listing 7. certifiedValue(): verifying replicated agreement

If both conditions hold, the leader concludes that the block is safe to commit and moves forward with it.

The Unbound Case. If no certified value exists in the collected states, the protocol uses the `unbound()` rule. This happens when all received states have epoch 0, meaning no meaningful value was committed yet. In this case, the leader proceeds with its own block:

```
for (ConsensusMessage state : states) {
    if (state.getEpoch() != 0) {
        return false;
    }
}
return true;
```

Listing 8. unbound(): default to leader proposal

The `Conditional Collect` phase then concludes by broadcasting a `WRITE` message with the selected value (either bound or unbound).

WRITE and ACCEPT Phases. Upon receiving a `WRITE` message, nodes verify its validity and respond with an `ACCEPT` message if it matches their expectations. Once a quorum of `ACCEPT` messages is received, the block is committed to the blockchain.

Consensus Completion and Response. After successful agreement, the block is executed through the EVM and appended to the local ledger. The result is then returned to the client via a `TRANSFER_RESULT` or equivalent message. Execution ensures consistency by applying the same block in the same order across all honest nodes.

Safety and Liveness. The protocol guarantees **safety** at all times, no two correct nodes will ever commit different blocks at the same epoch. **Liveness** is ensured when the leader behaves correctly and communication is reliable. All consensus messages are authenticated using digital signatures and exchanged over reliable point-to-point links.

This implementation balances simplicity and fault tolerance while remaining testable and modular for educational and experimental purposes.

3 Security and Reliability

3.1 Authenticated Perfect Links

DepChain ensures reliable and secure communication between processes using a custom abstraction called AuthenticatedPerfectLink<T>. This layer operates over UDP and provides two key guarantees:

- **Reliable Delivery:** Every message sent is eventually delivered to the recipient, even over an unreliable transport like UDP.
- **Authenticity and Integrity:** Each message is digitally signed and verified to ensure its origin and prevent tampering.

Reliable Delivery over UDP. Although UDP does not guarantee message delivery, DepChain overcomes this limitation by implementing a retransmission protocol with acknowledgments (ACKs). Every message (except ACKs) is assigned a unique identifier and retransmitted until the sender receives an ACK from the recipient.

Each send operation spawns a dedicated thread that:

1. Signs the message using the sender's private key.
2. Serializes it into JSON.
3. Sends the message via UDP.
4. Waits for an ACK, doubling the timeout with exponential backoff.

ACKs are themselves signed messages and are sent back upon successful reception and validation of a message. Once an ACK is received, retransmission stops.

Local Optimization. When a node sends a message to itself, it bypasses the network entirely and adds the message to a local queue. This improves performance and avoids unnecessary serialization or signing overhead.

Authenticity and Signature Verification. Every message includes a digital signature generated with the sender's private key. Upon reception, the recipient validates the signature using the sender's public key (retrieved from known configuration). If the signature is invalid, the message is discarded and an error is logged.

This ensures:

- Only legitimate participants can send valid messages.
- No attacker can forge or tamper with the content.

Duplicate Suppression. To ensure exactly-once semantics, each node maintains a `CollapsingSet` per sender, tracking received message IDs. If a duplicate message is detected, it is ignored, and no further action is taken.

ACK Generation. ACKs are sent only for non-ACK messages and only if the message was received from the network (not locally). The ACK contains the same message ID, is signed, and is sent back to the original sender using the same socket.

This protocol allows DepChain to simulate *authenticated perfect links*, which are a foundational assumption in most Byzantine fault-tolerant algorithms. By combining reliable delivery, signature validation and duplicate suppression, the system guarantees that:

- If a correct process sends a message to another correct process, it will eventually be delivered.
- No message is delivered more than once.
- All delivered messages are verifiably authentic.

3.2 Byzantine Scenarios and How to Activate Them

To evaluate the system's resilience to malicious or faulty behavior, DepChain allows the injection of predefined Byzantine scenarios at the node level. These scenarios simulate deviations from the correct protocol execution, enabling controlled testing of the system's fault tolerance under adversarial conditions.

Available Byzantine Behaviors. Currently, four types of Byzantine behavior can be configured for blockchain nodes:

- **DROP:** The node silently ignores all received messages, effectively becoming unresponsive.
- **INVALID SIGNATURE:** The node deliberately sends messages with invalid digital signatures, breaking authenticity guarantees.
- **WRONG BLOCK:** The node replaces the correct block in consensus messages with a randomly generated one, disrupting agreement.
- **DELAY:** The node artificially delays processing messages, simulating network or computational latency.

Activation Through Configuration. These behaviors are configured statically using the `puppet-master.py` orchestration script. This script controls the initialization of all system components and allows developers to assign specific configurations to each node process. For each Byzantine behavior, there is a corresponding configuration file that specifies which fault to activate. By changing the selected configuration before launching the system, different fault scenarios can be tested without modifying any source code.

Scope and Isolation. At this stage, Byzantine behaviors are limited exclusively to **node processes**. Client processes remain correct and are not affected by fault injection. This controlled scope ensures that the impact of faulty nodes can be studied in isolation, without introducing variability from the client side.

Purpose and Utility. The main purpose of these scenarios is to validate the protocol's ability to maintain safety and liveness properties under adversarial conditions. The use of static configuration files allows reproducibility and simplifies testing across multiple Byzantine scenarios, which is particularly useful during protocol validation and stress testing phases.

3.3 Memory Cleanup on the Client Side

To avoid memory leaks and ensure efficient resource usage, the client library implements a cleanup mechanism based on the concept of epochs. As each operation (e.g., transfer, show profile) is associated with a unique epoch identifier, the client keeps track of which epochs have already reached consensus and are considered completed.

Once an epoch has been successfully processed, older epochs that are no longer needed are cleared from memory. This is done using a dedicated method that removes outdated entries from both the set of completed epochs and the internal message buffer. This proactive strategy ensures that the client does not retain obsolete data structures indefinitely, which would otherwise lead to increased memory usage over time.

This memory cleanup mechanism is crucial in a long-running system where many operations are executed over time. It helps maintain stable performance and prevents uncontrolled memory growth during prolonged execution.

3.4 Key Generation

The generation of public-private key pairs is performed using the RSA algorithm with 2048-bit keys. The program responsible for this task is implemented in the file `KeyPairGen.java`, which creates specific directories for each node and client, storing the respective keys in binary files.

To compile and run this program, OpenJDK 21 and Maven are used, ensuring automatic creation of keys for the system's nodes and clients.

4 Future Improvements

4.1 Additional Byzantine Behavior Cases

More Byzantine behavior cases should be added to test the system more thoroughly. Some of these are harder to implement but valuable, such as equivocation attacks, selective message omission, or coordinated malicious behavior among faulty nodes.

4.2 Leader Change

This implementation does not support leader change. However, the leader selection process raises several Byzantine scenarios that would be interesting to explore, such as leader failures or malicious leaders disrupting consensus.

4.3 Instance Info and Dynamic User Creation

For better organization, an `InstanceInfo` class should be created and implemented. Although the class exists, it has not been fully developed. Dynamic user creation would also be beneficial; for example, creating a dedicated node for the system that allows adding users without manual configuration changes. This would enable testing of quorum validity in larger, more dynamic groups.

4.4 Timer Improvements

Currently, timers have some issues as they do not send the expected messages, but this does not affect the blockchain's operation. Improving the timers would enhance reliability and performance.

4.5 CleanEpoch on Nodes

Implementing `cleanEpoch` on the nodes is straightforward and would help in better state management during epoch transitions.

5 Conclusion

The *Dependable Chain* (DepChain) project successfully delivers a simplified permissioned blockchain system with high dependability, tailored for environments with static membership and known cryptographic identities. By integrating a robust Byzantine fault-tolerant consensus protocol derived from the Byzantine Read/Write Epoch Consensus algorithm, DepChain ensures reliable transaction processing and resilience against malicious or arbitrary behaviors, achieving safety under all conditions and liveness with a correct leader. The system's modular architecture, comprising four nodes and two clients, supports native cryptocurrency transfers (DepCoin) and ERC-20-compliant smart contract execution (ISTCoin), with secure communication guaranteed through authenticated perfect links over UDP.

Key achievements include the effective initialization of the blockchain via a genesis block, seamless smart contract deployment and a transaction execution model backed by Hyperledger Besu's EVM. The implementation of Byzantine fault injection through configurable scenarios (e.g., DROP, INVALID SIGNATURE, WRONG BLOCK, DELAY) enables rigorous testing of the system's fault tolerance. Additionally, client-side memory cleanup and RSA key generation enhance efficiency and security. The use of tools like Web3J for ABI encoding, Solidity for smart contracts and a Python-based launcher for orchestration demonstrates a practical and extensible design.

However, opportunities for improvement remain. Future work could expand Byzantine behavior testing with more complex scenarios like equivocation attacks, implement dynamic leader changes to handle leader failures and enhance user management with dynamic creation mechanisms. Improvements in timer reliability and node-side epoch cleanup would further optimize performance and state management. These enhancements would strengthen DepChain's robustness and scalability, making it a more versatile platform for dependable distributed applications. Overall, DepChain provides a solid foundation for educational and experimental purposes, balancing simplicity, fault tolerance and extensibility in a permissioned blockchain context.