DataSys Coin BlockChain

Dheeraj Reddy Banda, Dhanush Bathineni, and Anshuman Raturi

Illinois Institute of Technology

December 2023

Abstract

Blockchain technology stands out as a transformative innovation, introducing a decentralized, secure, and transparent system for handling transactions. It disrupts the reliance on intermediaries like banks and governments, with potential applications across sectors such as finance, supply chain, and healthcare, positioning it alongside influential technologies like the Internet, mobile computing, and artificial intelligence.

This project delves into the meticulous creation of the DataSys Coin (DSC) Blockchain from scratch, utilizing the Java programming language and incorporating libraries for enhanced efficiency. A comprehensive documentation of dependencies, presented in a configure file, ensures transparency throughout the implementation process. The evaluation of the DSC implementation spans both local and Chameleon cloud environments, with a focus on measuring functionality and performance in terms of throughput and latency. The Chameleon cloud assessment involves deploying the blockchain on robust bare-metal instances and virtual machines or containers with specific resource constraints, providing insights into the system's capabilities under various conditions.

1 Introduction

Blockchain technology's emergence marks a pivotal shift in transaction management, promising decentralization, heightened security, and transparency. Its disruption of traditional intermediaries like banks heralds a new era of efficient, trust-based transactions, placing blockchain in the league of influential technologies alongside the Internet, mobile computing, and artificial intelligence.

1.1 Motivation

Implementation of DSC Blockchain:

- Develop the DSC Blockchain from scratch.
- Utilize Java as programming language.

Integration of Libraries:

- Use libraries such as blake3,YAML, bitcoinj, Java RMI to make the implementation faster and easier.

Local Environment Testing:

- Evaluate DSC Blockchain functionality in a local environment.
- Conduct rigorous testing to ensure the security, transparency, and decentralization features meet the project objectives.

Chameleon Cloud Evaluation:

- Deploy DSC Blockchain on the Chameleon cloud infrastructure (https://www.chameleoncloud.org).
- Utilize bare-metal instances with at least 24 cores and 128GB of RAM for in-depth performance evaluation.
- Assess functionality, throughput, and latency on up to 16 instances (virtual machines or containers) with 2 cores, 4GB RAM, and 12GB storage space

2 Problem Statement

In the landscape of blockchain technology, recognized for its decentralized and distributed nature, there exists a unique challenge: to implement the DataSys Coin (DSC) Blockchain within a centralized architecture. The conventional strength of blockchain lies in its ability to operate without a central authority, yet this project seeks to explore and address the complexities associated with adapting blockchain principles to a centralized model.

The primary objective is to develop a centralized architecture for the DSC Blockchain, utilizing Java as the programming language. This entails the meticulous construction of a secure, transparent, and efficient system for recording and verifying transactions, departing from the typical decentralized paradigm. While blockchain technology is celebrated for its autonomy from intermediaries like banks and governments, this project seeks to investigate the implications and potential advantages of a centralized approach.

Key challenges include redefining the consensus mechanism, ensuring security in a centralized environment, and optimizing performance within the constraints of a single, central node. The implementation should remain faithful to the core principles of blockchain, such as transparency and immutability, while accommodating the nuances of a centralized structure.

This project aims to contribute insights into the feasibility and trade-offs associated with implementing a blockchain, specifically the DSC Blockchain, in a centralized architecture. It requires a careful balance between preserving the essence of blockchain technology and adapting it to a model that diverges from its traditionally decentralized nature.

3 Proposed Solution

To address the challenge of implementing the DataSys Coin (DSC) Blockchain in a centralized architecture, a carefully crafted solution is proposed. To address the challenge of implementing the DataSys Coin (DSC) Blockchain within a centralized architecture, the proposed solution involves the meticulous design and implementation of six distinct components. Each of these components represents separate processes capable of seamless communication through network sockets, following centralized architecture.

3.1 Components

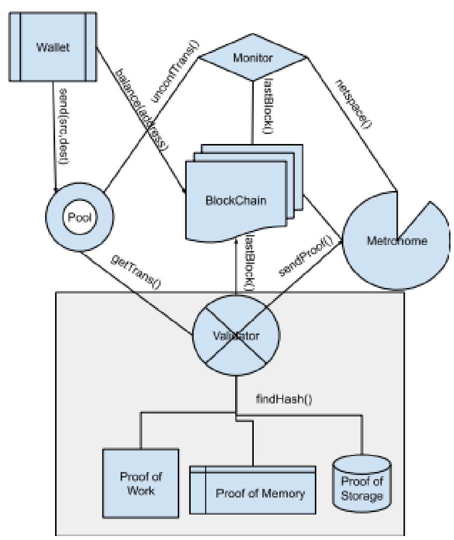


Figure 1: DataSys Coin Blockchain Centralized Architecture

3.1.1 Wallet

the Wallet component offers the following key functionalities:

1. Wallet Creation:
   - Enables the generation of new wallets for users participating in the blockchain network.
   - Creates public and private key pairs using SHA256, essential for secure transaction processing and wallet identification.
2. Transaction Sending:

- Facilitates the initiation and submission of cryptocurrency transactions to the blockchain.
- Allows users to specify transaction details, such as the recipient's address and the amount to be transferred.

3. Balance Viewing:
   - Provides users with real-time access to their wallet balance.
   - Retrieves and displays the current balance by querying the blockchain ledger.
4. Evaluation Function
   - latency function used for latency evaluation
   - throughput function used for throughput evaluation

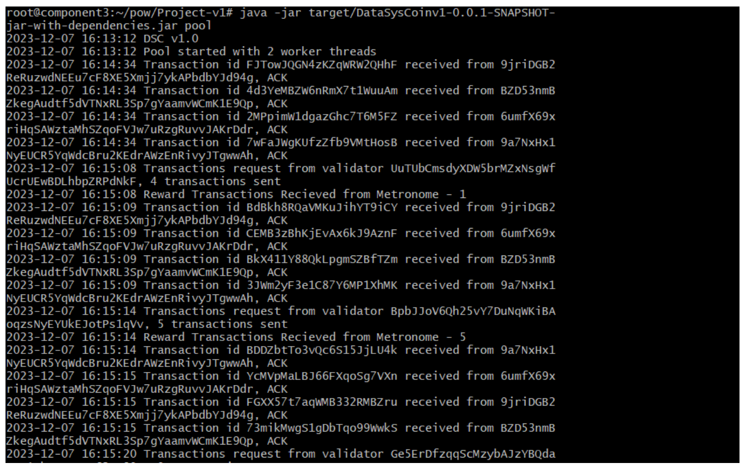
The Wallet component acts as an intuitive gateway for users to interact with the blockchain, making financial transactions secure and accessible. By seamlessly integrating into the centralized architecture, the Wallet enhances the overall user experience of the DSC Blockchain, ensuring a user-friendly and efficient interface for managing cryptocurrency assets. The wallet has the following options:

- help
- create
- key
- balance
- send
- transaction

help: Help displays the available options in the wallet
create: Creates new wallet using SHA256 to create public/private keys of 256bit length, and stores the keys in dsc-config.yaml and dsc-key.yaml in Base58 encoding. In case the wallet already exists, it will show a message that a wallet already exists.
key: Displays the keys
balance: Displays the balance of the wallet
send: Sends coins to other wallets. It establishes communication with the pool server to dispatch a transaction and patiently awaits confirmation that the transaction has been successfully received. It's important to highlight that the Transaction ID is assigned by the wallet itself. It's noteworthy that the pool server is not tasked with processing the transaction; rather, its role is to acknowledge the receipt of the transaction and relay that confirmation back to the originating wallet.
transaction: Connects with the pool server, sends a transaction ID, and awaits

a response regarding the transaction status. The potential responses include "submitted," "unconfirmed," or "unknown." In the event of a "unknown" response from the pool, a subsequent inquiry is directed to the blockchain to inquire about the transaction ID. Responses from the blockchain can be "confirmed" or "invalid".

3.1.2 Pool



```
root@component3:~/pow/Project-v1# java -jar target/DataSysCoinv1-0.0.1-SNAPSHOT-
jar-with-dependencies.jar pool
2023-12-07 16:13:12 DSC v1.0
2023-12-07 16:13:12 Pool started with 2 worker threads
2023-12-07 16:14:34 Transaction id FJTowJQGN4zKZqWRW2QHhF received from 9jriDGB2
ReRuzwdNEEu7cF8XE5Xmjj7ykAPbdbYJd94g, ACK
2023-12-07 16:14:34 Transaction id 4d3YeMBZW6nRmX7t1WuuAm received from BZD53nmB
ZkegAudtf5dVTNxRL3Sp7gYaamvWCmK1E9Qp, ACK
2023-12-07 16:14:34 Transaction id 2MPpimW1dgazGhc7T6M5FZ received from 6umfX69x
riHqSAWztaMhSZqoFVJw7uRzgRuvvJAKrDdr, ACK
2023-12-07 16:14:34 Transaction id 7wFaJWgKUfzZfb9VMtHosB received from 9a7NxHx1
NyEUCR5YqWdcBru2KEdrAWzEnRivyJTgwwAh, ACK
2023-12-07 16:15:08 Transactions request from validator UuTUbCmsdyXDW5brMZxNsgWf
UcrUEwBDLhbpZRPdNkF, 4 transactions sent
2023-12-07 16:15:08 Reward Transactions Recieved from Metronome - 1
2023-12-07 16:15:09 Transaction id BdBkh8RQaVMKuJihYT9iCY received from 9jriDGB2
ReRuzwdNEEu7cF8XE5Xmjj7ykAPbdbYJd94g, ACK
2023-12-07 16:15:09 Transaction id CEMB3zBhKjEvAx6kJ9AznF received from 6umfX69x
riHqSAWztaMhSZqoFVJw7uRzgRuvvJAKrDdr, ACK
2023-12-07 16:15:09 Transaction id BkX411Y88QkLpgmSZBfTZm received from BZD53nmB
ZkegAudtf5dVTNxRL3Sp7gYaamvWCmK1E9Qp, ACK
2023-12-07 16:15:09 Transaction id 3JWm2yF3e1C87Y6MP1XhMK received from 9a7NxHx1
NyEUCR5YqWdcBru2KEdrAWzEnRivyJTgwwAh, ACK
2023-12-07 16:15:14 Transactions request from validator BpbJJoV6Qh25vY7DuNqWKiBA
oqzsNyEYUkEJotPs1qVv, 5 transactions sent
2023-12-07 16:15:14 Reward Transactions Recieved from Metronome - 5
2023-12-07 16:15:14 Transaction id BDDZbtTo3vQc6S15JjLU4k received from 9a7NxHx1
NyEUCR5YqWdcBru2KEdrAWzEnRivyJTgwwAh, ACK
2023-12-07 16:15:15 Transaction id YcMVpMaLBJ66FXqoSg7VXn received from 6umfX69x
riHqSAWztaMhSZqoFVJw7uRzgRuvvJAKrDdr, ACK
2023-12-07 16:15:15 Transaction id FGXX57t7aqWMB332RMBZru received from 9jriDGB2
ReRuzwdNEEu7cF8XE5Xmjj7ykAPbdbYJd94g, ACK
2023-12-07 16:15:15 Transaction id 73mikMwgS1gDbTqo99WwkS received from BZD53nmB
ZkegAudtf5dVTNxRL3Sp7gYaamvWCmK1E9Qp, ACK
2023-12-07 16:15:20 Transactions request from validator Ge5ErDfzqqScMzybAJzYBQda
```

Figure 2: Console output of pool

A pool server assigned with the singular responsibility of accepting a transaction, displaying it on the screen, and promptly responding to queries with an acknowledgment of the received transaction. It uses two data structures tailored for transactions. These data structures are designated to organize transactions into two distinct categories: the first Queue, denoted as "unprocessed," encompasses transactions originating from wallets, while the second Map, named "unconfirmed," comprises transactions initiated by a validator. Notably, the process involves transferring a transaction from the "submitted" stack to the "unconfirmed" stack upon a validator's transaction request. Subsequently, upon confirmation of a block, transactions appended to the last block must be expunged from the "unconfirmed" stack.

3.1.3 Validator

The Validator assumes a pivotal role in the DataSysCoinv1 system, contributing significantly to consensus building, transaction validation, and the creation of blocks. It starts by utilizing the RMI Registry on 'ValidatorRead.getValidatorPort()'.

The validator first registers with the metronome. It waits for the signal from metronome for searching for new block.

The registration process with the Metronome involves providing essential information such as IP address, port, and public key, followed by logging the registration result.

For Proof Selection, the Validator reads the selected proof type from the configuration. If Proof of Work is chosen, the process entails initiating Proof of Work, extracting parameters from the Blockchain, and employing a separate thread for Proof of Work calculations. The resultant proof is then forwarded to Metronome for approval. Upon approval, the Validator proceeds to create a block.

In the case of Proof of Memory, the Validator initiates the Proof of Memory process involving ‘pomThreads‘ and memory considerations. The fingerprint and public key are stored for memory calculations, utilizing multithreading to enhance the efficiency of both Proof of Work and Proof of Memory calculations.

Blockchain Interaction is a critical aspect of the Validator’s responsibilities. This involves communication with the blockchain, computation of block-related parameters, validation, and inclusion of transactions in the block. The completed block is then transmitted to the blockchain.

Overall, the Validator orchestrates a series of intricate processes, ranging from consensus-building mechanisms to the execution of specific proofs, ultimately contributing to the robustness and functionality of the DataSysCoinv1 system.

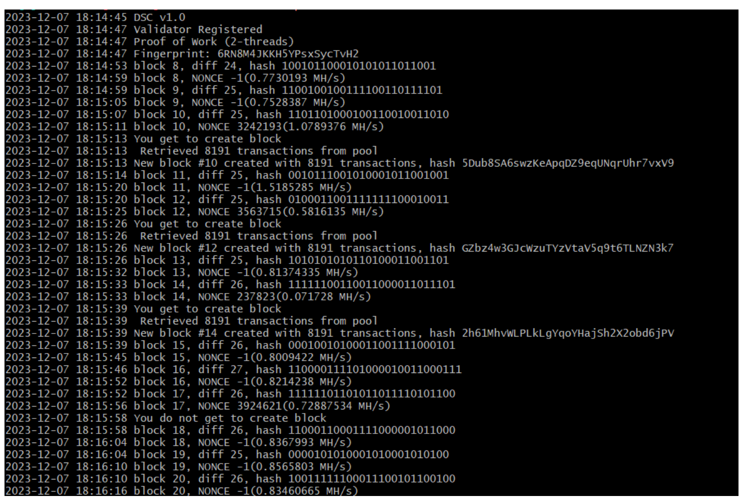


Figure 3: Console output of pow

Main functions

1. Register (to register with metronome)
2. Proof of Work

```java
@Override
public void run() {
    int ready = 0;
    int totalLength = this.input.getFingerprintb().length + this.input.getPublic_keyb().length + Integer.BYTES
            + Long.BYTES;
    /*ByteBuffer buffer = ByteBuffer.allocate(totalLength);
    buffer.put(this.input.getFingerprintb());
    buffer.put(this.input.getPublic_keyb());
    buffer.putInt(this.threadId);*/
    Blake32 hasher = new Blake32();
    while (true) {
        while (ready == this.input.ready.get()) {
            continue;
        }
        ready = this.input.ready.get();
        boolean found = false;
        long nonce = 0;
        long startTime = System.currentTimeMillis();
        long duration = 6000;
        while (System.currentTimeMillis() - startTime < duration && this.input.found.get() == false) {
            ByteBuffer buffer = ByteBuffer.allocate(totalLength);
            buffer.put(this.input.getFingerprintb());
            buffer.put(this.input.getPublic_keyb());
            buffer.putInt(this.threadId);
            buffer.putLong(nonce);
            byte[] inputBytes = buffer.array();
            byte[] hashBytes = hasher.blakeInBytes(inputBytes);
            //System.out.println(hasher.diffZeroAndOne(hashBytes, 2));
            found = compareFirstBits(hashBytes, this.input.getPrevHash(), input.getDiff());
            if(found) {
                this.input.found.set(found);
                this.input.setNonce(nonce);
                this.input.setThreadId(this.threadId);
            }else {
                nonce++;
            }

        }
        //log.logWithTimestamp("Thread ID: " + this.threadId + ", nonce: " + nonce);
```

Figure 4: proof of work code

```
2023-12-07 16:28:31 DSC v1.0
2023-12-07 16:28:34 Proof of Memory (2-threads, 1GB RAM)
2023-12-07 16:28:34 Fingerprint: 3FQuoQskFTfyhxmWre4XNk
2023-12-07 16:28:34 gen/org 1GB hashes using 2 passes
2023-12-07 16:28:34 generating hashes [Thread #1]
2023-12-07 16:28:34 generating hashes [Thread #0]
2023-12-07 16:28:47 finished generating hashes [Thread #1]
2023-12-07 16:28:47 finished generating hashes [Thread #0]
2023-12-07 16:28:47 sorting hashes
2023-12-07 16:29:07 finished sorting hashes
2023-12-07 16:29:07 gen/org 1GB hashes (32.0 sec)
2023-12-07 16:29:07 Validator Registered
2023-12-07 16:29:11 block 13, diff 17, hash 11100110000101100
Key not found within the time limit
2023-12-07 16:29:17 block 14, diff 16, hash 1101001010100100
Key not found within the time limit
2023-12-07 16:29:23 block 15, diff 15, hash 000001100111101
Key found at index: 6327026
2023-12-07 16:29:23 hashFound bHSD6CgbkK1zi9gb1a1vCEg5reos3hyh nonce 4049825 threadId 1
2023-12-07 16:29:23 input 4049825 nonce 4049825 threadId 1
2023-12-07 16:29:23 prev hash bGYDeQKZqXfJGFL8TaPdvyoTYvbNwAPg fingerprint 3FQuoQskFTfyhxmWre4XNk publicKey AbXU7iPhwvp93HhVR9GcGCG4V7K8as1nGiJEhXd46jSt
2023-12-07 16:29:29 You get to create block
2023-12-07 16:29:29  Retrieved 8 transactions from pool
2023-12-07 16:29:29 New block #15 created with 8 transactions, hash 47zEQamoMM24NNNp6N8EQs422FdDdHjTX
2023-12-07 16:29:29 block 16, diff 15, hash 001000100011101
Key not found within the time limit
```

Figure 5: Console output of pom

```java
@Override
public void run() {
    // TODO Auto-generated method stub
    int totalLength = this.hashStoring.getFingerprintb().length + this.hashStoring.getPublic_keyb().length + Integer.BYTES
            + Long.BYTES;
    Blake32 hasher = new Blake32();
    long nonce = 0;

    long memoryHashes = hashStoring.memoryHashes.get();
    log.logWithTimestamp("generating hashes [Thread #" + this.threadId + "]");
    while(hashStoring.totalHashes.get() <= memoryHashes) {
        Hashes hash = new Hashes();
        ByteBuffer buffer = ByteBuffer.allocate(totalLength);
        buffer.put(this.hashStoring.getFingerprintb());
        buffer.put(this.hashStoring.getPublic_keyb());
        buffer.putInt(this.threadId);
        buffer.putLong(nonce);
        byte[] inputBytes = buffer.array();
        byte[] hashBytes = hasher.blakeInBytes(inputBytes);
        //log.logWithTimestamp("hashbytes: " + Base58.encode(hashBytes));
        hash.setHash(hashBytes);
        hash.setNonce(nonce);
        hash.setThreadId(this.threadId);
        hashes.addToHashes(hash);
        nonce++;
        hashStoring.totalHashes.incrementAndGet();
        //log.logWithTimestamp("hash " + Base58.encode(hashes.getHash()) + " nonce " + hashes.getNonce());
    }
    log.logWithTimestamp("finished generating hashes [Thread #" + this.threadId + "]");
    hashStoring.done.incrementAndGet();
}

}
```

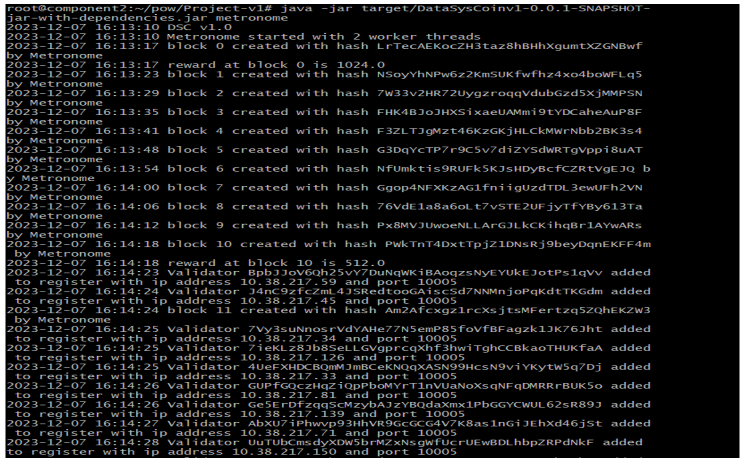
Figure 6: proof of memory code

3. Proof of Memory

3.1.4 Metronome

Metronome serves as a multifaceted entity, encompassing critical functionalities such as transaction processing, difficulty adjustment, and seamless interaction with both the blockchain and the transaction pool. The operational aspects of Metronome are detailed as follows:

- RMI Registry Setup:
  - Initiates the creation of an instance of MetronomeImpl.
  - Establishes a binding between the created instance and the RMI registry for subsequent communication.
- Interaction with Transaction Pool (PoolKeyInterface):
  - Engages in communication with the transaction pool through RMI, utilizing the PoolKeyInterface. It uses this to send reward transactions to the pool.
- Register interface
  - Provides and interface for the validators to register with metronome.
- Transaction Processing:
  - Checks the readiness of Metronome to process transactions through a signaling mechanism.
  - Gathers pertinent information from the MetronomeImpl instance, including details such as the creator, rewards, and proofs.
  - If rewards surpass a threshold of 1, Metronome creates transactions with corresponding rewards for each proof. Otherwise, it generates a transaction with a modest percentage (1%) of the total value and dispatches them to the transaction pool.
- Proof Handling and Signal Sending:
  - Adjusts the difficulty level and clears proofs within the MetronomeImpl instance.
  - Sends signals to synchronize the system.
  - Introduces a brief delay, sleeping for 6000 milliseconds (6 seconds).
- Block Creation:
  - In the absence of new proofs, Metronome retrieves block information from the MetronomeImpl instance.



```
root@component2:~/pow/Project-v1# java -jar target/DataSysCoinv1-0.0.1-SNAPSHOT-
jar-with-dependencies.jar metronome
2023-12-07 16:13:10 DSC v1.0
2023-12-07 16:13:10 Metronome started with 2 worker threads
2023-12-07 16:13:17 block 0 created with hash LrTecAEKocZH3taz8hBHhXgumtXZGNBwf
by Metronome
2023-12-07 16:13:17 reward at block 0 is 1024.0
2023-12-07 16:13:23 block 1 created with hash NSoyYhNPw6z2KmSUKfwfhz4xo4boWFLq5
by Metronome
2023-12-07 16:13:29 block 2 created with hash 7W33v2HR72UygzroqqVdubGzd5XjMMPSN
by Metronome
2023-12-07 16:13:35 block 3 created with hash FHK4BJoJHXSixaeUAMmi9tYDCaheAuP8F
by Metronome
2023-12-07 16:13:41 block 4 created with hash F3ZLTJgMzt46KzGKjHLCkMWrNbb2BK3s4
by Metronome
2023-12-07 16:13:48 block 5 created with hash G3DqYcTP7r9C5v7diZYSdWRTgVppi8uAT
by Metronome
2023-12-07 16:13:54 block 6 created with hash NfUmktis9RUFk5KJsHDyBcfCZRtVgEJQ b
y Metronome
2023-12-07 16:14:00 block 7 created with hash Ggop4NFXKzAG1fniigUzdTDL3ewUFh2VN
by Metronome
2023-12-07 16:14:06 block 8 created with hash 76VdE1a8a6oLt7vSTE2UFjyTfYBy613Ta
by Metronome
2023-12-07 16:14:12 block 9 created with hash Px8MVJUwoeNLLArGJLkCKihqBr1AYwARs
by Metronome
2023-12-07 16:14:18 block 10 created with hash PWkTnT4DxtTpjZ1DNsRj9beyDqnEKFF4m
 by Metronome
2023-12-07 16:14:18 reward at block 10 is 512.0
2023-12-07 16:14:23 Validator BpbJJoV6Qh25vY7DuNqWKiBAoqzsNyEYUkEJotPs1qVv added
 to register with ip address 10.38.217.59 and port 10005
2023-12-07 16:14:24 Validator J4nC9zfcZmL4JSRedtooGAiscSd7NNMnjoPqKdtTKGdm added
 to register with ip address 10.38.217.45 and port 10005
2023-12-07 16:14:24 block 11 created with hash Am2Afcxgz1rcXsjtsMFertzq5ZQhEKZW3
 by Metronome
2023-12-07 16:14:25 Validator 7Vy3suNnosrVdYAHe77N5emP85foVfBFagzk1JK76Jht added
 to register with ip address 10.38.217.34 and port 10005
2023-12-07 16:14:25 Validator 7ieKLz8Jb8SeLLGVgprcqXhf3hwiTghCCBkaoTHUKfaA added
 to register with ip address 10.38.217.126 and port 10005
2023-12-07 16:14:25 Validator 4UeFXHDCBQmMJmBCeKNQqXASN99HcsN9viYKytW5q7Dj added
 to register with ip address 10.38.217.33 and port 10005
2023-12-07 16:14:26 Validator GUPfGQczHqZiQpPboMYrT1nVUaNoXsqNFqDMRRrBUK5o added
 to register with ip address 10.38.217.81 and port 10005
2023-12-07 16:14:26 Validator Ge5ErDfzqqScMzybAJzYBQdaXmx1PbGGYCWUL62sR89J added
 to register with ip address 10.38.217.139 and port 10005
2023-12-07 16:14:27 Validator AbXU7iPhwvp93HhVR9GcGCG4V7K8as1nGiJEhXd46jSt added
 to register with ip address 10.38.217.71 and port 10005
2023-12-07 16:14:28 Validator UuTUbCmsdyXDW5brMZxNsgWfUcrUEwBDLhbpZRPdNkF added
to register with ip address 10.38.217.150 and port 10005
```

Figure 7: Console output of metronome

   - Initiates the creation of a new block and transmits it to the DSC blockchain using RMI.

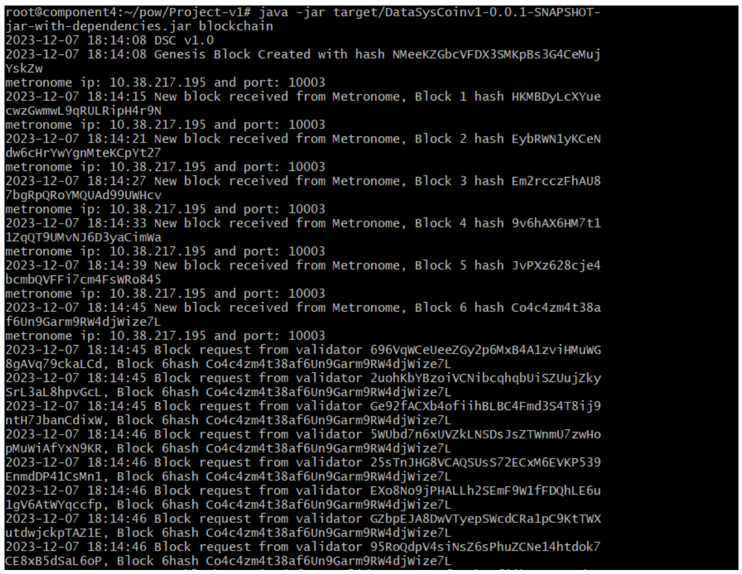
These functionalities collectively underscore the comprehensive role of Metronome in orchestrating key processes within the DataSysCoinv1 system, ranging from managing transactions to dynamically adjusting difficulty levels and ensuring seamless integration with the blockchain.

3.1.5 Blockchain

The entity responsible for overseeing the intricacies of the blockchain, transactions, and account balances operates through the utilization of Remote Method Invocation (RMI) to facilitate seamless communication. Key functionalities of this component include:

- Genesis Block Initialization:
   - Undertakes the critical task of setting up the initial block, establishing the foundation upon which the entire blockchain is built.
- Continuous Block Monitoring:
   - Maintains an ongoing vigilance for the emergence of new blocks within the blockchain, ensuring a real-time awareness of changes and updates.
- Transaction Processing:
   - Retrieves transactions from newly formed blocks, encompassing the extraction of pertinent transaction details.

- Dynamically updates recipient and sender balances in accordance with the values associated with each transaction.
- Manages a comprehensive transaction history for sender addresses, ensuring a detailed record of all transactions initiated by specific addresses.



```
root@component4:~/pow/Project-v1# java -jar target/DataSysCoinv1-0.0.1-SNAPSHOT-
jar-with-dependencies.jar blockchain
2023-12-07 18:14:08 DSC v1.0
2023-12-07 18:14:08 Genesis Block Created with hash NMeeKZGbcVFDX3SMKpBs3G4CeMuj
YskZw
metronome ip: 10.38.217.195 and port: 10003
2023-12-07 18:14:15 New block received from Metronome, Block 1 hash HKMBDyLcXYue
cwzGwmwL9qRULRipH4r9N
metronome ip: 10.38.217.195 and port: 10003
2023-12-07 18:14:21 New block received from Metronome, Block 2 hash EybRWN1yKCeN
dw6cHrYwYgnMteKCpYt27
metronome ip: 10.38.217.195 and port: 10003
2023-12-07 18:14:27 New block received from Metronome, Block 3 hash Em2rcczFhAU8
7bgRpQRoYMQUAd99UWHcv
metronome ip: 10.38.217.195 and port: 10003
2023-12-07 18:14:33 New block received from Metronome, Block 4 hash 9v6hAX6HM7t1
1ZqQT9UMvNJ6D3yaCimWa
metronome ip: 10.38.217.195 and port: 10003
2023-12-07 18:14:39 New block received from Metronome, Block 5 hash JvPXz628cje4
bcmbQVFFi7cm4FsWRo845
metronome ip: 10.38.217.195 and port: 10003
2023-12-07 18:14:45 New block received from Metronome, Block 6 hash Co4c4zm4t38a
f6Un9Garm9RW4djWize7L
metronome ip: 10.38.217.195 and port: 10003
2023-12-07 18:14:45 Block request from validator 696VqWCeUeeZGy2p6MxB4A1zviHMuWG
8gAVq79ckaLCd, Block 6hash Co4c4zm4t38af6Un9Garm9RW4djWize7L
2023-12-07 18:14:45 Block request from validator 2uohKbYBzoiVCNibcqhqbUiSZUujZky
SrL3aL8hpvGcL, Block 6hash Co4c4zm4t38af6Un9Garm9RW4djWize7L
2023-12-07 18:14:45 Block request from validator Ge92fACXb4ofiihBLBC4Fmd3S4T8ij9
ntH7JbanCdixW, Block 6hash Co4c4zm4t38af6Un9Garm9RW4djWize7L
2023-12-07 18:14:46 Block request from validator 5WUbd7n6xUVZkLNSDsJsZTWnmU7zwHo
pMuWiAfYxN9KR, Block 6hash Co4c4zm4t38af6Un9Garm9RW4djWize7L
2023-12-07 18:14:46 Block request from validator 25sTnJHG8VCAQSUsS72ECxM6EVKP539
EnmdDP41CsMn1, Block 6hash Co4c4zm4t38af6Un9Garm9RW4djWize7L
2023-12-07 18:14:46 Block request from validator EXo8No9jPHALLh2SEmF9W1fFDQhLE6u
1gV6AtWYqccfp, Block 6hash Co4c4zm4t38af6Un9Garm9RW4djWize7L
2023-12-07 18:14:46 Block request from validator GZbpEJA8DwVTyepSWcdCRa1pC9KtTWX
utdwjckpTAZ1E, Block 6hash Co4c4zm4t38af6Un9Garm9RW4djWize7L
2023-12-07 18:14:46 Block request from validator 95RoQdpV4siNsZ6sPhuZCNe14htdok7
CE8xB5dSaL6oP, Block 6hash Co4c4zm4t38af6Un9Garm9RW4djWize7L
```

Figure 8: Console output of blockchain

This component, operating through the efficient communication mechanism of RMI, assumes a central role in the orchestration of the blockchain, ensuring the integrity of transactions, and meticulously managing account balances to maintain the robustness and functionality of the overall system.

3.1.6 Monitor

The monitoring system is designed to gather comprehensive statistics about the active system, focusing on key components. It provides insights into the current state of various elements within the system:

- BlockChain
  - Last Block Header: Captures details regarding the most recent block, including its header information.
  - Number of Unique Wallet Addresses: Keeps track of the count of distinct wallet addresses existing in the blockchain.
  - Number of Total Coins in Circulation: Monitors the total count of coins currently in circulation within the blockchain.

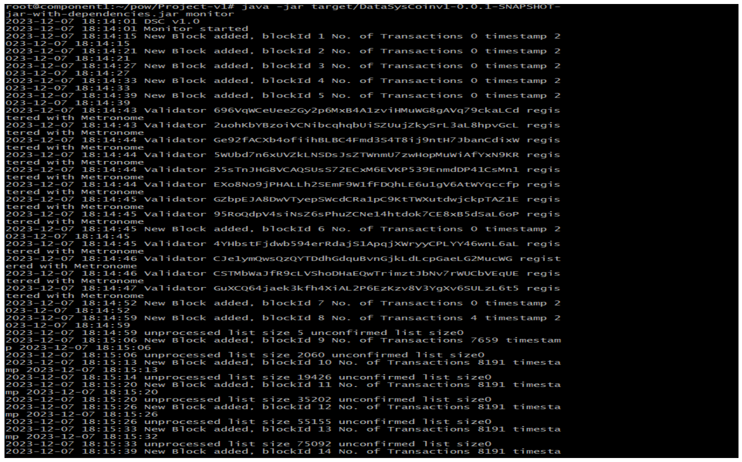


Figure 9: Console output of monitor

- Pool:
  - Number of Transactions in Submitted and Unconfirmed: Captures the transaction status, distinguishing between those in the "submitted" and "unconfirmed" pools.
- Metronome:
  - Number of Validators:Provides an overview of the current count of validators participating in the Metronome system.
  - Hashes per Second: Measures the rate at which hash calculations are being performed per second.
  - Total Hashes Stored: Keeps track of the cumulative number of hashes stored within the Metronome system.

This monitoring mechanism plays a crucial role in gauging the health and performance of the system, offering real-time statistics on the blockchain, transaction pools, and Metronome components. By systematically tracking these metrics, the monitoring system enhances the ability to analyze, optimize, and maintain the overall efficiency of the running system.

3.2 Libraries

We have used the following libraries:

- Blake3
- YAML
- BitcoinJ
- Java RMI

3.2.1 Blake3

We have used Blake3 for the generation of hashes. BLAKE3 is a cryptographic hash function that is:

- Much faster than MD5, SHA-1, SHA-2, SHA-3, and BLAKE2.
- Secure, unlike MD5 and SHA-1. And secure against length extension, unlike SHA-2.
- Highly parallelizable across any number of threads and SIMD lanes, because it's a Merkle tree on the inside.
- Capable of verified streaming and incremental updates, again because it's a Merkle tree.A PRF, MAC, KDF, and XOF, as well as a regular hash.
- One algorithm with no variants, which is fast on x86-64 and also on smaller architectures.

However, there is no official implementation of Blake3 in Java so, we have used an unoptimized blake3 implementation from https://github.com/rctcwyvrn/blake3

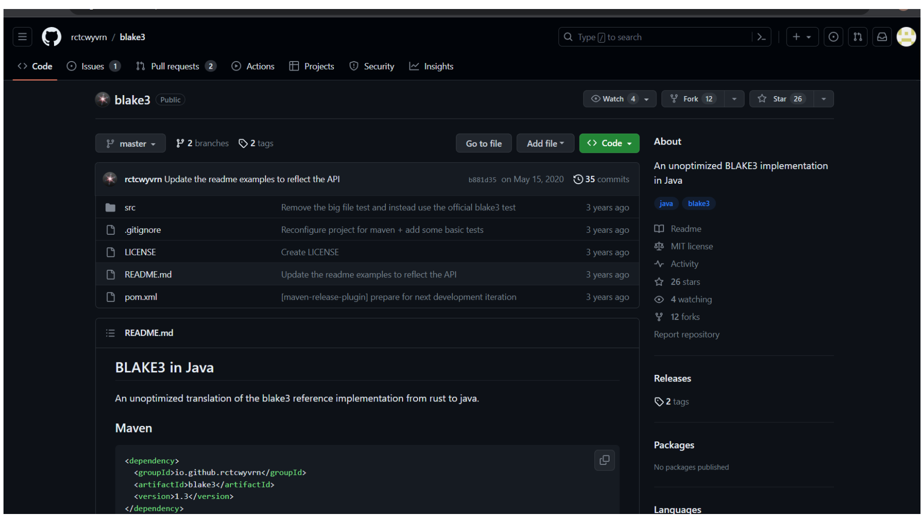


Figure 10: Blake 3 git repository

By Using Blake3 we were able to generate about a million hashes per second.

3.2.2 YAML

YAML serves as a data serialization language, offering a human-readable format for structuring and representing data. In our project, we have harnessed YAML to compose configuration files, encapsulating essential settings and parameters for the seamless configuration of the system. The readability and simplicity of YAML make it an ideal choice for expressing configurations, facilitating a clear and concise representation of various project-specific settings. This ensures that the configuration files are not only machine-readable but also easily comprehensible for human users, enhancing the overall maintainability and accessibility of the project.

3.2.3 BitcoinJ

BitcoinJ is a specialized library designed for interacting with the Bitcoin protocol. In our project, we have leveraged this library to implement Base58 encoding. Base58 is a binary-to-text encoding scheme commonly used in Bitcoin-related applications to represent data, such as Bitcoin addresses. BitcoinJ provides the necessary functionalities and tools to work with the Bitcoin protocol efficiently, and our utilization of this library specifically focuses on the implementation of Base58 encoding within the context of our project. This ensures a standardized and secure representation of data, aligning with Bitcoin's protocol specifications.

3.2.4 Java RMI

The RMI (Remote Method Invocation) serves as an essential API that facilitates the creation of distributed applications in Java. This mechanism allows objects to invoke methods on an object residing in a different Java Virtual Machine (JVM). In our project, we have harnessed the power of RMI to establish communication between various components of the system. This enables seamless interaction and method invocation between different JVMs, enhancing the overall interoperability and functionality of the distributed application. By leveraging RMI, we have effectively bridged communication gaps between components, contributing to a more integrated and collaborative system architecture.

4 Evaluation

Evaluation is done in Chameleon. 12-validators and 16-experiments(pow-8 and pom-8)

4.1 Proof of Work Evaluation

4.1.1 Latency

1. 1 wallet
   latency = 995.426/128 = 7.77 sec/transactions

2. 2 wallet

   latency = (454.325/64+452.142/64)/2 = 7.07 sec/transactions

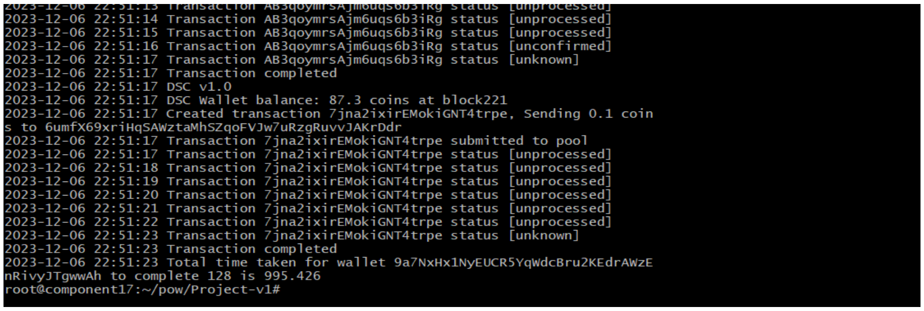
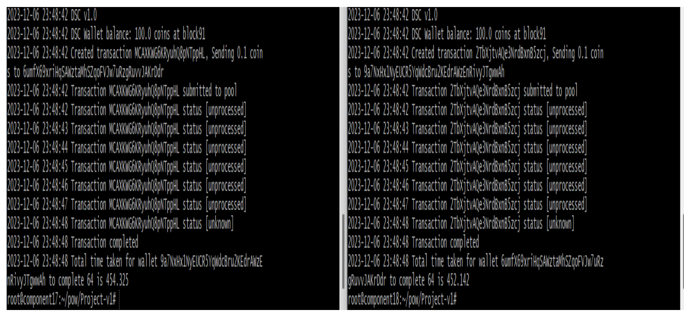


Figure 11: pow-latency-1 wallet



(a) 2 wallets - 1-2

Figure 12: 2 wallets - latency - pow

3. 4 wallet

   latency = (219.07/32+217.872/32+217.029/32+215.013/32)/4 = 6.725 sec/transactions

4. 8 wallet

   latency = (104.376/16+110.462/16+110.577/16+110.555/16+)/8 = 6.8 sec/transactions

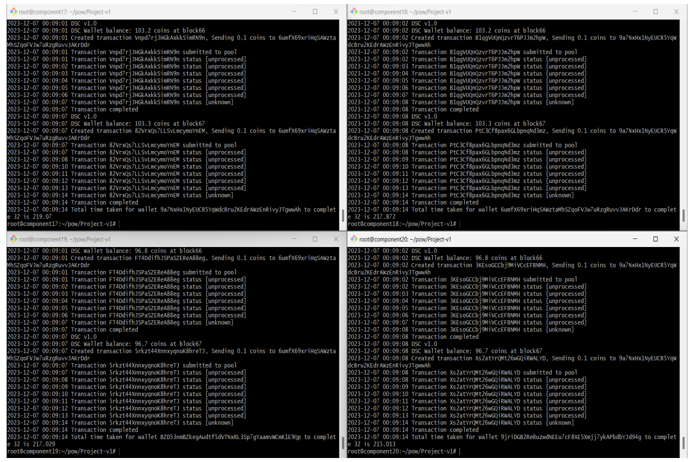
4.1.2 Throughput

1. 1 wallet
   throughput = 128000/118.182 = 1083 transactions/sec

2. 2 wallet

   throughput = (64000/113.007 + 64000/113.1) = 1131 transactions/sec

3. 4 wallet



(a) 4 wallets - 1-2-3-4

Figure 13: 4 wallets - latency - pow

throughput = (32000/114.677 + 32000/114.937 + 32000/115.536 + 32000/115.562)/4 = 1111.25 transactions/sec

4. 8 wallet

   throughput = (16000/111.951 + 16000/112.043 + 16000/112.224 + 16000/112.256 + 16000/112.373 + 16000/112.328 + 16000/112.734 + 16000/112.954) = 1140.2 transactions/sec

4.2 Proof of Memory Evaluation

4.2.1 Latency

1. 1 wallet
   latency = 1212.143/128 = 9.46 sec/transaction

2. 2 wallet

   latency = (572.957/64+573.002/64)/2 = 8.9 sec/transaction

3. 4 wallet

   latency = (331.961/32+331.91/32+331.9/32+331.9/32)/4 = 10.3 sec/transaction

4. 8 wallet

   latency = (189.476/16+189.417/16+189.41/16+189.38/16+189.55/16 +189.52/16+189.47/16+189.50/16)/8 = 11.3 sec/transaction

4.2.2 Throughput

1. 1 wallet
   throughput = 128000/183.301 = 699.4 transactions/sec

2. 2 wallet

   throughput = (64000/193.348 + 64000/193.594)/2 = 661.9 transactions/sec

3. 4 wallet

   throughput = (32000/145.5 + 32000/145.7 + 32000/146.1 + 32000/146.19)/4 = 877.7 transactions/sec

4. 8 wallet
   throughput = (16000/198.18 + 16000/198.659 + 16000/112.224 + 16000/198.625 + 16000/198.732 + 16000/198.826 + 16000/198.913 + 16000/198.947) = 647.1 transactions/sec

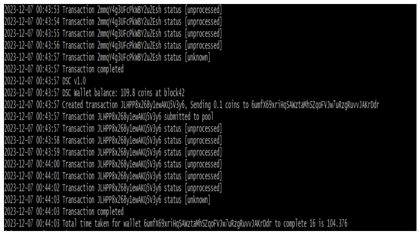
5 Conclusions

- average proof of work latency = 7.09 sec/transaction
- average proof of work throughput = 1116.25 transactions/sec
- average proof of memory latency = 9.99 sec/transaction
- average proof of memory throughput = 771.72 transactions/sec

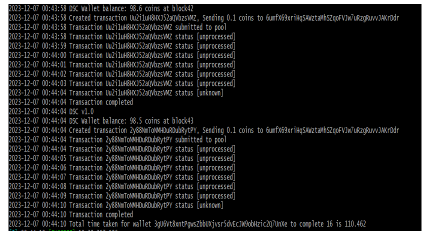
In proof of memory, the substantial 40-second duration required for hash generation during the initial phase directly impacts its efficiency. This prolonged time in generating hashes contrasts with the quicker computational processes in proof of work, leading to a notable efficiency gap between the two consensus mechanisms.

To address the time synchronization issue among components, a signaling mechanism was implemented to initiate execution. When the genesis block is generated, it dispatches a signal to the metronome, which subsequently relays signals to all registered validators. These validators then retrieve the previous hash from the blockchain and proceed with their tasks. Meanwhile, the metronome pauses for six seconds before resuming. Upon awakening, if a validator completes its task, it transmits a signal, prompting the creation of a block. In the absence of a winning validator, the metronome itself generates a block and resumes waiting for a signal from the blockchain. Whenever a block is appended to the blockchain, a signal is relayed to the metronome. Validators, too, await signals from the metronome after each task completion.

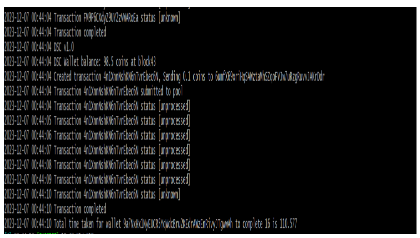
The code's design demonstrates a solid architecture, executing functions precisely as planned. It operates smoothly without encountering any major hurdles, indicating a well-thought-out approach that aligns seamlessly with the intended functionalities.



(a) 8 wallets - 1

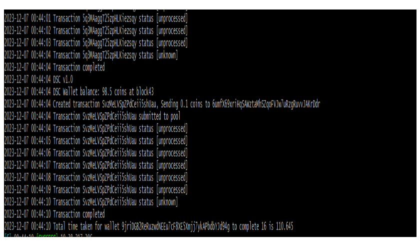


(b) 8 wallets - 2

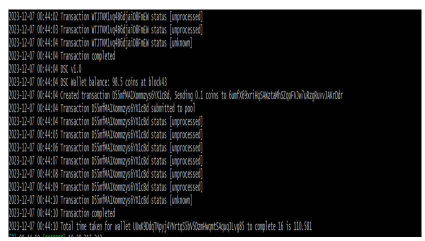


(c) 8 wallets - 3

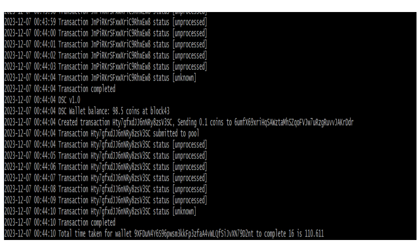
(d) 8 wallets - 4



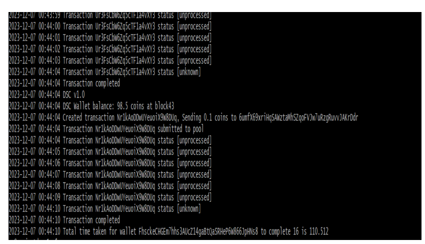
(e) 8 wallets - 5



(f) 8 wallets - 6



(g) 8 wallets - 7



(h) 8 wallets - 8

Figure 14: 8 wallets - latency - pow

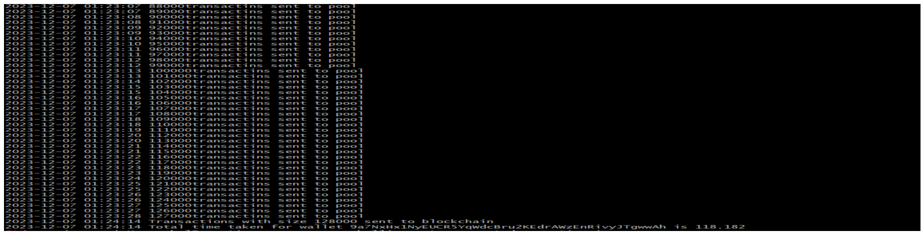
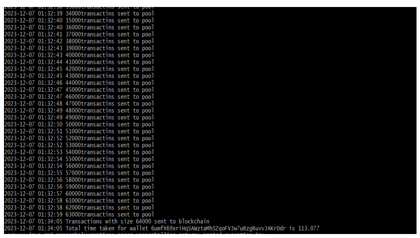
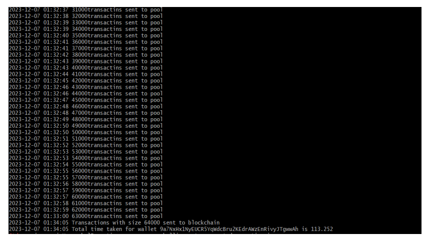


Figure 15: pow-throughput-1 wallet

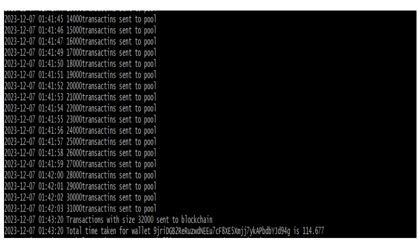


(a) 2 wallets - 1

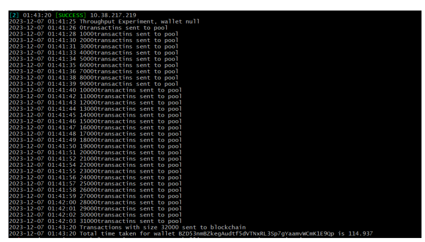


(b) 2 wallets - 2

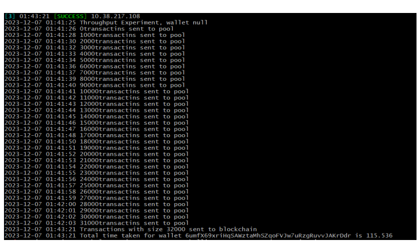
Figure 16: 2 wallets - throughput - pow



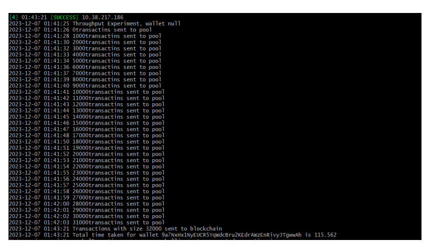
(a) 4 wallets - 1



(b) 4 wallets - 2

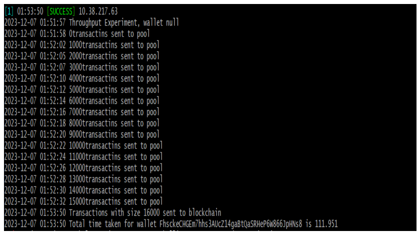


(c) 4 wallets - 3

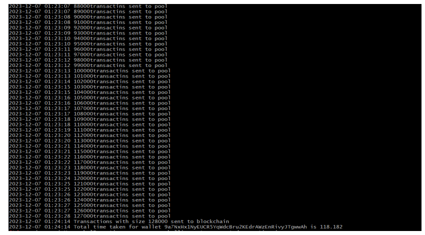


(d) 4 wallets - 4

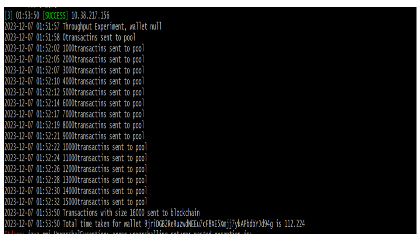
Figure 17: 4 wallets - throughput - pow



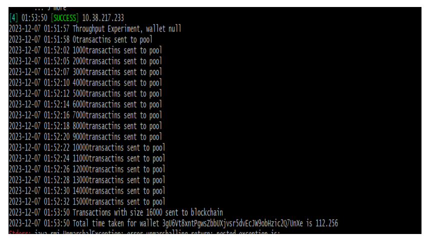
(a) 8 wallets - 1



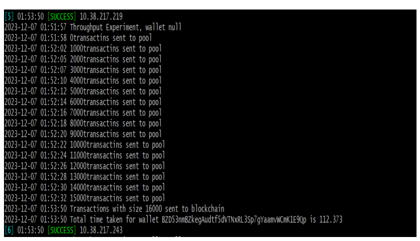
(b) 8 wallets - 2



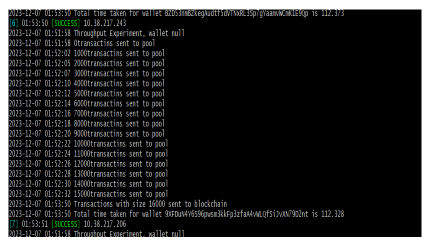
(c) 8 wallets - 3



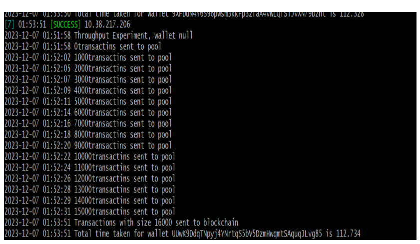
(d) 8 wallets - 4



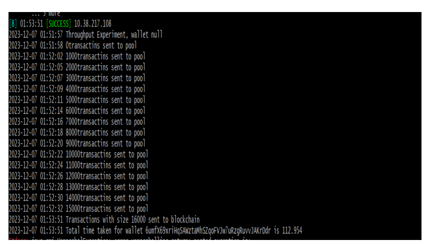
(e) 8 wallets - 5



(f) 8 wallets - 6



(g) 8 wallets - 7



(h) 8 wallets - 8

Figure 18: 8 wallets - throughput - pow

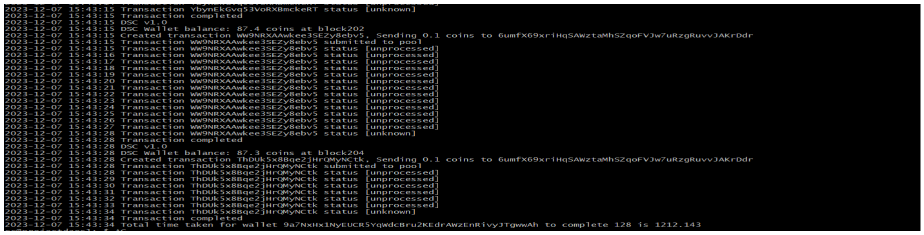
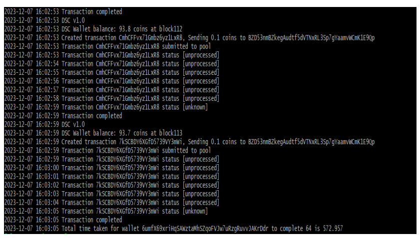
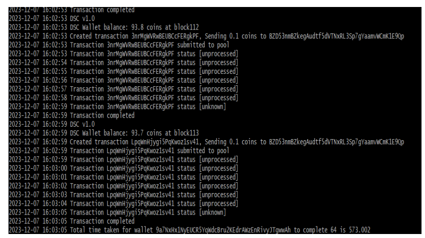


Figure 19: pom-latency-1 wallet

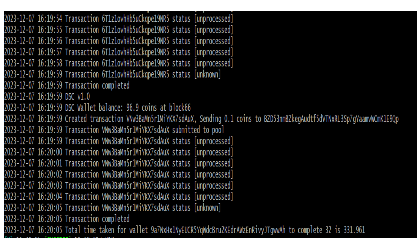


(a) 2 wallets - 1

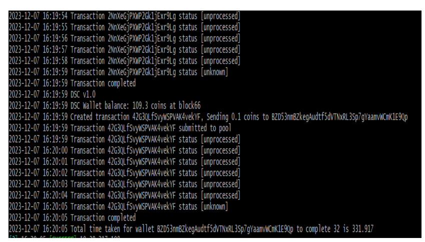


(b) 2 wallets - 2

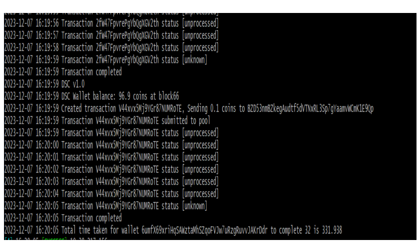
Figure 20: 2 wallets - latency - pom



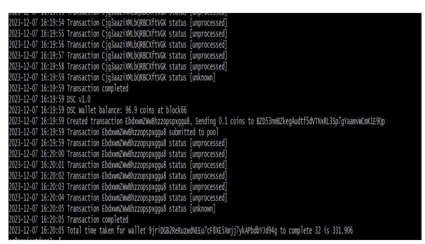
(a) 4 wallets - 1



(b) 4 wallets - 2

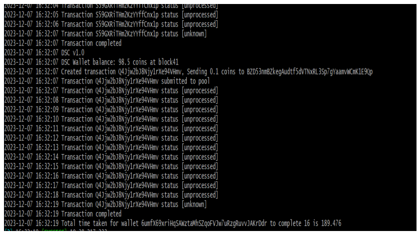


(c) 4 wallets - 3

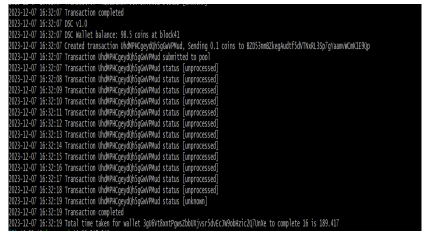


(d) 4 wallets - 4

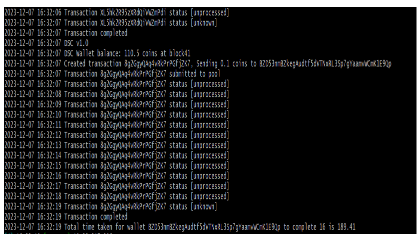
Figure 21: 4 wallets - latency - pom



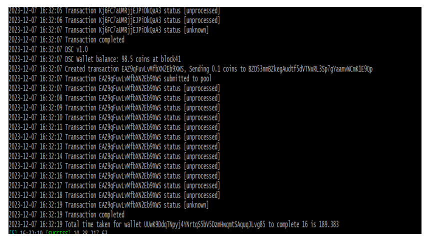
(a) 8 wallets - 1



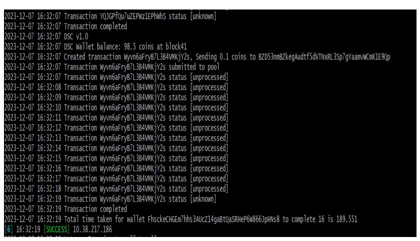
(b) 8 wallets - 2



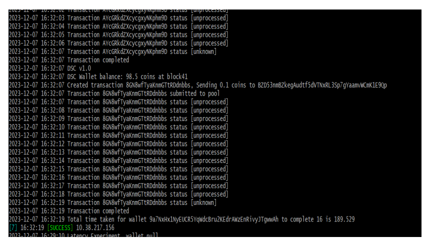
(c) 8 wallets - 3



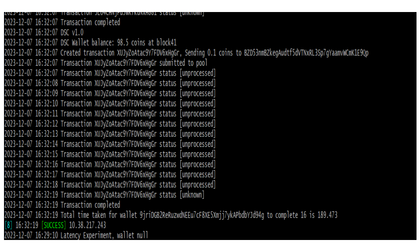
(d) 8 wallets - 4



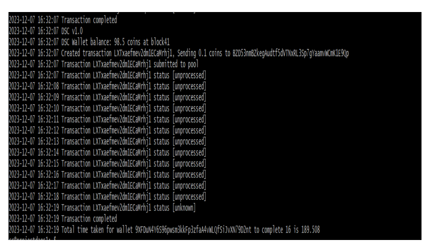
(e) 8 wallets - 5



(f) 8 wallets - 6



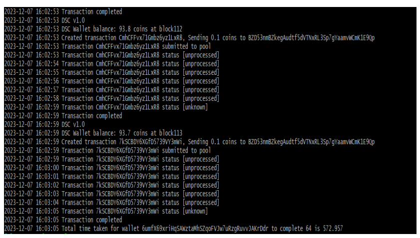
(g) 8 wallets - 7



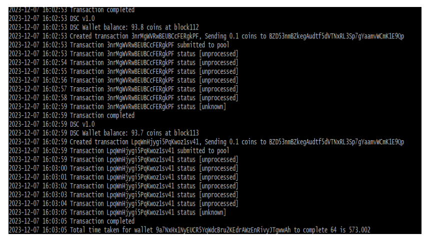
(h) 8 wallets - 8

Figure 22: 8 wallets - latency - pom

Figure 23: pom-throughput-1 wallet

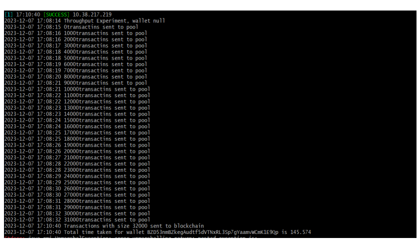


(a) 2 wallets - 1

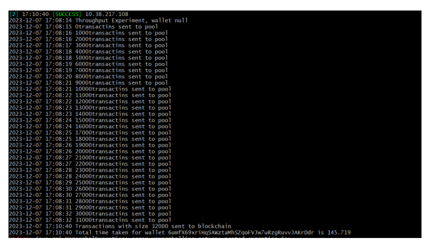


(b) 2 wallets - 2

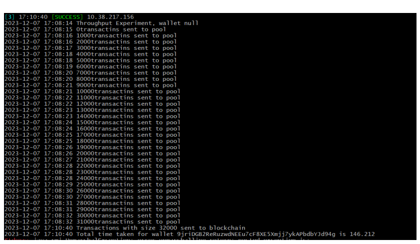
Figure 24: 2 wallets - throughput - pom



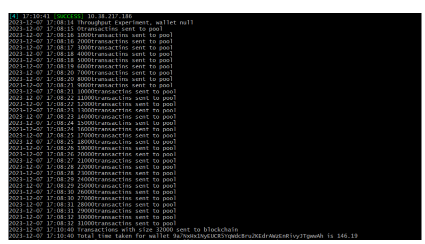
(a) 4 wallets - 1



(b) 4 wallets - 2

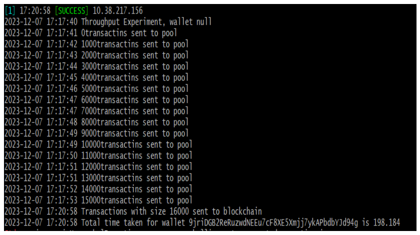


(c) 4 wallets - 3

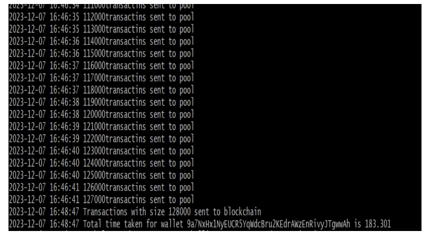


(d) 4 wallets - 4

Figure 25: 4 wallets - throughput - pom



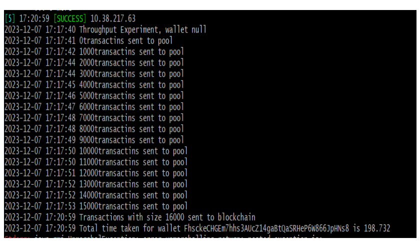
(a) 8 wallets - 1



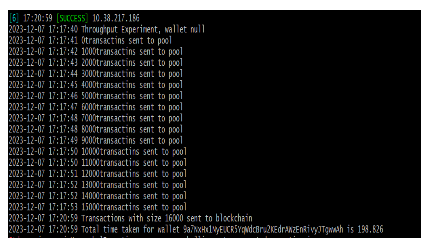
(b) 8 wallets - 2

(c) 8 wallets - 3

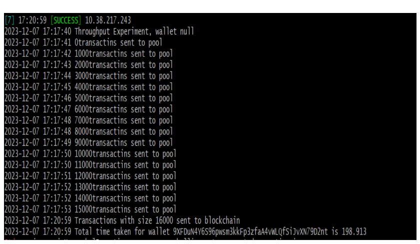
(d) 8 wallets - 4



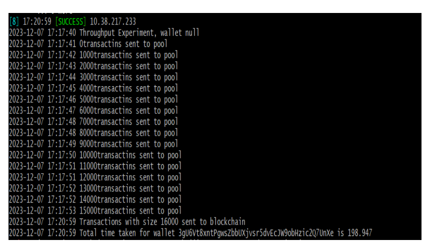
(e) 8 wallets - 5



(f) 8 wallets - 6



(g) 8 wallets - 7



(h) 8 wallets - 8

Figure 26: 8 wallets - throughput - pom