Building Proof of Stake in Rust — 03 Transactions

Jason Thye · Follow

4 min read · May 18, 2022

Overview

In this chapter we will introduce the concept of transactions. With this modification, we actually shift from our project from a “general purpose” blockchain to a cryptocurrency. As a result, we can send coins to addresses if we can show a proof that we own them in the first place.

To enable all this, a lot of new concepts must presented. This includes public-key cryptography, signatures and transactions inputs and outputs.

Public-key cryptography and signatures

In Public-key cryptography you have a keypair: a secret key and a public key. The public key can be derived from the secret key, but the secret key cannot be derived from the public key. The public key (as the name implies) can be shared safely to anyone.

Any messages can be signed using the private key to create a signature. With this signature and the corresponding public key, anyone can verify that the signature is produced by the private key the that matches the public key.

We will use a library called elliptic for the public-key cryptography, which uses elliptic curves. (= EDDSA)

Conclusively, two different cryptographic functions are used for different purposes in the cryptocurrency:

- Hash function (SHA256) for the Proof-of-work mining (The hash is also used to preserve block integrity)
- Public-key cryptography (EDDSA) for transactions (we'll be implementing in this chapter)

Private-keys and public keys (in EDDSA)

A valid private key is any random 32 byte string, eg.

```
5ae5066dd048ffb8f8628c44324e63c7b8782a026009a85a96935acb4921abbc
```

A valid public key is any random 32 byte string, e.g

```
5aede624154386ca358af195e13a46981b917ee8279f30a67d7a211a3d3e7243
```

The private key can be join after from the public key to form a keypair. This keypair can be use to sign a message and a signature is created on that message. Anyone else, given the public half of the keypair can easily verify if this a valid signature on that message.

Transactions overview

Before writing any code, let's get an overview about the structure of transactions. Transactions consists of two components: inputs and outputs.

Transaction outputs

Transaction outputs (txn_output) consists of an address and an amount of coins. The address is an EDDSA public-key. This means that the user having the private-key of the referenced public-key (to) will be able to access the coins.

```
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct TransactionOutput {
    pub to: String,
    pub amount: f64,
    pub fee: f64,
}

impl TransactionOutput {
    pub fn new(to: String, amount: f64, fee: f64)
-> Self {
        Self {
            to: to,
            amount: amount,
            fee: fee,
        }
    }
}
```

Transaction inputs

Transaction inputs (txn_input) provide the information "where" the coins are coming from. The signature is generated by signing the hash of transaction outputs (txn_output) using the sender wallet. Later on the signature can be verify using the from field, and the hash of transaction outputs. It gives proof that only the

user, that has the private-key of the referred public-key (from) could have created the transaction.

```
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct TransactionInput {
    pub timestamp: i64,
    pub from: String,
    pub signature: String,
}

impl TransactionInput {
    pub fn new(sender_wallet: &mut Wallet,
txn_output: &String) -> Self {
        Self {
            timestamp: Utc::now().timestamp(),
            from: sender_wallet.get_public_key(),
            signature:
sender_wallet.sign(txn_output),
        }
    }
}
```

It should be noted that the txn_input contains only the signature (created by the private-key), never the private-key itself. The blockchain contains public-keys and signatures, never private-keys.

Transaction structure

The transactions structure itself is quite simple as we have now defined txn_input and txn_output.

```
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct Transaction {
    pub id: Uuid,
    pub txn_type: TransactionType,
```

```
    pub txn_input: TransactionInput,
    pub txn_output: TransactionOutput,
}
```

Transaction signatures

It is important that the contents of the transaction cannot be altered, after it has been signed. As the transactions are public, anyone can access to the transactions, even before they are included in the blockchain.

When signing the transaction, only the transaction output will be signed. If any of the contents in the transactions output is modified, the signature must change, making the transaction invalid.

```
impl Transaction {

    pub fn new(
        sender_wallet: &mut Wallet,
        to: String,
        amount: f64,
        txn_type: TransactionType,
    ) -> Self {

        let txn_output =
TransactionOutput::new(to, amount,
TRANSACTION_FEE);
        let serialized =
serde_json::to_string(&txn_output).unwrap();
        let txn_input =
TransactionInput::new(sender_wallet,
&serialized);

        Self {
            id: Util::id(),
            txn_type: txn_type,
            txn_output: txn_output,
```

```
                txn_input: txn_input,
            }
        }
        ...
    }
```

Let's try to understand what happens if someone tries to modify the transaction:

1. Attacker runs a node and receives a transaction with content: "send 10 coins from address AAA to BBB" with signature 1ec82..
2. The attacker changes the receiver address to CCC and relays it forward in the network. Now the content of the transaction is "send 10 coins from address AAA to CCC"
3. However, as the receiver address is changed, the txId is not valid anymore. A new valid txId would be c937f...
4. If the serialized transaction output is set to the new value, the signature is not valid. The signature matches only with the original signature 1ec82..
5. The modified transaction will not be accepted by other nodes, since either way, it is invalid.

Transactions validation

We can now finally lay out the rules what makes a transaction valid:

Valid Transaction

The signatures in the txIns must be valid and the referenced outputs must have not been spent.

```
pub fn verify_txn(txn: &Transaction) ->
Result<bool, VerifyTxnError> {
    let txn_message = match
serde_json::to_string(&txn.txn_output) {
        Ok(txn_message) => txn_message,
        Err(e) => return
Err(VerifyTxnError::DecodeJsonErr(e)),
    };

    let result = match Util::verify_signature(
        &txn.txn_input.from,
        &txn_message,
        &txn.txn_input.signature,
    ) {
        Ok(result) => result,
        Err(e) => match e {
            VerifySigErr::DecodeStrError(_) =>
false,
            VerifySigErr::DecodeHexError(_) =>
false,
        },
    };

    Ok(result)
}

pub fn verify_signature(
    from_public_key: &String,
    message: &String,
    from_signature: &String,
    ) -> Result<bool, VerifySigErr> {

    let public_key =
hex::decode(from_public_key)?;
    let dalek_public_key =
PublicKey::from_bytes(&public_key)?;

    let signature = hex::decode(from_signature)?;
    let dalek_signature =
```

```
&Signature::from_bytes(&signature)?;

    Ok(dalek_public_key
        .verify(message.as_bytes(),
dalek_signature)
        .is_ok())
}
```

Conclusions

We included the concept of transactions to the blockchain. However, creating transactions is still very difficult. We must manually create the inputs and outputs of the transactions and sign them using our private keys. This will change when we introduce wallets in the next chapter.

Next (Wallet) >

Blockchain Rust Proof Of Stake

Written by Jason Thye

12 Followers

Writings about blockchain technology & Web3 application