

## 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.

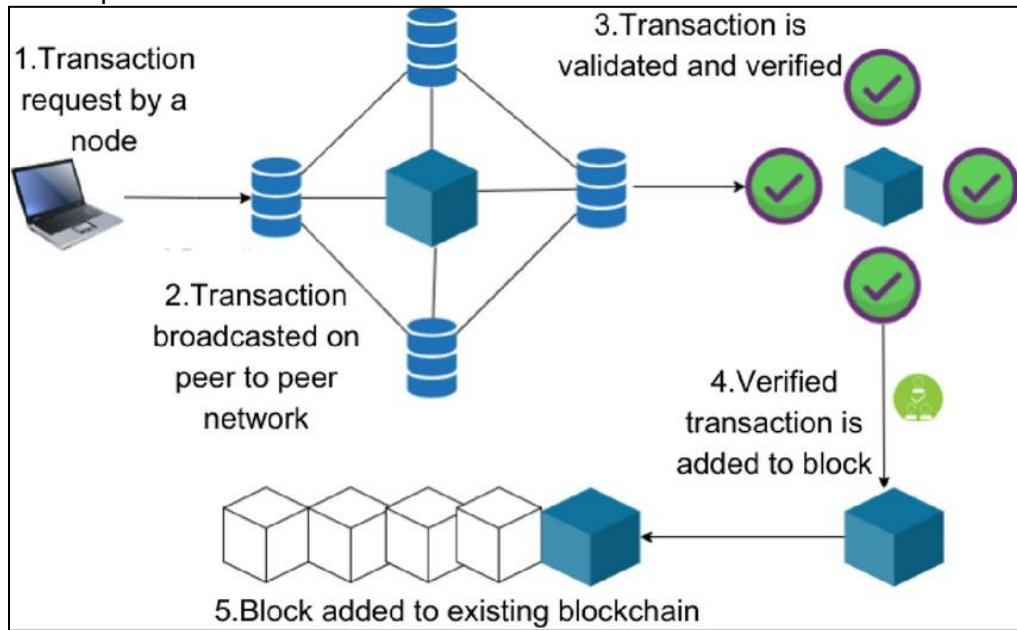
- **Step 5: Mining and Block Creation**

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]
            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

    # Check previous hash link
    def check_previous_hash_link(self, block):
        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

    # Check proof of work
    def check_proof_of_work(self, previous_proof, proof):
        previous_proof_hex = previous_proof.encode('hex')
        proof_hex = proof.encode('hex')

        hash_operation = hashlib.sha256(
            str(proof_hex**2 - previous_proof_hex**2).encode()
        ).hexdigest()

        if hash_operation[:4] != '0000':
            return False
        previous_block = block
        block_index += 1

    # 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()
        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__)

@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)
    block = blockchain.create_block(proof, previous_hash)
    response = jsonify({
        'index': block['index'],
        'timestamp': block['timestamp'],
        'proof': block['proof'],
        'previous_hash': block['previous_hash'],
        'transactions': block['transactions']
    })
    blockchain.add_transaction(sender='0', receiver='user', amount=1)
    return response

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

@app.route('/add_node', methods=['POST'])
def add_node():
    node_address = request.json['node']
    blockchain.add_node(node_address)
    response = jsonify({'message': 'Node added successfully'})
    return response

@app.route('/replace_chain', methods=['GET'])
def replace_chain():
    replaced = blockchain.replace_chain()
    if replaced:
        response = jsonify({'message': 'Chain replaced successfully'})
    else:
        response = jsonify({'message': 'Chain not replaced'})
    return response

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

```

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

POST [http://127.0.0.1:5000/connect\\_node](http://127.0.0.1:5000/connect_node) Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "nodes": [
3     "http://127.0.0.1:5001",
4     "http://127.0.0.1:5002",
5     "http://127.0.0.1:5003"
6   ]
7 }
```

Status: 201 CREATED Size: 173 Bytes Time: 5 ms

Response Headers 5 Cookies Results Docs { } ⚙

```
1 {
2   "message": "All the nodes are now connected. The Blockchain now
3   contains the following nodes, Omkar G ,
4   "total_nodes": [
5     "127.0.0.1:5003",
6     "127.0.0.1:5001",
7     "127.0.0.1:5002"
8   ]
}
```

Response Chart ↻

From 5001

POST [http://127.0.0.1:5001/connect\\_node](http://127.0.0.1:5001/connect_node) Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "nodes": [
3     "http://127.0.0.1:5000",
4     "http://127.0.0.1:5002",
5     "http://127.0.0.1:5003"
6   ]
7 }
```

Status: 201 CREATED Size: 173 Bytes Time: 6 ms

Response Headers 5 Cookies Results Docs { } ⚙

```
1 {
2   "message": "All the nodes are now connected. The Blockchain now
3   contains the following nodes, Omkar G ,
4   "total_nodes": [
5     "127.0.0.1:5002",
6     "127.0.0.1:5000",
7     "127.0.0.1:5003"
8   ]
}
```

Response Chart ↻

From 5002

Status: 201 CREATED Size: 173 Bytes Time: 5 ms

```

1 {
2   "message": "All the nodes are now connected. The Blockchain now
    contains the following nodes, Omkar G .",
3   "total_nodes": [
4     "127.0.0.1:5001",
5     "127.0.0.1:5003",
6     "127.0.0.1:5000"
7   ]
8 }
```

From 5003

Status: 201 CREATED Size: 173 Bytes Time: 4 ms

```

1 {
2   "message": "All the nodes are now connected. The Blockchain now
    contains the following nodes, Omkar G .",
3   "total_nodes": [
4     "127.0.0.1:5000",
5     "127.0.0.1:5001",
6     "127.0.0.1:5002"
7   ]
8 }
```

## Step 2: Add transaction

Status: 201 CREATED Size: 56 Bytes Time: 4 ms

```

1 {
2   "message": "This transaction will be added to Block 2"
3 }
```

### Step 3: Mine different blocks

Node 5001 mined 5 times

The screenshot shows a Postman request to `http://127.0.0.1:5001/mine_block`. The response status is `200 OK`, size is `309 Bytes`, and time is `79 ms`. The response body is a JSON object:

```
1 {  
2   "index": 6,  
3   "message": "Congratulations, you just mined a block, Omkar G !",  
4   "previous_hash":  
5     "b3a2e1cbda68fcadfb7b33a82a7ef1a4a140529d1d56336d48e18043fc4  
6     e78",  
7   "proof": 48191,  
8   "timestamp": "2026-02-06 20:00:05.009830",  
9   "transactions": [  
10    {  
11      "amount": 1,  
12      "receiver": "OmkarG ",  
13      "sender": "8169fe49ecb046298a53c44719563e6a"  
14    }  
15  ]  
16 }
```

Node 5000 mined 2 times

The screenshot shows a Postman request to `http://127.0.0.1:5000/mine_block`. The response status is `200 OK`, size is `304 Bytes`, and time is `65 ms`. The response body is a JSON object:

```
1 {  
2   "index": 3,  
3   "message": "Congratulations, you just mined a block, Omkar G !",  
4   "previous_hash":  
5     "92e355366e79678af9fb192766aaec9380049aa44354f461b6eb64701700  
6     c3e",  
7   "proof": 45293,  
8   "timestamp": "2026-02-06 20:00:48.925377",  
9   "transactions": [  
10    {  
11      "amount": 1,  
12      "receiver": "OG ",  
13      "sender": "a6a97fb24b0549bf961222858d698ddf"  
14    }  
15  ]  
16 }
```

Node 5002 mined 3 times

GET [http://127.0.0.1:5002/mine\\_block](http://127.0.0.1:5002/mine_block)

Status: 200 OK | Size: 307 Bytes | Time: 32 ms

Response Headers: Content-Type: application/json; charset=utf-8

```

1  {
2    "index": 4,
3    "message": "Congratulations, you just mined a block, Omkar G",
4    "previous_hash":
5      "ec385b4e7bf3f23c77ae37db4a0bba752aab43c24f722ac0e9b23b8d825a
6      f3f",
7    "proof": 21391,
8    "timestamp": "2026-02-06 20:01:21.559711",
9    "transactions": [
10      {
11        "amount": 1,
12        "receiver": "Friend",
13        "sender": "334aa658d9954e1390713104408b197d"
14      }
15    }
16  }

```

#### Step 4: Check chain before consensus

GET [http://127.0.0.1:5000/get\\_chain](http://127.0.0.1:5000/get_chain)

Status: 200 OK | Size: 604 Bytes | Time: 4 ms

Response Headers: Content-Type: application/json; charset=utf-8

```

1  {
2    "index": 0,
3    "previous_hash": "92e355366e79678af9fb192766aaec9380049aa44354f461b6eb647
4      01700c3e",
5    "proof": 45293,
6    "timestamp": "2026-02-06 20:00:48.925377",
7    "transactions": [
8      {
9        "amount": 1,
10       "receiver": "OG",
11       "sender": "a6a97fb24b0549bf961222858d698ddf"
12     }
13   ],
14   "length": 1
15 }

```

GET [http://127.0.0.1:5001/get\\_chain](http://127.0.0.1:5001/get_chain)

Status: 200 OK | Size: 1.32 KB | Time: 4 ms

Response Headers: Content-Type: application/json; charset=utf-8

```

1  {
2    "index": 0,
3    "previous_hash": "b3a2e1cbda68fcadfb7b33a82a7ef1a4a140529d1d5633d648e1804
4      3cfca4e78",
5    "proof": 48191,
6    "timestamp": "2026-02-06 20:00:05.009830",
7    "transactions": [
8      {
9        "amount": 1,
10       "receiver": "OmkarG",
11       "sender": "8169fe49ecb046298a53c44719563e6a"
12     }
13   ],
14   "length": 1
15 }

```

GET http://127.0.0.1:5002/get\_chain

**Query Parameters**

<input type="checkbox"/> parameter	value
------------------------------------	-------

**Response**

```
Status: 200 OK Size: 854 Bytes Time: 3 ms

{
  "index": "ec385b4e7bf3f23c77ae37db4a0bba752aab43c24f722ac0e9b23b8d825af3f",
  "proof": 21391,
  "timestamp": "2026-02-06 20:01:21.559711",
  "transactions": [
    {
      "amount": 1,
      "receiver": "Friend",
      "sender": "334aa658d9954e1390713104408b197d"
    }
  ],
  "length": 4
}
```

GET http://127.0.0.1:5003/get\_chain

**Query Parameters**

<input type="checkbox"/> parameter	value
------------------------------------	-------

**Response**

```
Status: 200 OK Size: 367 Bytes Time: 4 ms

{
  "index": "94fb7cc3ae3008335add846c4ace6a46204bfa3aa400dabe7a835e44587dfa8a",
  "proof": 533,
  "timestamp": "2026-02-06 20:01:47.127618",
  "transactions": [
    {
      "amount": 1,
      "receiver": "Friend2",
      "sender": "42f74bf7bd5d40b2aa85cf1f2723befa"
    }
  ],
  "length": 2
}
```

## Step 5: Apply consensus

GET http://127.0.0.1:5000/replace\_chain

**Query Parameters**

<input type="checkbox"/> parameter	value
------------------------------------	-------

**Response**

```
Status: 200 OK Size: 1.4 KB Time: 18 ms

{
  "index": 6,
  "previous_hash": "b3a2e1cbdaf68fcadfb7b33a82a7ef1a4a140529d1d56336d48e18043fc4e78",
  "proof": 48191,
  "timestamp": "2026-02-06 20:00:05.009830",
  "transactions": [
    {
      "amount": 1,
      "receiver": "OmkarG",
      "sender": "8169fe49ecb046298a53c44719563e6a"
    }
  ]
}
```

## Step 6: Verify Consensus

GET  Send

Query Headers 2 Auth Body Tests Pre Run

Query Parameters

parameter  value

Status: 200 OK Size: 1.32 KB Time: 5 ms

Response Headers 5 Cookies Results Docs

```
54     "proof": 48191,
55     "timestamp": "2026-02-06 20:00:05.009830",
56     "transactions": [
57       {
58         "amount": 1,
59         "receiver": "OmkarG",
60         "sender": "8169fe49ecb046298a53c44719563e6a"
61       }
62     ],
63     "length": 6
64   }
```

GET  Send

Query Headers 2 Auth Body Tests Pre Run

Query Parameters

parameter  value

Status: 200 OK Size: 1.32 KB Time: 4 ms

Response Headers 5 Cookies Results Docs

```
54     "proof": 48191,
55     "timestamp": "2026-02-06 20:00:05.009830",
56     "transactions": [
57       {
58         "amount": 1,
59         "receiver": "OmkarG",
60         "sender": "8169fe49ecb046298a53c44719563e6a"
61       }
62     ],
63     "length": 6
64   }
```

GET  Send

Query Headers 2 Auth Body Tests Pre Run

Query Parameters

parameter  value

Status: 200 OK Size: 1.32 KB Time: 4 ms

Response Headers 5 Cookies Results Docs

```
54     "proof": 48191,
55     "timestamp": "2026-02-06 20:00:05.009830",
56     "transactions": [
57       {
58         "amount": 1,
59         "receiver": "Omkar G",
60         "sender": "8169fe49ecb046298a53c44719563e6a"
61       }
62     ],
63     "length": 6
64   }
```

## **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.