Omkar Gholap **D20A** **09**

Blockchain Experiment 3

AIM: Create a Cryptocurrency using Python and perform mining in the Blockchain created.

THEORY:

Q1: Challenges in P2P networks.

Peer-to-Peer (P2P) networks are decentralized systems where nodes communicate directly without a central authority. Although they provide transparency and fault tolerance, several challenges exist:

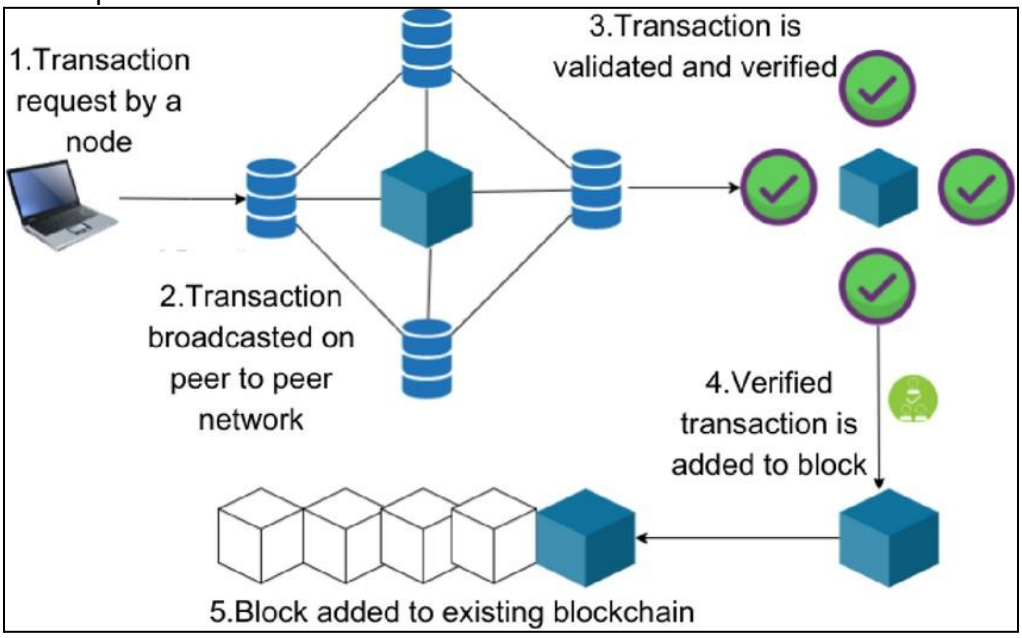
- **Security Issues**
  Nodes may behave maliciously and attempt attacks such as double-spending or fake transactions.
  Sybil attacks occur when one entity creates multiple fake identities to control the network.
- **Network Latency**
  Since nodes are distributed globally, data propagation can be slow.
  Delays in block propagation may lead to temporary forks in the blockchain.
- **Consensus Management**
  All nodes must agree on a single version of the blockchain.
  Maintaining consensus requires complex algorithms like Proof-of-Work or Proof-of-Stake.
- **Scalability**
  As more nodes join, communication overhead increases.
  High transaction volume can slow down processing.
- **Resource Consumption**
  Mining consumes high computational power and electricity.
  Storage requirements grow as the blockchain size increases.

Q2: How transactions are performed on the network?

Blockchain transactions follow a structured process:

- **Step 1: Transaction Creation**
  A user creates a transaction containing sender address, receiver address, and amount.
  The transaction is digitally signed using the sender's private key.
- **Step 2: Transaction Broadcast**
  The transaction is broadcasted to all nodes in the P2P network.
- **Step 3: Verification**
  Nodes verify Digital signatures, Account balance, Transaction validity
- **Step 4: Mempool Storage**
  Valid transactions are stored temporarily in the mempool awaiting confirmation.
- **Step 5: Mining and Block Creation**
  Miners select transactions from the mempool.
  Transactions are grouped into a block and mined using Proof-of-Work.
- **Step 6: Block Addition**

The new block is added to the blockchain after consensus. Transactions become permanent and immutable.



Q3: Explain the role of mempools

A mempool (memory pool) is a temporary storage area for unconfirmed transactions. Functions of Mempools:

- Stores pending transactions before they are added to a block.
- Helps miners select transactions for block creation.
- Prevents network congestion by organizing transaction queues.

Q4: Write briefly about the libraries and the tools used during implementation.

Libraries used:

- **datetime**
  Used to generate timestamps for each block created in the blockchain.
- **hashlib**
  Used to perform SHA-256 hashing for block identification and proof-of-work mining.
- **json**
  Used for encoding and decoding blockchain data into JSON format for API communication.
- **uuid4 (from uuid module)**
  Used to generate a unique identifier for each node in the blockchain network.
- **urlparse (from urllib.parse)**
  Used to extract and manage network node addresses.
- **Flask**
  Used to create RESTful API endpoints such as mine_block, get_chain, add_transaction, connect_node, and replace_chain for blockchain operations.
- **request (from Flask)** request is used to receive input data from API calls.
- **jsonify (from Flask)** jsonify is used to return blockchain responses in JSON format.

- **requests**
  Used for communication between different blockchain nodes to implement consensus and longest chain replacement.

Tools used:

- **VS Code**
  Used as the development environment to write and execute Python code.
- **Thunder Client (VS Code Extension)**
  Used to test REST API endpoints such as mining blocks, adding transactions, connecting nodes, and applying consensus.

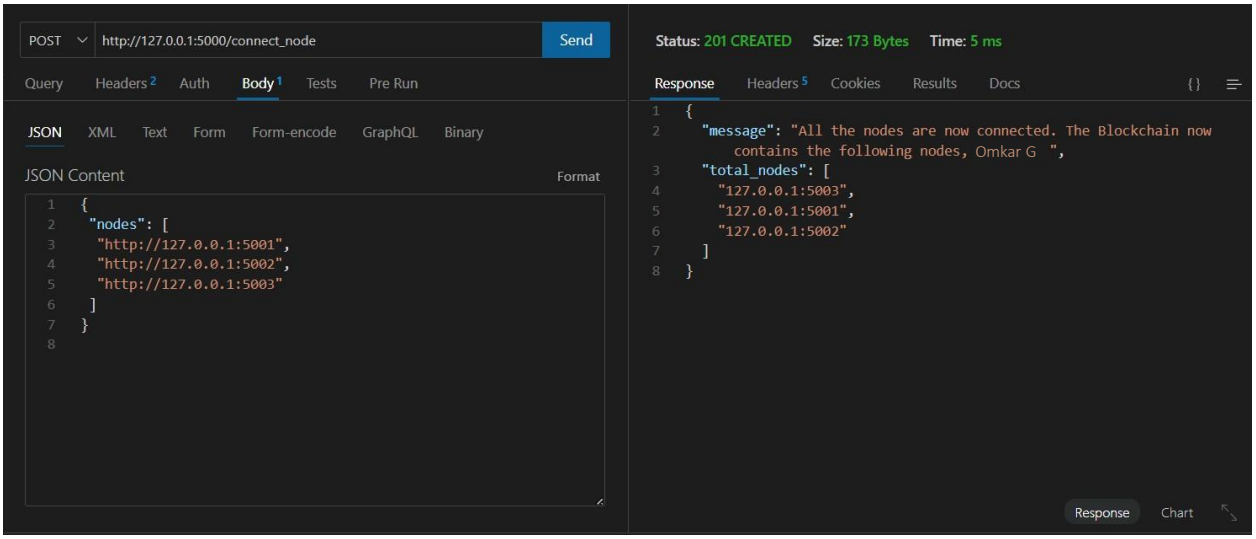
TASKS PERFORMED:

```
# Required:
# pip install Flask==0.12.2
# pip install requests==2.18.4

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


# ---------- Blockchain Class ----------

class Blockchain:

  def __init__(self):
    self.chain = []
    self.transactions = []     # Pending transactions
    self.nodes = set()         # Connected nodes
    self.create_block(proof=1, previous_hash='0')  # Genesis block

  def create_block(self, proof, previous_hash):
    block = {
      'index': len(self.chain) + 1,
      'timestamp': str(datetime.datetime.now()),
      'proof': proof,
      'previous_hash': previous_hash,
      'transactions': self.transactions
    }
    self.transactions = []     # Reset transactions after adding to block
    self.chain.append(block)
    return block

  def get_previous_block(self):
    return self.chain[-1]

  # Proof of Work algorithm
  def proof_of_work(self, previous_proof):
    new_proof = 1
    while True:
      hash_operation = hashlib.sha256(
        str(new_proof**2 - previous_proof**2).encode()
      ).hexdigest()
      if hash_operation[:4] == '0000':
        return new_proof
      new_proof += 1

  # Hash a block
  def hash(self, block):
    encoded_block = json.dumps(block, sort_keys=True).encode()
    return hashlib.sha256(encoded_block).hexdigest()

  # Validate blockchain integrity
  def is_chain_valid(self, chain):
    previous_block = chain[0]
    block_index = 1

    while block_index < len(chain):
      block = chain[block_index]

      # Check previous hash link
      if block['previous_hash'] != self.hash(previous_block):
        return False

      # Check proof of work
      previous_proof = previous_block['proof']
      proof = block['proof']
      hash_operation = hashlib.sha256(
        str(proof**2 - previous_proof**2).encode()
      ).hexdigest()

      if hash_operation[:4] != '0000':
        return False

      previous_block = block
      block_index += 1

    return True

  # Add new transaction
  def add_transaction(self, sender, receiver, amount):
    self.transactions.append({
      'sender': sender,
      'receiver': receiver,
      'amount': amount
    })
    previous_block = self.get_previous_block()
    return previous_block['index'] + 1

  # Register a new node
  def add_node(self, address):
    parsed_url = urlparse(address)
    self.nodes.add(parsed_url.netloc)

  # Consensus: replace chain with longest valid chain
  def replace_chain(self):
    network = self.nodes
    longest_chain = None
    max_length = len(self.chain)

    for node in network:
      response = requests.get(f'http://{node}/get_chain')

      if response.status_code == 200:
        length = response.json()['length']
        chain = response.json()['chain']

        if length > max_length and self.is_chain_valid(chain):
          max_length = length
          longest_chain = chain

    if longest_chain:
      self.chain = longest_chain
      return True

    return False


# ---------- Flask Web App ----------
```

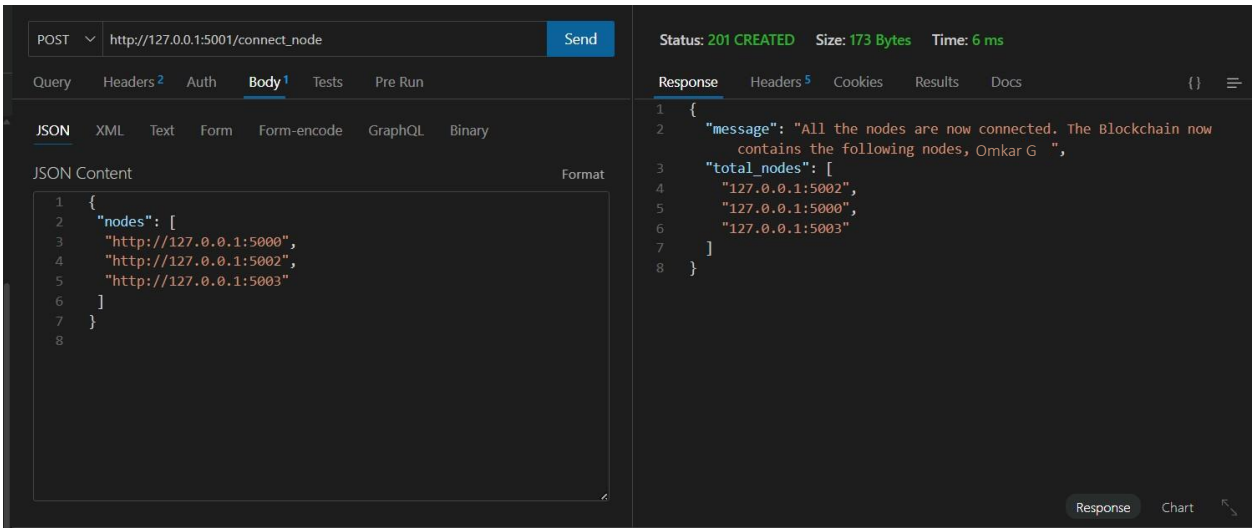
```
app = Flask(__name__)

# Unique address for this node
node_address = str(uuid4()).replace('-', '')

# Create blockchain instance
blockchain = Blockchain()


# Mine a new block
@app.route('/mine_block', methods=['GET'])
def mine_block():
    previous_block = blockchain.get_previous_block()
    previous_proof = previous_block['proof']
    proof = blockchain.proof_of_work(previous_proof)
    previous_hash = blockchain.hash(previous_block)

    # Mining reward
    blockchain.add_transaction(
        sender=node_address,
        receiver='Anu',
        amount=1
    )

    block = blockchain.create_block(proof, previous_hash)

    response = {
        'message': 'Block mined successfully.',
        'index': block['index'],
        'timestamp': block['timestamp'],
        'proof': block['proof'],
        'previous_hash': block['previous_hash'],
        'transactions': block['transactions']
    }

    return jsonify(response), 200


# Get full blockchain
@app.route('/get_chain', methods=['GET'])
def get_chain():
    response = {
        'chain': blockchain.chain,
        'length': len(blockchain.chain)
    }
    return jsonify(response), 200


# Check if blockchain is valid
@app.route('/is_valid', methods=['GET'])
def is_valid():
    if blockchain.is_chain_valid(blockchain.chain):
        response = {'message': 'The Blockchain is valid.'}
    else:
        response = {'message': 'The Blockchain is not valid.'}

    return jsonify(response), 200


# Add a new transaction
@app.route('/add_transaction', methods=['POST'])
def add_transaction():
    json_data = request.get_json()
    transaction_keys = ['sender', 'receiver', 'amount']

    if not all(key in json_data for key in transaction_keys):
        return 'Missing transaction data', 400

    index = blockchain.add_transaction(
        json_data['sender'],
        json_data['receiver'],
        json_data['amount']
    )

    response = {
        'message': f'Transaction will be added to Block {index}'
    }

    return jsonify(response), 201


# Connect new nodes
@app.route('/connect_node', methods=['POST'])
def connect_node():
    json_data = request.get_json()
    nodes = json_data.get('nodes')

    if nodes is None:
        return "No node", 400

    for node in nodes:
        blockchain.add_node(node)

    response = {
        'message': 'Nodes connected successfully.',
        'total_nodes': list(blockchain.nodes)
    }

    return jsonify(response), 201


# Replace chain if longer valid chain exists
@app.route('/replace_chain', methods=['GET'])
def replace_chain():
    if blockchain.replace_chain():
        response = {
            'message': 'Chain replaced by the longest one.',
            'new_chain': blockchain.chain
        }
    else:
        response = {
            'message': 'Current chain is the longest.',
            'actual_chain': blockchain.chain
        }

    return jsonify(response), 200


# Run the app
app.run(host='0.0.0.0', port=5000)
```

Step 1: Connect Nodes

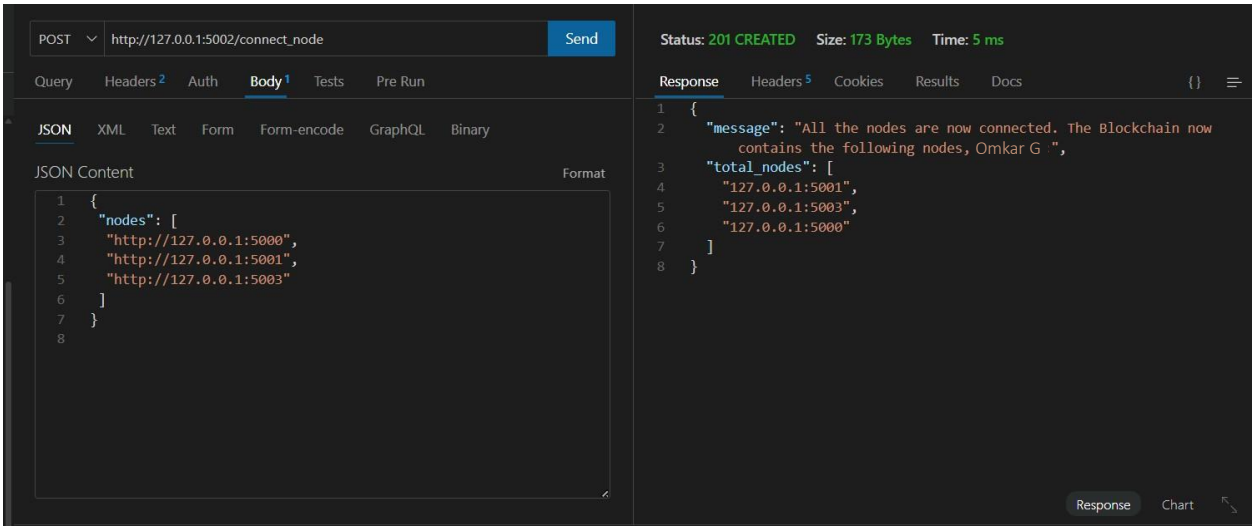
From 5000



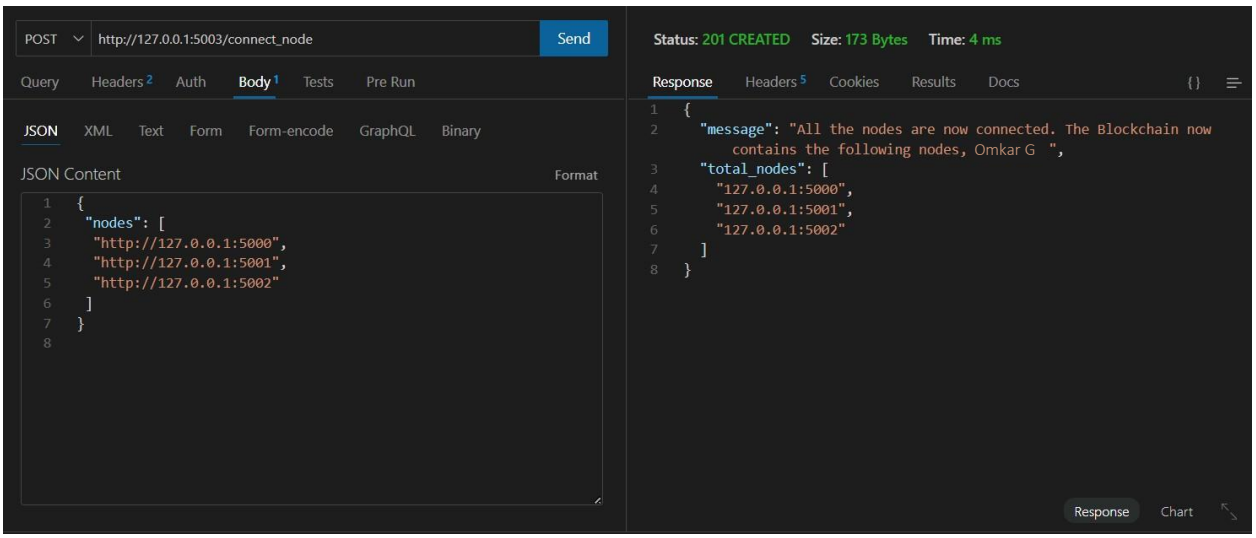
From 5001



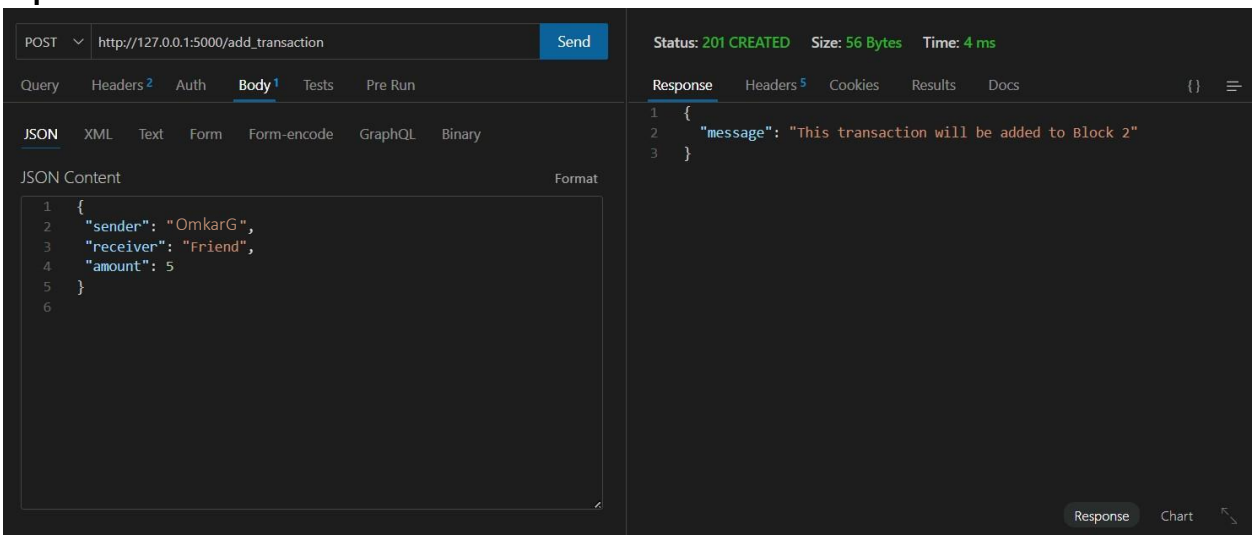
From 5002



From 5003

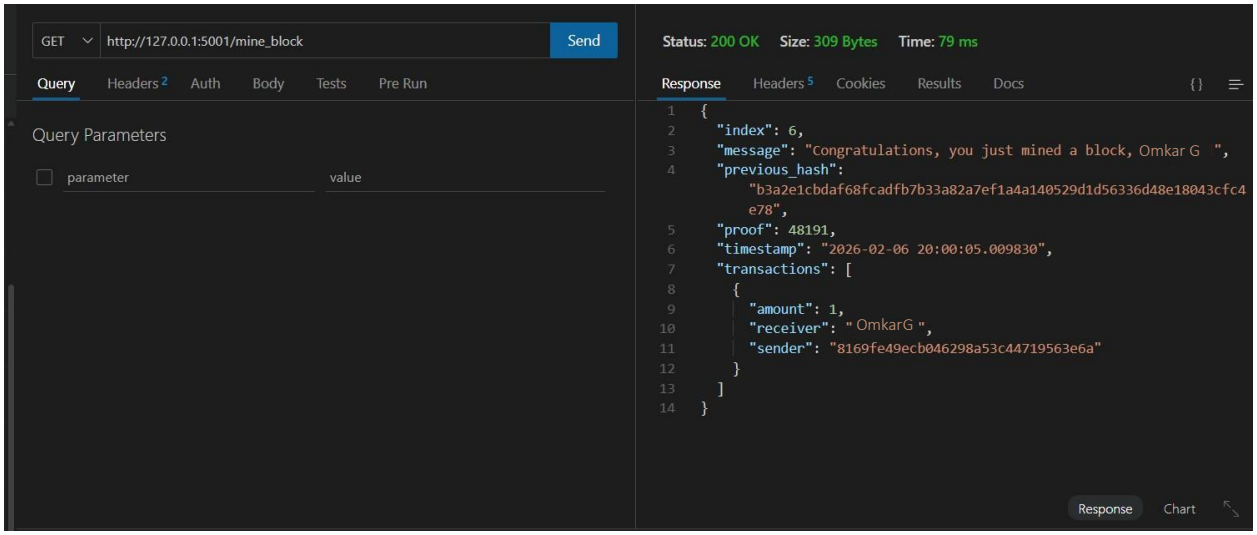


Step 2: Add transaction

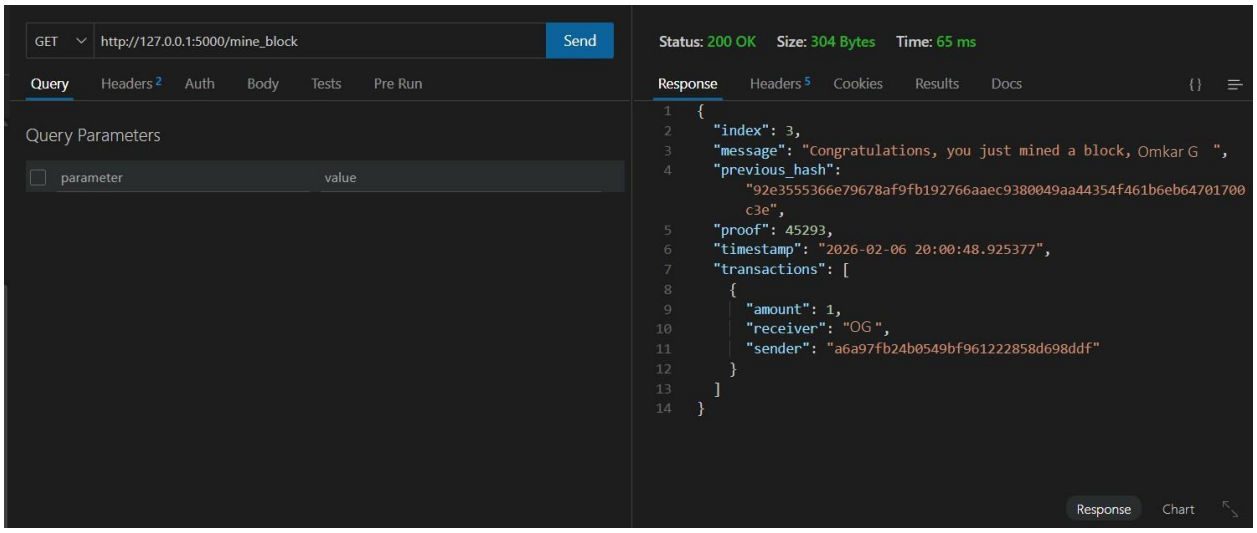


Step 3: Mine different blocks

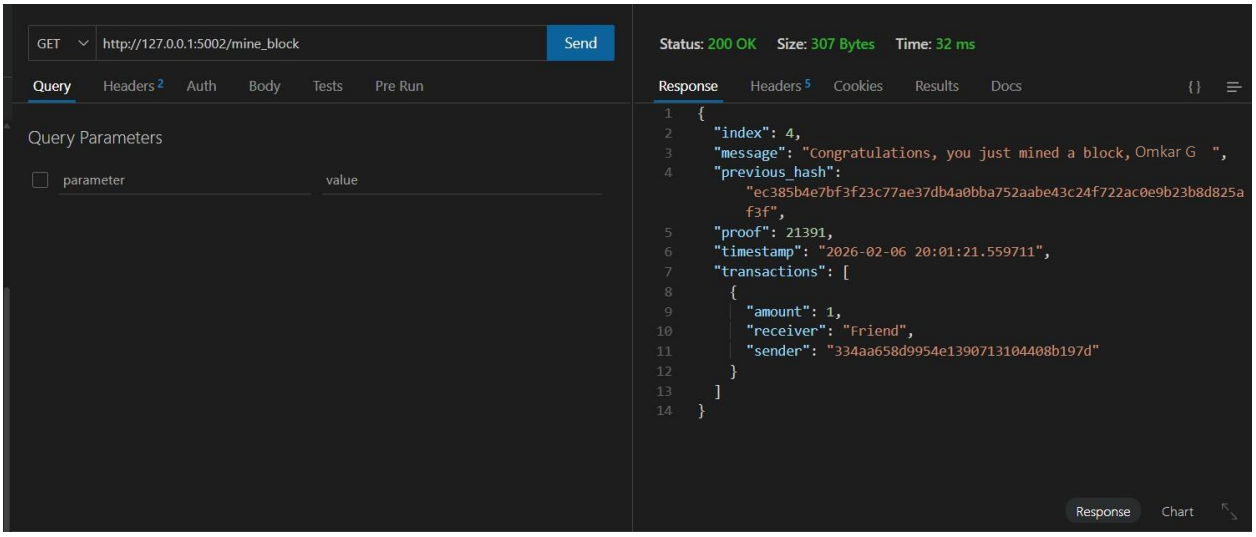
Node 5001 mined 5 times



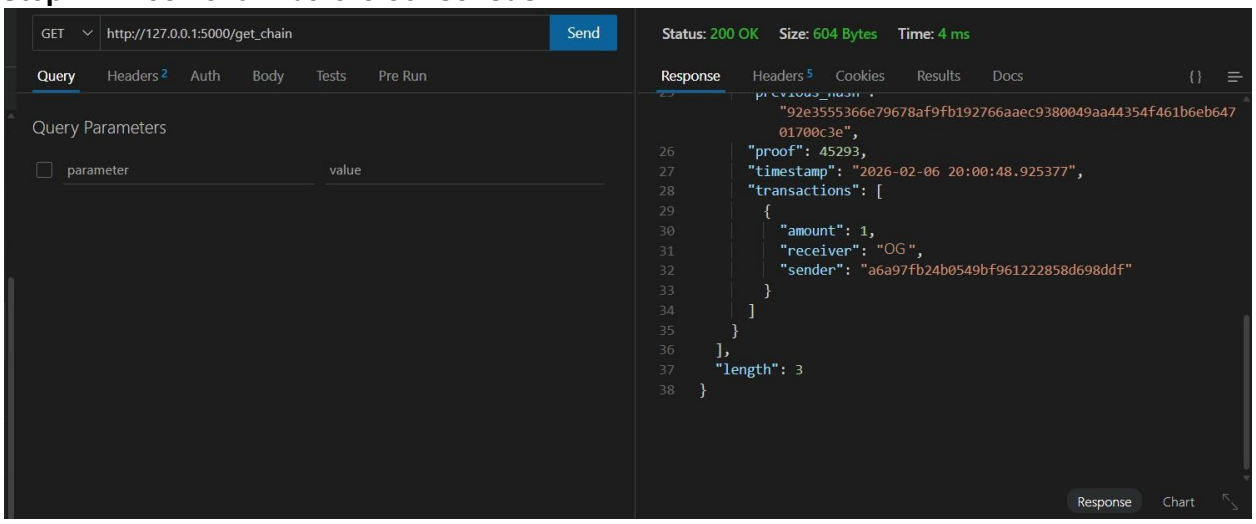
Node 5000 mined 2 times

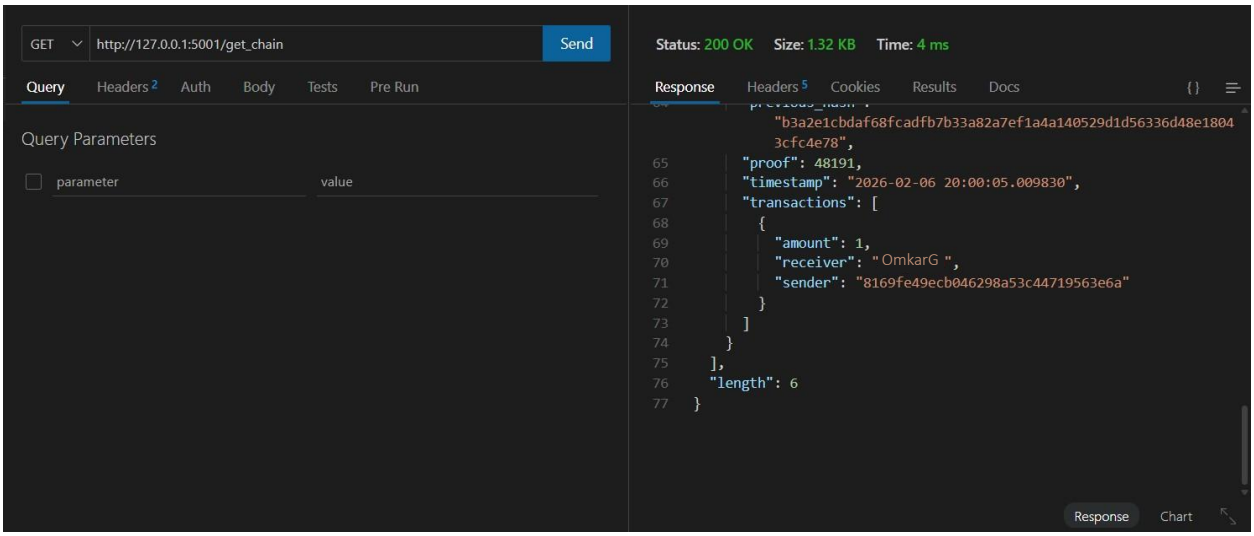


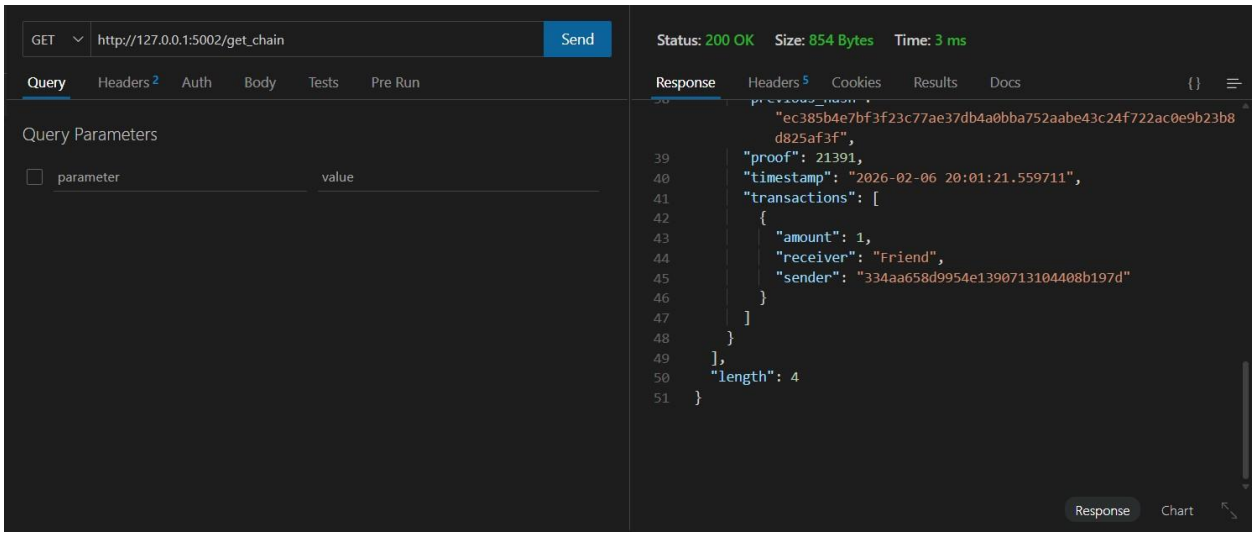
Node 5002 mined 3 times

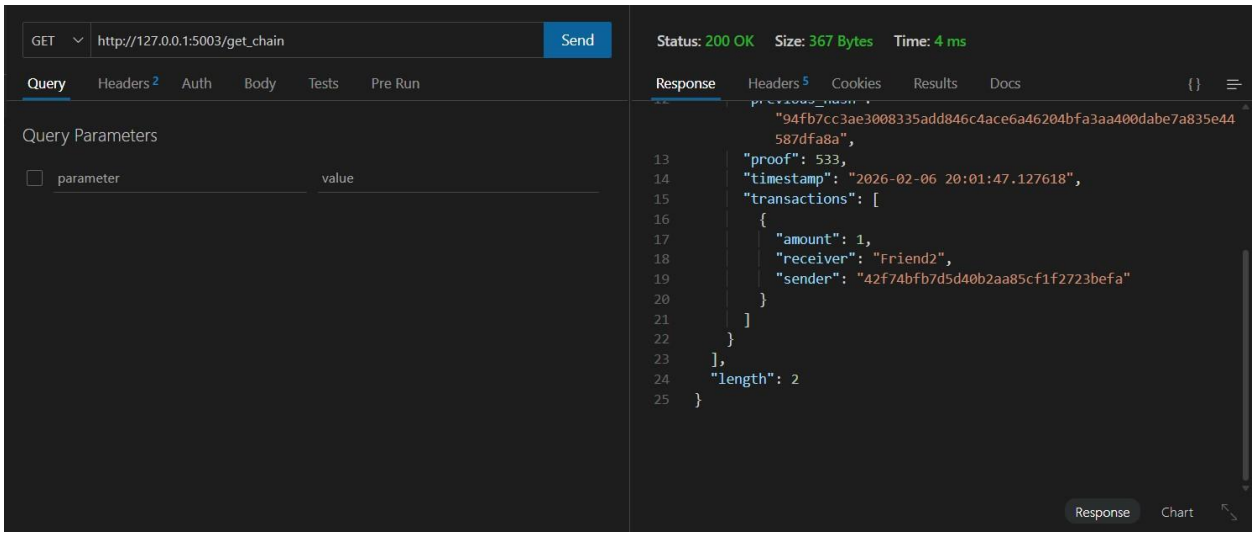


Step 4: Check chain before consensus

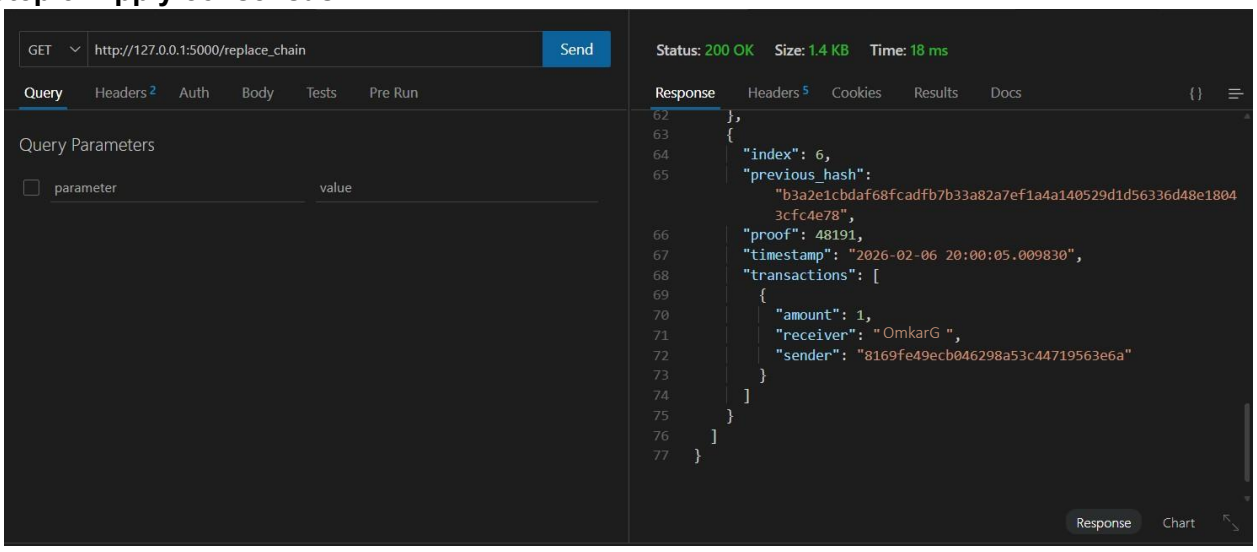




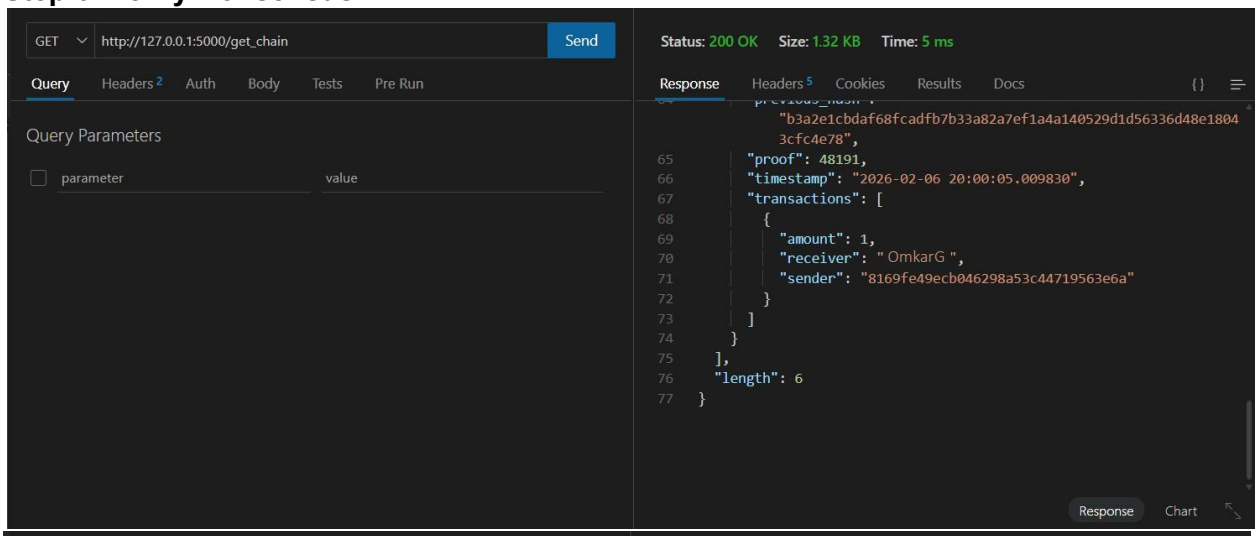


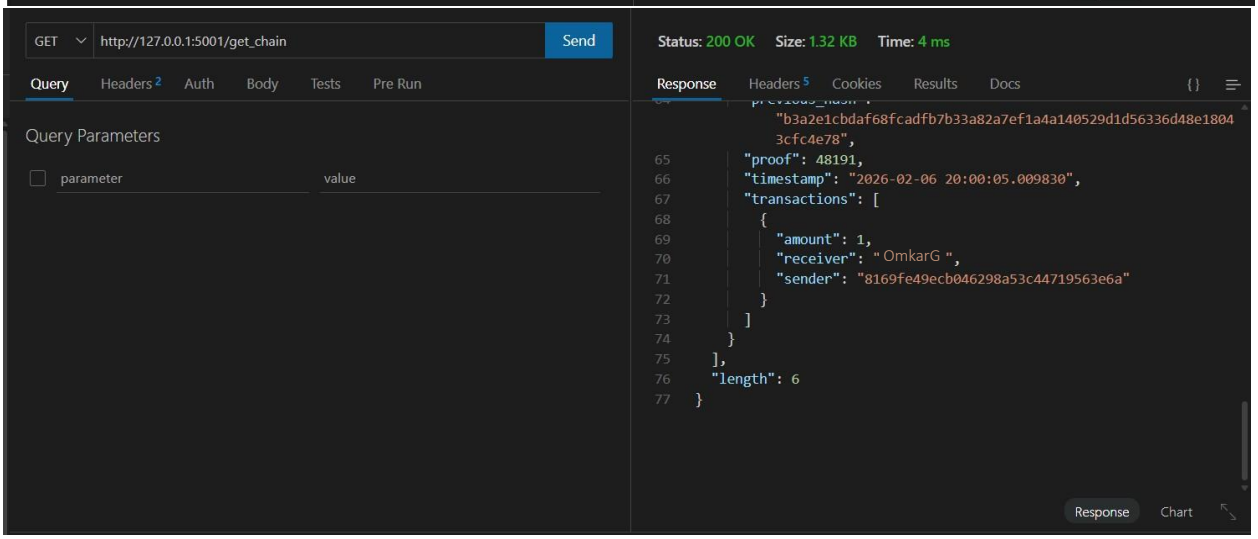


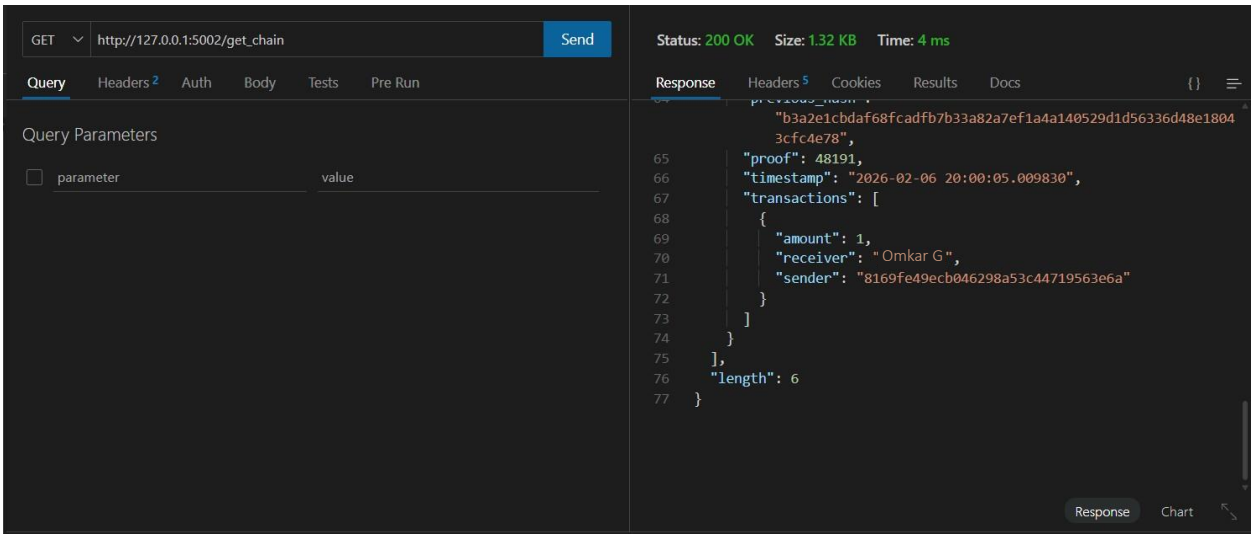
Step 5: Apply consensus



Step 6: Verify Consensus







CONCLUSION:

In this experiment, a cryptocurrency blockchain was successfully created using Python and Flask, where multiple nodes performed transactions and mining independently. Different blockchain lengths were generated across nodes and the consensus mechanism based on the longest chain rule was applied using the replace_chain function. After consensus, all nodes synchronized to the longest valid chain, demonstrating how decentralized networks maintain consistency, security, and agreement without a central authority.