DataSys Coin (DSC) Blockchain

Sashank Lakshmana Bommadevara

Abdurakhmon Urazboev

Syed Alle Mustafa

Illinois Institute of Technology

12/05/2023

Abstract

Our system is a blockchain-based solution crafted for secure and transparent data sharing among multiple parties. Emphasizing data integrity, confidentiality, and availability, the architecture utilizes a tailored blockchain network implemented with Python, Flask for communication, and Blake3 for encryption. This network comprises diverse elements such as a blockchain server, metronome server, pool, validators, wallets, and a monitor. Evaluation focuses on criteria like performance, security, and scalability, aiming to offer insights into the practicality of blockchain technology in real-world scenarios. Driven by the transformative potential of blockchain, the project leverages the strengths of Python, Flask, and Blake3. The problem statement underscores the revolutionary impact of blockchain on transactions across industries. The proposed solution involves essential components like Wallet, Mining Pool Server, Metronome Server, Validator, Blockchain Server, and Monitor, each playing a unique role in establishing a decentralized and secure blockchain network. For example, the Validator manages proof algorithms such as Proof of Work, Proof of Space, and Proof of Memory. The Blockchain Server is tasked with creating and maintaining the blockchain ledger, while the Monitor provides blockchain statistics and oversees transactions and validator statistics. The report underscores rigorous testing and evaluation procedures to guarantee the scalability, reliability, and performance of the DataSys Coin blockchain project.

1 Introduction

This project aims to implement a blockchain based system for secure and transparent data sharing between multiple parties. The system will be designed to ensure data integrity, confidentiality, and availability while providing a decentralized and tamper-proof environment. The project will involve the development of a custom blockchain network using Python, Flask for Communication, and Blake3 for encryption. The network will consist of the following components: blockchain server, metronome server, pool, validators, wallets, and monitor. The blockchain server will be responsible for maintaining the blockchain ledger, while the metronome server will provide a time synchronization service. The pool will be used to store unconfirmed and submitted transactions, and the validator will be responsible for validating transactions based on different proof types, and adding them to the blockchain. The wallet will be used to store digital assets, and the monitor will provide real-time monitoring of the blockchain network. The project's success will be evaluated based on the performance, security, and scalability of the blockchain network. The project's outcome will provide insights into the feasibility of blockchain technology for real-world applications.

1.1 Motivation

The motivation behind DataSys Coin, our blockchain project, is the profound impact we believe blockchain technology can have on revolutionizing the way transactions operate. Our enthusiasm is rooted in the understanding that blockchain, with its decentralized and secure structure, holds immense potential for transforming transaction systems. The choice of Python, known for its versatility, and Flask, appreciated for its ease of communication, underscores our commitment to user-friendly and adaptable solutions. The robust cryptography features of Blake3 further contribute to ensuring the security of our data and key information. Rigorous testing across various environments, using diverse validation methods, is an essential part of our endeavor to guarantee the scalability and reliability of DataSys Coin. This motivation originates from our dedication to crafting a dependable and impactful blockchain solution, outlined in this comprehensive project report.

2 Problem Statement

The motivation behind the implementation of the DataSys Coin (DSC), this Blockchain project lies in the game-changing nature of blockchain technology. Recognized for its revolutionary impact, blockchain offers a decentralized, secure, and transparent system for recording and verifying transactions, eliminating the need for intermediaries such as banks or governments. This technology has the potential to reshape various industries, including finance, supply chain management, and healthcare, parallel to the transformative influence of the Internet, mobile computing, and artificial intelligence. The project aims to contribute practically to the understanding and application of blockchain technology through hands-on implementation and thorough evaluation, specifically leveraging Python, Flask, and Blake3.

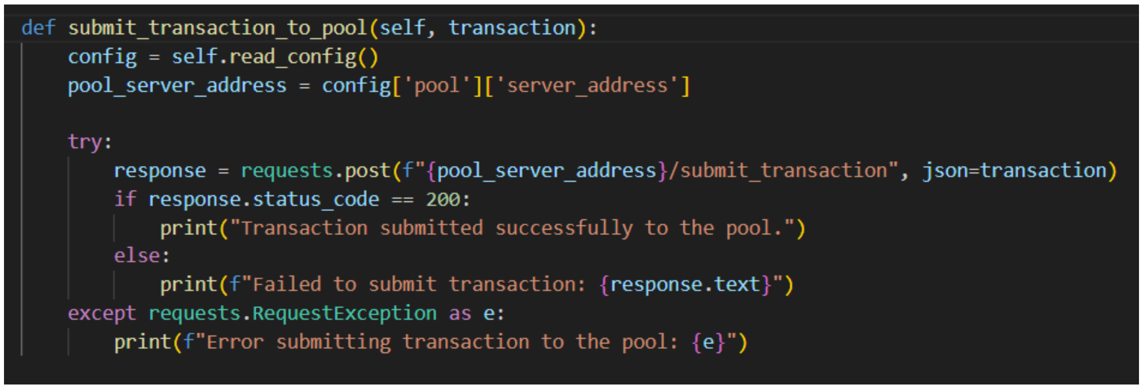
3 Proposed Solution

Our proposed solution comprises several integral components, each contributing to a well-defined control flow within the system. The details of these components are outlined below, providing a comprehensive understanding of the proposed solution's architecture and functionality.

3.1 Components

3.1.1 Wallet

The Wallet class embodies the functionalities of a digital currency wallet, providing a spectrum of operations accessible through a command-line interface. During initialization, the class employs the RSA algorithm to create and oversee key pairs, securing them via the Blake3 hash function. The list of operations encompasses wallet creation, key display, balance verification from the blockchain server, transaction initiation by sending it to the pool, and the tracking of transaction statuses through both pool and blockchain server. The wallet engages with both a blockchain server and a mining pool through targeted HTTP requests to designated endpoints and port number. Additionally, a testing mode was also incorporated for the simulation of transactions, facilitating the measurement and reporting of metrics such as latency and throughput, Which we have used for evaluating the performance of the network.



```python
def submit_transaction_to_pool(self, transaction):
    config = self.read_config()
    pool_server_address = config['pool']['server_address']

    try:
        response = requests.post(f"{pool_server_address}/submit_transaction", json=transaction)
        if response.status_code == 200:
            print("Transaction submitted successfully to the pool.")
        else:
            print(f"Failed to submit transaction: {response.text}")
    except requests.RequestException as e:
        print(f"Error submitting transaction to the pool: {e}")
```

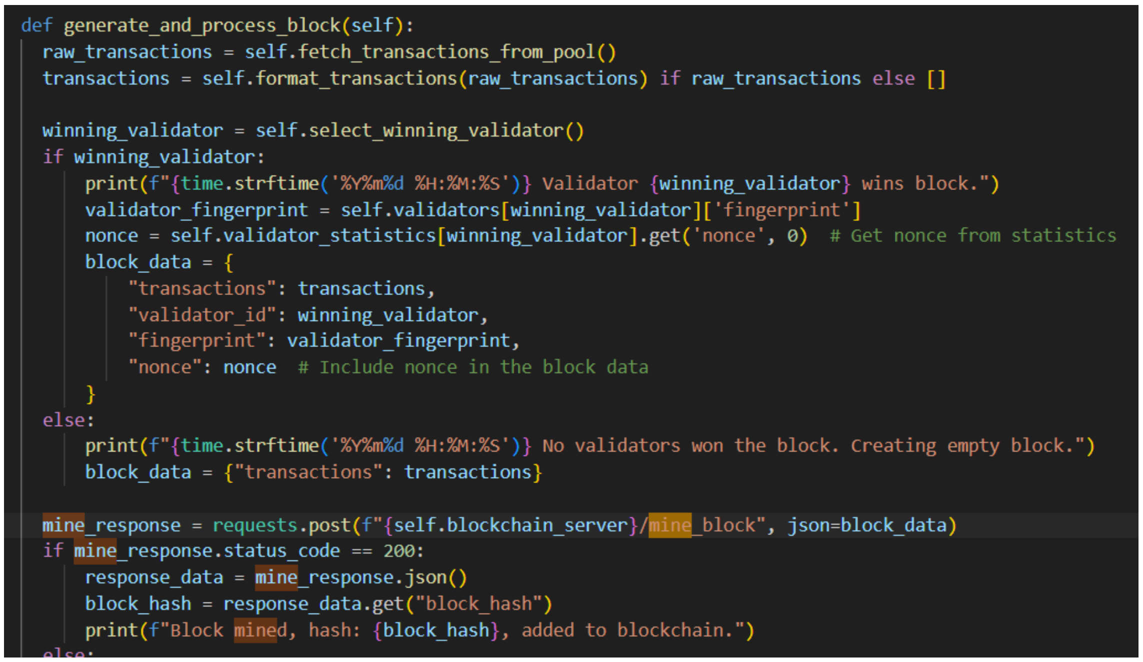
Figure 1: Submitting Transaction

3.1.2 Mining Pool Server

The mining pool server is a crucial component of the blockchain network, serving as a hub for processing transactions from wallet requests and providing them to the metronome server through a Flask application. The pooling server utilizes threading to handle multiple requests concurrently. It maintains two ordered dictionaries to manage submitted and unconfirmed transactions, ensuring secure transaction processing. The server starts by creating a Flask process, and wallets interact with it by submitting transactions, where the metronome puts it into unconfirmed, and then the blockchain server removes it from that list, so that the wallet can directly inquire about the status of it from the blockchain server. It also provides functionality for listing transactions and obtaining transaction counts. This mining pool server acts as a bridge between wallets, metronome and blockchain server, facilitating transaction management, monitoring, and network interaction.

3.1.3 Metronome Server

The Metronome server operates as a vital component within the blockchain server, orchestrating the block generation process at regular intervals. The metronome server employs Flask for communication and threading for concurrent execution. It interfaces with both the blockchain server and the mining pool server, ensuring synchronized block creation. The Metronome server runs a continuous loop, periodically generating and processing blocks by fetching transactions from the pool, formatting them, and selecting a winning validator based on provided information. List of validators contribute to the block generation process, and their statistics, including proof type, nonce, and hash rate, are updated and considered in the selection of the winning validator. The server also offers endpoints for starting and stopping the Metronome, registering validators, reporting statistics, obtaining validator statistics, and retrieving the current blockchain difficulty. Overall, the Metronome server plays a crucial role in maintaining the integrity and consistency of the blockchain network.



```python
def generate_and_process_block(self):
    raw_transactions = self.fetch_transactions_from_pool()
    transactions = self.format_transactions(raw_transactions) if raw_transactions else []

    winning_validator = self.select_winning_validator()
    if winning_validator:
        print(f"{time.strftime('%Y%m%d %H:%M:%S')} Validator {winning_validator} wins block.")
        validator_fingerprint = self.validators[winning_validator]['fingerprint']
        nonce = self.validator_statistics[winning_validator].get('nonce', 0)  # Get nonce from statistics
        block_data = {
            "transactions": transactions,
            "validator_id": winning_validator,
            "fingerprint": validator_fingerprint,
            "nonce": nonce  # Include nonce in the block data
        }
    else:
        print(f"{time.strftime('%Y%m%d %H:%M:%S')} No validators won the block. Creating empty block.")
        block_data = {"transactions": transactions}

    mine_response = requests.post(f"{self.blockchain_server}/mine_block", json=block_data)
    if mine_response.status_code == 200:
        response_data = mine_response.json()
        block_hash = response_data.get("block_hash")
        print(f"Block mined, hash: {block_hash}, added to blockchain.")
    else:
```

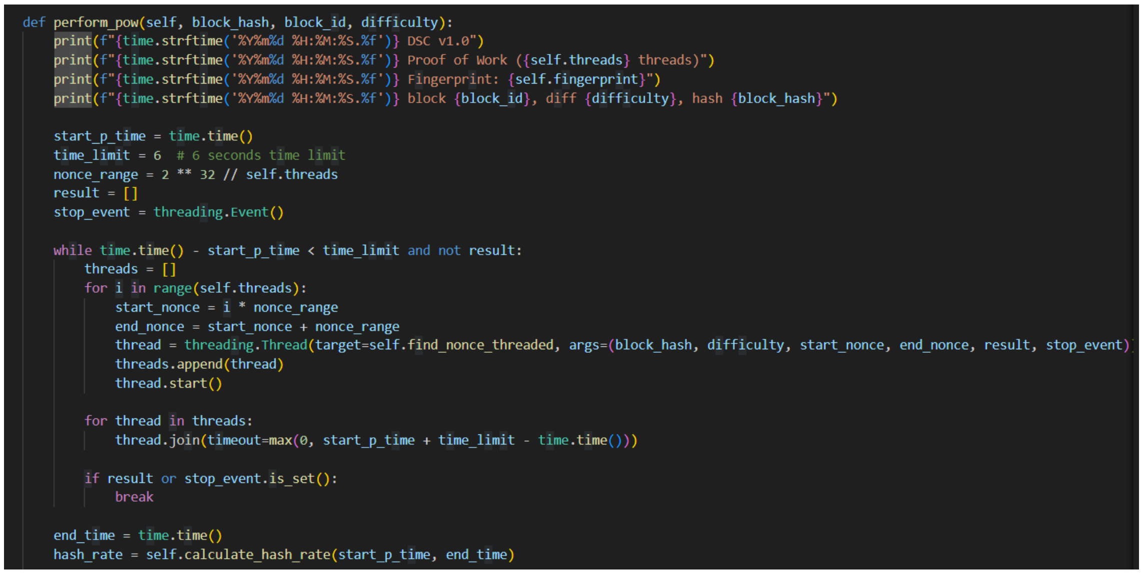
Figure 2: Creating Block

3.1.4 Validator

Validator, is the main component The validator serves as the central component in the blockchain network, playing a crucial role in validating the current block based on information derived from the latest block on the blockchain server. This pivotal function is accomplished by utilizing three distinct classes, each designed to execute specific types of validation. Upon initialization, the validator is assigned a unique ID, a fingerprint, and the addresses of the blockchain and metronome servers. Subsequently, it proceeds to register itself on the metronome server and initialize the proof classes, namely Proof of Work, Proof of Space, and Proof of Memory. Once initialized, the validator retrieves the latest block from the blockchain server, assesses the current difficulty level, and initiates the configured proof algorithm specified in the configuration file. If the validation process successfully identifies the required proof, the validator promptly notifies the metronome server of the achievement. This orchestrated sequence of actions underscores the validator's pivotal role in contributing to the secure and reliable functioning of the blockchain network. Following are the Proof classes that it uses to perform proofs:

Proof of Work: This class utilizes multiple threads to concurrently search for a nonce meeting a specified difficulty level within a time limit. The class features methods for hash calculation, threaded nonce discovery, and overall

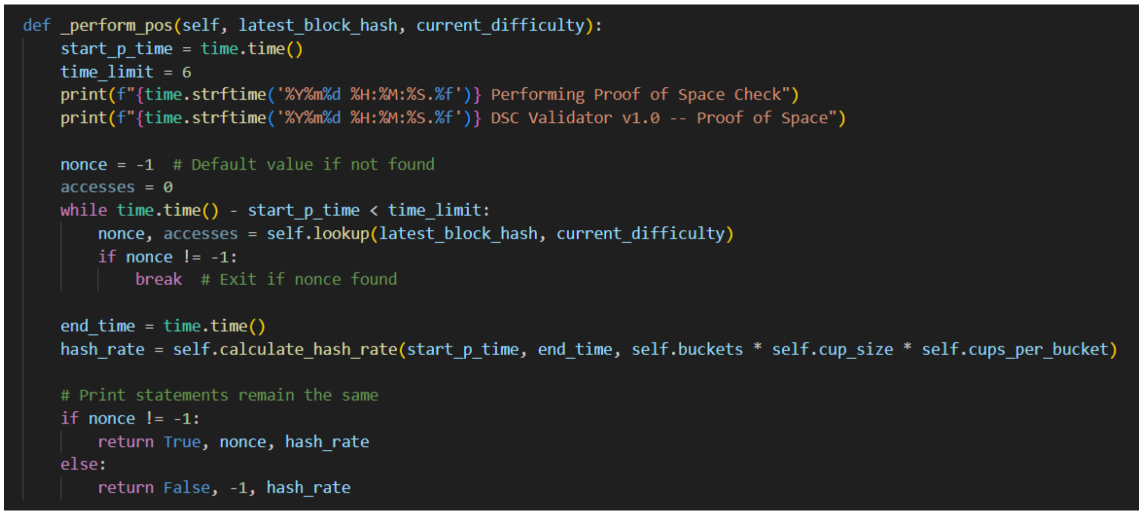
PoW execution.



```python
def perform_pow(self, block_hash, block_id, difficulty):
    print(f"{time.strftime('%Y%m%d %H:%M:%S.%f')} DSC v1.0")
    print(f"{time.strftime('%Y%m%d %H:%M:%S.%f')} Proof of Work ({self.threads} threads)")
    print(f"{time.strftime('%Y%m%d %H:%M:%S.%f')} Fingerprint: {self.fingerprint}")
    print(f"{time.strftime('%Y%m%d %H:%M:%S.%f')} block {block_id}, diff {difficulty}, hash {block_hash}")

    start_p_time = time.time()
    time_limit = 6  # 6 seconds time limit
    nonce_range = 2 ** 32 // self.threads
    result = []
    stop_event = threading.Event()

    while time.time() - start_p_time < time_limit and not result:
        threads = []
        for i in range(self.threads):
            start_nonce = i * nonce_range
            end_nonce = start_nonce + nonce_range
            thread = threading.Thread(target=self.find_nonce_threaded, args=(block_hash, difficulty, start_nonce, end_nonce, result, stop_event)
            threads.append(thread)
            thread.start()

        for thread in threads:
            thread.join(timeout=max(0, start_p_time + time_limit - time.time()))

        if result or stop_event.is_set():
            break

    end_time = time.time()
    hash_rate = self.calculate_hash_rate(start_p_time, end_time)
```

Figure 3: Proof of Work

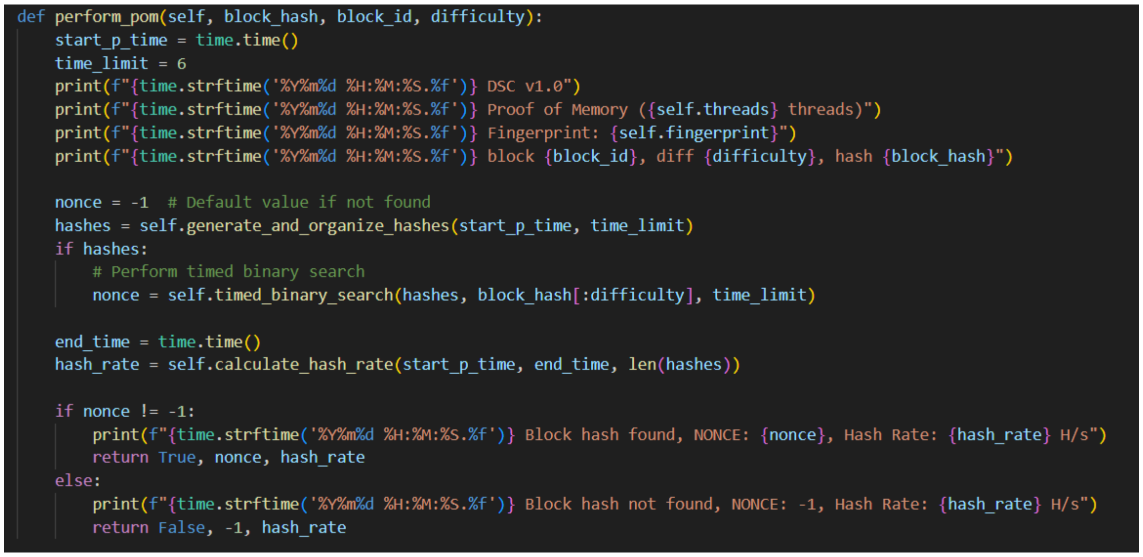
Proof of Space: This class efficiently manages vault creation, ensuring its existence, and saves configurations. Key functionalities of it include optimized hash lookup during the Proof of Space check. If the calculation of nonce is successful it returns true, with nonce and hash rate.



```python
def _perform_pos(self, latest_block_hash, current_difficulty):
    start_p_time = time.time()
    time_limit = 6
    print(f"{time.strftime('%Y%m%d %H:%M:%S.%f')} Performing Proof of Space Check")
    print(f"{time.strftime('%Y%m%d %H:%M:%S.%f')} DSC Validator v1.0 -- Proof of Space")

    nonce = -1  # Default value if not found
    accesses = 0
    while time.time() - start_p_time < time_limit:
        nonce, accesses = self.lookup(latest_block_hash, current_difficulty)
        if nonce != -1:
            break  # Exit if nonce found

    end_time = time.time()
    hash_rate = self.calculate_hash_rate(start_p_time, end_time, self.buckets * self.cup_size * self.cups_per_bucket)

    # Print statements remain the same
    if nonce != -1:
        return True, nonce, hash_rate
    else:
        return False, -1, hash_rate
```

Figure 4: Proof of Space

Proof of Memory: This class uses multi-threading to efficiently generate and organize hashes based on the available memory. It uses a timed binary search algorithm to find the correct nonce corresponding to the block hash.

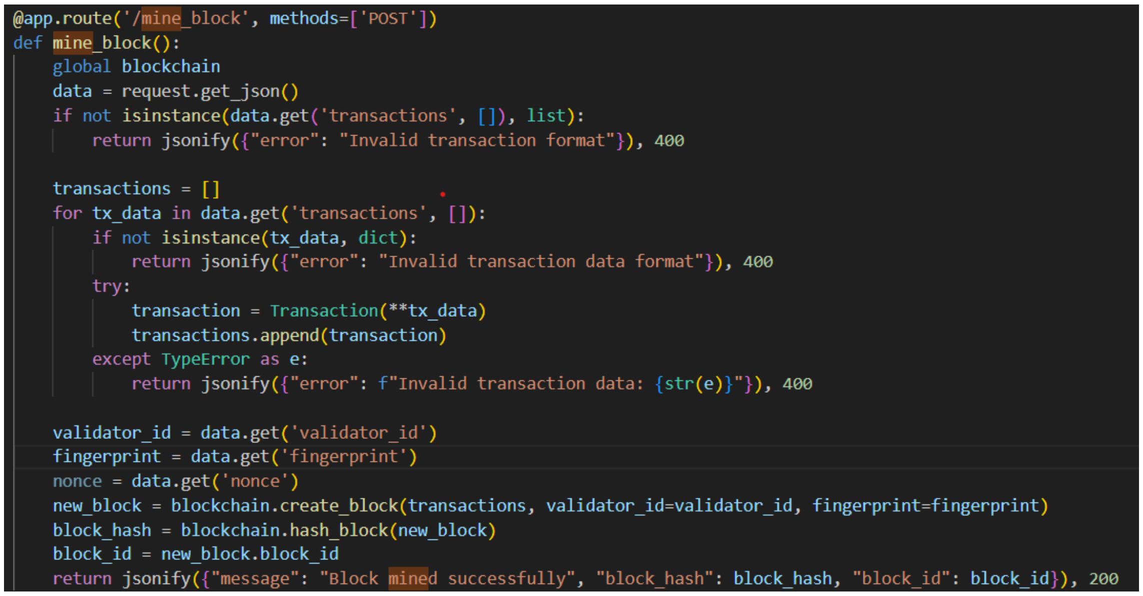


```
def perform_pom(self, block_hash, block_id, difficulty):
    start_p_time = time.time()
    time_limit = 6
    print(f"{time.strftime('%Y%m%d %H:%M:%S.%f')} DSC v1.0")
    print(f"{time.strftime('%Y%m%d %H:%M:%S.%f')} Proof of Memory ({self.threads} threads)")
    print(f"{time.strftime('%Y%m%d %H:%M:%S.%f')} Fingerprint: {self.fingerprint}")
    print(f"{time.strftime('%Y%m%d %H:%M:%S.%f')} block {block_id}, diff {difficulty}, hash {block_hash}")

    nonce = -1  # Default value if not found
    hashes = self.generate_and_organize_hashes(start_p_time, time_limit)
    if hashes:
        # Perform timed binary search
        nonce = self.timed_binary_search(hashes, block_hash[:difficulty], time_limit)

    end_time = time.time()
    hash_rate = self.calculate_hash_rate(start_p_time, end_time, len(hashes))

    if nonce != -1:
        print(f"{time.strftime('%Y%m%d %H:%M:%S.%f')} Block hash found, NONCE: {nonce}, Hash Rate: {hash_rate} H/s")
        return True, nonce, hash_rate
    else:
        print(f"{time.strftime('%Y%m%d %H:%M:%S.%f')} Block hash not found, NONCE: -1, Hash Rate: {hash_rate} H/s")
        return False, -1, hash_rate
```

Figure 5: Proof of Memory

3.1.5 Blockchain Server

The blockchain server initiates by generating a genesis block, crucial for the validator's validation of subsequent blocks. Subsequently, it updates wallet balances, embedding an initial amount of 200,000 coins for each wallet. The creation of a new block is triggered by the mineblock HTTP endpoint. This process involves utilizing the list of transactions, the previous block's hash, and a timestamp to construct a block. Following this, the status of each transaction is altered to true, facilitating wallet updates with the transaction details. Upon block creation, its hash is computed using the blake3 algorithm. The blockchain leverages Flask to respond to other components within the network.



```python
@app.route('/mine_block', methods=['POST'])
def mine_block():
    global blockchain
    data = request.get_json()
    if not isinstance(data.get('transactions', []), list):
        return jsonify({"error": "Invalid transaction format"}), 400

    transactions = []
    for tx_data in data.get('transactions', []):
        if not isinstance(tx_data, dict):
            return jsonify({"error": "Invalid transaction data format"}), 400
        try:
            transaction = Transaction(**tx_data)
            transactions.append(transaction)
        except TypeError as e:
            return jsonify({"error": f"Invalid transaction data: {str(e)}"}), 400

    validator_id = data.get('validator_id')
    fingerprint = data.get('fingerprint')
    nonce = data.get('nonce')
    new_block = blockchain.create_block(transactions, validator_id=validator_id, fingerprint=fingerprint)
    block_hash = blockchain.hash_block(new_block)
    block_id = new_block.block_id
    return jsonify({"message": "Block mined successfully", "block_hash": block_hash, "block_id": block_id}), 200
```

Figure 6: Mining a Block

3.1.6 Monitor

This component delivers blockchain statistics to the users, encompassing the last block header, unique wallet addresses, and the total coins in the network. Additionally, it monitors the total transactions in the pool and the validator statistics from the Metronome server.

3.2 Control Flow

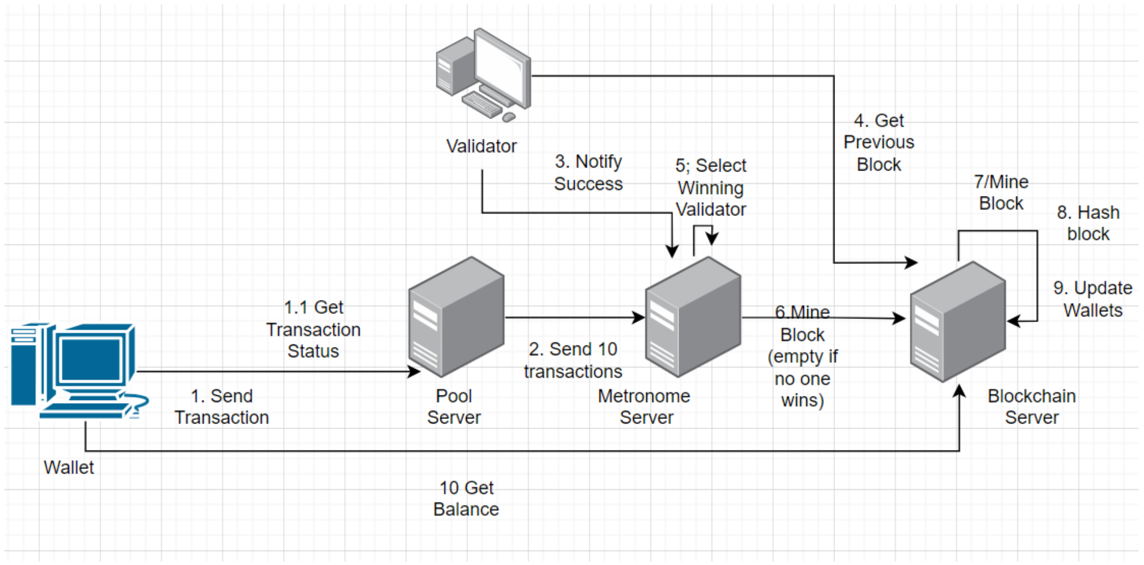


Figure 7: Control flow of the network

3.3 Extra Credit

For the extra credit part, about the distributed version graph, We have implemented certain parts to align with it. It's important to note that this implementation is not a fully real-time version but rather a proposed solution. For Blockchain, the pool, and metronome, we've divided the scripts into two parts. One focuses on core functionality, while the other serves as a handler file responsible for managing instances, network routes, synchronization, and other functions. To achieve real-time functionality, we require various synchronization functions and security measures for shared states, along with consensus algorithms to ensure agreement. Additionally, it's advisable to transition from Flask to another server like Gunicorn, capable of handling multiple instances in real-time, as Flask may not efficiently manage this requirement.

3.3.1 For Blockchain

Utilizing a database to maintain a ledger, we can implement consensus algorithms to verify added blocks and maintain a shared state across all blockchain instances.

3.3.2 For the Pool

Implementing functions to manage transactions across instances, ensuring no duplicates and handling other aspects. All instances should synchronize with each other.

3.3.3 For Metronome Server

Similar to blockchain and the pool, synchronization among instances is essential for generating blocks. Redis can be employed to maintain memory synchronization and facilitate block creation.

3.4 Challenges And Solutions

4 Evaluation

Leveraging the Chameleon Test bed, our experimentation involved a setup comprising 24 virtual machines. Within this environment, we configured 12 validators, a pool, 8 wallets, a blockchain server, a metronome server, and a monitor. Subsequently, we systematically conducted tests across all proof types, examining their performance under varying conditions with 1, 2, and 4 wallets.

4.1 Proof of Work

Clients	Min Latency	Max Latency	Avg Latency	Throughput 128000tx
1	7.02	6.07	7.11	160.90
2	7.02	6.07	7.95	193.08
4	7.10	6.22	8.00	227.83
8	7.08	6.03	8.05	259.52

Table 1: Performance on Proof Of Work

4.2 Proof of Space

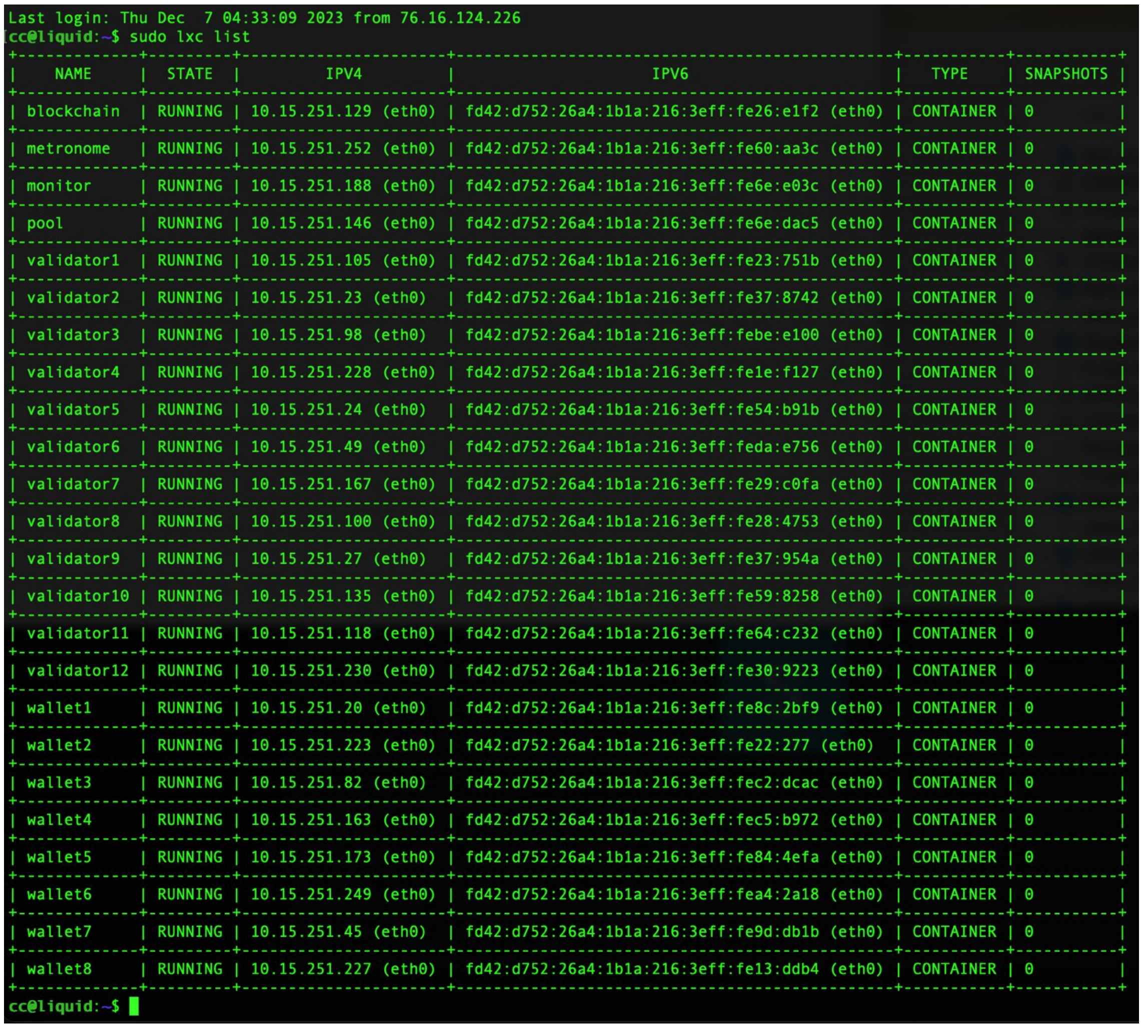
Clients	Min Latency	Max Latency	Avg Latency	Throughput 128000tx
1	5.45	3.04	6.11	166.18
2	5.97	3.16	6.15	196.92
4	6.00	3.06	6.20	236.30
8	6.02	3.62	6.25	274.11

Table 2: Performance on Proof Of Space

4.3 Proof of Memory

Clients	Min Latency	Max Latency	Avg Latency	Throughput 128000tx
1	5.98	2.03	6.15	166.27
2	6.00	2.77	6.15	194.53
4	6.02	2.82	6.20	231.49
8	6.15	4.05	6.25	266.22

Table 3: Performance on Proof Of Memory



```
Last login: Thu Dec  7 04:33:09 2023 from 76.16.124.226
cc@liquid:~$ sudo lxc list
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
|    NAME     |  STATE  |         IPV4         |                     IPV6                      |   TYPE    | SNAPSHOTS |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| blockchain  | RUNNING | 10.15.251.129 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe26:e1f2 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| metronome   | RUNNING | 10.15.251.252 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe60:aa3c (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| monitor     | RUNNING | 10.15.251.188 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe6e:e03c (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| pool        | RUNNING | 10.15.251.146 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe6e:dac5 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| validator1  | RUNNING | 10.15.251.105 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe23:751b (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| validator2  | RUNNING | 10.15.251.23 (eth0)  | fd42:d752:26a4:1b1a:216:3eff:fe37:8742 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| validator3  | RUNNING | 10.15.251.98 (eth0)  | fd42:d752:26a4:1b1a:216:3eff:febe:e100 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| validator4  | RUNNING | 10.15.251.228 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe1e:f127 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| validator5  | RUNNING | 10.15.251.24 (eth0)  | fd42:d752:26a4:1b1a:216:3eff:fe54:b91b (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| validator6  | RUNNING | 10.15.251.49 (eth0)  | fd42:d752:26a4:1b1a:216:3eff:feda:e756 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| validator7  | RUNNING | 10.15.251.167 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe29:c0fa (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| validator8  | RUNNING | 10.15.251.100 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe28:4753 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| validator9  | RUNNING | 10.15.251.27 (eth0)  | fd42:d752:26a4:1b1a:216:3eff:fe37:954a (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| validator10 | RUNNING | 10.15.251.135 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe59:8258 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| validator11 | RUNNING | 10.15.251.118 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe64:c232 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| validator12 | RUNNING | 10.15.251.230 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe30:9223 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| wallet1     | RUNNING | 10.15.251.20 (eth0)  | fd42:d752:26a4:1b1a:216:3eff:fe8c:2bf9 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| wallet2     | RUNNING | 10.15.251.223 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe22:277 (eth0)  | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| wallet3     | RUNNING | 10.15.251.82 (eth0)  | fd42:d752:26a4:1b1a:216:3eff:fec2:dcac (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| wallet4     | RUNNING | 10.15.251.163 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fec5:b972 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| wallet5     | RUNNING | 10.15.251.173 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe84:4efa (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| wallet6     | RUNNING | 10.15.251.249 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fea4:2a18 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| wallet7     | RUNNING | 10.15.251.45 (eth0)  | fd42:d752:26a4:1b1a:216:3eff:fe9d:db1b (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
| wallet8     | RUNNING | 10.15.251.227 (eth0) | fd42:d752:26a4:1b1a:216:3eff:fe13:ddb4 (eth0) | CONTAINER | 0         |
+-------------+---------+----------------------+-----------------------------------------------+-----------+-----------+
cc@liquid:~$ 
```

Figure 8: Testing Environment

5 Conclusions

The implementation of the blockchain-based system, DataSys Coin, represents a comprehensive and innovative approach to secure and transparent transactions. By leveraging a custom blockchain network with Python, Flask, and Blake3, the project addresses crucial aspects of data integrity, confidentiality, and availability. The diverse components, including the Wallet, Mining Pool Server, Metronome Server, Validator, Blockchain Server, and Monitor, collectively contribute to the creation of a decentralized and tamper-proof environment. The project aligns with industry trends and explores the practical application of blockchain in real-world scenarios. The comprehensive evaluation criteria, encompassing performance, security, and scalability, provide a robust framework for assessing the viability and effectiveness of the implemented blockchain network. The Validator's handling of proof algorithms and the Blockchain Server's ledger maintenance underscore the project's commitment to security and reliability. The Monitor's role in delivering blockchain statistics and overseeing transactions and validator statistics enhances the transparency and monitoring capabilities of the system. Our project emphasizes rigorous testing and evaluation as essential components of ensuring scalability, reliability, and optimal performance.