

Vietnam National University, HCMC
University of Science
Faculty of Information Technology f



REPORT

Lab 01: Design and Implement a Minimal Layer 1 Blockchain

Group members:

Trần Văn Anh Thư - 22120357

Vũ Châu Minh Trí - 22120456

Khưu Hải Châu - 22120457

Dương Hoài Minh - 22120460

Course: Blockchain and Application

HCMC - 2025

Contents

1	Introduction	2
1.1	Objectives	2
1.2	System requirements	2
1.3	Project Structure	3
2	System Architecture	4
2.1	Design Choices	4
2.2	Components	4
2.2.1	Cryptography Module	4
2.2.2	Consensus Layer	5
2.2.3	Execution Layer	6
2.2.4	Network Simulator	6
2.2.5	Blockchain Node	6
2.3	Testing and Results	6
2.3.1	Test Case: <code>test_crypto.py</code>	7
2.3.2	Test Case: <code>test_execution.py</code>	7
2.3.3	Test Case: <code>test_consensus.py</code>	7
2.3.4	Test Case (E2E): Network Fault Tolerance and Safety	8
2.3.5	Deterministic Verification	8

Chapter 1

Introduction

1.1. Objectives

This project implements a minimal Layer 1 Blockchain from scratch, demonstrating core blockchain concepts including:

- **Byzantine Fault Tolerant (BFT) Consensus:** A simplified two-phase commit protocol (prevote/precommit) inspired by Tendermint
- **Cryptographic Security:** Ed25519 digital signatures with context separation to prevent replay attacks
- **Deterministic Execution:** Reproducible state transitions ensuring all nodes reach identical states
- **Network Simulation:** Realistic network conditions with configurable delays, packet drops, and rate limiting

The implementation serves as an educational tool to understand the fundamental building blocks of blockchain technology.

1.2. System requirements

The purpose of this system is to demonstrate that a set of distributed, communicating nodes can permanently and reliably agree on a single, ordered history of events (blocks). This agreement must hold even when messages are delayed, duplicated, reordered, or temporarily lost. A successful run requires that once a block is finalized, all honest nodes permanently agree on its height and hash, and no conflicting block can ever be finalized. The core guarantee is that every correct node must converge on the same sequence of finalized blocks.

1.3. Project Structure

```
root/
  config/
    default_config.json      # Default simulation configuration
    requirements.txt         # Python dependencies
  src/
    consensus.py            # BFT consensus state management
    crypto.py                # Ed25519 signatures and hashing
    execution.py             # Transaction execution and state
    logger.py                # Deterministic logging
    network.py               # Network simulator with delays/drops
    node.py                  # Blockchain node implementation
  tests/
    run_tests.py             # Main test runner
    test_consensus.py        # Consensus unit tests
    test_crypto.py           # Cryptography unit tests
    test_e2e.py               # End-to-end integration tests
    test_execution.py         # Execution layer tests
    verify_determinism.py   # Determinism verification
README.md
```

Chapter 2

System Architecture

2.1. Design Choices

To meet these requirements, the system incorporates three primary architectural choices:

- **Byzantine Fault Tolerant Consensus:** Two-phase voting protocol ensuring safety
- **Cryptographic Authentication:** Digital signatures preventing message forgery
- **Deterministic State Machine:** Reproducible execution across all nodes

It follows a modular architecture with clear separation of concerns:

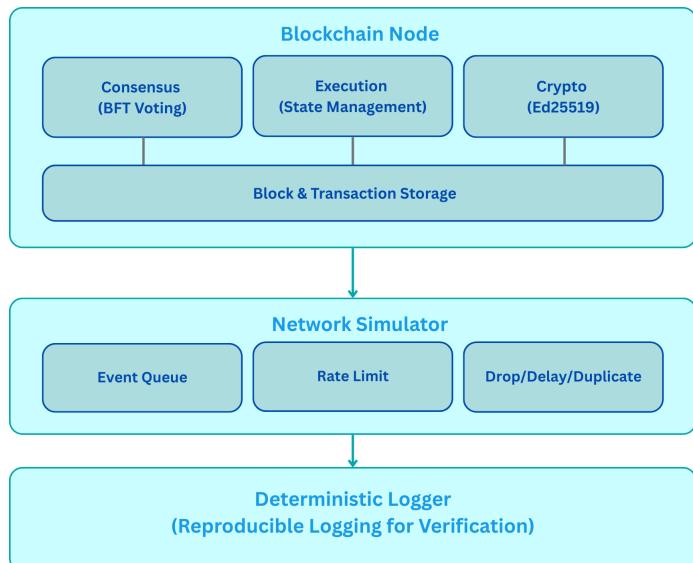


Figure 2.1: High-Level System Architecture

2.2. Components

2.2.1. Cryptography Module

File: `src/crypto.py`

The Cryptography Module provides the security foundation by ensuring data integrity and message authenticity across the network.

Key Functions

This module performs three critical tasks:

1. Uses **SHA-256** to generate deterministic hashes for all data structures (blocks, transactions), guaranteeing immutability and easy detection of tampering.
2. Implements **Ed25519 signatures** to allow validators to cryptographically sign all consensus votes and blocks, proving the origin of messages and preventing **forgery**.
3. Enforces **context separation** (incorporating message type and chain ID) into the data being signed. This prevents signatures intended for one context from being reused maliciously.

Context Separation

The signing process includes a context prefix to prevent signature reuse:

- TX: Transaction signatures
- VOTE: Consensus vote signatures
- HEADER: Block header signatures

2.2.2. Consensus Layer

File: `src/consensus.py` The consensus mechanism implements a simplified BFT protocol with two voting phases. It:

- Validates and stores prevotes and precommits.
- Checks for 2/3+ majority to finalize blocks.
- Prevents conflicting block finalization at the same height

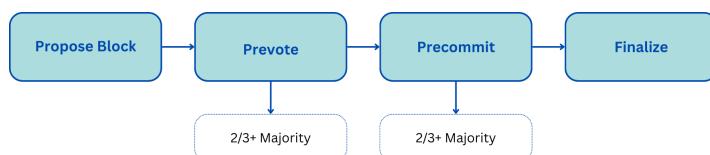


Figure 2.2: Protocol Flow

2.2.3. Execution Layer

File: `src/execution.py` The execution layer manages the application state and transaction processing.. It:

- Maintains a key-value store for application data
- Enforces ownership: Only the owner can modify keys prefixed with their public key
- Computes deterministic state commitment (hash of sorted state)

2.2.4. Network Simulator

File: `src/network.py`

Models unreliable P2P communication for realistic testing:

- **Unreliable Conditions:** Messages may be **delayed, duplicated, reordered, or dropped** with configurable rates.
- **Rate Limiting:** Each node enforces an outbound rate limit and can temporarily block overactive peers.
- **Deterministic Events:** Uses an event-driven priority queue (min-heap) to ensure messages are processed deterministically based on their simulated arrival time.

2.2.5. Blockchain Node

File: `src/node.py`

The orchestrator, integrating all components (Crypto, Consensus, Execution) to manage the node's lifecycle, including: transaction validation and queuing, block proposal, vote creation/reception, and applying finalized blocks to the local state.

2.3. Testing and Results

Testing and verification were performed to confirm that the minimal Layer 1 blockchain satisfies the core requirements of **safety, liveness, and determinism** under unreliable network conditions. The test suite includes both Unit Tests for component correctness and comprehensive End-to-End (E2E) Tests for integration and system behavior.

2.3.1. Test Case: test_crypto.py

Purpose: To verify the integrity and authentication mechanisms, including Ed25519 signature validity, deterministic hashing, and, critically, the context separation check.

- Expected Outcome: A signature generated for context VOTE must be rejected if verified under context HEADER. Identical data serialized and hashed must produce the same SHA-256 output.

Result Log in Terminal:

```
$ python tests/test_crypto.py
```

```
Ran 11 tests in 0.006s
```

```
OK
```

2.3.2. Test Case: test_execution.py

Purpose: To validate transaction application logic, specifically ensuring transactions only modify data owned by the sender and that the final state commitment is deterministic.

- Expected Outcome: A transaction attempting to modify a key not owned by the sender must be rejected. Valid, ordered transactions must result in an identical, verifiable state_hash.

Result Log in Terminal:

```
$ python tests/test_execution.py
```

```
Ran 10 tests in 0.006s
```

```
OK
```

2.3.3. Test Case: test_consensus.py

Purpose: To test the BFT vote counting logic, the **2/3+** majority rule, and the safety guard against conflicting block finalization.

- Expected Outcome: Finalization occurs only after receiving a strict majority of valid **Precommits**. The system must prevent finalizing two distinct blocks at the same height (safety).

Result Log in Terminal:

```
$ python tests/test_consensus.py
```

```
Ran 11 tests in 0.010s
```

```
OK
```

2.3.4. Test Case (E2E): Network Fault Tolerance and Safety

Purpose: To confirm the system maintains safety (single finalized block per height) and achieves liveness despite network faults (delays, drops, duplicates).

- Expected Outcome: Nodes must converge on the same sequence of finalized blocks, and no conflicting blocks should be finalized.

Result Log in Terminal:

```
$ python tests/test_e2e.py
```

```
Ran 8 tests in 0.032s
```

```
OK
```

2.3.5. Deterministic Verification

Script: `verify_determinism.py`

Purpose: To confirm the absolute reproducibility of the system, a primary requirement for verification. The script runs the complete simulation scenario twice with an identical configuration and random seed and then checks for byte-equality of the results.

- Expected Outcome: The output logs must be byte-identical, and all nodes must converge on the same final `state_hash` and finalized block history.

Result Log in Terminal:

```
$ python tests/verify_determinism.py
```

```
=====
```

BLOCKCHAIN DETERMINISM VERIFICATION

Lab 01 - Part 8: Determinism and Logging

```
=====
```

```
RUN 1: Starting deterministic scenario (seed=42)
```

```
=====
```

```
RUN 1: Completed successfully
```

- Log file: logs/run_1.log
- Summary file: logs/run_1_summary.json
- Total logs: 75
- Finalized blocks: 2

```
=====
```

```
=====
```

```
RUN 2: Starting deterministic scenario (seed=42)
```

```
=====
```

```
RUN 2: Completed successfully
```

- Log file: logs/run_2.log
- Summary file: logs/run_2_summary.json
- Total logs: 75
- Finalized blocks: 2

```
=====
```

Comparing log files...

Log files are BYTE-IDENTICAL

Comparing final states...

```
=====
```

COMPARISON RESULTS

```
=====
Node: Node_0
State hash matches: 24a7b20663030f0c...
Finalized heights match: [0, 1]
State data matches (3 entries)
```

```
Node: Node_1
State hash matches: 24a7b20663030f0c...
Finalized heights match: [0, 1]
State data matches (3 entries)
```

```
Node: Node_2
State hash matches: 24a7b20663030f0c...
Finalized heights match: [0, 1]
State data matches (3 entries)
```

```
Node: Node_3
State hash matches: 24a7b20663030f0c...
Finalized heights match: [0, 1]
State data matches (3 entries)
```

```
=====
FINAL VERDICT
=====
```

```
SUCCESS
Both runs produced IDENTICAL logs and final states.
The system is DETERMINISTIC and REPRODUCIBLE.
```