Lab Report
Blockchain Lab

ZHANG Xin

January 2020

Contents

1 Blockchain

1.1 Header Operations

1.1.1 Where is the address to the previous block stored?

The address to the previous block is stored in **previous hash** of the header.

1.1.2 implement `__init__`, `to_dict` and `to_json` in `block_header.py`

```
    def __init__(self, index, previous_hash, timestamp, nonce):
        # Store internally
        self.index = index
        self.previous_hash = previous_hash
        self.timestamp = timestamp
        self.nonce = nonce

    def to_dict(self):
        # Transform object into a dictionary for future transformation in JSON
        # The gave of the fields are the name of the variables
        return self.__dict__

    def to_json(self):
        # Transforms into a json string
        # use the option sort_key=True to make the representation unique
        return json.dumps(self.to_dict(), sort_keys=True)
```

It is important to sort the keys because if the the order is different, the hash result will also be different. The order will ensure a universal encoding

1.1.3 Which of the header's fields is unnecessary and redundant?

The index is unnecessary and redundant, because in the chain, every block is already linked by a order with the known previous hash, there is no use in maintaining an index.

1.1.4 implement the method `read_header` which returns a header from a dictionary

```
def read_header(header):
    # Implement these functions to help you
    # Takes a dictionary as an input
    return BlockHeader(**header)
```

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

The missing fields are:
Version: The version of the block.
Merkle Root: All of the transactions in this block, hashed together. Basically provides a singleline summary of all the transactions in this block.
Bits: A shortened version of the Target.

1.2 Transaction Operations

1.2.1 implement `__init__` and `to_dict` in the class `Transaction`

```
class Transaction(object):

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

    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
        return self.to_dict()
```

1.2.2 Which field is missing to make the transaction secured?

The signature of sender. Without the signature, the attacker can forge the sender's name to modify the transaction.

1.2.3 How is a person generally identified on a blockchain? How does a person prove ownership?

Through a public key, the ownership can be proven by the private key.

1.2.4 In the file `block_reader`, implement the method `read_transaction` which creates a Transaction from a dictionary.

```python
def read_transaction(transaction):
    # Same above for transformation
    return Transaction(**transaction)
```

1.3 Block Operations

1.3.1 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

Firstly, it is needed to define the classes for node and leaf:

```python
class MerkleTreeNode(object):

    def __init__(self):
        self.parent = None
        self.child_left = None
        self.child_right = None
        self.hash = None

class MerkleTreeLeaf(object):

    def __init__(self, transaction):
        self.parent = None
        self.transaction = transaction
        self.hash = compute_hash(transaction)
```

Then, in order to simplify and generalize the code, I modified the `MerkleTree` class in order to make it generate the nodes by itself (with the `generate_nodes` method) form the leaves:

```python
class MerkleTree(object):

    def __init__(self):
        self.root = None
        self.nodes = []  # Stores all other nodes except the leaf nodes
        self.leaves = []  # List of leaves of the tree. Leaves contain the transactions

    def add_node(self, node):
        # Add new node
        self.nodes.append(node)

    def add_leaf(self, leaf):
        # Add new leaf node
        self.leaves.append(leaf)

    def generate_nodes(self, list = None):
        temp = []
        if list is None:
            list = self.leaves
        # create the nodes of this layer
        if len(list) > 2:
            for index in range(0, len(list), 2):
                node = MerkleTreeNode()
                node.child_left = list[index].hash
                node.child_right = list[index+1].hash
                node.hash = list[index].parent
                temp.append(node)
        # make binary
            if len(temp) % 2 != 0:
                temp.append(temp[-1])
        # calculate the parents
            for index in range(0, len(temp), 2):
                temp[index].parent = compute_hash(temp[index].hash+temp[index+1].hash)
```

```
                temp[index + 1].parent = compute_hash(temp[index].hash + temp[index + 1].hash)
            for node in temp:
                self.nodes.append(node)
            self.generate_nodes(temp)
        # for the root node
        elif len(list)==2 :
            node = MerkleTreeNode()
            node.hash = list[0].parent
            node.child_left = list[0].hash
            node.child_right = list[1].hash
            self.root = node
            self.nodes.append(node)
```

Finally, for the function `create_merkle_tree`:

```
def create_merkle_tree(transactions):
    # Using the Merkle algorithm build the tree from a list of transactions in the block
    # transactions is list of Transaction
    merkle_tree = MerkleTree()
    for transaction in transactions:
        merkle_tree.add_leaf(MerkleTreeLeaf(transaction))
    # make sure that the tree can be binary
    if len(merkle_tree.leaves) % 2 != 0:
        merkle_tree.leaves.append(merkle_tree.leaves[-1])
    # compute the parents of the leaves
    for index in range(0, len(merkle_tree.leaves), 2):
        merkle_tree.leaves[index].parent = compute_hash(merkle_tree.leaves[index].hash + merkle_tree.leaves[index+1].hash)
        merkle_tree.leaves[index+1].parent = compute_hash(merkle_tree.leaves[index].hash + merkle_tree.leaves[index + 1].hash)
    merkle_tree.generate_nodes()
    return merkle_tree

def compute_hash(data):
    to_hash = json.dumps(data, sort_keys=True).encode('utf-8')
    return hashlib.sha256(to_hash).hexdigest()
```

1.3.2 What is the advantage of using the Merkle tree?

The Merkle Tree provide a way to prove both the integrity and validity of data. And the prove doesn't require a lot of memory. Moreover, the amount of information needed to be transmitted for a validation is also relatively low.

1.3.3 In the file `block.py`, implement the function `__init__`, `to_dict` and `to_json`

```
    def __init__(self, header, transactions):
        # Store everything internally
        # header is a BlockHeader and transactions is a list of Transaction
        # call create_merkle_tree function to store transactions in Merkle tree
        self.N_STARTING_ZEROS = 4
        self.header = header
        self.transactions = transactions
        pass

    def to_dict(self):
        # Turns the object into a dictionary
        # There are two fields: header and transactions
        # The values are obtained by using the to_dict methods
        return self.__dict__

    def to_json(self):
        # Transforms into a json string
        # use the option sort_key=True to make the representation unique
        return json.dumps(self.to_dict(), sort_keys=True)
```

1.3.4 In the file `block_reader`, implement the method `read_block` and the method `read_block_json`

```
def read_block(block):
    # Reads a block from a dictionary
    return Block(read_header(block["header"]), block["transactions"])


def read_block_json(block_json):
```

```
    # Reads a block in json format
    return json.loads(block_json)
```

1.3.5 For each block in the directory `blocks_to_prove` compute its Merkle root.

The Merkle roots are calculated with the following code:

```
import os
from block_reader import *

dir = 'blocks_to_prove'
for file in os.listdir(dir):
    with open(dir + '/' + file) as jsonp:
        print("For block: "+ file.title())
        block = read_block(json.load(jsonp))
        print("Merkle root: "+ block.transactions.root.hash)
```

And the output:

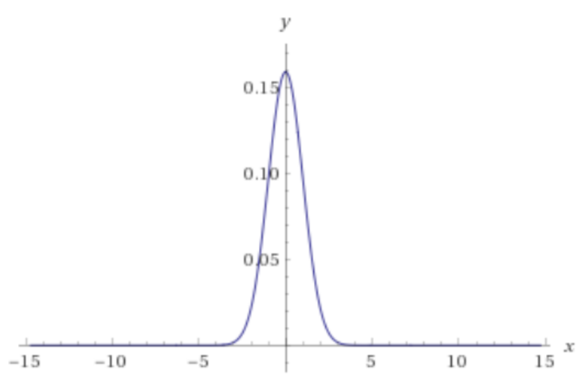
```
For block: Block0.Json
Merkle root: e6fce6128396968905dfcbc4f746c6ec717797d7fb7c0e4635087c199312d6ff
For block: Block1.Json
Merkle root: dfb8e32a7387bab859cdabc4fabd6901cdd80a137c41d957abc0ffdda9570cca
For block: Block2.Json
Merkle root: 4524ff1a88bc683d2d3fb372f612a87eb67f4cef7e6cfe2e3a0fef9a0977d9d5
For block: Block3.Json
Merkle root: 3a59743e61fc35550ce833121abb32456a1b537686fd18c9f99fc8694d74e586
For block: Block4.Json
Merkle root: 1c376ce9154bf2789b063aa41b0faa5eccbb73e48f9b40d331eedc4c39d5ac56
For block: Block5.Json
Merkle root: e89ef529be1e17779551a34023fe55c3cc2d1daa7703a84294b80b4c23725b41
For block: Block6.Json
Merkle root: 8b639fbf820b8c1bc018455e94230e6c619b6b8f10f4240461e93946628ecc62
For block: Block7.Json
Merkle root: f347ffa02d0dca22b3d46aa61592d119bb1bac81c594b75750b3fa4ec930944b
For block: Block8.Json
Merkle root: 360951cbbda687be7970b610ab423081c4e2294b0860af1cf6055c398039ecb5
For block: Block9.Json
Merkle root: 3df239fa1e842f413e168aaade6bc90fc28c2df5cb37f75ff5d48578629ec76b
```

2 Block Mining

2.1 Proof-of-Work

2.1.1 Why $H(x) = \frac{1}{2\pi}e^{-\frac{x^2}{2}}$ is a bad choice for pricing function?

We can plot this function:



31

From the plot, it can be easily observed that being a symmetric function, it is easy to find $x \neq x'$ such that $H(x) = H(x')$, it is not collision resistant. Moreover, it is easy to find the invese function $x = \pm 2^{\frac{1}{2}}(-\ln(2\pi y))^{\frac{1}{2}}$, so it is really easy to crack as we only need to try twice to get the x from a y.

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

```
    def set_nonce(self, new_nonce):
        # Set the nonce value
        self.nonce = new_nonce
```

Because in order to compute a new hash, we need to change the input. The **nonce** can take any value. Other options like the transactions require the merkle root be recomputed, and the range of the timestamp is limited a range of acceptable times. So the nonce is the best option to ensure the proof-of-work mechanism without affecting the normal fonctions of the blockchain.

2.1.3 In the class BlockHeader, write the function `get_hash`.

```
        def get_hash(self):
        # Use hashlib to hash the block using sha256
        # Use hexdigest to get a string result
        return hashlib.sha256(self.to_json().encode('utf-8')).hexdigest()
```

2.1.4 What is the advantage of hashing only the header and not the entire block?

Actually, as the header contains the *Merkle root* which is a hashing result of all the transactions, a hashing over the header is already a hashing over the entire block, but indirectly. Thus, with the same result, hashing only the header will save a lot of time and memory.

2.1.5 The huge security problem of the structure of our header

In the current structure, There is no Merkle root in the header, so there is the possibility of **Double-Spend Attack**, as the transactions cannot be confirmed by the hashing result of others.

2.1.6 In the class `Block`, implement the method `is_proof_ready` and `make_proof_ready`

```
    def is_proof_ready(self):
        # Check whether the block is proven
        # For that, make sure the hash begins by N_STARTING_ZEROS
        block_hash = self.header.get_hash()
        return block_hash.startswith('0'*self.N_STARTING_ZEROS,0)

    def make_proof_ready(self):
        # Transforms the block into a proven block
        nonce = 0
        while not self.is_proof_ready():
            nonce += 1
            self.header.set_nonce(nonce)
```

2.1.7 For all blocks in the directory `blocks_to_prove`, prove it and give the nonce and the value of the hash function.

```
dir = 'blocks_to_prove'
print("block : nonce, hash")
for file in os.listdir(dir) :
    with open(dir+"/" +file) as f:
        block_dict=json.load(f)
        block=read_block(block_dict)
        block.make_proof_ready()
        print(file[:-5]+" : "+str(block.header.nonce)+" , "+block.header.get_hash())
```

The output:

```
block : nonce, hash
block0 : 1438 , 0000aa66b0d1e8c3c7268f69d1ce2e39607c6ef33957a483c513b0372de1d2fe
block1 : 17773 , 000078ebc04a8c827de21e71eadddb5ce59d647999a3638433403cf7241bed27
block2 : 70441 , 000234700122aca31d18901bb723fdc4cf7f71dd47ec55f11f71e3693dddb52
block3 : 135140 , 0000180be1b98053bd2e0433c78d38d630afc5e45f91ede9dac86f4e704f1a80
block4 : 25067 , 0000bef8730d5c78679f854d0771b3d7fd9fce926df2e9d52391d7331ab981b9
block5 : 37716 , 000085656a163e42a67346007bab61ad890452f00832654c4020181d8afd140e
block6 : 22454 , 0000e41bbe5a1cb64a40e2e9d59da8994abae6df9345ec115d77781484a58bd8
block7 : 27395 , 0000a76aa2b0308782cedcf01a1958401bf2a219147588248dd8a2d0d3d74292
block8 : 188002 , 00006e24a6ddccbb6423d7ddaae7161f79cf863d45337f2b4bc9c6a650fe83eb
block9 : 59304 , 00009cc0b80cee2c556d93226e3802bb75c5e9c3ca7127d64165e833c605e8aa
```

2.1.8 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.

For the block block0.json, the running time needed to prove also increased as I increased the number of requested starting zeros.
From 7 leading zeros, the computation of the proof takes more than one minute.
According to the block published at `2020-01-17 19:01`, there are 18 leading zeros: (`00000000000000000010effc63faf969a85aa1ed60d70f48d2ade85f8f4c49ec`).

2.2 Verification

2.2.1 In the file `block_reader`, implement the method `read_chain`

```
def read_chain(chain):
    # read the chain from a json str
    # Returns a list of Block
    # This method does not do any checking
    return [read_block(item) for item in json.loads(chain)]
```

2.2.2 In the class `Blockchain`, implement the method `update_wallet`

```
    def update_wallet(self, block):
        # Update the values in the wallet
        # We assume the block is correct
        for temp in block.transactions.leaves:
            transaction = temp.transaction
            self.wallets[transaction.sender] = self.wallets.get(transaction.sender, 0) - transaction.
    amount
            self.wallets[transaction.receiver] = self.wallets.get(transaction.receiver, 0) + transaction.
    amount
```

2.2.3 In the class `Blockchain`, implement the method `add_block`.

```
    def add_block(self, block):
        # Add a block to the chain
        # It needs to check if a block is correct
        # Returns True if the block was added, False otherwise
        if block.is_proof_ready() and self.check_legal_transactions(block):
            self.chain.append(block)
            self.update_wallet(block)
            return True
        else:
            return False
```

2.2.4 For each blockchain in the directory `blockchain_wallets`, compute the value of the wallet of each user.

We test as below:

```
dir = 'blockchain_wallets'
for file in os.listdir(dir):
    with open(dir + '/' + file, "r") as f:
        block_chain = Blockchain()
        for block in read_chain(f.read()):
            block_chain.add_block(block)
        print("For " + file + " :")
        for (k, v) in block_chain.wallets.items():
            print(k + ": " + str(v))
```

The result:

```
For chain0.json :
admin: 99999999999940
alice: 23.066764710084918
bob: 6.989223624959041
fabian: 11.523123986955143
nicoleta: 5.216826891046529
jonathan: 30.33862527879706
julien: -17.13456449184271
For chain1.json :
admin: 99999999999940
```

```
alice: 11.573421224972464
bob: 24.42608955787173
fabian: 31.947976048929775
nicoleta: 38.56086202444834
jonathan: -58.858445784128584
julien: 12.350096927906254
For chain2.json :
admin: 99999999999940
alice: 5.936303698642465
bob: 26.594861814140405
fabian: 19.313046799661798
nicoleta: 9.908913284130959
jonathan: -9.00111029454074
julien: 7.2479846979650855
For chain3.json :
admin: 99999999999940
alice: 35.89240189063303
bob: -7.903430688406981
fabian: 9.026608870718398
nicoleta: 16.880933304284138
jonathan: 29.45328147362483
julien: -23.349794850853435
For chain4.json :
admin: 99999999999940
alice: 15.66006621865916
bob: 1.0293699508705805
fabian: 20.703883647477657
nicoleta: 22.899558304854377
jonathan: -8.77441317658521
julien: 8.481535054723452
For chain5.json :
admin: 99999999999940
alice: -7.387918978852761
bob: 19.706519086519396
fabian: 39.61254363196571
nicoleta: 13.887136660399259
jonathan: -15.04097994159423
julien: 9.222699541562665
For chain6.json :
admin: 99999999999940
alice: 21.610116364983185
bob: 38.474774833493164
fabian: -27.294742619818763
nicoleta: -7.352016177025115
jonathan: 28.343420550141666
julien: 6.218447048225925
For chain7.json :
admin: 99999999999940
alice: 33.975136783053735
bob: -6.9543470621640076
fabian: 13.99006707669638
nicoleta: 15.097137034419763
jonathan: -0.6485893087487211
julien: 4.54059547674286
For chain8.json :
admin: 99999999999940
alice: -1.502735584854065
bob: -23.67342650447773
fabian: 4.046044884172591
nicoleta: 22.885734714334557
jonathan: -14.336943022524588
julien: 72.58132551334926
For chain9.json :
admin: 99999999999940
alice: 22.530714125315022
bob: -4.8962305534617485
fabian: 52.08818494068747
nicoleta: -13.472038537464961
jonathan: 10.670854900471213
julien: -6.921484875547028
```

2.2.5 In the class Blockchain, implement the method check_legal_transactions

```
    def check_legal_transactions(self, block):
```

```
        # Check if the transactions of a block are legal given the current state
        # of the chain and the wallet
        # Returns a boolean
        wallets_copy = self.wallets.copy()
        for temp in block.transactions.leaves:
            transaction = temp.transaction
            balance = wallets_copy[transaction.sender]
            if balance < transaction.amount:
                return False
            else:
                wallets_copy[transaction.sender] = wallets_copy.get(transaction.sender, 0) - transaction.amount
                wallets_copy[transaction.receiver] = wallets_copy.get(transaction.receiver, 0) + transaction.amount
        return True
```

2.2.6 For each blockchain in the directory `blockchain_incorrect`, check if it is correct.

It is checked as below:

```
dir = 'blockchain_incorrect'
for file in os.listdir(dir):
    with open(dir + '/' + file, "r") as f:
        block_chain = Blockchain()
        correct = True
        for block in read_chain(f.read()):
            if not block_chain.add_block(block):
                print(file[:-5] + ' incorrect, first error in block ' + str(block.header.index))
                correct = False
                break
        if correct:
            print(file[:-5] + ' is correct')
```

The result:

```
chain0 incorrect, first error in block 4
chain1 incorrect, first error in block 7
chain2 incorrect, first error in block 3
chain3 incorrect, first error in block 4
chain4 incorrect, first error in block 2
chain5 incorrect, first error in block 3
chain6 incorrect, first error in block 2
chain7 incorrect, first error in block 3
chain8 incorrect, first error in block 4
chain9 incorrect, first error in block 2
```