Building Proof of Stake in Rust — 04 Wallet

Jason Thye · Follow

3 min read · May 18, 2022

Photo by Konstantin Evdokimov on Unsplash

Overview

The goal of the wallet is to create a more abstract interface for the end user.

The end user must be able to

- Create a new wallet (private key in this case)
- View the balance of his wallet
- Send coins to other addresses

All of the above must work so that the end user must not need to understand how txn_input or txn_output work. Just like in e.g. Bitcoin: you send coins to addresses and publish your own address where other people can send coins.

Generating EdDSA (ed25519) keypair

In this tutorial we will use the simplest possible way to handle the wallet generation and storing: We will generate a keypair and print it on the command line.

First, we need to generate a Keypair, which includes both public and secret halves of an asymmetric key. To do so, we need a cryptographically secure pseudorandom number generator (CSPRNG). For this example, we'll use the operating system's builtin PRNG:

```
use ed25519_dalek::{Keypair, Signer};
use rand::rngs::OsRng;

pub fn generate_wallet() {
    let mut csprng = OsRng {};
    let keypair = Keypair::generate(&mut csprng);
    let pub_key =
hex::encode(keypair.public.to_bytes());
    println!("Your Public Key {}", pub_key);
    println!("Your Key Pair {:?}",
hex::encode(keypair.to_bytes()));
}
```

And as said, the public key (address) can be calculated from the private key.

```
pub fn get_public_key(&mut self) -> String {

hex::encode(Wallet::get_keypair(&self.keyPair).pu
blic.as_bytes())
}
```

Wallet balance

Getting the balance for a given address is quite simple: you just retrive it using the wallet address:

```
// Blockchain.rs
pub fn get_balance(&mut self, public_key:
&String) -> &f64 {
    self.accounts.get_balance(public_key)
}
```

As demonstrated in the code, the private key is not needed to query the balance of the address. This consequently means that anyone can solve the balance of a given address.

Using the wallet

Let’s also add a meaningful controlling endpoint to generate transaction.

```
// 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() {
            "ls p" =>
p2p::handle_print_peers(&swarm),
            "create wallet" =>
Wallet::generate_wallet(),
            "ls wallet" =>
p2p::handle_print_wallet(&mut swarm),
            ...
            cmd if cmd.starts_with("create txn")
=> p2p::handle_create_txn(cmd, &mut swarm),
              _ => error!("unknown command"),
              },
          }
    }
}

// p2p.rs

pub fn handle_create_txn(cmd: &str, swarm: &mut
Swarm<AppBehaviour>) {
    if let Some(data) = cmd.strip_prefix("create
txn") {
        let arg: Vec<&str> =
data.split_whitespace().collect();

        let to = arg.get(0).expect("No receipient
found").to_string();
        let amount = arg
            .get(1)
            .expect("No amount found")
            .to_string()
            .parse::<f64>()
            .expect("Convert amount string to
float");
        let category = arg.get(2).expect("No
txntype found").to_string();

        let txn_type = match category.as_str() {
            "txn" =>
crate::transaction::TransactionType::TRANSACTION,
            "stake" =>
crate::transaction::TransactionType::STAKE,
            "validator" =>
crate::transaction::TransactionType::VALIDATOR,
            _ =>
crate::transaction::TransactionType::TRANSACTION,
        };

        let behaviour = swarm.behaviour_mut();

        let mut wallet =
behaviour.blockchain.wallet.clone();

        if amount + transaction::TRANSACTION_FEE
            >
*behaviour.blockchain.get_balance(&wallet.get_pub
lic_key())
        {
            warn!("Wallet has insufficient
amount");
            return;
        }

        match Blockchain::create_txn(&mut wallet,
to, amount, txn_type) {
            Ok(txn) => {
                let json =
serde_json::to_string(&txn).expect("can jsonify
request");
                info!("Adding new transaction to
mempool");

behaviour.blockchain.mempool.add_transaction(txn)
;
                info!("Broadcasting new
transaction");
                behaviour
                    .floodsub
                    .publish(TXN_TOPIC.clone(),
json.as_bytes());
            }
            Err(_) => {
                warn!("Failed to create
transaction: Unable to serialized transactions
into json");
            }
        };
    }
}
```

As it is shown, the end user must only provide the address and the amount of coins for the node. The node will calculate the rest using the wallet.

Conclusions

We just implemented a wallet with simple transaction generation. Also, the only way to include a desired transaction in the blockchain is to create the block yourself. The nodes do not exchange information about transactions that are not yet included in the blockchain. This will be addressed in the next chapter.

Next (Transaction relaying) >