Kevin Yao

Anish Malladi

PH360 – Special Projects in Physics

Prof Anita Raja

HW 2 – Question 2

13.02.2018

***Tutorial – Building a Blockchain***

**Representing a blockchain**

The tutorial represents the blockchain as a class. The class contains the methods that can be used on the class to accomplish the various tasks required of the chain. The methods described are:

1. *Initialisation*

Initialisation entails creating the chain. It creates two empty lists – one to store the blockchain, the other to store transactions which are used to created blocks.

1. *New Block Creation*

The .new\_block method creates a new block, and then adds it to the end of the blockchain. The transactions will be compiled from the list, added to the block, and then the block will be added to the chain, with all the required properties.

1. *New Transaction Creation*

A new\_transaction with all the required information is created and added to the list of transactions.

Using these methods, the blockchain can be created and subsequently manipulated as required. Given that each method is explored in detail, for now, a simple placeholder definition of the class is sufficient, as seen below:

class Blockchain(object):

def \_\_init\_\_(self):

self.chain = []

self.current\_transactions = []

def new\_block(self):

# Creates a new Block and adds it to the chain

pass

def new\_transaction(self):

# Adds a new transaction to the list of transactions

pass

@staticmethod

def hash(block):

pass

@property

def last\_block(self):

# Returns the last Block in the chain

pass

The final part of the placeholder is a static method which computes the hash of the block, by using sha256 as a hash function, and a property which allows the user to see the last block in the chain. This is useful functionality that is applied later.

**Structure of a Block**

Every block comprises the following parts:

1. Index – the block identification index.
2. Timestamp – the Unix timestamp of the block’s creation/verification
3. Transaction list
4. “Proof” – proof of work, discussed later
5. Hash pointer to the previous block

block = {

'index': 1,

'timestamp': 1506057125.900785,

'transactions': [

{

'sender': "8527147fe1f5426f9dd545de4b27ee00",

'recipient': "a77f5cdfa2934df3954a5c7c7da5df1f",

'amount': 5,

}

],

'proof': 324984774000,

'previous\_hash': "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"

The transactions are added to the block using a combination of the .new\_block and the .new\_transaction methods. By encoding the previous\_hash into the block, the previous block is “set in stone”. The idea is that the hash is computed using the static method hash, which uses *all* the data in the block. If a block is edited or modified in any way, the hash algorithm will not match, and a subsequent rehash of the edited block will result in a new hash value. However, since the original hash was encoded in the new block, it will be apparent that the edited block has been tampered with, and all the subsequent blocks will contain incorrect hashes[[1]](#footnote-1).

**Creating a New Transaction**

The new transaction is one of the “helper methods” in the tutorial. It accomplishes three tasks:

1. It creates a transaction with sender, recipient and amount fields.
2. It adds this transaction to a list of transactions which will make up the next block
3. It then returns the index of the block that will contain the transaction being added – which is the *next block that will be mined.*

def new\_transaction(self, sender, recipient, amount):

"""

Creates a new transaction to go into the next mined Block

:param sender: <str> Address of the Sender

:param recipient: <str> Address of the Recipient

:param amount: <int> Amount

:return: <int> The index of the Block that will hold this transaction

"""

self.current\_transactions.append({

'sender': sender,

'recipient': recipient,

'amount': amount,

})

return self.last\_block['index'] + 1

current\_transactions is the list of transactions that will comprise the next block.

**Creating a New Block**

Creating a new block has various facets to it. It needs to initialise a block, index it, link it to the predecessor, and add the transactions.

def new\_block(self, proof, previous\_hash=None):

"""

Create a new Block in the Blockchain

:param proof: <int> The proof given by the Proof of Work algorithm

:param previous\_hash: (Optional) <str> Hash of previous Block

:return: <dict> New Block

"""

block = {

'index': len(self.chain) + 1,

'timestamp': time(),

'transactions': self.current\_transactions,

'proof': proof,

'previous\_hash': previous\_hash or self.hash(self.chain[-1]),

}

# Reset the current list of transactions

self.current\_transactions = []

self.chain.append(block)

return block

**Proof of Work**

The example in the tutorial uses a much-simplified model. While the valid\_proof static method requires that the first four digits of the hash be 0 – the same as the example in class (the live blockchain at blockchain.mit.edu), the difference lies in the proof\_of\_work algorithm. The proof of work in the tutorial calculates the proof based on the proof of the last block only.

def proof\_of\_work(self, last\_proof):

"""

Simple Proof of Work Algorithm:

- Find a number p' such that hash(pp') contains leading 4 zeroes, where p is the

previous p'

- p is the previous proof, and p' is the new proof

:param last\_proof: <int>

:return: <int>

"""

proof = 0

while self.valid\_proof(last\_proof, proof) is False:

proof += 1

return proof

@staticmethod

def valid\_proof(last\_proof, proof):

"""

Validates the Proof: Does hash(last\_proof, proof) contain 4 leading zeroes?

:param last\_proof: <int> Previous Proof

:param proof: <int> Current Proof

:return: <bool> True if correct, False if not.

"""

guess = f'{last\_proof}{proof}'.encode()

guess\_hash = hashlib.sha256(guess).hexdigest()

return guess\_hash[:4] == "0000"

Essentially, the algorithm hashes together the proof of the last block together with the proof of the current block, and then checks whether the first four digits of the resulting hash are 0. If they are, then the proof is returned, if not, the proof of the current block is incremented by 1, and so on, until the condition is met.

The MIT website, however, hashes together the data in the block, the *hash of the previous block*, and the proof (which is referred to as ‘nonce’), and then checks the resulting hash to see whether the first four digits are 0 or not.

**Python Flask Framework**

/transactions/new

Creates a new transaction and adds to the next block to be mined. Essentially this endpoint calls the new\_transaction method in python. It is a POST method, in which the transaction is posted to the client.

/mine

Mines the next block at the given node. Essentially calculates the proof brute force such that a hash condition is met. This is a GET method – it receives information from the client.

/chain

Returns the chain as it currently stands on the node in question. This is also a GET method.

*Endpoints:*

Endpoints are the unique urls which are used as pointers. Here, the endpoints point to Python functions, which are requested by the HTTP client (Postman). So, when the method /mine is called, it acts as a GET request from the endpoint, receives the function and executes it to mine the block.

**Implementation**

The implementation for the next sections – using Postman to GET and POST requests to the chain, was done by initialising the node on a computer using Spyder from the Anaconda Suite. Spyder allows for multiple consoles, which allowed for simultaneous execution of two separate files – simulating two nodes on ports 5000 and 5001 of the localhost.

It is to be noted that this entire process, which combined Spyder and Postman could be replaced by a simple terminal prompt and cURL. Similarly, the nodes could be extended to any number of machines, provided that they are on the same network.

**Consensus**

The algorithm is incredibly basic in that it only searches for one parameter, and based on a binary result, proceeds with resolution of conflicts. The method is resolve\_conflicts:

def resolve\_conflicts(self):

"""

This is our consensus algorithm, it resolves conflicts

by replacing our chain with the longest one in the network.

:return: True if our chain was replaced, False if not

"""

# pdb.set\_trace()

neighbours = self.nodes

new\_chain = None

# We're only looking for chains longer than ours

max\_length = len(self.chain)

# Grab and verify the chains from all the nodes in our network

for node in neighbours:

response = requests.get(f'http://{node}/chain')

if response.status\_code == 200:

length = response.json()['length']

chain = response.json()['chain']

# Check if the length is longer and the chain is valid

if length > max\_length and self.valid\_chain(chain):

max\_length = length

new\_chain = chain

# Replace our chain if we discovered a new, valid chain longer than ours

if new\_chain:

self.chain = new\_chain

return True

return False

The method first finds the length of the chain at the node where the method is called. It then compares the length of the chain of the nodes that are its neighbours. After catching the longest chain, it checks validity and returns true or false based on the outcome.

***NOTE***: It was found that there was an issue in the valid\_chain method, where the original code called for the previous\_hash of the last\_block, instead of the current block. This is currently an open issue on the GitHub ([Issue #66](https://github.com/dvf/blockchain/issues/66)) and by editing last\_block to block, the resolve\_conflicts method did not resolve incorrectly, defaulting to authoritative no matter the length of the chain. We used a pdb trace to verify this issue independently.

1. Conceivably, one could go and rehash *the entire blockchain beyond the tampered block*, but this would be cryptographically difficult. While it may be possible in the case of this simple example, it would not work in a stable system like Bitcoin, which depends on verification of the chain. [↑](#footnote-ref-1)