Crypto Lotto

By Ari Zaravelis

Crypto Lotto is a lottery system implemented with blockchain technology. In the past, many lotteries have been tampered with and hacked. Crypto Lotto was developed as a way to give better security and trust back into the game. By using the advantages of blockchain technology, I hope to make this possible. Crypto Lotto was also developed as a way to challenge the current monopoly the U.S government has on lotteries. I created this application as a way to eliminate this monopoly by introducing a decentralized system that will give rewards directly to the pockets of participants and miners. The winners of these lotto games will also be partly anonymous and there will be a chronological and permanent record of all the lotteries that have taken place. There are a few other advantages that Crypto Lotto brings to the table. For example, current state lotteries have to pay staff to maintain the servers for the lotteries. Crypto Lotto gets rid of this cost since it’s operating on a P2P network. Current state lotteries also have to pay staff to maintain the tickets and packaging while also paying staff to create new games. Crypto Lotto, on the other hand, simply needs miners to mine blocks. These blocks act as lottery games itself.

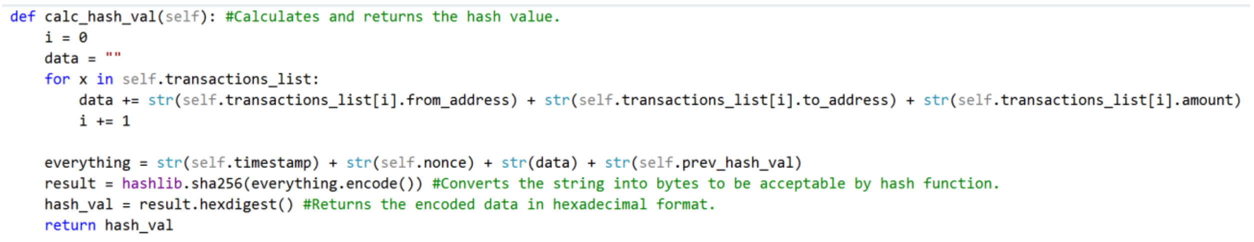
Note that this is a personal project of mine and is just a proof of concept. I have no intention on monetizing it or officially publishing it anywhere. I’ve developed this to help me better understand blockchain technology and gain experience in this field.

Cool Code Snippets with Brief Explanations:

In this project, I chose to use python. I created a total of 6 classes. A class called Block.py which helped create blocks.

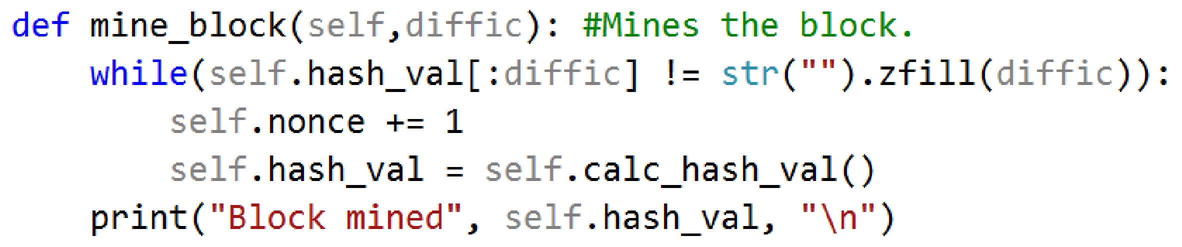
Functions in Block.py:

calc_hash_val is used to calculate the hash value by using the timestamp, nonce, data, and previous hash value.



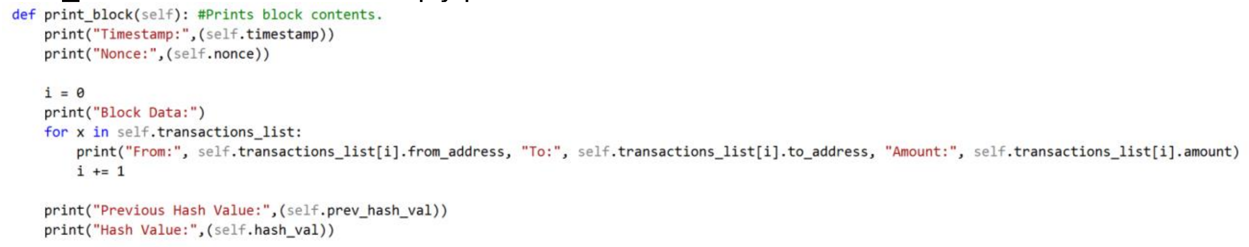
```
def calc_hash_val(self): #Calculates and returns the hash value.
    i = 0
    data = ""
    for x in self.transactions_list:
        data += str(self.transactions_list[i].from_address) + str(self.transactions_list[i].to_address) + str(self.transactions_list[i].amount)
        i += 1

    everything = str(self.timestamp) + str(self.nonce) + str(data) + str(self.prev_hash_val)
    result = hashlib.sha256(everything.encode()) #Converts the string into bytes to be acceptable by hash function.
    hash_val = result.hexdigest() #Returns the encoded data in hexadecimal format.
    return hash_val
```

Mine_block is a method that is used to mine a block. It iterates the nonce until it finds the correct nonce to find the valid hash value. The valid hash value is set by adjusting the difficulty level.



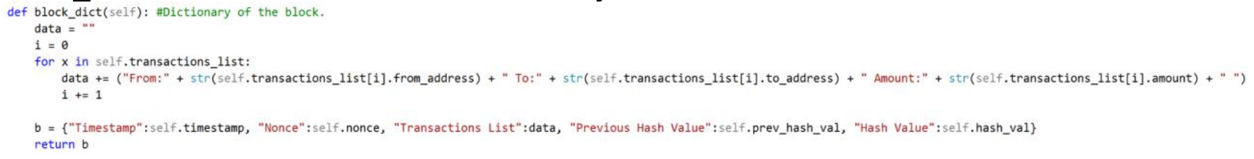
```
def mine_block(self,diffic): #Mines the block.
    while(self.hash_val[:diffic] != str("").zfill(diffic)):
        self.nonce += 1
        self.hash_val = self.calc_hash_val()
    print("Block mined", self.hash_val, "\n")
```

Print_block is a method that simply prints the block information.



```
def print_block(self): #Prints block contents.
    print("Timestamp:",(self.timestamp))
    print("Nonce:",(self.nonce))

    i = 0
    print("Block Data:")
    for x in self.transactions_list:
        print("From:", self.transactions_list[i].from_address, "To:", self.transactions_list[i].to_address, "Amount:", self.transactions_list[i].amount)
        i += 1

    print("Previous Hash Value:",(self.prev_hash_val))
    print("Hash Value:",(self.hash_val))
```

Block_dict is a method that returns a dictionary of the block.

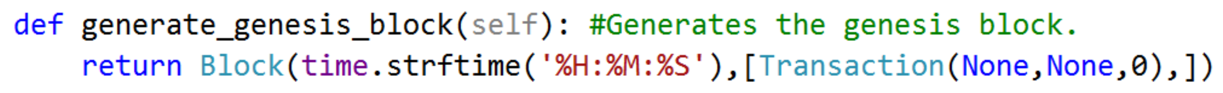


```
def block_dict(self): #Dictionary of the block.
    data = ""
    i = 0
    for x in self.transactions_list:
        data += ("From:" + str(self.transactions_list[i].from_address) + " To:" + str(self.transactions_list[i].to_address) + " Amount:" + str(self.transactions_list[i].amount) + " ")
        i += 1

    b = {"Timestamp":self.timestamp, "Nonce":self.nonce, "Transactions List":data, "Previous Hash Value":self.prev_hash_val, "Hash Value":self.hash_val}
    return b
```

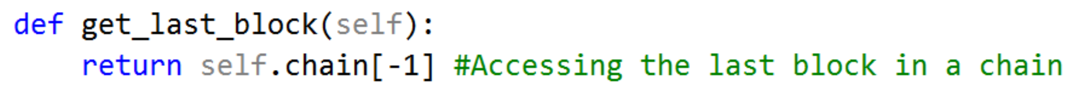
A class called BlockChain.py which helped create the chain of blocks. It had a few functions in it.

Functions in BlockChain.py:

generate_genesis_block is a method that generated the genesis block. It takes no transactions and all nodes start off with this block. Get_last_block is another method which simply returns the last block in a chain.

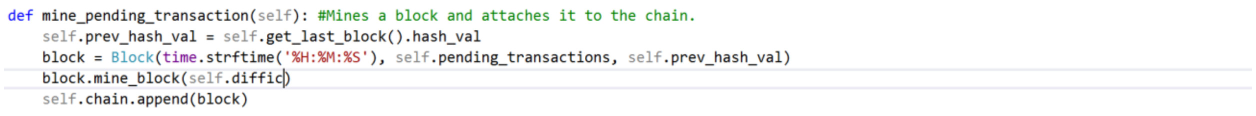


```
def generate_genesis_block(self): #Generates the genesis block.
    return Block(time.strftime('%H:%M:%S'),[Transaction(None,None,0),])
```

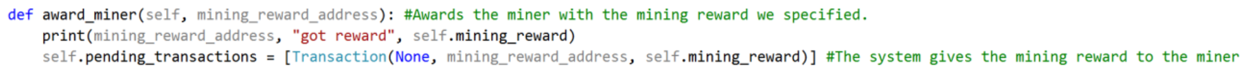


```
def get_last_block(self):
    return self.chain[-1] #Accessing the last block in a chain
```

mine_pending_transaction is a method that calls on the min_block method which mines a block. This mined block is then appended to the chain. award _miner is a method which awards the miner of the new block the mining reward which I specified to be 100.



```
def mine_pending_transaction(self): #Mines a block and attaches it to the chain.
    self.prev_hash_val = self.get_last_block().hash_val
    block = Block(time.strftime('%H:%M:%S'), self.pending_transactions, self.prev_hash_val)
    block.mine_block(self.diffic)
    self.chain.append(block)
```



```
def award_miner(self, mining_reward_address): #Awards the miner with the mining reward we specified.
    print(mining_reward_address, "got reward", self.mining_reward)
    self.pending_transactions = [Transaction(None, mining_reward_address, self.mining_reward)] #The system gives the mining reward to the miner
```

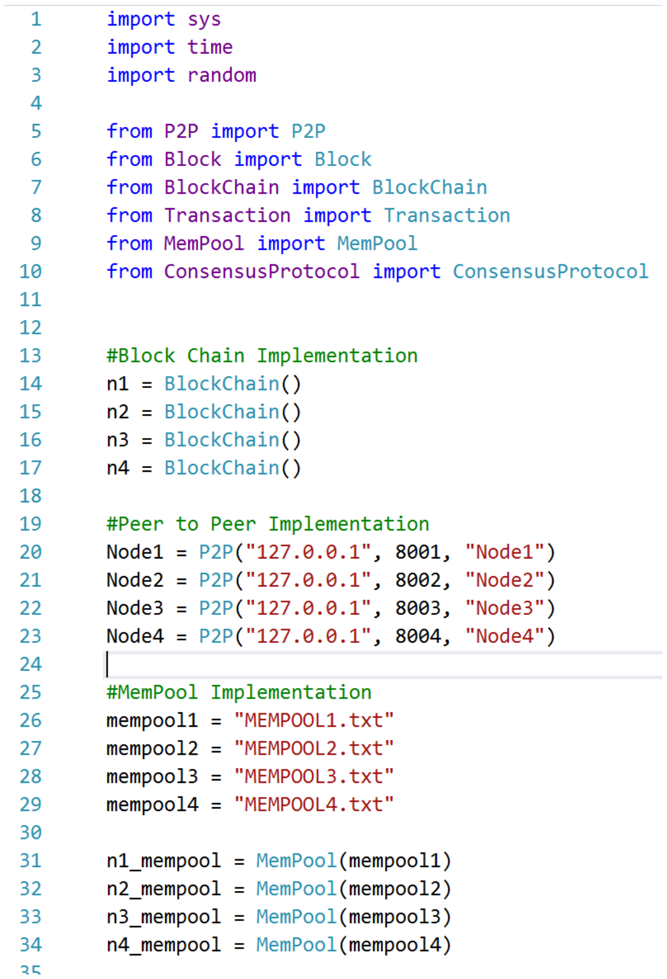
A class called MemPool.py which had functions to remove transactions and add transactions inside the MemPools

A class called ConsensusProtocol.py which had functions to figure out if there is a consensus or not.

A class called P2P.py which helped establish connections between the nodes and send messages between them as well.

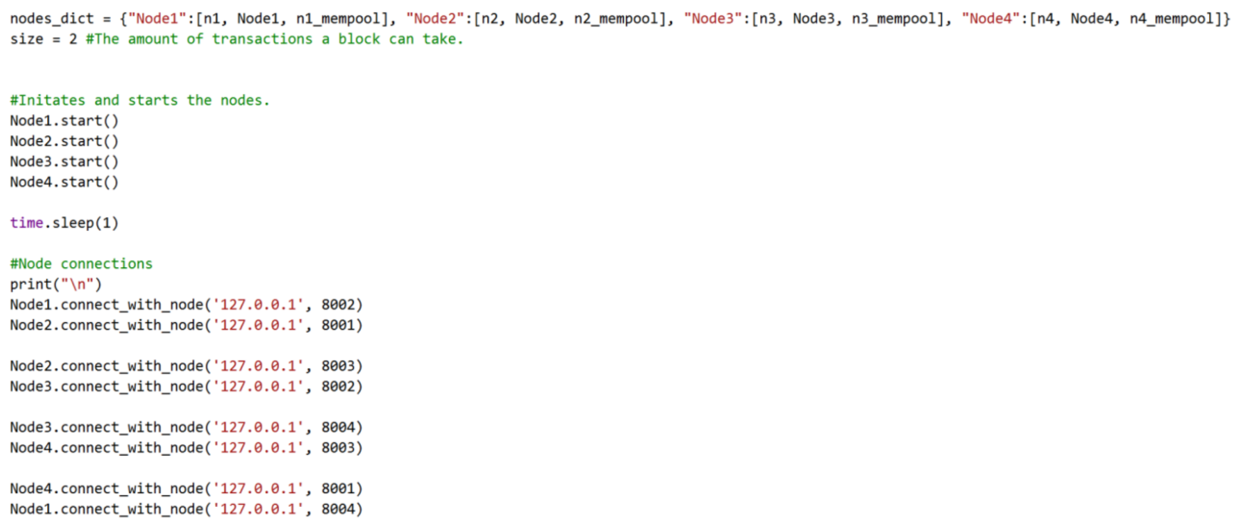
A class called Transaction.py which specifies the contents within a specific transaction such as from_address, to_addres, and amount.

Run code from this main file called lottery.py



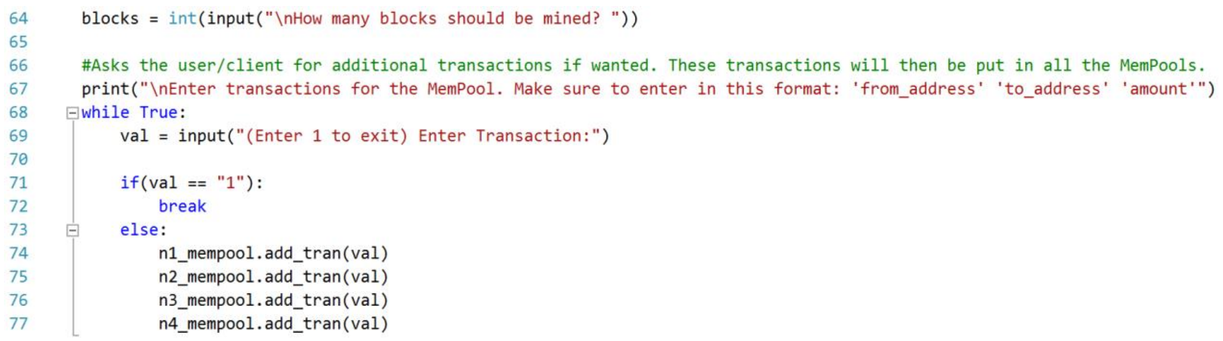
```
import sys
import time
import random

from P2P import P2P
from Block import Block
from BlockChain import BlockChain
from Transaction import Transaction
from MemPool import MemPool
from ConsensusProtocol import ConsensusProtocol


#Block Chain Implementation
n1 = BlockChain()
n2 = BlockChain()
n3 = BlockChain()
n4 = BlockChain()

#Peer to Peer Implementation
Node1 = P2P("127.0.0.1", 8001, "Node1")
Node2 = P2P("127.0.0.1", 8002, "Node2")
Node3 = P2P("127.0.0.1", 8003, "Node3")
Node4 = P2P("127.0.0.1", 8004, "Node4")

#MemPool Implementation
mempool1 = "MEMPOOL1.txt"
mempool2 = "MEMPOOL2.txt"
mempool3 = "MEMPOOL3.txt"
mempool4 = "MEMPOOL4.txt"

n1_mempool = MemPool(mempool1)
n2_mempool = MemPool(mempool2)
n3_mempool = MemPool(mempool3)
n4_mempool = MemPool(mempool4)
```

I imported the 6 classes I mentioned before into the main python file. I created 4 nodes called n1, n2, n3, and n4 by calling the BlockChain() class. I then implemented the P2P network by calling the P2P class. Each node also got its own MemPool which is basically a text file containing all the transactions.



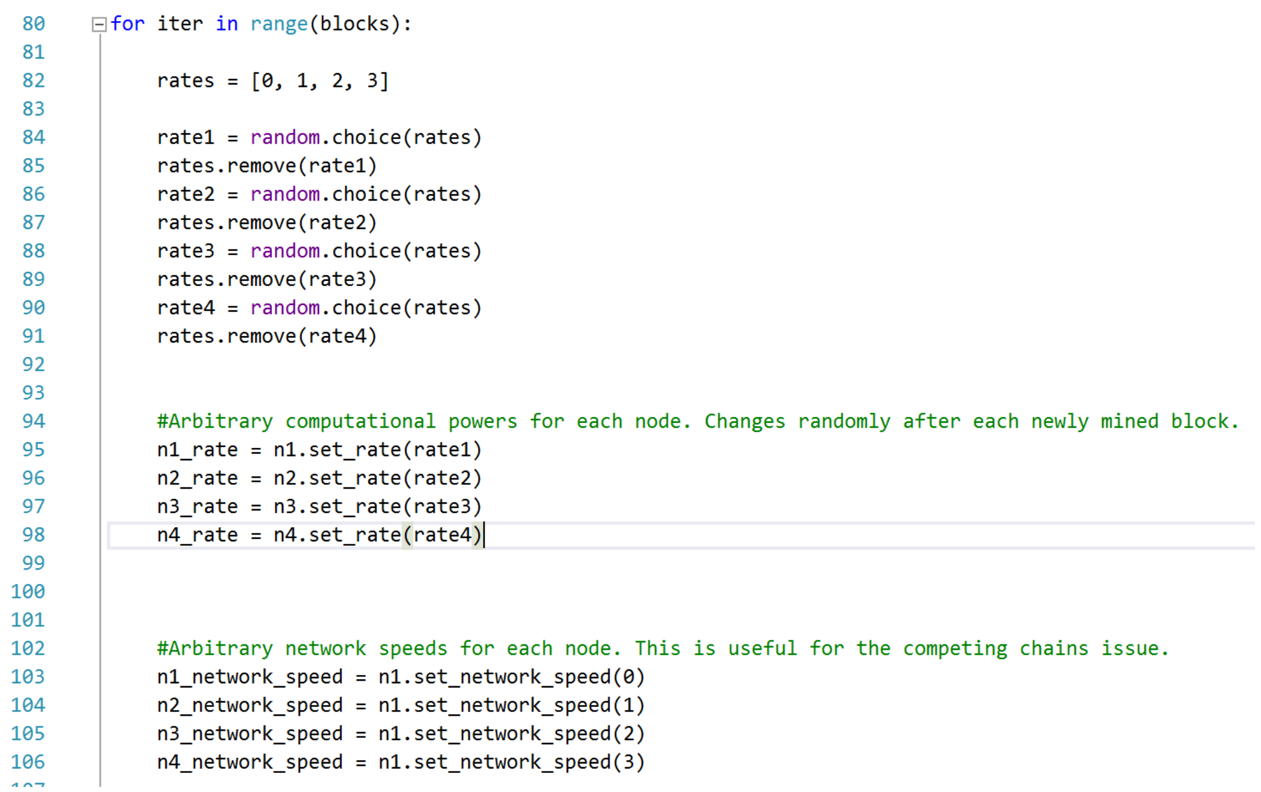
```
nodes_dict = {"Node1":[n1, Node1, n1_mempool], "Node2":[n2, Node2, n2_mempool], "Node3":[n3, Node3, n3_mempool], "Node4":[n4, Node4, n4_mempool]}
size = 2 #The amount of transactions a block can take.


#Initates and starts the nodes.
Node1.start()
Node2.start()
Node3.start()
Node4.start()

time.sleep(1)

#Node connections
print("\n")
Node1.connect_with_node('127.0.0.1', 8002)
Node2.connect_with_node('127.0.0.1', 8001)

Node2.connect_with_node('127.0.0.1', 8003)
Node3.connect_with_node('127.0.0.1', 8002)

Node3.connect_with_node('127.0.0.1', 8004)
Node4.connect_with_node('127.0.0.1', 8003)

Node4.connect_with_node('127.0.0.1', 8001)
Node1.connect_with_node('127.0.0.1', 8004)
```

I specified the size to be 2. This means that each block will only be able to accept 2 transactions. I then started each node and made the connections between all the nodes.



```
blocks = int(input("\nHow many blocks should be mined? "))

#Asks the user/client for additional transactions if wanted. These transactions will then be put in all the MemPools.
print("\nEnter transactions for the MemPool. Make sure to enter in this format: 'from_address' 'to_address' 'amount'")
while True:
    val = input("(Enter 1 to exit) Enter Transaction:")

    if(val == "1"):
        break
    else:
        n1_mempool.add_tran(val)
        n2_mempool.add_tran(val)
        n3_mempool.add_tran(val)
        n4_mempool.add_tran(val)
```

I then asked the user to enter the amount of blocks that should be mined. I also asked the user to enter any new transactions. These transactions are sent to all the MemPools of all the nodes.

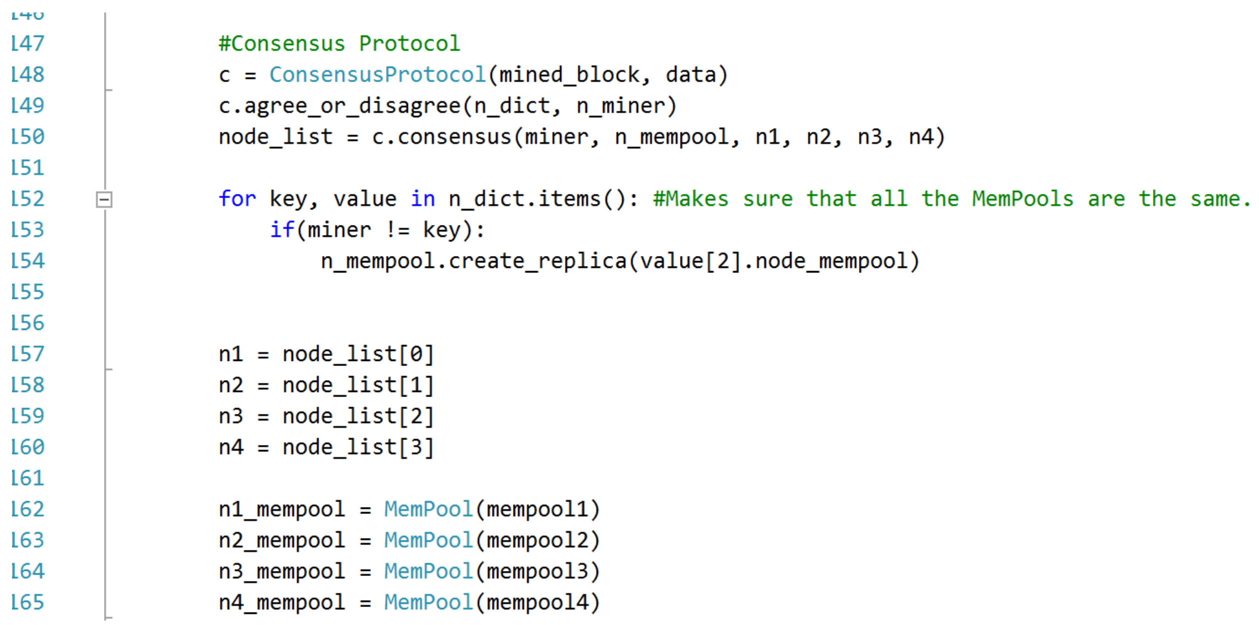


```python
for iter in range(blocks):

    rates = [0, 1, 2, 3]

    rate1 = random.choice(rates)
    rates.remove(rate1)
    rate2 = random.choice(rates)
    rates.remove(rate2)
    rate3 = random.choice(rates)
    rates.remove(rate3)
    rate4 = random.choice(rates)
    rates.remove(rate4)


    #Arbitrary computational powers for each node. Changes randomly after each newly mined block.
    n1_rate = n1.set_rate(rate1)
    n2_rate = n2.set_rate(rate2)
    n3_rate = n3.set_rate(rate3)
    n4_rate = n4.set_rate(rate4)



    #Arbitrary network speeds for each node. This is useful for the competing chains issue.
    n1_network_speed = n1.set_network_speed(0)
    n2_network_speed = n1.set_network_speed(1)
    n3_network_speed = n1.set_network_speed(2)
    n4_network_speed = n1.set_network_speed(3)
```

The rates for each node simulates the computational power for each node. Some nodes are faster than others, therefore, the fastest node always mines the new block first. Network speeds simulates the network speeds of all the nodes. The nodes with the fastest network speeds will be able to send messages across the network faster than nodes with slow network speeds.

```python
#Starts the mining process.
if(len(fastest_rates) == 1):
    print("\n")
    print("Nodes Starting Mining")
    for key, value in n_dict.items():
        if(key == fastest_rates[0]):
            miner = key
            n_miner = value[0]
            n_miner.pending_transactions = []
            n_mempool = value[2]


            for i in range(size):
                n_miner.create_transaction(Transaction(str(from_address[i]), str(to_address[i]), float(amount[i])))
                data = n_miner.pending_transactions

            print(miner, "Mined Block")
            n_miner.mine_pending_transaction()
            mined_block = n_miner.chain[-1]
            value[1].send_to_nodes(mined_block.block_dict())

            time.sleep(1)
```

This shows the mining process which implements the appropriate functions and such.



```
        #Consensus Protocol
        c = ConsensusProtocol(mined_block, data)
        c.agree_or_disagree(n_dict, n_miner)
        node_list = c.consensus(miner, n_mempool, n1, n2, n3, n4)

        for key, value in n_dict.items(): #Makes sure that all the MemPools are the same.
            if(miner != key):
                n_mempool.create_replica(value[2].node_mempool)


        n1 = node_list[0]
        n2 = node_list[1]
        n3 = node_list[2]
        n4 = node_list[3]

        n1_mempool = MemPool(mempool1)
        n2_mempool = MemPool(mempool2)
        n3_mempool = MemPool(mempool3)
        n4_mempool = MemPool(mempool4)
```

This shows the consensus protocol which demonstrates whether the nodes agree or disagree with the newly mined block.

I also addressed the competing chains issue which occurs if two nodes mine a block at the same time. This would happen in the application when two nodes have the same computational power which is demonstrated as rates in my project.

Lastly, I displayed all the chains for all the nodes by calling the appropriate classes and functions.