SRM Institute of Science and Technology
Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204
Department of Computer Applications
Circular – 2023-24
BCA DS 6th Sem

Blockchain Technology (UDS23D04J)

Lab Manual

Lab 1: Public Ledger vs. Private Ledger

Title: Public Ledger vs. Private Ledger

Aim: To understand the differences between public and private ledgers in blockchain technology.

Procedure:

1. Research the characteristics of public and private ledgers.
2. Create a table comparing them based on the following attributes:

   Access

   Network Actors

   Native Token

   Security

   Speed

   Examples

3. Write a brief report summarizing the key differences.

Source Code: (This is a conceptual lab, so no specific code is required. The output is a table and a report.)

Input: Research on public and private ledgers.

Expected Output:

A table comparing public and private ledgers.

A report summarizing the differences.

Lab 2: Peer-to-Peer Network Simulation

Title: Peer-to-Peer Network Simulation

Aim: To simulate a basic peer-to-peer (P2P) network.

Procedure:

1. Choose a simulation tool or programming language (e.g., Python with a networking library).
2. Design a simple P2P network with a few nodes.
3. Implement basic functionalities:

   Node connection/disconnection

   Message passing between nodes

4. Visualize the network and message flow.

Source Code:

```
# Example Python code (Conceptual)
import socket
import threading

def handle_connection(conn, addr):
    """Handles communication with a connected node."""
    print(f"Connection from {addr}")
    while True:
        data = conn.recv(1024)
        if not data:
            break
        print(f"Received from {addr}: {data.decode()}")
        conn.sendall(b"Message received")  # Echo back
    conn.close()
    print(f"Connection with {addr} closed")

def start_server(host, port):
    """Starts the P2P server."""
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((host, port))
    s.listen()
    print(f"Server listening on {host}:{port}")
    while True:
        conn, addr = s.accept()
        thread = threading.Thread(target=handle_connection, args=(conn,
addr))
        thread.start()

if __name__ == "__main__":
    host = '127.0.0.1'  # Or 0.0.0.0 to listen on all interfaces
    port = 12345
    start_server(host, port)
```

Input: Run the Python code (or equivalent in another language). Connect multiple instances to simulate nodes. Send messages between them.

Expected Output:

Simulation of nodes connecting and sending messages.

Visualization (if implemented) of the network.

Output of messages being sent and received.

Lab 3: Explore Blockchain Tools

Title: Explore Blockchain Tools

Aim: To explore various tools and platforms available for blockchain development.

Procedure:

1. Research different blockchain tools (e.g., Ganache, Truffle, Remix, Hyperledger Fabric, Corda).
2. For each tool, document its:

   Purpose

   Features

   Use cases

   Setup process (briefly)

3. Choose one tool and set up a basic development environment.

Source Code: (This is primarily an exploration lab, so the "source code" is the configuration and setup of the chosen tool.) For example, if you choose Ganache, you might include the configuration file.

Input: Research on blockchain tools.

Expected Output:

A report summarizing different blockchain tools.

A basic development environment set up with one chosen tool.

Lab 4: Bitcoin Wallet Creation and Transactions

Title: Bitcoin Wallet Creation and Transactions

Aim: To create a Bitcoin wallet and perform basic transactions.

Procedure:

1. Choose a Bitcoin wallet (e.g., a software wallet like Electrum, or a testnet wallet).
2. Install and set up the wallet.
3. Generate a Bitcoin address.
4. Obtain testnet Bitcoin (if using testnet).
5. Send a transaction to another address.
6. Verify the transaction on a block explorer.

Source Code: (This lab involves using existing software, not writing code. The "source code" is the sequence of commands/actions within the wallet software.)

Input: Interaction with a Bitcoin wallet application.

Expected Output:

A created Bitcoin wallet.

A generated Bitcoin address.

A successful Bitcoin transaction.

Verification of the transaction on a block explorer.

Lab 5: Bitcoin Mining Simulation

Title: Bitcoin Mining Simulation

Aim: To simulate the process of Bitcoin mining.

Procedure:

1. Research the Bitcoin mining process (proof-of-work).
2. Develop a simplified simulation of the mining process (e.g., in Python). This could involve:

    Generating a block header.

    Finding a nonce that satisfies a target difficulty.

3. Track the time or computational effort required to "mine" a block.

Source Code:

```
# Example Python code (Conceptual)
import hashlib
import time

def mine_block(block_header, difficulty):
    """Simulates mining a block."""
    nonce = 0
    while True:
        data = block_header + str(nonce)
        hash_value = hashlib.sha256(data.encode()).hexdigest()
        if hash_value.startswith('0' * difficulty):
            print(f"Found nonce: {nonce}")
            print(f"Hash: {hash_value}")
            return hash_value, nonce
        nonce += 1

if __name__ == "__main__":
    block_header = "Block #587234"  # Example block header
    difficulty = 4  # Number of leading zeros required
    start_time = time.time()
    mined_hash, mined_nonce = mine_block(block_header, difficulty)
    end_time = time.time()
    print(f"Time taken: {end_time - start_time:.2f} seconds")
```

Input: Run the Python code (or equivalent). Adjust the block_header and difficulty.

Expected Output:

A simulated "mined" block (a hash that meets the difficulty requirement).

The nonce that was found.

The time taken to mine the block.

Lab 6: Cryptographic Hash Functions for Password Verification

Title: Implementation of Cryptographic Hash Functions for Password Verification

Aim: To implement and understand the use of cryptographic hash functions for password verification.

Procedure:

1. Research cryptographic hash functions (e.g., SHA-256, bcrypt, scrypt).
2. Implement a function to hash a password using a chosen algorithm (e.g., in Python using the hashlib or bcrypt libraries).
3. Implement a function to verify a password against a stored hash.
4. Demonstrate the process of:

   Hashing a password.

   Storing the hash.

   Verifying a user-provided password against the stored hash.

Source Code:

```
# Example Python code
import hashlib
import bcrypt  # You might need to install this: pip install bcrypt

def hash_password_sha256(password):
    """Hashes a password using SHA-256 (for demonstration - use bcrypt in
practice)."""
    hashed_password = hashlib.sha256(password.encode()).hexdigest()
    return hashed_password

def hash_password_bcrypt(password):
    """Hashes a password using bcrypt."""
    hashed_password = bcrypt.hashpw(password.encode('utf-8'),
bcrypt.gensalt())
    return hashed_password.decode('utf-8')

def verify_password_bcrypt(password, stored_hash):
    """Verifies a password against a bcrypt hash."""
    return bcrypt.checkpw(password.encode('utf-8'), stored_hash.encode('utf-
8'))

if __name__ == "__main__":
    password = "mysecretpassword"

    # SHA-256 (for demonstration - NOT recommended for real passwords)
    hashed_password_sha256 = hash_password_sha256(password)
    print(f"SHA-256 Hash: {hashed_password_sha256}")

    # bcrypt (Recommended for real passwords)
    hashed_password_bcrypt = hash_password_bcrypt(password)
    print(f"bcrypt Hash: {hashed_password_bcrypt}")

    # Verify with bcrypt
    if verify_password_bcrypt(password, hashed_password_bcrypt):
        print("Password verified successfully (bcrypt)")
    else:
```

```
        print("Password verification failed (bcrypt)")

    # Verify with incorrect password
    if verify_password_bcrypt("wrongpassword", hashed_password_bcrypt):
        print("Password verified successfully (bcrypt) - SHOULD NOT HAPPEN")
    else:
        print("Password verification failed (bcrypt) - Correctly failed")
```

Input: Run the Python code. Provide a password to hash and then verify it.

Expected Output:

The hash of the password.

Confirmation of successful password verification.

Lab 7: Building a Distributed Peer-to-Peer Network

Title: Building a Distributed Peer-to-Peer Network

Aim: To build a distributed peer-to-peer (P2P) network.

Procedure:

1. Design the architecture of the P2P network (e.g., using a specific protocol or framework).
2. Implement node discovery (how nodes find each other).
3. Implement message routing (how messages are sent between nodes).
4. Implement data sharing or synchronization.
5. Test the network with multiple nodes running on different machines or virtual machines.

Source Code:

```
# Example Python code (Conceptual - this is a complex lab, and a full
implementation is extensive)
import socket
import threading
import json
import time

class Node:
    def __init__(self, host, port, initial_peers=None):
        self.host = host
        self.port = port
        self.peers = set()
        if initial_peers:
            self.peers.update(initial_peers)
        self.server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.server_socket.bind((self.host, self.port))
        self.server_socket.listen()
        self.running = True
        self.message_queue = [] # Store received messages

    def start(self):
        """Starts the node's server and connection handling."""
        threading.Thread(target=self.listen_for_connections).start()
        threading.Thread(target=self.process_messages).start()
        print(f"Node started on {self.host}:{self.port}")

    def stop(self):
        """Stops the node."""
        self.running = False
        self.server_socket.close()
        print(f"Node stopped on {self.host}:{self.port}")

    def connect_to_peer(self, peer_host, peer_port):
        """Connects to another peer."""
        try:
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect((peer_host, peer_port))
            self.send_message(s, {'type': 'connect', 'host': self.host,
'port': self.port})
            self.peers.add((peer_host, peer_port))
```

```
            threading.Thread(target=self.handle_connection, args=(s,
(peer_host, peer_port))).start()
            print(f"Connected to peer {peer_host}:{peer_port}")
            return True
        except Exception as e:
            print(f"Failed to connect to peer {peer_host}:{peer_port}: {e}")
            return False

    def listen_for_connections(self):
        """Listens for incoming connections from other nodes."""
        while self.running:
            try:
                conn, addr = self.server_socket.accept()
                threading.Thread(target=self.handle_connection, args=(conn,
addr)).start()
            except socket.error:
                if not self.running:
                    break # Socket was closed
                else:
                    raise

    def handle_connection(self, conn, addr):
        """Handles communication with a connected node."""
        print(f"Connection from {addr}")
        while self.running:
            try:
                data = conn.recv(4096)
                if not data:
                    break
                message = json.loads(data.decode())
                self.message_queue.append((message, addr)) # Add message to
queue
            except (socket.error, json.JSONDecodeError) as e:
                print(f"Error handling connection from {addr}: {e}")
                break
        conn.close()
        if addr in self.peers:
          self.peers.remove(addr)
        print(f"Connection with {addr} closed")

    def send_message(self, sock, message):
        """Sends a message to a connected socket."""
        try:
            sock.sendall(json.dumps(message).encode())
        except socket.error as e:
            print(f"Error sending message: {e}")

    def broadcast_message(self, message):
        """Broadcasts a message to all connected peers."""
        for peer_host, peer_port in self.peers:
            try:
              s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
              s.connect((peer_host, peer_port))
              self.send_message(s, message)
              s.close()
            except socket.error as e:
                print(f"Error broadcasting to {peer_host}:{peer_port} - {e}")

    def process_messages(self):
        """Processes messages from the message queue."""
        while self.running:
            if self.message_queue:
                message, sender = self.message_queue.pop(0)
                self.handle_message(message, sender)
            else:
                time.sleep(0.1)  # Sleep briefly to avoid busy-waiting
```

```
    def handle_message(self, message, sender):
        """Handles different types of messages."""
        if message['type'] == 'connect':
            peer_host = message['host']
            peer_port = message['port']
            if (peer_host, peer_port) != (self.host, self.port) and
(peer_host, peer_port) not in self.peers:
                self.peers.add((peer_host, peer_port))
                print(f"Peer {peer_host}:{peer_port} connected")
                # Send our peers to the newly connected peer
                self.send_message_to_peer((peer_host, peer_port), {'type':
'peers', 'peers': list(self.peers)})
        elif message['type'] == 'peers':
            new_peers = message['peers']
            for peer_host, peer_port in new_peers:
              if (peer_host, peer_port) != (self.host, self.port) and
(peer_host, peer_port) not in self.peers:
                self.connect_to_peer(peer_host, peer_port)
        elif message['type'] == 'text':
            print(f"Received text message from {sender}: {message['text']}")
            self.broadcast_message({'type': 'text', 'text': message['text']})
# Relay message
        else:
            print(f"Received unknown message type from {sender}: {message}")

    def send_message_to_peer(self, peer, message):
      """ Send message to a specific peer"""
      try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(peer)
        self.send_message(s, message)
        s.close()
      except socket.error as e:
        print(f"Error sending message to peer {peer}: {e}")

if __name__ == "__main__":
    # Example usage:
    node1 = Node('127.0.0.1', 12345)
    node2 = Node('127.0.0.1', 12346, initial_peers=[('127.0.0.1', 12345)])  #
node2 connects to node1
    node1.start()
    node2.start()

    time.sleep(2) # Wait for connections to establish

    # Send a message from node1 to the network
    node1.broadcast_message({'type': 'text', 'text': 'Hello from Node 1!'})

    time.sleep(5)  # Let the network run for a bit
    node1.stop()
    node2.stop()
```

Input: Run the Python code (or equivalent). Start multiple nodes, connect them, and send messages.

Expected Output:

Nodes connecting to form a network.

Messages being routed between nodes.

Data sharing or synchronization between nodes.

Lab 8: Consensus Mechanism Simulation

Title: Consensus Mechanism Simulation

Aim: To simulate a consensus mechanism used in blockchain.

Procedure:

1. Research different consensus mechanisms (e.g., Proof-of-Work, Proof-of-Stake, Delegated Proof-of-Stake, Raft).
2. Choose one consensus mechanism to simulate.
3. Develop a simulation of the chosen mechanism (e.g., in Python). This might involve:

   Simulating nodes in the network.

   Simulating the process of proposing and validating blocks.

   Simulating the consensus process (e.g., voting, leader election).

4. Analyze the behavior of the simulation under different conditions (e.g., node failures, network latency).

Source Code:

```
# Example Python code (Conceptual - Proof-of-Stake simulation)
import random
import time

class Node:
    def __init__(self, node_id, stake):
        self.node_id = node_id
        self.stake = stake
        self.proposed_block = None
        self.voted_for = None

    def propose_block(self, block_data):
        """Proposes a new block."""
        self.proposed_block = {'node_id': self.node_id, 'data': block_data,
'timestamp': time.time()}
        return self.proposed_block

    def vote_for_block(self, block):
        """Votes for a block."""
        # Simplified voting logic:  Higher stake = more influence
        if self.stake > 10:  # Example threshold
            self.voted_for = block
            return True
        else:
            return False

def simulate_pos(nodes, block_data):
    """Simulates a simplified Proof-of-Stake consensus."""
    print("Starting Proof-of-Stake Simulation")

    # 1. Block Proposal
    proposer = random.choices(nodes, weights=[node.stake for node in nodes],
k=1)[0] # Select proposer
    proposed_block = proposer.propose_block(block_data)
    print(f"Node {proposer.node_id} proposed block: {proposed_block}")
```

```
    # 2. Voting
    votes = 0
    for node in nodes:
        if node.vote_for_block(proposed_block):
            votes += node.stake
            print(f"Node {node.node_id} voted for the block")

    # 3. Block Validation
    total_stake = sum(node.stake for node in nodes)
    if votes > total_stake / 2: # Simple majority
        print("Block validated!")
        return proposed_block
    else:
        print("Block validation failed.")
        return None

if __name__ == "__main__":
    # Create some nodes with different stakes
    nodes = [
        Node(1, 15),
        Node(2, 5),
        Node(3, 20),
        Node(4, 10),
        Node(5, 30),
        Node(6, 20)
    ]
    block_data = "Transaction Data"
    simulate_pos(nodes, block_data)
```

Input: Run the Python code (or equivalent). Configure the nodes and their stakes.

Expected Output:

Simulation of the chosen consensus mechanism.

Output showing the process of block proposal, voting, and validation.

Analysis of the simulation's behavior.

Lab 9: Blockchain Network Creation with Application

Title: Blockchain Network Creation with Application

Aim: To create a simple blockchain network and deploy a basic application on it.

Procedure:

1. Choose a blockchain platform (e.g., Ethereum, Hyperledger Fabric).
2. Set up a local blockchain network using the chosen platform. This might involve:

   Installing the platform's tools.

   Configuring nodes.

   Creating a genesis block.

3. Develop a simple application (e.g., a basic smart contract on Ethereum, or a chaincode on Hyperledger Fabric).
4. Deploy the application to the blockchain network.
5. Interact with the application (e.g., call functions in the smart contract).

Source Code: (This lab involves both setup and application development. The "source code" includes:

Configuration files for the blockchain network.

The source code of the application (e.g., Solidity smart contract).

Input: Interaction with the deployed application on the blockchain network.

Expected Output:

A running local blockchain network.

A deployed application on the network.

Successful interaction with the application.

Lab 10: Ethereum Network Setup

Title: Ethereum Network Setup

Aim: To set up a local Ethereum network.

Procedure:

1. Install Ethereum development tools (e.g., Ganache, geth).
2. Set up a local development network (e.g., using Ganache or a private geth network).
3. Configure accounts and connect to the network.
4. Deploy a simple smart contract to the network (optional, but recommended).

Source Code: (This lab involves setting up a development environment. The "source code" includes:

Configuration settings for Ganache or geth.

(Optional) Source code of a simple Solidity smart contract.

Input: Commands to set up and run the Ethereum network.

Expected Output:

A running local Ethereum network.

Accounts created on the network.

(Optional) A deployed smart contract.

Lab 11: Solidity Smart Contract Development

Title: Solidity Smart Contract Development

Aim: To develop smart contracts using the Solidity programming language.

Procedure:

1. Learn the basics of the Solidity language.
2. Write several simple smart contracts (e.g., a simple storage contract, a token contract).
3. Compile the smart contracts.
4. Deploy the smart contracts to a local Ethereum network (e.g., using Remix or Truffle).
5. Interact with the deployed smart contracts (e.g., call functions).

Source Code:

```
// Example Solidity code (Simple Storage Contract)
pragma solidity ^0.8.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Input: Solidity code, commands to compile and deploy the contract, and function calls to the deployed contract.

Expected Output:

Compiled Solidity smart contracts.

Deployed smart contracts on a local Ethereum network.

Successful interaction with the smart contract functions.

Lab 12: Explore Decentralized Applications (DApps)

Title: Explore Decentralized Applications (DApps)

Aim: To explore existing decentralized applications (DApps).

Procedure:

1. Research different types of DApps (e.g., DeFi, NFTs, gaming, social media).
2. Choose one DApp to explore in detail.
3. Use the chosen DApp (if possible).
4. Analyze the DApp's architecture, functionality, and benefits/drawbacks.
5. Write a report summarizing your findings.

Source Code: (This is an exploration lab. The "source code" is the analysis of the DApp's architecture, which may involve reviewing publicly available information, not writing code.)

Input: Research andinteraction with a chosen DApp.

Expected Output:

A report analyzing the architecture, functionality, and pros/cons of a DApp.

Lab 13: Understanding Zcash

Title: Understanding Zcash

Aim: To understand the privacy-focused cryptocurrency Zcash.

Procedure:

1. Research the features of Zcash, focusing on its privacy-enhancing technologies (e.g., zk-SNARKs).
2. Compare Zcash with other cryptocurrencies (e.g., Bitcoin, Ethereum) in terms of privacy.
3. Explore the use cases and limitations of Zcash.
4. Write a report summarizing your understanding of Zcash.

Source Code: (This is a research lab. There is no source code.)

Input: Research on Zcash.

Expected Output:

A report summarizing the features, comparisons, use cases, and limitations of Zcash.

Lab 14: Case Study about Different Attacks

Title: Case Study about Different Attacks

Aim: To study different types of attacks on blockchain technology.

Procedure:

1. Research various attacks on blockchain (e.g., 51% attack, double-spending attack, Sybil attack, smart contract vulnerabilities).
2. Choose a few specific attacks to study in detail.
3. For each chosen attack, analyze:

    How the attack works.

    The potential impact of the attack.

    Methods to prevent or mitigate the attack.

4. Present your findings as case studies.

Source Code: (This is a research lab. The "source code" is the detailed description and analysis of the attacks.)

Input: Research on blockchain attacks.

Expected Output:

Case studies describing different types of attacks on blockchain technology.

Lab 15: Simple Application using Web3

Title: Simple Application using Web3

Aim: To build a simple application that interacts with a blockchain using a Web3 library.

Procedure:

1. Choose a Web3 library (e.g., web3.js, web3.py).
2. Set up a development environment with the chosen library.
3. Connect to a local or testnet blockchain (e.g., Ganache, Sepolia).
4. Write a simple application that interacts with a smart contract (e.g., reading data from a contract, sending a transaction).

Source Code:

```
// Example JavaScript code (web3.js)
const Web3 = require('web3');

// Replace with your Infura project ID or local provider URL
const web3 = new Web3('http://127.0.0.1:7545'); // Example: Ganache

// Replace with the ABI of your deployed smart contract
const contractABI = [
  {
    "inputs": [
      {
        "internalType": "uint256",
        "name": "x",
        "type": "uint256"
      }
    ],
    "name": "set",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
  },
  {
    "inputs": [],
    "name": "get",
    "outputs": [
      {
        "internalType": "uint256",
        "name": "",
        "type": "uint256"
      }
    ],
    "stateMutability": "view",
    "type": "function"
  }
];

// Replace with the address of your deployed smart contract
const contractAddress = '0xYourContractAddress'; // Replace this

async function main() {
  try {
    // Get accounts
    const accounts = await web3.eth.getAccounts();
    const defaultAccount = accounts[0];
```

```
    // Create a contract instance
    const simpleStorageContract = new web3.eth.Contract(contractABI,
contractAddress, { from: defaultAccount });

    // Call the set function
    console.log('Calling set function...');
    const tx = await simpleStorageContract.methods.set(123).send({ from:
defaultAccount });
    console.log('Transaction hash:', tx.transactionHash);

    // Call the get function
    console.log('Calling get function...');
    const storedData = await simpleStorageContract.methods.get().call();
    console.log('Stored data:', storedData.toString());
  } catch (error) {
    console.error('Error:', error);
  }
}

main();
```

Input: JavaScript code (or Python code with web3.py), contract ABI, contract address, and commands to run the application.

Expected Output:

Connection to a blockchain.

Interaction with a smart contract (e.g., setting and retrieving data).

Output of the data retrieved from the blockchain.