Protecting democracy with a trustless blockchain based decentralised election system

Sumuk Shashidhar

March 11, 2021

Abstract

Democracy fades as sophisticated attempts of voterfraud are detected, with some even succeeding. VoteBlock attempts to protect democracy by decentralising the election process to ensure the lack of a single point of failure or control, with the help of a blockchain. It must be understood that while VoteBlock secures the election process, it does not secure the voter registration process essential for authorizing each voter.

Contents

1 Introduction

2 Structure of A Block

2.1 Components

Each block consists of the following elements essential to recording votes and identifying induviduality of each voter.

1. Index - To enumerate each block
2. Transactions - The given transactions in each block (votes)
3. Timestamp - The timestamp of each block's creation
4. Previous Hash - The hash of the previous block to facilitate the blockchain
5. Nonce - The number only used once. Used to supplement the rest of the data to generate a desired hash pattern

2.2 Hashing

SHA-256 hashing is used, where the data of the block is first dumped into a string, and then computed after being unicode encoded.

2.3 Code Used

block.py

```
from hashlib import sha256
import json

class Block:
        def __init__(self, index, transactions, timestamp, previous_hash, nonce=0):
                        self.index = index
                        self.transactions = transactions
                        self.timestamp = timestamp
                        self.previous_hash = previous_hash
                        self.nonce = nonce

        def compute_hash(self):
                block_string = json.dumps(self.__dict__, sort_keys=True)
                return sha256(block_string.encode()).hexdigest()
```

3 The Blockchain

3.1 Components

3.1.1 Basic Structuring

The Blockchain consists of a two objects.

1. Unconfirmed Transactions - To store a list of unconfirmed transactions.
2. Chain - To store the chain data.

Difficulty The difficulty is a simple integer that determines how hard it is to mine a block. The higher this integer is set, the more difficult it is to mine.

Method This is accomplished by using the number of leading zeros to the hash. A difficulty of 2 will ensure that there are two leading zeros for each accepted hash.

3.1.2 Genesis Block

The genesis block of any block chain is the first mined block of that blockchain. Here, we mine it with a list of empty transactions and null (0) data, and we add it to the chain.

3.1.3 Last Block Property

A useful property that is used to retrieve the last block added to the blockchain.

3.1.4 Add Block Methodology

It is very simple to add a block to the blockchain. It requires only the block object and the proof of work.

Verification includes:

- Checking if the proof is valid.
- The previous hash referred in the block and the hash of latest block in the chain match.

3.2 Code Used

```
from block import Block
import time

class Blockchain:
    difficulty = 2

    def __init__(self):
        self.unconfirmed_transactions = []
        self.chain = []

    def create_genesis_block(self):
        genesis_block = Block(0, [], 0, "0")
        genesis_block.hash = genesis_block.compute_hash()
        self.chain.append(genesis_block)

    @property
    def last_block(self):
        return self.chain[-1]

    def add_block(self, block, proof):
        previous_hash = self.last_block.hash

        if previous_hash != block.previous_hash:
            return False
```

```
        if not Blockchain.is_valid_proof(block, proof):
            return False

        block.hash = proof
        self.chain.append(block)
        return True

    @staticmethod
    def proof_of_work(block):
        """
        Function that tries different values of nonce to get a hash
        that satisfies our difficulty criteria.
        """
        block.nonce = 0

        computed_hash = block.compute_hash()
        while not computed_hash.startswith('0' * Blockchain.difficulty):
            block.nonce += 1
            computed_hash = block.compute_hash()

        return computed_hash

    def add_new_transaction(self, transaction):
        self.unconfirmed_transactions.append(transaction)

    @classmethod
    def is_valid_proof(cls, block, block_hash):
        """
        Check if block_hash is valid hash of block and satisfies
        the difficulty criteria.
        """
        return (block_hash.startswith('0' * Blockchain.difficulty) and
                block_hash == block.compute_hash())

    @classmethod
    def check_chain_validity(cls, chain):
        result = True
        previous_hash = "0"

        for block in chain:
            block_hash = block.hash
            # remove the hash field to recompute the hash again
            # using `compute_hash` method.
            delattr(block, "hash")

            if not cls.is_valid_proof(block, block_hash) or \
                    previous_hash != block.previous_hash:
                result = False
                break

            block.hash, previous_hash = block_hash, block_hash

        return result
```

```
def mine(self):
"""
This function serves as an interface to add the pending
transactions to the blockchain by adding them to the block
and figuring out Proof Of Work.
"""
if not self.unconfirmed_transactions:
return False

last_block = self.last_block

new_block = Block(index=last_block.index + 1,
transactions=self.unconfirmed_transactions,
timestamp=time.time(),
previous_hash=last_block.hash)

proof = self.proof_of_work(new_block)
self.add_block(new_block, proof)

self.unconfirmed_transactions = []
return True
```