PUT Blockchain Implementation

Eng. K. Kłodziński, Eng. K. Baran

June 15, 2023

Contents

Part I
Introduction

Blockchain technologies were firstly proposed by Satoshi Nakamoto to counter the problem of double spending with digital currencies. This lead to the implementation of Bitcoin as a first Cryptocurrency. The world of cryptocurrencies and blockchain evolved after that to include different consensus mechanisms, verification processes and more. It is worth to say, that blockchain can be implemented not only in cryptocurrencies, but also in smart contracts and voting systems.

We designed our own cryptocurrency blandly following this requirements:

1. The blockchain should be implemented using a programming language of your choice with no restrictions.
2. The blockchain should support the addition of new blocks containing transaction data, ensuring secure storage.
3. Cryptographic mechanisms such as hashing and digital signatures should be employed to ensure the integrity and security of the data. External libraries like OpenSSL can be used for cryptographic functionalities.
4. Consensus mechanisms such as proof-of-work or proof-of-stake should be implemented to ensure the validity of the blocks.

Our project also took into account the following optional requirements:

1. The blockchain should include a user interface allowing users to add and view transactions, as well as to observe the current state of the blockchain.
2. Basic security features such as authentication and access control should be incorporated to prevent unauthorized access to the blockchain.
3. The blockchain should be thoroughly tested and evaluated for both its performance and security.

Lastly, this report will include information about the implementation we used, the technologies involved and design decisions. We will also describe somewhat in depth how to implementation should work and some problems we encountered along the way.

Part II
The implementation

Here we will describe the implementation, technologies used and problems we encountered along the way. Function definitions and descriptions are in part III.

1 Technologies

We implemented our cryptocurrency, which from now on will be called **PUT coins**, in **C++** with the usage of sockets. This choice was driven by previous knowledge of the programming language and efficiency of socket programming as well as availability of OpenSSL libraries.

The code begun on a Windows machine, but was later ported to work on GNU/Linux, which provides the crypto library out of the box in many distributions.

2 Design choices

We choose a centralized approach. This choice was driven by the nature of socket programming in C/C++. Apart from the server we decided to split clients and miners. The server is responsible for syncing the client with current blocks and receiving transactions from them. These transactions are to be sent to miners, which have been synced upon connecting. Once a new block is able to be mined, the newest block should be chosen to be added to the chain. Both the server and the user have a copy of the blockchain, while the miner only have a list of transactions on the ledger and last hashes.

We decided to limit ledgers to 10 transactions. Each transaction is composed of sender and receiver IDs, transaction amount and a signature of the transaction. This signature is achieved with an RSA 2048 bit key. This key has to be generated separately and has to be in PEM format. An example for generating the key is given in the Github repository. The public RSA key is used to identify the user and is also used by the miner. This way, the server knows who to award the new mined coins.

To prevent spending money we do not have, the client checks transactions on the blockchain and syncs the current coins with it. Since we decided to not give it transactions on ledgers, the client keeps an unconfirmed money amount. Once a new block is available, the current coins will be synced accordingly.

We choose a proof-of-work consensus mechanism with SHA256 as the hashing algorithm. For each block, the difficulty is always set to 1, which means one leading zero byte in the block hash.

For the mining process, proof-of-work is an integer with is incremented until the difficulty is satisfied. Since we acknowledged that no miners could be connected to the server, the latter mines a block itself when enough transactions are available.

3 Problems and tests

The main problem we encountered during the implementation was correctly syncing the clients and miners with the server. First we though of having the user mine blocks when available. This posed the problem of deciding who mined the block first, what to do with excess transactions and many more. Since we did not want to loose blocks or transactions and neither have clients with different block, we split the mining part.

Another problem we encountered was transaction verification. The RSA public key is sent by the user to the server. This key is received correctly as plaintext. Unfortunately we were from there not able to load the key locally. Attempting to send the loaded key as bytes resulted in the same problem.

During design we tested if everything worked correctly, which unfortunately was not the case. We were able to experience not only Segmentation Fault errors, but also IOT operation errors.

4 How it should work

First of all a public and private RSA 2048 key should be generated and saved in PEM format. The sever can then be run from the command line.

4.1 Client

The client connects to the server, sends it public key and received its user ID for the transactions. After that the user is synced with the latest blocks by sending to the server the current block hash it has and receiving the remaining ones. The transactions on the blocks are checked to calculate user coins. The user is then displayed it current PUT coins amount.

The user can choose to add a transaction. If so, he will be asked for a received ID and an amount. The amount has to be positive and not greater that it current unconfirmed coins. If everything is correct, a transaction is generated and sent to the server. Unconfirmed coins are updated accordingly. The user then sends its latest block hash to the server to see if he is up to date. If not,

he will receive the latest block available and sync it current coins.

The user can also choose to do nothing and disconnect.

4.2 Miner

The miner sends to the server the client public RSA key. It then receives the latest transactions on the ledger, the hash of the last block and waits for new transactions. If enough are available it sends the new block to the server along with a timestamp.

4.3 Server

The server does all the hard work. To distinguish clients from miners, when a new connection is initialised it receives an identity; *client* for clients and *minerr* for miners. It syncs them accordingly to their own needs.

When a new transaction is received by the client, the transaction is added to the current ledger and sent to the miners if available. After that it syncs the client with the newest block if available.

If enough transactions are available a new block is calculated. If a miner sends a block to the server. It is checked with other timestamps to see which one was the first. The server one is not taken into account. A reward should be sent to the client that calculated the block first.

Part III
Documentation

This section contains a brief description of what most functions do, their parameters and return types.

5 block_chain.hpp

This files contains the main functions relative to the blockchain implementation. Transaction structure and block structure definition, functions to create transactions, add them to the chain, sign transaction and many more.

This file is namespaced as *put::blockchain::block_chain*.

5.1 Constants

- **BLOCK_SIZE** = 10

5.2 struct transaction_t

- *uint16_t transaction_id* – ID of the transaction
- *uint64_t sender_id* – ID of the sender
- *uint64_t recipient_id* – ID of the receiver
- *uint64_t transaction_amount* – PUT coins amount
- *char signature[256]* = {*0*} – transaction signature

5.3 struct transaction_block_t

- *unsigned char previous_block_hash[SHA256_DIGEST_LENGTH]* – SHA256 of the previous block
- *transaction_t transactions[BLOCK_SIZE]* – transaction ledger
- *uint64_t proof_of_work = NULL* – proof-of-work for the current block

5.4 class block_chain

5.4.1 Private fields

- *RSA *private_key = nullptr* – RSA context with the private key file
- *uint64_t last_transaction_id* – ID of the last transaction which will be later increased

- *std::vector¡transaction_t¿ transactions* – transactions on the current ledger appended later when new block will be generated
- *transaction_block_t newest_transaction_block* – last generated transaction block

5.4.2 Public fields

block_chain(uint64_t last_transaction_id) Constructor without private key. Necessary for blockchain miners as they do not need to add transactions to the ledger. Index of last transaction is used for incrementing transaction id when creating transactions.

- *last_transaction_id* – index of last transaction

block_chain(std::string private_key_file_name, uint64_t last_transaction_id) Constructor with private RSA key in *PEM* format. This key is used to sign transactions. Index of last transaction is used for incrementing transaction id when creating transactions.

- *private_key_file_name* – string with path to private key file. Can be relative.
- *last_transaction_id* – index of last transaction

void set_private_key(std::string private_key_file_name)
This function reads the RSA private key in PEM format and sets the current RSA context with this key.

- *private_key_file_name* – string with path to private key file. Can be relative.

transaction_t add_transaction(uint64_t sender_id, uint64_t recipient_id, uint64_t transaction_amount)
Adds a transaction to the current ledger. Returns error if the ledger does not have space for new transactions. A new block should be manually generated to clear the ledger. Transactions is signed with the RSA private key.

- *sender_id* – the ID of the sender
- *recipient_id* – the ID of the recipient
- *transaction_amount* – the amount of PUT coins to send

transaction_t create_transaction(uint64_t sender_id, uint64_t recipient_id, uint64_t transaction_amount)
Creates a transaction but does not add it to the current ledger. Increases the transaction ID.

- *sender_id* – the ID of the sender

- *recipient_id* – the ID of the recipient
- *transaction_amount* – the amount of PUT coins to send

void add_transaction(transaction_t transaction)
Adds a previously generated transaction to the current block.

- *transaction* – the transaction to add to the ledger

transaction_block_t create_transaction_block(unsigned char previous_block_hash[SHA256_DIGEST_LENGTH])
Creates a new transaction block if enough transactions are on the ledger. It flushes the current transaction ledger and sets this block as the newest block. It requires the has of the previous block as a parameter. This can be taken from *get_transaction_hash*.

- *previous_block_hash[SHA256_DIGEST_LENGTH]* – hash of the previous transaction block

Returns the generated block.

unsigned char *get_transaction_block_hash()
Gets the last transaction block hash. It is generated anew every call as this has is not stored in the block.

void get_transaction_block_hash(unsigned char * transaction_hash)
Gets the last transaction block and saves it into *transaction_hash* memory block.

- *transaction_hash* – pointer to beginning of char array