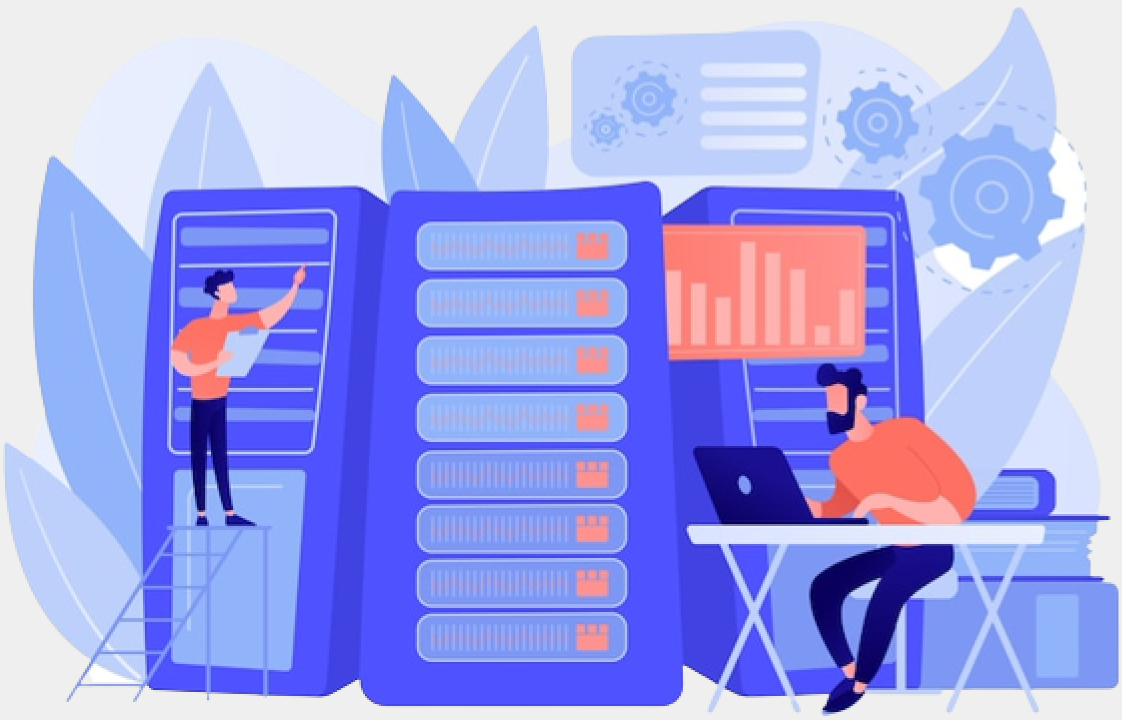
Blockchain

Assignment 3



Ali Hamza 20i1881
Abdullah Malik 20i0930

29 October, 2023

Tasks

The code starts by defining a Transaction class. This class represents a transaction in the blockchain. Each transaction has a sender, a receiver, an amount, and a digital signature. The __str__ method is used to provide a string representation of the transaction, which displays the sender, receiver, and the amount.

Transaction

```
class Transaction:
    def __init__(self, sender, receiver, amount, signature=None):
        self.sender = sender
        self.receiver = receiver
        self.amount = amount
        self.signature = signature

    def __str__(self):
        return f"{self.sender} -> {self.receiver}: {self.amount}"
```

73] ✓ 0.0s

The generate_keys_for_nodes function is responsible for generating public and private key pairs for a specified number of nodes. Each public key is written to a file named "public_keys.txt" in PEM format, prefixed with its associated block number.

Key Pairs

```
def generate_keys_for_nodes(num_nodes):
    with open("public_keys.txt", "w") as f:
        for i in range(num_nodes):
            private_key, public_key = generate_keypair()

            # Convert public key to PEM format
            pem = public_key.public_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PublicFormat.SubjectPublicKeyInfo
            )

            # Write the public key to the file with a block number
            f.write(f"Block {i}:\n")
            f.write(pem.decode() + "\n\n")
```

74] ✓ 0.0s

The Block class represents a block in the blockchain. Each block has an index, timestamp, a list of transactions, a previous hash, a nonce, and its own hash. The hash_block method computes the hash of the block using its attributes. The make_genesis_block function creates the very first block in the blockchain, known as the genesis block. This block has a predefined index of 0, the current timestamp, and a single transaction labeled "Genesis Block".

1. Genesis Block:

```python
from datetime import datetime
import hashlib

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

    def hash_block(self):
        # Combine all transaction data into one string
        combined_data = ''.join(str(tx) for tx in self.transactions)
        return hashlib.sha256(f"{self.index}{self.timestamp}{combined_data}{self.previous_hash}{self.nonce}".encode()).hexdigest()



def make_genesis_block():
    """Make the first block in a block-chain."""
    return Block(index=0, timestamp=datetime.now(), transactions=["Genesis Block"], previous_hash="0")
```

The generate_keypair function uses the cryptography library to generate a private and public key pair for elliptic curve cryptography. The function returns both the private and public keys.

2. Key Pair Generation:

Using the cryptography library:

```python
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import ec

def generate_keypair():
    private_key = ec.generate_private_key(ec.SECP256R1(), default_backend())
    public_key = private_key.public_key()

    return private_key, public_key
```

276] ✓ 0.0s

The sign_transaction function is used to digitally sign a transaction. It takes in a private key and data (the transaction details) and returns a digital signature using the ECDSA algorithm with SHA256 as the hash function.

3. Digital Signature:

```
from cryptography.hazmat.primitives import hashes

def sign_transaction(private_key, data):
    signature = private_key.sign(data.encode(), ec.ECDSA(hashes.SHA256()))
    return signature
```

[277] ✓ 0.0s

The next_block function is responsible for creating a new block in the blockchain. It signs each transaction in the block using a generated private key. The block is then mined by adjusting its nonce until its hash meets a certain difficulty target (in this case, starting with "00000A0"). If the block isn't mined within 10 minutes, the timestamp is refreshed, and the mining process continues.

4. Mining and New Blocks:

```
import time

def next_block(pre_block, transactions=[]):
    index = pre_block.index + 1
    timestamp = datetime.now()

    # Sign each transaction
    private_key, _ = generate_keypair()
    for transaction in transactions:
        signature = sign_transaction(private_key, str(transaction))
        transaction.signature = signature

    block = Block(index, timestamp, transactions, pre_block.hash)


    # Print block information
    print(f"Creating Block #{index}")
    print(f"Timestamp: {timestamp}")
    # print(f"Data: {data}")
    print(f"Previous Hash: {pre_block.hash}")

    # Mining
    target_difficulty = "00000A0"
    start_time = time.time()
    increment = 1  # Initialize increment value
```

```python
    # Mining
    target_difficulty = "00000A0"
    start_time = time.time()
    increment = 1  # Initialize increment value

    while not block.hash.startswith(target_difficulty):
        block.nonce += increment  # Adjust the nonce by the increment value
        block.hash = block.hash_block()
        if time.time() - start_time > 600:  # 10 minutes
            print("Refreshing timestamp and retrying...")
            block.timestamp = datetime.now()  # Refresh the timestamp
            start_time = time.time()  # Reset the start time
            increment += 1  # Increase the increment value

    # Print mining information
    print(f"Block #{index} mined successfully.")
    print(f"Hash: {block.hash}")
    print(f"Nonce: {block.nonce}")
    print(f"Time taken: {time.time() - start_time} seconds")

    return block
```

8] ✓ 0.0s

The create_blockchain function initializes the blockchain with the genesis block. It then adds new blocks to the blockchain. Each new block contains 20 demo transactions. The sender, receiver, and amount for each transaction are generated based on loop counters.

5. Chain of Blocks:

```python
def create_blockchain():
    blockchain = [make_genesis_block()]

    for i in range(1, 20):
        # Generate a list of demo transactions for each block
        transactions = []
        for j in range(20):  # 20 transactions
            sender = f"Alice_{j}"
            receiver = f"Bob_{j}"
            amount = (i * 100) + j  # example amount
            transaction = Transaction(sender, receiver, amount)
            transactions.append(transaction)

        blockchain.append(next_block(blockchain[-1], transactions))

    return blockchain
```

[279] ✓ 0.0s

+ Code + Markdown

In the main execution section, public and private key pairs are generated for 20 nodes. The blockchain is then created with the specified number of blocks. Finally, the transactions for each block in the blockchain are displayed, showing details like the sender, receiver, amount, and the digital signature.

Main

```python
# Main
if __name__ == "__main__":
    num_nodes = 20  # or any number you want
    generate_keys_for_nodes(num_nodes)

    blockchain = create_blockchain()

    # Displaying transactions for each block
    for block in blockchain:
        print(f"Block #{block.index} Transactions:")
        for tx in block.transactions:
            if isinstance(tx, Transaction):  # Check if it's a Transaction object
                print(f"Sender: {tx.sender}")
                print(f"Receiver: {tx.receiver}")
                print(f"Amount: {tx.amount}")
                print(f"Signature: {tx.signature}")
                print("------")
        print("\n")
```

[280] 2m 57.4s

Generated Transactions:

```
Creating Block #1
Timestamp: 2023-10-29 22:58:19.933384
Previous Hash: fbf2b19422e049b6b382eb74c4c5c83f0212654f4bfe49471f18d0604e07d964
Block #1 mined successfully.
Hash: 0000004f6446edc63e2cf39d0d913ca83cab7769ecd9d22e331f39ca8fc060c0
Nonce: 1560699
Time taken: 29.777577877044678 seconds
Creating Block #2
Timestamp: 2023-10-29 22:58:49.712980
Previous Hash: 0000004f6446edc63e2cf39d0d913ca83cab7769ecd9d22e331f39ca8fc060c0
Block #2 mined successfully.
Hash: 0000004f6c13b988cf4eb3003d73360934bed44597acadb2f4d3c7cccf753316
Nonce: 5431604
Time taken: 111.94909691810608 seconds
Creating Block #3
Timestamp: 2023-10-29 23:00:41.664970
Previous Hash: 0000004f6c13b988cf4eb3003d73360934bed44597acadb2f4d3c7cccf753316
Block #3 mined successfully.
Hash: 000000b4b7f7ce057705a33c350cd10f6ad8c5f7757e6a3548a53a398fe8c78a
Nonce: 11714683
Time taken: 243.89662075042725 seconds
```

Public Keys: