Rust Blockchain Project

Implementation Report

GOCHE Elie

September 3, 2025

Contents

1 Introduction

This project consists in implementing a simplified blockchain in the Rust programming language. The main objective is to model how a blockchain-like system operates, integrating a Proof of Work (PoW) mechanism for block validation. Each block contains, in addition to the usual data (parent hash, nonce, miner identity), an original piece of information: a dance move chosen by the miner. The project includes block creation and validation, mining difficulty management, and network interaction with a local server. The purpose is to deepen understanding of fundamental blockchain principles while practicing programming with Rust.

2 Code Structure

The project is organized around several modules:

- `block.rs`: contains the `block` structure, the block hashing function, block validation, and Proof of Work (PoW).
- `network.rs`: enables a miner to send the blocks they create to the server and to periodically fetch new blocks from this server.
- `main.rs`: main entry point, mining logic, blockchain display, and block creation.
- `simpletree.rs`: for managing a tree structure of blocks (the blockchain).

3 Implemented Features

This section presents, for each module of the project, the key features developed in addition to the base code provided by the instructor.

3.1 Module `block.rs`

3.1.1 `hash_block` function

This function computes the block hash by combining the fields `parent_hash`, `miner`, `nonce` and `dancemove`, then applying the SHA-256 algorithm to produce a unique fingerprint of the block.

```rust
pub fn hash_block(&self) -> [u8; 32] {
    let mut hasher = Sha256::new();

    hasher.update(&self.parent_hash);
    hasher.update(self.miner.as_bytes());
    hasher.update(&self.nonce.to_le_bytes());
    hasher.update(&[self.dancemove as u8]);

    hasher.finalize().into()
}
```

Listing 1: `hash_block` function

3.1.2 `pow_check` function

This function verifies whether a hash satisfies the required difficulty level. It counts the number of leading zero bits at the beginning of the hash and returns true if this number is greater than or equal to the difficulty.

```rust
pub fn pow_check(&self, hash: &[u8], difficulty: u32) -> bool {
    let mut leading_bits = 0;

    for byte in hash {
        if *byte == 0 {
            leading_bits += 8;
        } else {
            leading_bits += byte.leading_zeros();
            break;
        }
    }

    leading_bits >= difficulty
}
```

Listing 2: `pow_check` function

3.1.3 `solve_block` function

This function attempts to solve the block by finding a `nonce` value such that the block hash meets the specified difficulty (number of leading zero bits at the start of the hash). It uses a random generator to test different values until success or until a maximum number of iterations is reached.

```rust
pub fn solve_block<R: RngCore>(
    &mut self,
    rng: &mut R,
    difficulty: u32,
    max_iteration: Option<u64>,
) -> Option<Vec<u8>> {
    for _ in 0..max_iteration.unwrap_or(u64::MAX) {
        self.nonce = rng.next_u64();
        let hash = self.hash_block();

        if self.pow_check(&hash, difficulty) {
            return Some(hash.to_vec());
        }
    }
    None
}
```

Listing 3: `solve_block` function

3.2 Module `main.rs`

3.2.1 Mining

`mine` function The `mine` function allows a miner to participate in the blockchain network by mining new blocks. It first establishes a network connection to receive existing blocks. If no genesis block is present, it creates one, mines it, and sends it to the network. Next, it builds a local blockchain from the received blocks, identifies the deepest leaf (and with the smallest nonce), then creates a new block based on that leaf. This block is then mined according to the imposed difficulty, and once the proof of work is solved, it is sent to the network. This process is repeated in a loop, ensuring continuous contribution to the evolution of the chain.

```rust
fn mine(difficulty: &u32, miner_name: &String, max_iter: &Option<u64>) {
    let (tx_net_send, rx_net) = mpsc::sync_channel(1);
```

```
    let (tx_net, rx_net_ctrl) = mpsc::channel();

    thread::spawn(move || {
        let mut net = NetworkConnector::new(tx_net_send, rx_net_ctrl);
        net.sync().expect("Network failure");
    });

    let mut rng: ThreadRng = rand::rng();

    loop {
        let received = match rx_net.recv() {
            Ok(blocks) => blocks,
            Err(_) => {
                eprintln!("Failed to receive from network.");
                continue;
            }
        };

        // Search or create a genesis block
        let genesis = received.iter()
            .find(|b| b.is_genesis(*difficulty))
            .cloned()
            .unwrap_or_else(|| {
                let mut block = Block::new(vec![], "Genesis".to_string()
                    , 0, random_dancemove(&mut rng));
                block.solve_block(&mut rng, *difficulty, *max_iter);
                tx_net.send(block.clone()).expect("Failed to send
                    genesis block");
                block
            });

        // Build local blockchain
        let (chain, _) = Blockchain::new_from_genesis_and_vec(genesis.
            clone(), received);

        // Find the deepest leaf with the smallest nonce
        let leaf = chain.blocks
            .deepest_leafs()
            .into_iter()
            .min_by_key(|b| b.value().nonce)
            .unwrap()
            .value()
            .clone();

        // Create and mine a new block
        let mut new_block = Block::new(leaf.hash_block().to_vec(),
            miner_name.to_string(), 0, random_dancemove(&mut rng));
        new_block.solve_block(&mut rng, *difficulty, *max_iter);

        tx_net.send(new_block).expect("Failed to send block");

        println!("Current blockchain state:\n{}", chain);
    }
}
```

Listing 4: `mine` function

3.2.2 Random dance move selection

Before mining a block, the miner randomly selects a `DanceMove`. Each move (Y, M, C, A) has an equal probability of being chosen. This choice is made using a pseudo-random number generator.

```
fn random_dancemove(rng: &mut StdRng) -> DanceMove {
    match rng.random_range(0..4) {
        0 => DanceMove::Y,
        1 => DanceMove::M,
        2 => DanceMove::C,
        _ => DanceMove::A,
    }
}
);
```

Listing 5: Random `DanceMove` selection

3.2.3 Parent selection in the blockchain

The miner always selects an existing block as the parent for a new block to mine. By default, it chooses a block located at the maximum depth of the blockchain. However, if several blocks have the same depth, the miner selects the one whose `nonce` is the lowest. This ensures deterministic behavior in the case of competing branches of equal height.

```
let leaf_block = deepest_leaves
    .iter()
    .min_by_key(|leaf| leaf.value().nonce)
    .unwrap()
    .value()
    .clone();

let mut new_block = block::new(
    leaf_block.hash_block().to_vec(),
    miner_name.to_string(),
    0,
    dancemove,
);
```

Listing 6: Parent block selected for mining

This logic relies on the `deepest_leafs()` function defined in the `simpletree.rs` module, which returns all leaves located at the maximum depth of the tree.

3.3 Module `simpletree.rs`

3.3.1 `deepest_leafs` function

This function traverses the tree to find all the deepest leaf nodes. It uses an internal recursive function to track the maximum depth reached and returns a list of references to those nodes.

```rust
pub fn deepest_leafs(&self) -> Vec<&TreeNode<T>> {
    fn helper<'a, T: Default + Parenting>(
        node: &'a TreeNode<T>,
        depth: usize,
        max_depth: &mut usize,
        deepest: &mut Vec<&'a TreeNode<T>>,
    ) {
        if node.children.is_empty() {
            if depth > *max_depth {
                *max_depth = depth;
                deepest.clear();
                deepest.push(node);
            } else if depth == *max_depth {
                deepest.push(node);
            }
        } else {
            for child in &node.children {
                helper(child, depth + 1, max_depth, deepest);
            }
        }
    }

    let mut max_depth = 0;
    let mut result = Vec::new();
    helper(self, 1, &mut max_depth, &mut result);
    result
}
```

Listing 7: `deepest_leafs` function

4 Challenges Encountered

- **Choosing the parent to mine in case of concurrent branches:** It was necessary to define a deterministic strategy for selecting the parent block to mine when multiple branches of the blockchain had the same depth. Choosing the block with the lowest nonce limits divergences while ensuring identical behavior among miners.
- **Concurrent mining by multiple miners:** Handling multiple miners mining in parallel on the same blockchain raised synchronization issues. A key challenge was preventing a miner from continuing to search for a proof of work for a block that another miner had already solved and propagated. This would require frequently updating the local blockchain and an interruption mechanism for ongoing mining when a new valid block is received. Although considered, this feature could not be fully implemented within the project timeline.

5 Conclusion

This project led to the implementation of a simplified blockchain in Rust, thereby improving understanding of both the Rust language and the fundamental principles of blockchain, such as

block structure, Proof of Work (PoW), the genesis block, and the mining process.

All unit tests were successfully validated, and the code compiles without any errors or warnings.

6 Appendices

Compilation and execution

For the server:

```
cargo run --bin server
```

For the miner(s):

```
cargo run --bin miner mine
-d DIFFICULTY
-m MINER_NAME
```