Building Proof of Stake in Rust — 02 Proof of Stake

Jason Thye · Follow

7 min read · May 18, 2022

Photo by Xavier Foucrier on Unsplash

Overview

In this chapter we will implement the consensus to decide in a distributed way what are the block that should be next in the blockchain. There are many different consensuses to achieve this goal using different rules (read more here). The two most famous consensuses are Proof of Work (PoW) and Proof of Stake (PoS).

With Proof of Work the more computational power your node have the more chance you have to find the next block, nodes have to calculates a lot of hashes to find a block. This process is called mining and uses a lot of electricity. In order to avoid such a waste, Peercoin introduced the Proof of Stake consensus that gives more chance to find a block to the nodes that holds more coins. This process is called minting. The Proof of Stake algorithm implemented here are based on this article: https://blog.ethereum.org/2014/07/05/stake/ from Vitalik Buterin. Also, keep in mind that there are many different implementation of Proof of Stake.

This chapter also shows how to control the block generation interval by changing the difficulty.

It should be noted that we do not yet introduce transactions in this chapter. This means there is actually no incentive for the minters to generate a block. Generally in cryptocurrencies, the node is rewarded for finding a block, but this is not the case yet in our blockchain.

Difficulty and Proof of Stake Puzzle

We will add three proprieties to the block structure:

- `validator` the address of the node who is minting (finding a block)
- `signature` the encrypted hash after the validator signed on the block hash
- `difficulty` a number that will be used to keep the time interval between each block the same

Here is the current block structure:

```
impl Block {

    pub fn new(
        id: usize,
        previous_hash: String,
        timestamp: i64,
        txn: Vec<Transaction>,
        difficulty: u32,
        mut validator_wallet: Wallet,
    ) -> Self {

        info!("creating block...");

        let validator =
validator_wallet.get_public_key();
        let hash = block::calculate_hash(
            &id,
            &timestamp,
            &previous_hash,
            &txn,
            &validator,
            &difficulty,
        );
        let signature =
validator_wallet.sign(&hash);

        Self {
            id,
            hash,
            previous_hash,
            timestamp,
            txn,
            validator,
            signature,
            difficulty,
        }
    }

    ...
}
```

Also, the calculation of the hash needs to be updated with the new data:

```
    pub fn calculate_hash(
        id: &usize,
        timestamp: &i64,
        previous_hash: &str,
        txn: &Vec<Transaction>,
        validator: &String,
        difficulty: &u32,
    ) -> String {

        info!("calculating hash...");

        let hash = serde_json::json!({
            "id": id,
            "previous_hash": previous_hash,
            "transactions": txn,
            "timestamp": timestamp,
            "validator": validator,
            "difficulty": difficulty,
        });

        Util::hash(&hash.to_string())
    }
```

Now let's see the Proof of Stake puzzle that is used to decide if a block is found in pseudo code from Vitalik Buterin: https://blog.ethereum.org/2014/07/05/stake/

```
SHA256(prevhash + address + timestamp) <= 2^256 *
balance / diff
```

And below the implementation of it in Rust

```
use num_bigint::BigUint;
use sha256::digest;

pub fn is_staking_valid(
    balance: u64,
    difficulty: u32,
    timestamp: i64,
    previous_hash: &String,
    address: &String,
) -> bool {
    let base = BigUint::new(vec![2]);
    let big_balance_diff_mul = base.pow(256) *
balance as u32;
    let big_balance_diff = big_balance_diff_mul /
difficulty as u64;

    let data_str = format!("{}{}{}",
previous_hash, address, timestamp.to_string());
    let sha256_hash = digest(data_str);
    let decimal_staking_hash =
BigUint::parse_bytes(&sha256_hash.as_bytes(),
16).expect("Convert SHA256 hash to big number");

    decimal_staking_hash <= big_balance_diff
}
```

Since the puzzle compares very big number, we need to use a library that can handle these numbers

Also, as you can see the `SHA256()` function takes the `prevhash` that cannot change, the `address` that cannot change without changing the `balance` and the `timestamp` that changes only every second, that is why you can try to find a block every second which makes it consumming a lot less energy than proof of work.

Note that Proof of Stake has one big limitation: when you start the blockchain with every account has 0 coins, no one would be able to stake it's coin and find a block. There are different ways to solve this problem:

- Generate an amount of coins in the genesis block and distribute them manually to the nodes (or sell them)
- Use Proof of Stake consensus along with Proof of Work to give chance to the accounts without coins to succeed
- Simulate an increment of the balance in a limited number of blocks (what we can see in the code above)

Also, note that it's still possible for nodes to cheat by changing the timestamp. In order to avoid that some cryptocurrency adds checkpoints (which are centralized and sent by an authority).

Finding a block

As describe above, to find a block we need to try the puzzle every second. It is done with by the following code:

```
// main.rs

let (pos_mining_sender, mut pos_mining_rcv) =
mpsc::unbounded_channel();

async fn main() {

  ...

  let mut planner = periodic::Planner::new();
  planner.start();

  // Run every second
  planner.add(
      move ||
pos_mining_sender.send(true).expect("can send
mining event"),

periodic::Every::new(Duration::from_secs(1)),
  );

  loop {
    let evt = {
        select! {
            ...
            _ = pos_mining_rcv.recv() => {
                    Some(p2p::EventType::Mining)
                },
            ...
        }
    };

    if let Some(event) = evt {
      match event {

        ...

        p2p::EventType::Mining => {

          if let Some(block) =
swarm.behaviour_mut().blockchain.mine_block_by_st
ake() {

              swarm
                  .behaviour_mut()
```

```
                    .blockchain
                    .add_new_block(block.clone());

                info!("broadcasting new block");

                let json =
serde_json::to_string(&block).expect("can jsonify
request");
                swarm
                    .behaviour_mut()
                    .floodsub

.publish(p2p::BLOCK_TOPIC.clone(),
json.as_bytes());
            };
        }

        ...

      }
    }
  }
}

// blockchain.rs
pub fn mine_block_by_stake(&mut self) ->
Option<Block> {
    if self.mempool.transactions.is_empty() {
        // Skip mining because no transaction in
mempool
        return None;
    }

    let balance = self
        .stakes

.get_balance(&self.wallet.get_public_key())
        .clone();
    let difficulty = self.get_difficulty();

    info!("Mining new block with difficulty {}",
difficulty);

    let timestamp = Utc::now().timestamp();
    let previous_hash =
self.chain.last().unwrap().hash.clone();
    let address = self.wallet.get_public_key();

    if Blockchain::is_staking_valid(balance,
difficulty, timestamp, &previous_hash, &address)
{
        Some(self.create_block(timestamp))
    } else {
        None
    }
}
```

Consensus on the difficulty

We have now the means to find and verify the hash for a given difficulty, but how is the difficulty determined? There must be a way for the nodes to agree what the current difficulty is. For this we introduce some new rules that we use to calculate the current difficulty of the network.

Lets define the following new constants for the network:

- `BLOCK_GENERATION_INTERVAL_SECONDS` , defines how often a block should be found. (in Bitcoin this value is 10 minutes)
- `DIFFICULTY_ADJUSTMENT_INTERVAL_BLOCKS` , defines how often the difficulty should adjust to the increasing or decreasing network hashrate. (in Bitcoin this value is 2016 blocks)

We will set the block generation interval to 30s and difficulty adjustment to 2 blocks. These constants do not change over time and they are hard coded.

```
const BLOCK_GENERATION_INTERVAL_SECONDS: usize =
30;

const DIFFICULTY_ADJUSTMENT_INTERVAL_BLOCKS:
usize = 2;
```

Now we have the means to agree on a difficulty of the block. For every 10 blocks that is generated, we check if the time that took to generate those blocks are larger or smaller than the expected time. The expected time is calculated like this: `BLOCK_GENERATION_INTERVAL_SECONDS * DIFFICULTY_ADJUSTMENT_INTERVAL_BLOCKS`. The expected time represents the case where the hashrate matches exactly the current difficulty.

We either increase or decrease the difficulty by one if the time taken is at least two times greater or smaller than the expected difficulty. The difficulty adjustment is handled by the following code:

```
pub fn get_difficulty(&mut self) -> u32 {
    let last_block = self.chain.last().unwrap();
    if last_block.id %
DIFFICULTY_ADJUSTMENT_INTERVAL_BLOCKS == 0 &&
last_block.id != 0 {
        let prev_difficulty_block =
            &self.chain[self.chain.len() - 1 -
DIFFICULTY_ADJUSTMENT_INTERVAL_BLOCKS];

        let time_taken = last_block.timestamp -
prev_difficulty_block.timestamp;
        let time_expected =
            DIFFICULTY_ADJUSTMENT_INTERVAL_BLOCKS
* BLOCK_GENERATION_INTERVAL_SECONDS;
```

```
        if time_taken < (time_expected / 2) as
i64 {
            last_block.difficulty + 1
        } else if time_taken > (time_expected *
2) as i64 {
            if last_block.difficulty <= 1 {
                1
            } else {
                last_block.difficulty - 1
            }
        } else {
            last_block.difficulty
        }
    } else {
        last_block.difficulty
    }
}
```

Cumulative difficulty

In the part 1 version of the blockchain, we chose always the "longest" blockchain to be the valid. This must change now that difficulty is introduced. For now on the "correct" chain is not the "longest" chain, but the chain with the most cumulative difficulty. In other words, the correct chain is the chain which required most resources (hashRate * time) to produce.

To get the cumulative difficulty of a chain we calculate 2^difficulty for each block and take a sum of all those numbers. We have to use the 2^difficulty as we chose the difficulty to represent the number of zeros that must prefix the hash in binary format. For instance, if we compare the difficulties of 5 and 11, it requires 2^(11–5) = 2^6 times more work to find a block with latter difficulty.

This property is also known as “Nakamoto consensus” and it is one of the most important inventions Satoshi made, when s/he invented Bitcoin. In case of forks, minters must choose on which chain the they decide put their current resources (hashRate). As it is in the interest of the minters to produce such block that will be included in the blockchain, the minters are incentivized to eventually to choose the same chain.

Conclusions

The Proof of Stake consensus is a quite interesting alternative to the Proof of Work since it doesn’t use as much energy. Also there are different attacks that can be done, such as changing the timestamp against what this naivecoin is not protecting against.

Also, in the current implementation, the account balance is not calculated based on the blockchain but written and read from the chain itself (for simplification purpose since I was mostly interested about the puzzle itself). It is not good at all and should be solved!

Next (Transactions) >

Blockchain Rust Proof Of Stake