Anuprita Mhapankar **D20A** **34**

Blockchain Experiment 3

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

THEORY:

Q1: Challenges in P2P networks.

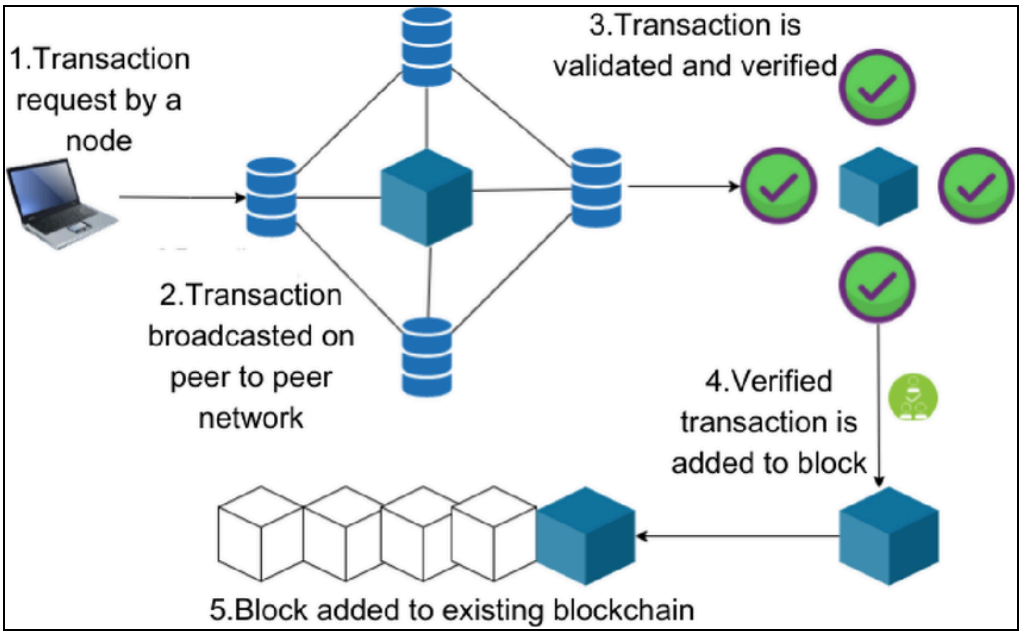
Peer-to-Peer (P2P) networks are decentralized systems where nodes communicate directly without a central authority. Although they provide transparency and fault tolerance, several challenges exist:

- **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:

```
# To be installed:
# Flask==0.12.2: pip install Flask==0.12.2
# requests==2.18.4: pip install requests==2.18.4

# Importing the libraries
import datetime
import hashlib
import json
from flask import Flask, jsonify, request
import requests
from uuid import uuid4
# Generate a unique id that is in hex
from urllib.parse import urlparse
# To parse url of the nodes

# Part 1 - Building a Blockchain

class Blockchain:

    def __init__(self):
        self.chain = []
        self.transactions = []
# Adding transactions before they are added to a block
        self.create_block(proof = 1, previous_hash = '0')
        self.nodes = set()
# Set is used as there is no order to be maintained as the nodes can be
from all around the globe

    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}
# Adding transactions to make the blockchain a cryptocurrency
        self.transactions = []
# The list of transacction should become empty after they are added to a
block
        self.chain.append(block)
        return block

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

    def proof_of_work(self, previous_proof):
        new_proof = 1
        check_proof = False
        while check_proof is False:
            hash_operation = hashlib.sha256(str(new_proof**2 -
previous_proof**2).encode()).hexdigest()
            if hash_operation[:4] == '0000':
                check_proof = True
            else:
                new_proof += 1
        return new_proof

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

    def is_chain_valid(self, chain):
        previous_block = chain[0]
        block_index = 1
        while block_index < len(chain):
            block = chain[block_index]
            if block['previous_hash'] != self.hash(previous_block):
                return False
            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

    # This method will add the trqnsaction to the list of trnsactions
    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
# It will return the block index to which the transaction should be added

    # This function will add the node containing an address to the set of
nodes created in init function
    def add_node(self, address):
        parsed_url = urlparse(address)
# urlparse will parse the url from the address
        self.nodes.add(parsed_url.netloc)
# Add is used and not append as it's a set. Netloc will only return
'127.0.0.1:5000'

    # Consensus Protocol. This function will replace all the shorter chain
with the longer chain in all the nodes on the network
    def replace_chain(self):
        network = self.nodes
# network variable is the set of nodes all around the globe
        longest_chain = None
# It will hold the longest chain when we scan the network
        max_length = len(self.chain)
# This will hold the length of the chain held by the node that runs this
function
        for node in network:
            response = requests.get(f'http://{node}/get_chain')
# Use get chain method already created to  get the length of the chain
            if response.status_code == 200:
```

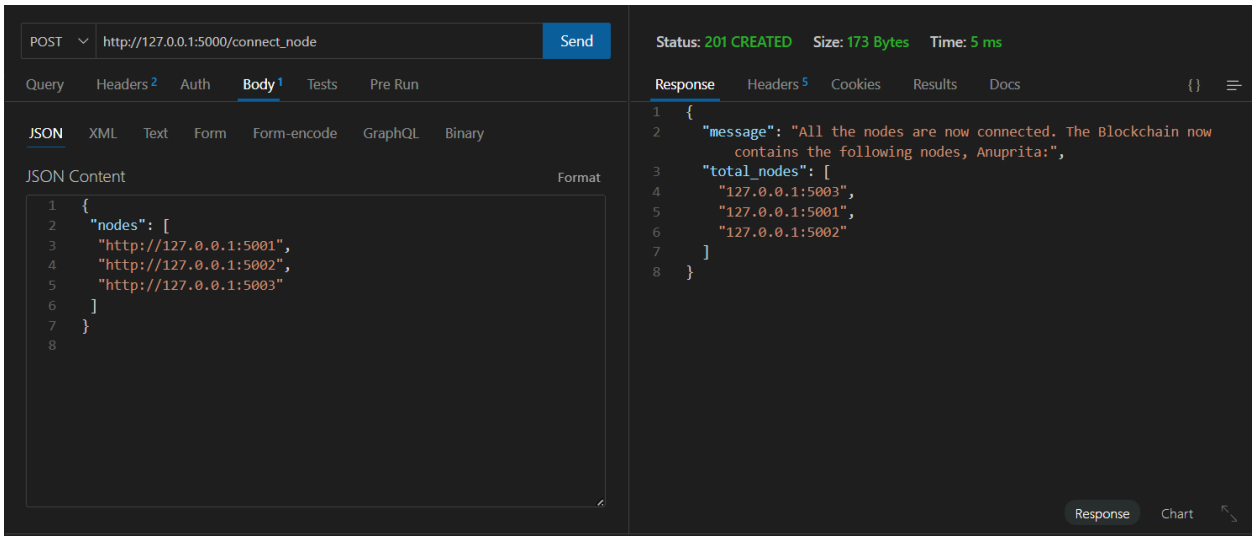
```
                length = response.json()['length']
# Extract the length of the chain from get_chain fiunction
                chain = response.json()['chain']
                if length > max_length and self.is_chain_valid(chain):
# We check if the length is bigger and if the chain is valid then
                    max_length = length
# We update the max length
                    longest_chain = chain
# We update the longest chain
        if longest_chain:
# If longest_chain is not none that means it was replaced
            self.chain = longest_chain
# Replace the chain of the current node with the longest chain
            return True
        return False
# Return false if current chain is the longest one

# Part 2 - Mining our Blockchain

# Creating a Web App
app = Flask(__name__)

# Creating an address for the node on Port 5000. We will create some other
nodes as well on different ports
node_address = str(uuid4()).replace('-', '')
#

# Creating a Blockchain
blockchain = Blockchain()

# Mining 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)
    blockchain.add_transaction(sender = node_address, receiver = 'Anu',
amount = 1) # Hadcoins to mine the block (A Reward). So the node gives 1
hadcoin to Abcde for mining the block
```

```
    block = blockchain.create_block(proof, previous_hash)
    response = {'message': 'Congratulations, you just mined a block,
Anuprita!',
                'index': block['index'],
                'timestamp': block['timestamp'],
                'proof': block['proof'],
                'previous_hash': block['previous_hash'],
                'transactions': block['transactions']}
    return jsonify(response), 200

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

# Checking if the Blockchain is valid
@app.route('/is_valid', methods = ['GET'])
def is_valid():
    is_valid = blockchain.is_chain_valid(blockchain.chain)
    if is_valid:
        response = {'message': 'The Blockchain is valid, Anuprita.'}
    else:
        response = {'message': 'The Blockchain is not valid, Anuprita.'}
    return jsonify(response), 200

# Adding a new transaction to the Blockchain
@app.route('/add_transaction', methods = ['POST'])
# Post method as we have to pass something to get something in return
def add_transaction():
    json = request.get_json()
# This will get the json file from postman. In Postman we will create a
json file in which we will pass the values for the keys in the json file
    transaction_keys = ['sender', 'receiver', 'amount']
    if not all(key in json for key in transaction_keys):
# Checking if all keys are available in json
        return 'Some elements of the transaction are missing', 400
    index = blockchain.add_transaction(json['sender'], json['receiver'],
json['amount'])
```

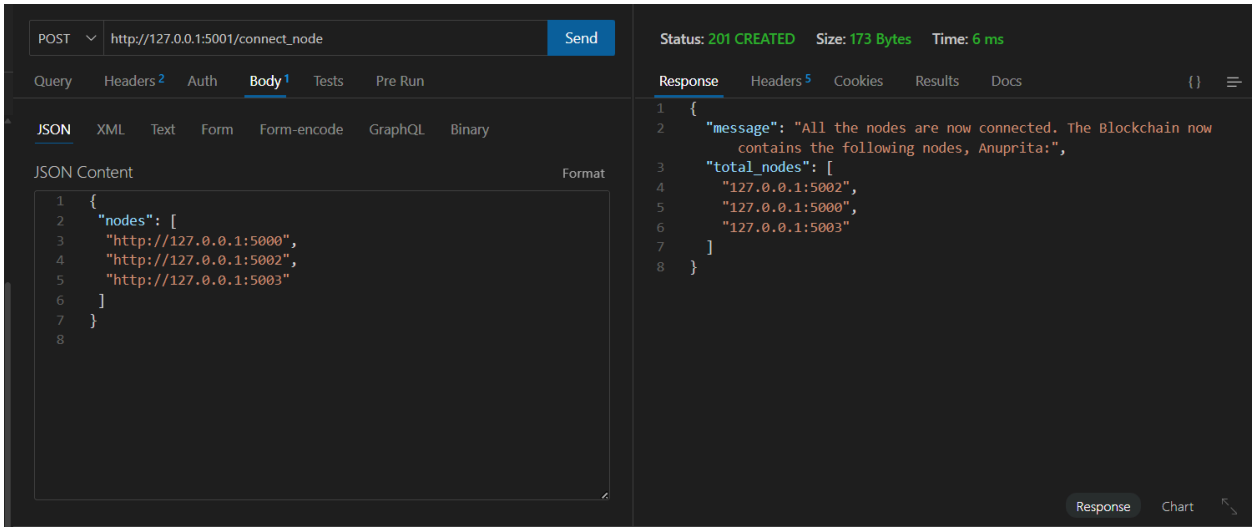
```
    response = {'message': f'This transaction will be added to Block
{index}'}
    return jsonify(response), 201
# Code 201 for creation

# Part 3 - Decentralizing our Blockchain

# Connecting new nodes
@app.route('/connect_node', methods = ['POST'])
# POST request to register the new nodes from the json file
def connect_node():
    json = request.get_json()
    nodes = json.get('nodes')
# Get the nodes from json file
    if nodes is None:
        return "No node", 400
    for node in nodes:
        blockchain.add_node(node)
    response = {'message': 'All the nodes are now connected. The
Blockchain now contains the following nodes, Anuprita:',
                'total_nodes': list(blockchain.nodes)}
    return jsonify(response), 201

# Replacing the chain by the longest chain if needed
@app.route('/replace_chain', methods = ['GET'])
def replace_chain():
    is_chain_replaced = blockchain.replace_chain()
    if is_chain_replaced:
        response = {'message': 'The nodes had different chains so the
chain was replaced by the longest one.',
                    'new_chain': blockchain.chain}
    else:
        response = {'message': 'All good. The chain is the largest one.',
                    'actual_chain': blockchain.chain}
    return jsonify(response), 200

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

Step 1: Connect Nodes

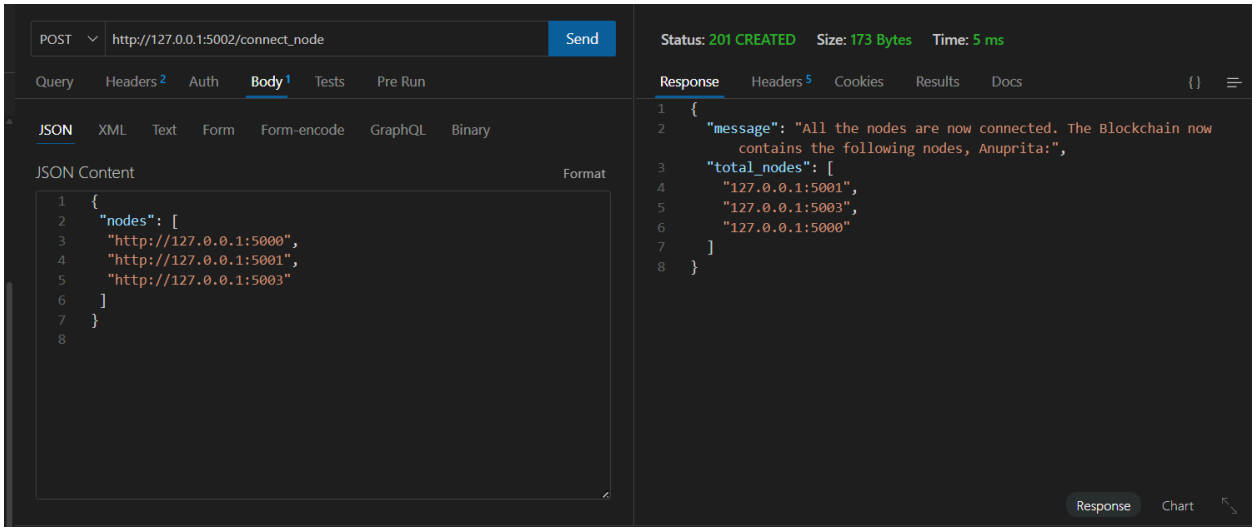
From 5000



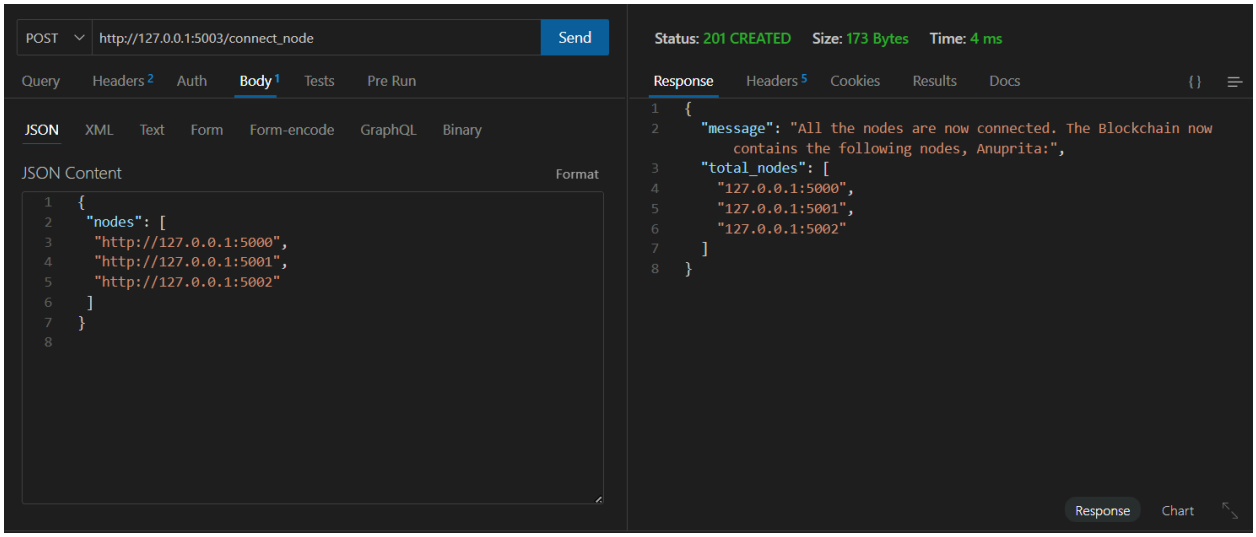
From 5001



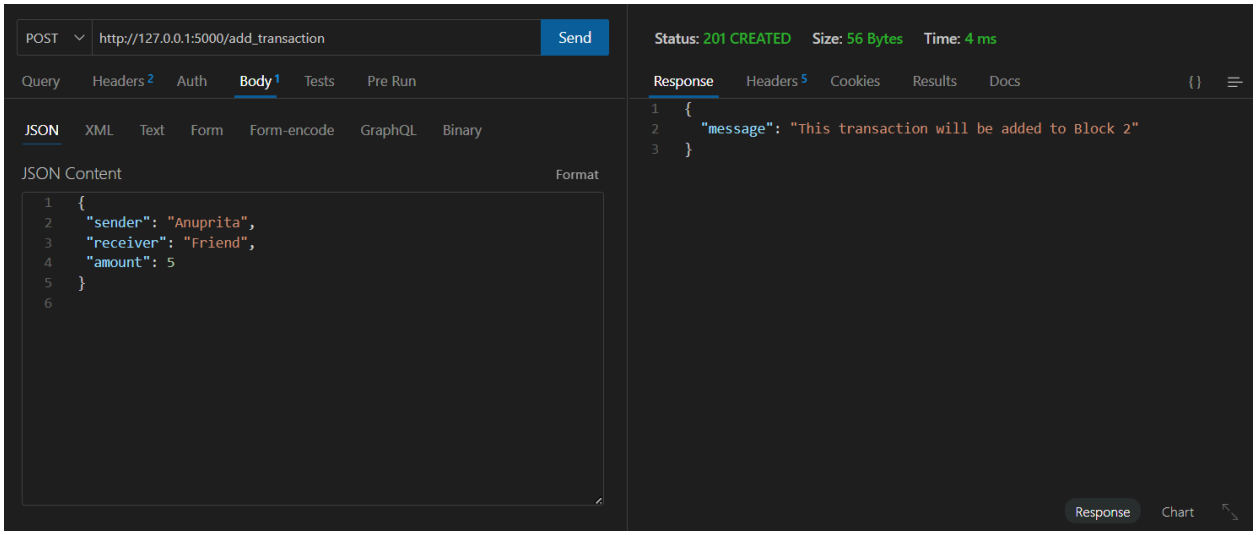
From 5002



From 5003

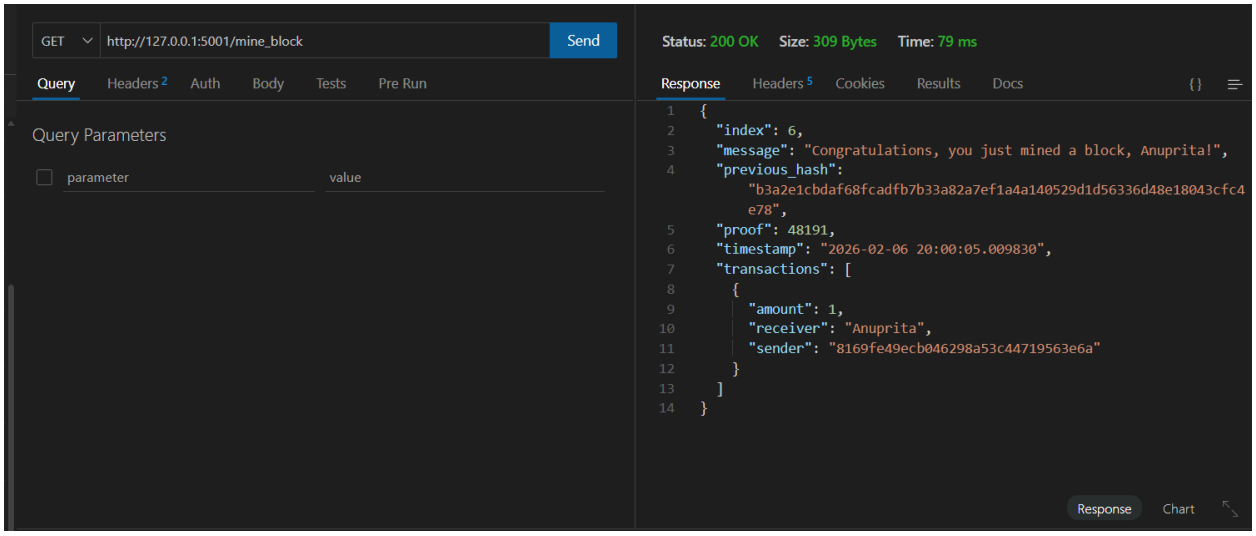


Step 2: Add transaction

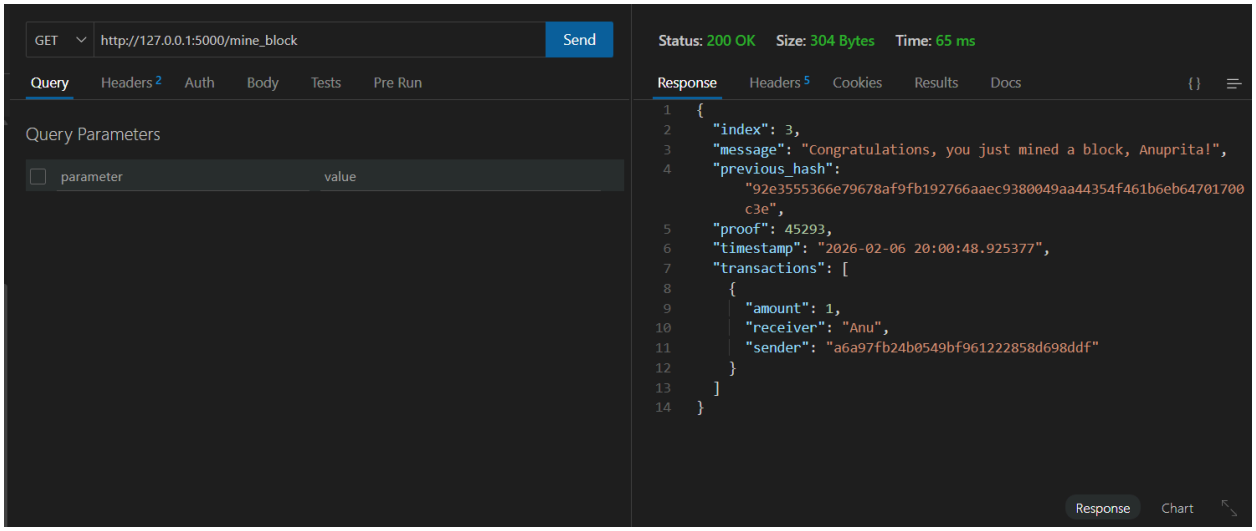


Step 3: Mine different blocks

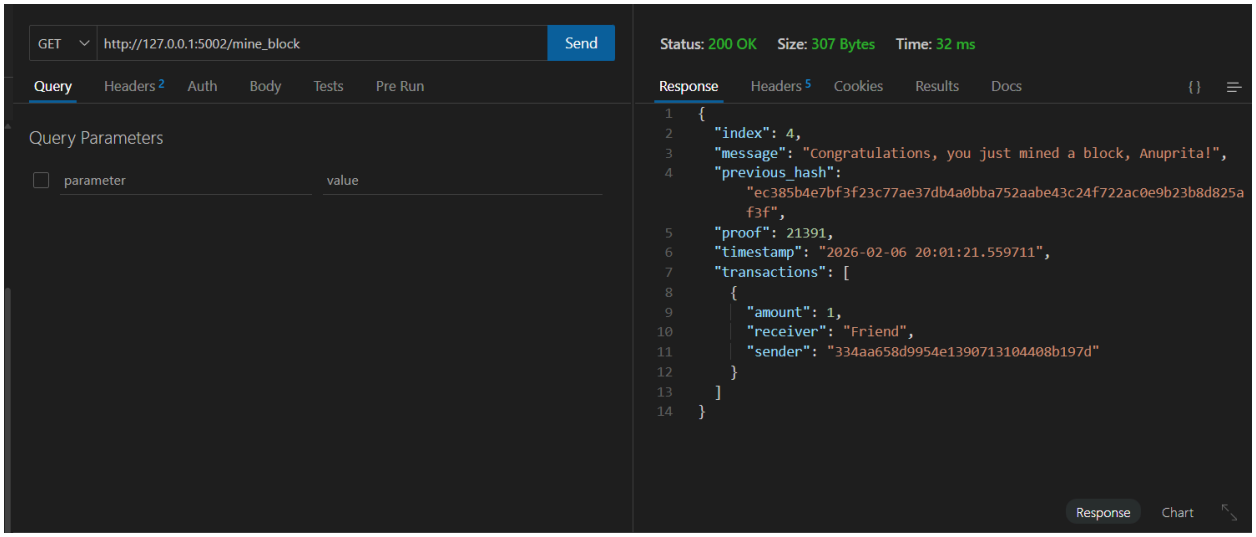
Node 5001 mined 5 times



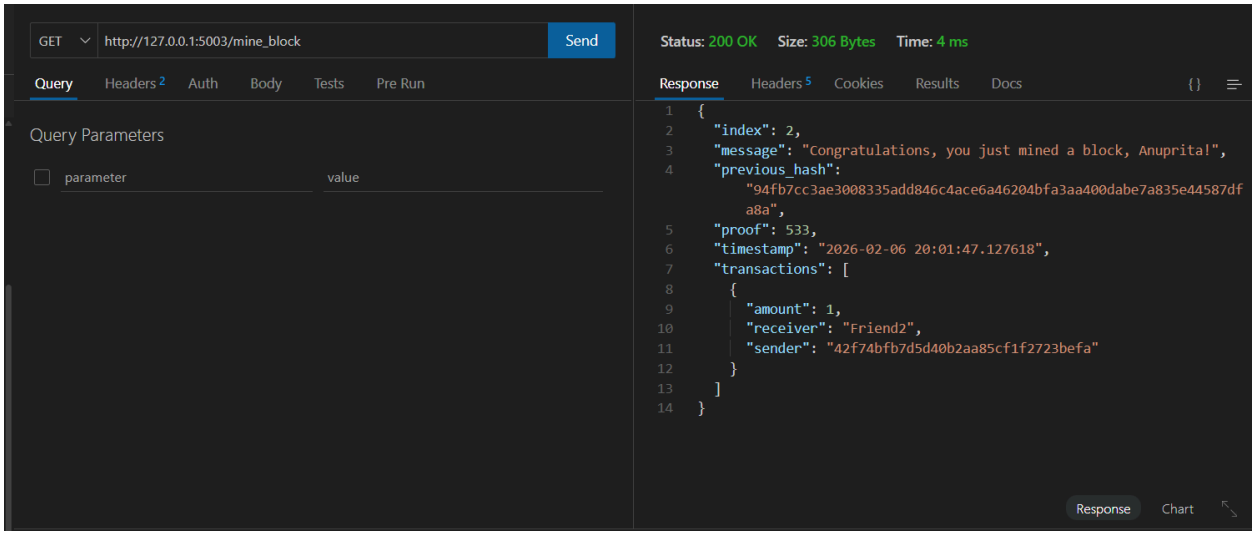
Node 5000 mined 2 times



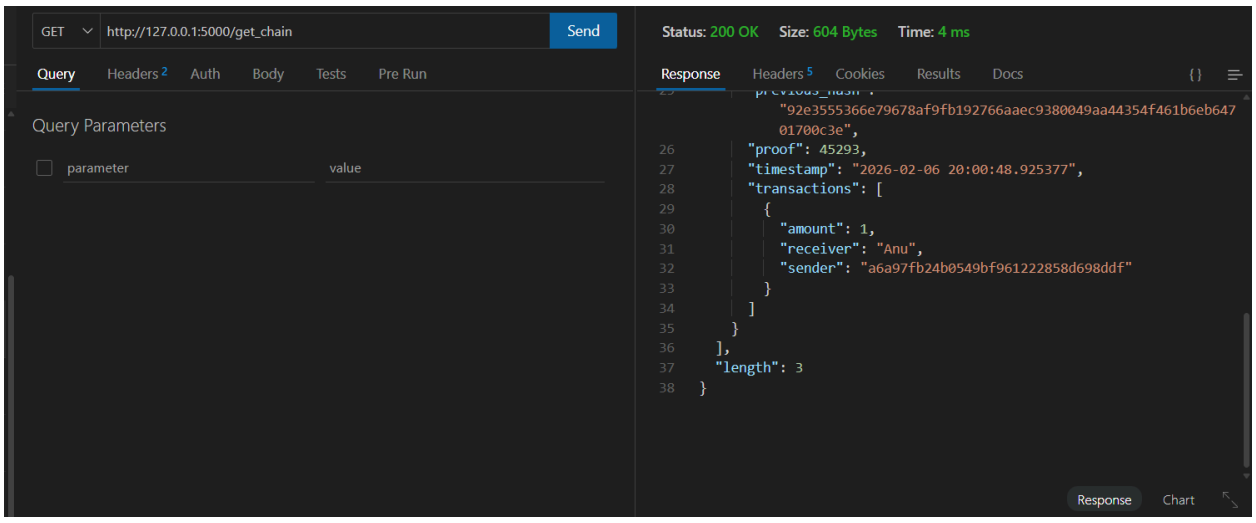
Node 5002 mined 3 times

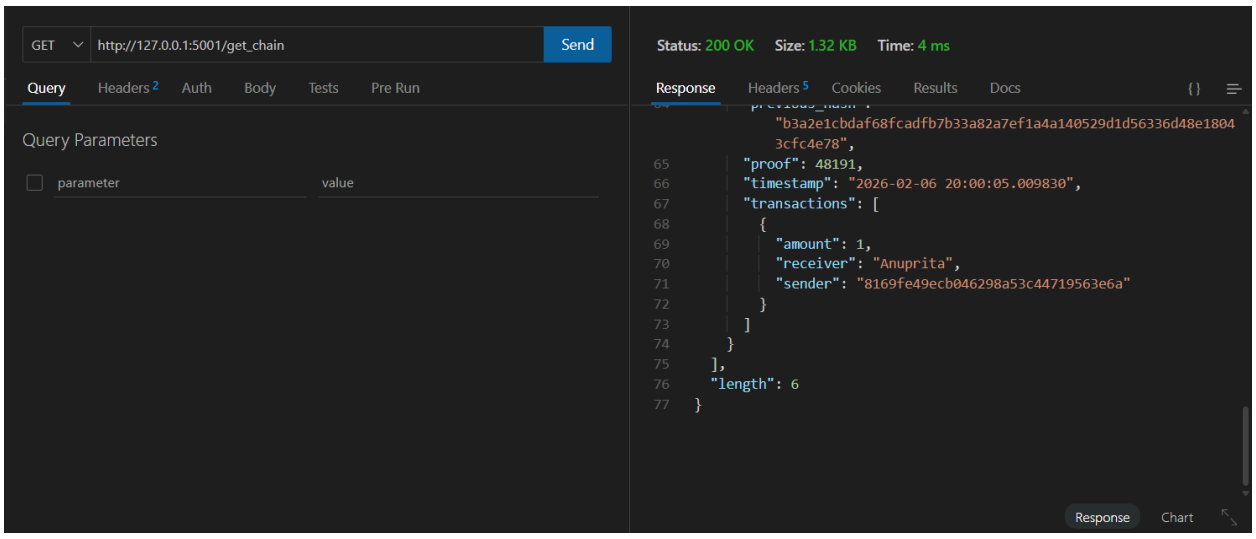


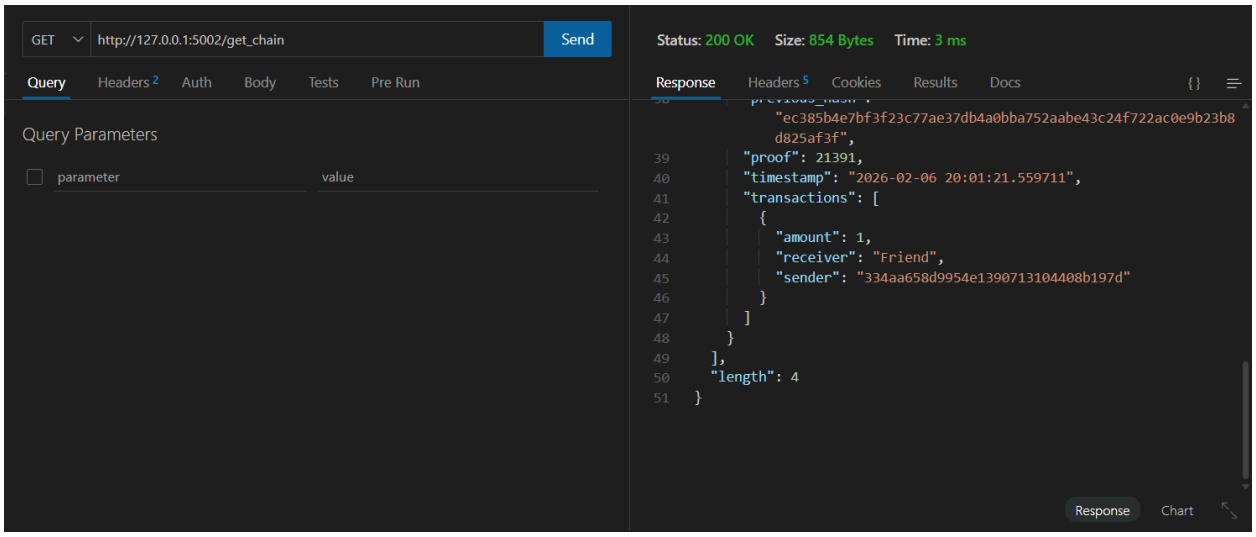
Node 5003 mined 1 time

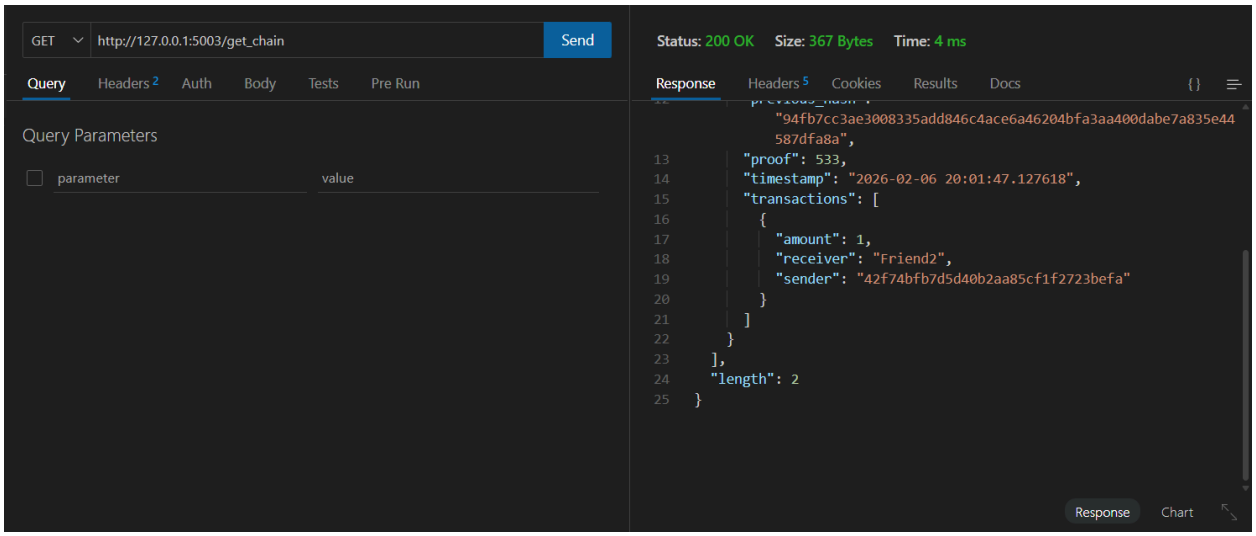


Step 4: Check chain before consensus

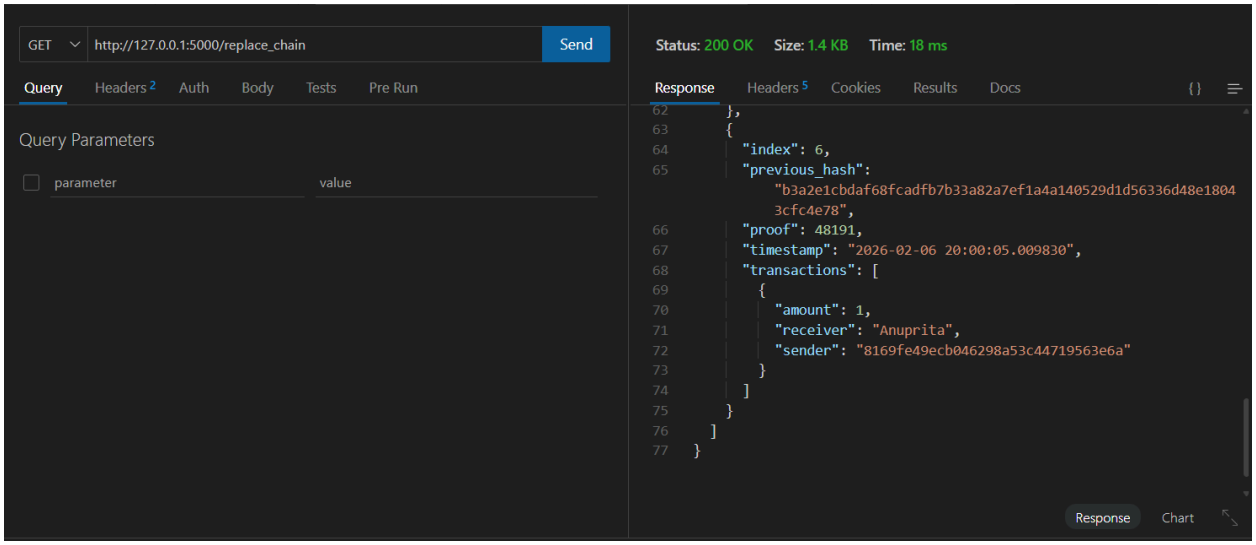


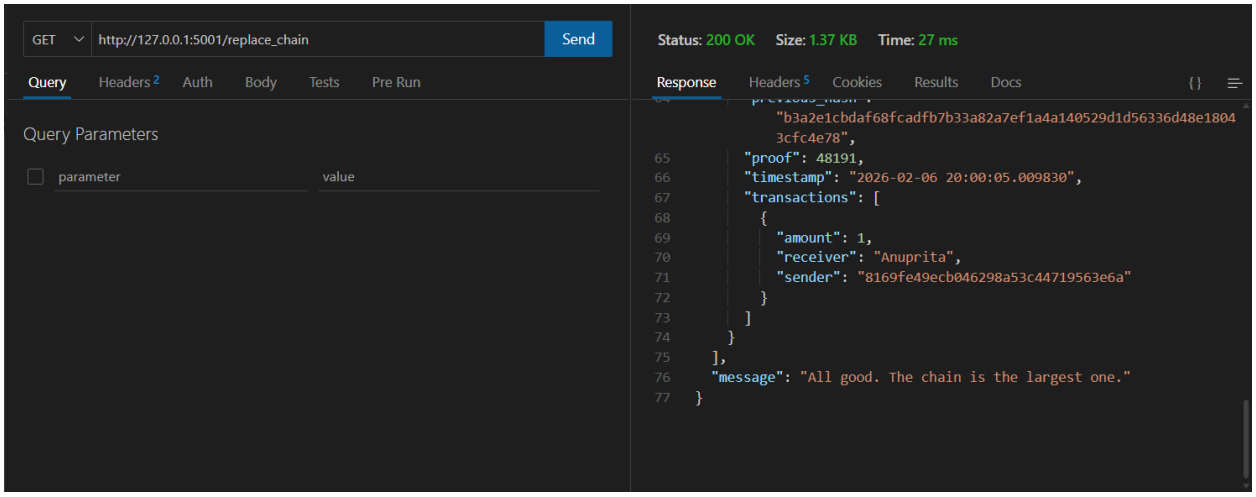


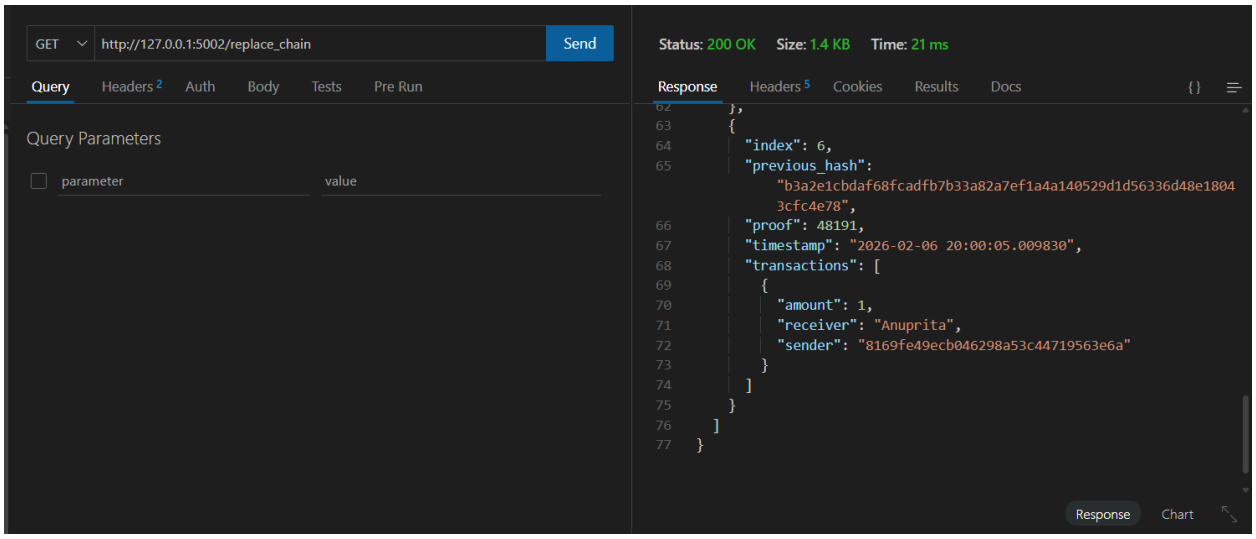


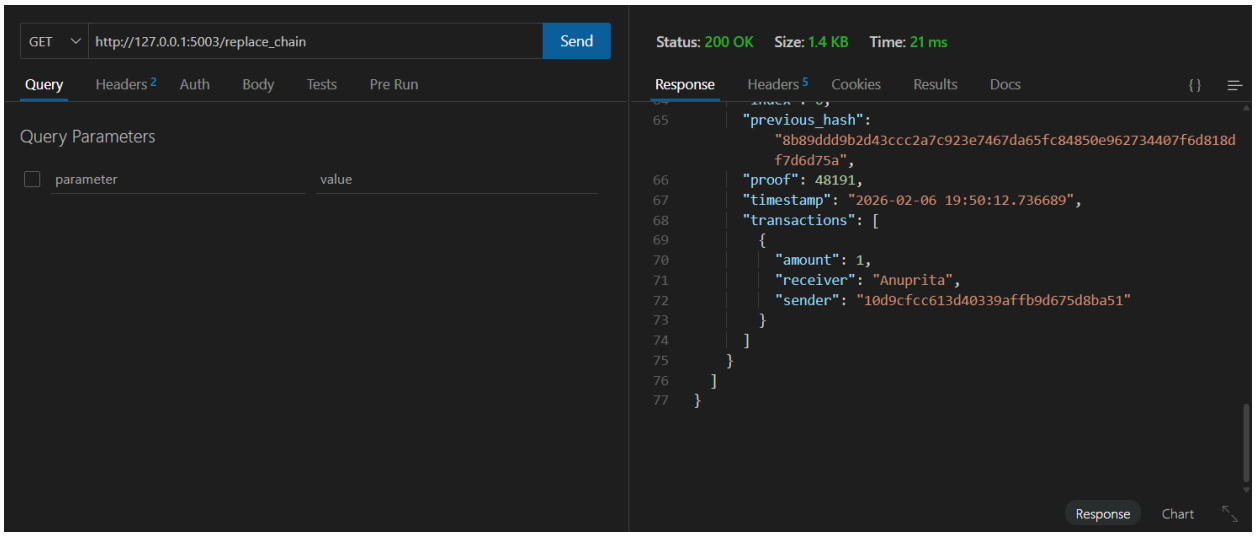


Step 5: Apply consensus

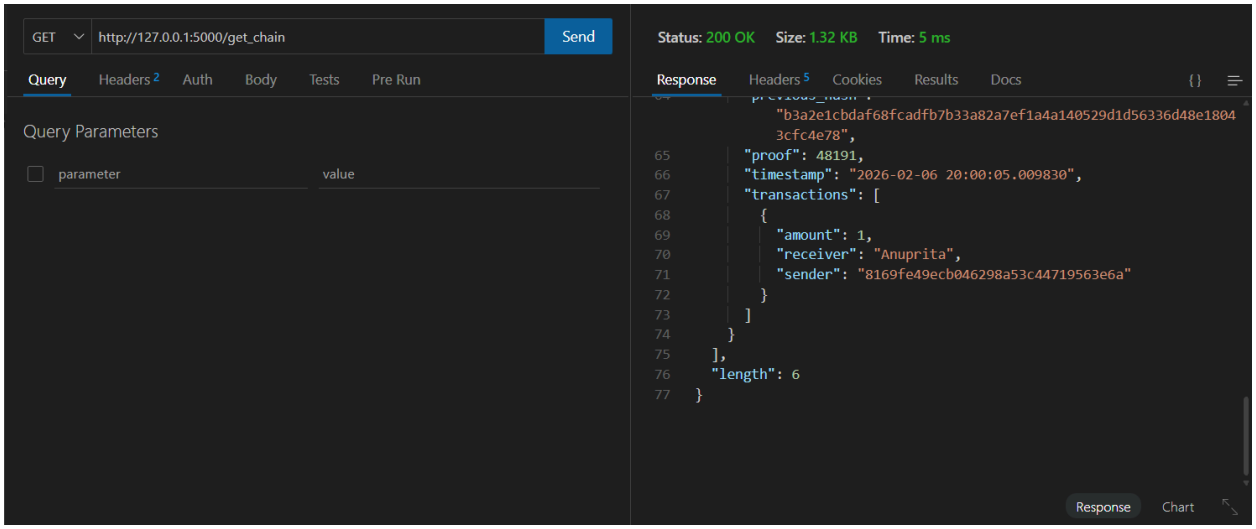


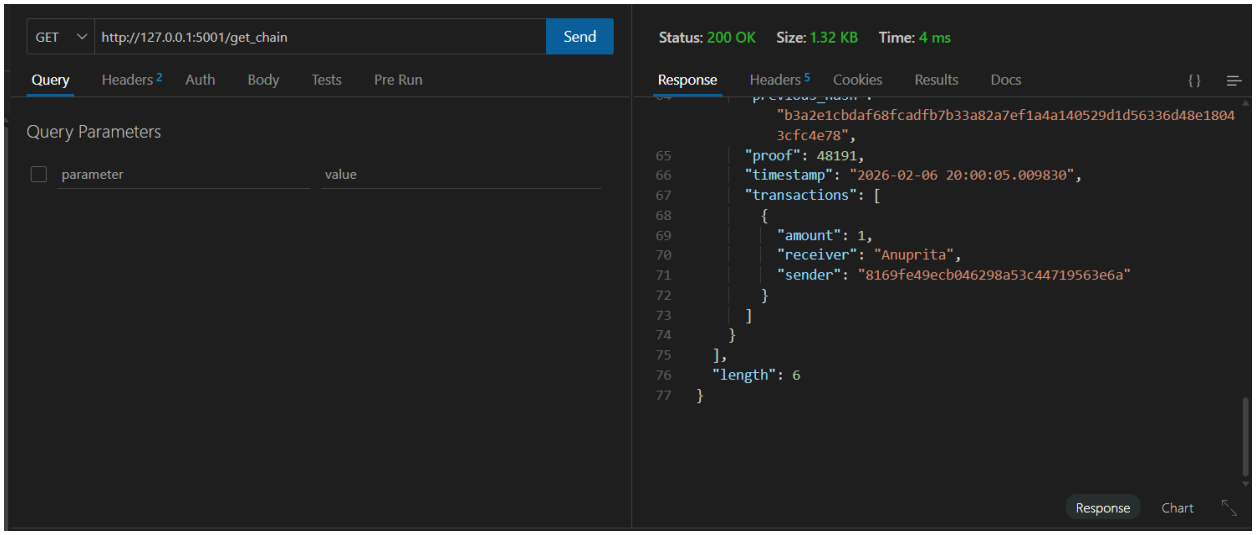

GET
http://127.0.0.1:5001/replace_chain
Send
Query
Headers 2
Auth
Body
Tests
Pre Run
Query Parameters
parameter
value
Status: 200 OK
Size: 1.37 KB
Time: 27 ms
Response
Headers 5
Cookies
Results
Docs
"b3a2e1cbdaf68fcadfb7b33a82a7ef1a4a140529d1d56336d48e18043cfc4e78",
"proof": 48191,
"timestamp": "2026-02-06 20:00:05.009830",
"transactions": [
"amount": 1,
"receiver": "Anuprita",
"sender": "8169fe49ecb046298a53c44719563e6a"
],
"message": "All good. The chain is the largest one."

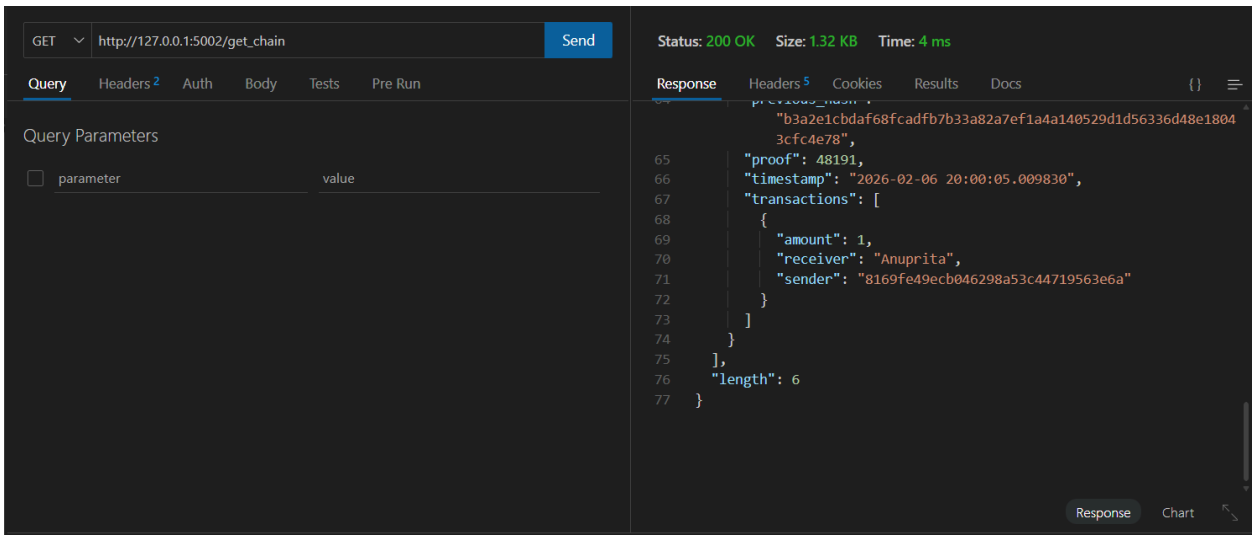

GET
http://127.0.0.1:5002/replace_chain
Send
Query
Headers 2
Auth
Body
Tests
Pre Run
Query Parameters
parameter
value
Status: 200 OK
Size: 1.4 KB
Time: 21 ms
Response
Headers 5
Cookies
Results
Docs
"index": 6,
"previous_hash": "b3a2e1cbdaf68fcadfb7b33a82a7ef1a4a140529d1d56336d48e18043cfc4e78",
"proof": 48191,
"timestamp": "2026-02-06 20:00:05.009830",
"transactions": [
"amount": 1,
"receiver": "Anuprita",
"sender": "8169fe49ecb046298a53c44719563e6a"
Response
Chart

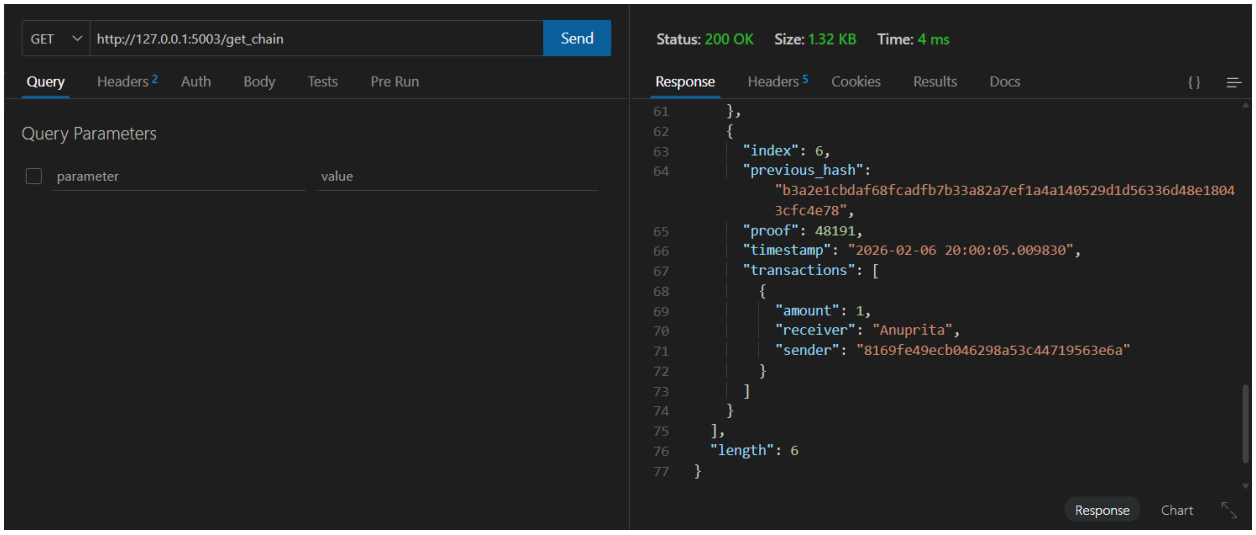

GET
http://127.0.0.1:5003/replace_chain
Send
Query
Headers 2
Auth
Body
Tests
Pre Run
Query Parameters
parameter
value
Status: 200 OK
Size: 1.4 KB
Time: 21 ms
Response
Headers 5
Cookies
Results
Docs
"previous_hash": "8b89ddd9b2d43ccc2a7c923e7467da65fc84850e962734407f6d818df7d6d75a",
"proof": 48191,
"timestamp": "2026-02-06 19:50:12.736689",
"transactions": [
"amount": 1,
"receiver": "Anuprita",
"sender": "10d9cfcc613d40339affb9d675d8ba51"
Response
Chart

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.