Building Proof of Stake in Rust — 01 - Blockchain

Jason Thye · Follow

7 min read · May 18, 2022

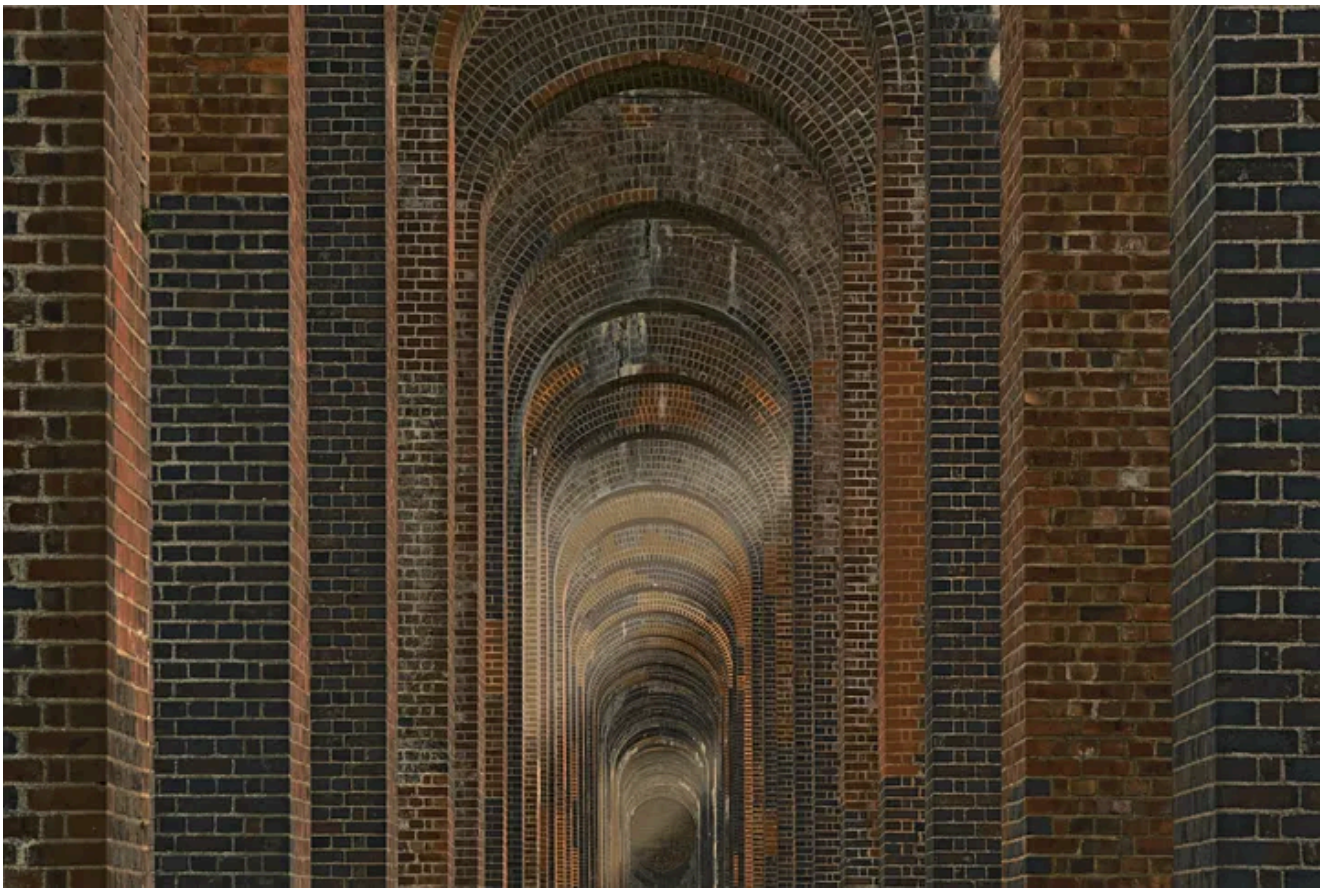

Photo by Viktor Forgacs on Unsplash

Overview

The basic concept of blockchain is quite simple: a distributed database that maintains a continuously growing list of ordered records. In this chapter we will implement toy version of such blockchain. At the end of the chapter we will have the following basic functionalities of blockchain:

- A defined block and blockchain structure
- Methods to add new blocks to the blockchain with arbitrary data
- Blockchain nodes that communicate and sync the blockchain with other nodes
- A simple CLI to control the node

Block structure

We will start by defining the block structure. Only the most essential properties are included at the block at this point.

- id : The height of the block in the blockchain
- hash: A sha256 hash taken from the content of the block
- previous_hash: A reference to the hash of the previous block
- timestamp: A timestamp
- txn: Transaction that is included in the block

The code for the block structure looks like the following:

```
pub struct Block {
    pub id: usize,
    pub hash: String,
    pub previous_hash: String,
    pub timestamp: i64,
    pub txn: Vec<Transaction>
}
```

Block hash

The block hash is one of the most important property of the block. The hash is calculated over all data of the block. This means that if anything in the block changes, the original hash is no longer valid. The block hash can also be thought as the unique identifier of the block. For instance, blocks with same index can appear, but they all have unique hashes.

We calculate the hash of the block using the following code:

```
use serde::{Deserialize, Serialize};
use sha256::digest_bytes;

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

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

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

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

pub fn hash(data: &String) -> String {
    digest_bytes(data.as_bytes())
}
```

It should be noted that the block hash has not yet nothing to do with mining, as there is no proof-of-work problem to solve. We use block hashes to preserve integrity of the block and to explicitly reference the previous block.

An important consequence of the properties `hash` and `previous_hash` is that a block can't be modified without changing the hash of every consecutive block.

This is demonstrated in the example below. If the data in block 44 is changed from "DESERT" to "STREET", all hashes of the consecutive blocks must be changed. This is because the `hash` of the block depends on the value of the `previous_hash` (among other things).

This is an especially important property when proof-of-work is introduced. The deeper the block is in the blockchain, the harder it is to modify it, since it would require modifications to every consecutive block.

Genesis block

Genesis block is the first block in the blockchain. It is the only block that has no previous_hash. We will hard code the genesis block to the source code:

```
pub fn genesis(wallet: Wallet) -> Block {
    info!("Creating genesis block...");
    Block::new(0, String::from("genesis"),
1648994652, vec![], wallet)
}
```

Generating a block

To generate a block we must know the hash of the previous block and create the rest of the required content (index, hash, data and timestamp). Block data is something that is provided by the end-user but the rest of the parameters will be generated using the following code:

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

        Self {
            id,
            hash,
            previous_hash,
            timestamp,
            txn,
            validator:
```

```
validator_wallet.get_public_key(),
            signature: signature,
            difficulty: difficulty,
        }
    }

  ...

}
```

Storing the blockchain

For now we will only use an in-memory array to store the blockchain. This means that the data will not be persisted when the node is terminated.

```
impl Blockchain {

    pub fn new(wallet: Wallet) -> Self {

        let genesis =
Blockchain::genesis(wallet.clone());

        Self {
            chain: vec![genesis],
            mempool: Mempool::new(),
            wallet: wallet,
            accounts: Account::new(),
            stakes: Stake::new(),
            validators: Validator::new(),
        }
    }

    ...

  }
```

Validating the integrity of blocks

At any given time we must be able to validate if a block or a chain of blocks are valid in terms of integrity. This is true especially when we receive new blocks from other nodes and must decide whether to accept them or not.

For a block to be valid the following must apply:

- The index of the block must be one number larger than the previous
- The `previous_hash` of the block match the `hash` of the previous block
- The `hash` of the block itself must be valid

This is demonstrated with the following code:

```
impl Blockchain {

  ...

  pub fn is_valid_block(&mut self, block: Block)
-> bool {

      let prev_block =
self.chain.last().unwrap();

      if block.previous_hash != prev_block.hash {

          warn!("block with id: {} has wrong
previous hash {} vs {} ",

          block.id, block.previous_hash,
prev_block.hash);
          return false;

      } else if block.hash !=
block::calculate_hash(
              &block.id,
```

```
                &block.timestamp,
                &block.previous_hash,
                &block.txn,
                &block.validator,
                &block.difficulty,
            )
        {
            warn!("block with id: {} has invalid
hash", block.id);
            return false;

        } else if prev_block.id + 1 != block.id {
            warn!("block with id: {} is not the
next block after the latest: {}", block.id,
prev_block.id);
            return false;

        } else if
!Block::verify_block_signature(&block) {
            warn!("block with id: {} has invalid
validator signature", block.id);
            return false;

        } else if !self.verify_leader(&block) {
            warn!("block with id: {} has invalid
validator", block.id);
            return false;

        }

        self.execute_txn(&block);

        info!("Add new block to current chain");

        self.chain.push(block);
        self.mempool.clear();

        true
    }

    ...

}
```

Now that we have a means to validate a single block we can move on to validate a full chain of blocks. We first check that the first block in the chain matches with the `genesisBlock`. After that we validate every consecutive block using the previously described methods. This is demostrated using the following code:

```
impl Blockchain {

  ...

  pub fn is_valid_chain(&mut self, chain:
&Vec<Block>) -> bool {

      if *chain.first().unwrap() !=
Blockchain::genesis(self.wallet.clone()) {
          return false;
      }

      for i in 0..chain.len() {

          if i == 0 {
              continue;
          };

          let block = &chain[i];
          let prev_block = &chain[i - 1];

          if prev_block.hash !=
block.previous_hash {

              warn!("block with id: {} has wrong
previous hash", block.id);
              return false;

          } else if prev_block.id + 1 != block.id
{

              warn!("block with id: {} is not the
next block after the latest: {}", block.id,
prev_block.id);
              return false;
```

```
            }
        }

        true
    }

    ...

}
```

Choosing the longest chain

There should always be only one explicit set of blocks in the chain at a given time. In case of conflicts (e.g. two nodes both generate block number 18) we choose the chain that has the longest number of blocks.

This is logic is implemented using the following code:

```
impl Blockchain {

    ...

    pub fn replace_chain(&mut self, chain:
&Vec<Block>) {

        if chain.len() <= self.chain.len() {

            warn!("Received chain is not longer
than the current chain");
            return;
        } else if !self.is_valid_chain(chain) {

            warn!("Received chain is invalid");
            return;
        }

        info!("Replacing current chain with new
chain");

        self.reset_state();
        self.execute_chain(chain);
```

```
            self.chain = chain.clone();
        }

        ...
    }
```

Communicating with other nodes

An essential part of a node is to share and sync the blockchain with other nodes. The following rules are used to keep the network in sync.

- When a node generates a new block, it broadcasts it to the network
- When a node connects to a new peer it querys for the latest chain
- When a node encounters a chain that has an id larger than the current known block, it replaces its current chain to the one that's longer.

We will use libp2p for the peer-to-peer communication. We'll use the FloodSub protocol, a simple publish/subscribe protocol, for communication between the nodes. We'll also use mDNS, which is a protocol for discovering other peers on the local network.

Controlling the node

We also initialize a buffered reader on stdin so we can read incoming commands from the user.

For user input, we have commands below:

- ls p — lists all peers
- ls c — prints the local blockchain
- create wallet — generate keypair
- ls wallet — show node's wallet address
- ls bal — show node's wallet balance
- ls validator — show validators
- ls stakes — show stakes
- set wallet — update the node's wallet using the supplied keypair

eg. set wallet generated_keypair

- create txn

eg. create txn receiver_address amount txn_type

txn_type:

- txn
- stake
- validator

```
async fn main() {
    ...
```

```
  loop {
    let evt = {
        select! {
            line = stdin.next_line() =>
Some(p2p::EventType::Input(line.expect("can get
line").expect("can read line from stdin"))),
            ...
        }
    };

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

            ...

            p2p::EventType::Input(line) => match
line.as_str() {
              "ls p" =>
p2p::handle_print_peers(&swarm),
              "create wallet" =>
Wallet::generate_wallet(),
              "ls wallet" =>
p2p::handle_print_wallet(&mut swarm),
              "ls c" =>
p2p::handle_print_chain(&swarm),
              "ls bal" =>
p2p::handle_print_balance(&swarm),
              "ls validator" =>
p2p::handle_print_validator(&swarm),
              "ls stakes" =>
p2p::handle_print_stake(&swarm),
              cmd if cmd.starts_with("set
wallet") => p2p::handle_set_wallet(cmd, &mut
swarm),
              cmd if cmd.starts_with("create
txn") => p2p::handle_create_txn(cmd, &mut swarm),
                _ => error!("unknown command"),
                },
            }
        }
    }

  ...
```

```
}
```

Way to start the node and print the local chain using command line:

```
# Start the node
> RUST_LOG=info cargo run
> ls c
```

Conclusions

This example is for now just a toy “general purpose” blockchain. Moreover, this chapter shows how some of the basic principles of blockchain can be implemented in quite a simple way. In the next chapter, we will add the proof-of-stake algorithm (minting) to this example.

Next (Proof of Stake) >

Blockchain Rust Proof Of Stake