SMART GRID TRANSACTIONS VIA THE BLOCKCHAIN

A Dissertation

by

SHAIKHA SAAD S A AL-QAHTANI

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Joseph Boutros
Co-Chair of Committee,	Robert Balog
Committee Member,	Othmane Bouhali
Head of Department,	Efstratios N. Pistikopoulos

December 2021

Major Subject: Energy

ABSTRACT

With more outages in recent times, the energy sector needs to shift to more reliable electricity grid. Smart grids are the new solution where it is much more reliable and secure than the traditional electricity grid. The use of smart grid would help with integrating new technologies to the grid such as the blockchain. The blockchain is a public distributed ledger which stores transactions in blocks that are chained together by cryptographic hashes. The blockchain is managed by a peer-to-peer network autonomously without the need of a central entity. The use of blockchain in smart grids will help in removing the third parties in transactions, which makes it easier for smaller energy providers to enter the market. Also, the blockchain is immutable due to how each block is chained together, if you alter a block, it causes a chain effect and changes subsequent blocks. Thus, it is impossible to hack as other nodes in the network have a copy of the blockchain and they can easily identity a corrupted blockchain. In this paper, we explore the use of blockchain in handling smart grid transactions. A smart grid specialized blockchain was developed in Python. The users can interact with the blockchain by sending HTTP requests over Postman. Postman is a program for API testing, and it is the main user interface. The blockchain developed is account based, so the user has to generate an e-Wallet before running transactions on the blockchain. There are three types of users in the blockchain, a generator, a prosumer, and a customer. Based on the type, the user can then either generate electricity and sell it over the blockchain or purchase the electricity then consume it. The users can either run transactions or mine blocks in the blockchain.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Joseph Boutros (chair), Professor Robert Balog (co-chair) and Professor Othmane Bouhali (member). All other work conducted for the thesis was completed by the student independently.

Funding Sources

None.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

Page

INTRODUCTION

The energy trends show a huge increase in electricity demand worldwide. The increase in electricity demand and consumption pushes for newer technology that are more reliable and less prone to outages. The smart grid is the new enhanced electric grid which includes smart meters and appliances. The smart grid is more reliable and secure than the legacy electricity grid, as it provides a two-way communication from the generator to the customers, which ensures that the electricity has been delivered.

Power grids nowadays are based on multiple sources such as the sun, the wind, and fossil fuels. It is difficult to integrate all of the newer greener sources of energy into the legacy electric grid. The need for integrating the energy coming from renewable sources pushes for technologies like the smart grid to replace the legacy electric gird. Besides classifying the energy nature, the smart grid itself involves other complex transactions such as advanced metering, managing electric vehicles charging, trading of decentralized energy, and the monitoring and control of the cyber-physical network.

The use of blockchain to handle the smart grid transactions would be investigated in this thesis. A blockchain is a shared, encrypted record that is maintained by a network of computers. These computers verify transactions, like the transfer of cryptocurrency between two individual users. The blockchain record is a tree of blocks linked via a cryptographic hash function. A leaf block cannot be modified without changing all previous blocks. The blockchain is an efficient technique to conduct, verify, and record smart grid transactions such as those between consumers and generators. The blockchain guarantees the integrity of registered transactions and their non-repudiation. The use of blockchain in smart grids would ensure a faster and easier integration of new electricity providers.

BLOCKCHAIN IN ENERGY

The use of smart grid technology has been on the rise recently in the energy sector. The reasons for the switch to smart grid is that it is far more reliable and secure than the legacy electric grid. It allows for two-way communication from the generator to the consumer and vice versa. In addition, the smart grid allows for an easier and faster integration of renewable energy sources to the grid. Furthermore, the energy sector is looking into blockchain platforms to run the transactions done in the smart grid. The blockchain is a public ledger which records all of the transactions in the network. It provides a reliable decentralized solution in energy trade. It ensures that the electricity would be provided to the customer after the payment has been placed, as the blockchain seamlessly record all transactions without having a central entity controlling those transactions. Blockchain is mostly known for being used in cryptocurrencies, like the Bitcoin, but there is another version used in energy applications known as Ethereum. One of the most noteworthy implementations of the blockchain in energy trading using Ethereum is in the Brooklyn microgrid launched by LO3Energy alongside ConsenSys, which is a blockchain company [2].

A. Impact of the blockchain on the energy sector

Blockchain technologies will surely affect the business models of the current energy sector and the trading sector associated to it. The blockchain can be applied to a large number of cases and its impact on the energy sector is summarized as found in the literature [3]:

- Payments: The blockchain can realize automated billing for customers and distributed generators. Micro payments, pay-as-you-go transactions, and pre-paid meters are simpler to make via the blockchain.
- Sales and marketing: The blockchain combined to machine learning can customize energy products to fit customers' needs and habits.

- Green trading: Blockchains systems are currently being developed for green certificates trading.
- Automation: The blockchain leads to decentralized energy systems and microgrids. P2P operations will go through the blockchain. This may affect the revenue of traditional operators.
- Smart grid applications and data transfer: The blockchain can play the role of the Internet for the energy grid. Data communications involve smart meters, advanced sensors, network monitoring equipment, control and energy management systems, smart home energy controllers and building monitoring systems.
- Grid management: The Blockchain could assist in the management of decentralized networks. As a result, blockchains might affect the tariffs for network use.
- Security and identity management: Because it is based on cryptography, the blockchain could safeguard privacy, data confidentiality, and identity management.
- Sharing of resources: Sharing resources between users in blockchain could be used for charging stations of electrical vehicles, and for data sharing.
- Competition: Introducing blockchain to the smart grid would help smaller establishments starting up in energy trade making a fairer competition as opposed to having market monopoly where one entity controls the grid, and this could reduce energy tariffs.
- Transparency: As the blockchain is immutable and completely transparent, it helps with improving audits and compliance to regulations.

B. Transformation of the market

Implementing blockchain trading platforms in the energy sector could potentially change the energy market structure. Figure 1 shows the transformation of the market with blockchain

based on a consulting agency pwc [3]. The figure shows that a central bank, energy retailers, meter operators and traders are no longer needed when a blockchain trading platform is implemented. As the blockchain would handle all energy transaction without needing all those entities. This would encourage new energy establishments to join the energy market. However, such huge shift in the energy market is not easy, as there are challenges that comes with implementing it. First, bank transactions happen in the order of seconds, while transactions in blockchain are in the order of minutes. The blockchain is slower mainly due to using such huge computing power and memory to render a block to be added in the blockchain. So, at the moment, a conventional bank transaction is faster than a blockchain transaction. Thus, it would be hard to change the current energy market structure at a short period of time, but it is possible to change a part of it.

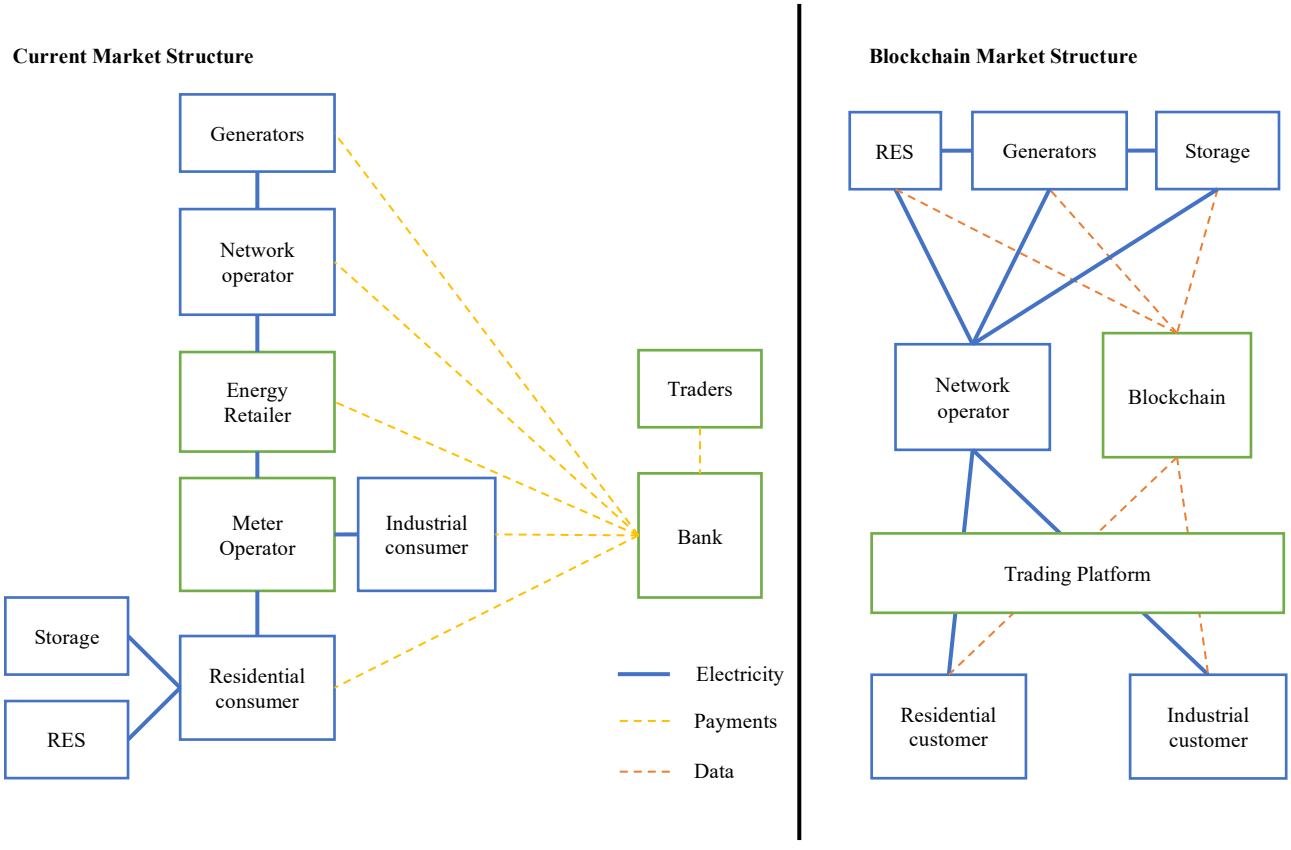


Figure 1. Transformation of the Market with Blockchain (adapted from PWC [3]).

C. Basic blockchain architecture in smart grids

The basic blockchain architecture in the case of smart grids consists of nodes connected on a peer-to-peer network. Anybody can be a node in the blockchain. The nodes can either be a generator node where they generate and sell electricity, or a customer node who would purchase and use the electricity. Also, there could be a node who acts as both a customer and a generator known as a prosumer. Finally, a distribution node where all transactions are verified and routed [1]. All of the actions of nodes and examples are shown in table 1. This way, the smart grid transactions would be secure, and it would allow for small generators to enter the energy market.

Table 1. The nodes in blockchain application in smart grids [1]

Nodes	Action	Example
Generators	Generate and sell electricity	Power Plants
Customers	Purchase and use electricity	Buildings
Prosumers	Generate, sell, purchase, and use electricity	Buildings with power generations
Distribution Point	Verify and route transactions	Step down transformer at every node

D. How blockchain is used in smart grids

The blockchain in smart grids is implemented by the following process; A generator node would generate a token along with the electricity generated. The generator node would sell the electricity over the network looking for a customer node. The system would check that the electricity is available and allocate the sold electricity before publishing the sell order. The customer node would purchase the token. The system would check the customer's balance before the purchase order is published. Then a smart contract is generated for that specific sell and

purchase order, connecting the generator to the customer by a transaction which would be added to the blockchain. This ensures a reliable transfer of electricity where both sides have been validated before the transaction has been made. Once the customer receives the allocated electricity that was specified in that transaction the token is then transferred to an address which cannot be recovered so it cannot be used again. This is known as “burning” the token in blockchain [1].

Transactions are then verified by the distribution point nodes also known as miners in the network. The miners check the generator and customer accounts, and traceback whether the customer is the rightful owner of that specific token. Also, the miner would verify whether the generator has the allocated electricity to provide to the customer. Once all has been verified by the miner then the block of transactions is added to the blockchain and sent to all of the nodes to the network confirming that the smart contract has been executed [1].

THE THEORY OF BLOCKCHAIN

A. Hashing

Hash functions take an input x of a random size and give an output for a fixed size Hash(x). It is a one-way encryption where it is very easy to hash but mathematically almost impossible to find the reverse of the hash for that block. Hash functions are deterministic, it will always have the same hash value for the same input. However, all hash outputs are so random that if you change a single digit on the input, the hash output for the same hash function would be so different that it cannot be correlated. Hash functions are used in blockchain as they are not reversible, making the blockchain secure and not able to traceback.

A.1 Hash functions' features [5]:

- **Collision resistance:** it is difficult to find two messages M1 and M2 such that they would generate the same hash output Hash(M1) = Hash(M2).
- **Preimage resistance**: it is difficult to find the reverse of the hash function (The message M1) given only the hash value Hash(M1).
- **Second preimage resistance:** given a pair of input and output of a hash function M1 and Hash(M1), it is difficult to find another input M2 that would give the same output of the previous input Hash(M1) = Hash(M2).

A.2 SHA2

SHA2 stands for Secure Hash Algorithm 2, SHA2 has two main algorithms SHA256 and SHA512. The numbers 256 and 512 refer to the size of hash output in bits. The hash algorithm SHA256 is commonly used in blockchain applications because it was first suggested by Satoshi Nakamoto in his white paper about Bitcoin [7]. SHA2 is a compression-based hash function using the Merkle–Damgård construction (MD). The algorithm depends on splitting the message input

into same sized blocks with an internal state using a compression function to hash a message as shown in figure 2 [8].

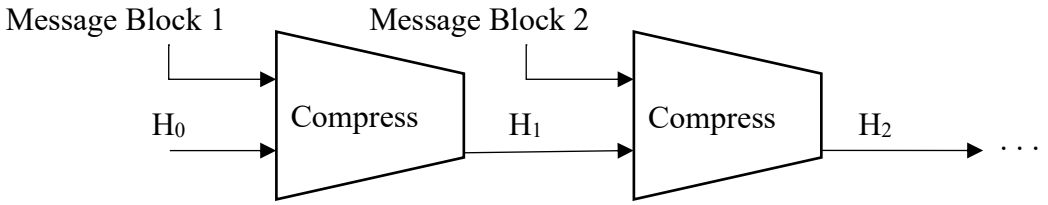


Figure 2. The Merkle–Damgård construction using a compression function called compress.

The message block size is fixed for each function, SHA256 uses blocks of size 512bits while SHA512 uses blocks of size 1024bits. If the input cannot be split into equal size blocks, then the algorithm will pad the input. The MD construction pads the input by taking the excess bits to create the last block, then append 1 bit followed by zero bits. After that it would append encoded bits at the end which express the length of the input. SHA2 has been proven to be secure as it resisted vulnerability tests. However, SHA2 uses a similar algorithm as its predecessor SHA1, which is vulnerable to attacks, so it is believed that SHA2 is bound to fail.

A.3 SHA3

In case SHA2 fails, SHA3 could be used instead. SHA3 differs entirely in the algorithm as it is a permutation-based sponge hash function. Thus, SHA3 in theory is more secure than SHA2 since it has a different internal algorithm that does not resemble a previous hash function that is vulnerable. The methodology of a permutation-based sponge hash function begins with XORing the first message block to an initial predefined value of the internal state. Then the initial internal state value will be transformed by using a premutation function **P**. The result of the permutation will be XORed onto the second message block and so on. This is known as the absorbing phase as

seen in figure 3 [8]. After that comes the squeezing phase, once all the message blocks have been injected. **P** is applied again to extract a block of bits from the state to form the final hash.

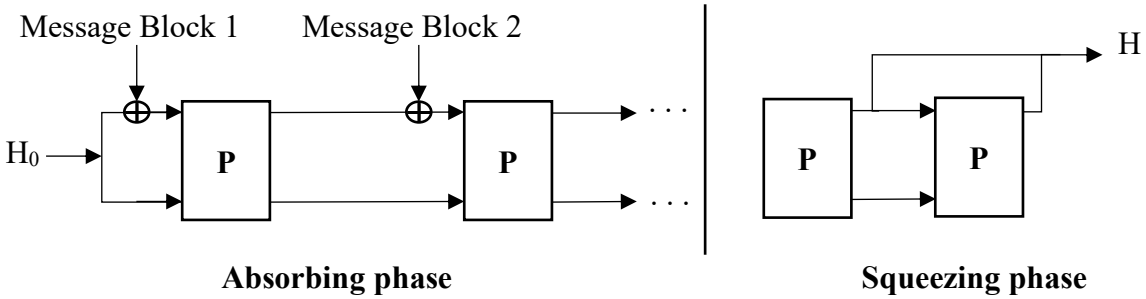


Figure 3. The Sponge Construction.

The padding in sponge functions is simpler than in compression function-based hashes, it appends the message with 1 bit and zeros without including the length of the message. The permutation **P** should be random without any statistical bias for the sponge construction to be secure. SHA3 provides a strong permutation algorithm that has no bias, so it is considered the most secure hash function and it is believed that it won't fail anytime soon.

A.4 Possible attacks on hash functions:

- **Collision attack:** The attacker tries to find two messages M1 and M2 such that they would generate the same hash output Hash(M1) = Hash(M2). Attackers utilize **Birthday Attacks** to find such hash collision. Basically, if you have a set of N messages and hash values, it is possible to produce N*(N-1)/2 potential collisions by taking each pair of two hashes (in the order of N^2). It is known as a birthday attack because it is based on the birthday paradox; if there were 23 people in a room there is a probability of 1/2 to have two people sharing the same birthday.

- **Preimage attack:** The attacker tries to retrieve the original message from the hash value.
- **Second-preimage attack:** The attacker starts with a message M1 and tries to find another message M2 such that Hash(M1)=Hash(M2)
- **Length extension attack:** The attacker takes the hash result Hash(M) of a message M without knowing the contents of the message. M is split into two blocks M1 and M2 after padding. Then the attacker can calculate the Hash(M1||M2||M3) where M3 is a message chosen by the attacker. This due to the fact that the hash of M1 concatenated with M2 gives the chaining value after M2 as seen in figure 4. So, you can add more blocks to the message without knowing the original message. This attack method is the main threat to the Merkle–Damgård construction. The attacker won't be able to affect the main features for hash functions, but this is a security threat that makes SHA2 and all hash functions based on MD construction vulnerable.

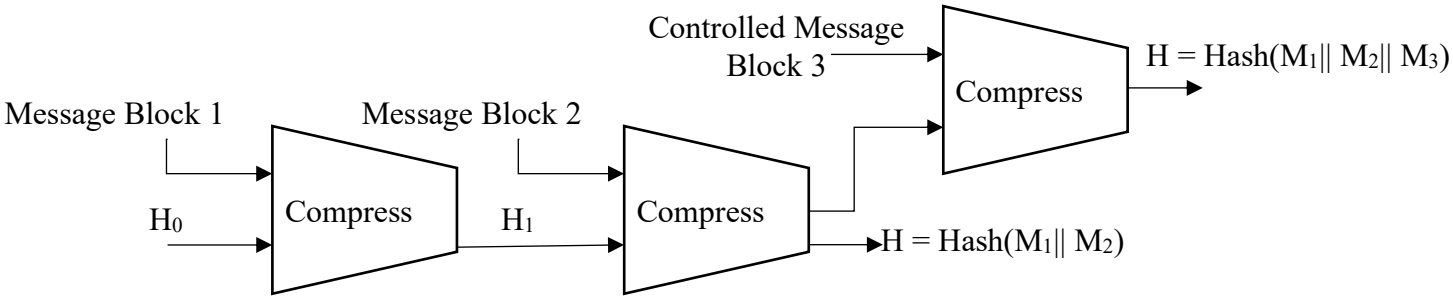


Figure 4. The length-extension attack.

A.5 Hash functions on Python

Hashlib is a python library that is home to hash functions and functions that treat the hash digest [9]. The examples below hashes the message "Blockchain2021" using three hash functions,

SHA256, SHA3_256, and SHA3_512. You can see each function outputs a completely random and different output. For SHA256 and SHA3_256 the output is 256 bits which is 64 hexadecimal digits, while SHA3_512 outputs 512 bits (128 hexadecimal digits) as seen in the below results.

- **Example of SHA256 in Python:**

```
>>> import hashlib
>>> message = "Blockchain2021"
>>> hash = hashlib.sha256(str(message).encode()).hexdigest()
>>> hash
'7561c90bf9454fb92c49ffcf1d3045c83c579c9d4c3575e68ee38f3256bef673'
>>> len(str(hash))
64
```

- **Example of SHA3_256 in Python:**

```
>>> hash_sha3_256 =hashlib.sha3_256(str(message).encode()).hexdigest()
>>> hash_sha3_256
'fe80c32aeed40ec815e5e74ff3dc309463f23031e1f1049f6376af070c060fa1'
>>> len(str(hash_sha3_256))
64
```

- **Example of SHA3_512 in Python:**

```
>>> hash_sha3_512 =hashlib.sha3_512(str(message).encode()).hexdigest()
>>> hash_sha3_512
'1a17f9f8a0219bd31b5c48d5926a29f7455d6cbb703c95370afc34fb09105a3b18712
bc5a29d452a6422498b692c04f96fdd9dfc02ec7f567933b1d3e2751ee1'
>>> len(str(hash_sha3_512))
128
```

B. Chaining blocks

A blockchain is a long series of blocks which are added by nodes in the network in a sequential manner and is secured by hash functions. The block encapsulates a series of transactions. The block structure can be seen in figure 5. It consists of a block header, the transactions, a timestamp and other block fields (a nonce number, block version etc.) [6]. Once the block is full, it is hashed, and a new block would be generated with the hash value of the previous block, which creates a “Blockchain”. The first ever block in a blockchain is known as the genesis block, it is hardcoded into the blockchain as there is no previous block to chain it with. The

previous hash value in the genesis block is usually set to zero alongside other parameters that are required to be calculated or "mined" to chain the block.

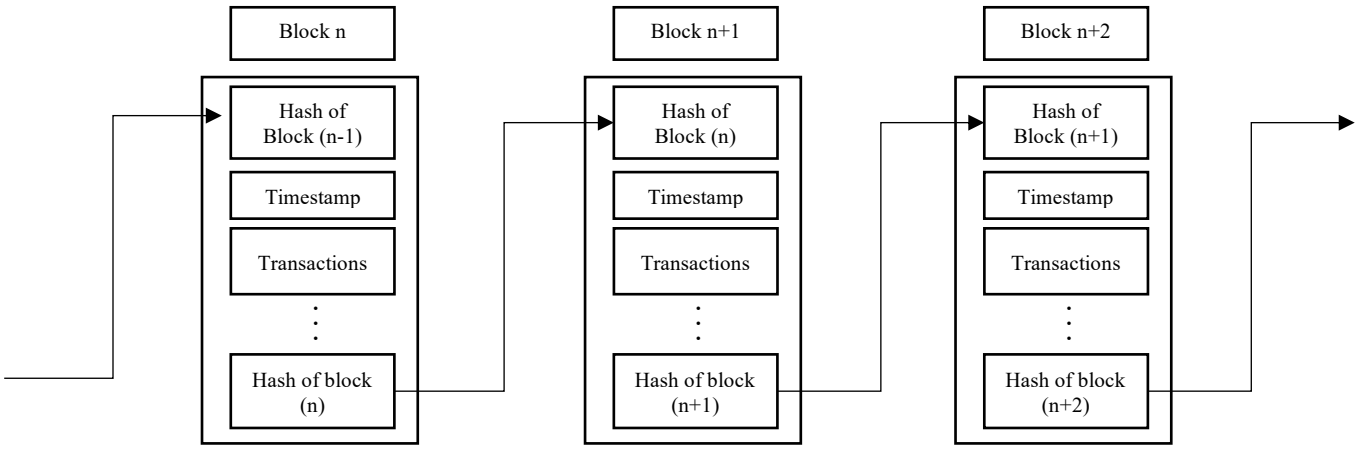


Figure 5. Simplified Example of a blockchain.

C. Mining Blocks

The blockchain is in a decentralized network, where each node contains a copy of the blockchain. The most important nodes in a blockchain are known as the "miner" nodes because they validate each block that is chained to the blockchain. The miners would try to compete in validating the block faster than the other miners in the network (Also known as mining). Mining uses a lot of computational power and memory, which is the essence of security in blockchain. Since there are usually a lot of miners on the network trying to mine the same block, a **consensus mechanism** needs to be put in place to select which miner will add the block to the blockchain. There are many consensus mechanisms but the most famous one, used in the bitcoin blockchain, is known as proof-of-work. Proof-of-work puts in place a very difficult mathematical problem that involves hashing large numbers to meet a certain criterion. For example, the hash of the block must have 16 zero bits in the first 16 digits. So, the miner would add a none sense number also known as the nonce to the block parameters and they would increment the nonce number with every new guess for the hash value. The nonce number is essentially the proof that the miner

worked to mine this block. They would repeat the process until they calculate the block that meets the criteria and add it to the chain. However, there would be many block candidates so only the fastest node would add the new block to the chain. The computation of such a nonce number requires high computing capabilities from the miners, so the winner node would get a reward which acts as an incentive for the miners. The difficulty of the proof-of-work increases when more miners join the blockchain, since there is more competition.

D. Blockchain Fork

In the case of two miners finishing a block at the same time, both blocks would be added to the chain, this is known as a fork in blockchain. However, the next fastest block added to one of the parallel blocks means that this path would be chosen, and the other block would be dropped. In other words, the longer chain wins. Figure 6 shows an illustration of how a blockchain fork is resolved.

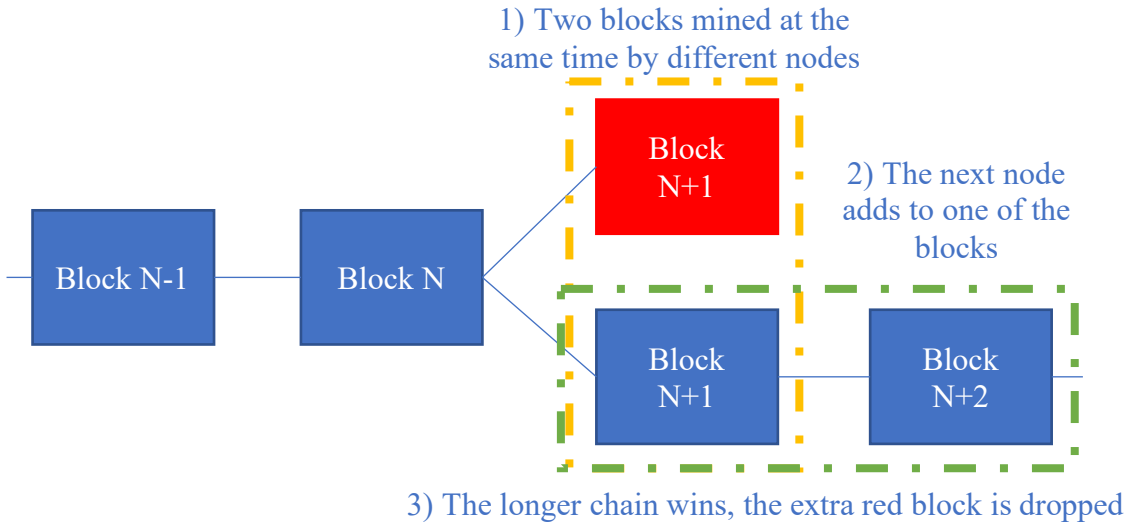


Figure 6. Blockchain Fork Mechanism.

E. Blockchain Transactions

Transactions are the main data in each block. With no transactions, the blockchain would not hold any value. Each blockchain transaction consists of a sender's address, receiver's address,

and the amount to be transferred between the two parties. Transactions are added to the blocks once the miner nodes verify the validity of such transaction. Invalid transactions are voided, and it would not transfer the amount just like a normal credit card transaction. However, credit card transactions are verified by written signatures, how can we replicate this concept virtually? This is where the concept of digital signatures comes into play. Digital signatures are important as it provide nonrepudiation, and it would stop users from spending other users' money. There are a lot of digital signature schemes that are more reliable than a written signature. In blockchain, an asymmetric public key encryption is used where it consists of a public and private key pair. The private key is used by the sender to sign the transaction while the public key is used by the miner nodes to verify the transaction.

E.1. RSA Digital Signature Algorithm

RSA (Rivest–Shamir–Adleman) is an encryption algorithm that is also used for digital signatures. The blockchain could use RSA as the algorithm to digitally sign the transactions. First, the sender generates the public and private keys that would be used for his digital signature. Then the sender would use the private key generated to sign and the public key is public for anyone to verify the signature.

- **Key Generation Steps:**
  - Select two large prime and distinct numbers p and q
  - Calculate n = p*q
  - Calculate λ=lcm(p-1,q-1)
  - Select e, an integer where $1 < e < \lambda$ and gcd(e, λ) = 1
  - Determine d where d*e = 1 modulo λ
  - **Public key:** n and e. **Private key:** p, q, λ, and d

- **Verification Steps:**
  - Sender signs the transaction T using the private key d by computing s = T^d mod n
  - Receiver verifies the signature using the public key e by computing T'= s^e mod n
  - If T' = T then the transaction is valid

The security of RSA algorithm depends on the privacy of p and q and how large are they. For example, RSA256 is easily breakable because n can be factorized retrieving p and q easily. Thus, it is recommended to used RSA4096 to ensure it won't be hackable.

E.2 Elliptic Curves Digital Signature Algorithm

The blockchain could use Elliptic Curve Digital Signature Algorithm (ECDSA) to sign the transactions done. The ECDSA algorithm makes use of arithmetic operations like point addition and scalar multiplication to generate the public and private keys. The plot of an Elliptic curve has the property that a nonvertical line intersecting two non-tangent points will always intersect a third point on the curve. Thus, the point addition can be defined as finding the third point on the curve from the "adding" the two points. In ECDSA, the same operation is done but with a large prime number. The equation of the elliptic curve is $y^2=x^3+ax+b$.

Figure 7 (left) shows the plot of the elliptic curve used by bitcoin with a=0 and b=7. Figure 7 (middle) shows the arithmetic operation of **point addition** in elliptic curves, where P+Q = R. It is calculated by plotting a line that goes through the points P and Q and finding the third point that intersects the curve R'. The third point is reflected by the x axis to find the result of the addition R. Figure 7 (right) shows the arithmetic operation of **point doubling** in elliptic curves, where P+P =R. It is calculated by plotting the tangent line on point P and finding the point that intersects that line with the curve R'. The point R' is then reflected by the x-axis to find the result of the doubling

R. All these operations are used in order to implement **scalar multiplication**, adding the point P to itself x amount of times is equal to x multiplied by P (R = x P).

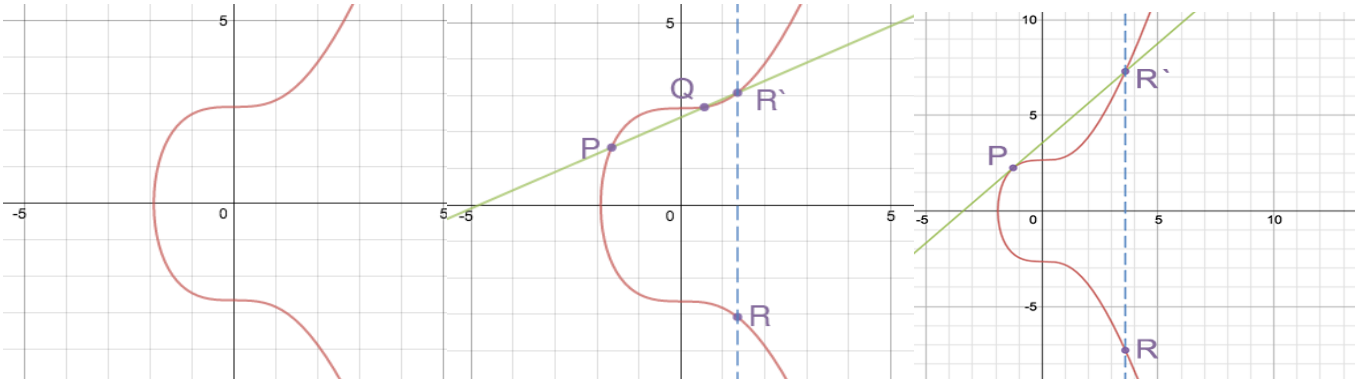


Figure 7. Bitcoin elliptic curve (left), Point addition example (middle), Point doubling example (right)

The application of Elliptic curves on the blockchain depends on the system. A different elliptic curve equation is used for different applications, choosing a value for the parameters a and b, as well as a prime modulo of a finite field, and a base point on the curve along with the order of the base point. The order needs to be a large prime number and that's how the base point is chosen. The security of ECDSA relies on having significantly large numbers of the prime modulo, base point, and the order. Having such large values makes it almost impossible to attack such algorithm and reverse engineer it to find the original values.

Finally, to use elliptic curves for digital signatures, a private key has to be chosen randomly between 1 and the order. The public key is then calculated by using scalar multiplication of the base point by the private key which is given by the equation:

Public Key = Base Point * Private Key

The equation shows that the number of private keys that could be generated is equal to the order. The ECDSA has proven to be secure as it resisted testing for vulnerabilities for very large numbers of the order, prime modulo and the base. The only security flaw is in the protection of the private key or if a static key was chosen as opposed to a randomly generated number every time a new transaction is made.

E.3 Comparing ECDSA and RSA signatures

With small key size of 256 bits ECDSA is more secure while RSA is easily breakable. Thus, signing with ECDSA utilizes smaller computational power as opposed to RSA because it uses smaller key sizes without compromising security. ECDSA is faster at computing the key pair and at signing messages. RSA holds more storage space as it requires to store large key numbers while ECDSA consumes less space [10].

E.4 Digital Signatures Implementation on Python

RSA: PyCrypto is a python library which has functions that allows us to digitally sign messages using RSA [11]. In the example below, RSA 4096 was used as it is very secure, but it was slow when generating the key.

- **Example for signing the message "Blockchain2021" using RSA4096:**

```
>>> from Crypto.Hash import SHA256
>>> from Crypto.PublicKey import RSA
>>> from Crypto.Signature import PKCS1_v1_5
>>> message = b'Blockchain2021'

>>> key = RSA.generate(4096)

>>> with open('private_key.pem', 'wb') as f:
...     f.write(key.exportKey('PEM'))
>>> with open('public_key.pem', 'wb') as f:
...     f.write(key.publickey().exportKey('PEM'))

>>> hashed_message = SHA256.new(message)
>>> signer = PKCS1_v1_5.new(key)
>>> signature = signer.sign(hashed_message)
```

Example for verifying the message in RSA4096:

```
>>> with open('pubkey.pem', 'rb') as f:
...     key = RSA.importKey(f.read())

>>> verifier = PKCS1_v1_5.new(key)
>>> if verifier.verify(hashed_message, signature):
...     print('the signature is valid!')
... else:
...     print('The signature is invalid')
the signature is valid!
```

ECDSA: PyNaCl is a python library that contains functions that uses Ed25519 algorithm, which is a special case of elliptic curves called Edwards-curve Digital Signature Algorithm (EdDSA). This algorithm provides very fast signing with very fast key generation without compromising the security of the keys.

- **Example for signing the message "Blockchain2021" using ECDSA:**

```
>>> import nacl.encoding
>>> import nacl.signing
>>> private_key = nacl.signing.SigningKey.generate()

>>> message = 'Blockchain2021'

>>> signed = private_key.sign(message.encode())
```

- **Example for verifying the message in ECDSA:**

```
>>> public_key = private_key.verify_key

>>> public_key.verify(signed)
b'Blockchain2021'
```

F. E-wallet

A blockchain wallet is a digital wallet that gives the user a sense of reliability as they will have a place to store and manage their money. Blockchains that depend on an account-based model would require the user to create a wallet before interacting with the blockchain. The wallet would contain the user's public and private keys. So, whenever a user would like to transfer any amount to another user in the blockchain, he would need to know the public key of the other user. The famous blockchain Ethereum is based on this model, which makes it a reliable blockchain. It's reliable because only the user holds the private key that grants him access over control of his money in the wallet. Not all blockchains will utilize the concept of a wallet, as the Bitcoin blockchain depends on another model of cash flows [12].

G. Smart Contracts

A smart contract is like a real-life contract except that it is written in a line of codes stored in the blockchain. It contains a predetermined agreement between the sender and the receiver to send a specified amount of money when certain conditions apply. The biggest blockchain platform that supports smart contracts is the Ethereum blockchain, Bitcoin also supports smart contracts but with limited capacity. In Ethereum a special programming language Solidity was written specifically to pass smart contracts in the blockchain. The benefit of a smart contract in the blockchain over a normal physical contract is removing the third party in contracts. The blockchain would validate all the conditions in the contract before execution without the need of a central entity.

H. Blockchain Advantages

H.1 Double spending

The blockchain solves the double spending problem in digital cash without having a central entity managing it. The double spending issue does not happen when it comes to physical money, as once the customer pays the seller physical money, he cannot copy that same cash and reuse it in another transaction. However, this is a huge issue when it comes to digital money. Blockchain helps avoid such problems by having the nodes verify such transactions and ensuring that the token bought and sold is burned and not reused.

H.2 Immutability

The blockchain is immutable, as it is almost impossible to change or edit the blockchain. That is due to the fact that it is in a public distributed ledger. If you change a single block in the chain, you would need to keep building on that block more fraudulent blocks. However, other miners in the network would keep adding to the valid blocks. Thus, the nodes who receive the

blockchain can easily identify the fake blocks. The fraudulent block would be on a shorter chain, since all miners would be faster than a single node sending fake blocks. To be able to fool the whole network, the node must have the power to change at least 50% plus one blockchain stored in all nodes simultaneously[1]. Therefore, the network would be completely secure if there are more honest nodes operating than fraudulent nodes.

H.3 Privacy

Conventional banks keep transactions private by ensuring confidentiality, but the blockchain is broadcasted to all the nodes in the network. Leaving all transactions shared into the public, which might bring up the question of whether the blockchain is private or not. Although the transactions are made public, the addresses of the users who made such transactions are hidden by digital signatures. To ensure absolute privacy a new pair of the public and private keys should be generated with each transaction, so the digital signature would change every time leaving no trace behind. A node on the network will only know the transactions made but would never know who made those transactions. Making the blockchain even better than banks when it comes to privacy.

THE BLOCKCHAIN SERVER

A. Postman

Postman is a platform for API (Application Programming Interface) testing [14]. Postman was used as an HTTP client to send HTTP requests to interact with the blockchain over a peer-to-peer network. In postman, you can create a collection of HTTP requests to be tested on the server. In the blockchain server, only two HTTP request types were used in communicating with the blockchain; "GET" and "POST" requests.

- **GET Request:** A GET request is when the user communicates with the server to retrieve data without changing, deleting, adding, or updating the data in the server. The GET request input is written in the URL only and it does not require any additional input. For example, sending a GET request to view the blockchain.
- **Post Request:** A POST request is when the user posts data on the server. As opposed to a GET request, a POST request modifies the data in the server. Also, the request requires additional input to the URL, which would be inside the body of the POST request. For example, adding a new transaction to the blockchain.
- **HTTP responses:** An HTTP response is the server responding to the user's request. There are universal HTTP response status codes that gives an indication of how the request interacted with the server. Table 2 shows the HTTP status codes that were used or encountered when running the blockchain.

Table 2. HTTP response status codes [15]

Types	Codes	Meaning
Successful Response	200 OK	Successful request
	201 Created	Successful request and a new resource was created
Client Error Response	400 Bad Request	Invalid syntax
	404 not found	Unrecognized URL
Server Error Response	500 Internal Server error	Error in the server
	502 Bad Gateway	Invalid response

B. Flask

Flask is micro web framework used to build web applications written in python [16]. It is used to contain the blockchain, thus the blockchain could be used by anyone online using a server. In python, "app = Flask(__name__)" is written to create our web application. A python decorator "route ()" is then used to define the URL that triggers the function. Each function would return a response in HTML format.

THE BLOCKCHAIN PROGRAM

There are 5 python programs written to run the blockchain:

- *hash.py*
- *PoW.py*
- *signatures.py*
- *Blockchain.py*
- *main.py*

A. Hash Algorithm

The first python program is "*hash.py*" is shown in figure 8. It has only one function which is created to set the hash algorithm to be used in the whole blockchain. The hash algorithm used in this blockchain is SHA3_256 from the python library hashlib. SHA3_256 was used as opposed to SHA256 because it is more secure as mentioned in the previous sections about Hash functions. The string of the input is taken and encoded to put the input in the right format that is expected from the SHA3_256 function. Then the hash of the function is converted to hexadecimal using .hexdigest() function which would result in giving us a string with 64 hex digits as opposed to 256 bits.

```
import hashlib

#hash function
def hash(hash_input):
    hash_output = hashlib.sha3_256(str(hash_input).encode()).hexdigest()
    return hash_output
```

Figure 8. "hash" function in "hash.py".

B. Proof of Work Algorithm

The second python program is "*PoW.py*", which stands for Proof of Work, is shown in figure 9. This program contains the proof of work algorithm used in the blockchain. First, the "*hash.py*" program was called to be implemented in this piece of code. The function proof_of_work takes five inputs, the first input "difficulty" is used to set the difficulty of our algorithm and the other four inputs are the block parameters. This algorithm takes the block

parameters and generates the hash of the block with the condition set by the difficulty parameter. There are four modes for difficulty in this function shown in table 3. To meet the condition of the difficulty variable, a nonce number is added to the input of the hash. The nonce is incremented until the desired condition is met. The output of this function is a list of the hash of the block and the nonce number.

Table 3. Difficulty variable conditions

Difficulty	Condition	Example
0	The generated hash output's first hex digit must be zero	0x0hash
1	The generated hash output's first two hex digits must be zero	0x00hash
2	The generated hash output's first three hex digits must be zero	0x000hash
Not set	The generated hash output's first four hex digits must be zero	0x0000hash

```
from hash import hash

#proof of work methodolody
def proof_of_work(difficulty ,block, timestamp, previous_hash, transactions):
        nonce = 0
        i = False

        if difficulty == 0:
                while i is False:
                        hashed_block = hash(str(block) + str(timestamp) + str(nonce) + str(previous_hash) + str(transactions))
                        if hashed_block[0:1] == '0':
                                i = True
                        else:
                                nonce += 1

        elif difficulty == 1:
                while i is False:
                        hashed_block = hash(str(block) + str(timestamp) + str(nonce) + str(previous_hash) + str(transactions))
                        if hashed_block[0:2] == '00':
                                i = True
                        else:
                                nonce += 1
        elif difficulty == 2:
                while i is False:
                        hashed_block = hash(str(block) + str(timestamp) + str(nonce) + str(previous_hash) + str(transactions))
                        if hashed_block[0:3] == '000':
                                i = True
                        else:
                                nonce += 1
        else:
                while i is False:
                        hashed_block = hash(str(block) + str(timestamp) + str(nonce) + str(previous_hash) + str(transactions))
                        if hashed_block[0:4] == '0000':
                                i = True
                        else:
                                nonce += 1
```

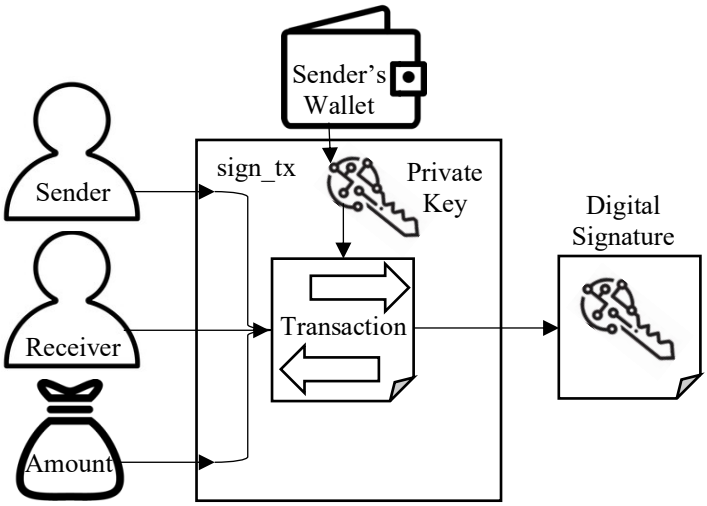
Figure 9. "proof-of-work" function in "PoW.py"

C. Digital Signature Algorithm

The third python program "*signatures.py*" is shown in figures 10, 11, and 12. It has two functions, one to sign a transaction and the other to verify the signature of the transaction. First the library "JSON" is imported as JSON is used to store text and it is mainly used for web applications communicating with a server. In this blockchain .JSON files are used to store the user's private and public keys along with other credentials in an e-Wallet. Then from Pynacl library, nacl libraries are installed for the signature algorithm. The sign_tx function as seen in figure 11, takes the three transaction parameters as the input: the sender, receiver, and amount. It appends all the inputs under a transaction dictionary. Then the function opens the sender's wallet from a JSON file to retrieve the private key. Then, the private key is used to sign the transaction. The output of this function is the signature of the transaction. The verify_tx function as seen in figure 12 takes two inputs, the public key of the sender and the transaction. First it would extract the signature from the transaction, and it would use the public key to verify the transaction. The function would return "False" if the signature is invalid and "True" otherwise.

```
import json
from nacl.encoding import HexEncoder
from nacl.exceptions import BadSignatureError
from nacl.signing import SigningKey, VerifyKey
```

Figure 10. Libraries used in "signatures.py".



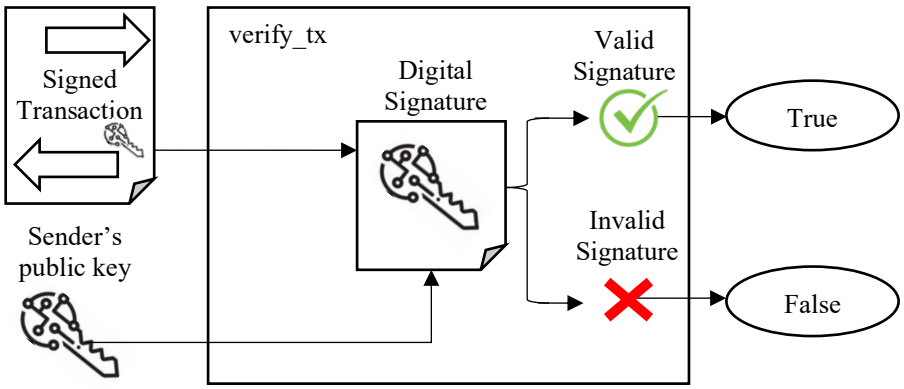

```python
#sign transaction
def sign_tx(sender, receiver, amount):
    transaction = {'sender' : sender,
                   'receiver' : receiver,
                   'amount' : amount }

    with open(f'{sender}.json', 'r') as account:
        wallet = json.load(account)
        private_key = wallet['private_key']

    transaction_bytes = json.dumps(transaction, sort_keys = True).encode("ascii")
    signing_key = SigningKey(private_key, encoder=HexEncoder)
    signature = HexEncoder.encode(signing_key.sign(transaction_bytes).signature).decode("ascii")

    return signature
```

Figure 11. "sign_tx" function in "signatures.py".




```python
#verify transaction
def verify_tx(public_key ,transaction):

    signature = transaction.pop('signature')
    signature_bytes = HexEncoder.decode(signature)

    transaction_bytes = json.dumps(transaction, sort_keys=True).encode("ascii")
    verify_key = VerifyKey(public_key, encoder=HexEncoder)

    try:
        verify_key.verify(transaction_bytes, signature_bytes)
    except BadSignatureError:
        return False
    else:
        return True
```

Figure 12. "verify_tx" function in "signatures.py".

D. Blockchain Class

The fourth program is "*Blockchain.py*", it contains six functions that has all the blockchain main features. First, the previous python programs were imported into this program as seen in figure 13.

```
import datetime
import json
from json.decoder import JSONDecodeError

import requests
from uuid import uuid4
from urllib.parse import urlparse

from nacl.encoding import HexEncoder
from nacl.exceptions import BadSignatureError
from nacl.signing import SigningKey, VerifyKey

from signatures import sign_tx, verify_tx
from PoW import proof_of_work
from hash import hash
```

Figure 13. Libraries imported into "Blockchain.py".

In the beginning of the blockchain class, all of the lists that would be utilized later on in the functions are initialized as empty lists as seen in figure 14. The first list self.chain would contain our blockchain. The second list self.tx would contain the verified transaction to be added to the list. The third list self.pending_tx would contain the transactions yet to be verified by the miner of the block. Then the program would check if there exists a previous blockchain, then it would append it to the empty list self.chain. If a previous blockchain does not exist, then it would create a new one starting with the genesis block. Given that the genesis block is the first block, it would be assigned a value of 0 for the nonce and the previous hash. It would also hold no transactions, and the hash is just the hash of the date and time without including any proof of work. Finally, the set of nodes is initialized empty, so when nodes connect it would populate this set.

```
class Blockchain:

    def __init__(self):
        #empty list initializations
        self.chain = []
        self.tx = []
        self.pending_tx = []

        #loading previous chain or creating the genesis block
        try:
            with open("blockchain.json", "r") as file:
                old_chain = json.load(file)
                chain = old_chain['chain']
                length = old_chain['length']
                self.chain = chain

        except (JSONDecodeError, FileNotFoundError):
            genesis_block = {'block' : 1,
                             'timestamp' : str(datetime.datetime.now()),
                             'previous_hash' : 0,
                             'nonce' : 0,
                             'transactions' :   None,
                             'hash' : hash(str(datetime.datetime.now()))}
            self.chain = [genesis_block]

        #creating the set of nodes
        self.nodes = set()
```

Figure 14. Initializations in "Blockchain.py".

The first function defined in the blockchain class is new_block() as seen in figure 15, it is for creating a new block to be chained to the blockchain. No input parameters are passed to the function. The block parameters are defined here, first we have “block” which is the index of the block. “timestamp” is for printing the time the block was mined. “transactions” would contain all of the contents from self.tx list. “previous_hash” is the previous block’s “hash” value. In order to

```
    #creating a block
    def new_block(self):
        block = len(self.chain) + 1
        timestamp = str(datetime.datetime.now())
        transactions = self.tx
        previous_block = self.last_block()
        previous_hash = previous_block['hash']


        new_block = {'block' : block,
                     'timestamp' : timestamp,
                     'previous_hash' : previous_hash,
                     'transactions' :  transactions,
                     'nonce' : proof_of_work(len(self.nodes),block, timestamp, previous_hash, transactions)[1],
                     'hash' : proof_of_work(len(self.nodes),block, timestamp, previous_hash, transactions)[0]}

        self.tx = []
        self.chain.append(new_block)
        return new_block

    #getting the last block in the blockchain
    def last_block(self):
        return self.chain[-1]
```

Figure 15. "new_block” function in "Blockchain.py".

fetch the last block, another function was defined as last_block() which when triggered gives the output of the last block and that is how the previous hash was retrieved. The current “hash” and “nonce” number is given by passing all the block parameters to the proof of work function with the difficulty set equal to the number of nodes connected to the blockchain. With more nodes connected, the chase to adding a block to the blockchain becomes more competitive and thus the proof of work algorithm gets harder. After filling all the block parameters, the transactions list is emptied out and the new block generated will be appended to our blockchain list. The output of this function is the new block, example of a block in the blockchain is seen in figure 16 and 17.


```
"block": 17,
"hash": "06058780cbf1c295d11d0cf7804046038c2b08c2960dd6a7c90aa3cff85843b5",
"message": "All transactions have been verified and a new block has been added to
    the blockchain",
"nonce": 3,
"previous_hash":
    "054a73bc2904c92d71e7f91642522d7e1d13b9b297675e267ba1e02c7666fe6e",
"timestamp": "2021-06-03 07:38:11.008027",
"transactions": [
    {
        "amount": 1,
        "receiver":
            "38a94988e07e6b465b18c485d3a1375f5a491e913998d59830d613da35234e45",
        "sender":
            "7439416d6ff05003e1774beaa7a26cda6be15354f01d02f19b8dca6b4ce29ebf",
        "signature":
            "ce5c6443e92aecd92ff729a60ddfdc52286b6c9ae37238f7c5b1c75ae15a9041d01fe
            cde3a95a7f4e6b2877257ffcafdab3c404c42550adfba483886e3b4a903"
    },
    {
        "block_reward": 1,
        "miner": "458c22ffd7954c14929b2cb9ec7b4016"
    }
]
```

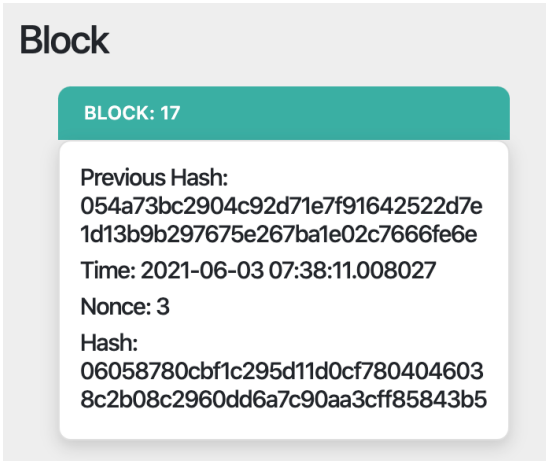



Figure 16. Example of a block in blockchain in Postman

The next function is to verify the transactions “verified_tx”. The function does not have a required input as it would check the transactions in the list pending_tx. It will loop all over the transactions that were pending and check for their validity. There are three conditions that are being checked before verifying the transaction:

1) Check whether the sender has enough balance to send coins to the receiver.
2) Check if the signature given in the transaction is valid by passing it through the verify_tx function defined in “*signatures.py*”.

3) Check whether the receiver has enough tokens to sell to the sender.

If all conditions are met, then the transaction is verified and appended to self.tx which are the transactions that would be included in the next mined block. The transaction would print the sender's and receiver's public key addresses instead of their names. Additionally, the sender's wallet will be modified to add the purchased tokens and subtract the number of coins used, and the receiver's wallet will be modified vice versa. The function can be seen in figure 17.

```
#Verifying transactions
def verified_tx(self):
    index = 0
    while index < len(self.pending_tx):
        transaction = self.pending_tx[index]
        sender = transaction['sender']
        receiver = transaction['receiver']
        amount = transaction['amount']
        signature = transaction['signature']

        try:
            with open(f'{sender}.json' , 'r') as account:
                s_wallet = json.load(account)
                try:
                    with open(f'{receiver}.json' , 'r') as account:
                        r_wallet = json.load(account)

                    if (int(amount) <= int(s_wallet['coins'])) and verify_tx(s_wallet['public_key'], transaction) and (int(amount) <= int(r_wallet['tokens'])) is True:
                            new_s_balance = int(s_wallet['coins']) - int(amount)
                            new_s_tokens = int(s_wallet['tokens']) + int(amount)
                            new_s_wallet =  {"user" : s_wallet['user'],
                                             "password_hash" : s_wallet['password_hash'],
                                             "private_key": s_wallet['private_key'],
                                             "public_key": s_wallet['public_key'],
                                             "coins" : new_s_balance,
                                             "tokens" : new_s_tokens}
                            with open(f'{sender}.json' , 'w' ) as account:
                                json.dump(new_s_wallet, account)

                                with open(f'{receiver}.json' , 'r') as account:
                                    r_wallet = json.load(account)
                                    new_r_balance = int(r_wallet['coins']) + int(amount)
                                    new_r_tokens = int(r_wallet['tokens']) - int(amount)
                                    new_r_wallet = {"user" : r_wallet['user'],
                                                    "password_hash" : r_wallet['password_hash'],
                                                    "private_key": r_wallet['private_key'],
                                                    "public_key": r_wallet['public_key'],
                                                    "coins" : new_r_balance,
                                                    "tokens" : new_r_tokens}
                                    with open(f'{receiver}.json' , 'w') as account:
                                        json.dump(new_r_wallet, account)

                                        self.tx.append({'sender' : s_wallet['public_key'],
                                                        'receiver': r_wallet['public_key'],
                                                        'amount' : amount,
                                                        'signature' : signature})
                                        index += 1
                    else:
                        index += 1
                except (JSONDecodeError, FileNotFoundError):
                    index += 1
        except (JSONDecodeError, FileNotFoundError):
            index += 1
```

Figure 17. "verified_tx" function in "Blockchain.py".

The password check function seen in figure 18 is used to authenticate the credentials of the user to access the wallet. It takes four inputs, the user and password that was inputted and the actual_user and actual_password that was saved in the wallet. The inputted password is then hashed in order to compare it with the stored password hash in the wallet. This function returns True if the credentials matched and false otherwise.

```
def password_check(self, user, password, actual_user, actual_password):
    hashed_password = hash(password)
    if str(user) == str(actual_user) and hashed_password  == actual_password:
        return True
    else:
        return False
```

Figure 18. “password_check” function in "Blockchain.py"

The check_chain function as seen in figure 19 is used to check if your chain is valid or if it had been corrupted. it takes the blockchain as the input and it would loop all over the blocks in the blockchain to check whether the previous hash of the block matches the hash of the previous block. It would return False if there was a discrepancy and it would return True otherwise.

```
#check our blockchain
def check_chain(self, chain):
    block_index = 1

    while block_index < len(chain):
        block = chain[block_index]
        last_block = chain[block_index - 1]
        if block['previous_hash'] != last_block['hash']:
            return False
        block_index += 1


    return True
```

Figure 19. “check_chain” function in "Blockchain.py"

The new_node function as seen in figure 20 is used to add new nodes to our blockchain network. The input of the function is the address of the node to be added. The address is in the form of “http://127.0.0.1:{port#}”. The input address would be then parsed using the urlparse()

function. In order to extract the address of the node the parsed url is passed by .netloc. Once it is extracted, the address is added to the nodes set.

```
#add nodes in the network
def new_node(self, address):
    parsed_url = urlparse(address)
    self.nodes.add(parsed_url.netloc)
```

Figure 20. “new_node” function in "Blockchain.py"

The last function in the Blockchain class is the fork methodology and it is shown in figure 21. No input is required for this function. It checks all of the connected nodes to the blockchain and compares the length parameter in the list self.chain. If there exists a chain in another node that is longer than the chain on the current node then the chain would be replaced, and it would return a “True” output. If the current chain is the longest chain, then nothing would be changed, and the function would return a “False” output.

```
#Fork methodology
def fork(self):
    network = self.nodes
    longest_chain = None
    max_length = len(self.chain)
    for node in network:
        response = requests.get(f'http://{node}/view_chain')
        if response.status_code == 200:
            length = response.json()['length']
            chain = response.json()['chain']
            if length > max_length and self.check_chain(chain):
                max_length = length
                longest_chain = chain
    if longest_chain:
        self.chain = longest_chain
        return True
    return False
```

Figure 21. “fork” function in "Blockchain.py"

E. Main Program

In “*main.py*” the previous python programs are imported along with the necessary libraries as shown in figure 22. In this piece of code, the library Flask is used to set up the blockchain server. Once a server is set up, the node working on this specific port needs an address. The node needs an address because once a miner mines a block, it can receive a block

reward “incentive” and it is sent to the miner from the node address. After that the “Blockchain” class is introduced as “blockchain” and will be used throughout the code.

```
import datetime
import json
from json.decoder import JSONDecodeError
from flask import Flask, jsonify, request
import requests
from uuid import uuid4
from urllib.parse import urlparse

from nacl.encoding import HexEncoder
from nacl.exceptions import BadSignatureError
from nacl.signing import SigningKey, VerifyKey

from signatures import sign_tx, verify_tx
from hash import hash
from Blockchain import Blockchain

#postman commands:
app = Flask(__name__)

#retrieving node address
node_address = str(uuid4()).replace('-','')
blockchain = Blockchain()
```

Figure 22. Libraries import in "main.py".

All the functions in “main.py” are to set the URL to interact with the blockchain server. The first function is seen in figure 23. This function sets the URL “/view_chain” as a “GET” HTTP request. Once the user sends the request, he would be able to view the blockchain in postman. If the request had no error, it would send back an HTTP response of 200 (OK). Embedded in this request the blockchain will be saved in a .JSON file so it can be retrieved later once the blockchain server is terminated.

```
#checking the chain and saving it
@app.route('/view_chain', methods = ['GET'])
def view_chain():
    response = {'chain' : blockchain.chain,
                'length' : len(blockchain.chain)}

    with open("blockchain.json", "w") as file:
        json.dump(response, file)
    return jsonify(response), 200
```

Figure 23. “view_chain” function in "main.py".

The next URL is to set up a wallet for the user "/new_user" as seen in figure 24. The URL is a "POST" request where the user needs to write additional information to the URL link. The information written by the user needs to be in. JSON. The additional information required are the credentials needed to open the user's wallet later. The credentials are the username, user type and the password. The username and type are stored in plaintext while the password is hashed first before storing it in the wallet. There are three types of users in this blockchain, a customer, generator, and a prosumer. The customer will have no tokens as he has no electricity to generate. The generator will get 10 tokens initially as he would generate electricity and sell tokens to the customers. The prosumer would have 5 tokens as the generation capability would be less than the generator but still, he would be able to sell tokens. Then the private key is randomly generated using SigningKey.generate() function from the nacl library. Then the public key would be generated from the private key by using the verify_key function.

```
#user generating their wallet
@app.route('/new_user', methods = ['POST'])
def new_user():
    credentials = request.get_json()
    private_key = SigningKey.generate()
    public_key = private_key.verify_key

    hashed_password = hash(credentials['password'])
    user_type = credentials['type']

    if user_type == 'customer':
        tokens = 0
    elif user_type == 'generator' :
        tokens = 10
    elif user_type == 'prosumer':
        tokens = 5
    else:
        response = 'please enter a valid user type'
```

Figure 24. "new_user" function in "main.py" 1/2.

Once all parameters are retrieved the wallet is defined as seen in figure 25. The private and public key are stored as hexadecimal digits in the wallet. Then the wallet is stored in a .JSON file named as username.JSON so each user will have a wallet under his name. If a wallet already exists under that name, then the response to this function would be “wallet already exists”. If not, then the response would preview the wallet. This POST request returns an HTTP response of 201 (created) as it created a new wallet.

```
    wallet = {"type" : credentials['type'],
              "user" : credentials['user'],
              "password_hash" : hashed_password,
              "private_key": private_key.encode(encoder=HexEncoder).decode(),
              "public_key": public_key.encode(encoder=HexEncoder).decode(),
              "tokens" : tokens,
              "coins" : 0}


    user = credentials['user']

    try:
        with open(f'{user}.json', 'r') as file:
            response = 'wallet already exists'
    except (JSONDecodeError, FileNotFoundError):
        with open(f'{user}.json', 'w') as file:
            json.dump(wallet, file)
            response = wallet


    return jsonify(response) , 201
```

Figure 25. “new_user” function in "main.py" 2/2.

The next URL is /view_wallet as seen in figure 26. This request is set for the user to view their wallet. It is a “POST” request as the user needs to input additional data to the URL which is his credentials. The user cannot access his wallet without providing correct credentials, because it has sensitive data such as the private key. The function password_check from “*Blockchain.py*” is used to authenticate the user. If the user wrote the wrong credentials, the response provided in

Postman would be “wrong username or password”. If the credentials were correct, then the user would view the wallet and it would send an HTTP response of 200 (OK).

```
#user checking their wallet
@app.route('/view_wallet', methods = ['POST'])
def view_wallet():

    credentials = request.get_json()
    user = credentials['user']

    try:
        with open(f'{user}.json' , 'r') as account:
            wallet = json.load(account)
            if  blockchain.password_check(credentials['user'], credentials['password'], wallet['user'],wallet['password_hash']) is True:
                response = {'private_key' :  wallet['private_key'],
                            'public_key' : wallet['public_key'],
                            'coins' : wallet['coins'],
                            'tokens' : wallet['tokens']}
            else:
                response = 'wrong username or password'
    except (JSONDecodeError, FileNotFoundError):
        response = 'wrong username or password'

    return jsonify(response), 200
```

Figure 26. “view_wallet” function in "main.py".

The fourth URL is “/deposit” as seen in figure 27. It is a “POST” request as the user needs to input his credentials along with the number of coins to be deposited into his account. The wallet will be updated with new amount and a response of 201 (created) will be returned. In postman the new wallet would be previewed.

```
#user depositing in the wallet
@app.route('/deposit', methods = ['POST'])
def deposit():
    credentials = request.get_json()
    user = credentials['user']

    try:
        with open(f'{user}.json' , 'r') as account:
            wallet = json.load(account)
            coins = wallet['coins']

            if  blockchain.password_check(credentials['user'], credentials['password'], wallet['user'],wallet['password_hash']) is True:
                new_balance = int(wallet['coins'])+int(credentials['deposit'])
                response =  { 'private_key' : wallet['private_key'],
                              'public_key' : wallet['public_key'],
                              'coins' : new_balance }

                new_wallet = {'user' : wallet['user'],
                              'password_hash' : wallet['password_hash'],
                              'private_key' : wallet['private_key'],
                              'public_key' : wallet['public_key'],
                              'coins' : new_balance,
                              'tokens' : wallet['tokens']}

                with open(f'{user}.json', 'w') as account:
                    wallet = json.dump(new_wallet, account)
            else:
                response = 'wrong username or password'

    except (JSONDecodeError, FileNotFoundError):
        response = 'wrong username or password'

    return jsonify(response), 201
```

Figure 27. “deposit” function in "main.py".

The fifth URL is “/mine” as seen in figure 28 and it is for mining block. The HTTP request type is a “GET” request and the methodology on how it works can be seen in figure 29. Once the new block have been generated a message is printed on Postman “All transactions have been verified and a new block has been added to the blockchain” and an HTTP response of 201 (created) is returned.

```
#minning blocks algorithm
@app.route('/mine', methods = ['GET'])
def mine_block():
    blockchain.verified_tx()
    blockchain.pending_tx = []
    transaction = { 'sender' : node_address,
                    'receiver' : 'miner',
                    'block_reward' : 1}
    with open(f'{port}.json', 'r') as account:
        balance = json.load(account)
        new_coin = 1 + int(balance['coins'])
        new_balance = { 'coins' : new_coin }
    with open(f'{port}.json', 'w') as account:
        json.dump(new_balance , account)

    blockchain.tx.append(transaction)

    block = blockchain.new_block()
    response = {'message' : 'All transactions have been verified and a new block has been added to the blockchain',
                'block' : block['block'],
                'timestamp' : block['timestamp'],
                'nonce' : block['nonce'],
                'previous_hash' : block['previous_hash'],
                'transactions': block['transactions'],
                'hash' : block['hash']}
    return jsonify(response), 200
```

Figure 28. “mine” function in "main.py".

The steps for mining a block are:

1) Verify the pending transactions using verified_tx function:
    a. Verify the digital signatures using verify_tx function
    b. Verify if there is enough balance in the wallet
2) Generate a new coin as a reward for the miner
3) Run the proof of work algorithm on the block using proof_of_work function
4) Generate the new block and append it to the blockchain using new_block function

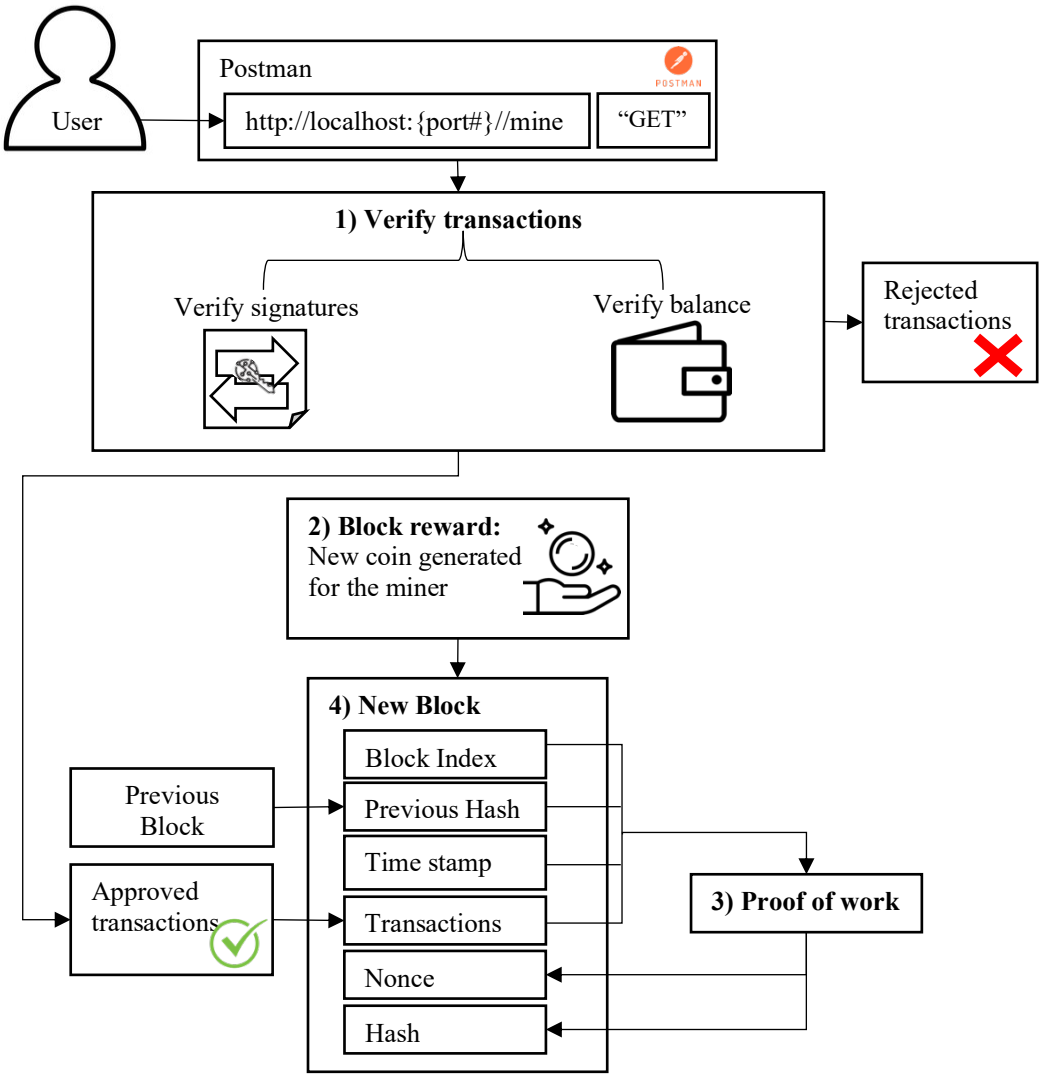


Figure 29. Mining algorithm.

The sixth URL is "/add_transaction" as seen in figure 30. It is a "POST" request as the sender needs to input his credentials and the receiver's name along with the number of coins to be sent to the receiver. If the credentials are correct then the transaction is signed using sign_tx function and the transaction is appended to pending_tx list. The response printed in postman would be a message saying that the transaction will be added once a block is mined and the transaction has been verified. A response of 201 (created) will be returned.

```python
#adding a new transaction
@app.route('/add_transaction', methods = ['POST'])
def add_transaction():
    tx = request.get_json()

    sender = tx['sender']
    receiver= tx['receiver']
    amount = tx['amount']

    try:
        with open(f'{sender}.json' , 'r') as account:
            s_wallet = json.load(account)
            try:
                with open(f'{receiver}.json' , 'r') as account:
                    r_wallet = json.load(account)
                    if  blockchain.password_check(tx['sender'], tx['password'], s_wallet['user'],s_wallet['password_hash']) is True:


                        signature = sign_tx(sender, receiver, amount)

                        blockchain.pending_tx.append({'sender' : sender,
                                                      'receiver': receiver,
                                                      'amount' : amount,
                                                      'signature' : signature})

                        last_block = blockchain.last_block()
                        block = last_block['block'] + 1
                        response = f'{sender} sends {amount} coins to {receiver} and gets {amount} tokens, transaction will be added in block {block} after verification'

                    else:
                        response = 'wrong username or password'
            except (JSONDecodeError, FileNotFoundError):
                response = 'receivers wallet does not exist'
    except (JSONDecodeError, FileNotFoundError):
        response =  'wrong username or password'

    return jsonify(response), 201
```

Figure 30. “add_transaction” function in "main.py".

The seventh URL is “/validate_chain” as seen in figure 31. It is a “GET” request and the functions calls the check_chain function from the Blockchain class. If the check_chain function returns a True value it would send a response of “The chain is valid”, otherwise it would send “The chain is corrupt”. A response of 200 (OK) will be returned.

```python
#checking if the chain is valid
@app.route('/validate_chain', methods = ['GET'])
def validate_chain():
    chain = blockchain.check_chain(blockchain.chain)
    if chain is True:
        response = {'message' : 'The chain is valid'}
    else:
        response = {'message' : 'The chain is corrupt'}
    return jsonify(response), 200
```

Figure 31. “validate_chain” in "main.py".

The eighth URL is “/node” as seen in figure 32. This function connects new nodes to the blockchain server. It is a “POST” request as the user needs to input the node address to be included in the blockchain. The new node address would be appended in the nodes set and an HTTP 201 (created) response would be returned. If the user did not input a node address, the function would return 400 (Bad Request) which means the syntax was invalid.

```
#adding a new node to the network
@app.route('/node', methods = ['POST'])
def node():
    json = request.get_json()
    nodes = json.get('nodes')
    if nodes is None:
        return "No node", 400
    for node in nodes:
        blockchain.new_node(node)
    response = {'message': 'all nodes are connected, blockchain contains:',
                'total_nodes' : list(blockchain.nodes)}
    return jsonify(response), 201
```

Figure 32.“node” function in "main.py".

The ninth URL is “/update_chain” as seen in figure 33. This is where the node would check the longest chain and update his own chain. It is a “GET” request and it calls the fork function from the Blockchain class. If the fork function returns a True value, it would update the blockchain and send a response of “chain was replaced”. Otherwise, if the fork returns a False value, it won’t replace the chain instead it would send a response of “chain is the longest nothing was replaced”. A response of 200 (OK) will be returned.

```
#replacing shorter chains with longer ones
@app.route('/update_chain', methods = ['GET'])
def update_chain():
    update = blockchain.fork()
    if update:
        response = {'message' : 'chain was replaced',
                    'new_chain': blockchain.chain}
    else:
        response = {'message' : 'chain is the longest nothing was replaced',
                    'actual_chain' : blockchain.chain}
    return jsonify(response), 200
```

Figure 33. “update_chain” function in "main.py".

The piece of code in figure 34 helps to run the server on the assigned port number. On the terminal, the user can run main.py on port number 5000 by running this command “python main.py -p 5000” if no port number was defined then the default port number is 5000.

```
if __name__ == '__main__':
    from argparse import ArgumentParser

    parser = ArgumentParser()
    parser.add_argument('-p', '--port', default=5000, type=int, help='port to listen on')
    args = parser.parse_args()
    port = args.port
    try:
        with open (f'{port}.json', 'r') as account:
            json.load(account)
    except (JSONDecodeError, FileNotFoundError):
        with open (f'{port}.json', 'w') as account:
            balance = {'coins' : 0 }
            json.dump(balance, account)
    app.run(host='0.0.0.0', port=port)
```

Figure 34. Code assigning the port number.

PROCEDURE

To run the server, you need to open the terminal and write the command “python3 main.py” as shown in figure 35. Then you need to open Postman to interact with the server. and run the already set up HTTP requests to the server.

```
shaikha@Shaikhas-MBP Final % python3 main.py
 * Serving Flask app 'main' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://192.168.1.47:5000/ (Press CTRL+C to quit)
```

Figure 35. Starting up the server in the terminal.

The Postman user interface is shown in figure 36, numbered fields are as follows:

1) Choose the type of the request, in this case whether it is a GET or POST request
2) Enter the URL for the request, in this case there are nine options:
    a. http://localhost:{port#}/new_user
    b. http://localhost:{port#}/deposit
    c. http://localhost:{port#}/view_wallet
    d. http://localhost:{port#}/add_transaction
    e. http://localhost:{port#}/mine
    f. http://localhost:{port#}/view_chain
    g. http://localhost:{port#}/node
    h. http://localhost:{port#}/update_chain
    i. http://localhost:{port#}/validate_chain
3) If it is a POST request enter the additional information in the body of the request
4) Ensure the additional are in JSON format

5) Send the HTTP request

6) The response is shown in the field below

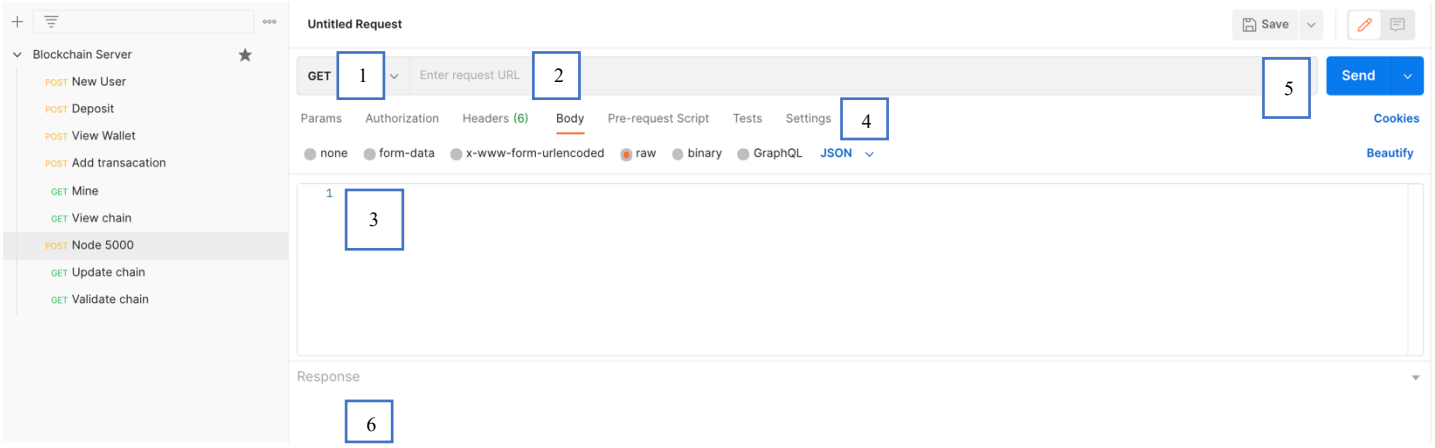


Figure 36. Postman user interface.

A. Adding new users

To Create the new users that will be sending transactions over the blockchain, use /new_user URL on Postman and choose a “POST” request as shown in figures 37and 38. You need to input the credentials and type of the user (customer, generator, or prosumer) in the body of the request and ensure it is in JSON format.

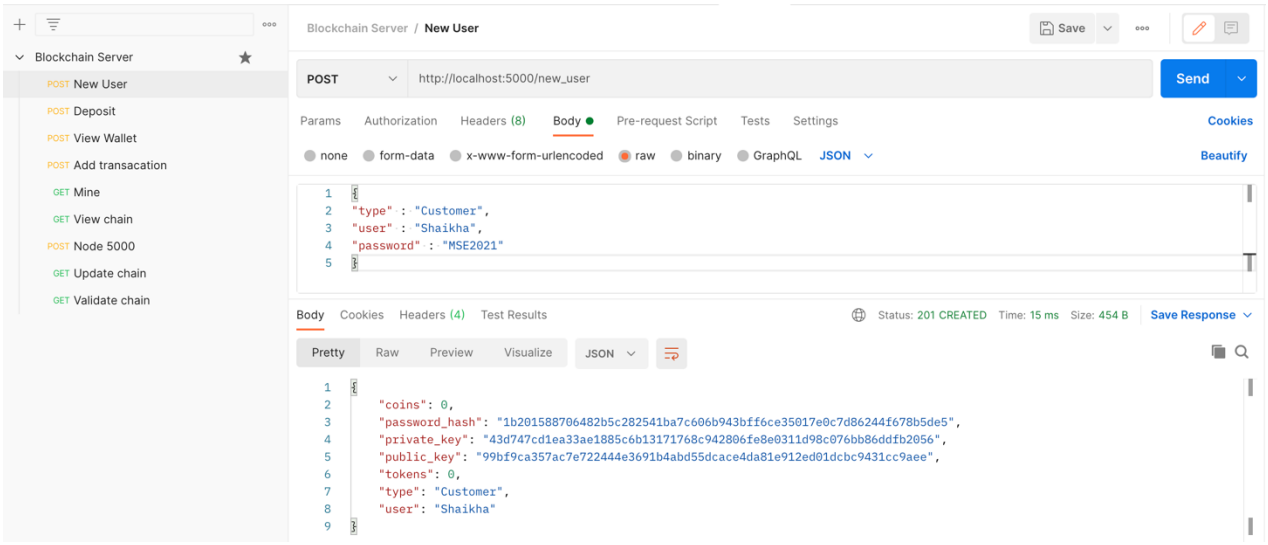


Figure 37. Creating a new user "customer".

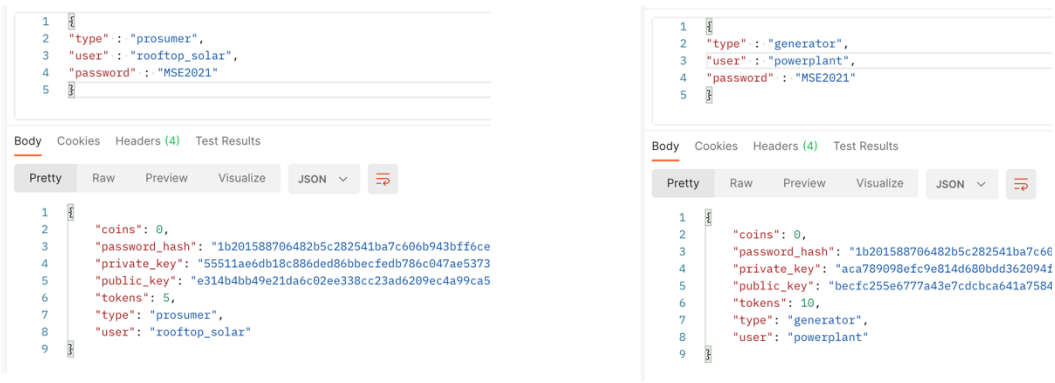


Figure 38. Creating a new user "prosumer" and a new “generator”

B. Deposit to the wallet

You need to deposit coins into the user’s wallet by using the URL “/deposit” and choosing “POST” request as shown figure 39. You need to input the credentials and number of coins to be deposited in the body of the request and ensure it is in JSON format.

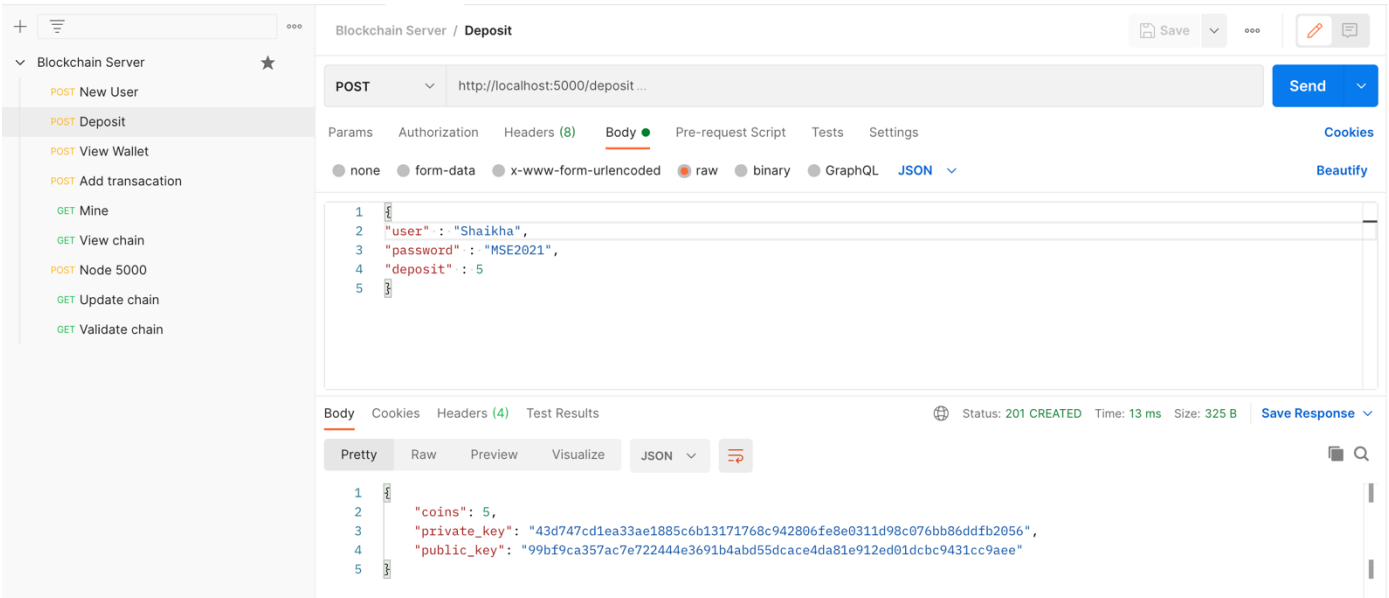


Figure 39. Depositing coins into the wallet.

C. View the wallet

The user can then view the wallet by using "/view_wallet" and choosing "POST" request as shown in figure 40. Here the user only needs to input his credentials to be able to view the wallet.

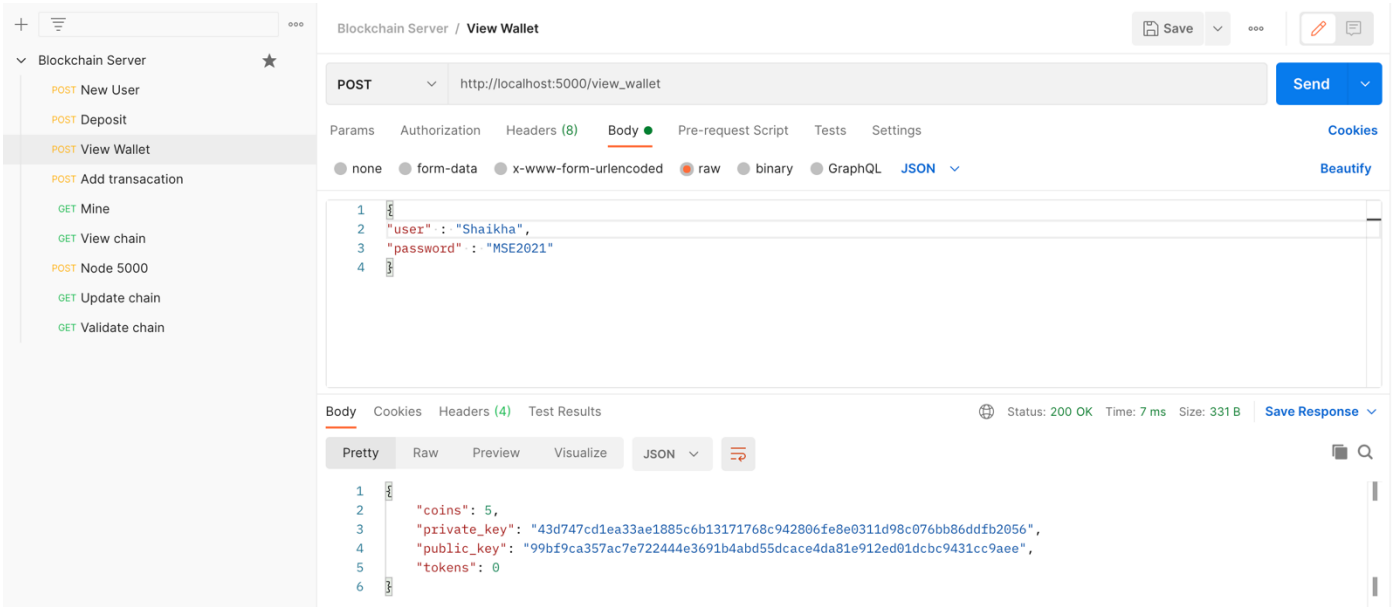


Figure 40. The user viewing the wallet.

D. Add transactions to the blockchain

The user can add a transaction by using the URL extension /add_transaction and choose "POST" as the HTTP request type, as seen in figures 41 and 42. The user needs to specify in the body the sender(username), receiver, amount to be transferred and his password. The transaction

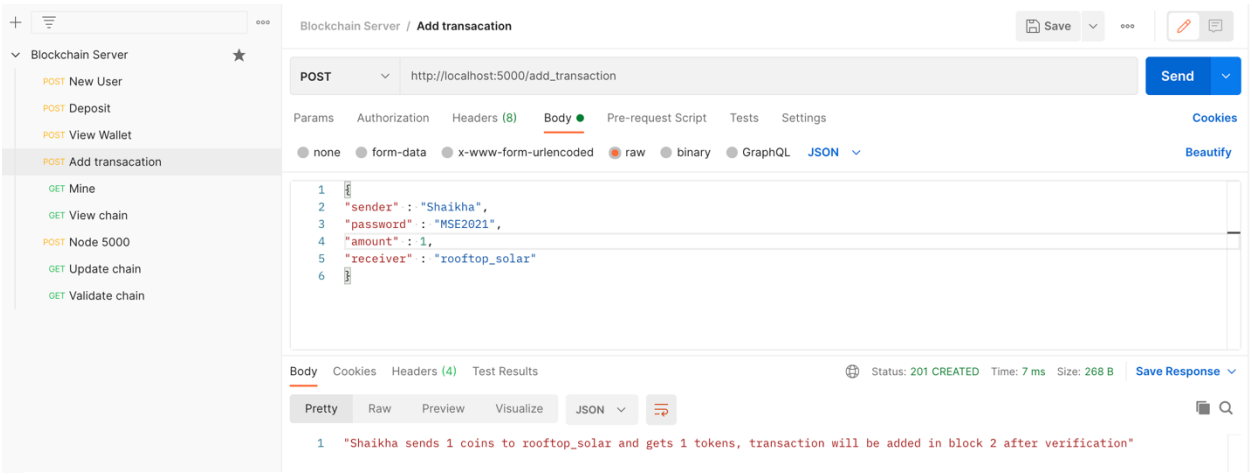


Figure 41, Customer sending coins to a prosumer and getting tokens.

won't go through until it has been verified by the miner and so if the user views the wallet, no changes will be done yet.

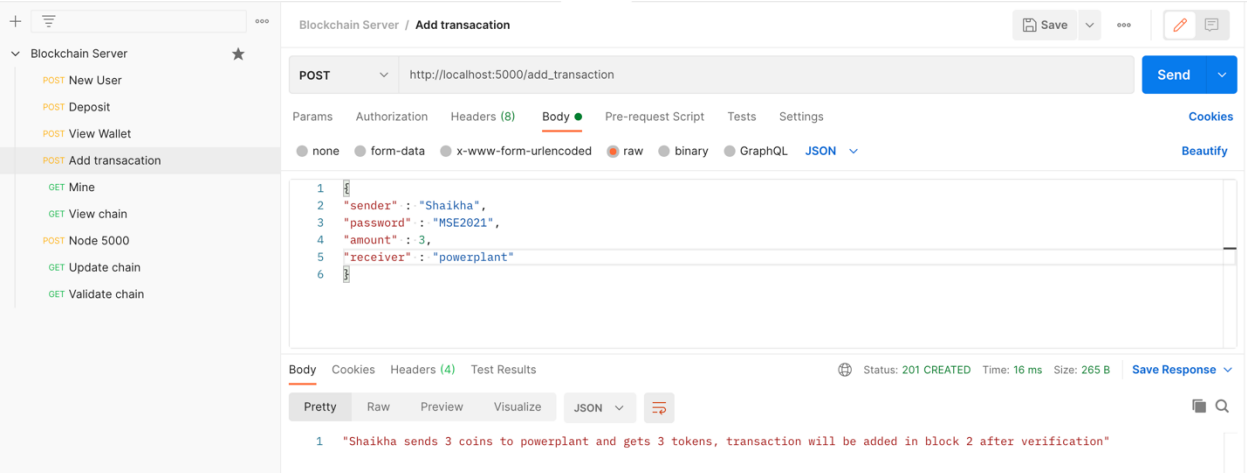


Figure 42. Customer sending coins to a generator and getting tokens.

E. Mine new blocks

To mine a new block the URL extension is /mine and it is a "GET" request so the user will not input any additional information as seen in figure 44. The response would be the new mined block. In "pretty" format you can see the details of the block, all the transactions and the block reward for mining. Also, you can see the block header information in short format when clicking on "visualize" as seen in figure 43. There is only one node connected to the blockchain, so the proof of work algorithm is set to easy. The proof of work sets the hash value to begin with one zero hexadecimal digit, as seen from the hash of the block. After the block has been mined, the miner has verified all the transactions when mining, so if the user views the wallet now, there will be deductions and changes.

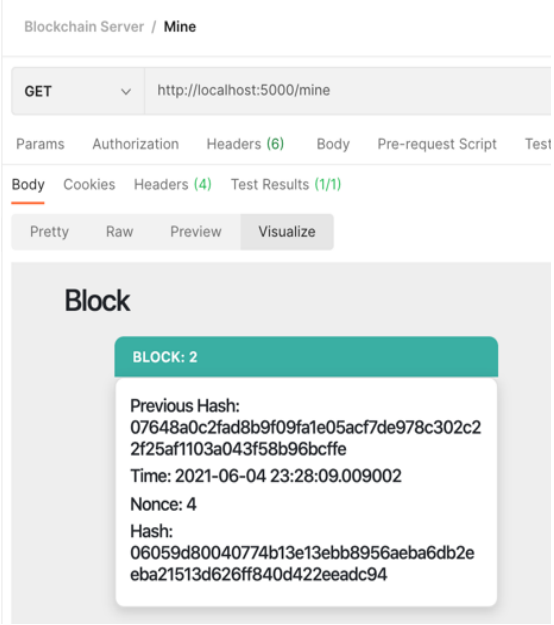


Figure 43. Block header.

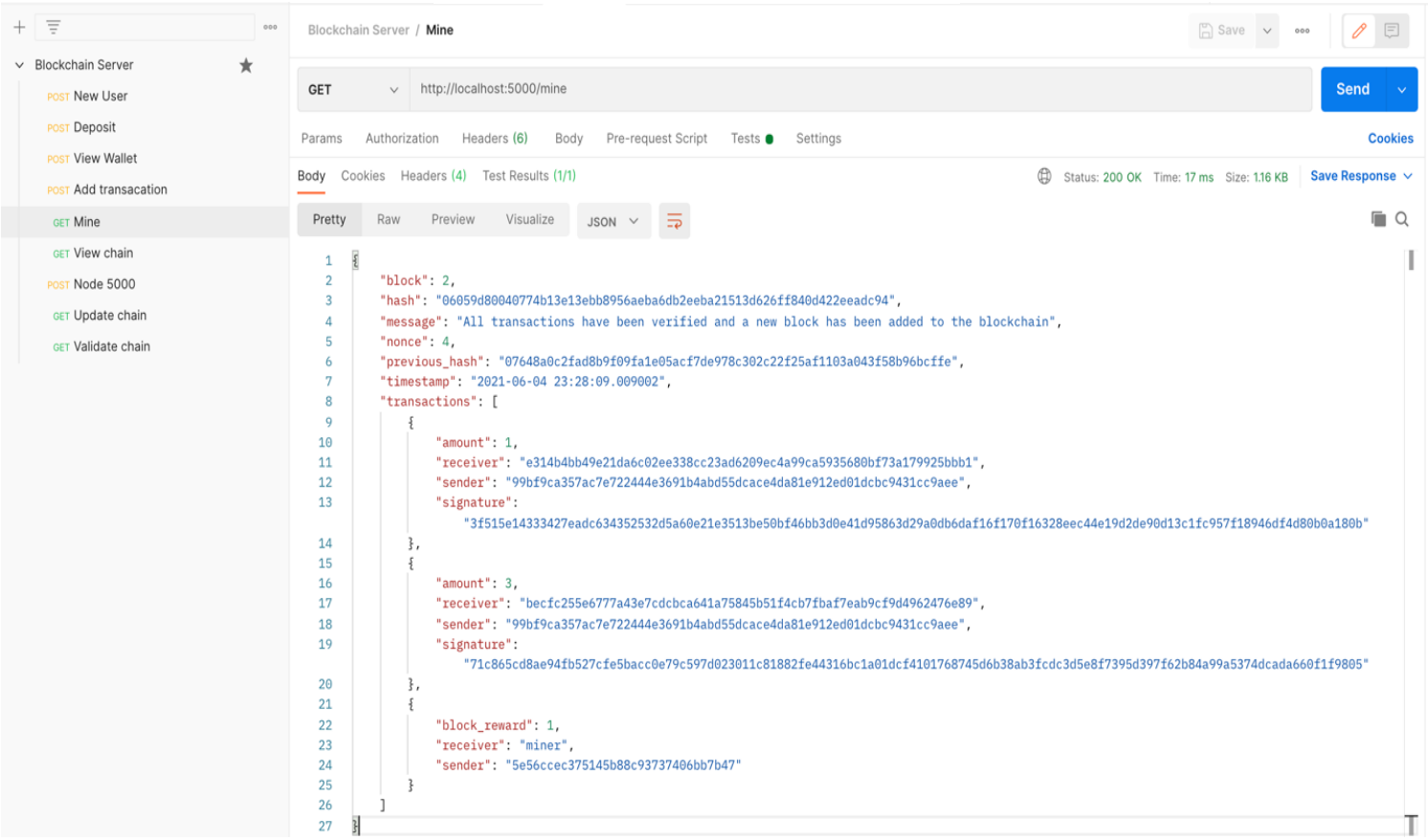


Figure 44. Mining a block

F. Viewing the blockchain

To view the whole blockchain, the user needs to input the URL extension of /view_chain and choose “GET”. No additional information required for this request. Figure 45 shows the response in visualize format. You can see that there exists the genesis block (first block) as it is hard coded into the blockchain with a previous hash value of zero and nonce value of zero since it was not mined.

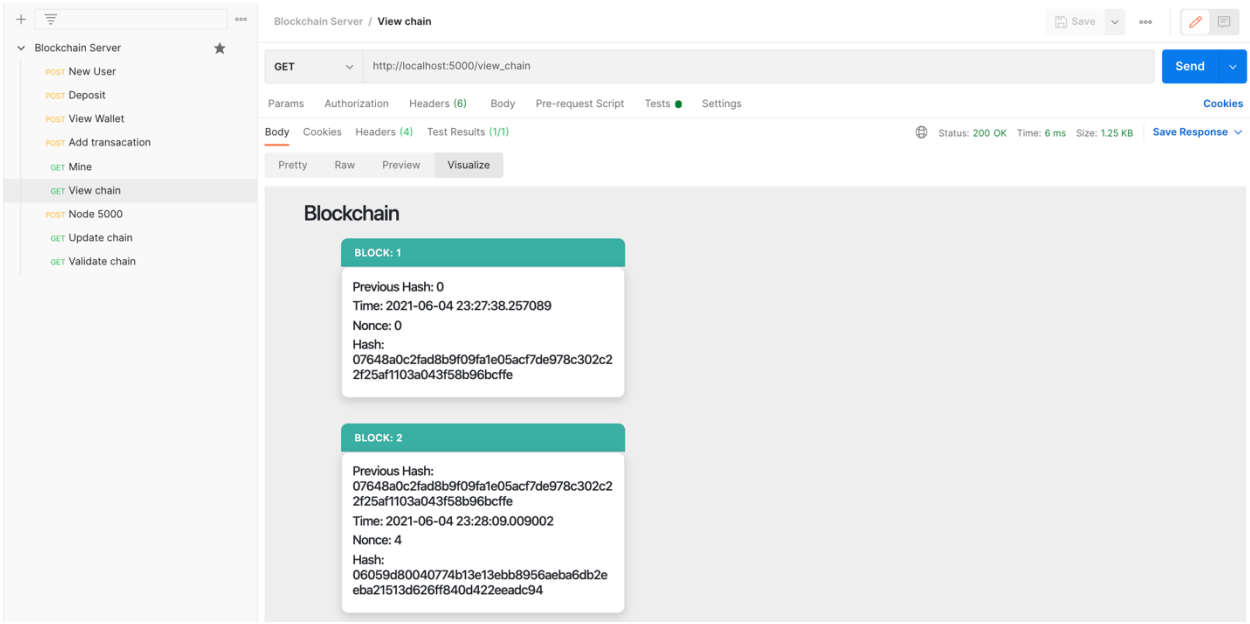


Figure 45. The blockchain.

G. Create new node to join the blockchain server

To create a new a node to join the blockchain server, a new connection needs to be established. Type the command line “python3 main.py -p 5001” in the terminal as seen in figure 46.

```
shaikha@Shaikhas-MBP Final % python3 main.py -p 5001
 * Serving Flask app 'main' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://192.168.1.47:5001/ (Press CTRL+C to quit)
```

Figure 46. Command for starting up a new node.

The first node is currently working on the default port 5000, so the new node will work on port 5001. Then on Postman you input the URL extension /node and choose “POST” as seen in figure 47. In the body of the request, you type the address of the node to be connected. You must do it on both nodes, connect node 5000 to 5001 and vice versa. Once a new node joins the network the proof of work algorithm will increase in difficulty. Figure 48 shows the block header of a block mined on port 5001, you can see the hash value now has two zero hexadecimal digits in the beginning as opposed to only one digit when there was only one node. If you add more nodes the proof of work gets even more difficult as shown in figures 49 and 50.

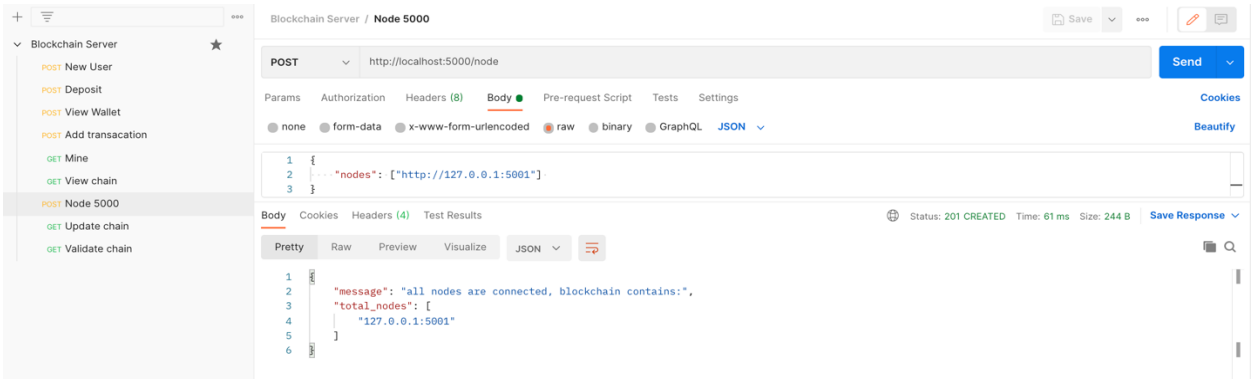


Figure 48. Connecting node 5000 to node 5001

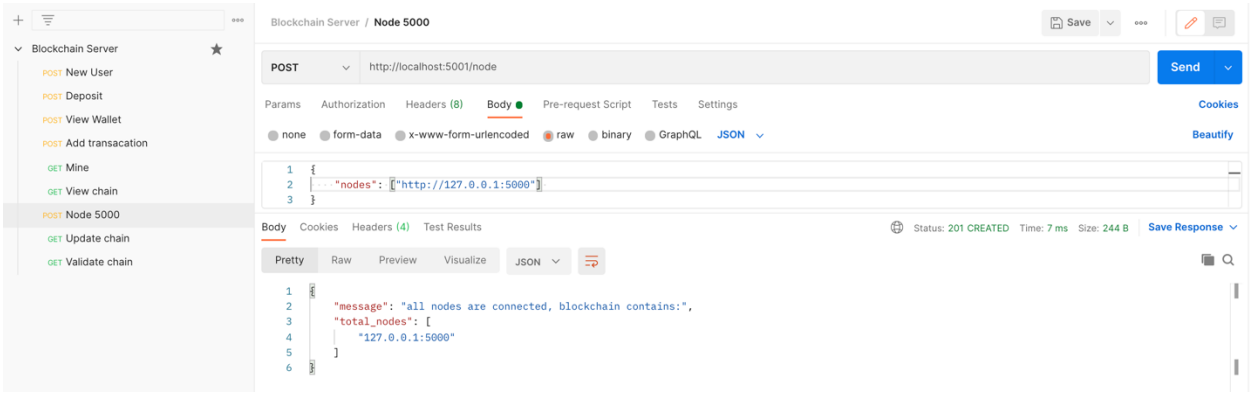


Figure 47. Connecting node 5001 to node 5000

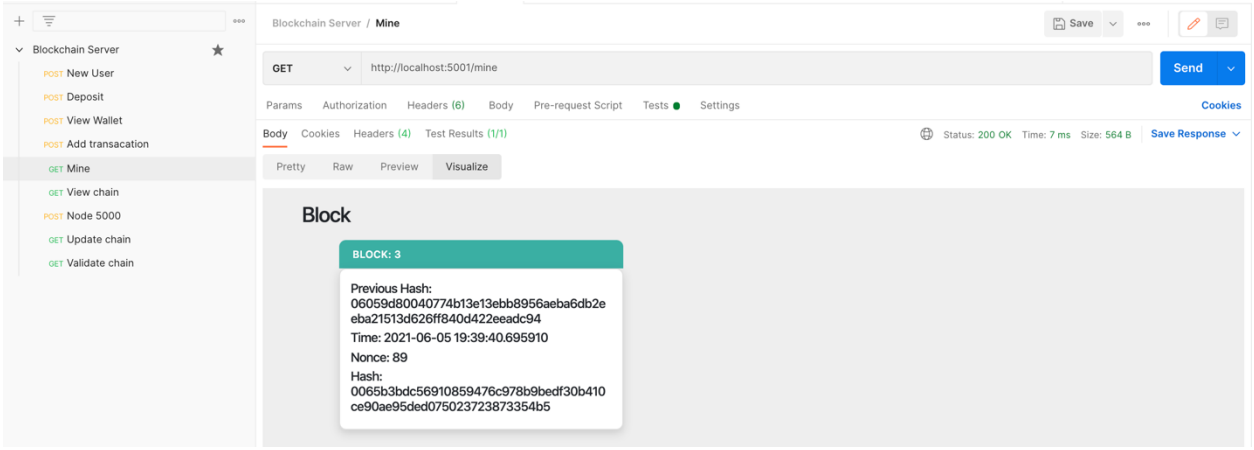


Figure 49. Mining a block with two nodes connected (hash value begins with two zero hex digits)

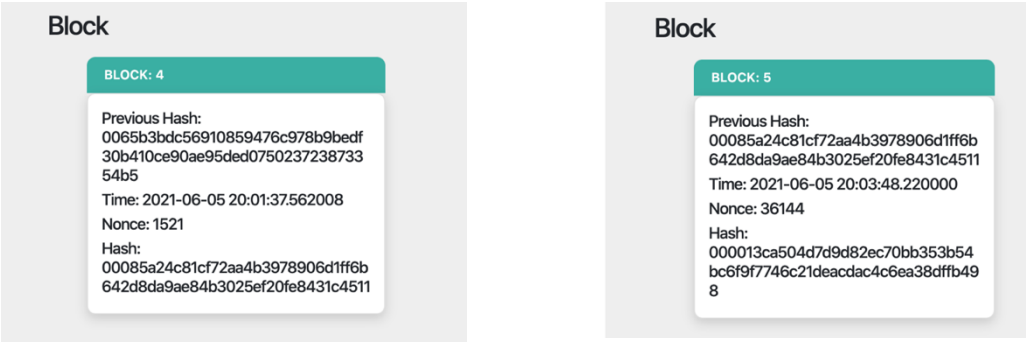


Figure 50. Minning a block with 3 nodes (left) 4 nodes (right) connected to the blockchain

H. Updating the blockchain

Now if you view the chain on the node that is working on port 5001, it will have a longer chain with one extra block mined than in the node on port 5000 as seen in figure 51. The blockchain must be the same on all nodes. So, each node must update the blockchain constantly by looking through the network and obtaining the longest valid chain. So, the user must send the request /update_chain on all nodes to unify the blockchain. It is a “GET” request as seen in figures 52 and 53. If the blockchain on the node has been replaced a message is displayed “chain was replaced” alongside the new blockchain. If the node had the longest blockchain, then it would display “chain is the longest, nothing was replaced” with the unchanged blockchain.

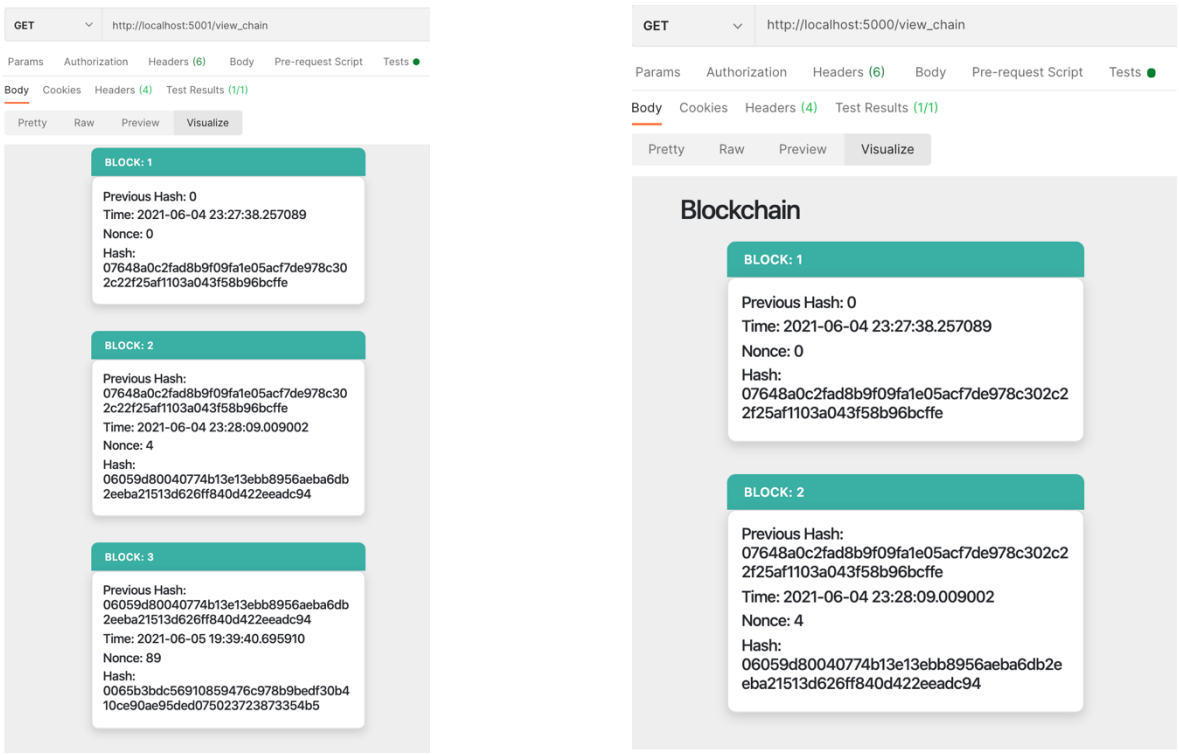


Figure 51. Blockchain on node 5001 (left) and on 5000 (right)

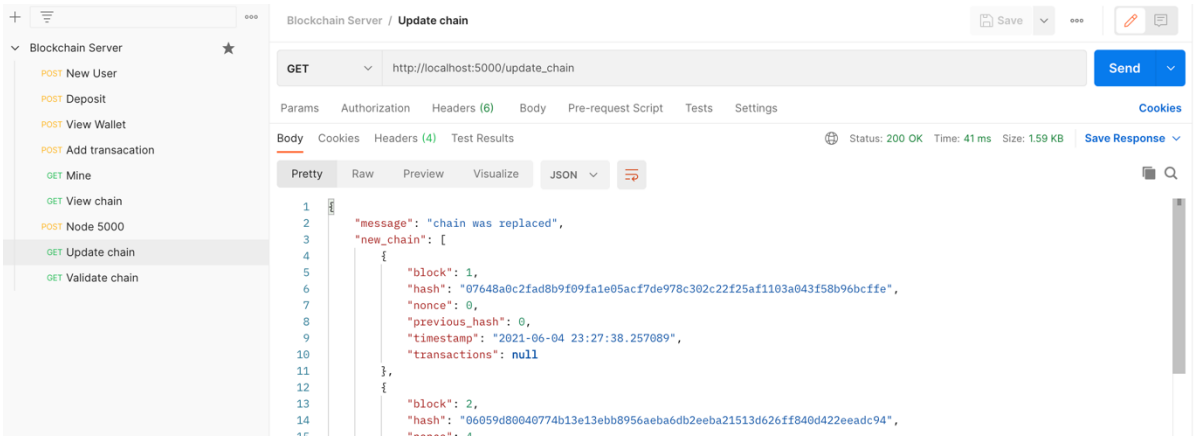


Figure 52. Chain was replaced in node 5000.

Figure 53. Blockchain in node 5001 was not changed.

I. Checking the blockchain

To check whether the blockchain is valid or has been corrupted, the user can validate the chain by using the request /validate_chain and choose “GET”. If the blockchain was not tampered with, it would return “The chain is valid” as seen in figure 54. If the blockchain was edited it would return “The chain is corrupted” as seen in figure 55.

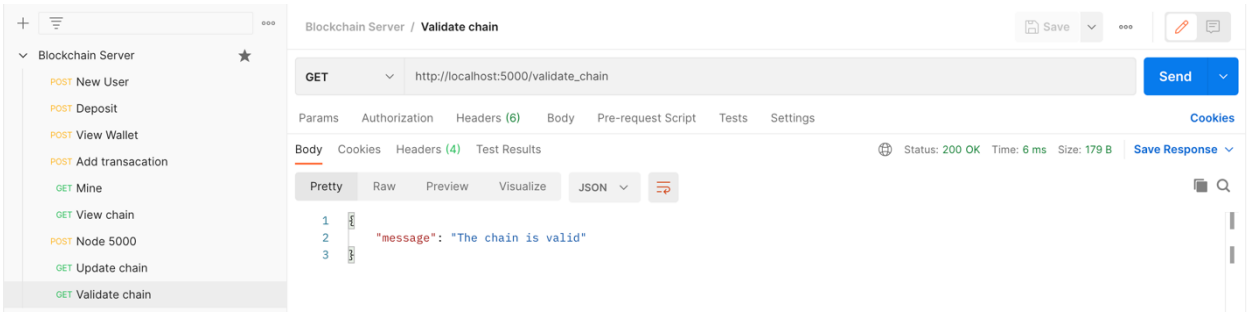


Figure 55. A valid blockchain.

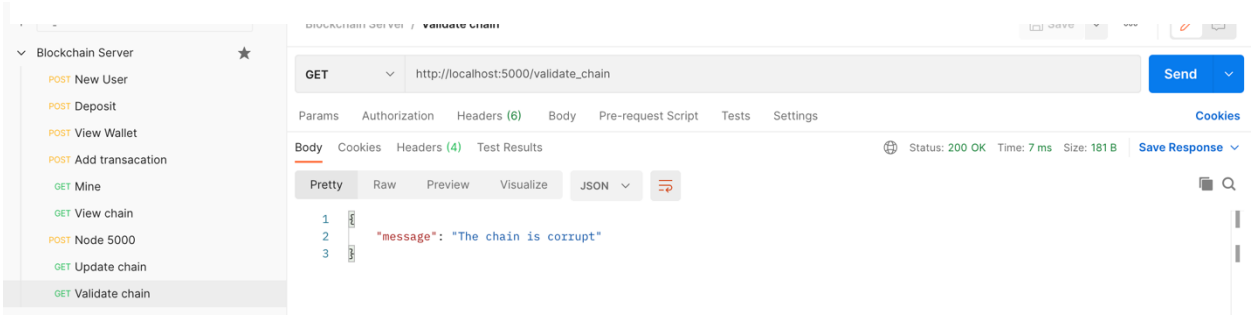


Figure 54. A corrupt blockchain.

CONCLUSION AND PROSPECTIVES

The final blockchain is built in Python and Postman is used to communicate with the blockchain using HTTP requests. The blockchain is an account based blockchain with a specialized wallet generation for smart grid applications. The hashing algorithm used is SHA3 256 since it is better than SHA2 256 that is used in majority of blockchain applications. The Proof of work is based on the Bitcoin blockchain, and the proof of work difficulty increases when more nodes connect to the blockchain. The transactions are digitally signed using special form of elliptic curves called Edward's curve which is better than using RSA since it is faster, more efficient, and consumes less storage.

Challenges in building a blockchain:

- Absence of a universal programming language for blockchain programming
- Smart contracts implementation in blockchain requires studying a new programming language "Solidity"
- Ensuring that the security of the hash algorithm will not be compromised anytime soon.
- The proof of work consumes a lot of computational power

The proposed Improvements on the blockchain is to include smart contracts using Python to automate the transactions. Create new improved hash algorithms specialized for the blockchain to improve its security. Create a new consensus mechanism which won't consume a lot of computational power. Look into a heterogenous blockchain where there are different blocks for different sources of energy. For example, blocks to handle green energy, other blocks to handle transactions for standard energy and a third type of block coming from nuclear energy.

REFERENCES

[1] A.A.G. Agung and R. Handayani, "Blockchain for smart grid", *Journal of King Saud University – Computer and Information Sciences,* pp. 1-10, Jan. 2020.
Available at https://doi.org/10.1016/j.jksuci.2020.01.002

[2] T. Alladi, V. Chamola, J.J.P.C. Rodrigues, and S.A. Kozlov, "Blockchain in Smart Grids: A Review on Different Use Cases," *Sensors*, vol. 19, no. 22, pp. 1-25, Nov. 2019.

[3] M. Andoni, V. Robu, and D. Flynn, "Blockchain technology in the energy sector: A systematic review of challenges and opportunities," Renewable and Sustainable Energy Reviews (Elsevier), vol. 100, pp. 143-174, Feb. 2019.

[4] E. Rykwalder, O. Leech, D. Cawrey, and M. Shen, "The Math Behind the Bitcoin Protocol, an Overview," CoinDesk, Dec. 2020.
Available at https://www.coindesk.com/math-behind-bitcoin.

[5] V.G. Martínez, L. Hernández-Álvarez, and L.H. Encinas, "Analysis of the Cryptographic Tools for Blockchain and Bitcoin," Mathematics, vol. 8, pp. 131-144, Jan. 2020.

[6] M. Raikwar, D. Gligoroski, and K. Kralevska, "SoK of Used Cryptography in Blockchain," IEEE Access, vol. 7, pp. 148550-148575, Feb. 2020.

[7] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System", Oct. 2008.

[8] J. Aumasson, Serious Cryptography: A Practical Introduction to Modern Encryption, L. Chun, 2017.

[9] "hashlib — Secure hashes and message digests", Python, Jun. 2021.
Available at: https://docs.python.org/3/library/hashlib.html

[10] A. Ali, "Comparison and Evaluation of Digital Signature Schemes Employed In NDN Network", International Journal of Embedded systems and Applications(IJESA) Vol.5, No.2, Jun. 2015.

[11] W. Palant, "Python LanguageGenerating RSA signatures using pycrypto", RipTutorial, Sep. 2017. Available at https://riptutorial.com/python/example/19025/generating-rsa-signatures-using-pycrypto

[12] D. Van Flymen, Learn Blockchain by Building One, Oct. 2020.

[13] V. Osetskyi, "What Are Smart Contracts and Their Use Cases in Business", DZone, Jun. 2018. Available at: https://dzone.com/articles/what-is-smart-contracts-blockchain-and-its-use-cas-1

[14] H. Rajora, "API Testing with Postman", Toolsqa, Sep. 2018.

Available at: https://www.toolsqa.com/postman/api-testing-with-postman/

[15] “HTTP response status codes”, MDN Web Doc, May, 2021.
Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[16] “Flask user’s guid”, PalletesProjects, Jun. 2021.