Highly Dependable Systems
Report - Stage 2

André Pereira - 103082 - andre.aguiar.pereira@tecnico.ulisboa.pt
André Gamito - 104167 - andre.gamito@tecnico.ulisboa.pt
Miguel Bibi - 102737 - miguel.bibi@tecnico.ulisboa.pt

Instituto Superior Técnico - Tagus Park, Universidade de Lisboa
2024/2025

Abstract. This report presents the initial phase of the SEC-24/25 project, which aims to develop a highly dependable permissioned blockchain system, DepChain. The first stage focuses on implementing the consensus layer, utilizing the Byzantine Read/Write Epoch Consensus algorithm to ensure fault tolerance. DepChain operates under predefined assumptions, including static membership, a pre-established Public Key Infrastructure (PKI), and a fault-free leader. The implementation guarantees safety under all conditions and liveness when the leader functions correctly.
The consensus mechanism consists of a read phase, which determines if a previously committed value should be preserved, and a write phase, ensuring agreement on a single value among correct members. To enhance dependability, the system implements Stubborn Links (SLs) over UDP, ensuring persistent message retransmission, and Authenticated Perfect Links (APLs) for reliable and authenticated communication. A client library and a comprehensive test suite validate system correctness and resilience against Byzantine faults. This report outlines the design choices, challenges encountered, and solutions implemented to achieve a secure and reliable blockchain consensus layer.

Keywords: Byzantine Fault Tolerance · Authenticated Perfect Links · Transactions.

1 Introduction

This report presents the development of the first stage of the SEC-24/25 project, which aims to design a highly dependable permissioned blockchain system, DepChain. The initial phase focuses on implementing the consensus layer, leveraging the Byzantine Read/Write Epoch Consensus algorithm.

The system operates under a set of predefined assumptions, including a static membership, a pre-established Public Key Infrastructure (PKI), and the assumption that the leader remains fault-free. The implementation must ensure safety under all conditions and liveness only when the leader behaves correctly.

In addition to designing the consensus protocol, this phase includes the development of a client library to facilitate interactions with the blockchain service

and a test suite to validate system correctness and resilience against Byzantine behavior. This report outlines the design choices, challenges encountered, and solutions implemented to meet the project's objectives.

2 BFT Consensus

The Byzantine Fault Tolerant (BFT) Consensus ensures that all correct blockchain members in DepChain reach agreement despite Byzantine faults. DepChain uses the Byzantine Read/Write Epoch Consensus, structured into epochs, each managed by a leader. If the leader is correct, consensus is reached; otherwise, the epoch is aborted and restarted.

2.1 Reads

The read phase collects state information from a Byzantine quorum to determine if a previously decided value must be preserved (lock-in property).

1. The **leader** sends a **READ** request.
2. Members reply with their **STATE**, including:
   - **(valts, val)**: The most recent value confirmed in a Byzantine quorum.
   - **writeset**: All values ever written with their timestamps.
3. The leader checks if a value qualifies as **epoch-decided**, based on:
   - The highest timestamp among a Byzantine quorum.
   - Presence in at least *f+1* members writesets.
4. If a valid value is found, it proceeds to the write phase; otherwise, a new value may be proposed.

2.2 Writes

The **write phase** ensures all correct members commit to a single value.

1. The **leader** proposes a value (v), either from the read phase or newly chosen.
2. It sends a **WRITE(v)** request to all members.
3. Members accept v if they receive a Byzantine quorum of **WRITE** messages and respond with **ACCEPT(v)**.
4. If *2f+1* **ACCEPT(v)** messages are collected, the value is **epoch-decided**.

If the write phase fails due to a faulty leader, the epoch is **aborted** and restarted with a new leader.

This ensures **safety** (no conflicting decisions) and **liveness** (progress when the leader is correct).

3 Dependability Guarantees

3.1 Stubborn Links

Our Stubborn Links (SL) layer provides reliable point-to-point communication over UDP (`DatagramSocket`). It achieves this by persistently retransmitting serialized `AuthenticatedMessage` objects until an explicit acknowledgment (ACK) based on the unique `messageID` is received.

Sending (`sp2pSend`) involves spawning a thread per message to handle retransmissions, tracking ACK status via a `ConcurrentHashMap`. A central receiver thread listens for incoming packets, processing ACKs to halt corresponding retransmissions, or deserializing incoming messages, sending back an ACK, and delivering the message to the upper layer via a `MessageCallback`. Retransmissions are limited by a maximum attempt count to prevent indefinite loops. This ensures *at-least-once* delivery semantics, provided the destination is reachable within the attempt limit.

3.2 Authenticated Perfect Links

The Authenticated Perfect Links (APL) implementation builds upon Stubborn Links (SL) to provide reliable, authenticated, and confidential message delivery between processes. Our design ensures three critical properties inherited or enhanced from SL: reliable delivery, no duplication, and adds robust authenticity and confidentiality through a hybrid encryption scheme and message integrity checks.

Sending Process:

When an application requests to send a message (`payload`, `command`), the APL layer performs the following steps:

1. **Confidentiality (Encryption):**
   The message `payload` and `command` are encrypted using a symmetric AES key. This ensures the content remains confidential during transit.
   The AES key itself is then encrypted using the recipient's RSA public key. This asymmetric encryption allows secure sharing of the symmetric key needed for decryption only by the intended recipient.
2. **Integrity & Authentication (Hashing):**
   A SHA-256 hash is computed specifically on the encrypted payload. This hash serves as the `authString`. Hashing the ciphertext ensures that any tampering with the encrypted message during transit can be detected.
3. **Packaging:**
   An intermediate `Message` object is created containing the encrypted payload, encrypted command, the RSA-encrypted AES key, and the source identifier. An `AuthenticatedMessage` is then created, encapsulating this `Message` object along with the calculated `authString`. Each `AuthenticatedMessage` inherently possesses a unique message ID.

4. **Reliable Transmission:**
   The final `AuthenticatedMessage` is passed to the underlying Stubborn Links (`stubbornLink`) layer, which guarantees reliable delivery (*at-least-once* semantics) to the destination.

Receiving Process:

Upon receiving an `AuthenticatedMessage` via the `onMessageReceived` callback from Stubborn Links, the APL layer executes these steps:

1. **No Duplication:**
   The system first checks if an `AuthenticatedMessage` with the same message ID has already been received and processed by checking its presence in the `received` list. If it's a duplicate, the message is immediately discarded, fulfilling the "no duplication" property.
2. **Integrity & Authentication Verification:**
   The SHA-256 hash of the received encrypted payload is re-computed.
   This computed hash is compared against the `authString` included in the `AuthenticatedMessage`. If the hashes do not match, the message is considered tampered or corrupted and is discarded, ensuring message authenticity and integrity.
3. **Confidentiality (Decryption):**
   If the hash verification succeeds, the RSA-encrypted AES key is decrypted using the host's private key.
   The recovered AES key is then used to decrypt the message `payload` and `command`.

By combining the guaranteed delivery of Stubborn Links with hybrid encryption (AES for content, RSA for key exchange) and SHA-256 hashing of the ciphertext, the Authenticated Perfect Links layer provides reliable, *exactly-once*, confidential, and authenticated communication channels between network entities.

4 Design

4.1 System Architecture

Our architecture consists of a leader and three other members, all of whom communicate securely via fully authenticated links. The member ports and private/public keys are shared resources, and their location is stored in the resources.json and keys.json files. At startup, each member generates their public and private keys. Clients can make requests to our Client Library through a REST API, which then instructs the leader on actions to take, such as proposing a new value to the blockchain.

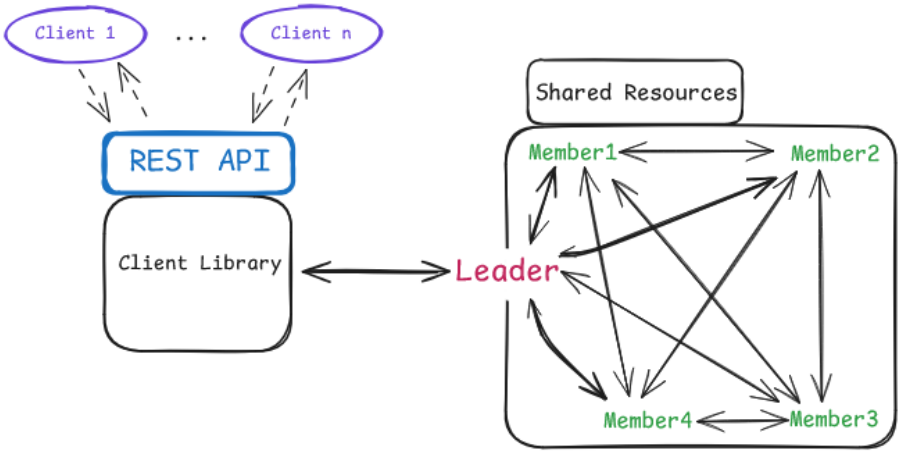

Fig. 1. Diagram

5 Transactions

5.1 DepCoin Transactions

DepCoin transactions are at the core of the DepChain blockchain, facilitating value transfers between accounts. Each transaction is a digitally signed instruction that specifies a sender, a receiver, an amount of DepCoin, and optionally, a data field for contract interactions.

A transaction follows the structure defined in the `Transaction` class and includes:

- **Sender and Receiver:** Represented by unique addresses corresponding to user or contract accounts.
- **Amount:** The value to transfer, expressed as a `double`.
- **Data:** Optional information for invoking smart contracts.
- **Signature:** A cryptographic proof generated using the sender's private key to ensure authenticity and integrity.

Transactions are validated through multiple checks implemented in the `WorldState` class:

1. **Account existence:** Both sender and receiver must exist in the current world state.
2. **Sufficient balance:** The sender must have a balance equal to or greater than the transaction amount.
3. **Signature verification:** The transaction signature must match the sender's public key.

Upon block proposal, the leader includes a batch of transactions in a `Block` object. Before appending the block to the chain, the system uses a deep copy of

the `WorldState` to validate all transactions, ensuring none would invalidate the ledger. If all transactions pass, the world state is updated accordingly.

Finally, each valid block, including all transactions and the resulting state, is serialized and persisted in JSON format. This guarantees transparency and auditability of every DepCoin movement within the chain.

5.2 ISTCoin Transactions

6 Tests

We developed tests to validate the correctness and resilience of the consensus protocol against both benign and Byzantine behaviors. The following tests were designed and executed:

6.1 "Yes Man" Test

In this test scenario, one member behaves as a **"Yes Man"**, meaning it *always responds positively* to proposed blocks by sending `WRITE` messages during the write phase—regardless of the block's validity. Contrary to an incorrect assumption, "Yes Man" does **not** proactively send blocks; it merely accepts any value it is asked to vote on. This test helps evaluate how the system handles members that blindly agree, potentially simulating faulty or overly trusting nodes.

6.2 "No Man" Test

Here, one member behaves as a **"No Man"**, which *always responds negatively* by either refusing to send `WRITE` messages or replying with an `ABORT` during consensus. Similar to "Yes Man", this node does **not** propose or initiate blocks, and instead acts as a permanently uncooperative participant. This test validates the system's ability to maintain liveness and fault tolerance when confronted with consistently rejecting nodes.

6.3 "Random Man" Test

The **"Random Man"** test introduces a member that behaves unpredictably by sending `WRITE` messages to some peers while sending `ABORT` messages to others, within the same consensus round. This simulates a Byzantine node with non-deterministic behavior, allowing us to assess the protocol's robustness when facing erratic or malicious members.

6.4 Flooding Test

To assess performance and batching behavior, the **Flooding Test** sends **12 transactions in rapid succession**. The system is expected to handle this burst by grouping the transactions into **4 blocks**, each containing **3 transactions**. This test checks the system's transaction throughput, block formation logic, and overall stability under high load scenarios.

6.5 Evaluation

All tests were executed in a controlled environment with fixed membership and a fault-free leader. The consensus protocol demonstrated its ability to maintain **safety**, preventing conflicting blocks even in the presence of Byzantine behavior. Furthermore, it preserved **liveness** under normal conditions, especially when the leader was correct. The tests helped validate the expected behavior of DepChain under diverse and adversarial conditions.

7 Possible Threats

7.1 Denial of Service (DoS)

The stubborn retransmission mechanism, while ensuring reliable delivery, introduces vulnerability to DoS attacks. An attacker could flood the network with messages that appear legitimate but contain invalid authentication strings, forcing receivers to perform computationally expensive hash verifications for each message.

7.2 Replay Attacks

Although our APL implementation prevents duplicate delivery of messages within a single session through message ID tracking, it may be vulnerable to replay attacks across different sessions. An attacker could capture authenticated messages and replay them in a future session if session identifiers are not incorporated into the authentication process.

References

1. Christian Cachin Rachid Guerraoui Luís Rodrigues, Introduction to Reliable and Secure Distributed Programming (2011)