LINFO2345 - Languages and algorithms for distributed applications

Project Report:

Proof of Stack

Student names	Noma	Mail
Juitcha Hendrix	67212000	hendrix.juitcha@student.uclouvain.be
Meli François	67202000	francois.meli@student.uclouvain.be

Teaching team:
Teacher: Prof. Peter Van Roy
T.A: Matthieu Pigaglio

December 6, 2025

Contents

1 Introduction

This project talks about one of consensus mechanisms used in blockchain technology to validate transactions (PoS). But first, by definition, a blockchain is a distributed ledger where each block references the hash of its predecessor to conserve integrity and immutability.

Usual cryptocurrencies, as seen in our cybersecurity course (LELEC2770), such as Bitcoin and early Ethereum, relied on *Proof of Work* (PoW), which secures the system but consumes vast resources (computational, electrical power consumption, ..., even time itself).

To reduce the overall cost, *Proof of Stake* (PoS) was introduced. In PoS, validators are selected based on different criteria that we'll discuss in the following lines.

The goal of this project is to implement a simplified PoS blockchain in `Erlang`; that is considered as a general-purpose, concurrent, functional programming language designed for building scalable and fault-tolerant systems.

The work is divided into three main parts: building a distributed ledger (part 1), designing a validator election protocol (part 2), and integrating a full proof of stake mechanism (part 3).

2 Important notes, data and logs

Warnings:

- There're prerequisites to this project. Before doing anything, first consult the detailed `README.md` provided in our archive for checking how our code works and how the log files are generated. We've chosen to not include too much files to limit the archive size.

- Each part generates specific files:

  - `-test1 genarates .csv files`
  - `-test2 generates .txt files (logs)`
  - `-test3 generate .csv files and put logs in the logs_test3.txt`

- Usually, It's at the compilation time that the `.txt log, .csv files` are generated.

- Just make sure to remove generated file before running tests again (see commands inside the `README.md`): E.g `rm *.beam blockchain_*.csv *_election_log.txt election_results.txt`

- For performance concern, we've also chosen to keep a centralized structure instead of 3 separated directories as it was asked. This, we explicitely declared inside the README.md how the project could produce the same behavioras faithful as with the `3-directories version`

- Github repo: `https://github.com/wilfred33/projetErlang/tree/latest_branch`

3 Part I - Blockchain Distributed Ledger

3.1 Node Structure

The module `node.erl` defines three types of nodes:

- **Builder**: generates and broadcasts blocks.
- **Validator**: validates blocks and participates in elections.
- **NonValidator**: stores the blockchain without contributing to its creation.

Each node maintains a state consisting of:

- a unique name,
- a type (builder / validator / non_validator),
- a local copy of the blockchain,
- a list of known nodes (*KnownNodes*),
- a CSV storage file.

3.2 Broadcast System

We implemented a push-based broadcast mechanism where each node sends the new block to all its known nodes.

```
broadcast_block(Block, KnownNodes, SenderName) ->
    lists:foreach(fun(NodeName) ->
        case NodeName of
            SenderName -> ok; % Do not send to oneself
            _ ->
                NodeAtom = list_to_atom(NodeName),
                NodeAtom ! {new_block, Block}
        end
    end, KnownNodes).
```

- **Asynchronous**: Uses Erlang message passing (the `!` operator).
- **Decentralized**: Each node forwards the block to the nodes it knows.
- **Robust**: Error handling implemented with `try-catch`.
- **No duplication**: The sender is checked to avoid resending to itself.

3.3 Network Topology

We chose a partial mesh topology:

3.3.1 Initialization

- Each node is started with:

  ```
  start(Type, Index, KnownNodes)
  ```

- Each node receives an initial list of known nodes.

3.3.2 Dynamics

- Nodes can be added dynamically using the message {`add_node, NewNode`}.
- The `KnownNodes` list is propagated during node creation.
- Each node maintains its own partial view of the network.

3.3.3 Advantages of This Approach

- **Scalability**: No single point of failure.
- **Resilience**: The network remains operational even if some nodes fail.
- **Flexibility**: Easy to add or remove nodes.
- **Realism**: Better reflects real blockchain networks.

3.4 Additional Validation and Testing

3.4.1 Multi-Node Testing

To ensure the robustness of our distributed ledger implementation, we conducted tests with configurations of 3, 7 and 10 nodes. Each test scenario was executed using the commands provided in the `README.md`, which automatically generate the corresponding log files. These logs capture the broadcast behavior, block creation, and storage operations across the network. The results confirm that our broadcast mechanism and block storage scale correctly with increasing network size.

3.4.2 Block Storage in CSV

Each node maintains its own local copy of the blockchain in a CSV file. For every block, the following fields are recorded:

- Block number
- Merkle tree root of the block
- Address of the builder node
- Hash of the previous block
- List of transaction IDs included in the block

This structure allow that the integrity of the chain can be verified independently by each node. The CSV format also facilitates easy inspection and validation of the stored data during testing and analysis.

4 Part II - Consensus

4.1 Election Workflow

Step 1: Triggering

The head of the current proposer group starts the election:

```
start_election(HeadValidatorName, AllValidators, AllNodes)
```

- Broadcasts {`start_election`} to all nodes.
- Blocks the creation of new blocks (`ElectionInProgress = true`).

Step 2: Ring Circulation

```
{shuffle_round, ShuffledList, RemainingValidators, Initiator}
```

- The head performs the first shuffle.
- Passes the list to the next validator in the ring.
- Each validator:
  - Re-shuffles the list,
  - Passes it to the next validator in sequence.
- The process continues until the list returns to the head.

Step 3: Selection of the New Proposer Group

When the list returns to the head:

```
{election_complete, FinalShuffledList}
```

- Selects the top 10% of validators.
- Forms the new proposer group.
- Broadcasts the result to all validators.

Step 4: Epoch Transition

The new head (first validator in the new list):

- Broadcasts {`start_new_epoch`} to all nodes,
- Unblocks block creation,
- Becomes the new leader.

4.2 Time Election

Proposer group election takes approximately 8000ms - The complete election process, from initiation to the establishment of a new proposer group, requires approximately 8 seconds under standard testing conditions.

4.2.1 Result Analysis

Data can be consulted in the election result file - Detailed timing metrics and election outcomes are recorded in the `election_result.txt` log file for performance analysis.

Individual validator actions observable in text files - Each validator's specific actions during the election process are documented in dedicated log files named `validator_X_election_log.txt`, where X represents the validator's identifier number

5 Part III - Proof of Stake Protocol

5.1 Proof of Stake System Architecture

5.1.1 Main Components

- **Centralized Transaction Pool:**

  ```
  start_transaction_pool(TransactionFile)
  ```

  – Centralized management of pending transactions
  – Provides batches of 10 transactions to builders
  – In-memory persistence with query interface

- **Centralized Block Controller:**

  ```
  block_controller(CurrentBuilder, AllValidators, AllNodes, BlockCount)
  ```

  – Orchestrates the complete creation/validation cycle
  – Manages automatic elections every 10 blocks
  – Ensures smooth transition between epochs

- **Integrated Workflow:**

  ```
  Transactions → Builder → ProposerGroup → Validation → Broadcast → Election
  ```

5.2 Proof of Stake Block Lifecycle

Phase 1: Block Creation

Current builder (head of the ProposerGroup):

1. Requests 10 transactions from the centralized pool
2. Creates a block with these transactions
3. Sends the block to the ProposerGroup for validation

Phase 2: Consensus Validation

Each validator in the ProposerGroup:

1. Receives the block via {`validate_block, Block, BuilderName`}
2. Verifies validity via `block:is_valid(Block)`
3. Sends its vote ({`block_approved`} or {`block_rejected_by`})

Phase 3: Majority Decision

Builder collects votes:

```
MajorityThreshold = (TotalValidators div 2) + 1
```

- If ≥ Majority approvals → Broadcast to the entire network
- Otherwise → Block rejected, new block created

Phase 4: Propagation

Once approved, the block is disseminated to all nodes using the broadcast mechanism from Part 1.

5.2.1 Centralized Counter Management

Counter managed by the central controller - The block count is tracked by the centralized block controller, which triggers elections at precise intervals.

5.2.2 Automatic Election Mechanism

Automatic election after 10 successful blocks - The system automatically initiates an election protocol once 10 blocks have been successfully validated and added to the blockchain, ensuring regular rotation of validator responsibilities.

6 Conclusion

This project provided a practical implementation of a simplified Proof of Stake blockchain in Erlang. Starting from the construction of a distributed ledger, we integrated block creation, transaction storage, and broadcasting mechanisms. We then designed and tested a consensus protocol to elect validator groups, ensuring fairness and resilience across epochs. Finally, we implemented the full Proof of Stake workflow, where validators collectively verify blocks before further processing.

Overall, the project demonstrated how distributed systems, cryptographic primitives, and consensus algorithms can combine to form stable and functionally-efficient blockchains.