**Bct IA 2**

**16010121017 - Viraj Bhansali**

**16010121033 - Nupur Chaudhari**

**A\_Review\_on\_Double\_Spending\_Problem\_in\_Blockchain**

### <https://ieeexplore.ieee.org/document/10183579>

### 

### **Introduction**

The double-spending problem is a critical issue in blockchain technology and digital currency systems. It occurs when a malicious user attempts to spend the same digital asset more than once, compromising the integrity of the transaction system. In a decentralized environment, such as blockchain, ensuring that digital transactions are accurately recorded and not duplicated is challenging due to the lack of a central authority. Double-spending poses significant threats, including financial losses, network instability, and diminished trust in the system. Various mechanisms, like consensus algorithms, are employed to address this issue and ensure transaction validity.

### **Problem Definition**

Double-spending occurs when a digital currency or token is spent multiple times due to vulnerabilities in the transaction verification process. In traditional banking, central authorities prevent such problems by maintaining a ledger. However, blockchain networks, which operate in a decentralized manner, require consensus mechanisms to validate transactions. Without effective consensus, the risk of double-spending increases, potentially allowing users to manipulate the system and causing economic damage.

### **Potential Solutions**

Several consensus mechanisms have been proposed to address the double-spending problem in blockchain networks. Two widely recognized solutions are Practical Byzantine Fault Tolerance (PBFT) and Proof of Work (PoW). These mechanisms help maintain network consensus and prevent malicious activities like double-spending by validating transactions and creating a secure ledger.

#### 1. Practical Byzantine Fault Tolerance (PBFT)

PBFT is a consensus mechanism designed to tolerate Byzantine faults, where some nodes in the network may act maliciously or fail. In the PBFT protocol, transactions are verified through a process of voting, where nodes reach consensus on the validity of a transaction before adding it to the blockchain. The network can tolerate a certain number of faulty nodes (up to one-third of the total), which ensures robustness against double-spending attacks. PBFT provides low-latency transaction confirmation, making it suitable for permissioned blockchain networks.

#### 2. Proof of Work (PoW)

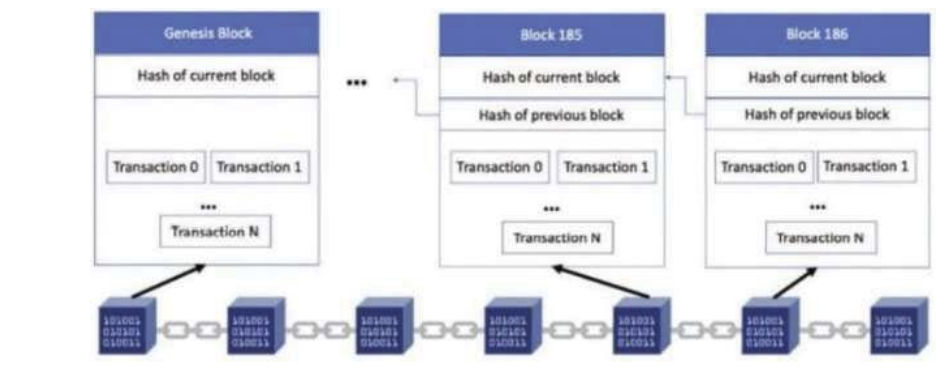
PoW is a consensus algorithm that requires nodes (miners) to solve complex mathematical puzzles to validate transactions and add new blocks to the blockchain. The process of solving these puzzles consumes significant computational resources, making it difficult for attackers to control the majority of the network's hashing power. By requiring a high amount of computational effort, PoW makes it economically infeasible for an attacker to double-spend since the cost of manipulating the blockchain outweighs the potential rewards.

**Literature Study**

Existing research paper points to several factors contributing to this issue, including network latency, malicious actors, and the design of consensus mechanisms like Proof of Work (PoW), Proof of Stake (PoS), and Practical Byzantine Fault Tolerance (PBFT).

While PoW and PoS are widely used, they are not foolproof. PBFT, on the other hand, is designed to address the problem in a different way. It works by having multiple nodes (or computers) agree on the validity of transactions, requiring a majority vote to confirm any transaction. This helps prevent double spending by ensuring that even if some nodes are compromised, the network can still function securely. However, PBFT is more communication-heavy, making it less efficient for larger networks. Emerging technologies such as zero-knowledge proofs and multi-party computation are also being explored to further strengthen the security of blockchain systems​.

The literature also proposes a solution where each pairs of nodes has a network observer placed between them.



* **Transaction Monitoring**
  + Whenever someone sends a cryptocurrency (like Bitcoin), network observers watch the transaction.
* **Tracking Details**
  + The observers keep track of important details like the sender, the amount, and the number of transactions.
* **Detecting Unusual Behavior**
  + If a person tries to send the same cryptocurrency to two different people at the same time, the observers notice this unusual activity.
* **Canceling the Fraudulent Transaction**
  + When suspicious behavior is detected, the observer cancels the second transaction to stop double-spending.
* **Sending a Warning**
  + The observer notifies the original sender that the second transaction was rejected because it seemed like an attempt to spend the same cryptocurrency twice.
  + When a network observer detects an irregularity in a transaction between two nodes, nearby nodes are alerted to the fraudulent activity. This helps them reroute transactions through secure paths and cut off connections with the suspicious node pair.

### **Implementation Techniques/Methodologies and Results**

**Longest chain method**

import hashlib

import time

import random

class Block:

def \_\_init\_\_(self, index, previous\_hash, timestamp, data, hash, nonce=0):

self.index = index

self.previous\_hash = previous\_hash

self.timestamp = timestamp

self.data = data

self.hash = hash

self.nonce = nonce

def \_\_repr\_\_(self):

return f"Block(index={self.index}, previous\_hash='{self.previous\_hash}', timestamp={self.timestamp}, data='{self.data}', hash='{self.hash}', nonce={self.nonce})"

class Blockchain:

def \_\_init\_\_(self):

self.chain = []

self.difficulty = 2 # Adjust difficulty for PoW

self.create\_genesis\_block() # Create the genesis block

def create\_genesis\_block(self):

genesis\_block = Block(

index=1,

previous\_hash='0',

timestamp=int(time.time()),

data="Genesis Block",

hash='0'\*64,

nonce=0

)

self.chain.append(genesis\_block)

def create\_block(self, nonce, previous\_hash, data="Block Data"):

block = Block(

index=len(self.chain) + 1,

previous\_hash=previous\_hash,

timestamp=int(time.time()),

data=data,

hash=self.hash\_block(nonce, previous\_hash, data),

nonce=nonce

)

self.chain.append(block)

return block

def hash\_block(self, nonce, previous\_hash, data):

block\_string = f"{nonce}{previous\_hash}{data}{int(time.time())}".encode()

return hashlib.sha256(block\_string).hexdigest()

def mine\_block(self, data="Block Data"):

previous\_block = self.chain[-1]

previous\_hash = previous\_block.hash

nonce = 0

while True:

hash = self.hash\_block(nonce, previous\_hash, data)

if hash[:self.difficulty] == '0' \* self.difficulty: # Check if hash meets difficulty

break

nonce += 1

new\_block = self.create\_block(nonce, previous\_hash, data)

print(f"Block mined: {new\_block}")

return new\_block

def validate\_chain(self):

for i in range(1, len(self.chain)):

current = self.chain[i]

previous = self.chain[i - 1]

if current.previous\_hash != previous.hash:

return False

if current.hash[:self.difficulty] != '0' \* self.difficulty:

return False

return True

def user\_driven\_simulation():

blockchain = Blockchain()

forks = [] # To keep track of separate chains (forks)

while True:

print("\nCurrent Blockchain:")

for block in blockchain.chain:

print(block)

print("\nMenu:")

print("1. Mine a new block")

print("2. Create a fork")

print("3. Extend a fork")

print("4. Resolve forks (Longest chain rule)")

print("5. Validate blockchain")

print("6. Exit")

choice = input("\nEnter your choice: ")

if choice == '1':

data = input("Enter block data: ")

blockchain.mine\_block(data=data)

elif choice == '2':

# Create a new fork from the current chain

fork = blockchain.chain[:]

forks.append(fork)

print(f"Fork {len(forks)} created.")

elif choice == '3':

# Allow the user to choose which fork to extend

if not forks:

print("No forks available to extend.")

continue

print(f"Available forks: 1 to {len(forks)}")

fork\_choice = int(input("Choose a fork to extend: ")) - 1

if 0 <= fork\_choice < len(forks):

fork = forks[fork\_choice]

data = input("Enter data for the new block: ")

nonce = random.randint(0, 1000) # Simulate mining by generating a random nonce

new\_block = Blockchain().create\_block(nonce, fork[-1].hash, data)

fork.append(new\_block)

print(f"Block added to Fork {fork\_choice + 1}.")

else:

print("Invalid fork choice.")

elif choice == '4':

# Apply the longest chain rule to resolve forks

if not forks:

print("No forks available for resolution.")

continue

max\_length = len(blockchain.chain)

selected\_fork = blockchain.chain

for fork in forks:

if len(fork) > max\_length:

max\_length = len(fork)

selected\_fork = fork

blockchain.chain = selected\_fork

forks = [] # Clear all forks after resolution

print("Forks resolved using the longest chain rule.")

elif choice == '5':

is\_valid = blockchain.validate\_chain()

print(f"The blockchain is {'valid' if is\_valid else 'invalid'}.")

elif choice == '6':

print("Exiting the simulation.")

break

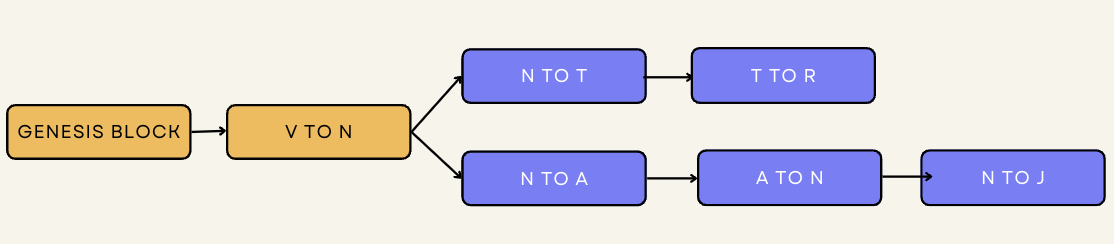
else:

print("Invalid choice. Please try again.")

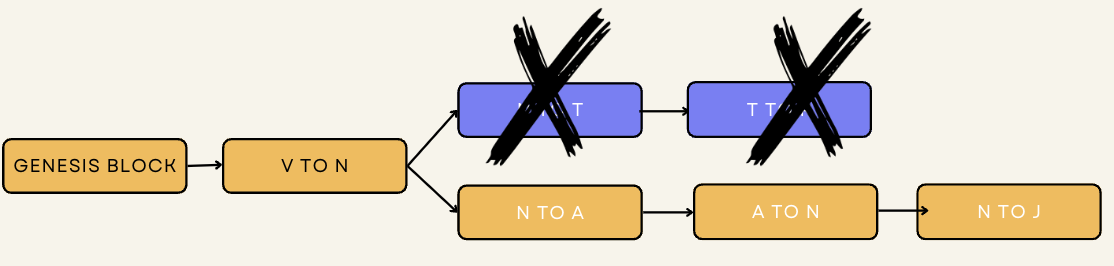
if \_\_name\_\_ == "\_\_main\_\_":

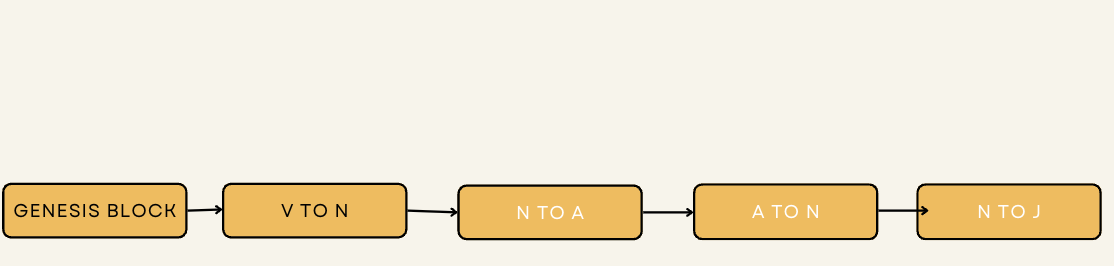
user\_driven\_simulation()

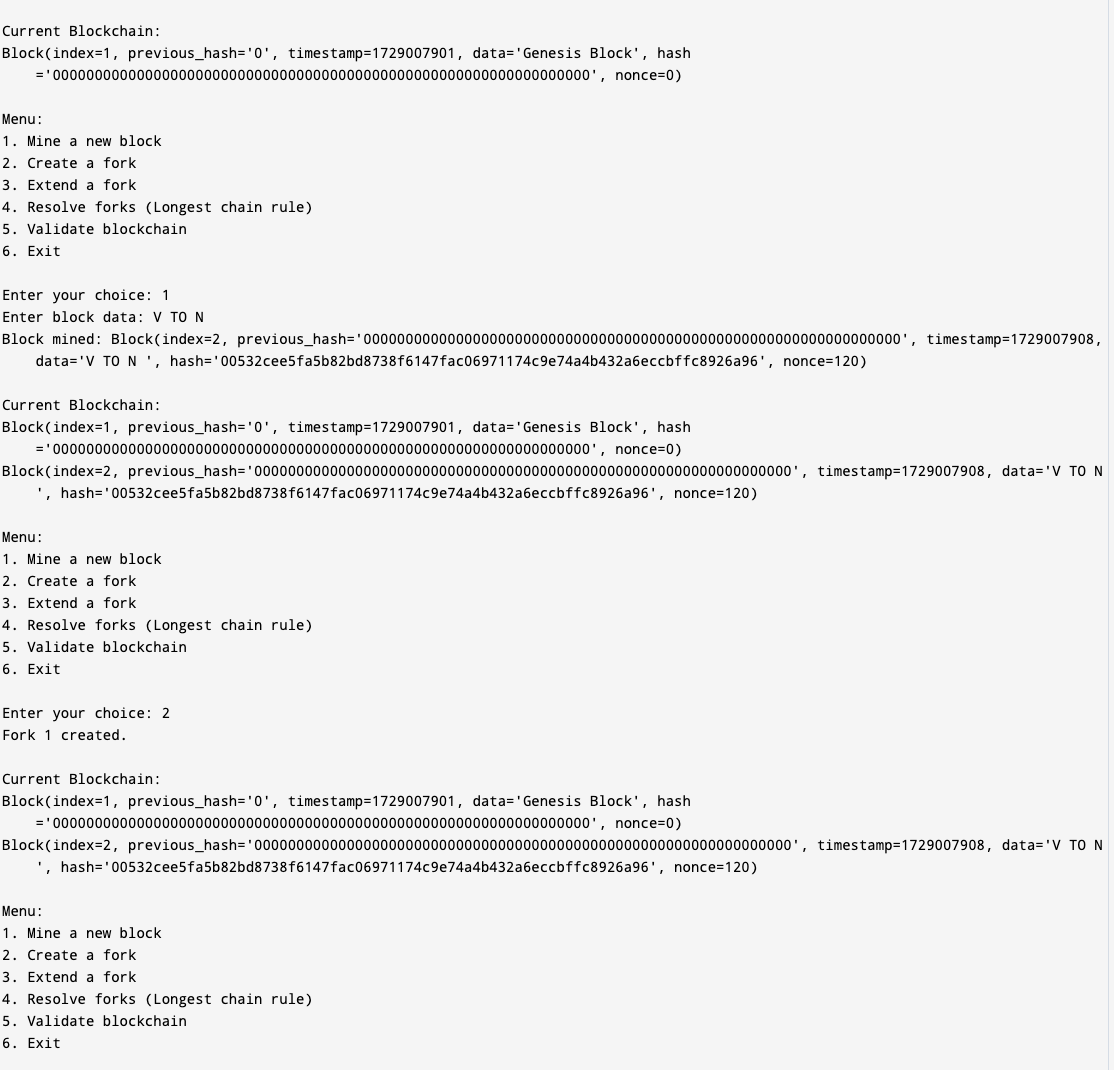
STIMULATION

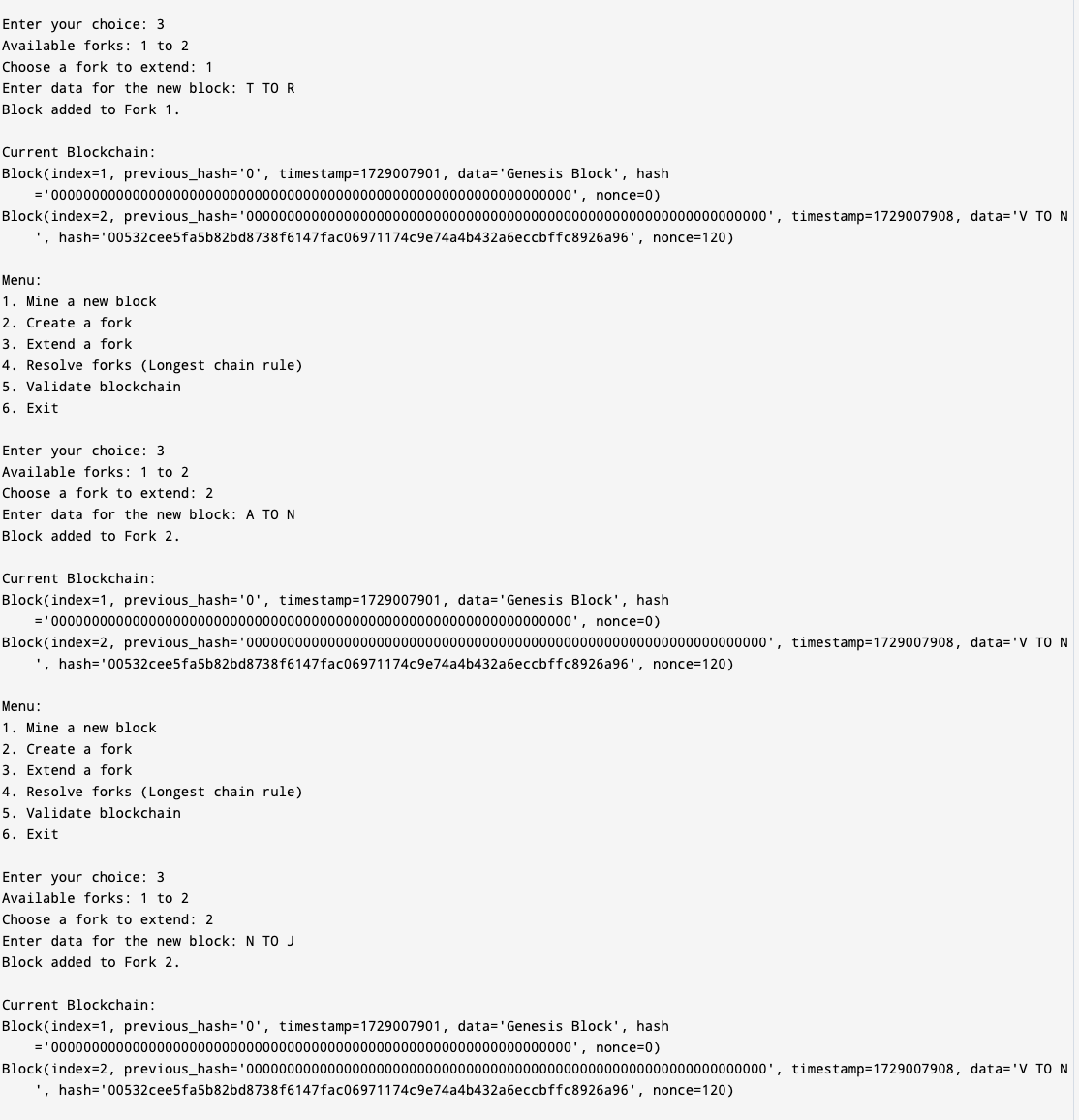


When the conflict is resolved   
Ie . the longest chain is selected as the new block chain   
And the other fork is discarded from the ledger of all nodes









**Check balance**

const crypto = require('crypto');

class Transaction {

constructor(fromAddress, toAddress, amount) {

this.fromAddress = fromAddress;

this.toAddress = toAddress;

this.amount = amount;

}

}

class Block {

constructor(timestamp, transactions, previousHash = '') {

this.timestamp = timestamp;

this.transactions = transactions;

this.previousHash = previousHash;

this.hash = this.calculateHash();

this.nonce = 0;

}

calculateHash() {

return crypto.createHash('sha256').update(

this.previousHash +

this.timestamp +

JSON.stringify(this.transactions) +

this.nonce

).digest('hex');

}

mineBlock(difficulty) {

while (this.hash.substring(0, difficulty) !== Array(difficulty + 1).join("0")) {

this.nonce++;

this.hash = this.calculateHash();

}

console.log(`Block mined: ${this.hash}`);

}

}

class Blockchain {

constructor() {

this.chain = [this.createGenesisBlock()];

this.difficulty = 2;

this.pendingTransactions = [];

this.miningReward = 100;

}

createGenesisBlock() {

return new Block(Date.now(), [], "0");

}

getLatestBlock() {

return this.chain[this.chain.length - 1];

}

minePendingTransactions(miningRewardAddress) {

const rewardTx = new Transaction(null, miningRewardAddress, this.miningReward);

this.pendingTransactions.push(rewardTx);

const block = new Block(Date.now(), this.pendingTransactions, this.getLatestBlock().hash);

block.mineBlock(this.difficulty);

console.log('Block successfully mined!');

this.chain.push(block);

this.pendingTransactions = [];

}

addTransaction(transaction) {

if (!transaction.fromAddress || !transaction.toAddress) {

throw new Error('Transaction must include from and to address');

}

if (transaction.amount <= 0) {

throw new Error('Transaction amount should be higher than 0');

}

const senderBalance = this.getBalanceOfAddress(transaction.fromAddress);

if (senderBalance < transaction.amount) {

throw new Error(`Not enough balance. Attempted to send ${transaction.amount}, but balance is ${senderBalance}`);

}

// Check if adding the transaction would exceed balance

const pendingAmount = this.pendingTransactions

.filter(tx => tx.fromAddress === transaction.fromAddress)

.reduce((acc, tx) => acc + tx.amount, 0);

if (pendingAmount + transaction.amount > senderBalance) {

throw new Error(`Double spending detected! Attempted to spend ${pendingAmount + transaction.amount}, but only ${senderBalance} is available.`);

}

this.pendingTransactions.push(transaction);

console.log(`Transaction added: ${JSON.stringify(transaction)}`);

}

getBalanceOfAddress(address) {

let balance = 0;

for (const block of this.chain) {

for (const trans of block.transactions) {

if (trans.fromAddress === address) {

balance -= trans.amount;

}

if (trans.toAddress === address) {

balance += trans.amount;

}

}

}

return balance;

}

isChainValid() {

for (let i = 1; i < this.chain.length; i++) {

const currentBlock = this.chain[i];

const previousBlock = this.chain[i - 1];

if (currentBlock.hash !== currentBlock.calculateHash()) {

return false;

}

if (currentBlock.previousHash !== previousBlock.hash) {

return false;

}

}

return true;

}

}

// Example usage

const myCoin = new Blockchain();

// Mine some initial rewards to address1

console.log('\nStarting the miner...');

myCoin.minePendingTransactions('address1'); // Rewarding address1 with 100 coins

console.log(`Balance of address1: ${myCoin.getBalanceOfAddress('address1')}`);

// Valid transaction: address1 sends 50 coins to address2

console.log('\nAttempting a valid transaction of 50 coins from address1 to address2...');

myCoin.addTransaction(new Transaction('address1', 'address2', 50));

console.log('Starting the miner to confirm the transaction...');

myCoin.minePendingTransactions('miner-address');

console.log(`Balance of address1: ${myCoin.getBalanceOfAddress('address1')}`);

console.log(`Balance of address2: ${myCoin.getBalanceOfAddress('address2')}`);

console.log(`Balance of miner: ${myCoin.getBalanceOfAddress('miner-address')}`);

// Attempt a double-spending attack: address1 tries to send 70 coins twice

console.log('\nAttempting a double-spending attack...');

try {

// Show balance before double-spending attempt

console.log(`Balance of address1 before double-spending attempt: ${myCoin.getBalanceOfAddress('address1')}`);

// First transaction of 70 coins, which should succeed

console.log('Attempting to send 70 coins from address1 to address3...');

myCoin.addTransaction(new Transaction('address1', 'address3', 70));

console.log('First transaction of 70 coins added.');

// Second transaction of 70 coins, which should fail due to insufficient balance

console.log('Attempting to send another 70 coins from address1 to address4...');

myCoin.addTransaction(new Transaction('address1', 'address4', 70));

console.log('Second transaction of 70 coins added.');

} catch (error) {

console.log('Error during transaction:', error.message);

}

console.log('\nBalances after attempting double spending:');

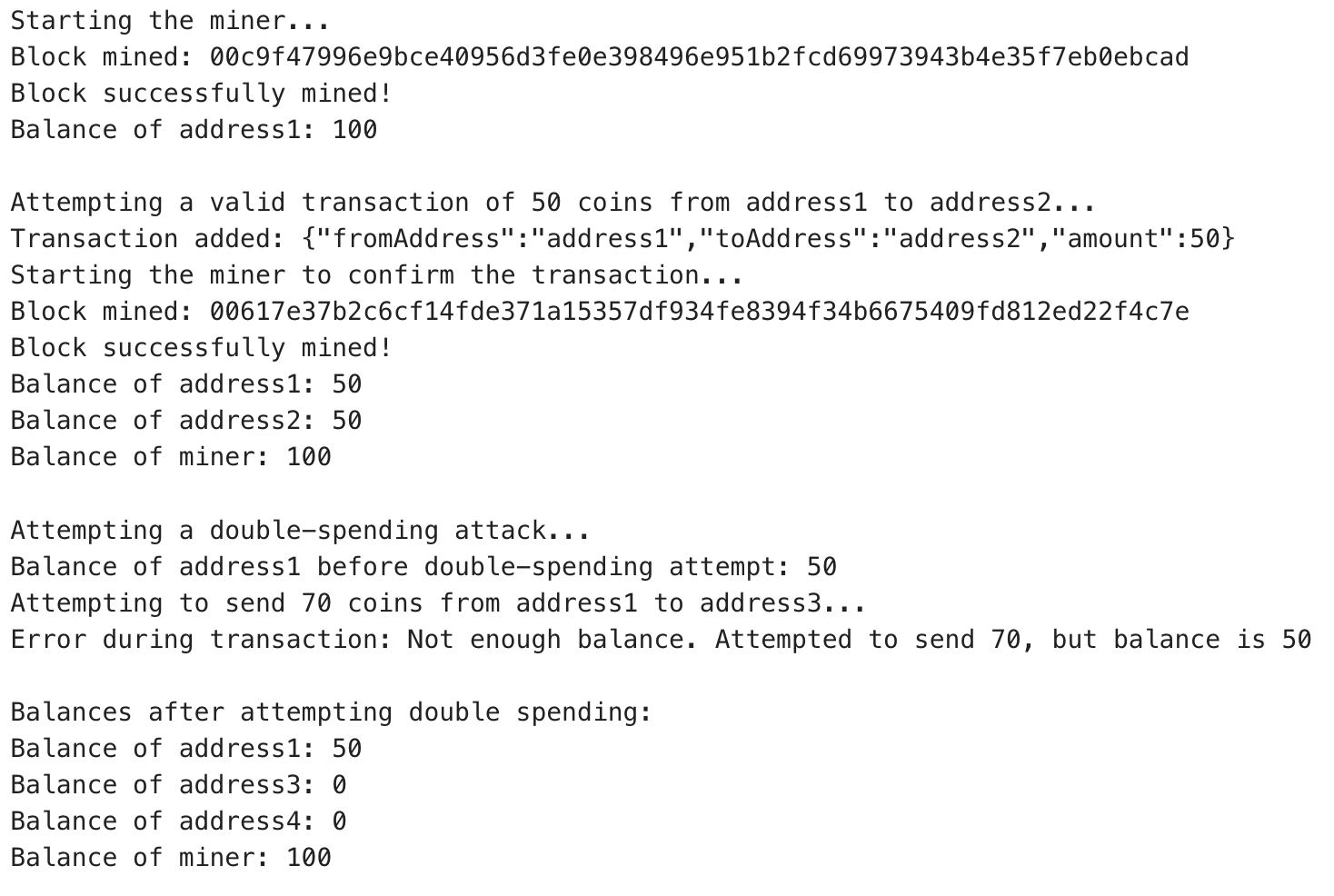
console.log(`Balance of address1: ${myCoin.getBalanceOfAddress('address1')}`);

console.log(`Balance of address3: ${myCoin.getBalanceOfAddress('address3')}`);

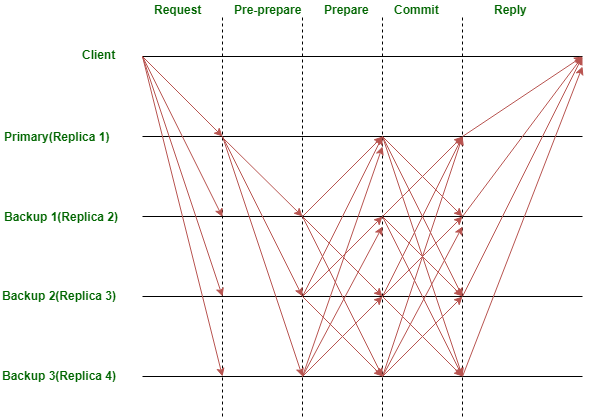
console.log(`Balance of address4: ${myCoin.getBalanceOfAddress('address4')}`);

console.log(`Balance of miner: ${myCoin.getBalanceOfAddress('miner-address')}`);

console.log(`Is blockchain valid? ${myCoin.isChainValid()}`);

Output   
  
  


PBFT solution steps



# **Example scenario :**

* **Transaction A**: Alice sends 1 BTC to Bob.
* **Transaction B**: Alice tries to send the same 1 BTC to Charlie (double spending).

1. **Pre-Prepare Phase:** **Leader Proposes Transaction A**: The leader node proposes the block with Transaction A (Alice to Bob) and broadcasts it to the network.
2. **Prepare Phase**: All nodes validate Transaction A, checking Alice’s account balance, and ensuring she has enough funds to send 1 BTC.
   * Each node sends a "prepare" message to all other nodes indicating that it has received and validated the proposal.
3. **Commit Phase**: Nodes reach a consensus and finalize the block with Transaction A.
   * Once a node receives a sufficient number (2/3 majority) of "prepare" messages, it moves to the commit phase.
   * Nodes broadcast "commit" messages, essentially saying, "I agree to add this block to the ledger."
   * Once a node receives a sufficient number of "commit" messages, it finalizes the block and adds it to its local blockchain.
4. **Double Spending Detection**: In PBFT, consensus requires agreement from 2/3 of the nodes.

* Once a block is added, it is considered final and irreversible.
  + If Alice later tries to submit Transaction B (sending the same BTC to Charlie), the nodes will reject it because they’ve already confirmed Transaction A. Alice’s 1 BTC is no longer available.

### **Conclusion**

Addressing the double-spending issue is crucial for the integrity of blockchain systems. Both PBFT and PoW offer viable solutions, with PBFT being more efficient in permissioned environments and PoW providing robust security in public blockchains. Future work may involve hybrid approaches that combine elements of both PBFT and PoW to optimize performance and security. The longest chain method ensures that the longest valid blockchain is accepted as the true version, making any shorter alternative chains less credible. Meanwhile, balance transaction checking verifies that a sender has sufficient funds before processing a transaction, rejecting any attempts to spend the same funds multiple times.

### **References**

### **A\_Review\_on\_Double\_Spending\_Problem\_in\_Blockchain**

<https://ieeexplore.ieee.org/document/10183579>