Blockchain Lab

Julien Romero, Nedeljko Radulovic, Nicoleta Preda

January 2020

1 Introduction

In this lab, you will create a small example of a blockchain to play with. First, the goal is to implement the operations that will allow you to construct the blockchain. Secondly, you should implement a proof-of-work algorithm. Finally, the third requirement is to verify if a blockchain is correct or not.

You are expected to provide the code you wrote during the lab and a report with your answers to the questions, in a pdf. **Give reasons for your answers.** The submission takes place on the Moodle of the lecture.

This work is **individual**. Any plagiarism will be penalized with a grade of 0 (whether you copy or you are being copied).

2 Blockchain

As the name suggests, a blockchain is a linked list of blocks. A block contains a set of transactions and metadata information. The metadata is generally called header. Figure 1 represents the abstraction of a block. That block stands for the initial block (its index is 0 and the previous hash has a default value 0). The timestamp is the time at the creation of the block and the nonce is the value computed by the proof-of-work algorithm (see later). Two transactions are represented: Admin (this is a default user who has virtually infinite credit) sends 10 crypto-currency to Alice and to Bob.

We notice that blocks need to be serialized as they need to be stored on a disk (to avoid loosing the data) and because they need to be sent through the network to other peers. In this implementation, we use JSON as a file format for representing the encoding.

```
{"header": {"index": 0,
            "nonce": 32013,
            "previous_hash": 0,
            "timestamp": 1547396822.9447556
            },
 "transactions":
 [{"amount":10, "index":0, "receiver":"alice", "sender":"admin"},
  {"amount":10, "index":1, "receiver":"bob", "sender":"admin"}]
}
```

2.1 Header Operations

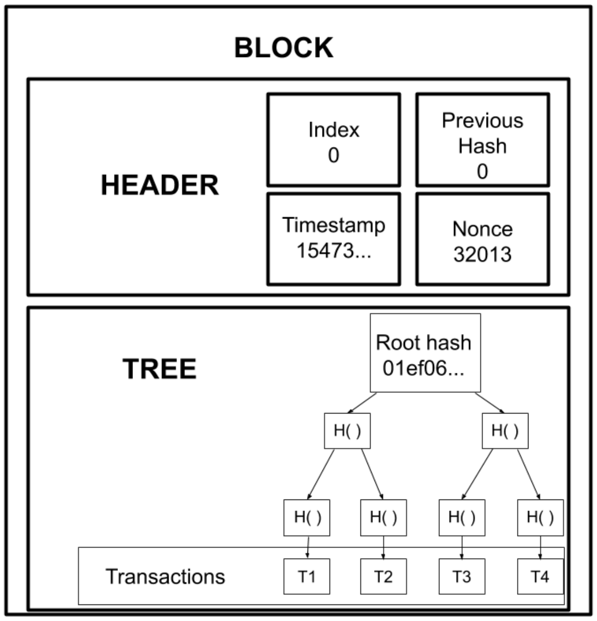

Figure 1: A block

The header contains information about the current block. Namely, we have the following fields:

- **index** : indicates the position of the block in the chain.
- **previous hash** : the hash of the previous block in the chain.
- **time stamp** : the time when the block is created.
- **nonce** : the proof used for the proof-of-work algorithm

Exercise 1. Where is the address to the previous block stored?

Exercise 2. In the file *block_header.py*, implement the functions **__init__** by storing internally the attributes, the function **to_dict** which puts the internal fields into a dictionary (the name of the fields are *index*, *previous_hash*, *timestamp* and *nonce*) and the function **to_json** (you should use the function *json.dumps* with the parameter sort_keys=True). The JSON representation will be used later by the hashing function in the proof-of-work algorithm. Why is it important to sort the keys?

```python
class BlockHeader(object):

    def __init__(self, index, previous_hash, timestamp, nonce):
        # Store internally
        pass

    def to_dict(self):
        # Transform object into a dictionary.
        # Use as field names the name of the variables
        pass

    def to_json(self):
        # Output: JSON string
        # Use the function json.dumps with the option sort_key=True
        #   to make the representation unique.
        pass

    def set_nonce(self, new_nonce):
        # Set the nonce value
        pass

    def get_hash(self):
        # Use hashlib to hash the block using sha256
        # Use hexdigest to get a string result
        pass
```

Example JSON serialization of a block header is:

```
{   "index": 0,
    "nonce": 32013,
    "previous\_hash": 0,
    "timestamp": 1547396822.9447556
}
```

Exercise 3. Which of the header's fields is unnecessary and redundant?

Exercise 4. In the file *block_reader*, implement the method **read_header** which returns a header from a dictionary.

Exercise 5. Compare our header with the header of the Bitcoin. Which fields are missing? Describe briefly what they are used for.

2.2 Transaction Operations

A transaction is composed of three main components: the *sender*, the *receiver* and the *amount* of money exchanged. In general, the *sender* and the *receiver* are identified by a public key. In this lab, for readability, we are simply going to use names.

Exercise 6. In the class *Transaction*, implement the functions **__init__** by storing internally the attributes and the function **to_dict** which puts the internal fields into a dictionary (the name of the fields are *sender* for the emitter of the transaction, *receiver* for the recipient of the transaction, *amount* for the amount of money exchanged and *index* which indicates the number of the transaction).

```python
class Transaction(object):

    def __init__(self, index, sender, receiver, amount):
        # Store internally
        pass

    def to_dict(self):
        # Transform object into a dictionary for future
            transformation in JSON
        # The names of the fields are the name of the variables
        pass
```

Exercise 7. Which field is missing to make the transaction secured? Discuss a possible attack.

Exercise 8. How is a person generally identified on a blockchain? How does a person prove ownership?

Exercise 9. In the file **block_reader**, implement the method **read_transaction** which creates a Transaction from a dictionary. This will be useful later when we will read blocks from files.

2.3 Block Operations

We now put everything together (header + transaction operations).

Exercise 10. In the file *merkle_tree* complete the function *create_merkle_tree* to implement the Merkle algorithm to store the transactions in the Merkle tree structure (for simplicity, store separately internal nodes and leaves in the tree structure, since the leaves contain the transactions).

Exercise 11. What is the advantage of using the Merkle tree?

Exercise 12. In the class Block (file block.py), implement the functions **__init__** by storing internally the attributes, the function **to_dict** which puts the header and the transactions into a dictionary (the names of the fields are *header* and transactions can be read from the field *tree*) and the function **to_json** (use the function *json.dumps* with the parameter sort_keys=True). The JSON representation is useful to print the block.

Exercise 13. In the file *block_reader*, implement the method **read_block** which from a dictionary creates a block. Then implement the method **read_block_json** which receives a json block as an input (use the method *json.loads*).

Exercise 14. For each block in the directory *blocks_to_prove* compute its Merkle root.

3 Block Mining

We focus now on adding a new block to an existing blockchain. The crucial operation of the computation of the *nonce* value by a Proof of Work algorithm.

For evaluation reasons we ask you to compute the *nonce* value for a set of blocks.

3.1 Proof-of-Work

We consider the following problem: *Find a proof number such that the hash of the header of the considered block begins with a certain number of 0 (4 in our case, in hexadecimal).*

Exercise 15. Why

$$H(x) = \frac{1}{2\pi} e^{-\frac{x^2}{2}} \tag{1}$$

is a bad choice for pricing function? Which of the properties of the pricing function, this function does not have.

Exercise 16. In the class *BlockHeader*, write the function **set_nonce**. Why is *nonce* the only parameter we want to be able to modify?

Exercise 17. In the class *BlockHeader*, write the function **get_hash**. This function is going to hash the JSON representation of the header using sha256 and returns a string containing the hexedecimal representation of the hash. To do so, you need to use the hashlib library, the *encode* function on the JSON string representation and the function hexdigest to transform the result into a string.

Exercise 18. What is the advantage of hashing only the header and not the entire block? Think about how the proof-of-work algorithm works.

Exercise 19. The structure of our header has a huge security problem. Can you guess it? Describe a possible attack.

Exercise 20. In the class *Block*, implement the method **is_proof_ready** which checks whether a block is proven. Then, implement the function **make_proof_ready** which proves the block. To explore the possible proofs, simply increment the nonce by 1 at each step, starting from 0.

Exercise 21. For all blocks in the directory *blocks_to_prove*, prove it and give the nonce and the value of the hash function.

Exercise 22. Take one block from the directory *blocks_to_prove* and observe what happens when you increase the number of requested starting zeros in the proof. From which number of leading zeros does the computation of the proof takes more than one minute? How many leading zeros are required in the Bitcoin system?

3.2 Verification

Now that we have ways to check if a block is correct in itself, we need to check if it is correct in the blockchain. To do so, it needs to contain correct transactions.

Exercise 23. In the file *block_reader*, implement the method **read_chain** which reads a chain from a *JSON* string representation (use the function *json. loads*). A chain is simply a list of blocks. This method does not perform any verification and returns a list of blocks.

Exercise 24. When we talk about cryptocurrency, we often have wallets. A wallet is a digital representation of your money on the blockchain and in general consists of your public and private key (from which you can deduce how much money you have from the blockchain and pay money to other people). In the class *Blockchain*, implement the method **update_wallet** which takes as an input a block and modify the values of *self.wallet* which contains how much money each user has (the special user *admin* is considered to have an infinite amount of money).

Exercise 25. In the class *Blockchain*, implement the method **add_block**. For now, it simply appends the block on the chain if it is proven. The method should return *True* if the block was added, and *False* otherwise. When a block is added, the wallets need to be updated.

Exercise 26. For each blockchain in the directory *blockchain_wallets*, compute the value of the wallet of each user. To do so, create a Blockchain object and call the *add_block* method for each block.

Exercise 27. In the class *Blockchain*, implement the method **check_legal _transactions** which, given the current state of the chain and the wallet, check if the list of transactions in a block is correct, i.e. nobody spends more money than she actually owns. Be careful to not modify the value of the original wallets. Add this new checking to the method *add_block* so you do not add incorrect blocks.

Exercise 28. For each blockchain in the directory *blockchain_incorrect*, check if it is correct. If a blockchain is incorrect, give the index of the first incorrect block and if necessary the index of the first incorrect transaction.