**Lab 3: Create a blockchain continued**

**Tutorial to extend the functionality of the blockchain developed in python**

[**Develop a blockchain application from scratch in Python | RUOCHI.AI (zhangruochi.com)**](https://zhangruochi.com/Develop-a-blockchain-application-from-scratch-in-Python/2020/04/21/)

**Blockchain is a way of storing digital data. (Bitcoin is logs of transfers from one account to the other). Data stored in the form of blocks linked (chained) together using cryptographic hashes.**

**The magic is in the way data is stored and added to the blockchain. Blockchain is essentially linked list that contains ordered data with few constraints**

* Blocks can’t be modified once added (append only)
* Specific rules for appending data to it
* Architecture distributed

Benefits of enforcing constraints: Immutability and durability of data, no single point of control or failure, verifiable audit trail of the order in which data was added.

**Store transactions into blocks (storing data) using JSON (post stored in blockchain)**

**A screenshot of a computer screen

Description automatically generated**

**Generic term “data” is replaced by the term “transactions”**

Transactions are packed into blocks and each block can contain one or many transactions. These are generated frequently and added to the blockchain. Multiple blocks will all have unique ID.

**A screen shot of a computer code

Description automatically generated**

**Add digital fingerprints to the blocks**

Cryptographic hash functions to prevent any tampering of data stored in the block. Hash function takes data of any size and produces a data of fixed size (Deterministic – same data result same hash, random and easy to compute).

**Chain the blocks**

Making a way that any change in previous blocks invalidates the entire chain. Bitcoin way is to create dependency among consecutive blocks by chaining them with the hash of the block immediately previous to them.

**Implement a proof of work algorithm**

Creating a hash difficult and random (adding a constraint by leading zeros – a new nonce).

**Add blocks to the chain**

Verify data has not been tampered with (proof of work is correct) and order of transactions is preserved (previous hash) add to the hash of latest block in chain

**Mining**

Process of putting unconfirmed transactions in a block and computing proof of work is known as the mining of blocks.

**Create Interfaces**

Blockchain node to interact with an application called Flask. A rest api that interacts with and invokes various operations in blockchain node.

**Establish consensus and decentralization**

Relinking the block with hashes and applying the PoW constraint. Data needs to be distributed and multiple nodes to maintain the blockchain. Create a mechanism to let a new node aware of peers in the network (single node to a peer-to-peer network). Achieving a simple consensus algorithm to agree upon the longest valid chain when chains of different participating nodes in the network appear to diverge.

**Build the application**

Application needs to connect to a node in the blockchain network to fetch the data and submit new data. Application has a HTML form to take user input and makes a post request to a connected node to add the transaction into unconfirmed transactions pools.

**Revision:**

**I am currently testing the python blockchain code that was created in labs at campus from a different computer. The code is working as it is currently running in the console**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer

Description automatically generatedI check the results from the Postman API to see the if we can retrieve data as per previous lab but to no avail**

**I see that I’ve entered the incorrect URL end point (chain\_block) where it should be (get\_chain)**

[**http://192.168.107.55:5000/get\_chain**](http://192.168.107.55:5000/get_chain)

**Here’s our response from Postman (API – Application Programming Interface) after entering the correct HTTP GET Request get\_chain. We create the first genesis block.**

A screenshot of a computer

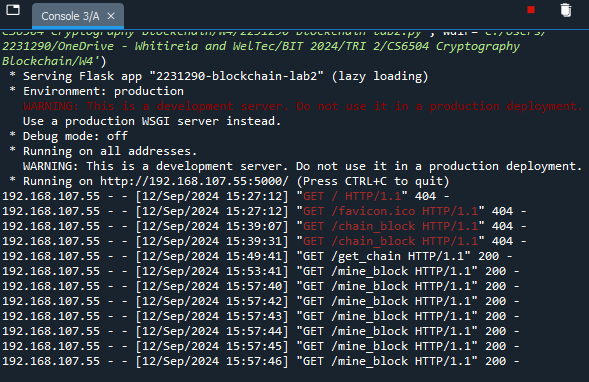
Description automatically generated

**Now I initiate the HTTP GET Request mine\_block and to our success**

A screenshot of a computer

Description automatically generated

**Summary of request indicates we have created a few more mining blocks and the index indicates the number of mines (Postman) shows how many we have created and the proof of work number and hash**



**We check if block chain is valid**

A screenshot of a computer

Description automatically generated

**Now we look to test the other blockchains in the tutorial first instructions is to pip install Flask request in Anaconda Prompt (Anaconda3)**

**A screen shot of a computer

Description automatically generated**

A black screen with white text

Description automatically generated



from hashlib import sha256

import json

import time

from flask import Flask, request

import requests

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):

"""

A function that return the hash of the block contents.

"""

block\_string = json.dumps(self.\_\_dict\_\_, sort\_keys=True)

return sha256(block\_string.encode()).hexdigest()

class Blockchain:

# difficulty of our PoW algorithm

difficulty = 2

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

self.unconfirmed\_transactions = []

self.chain = []

def create\_genesis\_block(self):

"""

A function to generate genesis block and appends it to

the chain. The block has index 0, previous\_hash as 0, and

a valid hash.

"""

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):

"""

A function that adds the block to the chain after verification.

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.

"""

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

app = Flask(\_\_name\_\_)

# the node's copy of blockchain

blockchain = Blockchain()

blockchain.create\_genesis\_block()

# the address to other participating members of the network

peers = set()

# endpoint to submit a new transaction. This will be used by

# our application to add new data (posts) to the blockchain

@app.route('/new\_transaction', methods=['POST'])

def new\_transaction():

tx\_data = request.get\_json()

required\_fields = ["author", "content"]

for field in required\_fields:

if not tx\_data.get(field):

return "Invalid transaction data", 404

tx\_data["timestamp"] = time.time()

blockchain.add\_new\_transaction(tx\_data)

return "Success", 201

# endpoint to return the node's copy of the chain.

# Our application will be using this endpoint to query

# all the posts to display.

@app.route('/chain', methods=['GET'])

def get\_chain():

chain\_data = []

for block in blockchain.chain:

chain\_data.append(block.\_\_dict\_\_)

return json.dumps({"length": len(chain\_data),

"chain": chain\_data,

"peers": list(peers)})

# endpoint to request the node to mine the unconfirmed

# transactions (if any). We'll be using it to initiate

# a command to mine from our application itself.

@app.route('/mine', methods=['GET'])

def mine\_unconfirmed\_transactions():

result = blockchain.mine()

if not result:

return "No transactions to mine"

else:

# Making sure we have the longest chain before announcing to the network

chain\_length = len(blockchain.chain)

consensus()

if chain\_length == len(blockchain.chain):

# announce the recently mined block to the network

announce\_new\_block(blockchain.last\_block)

return "Block #{} is mined.".format(blockchain.last\_block.index)

# endpoint to add new peers to the network.

@app.route('/register\_node', methods=['POST'])

def register\_new\_peers():

node\_address = request.get\_json()["node\_address"]

if not node\_address:

return "Invalid data", 400

# Add the node to the peer list

peers.add(node\_address)

# Return the consensus blockchain to the newly registered node

# so that he can sync

return get\_chain()

@app.route('/register\_with', methods=['POST'])

def register\_with\_existing\_node():

"""

Internally calls the `register\_node` endpoint to

register current node with the node specified in the

request, and sync the blockchain as well as peer data.

"""

node\_address = request.get\_json()["node\_address"]

if not node\_address:

return "Invalid data", 400

data = {"node\_address": request.host\_url}

headers = {'Content-Type': "application/json"}

# Make a request to register with remote node and obtain information

response = requests.post(node\_address + "/register\_node",

data=json.dumps(data), headers=headers)

if response.status\_code == 200:

global blockchain

global peers

# update chain and the peers

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

blockchain = create\_chain\_from\_dump(chain\_dump)

peers.update(response.json()['peers'])

return "Registration successful", 200

else:

# if something goes wrong, pass it on to the API response

return response.content, response.status\_code

def create\_chain\_from\_dump(chain\_dump):

generated\_blockchain = Blockchain()

generated\_blockchain.create\_genesis\_block()

for idx, block\_data in enumerate(chain\_dump):

if idx == 0:

continue # skip genesis block

block = Block(block\_data["index"],

block\_data["transactions"],

block\_data["timestamp"],

block\_data["previous\_hash"],

block\_data["nonce"])

proof = block\_data['hash']

added = generated\_blockchain.add\_block(block, proof)

if not added:

raise Exception("The chain dump is tampered!!")

return generated\_blockchain

# endpoint to add a block mined by someone else to

# the node's chain. The block is first verified by the node

# and then added to the chain.

@app.route('/add\_block', methods=['POST'])

def verify\_and\_add\_block():

block\_data = request.get\_json()

block = Block(block\_data["index"],

block\_data["transactions"],

block\_data["timestamp"],

block\_data["previous\_hash"],

block\_data["nonce"])

proof = block\_data['hash']

added = blockchain.add\_block(block, proof)

if not added:

return "The block was discarded by the node", 400

return "Block added to the chain", 201

# endpoint to query unconfirmed transactions

@app.route('/pending\_tx')

def get\_pending\_tx():

return json.dumps(blockchain.unconfirmed\_transactions)

def consensus():

"""

Our naive consnsus algorithm. If a longer valid chain is

found, our chain is replaced with it.

"""

global blockchain

longest\_chain = None

current\_len = len(blockchain.chain)

for node in peers:

response = requests.get('{}chain'.format(node))

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

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

if length > current\_len and blockchain.check\_chain\_validity(chain):

current\_len = length

longest\_chain = chain

if longest\_chain:

blockchain = longest\_chain

return True

return False

def announce\_new\_block(block):

"""

A function to announce to the network once a block has been mined.

Other blocks can simply verify the proof of work and add it to their

respective chains.

"""

for peer in peers:

url = "{}add\_block".format(peer)

headers = {'Content-Type': "application/json"}

requests.post(url,

data=json.dumps(block.\_\_dict\_\_, sort\_keys=True),

headers=headers)

# Uncomment this line if you want to specify the port number in the code

app.run(port=8000)