Building Proof of Stake in Rust — 05 Transaction Relaying

Jason Thye · Follow

4 min read · May 18, 2022

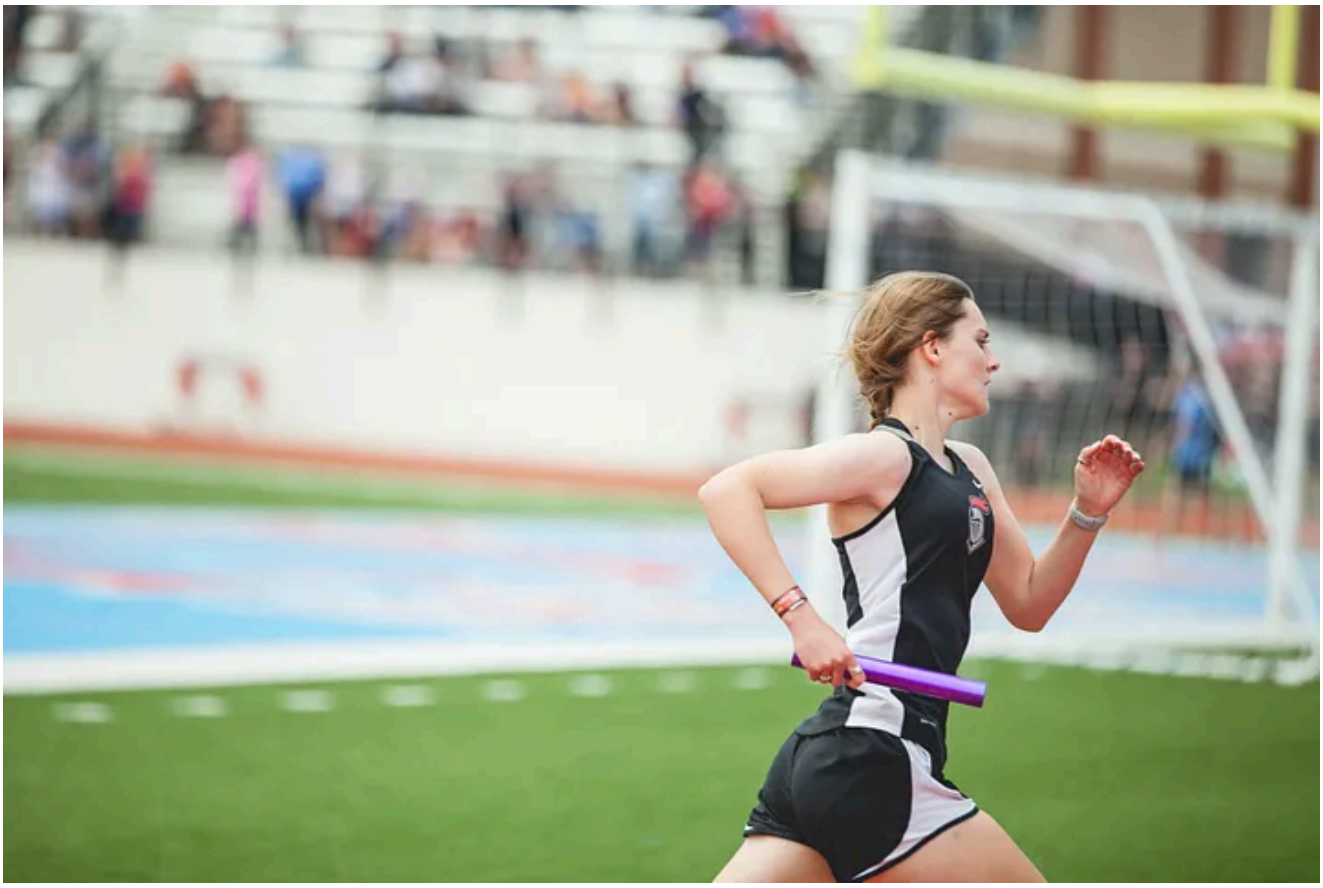

Photo by Zach Lucero on Unsplash

Overview

In this chapter we will implement the relaying of such transactions, that are not yet included in the blockchain. In bitcoin, these transaction are also known as “unconfirmed transactions”. Typically, when someone wants to include a transaction to the blockchain (send coins to some address) he broadcasts the transaction to the network and hopefully some node will mint the transaction to the blockchain.

This feature is very important for a working cryptocurrency, since it means you don’t need to mint a block yourself, in order to include a transaction to the blockchain.

As a consequence, the nodes will now share two types of data when they communicate with each other:

- the state of the blockchain (the blocks and transactions that are included to the blockchain)
- unconfirmed transactions (the transactions that are not yet included in the blockchain)

Transaction pool

We will store our unconfirmed transactions in a new entity called “transaction pool” (also known as “mempool” in bitcoin). Transaction pool is a structure that contains all of the “unconfirmed transactions” our node know of. In this simple implementation we will just use a list.

```
pub struct Mempool {
    pub transactions: Vec<Transaction>,
```

```
}
```

We will also introduce a new command to our node: “create txn”. This method creates a transaction to our local transaction pool. For now on we will use this method as the interface when we want to include a new transaction to the blockchain.

```
// main.rs

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() {
            ...
            cmd if cmd.starts_with("create txn")
=> p2p::handle_create_txn(cmd, &mut swarm),
          },
      }
  }
}
```

We create the transaction just like we did in part 4. We just add the created transaction to the pool instead of instantly trying to mint a block:

```
// Mempool.rs
pub fn add_transaction(&mut self, txn:
Transaction) {
    self.transactions.push(txn);
}
```

Broadcasting

The whole point of the unconfirmed transactions are that they will spread throughout the network and eventually some node will mint the transaction to the blockchain. To handle this we will introduce the following simple rule for the networking of unconfirmed transactions:

- When a node receives an unconfirmed transaction it has not seen before, it will broadcast the transaction to all peers.
- When a node first connects to another node, it will query for the transaction pool of that node. We will add three new topics to serve this purpose: `CHAIN_TOPIC`, `BLOCK_TOPIC` and `TXN_TOPIC`.

We'll use the FloodSub protocol, a simple publish/subscribe protocol, for communication between the nodes. Topics are basically "channels" to subscribe to. We can subscribe to "chains" and use them to send our local blockchain to other nodes and to receive theirs. The same is true for "blocks", which we'll use to broadcast and receive new blocks.

```
pub static CHAIN_TOPIC: Lazy<Topic> =
Lazy::new(|| Topic::new("chains"));
```

```
pub static BLOCK_TOPIC: Lazy<Topic> =
Lazy::new(|| Topic::new("blocks"));

pub static TXN_TOPIC: Lazy<Topic> = Lazy::new(||
Topic::new("transactions"));
```

To implement the described transaction broadcasting logic, we add code to handle the `TXN_TOPIC` channel. Whenever, we receive unconfirmed transactions, we try to add those to our transaction pool. If we manage to add a transaction to our pool, it means that the transaction is valid and our node has not seen the transaction before. In this case we broadcast our own transaction pool to all peers.

```
...
  else if let Ok(txn) = serde_json::from_slice::
<Transaction>(&msg.data) {
      info!("received new transaction from {}",
msg.source.to_string());

      if !self.blockchain.txn_exist(&txn) {
          info!("relaying new transaction");
          let json =
serde_json::to_string(&txn).expect("can jsonify
request");


self.floodsub.publish(TXN_TOPIC.clone(),
json.as_bytes());
          self.blockchain.add_txn(txn);
      }
  }

...
```

Validating received unconfirmed transactions

As the peers can send us any kind of transactions, we must validate the transactions before we can add them to the transaction pool. All of the existing transaction validation rules apply. For instance, the transaction must be correctly formatted, and the transaction inputs, outputs and signatures must match.

In addition to the existing rules, we add a new rule: a transaction cannot be added to the pool if any of the transaction inputs are already found in the existing transaction pool. This new rule is embodied in the following code:

```
// Blockchain.rs

pub fn add_txn(&mut self, txn: Transaction) {
    self.mempool.add_transaction(txn)
}

pub fn txn_exist(&mut self, txn: &Transaction) ->
bool {
    self.mempool.transaction_exists(txn)
}

// Mempool.rs
pub fn add_transaction(&mut self, txn:
Transaction) {
    if Transaction::verify_txn(&txn) {
        self.transactions.push(txn);
    } else {
        warn!("Failed adding to mempool: Invalid
transaction.");
    }
}

pub fn transaction_exists(&mut self, txn:
&Transaction) -> bool {
    self.transactions.contains(txn)
}
```

There is no explicit way to remove a transaction from the transaction pool. The transaction pool will however be updated each time a new block is found.

From transaction pool to blockchain

Let's next implement a way for the unconfirmed transaction to find its way from the local transaction pool to a block minted by the same node. This is simple: when a node starts to mint a block, it will include the transactions from the transaction pool to the new block candidate.

```
pub fn create_block(&mut self, timestamp: i64) ->
Block {
    info!("Creating new block...");

    Block::new(
        self.chain.len(),
        self.chain.last().unwrap().hash.clone(),
        timestamp,
        self.mempool.transactions.clone(),
        self.get_difficulty(),
        self.wallet.clone(),
    )
}
```

As the transactions are already validated, before they are added to the pool, we are not doing any further validations at this points.

Updating the transaction pool

As new blocks with transactions are minted to the blockchain, we must revalidate the transaction pool every time a new block is found.

The transaction pool will be updated with the following code:

```
pub fn add_new_block(&mut self, block: Block) {
    self.execute_txn(&block);
    info!("Add new block to current chain");
    self.chain.push(block);
    self.mempool.clear();
}
```

Conclusions

We can now include transactions to the blockchain without actually having to mint the blocks themselves. There is incentive for the nodes to include a received transaction to the block as we have implemented the concept of transaction fees.

Find the full source code here: https://github.com/emcthye/Proof-of-Stake-in-Rust

Blockchain Rust Proof Of Stake

Follow

Written by Jason Thye