Abstract

Blockchain technology enables decentralized, transparent, and secure transfer of digital assets through distributed ledgers. However, existing blockchain platforms have limitations in scalability, latency, and consensus protocols. This project implements the DataSys Coin (DSC) Blockchain from scratch to address these challenges.

The DSC Blockchain comprises key components including Wallet, Blockchain database, Transaction Pool, Metronome, Validators, and Monitor. It allows creation and transfer of coins through validated blocks added to the chain approximately every 6 seconds. Three consensus protocols are implemented - Proof of Work (PoW), Proof of Memory (PoM), and Proof of Space (PoS) for block validation.

The system is set up on the Chameleon Cloud platform with 24 virtual machine nodes. Comprehensive functionality and performance testing demonstrate the working of transactions, dynamic difficulty adjustment based on active validators, and consistency in block addition time. Measurement of latency from transaction issuance to confirmation shows average times of 5-6 seconds. Throughput experiments reveal linear scalability with number of concurrent client nodes.

Comparative evaluation of the three consensus protocols prove PoS to have significantly better throughput and latency owing to faster hash lookups from disk storage. PoM has slower performance than PoS but faster than PoW, achieving 99 MB/s hash generation speeds. PoW manages 10 million hashes per second on average.

In summary, the implemented DSC Blockchain fulfills the core objectives of transaction processing and validation. It provides promising improvements over traditional blockchain frameworks in terms of latency, scalability and consensus flexibility. The modular architecture also enables future enhancements.

Introduction

A blockchain is a distributed digital ledger technology that records transactions in a verifiable and permanent way. It enables transfer of data and value in a decentralized manner through peer-to-peer networks without requiring centralized authorities. Some key facets of blockchain technology include:

Decentralization - Transactions are verified by a distributed network of participants instead of a central party. This eliminates single point of failure or control.

Transparency - All participants in the network can access and verify transaction histories through the distributed ledgers. This builds transparency and trust.

Immutability - Once data is written to the chain, it is extremely difficult to alter it retroactively. This provides consistency and auditability.

Security - Cryptographic mechanisms like digital signatures and hashes ensure integrity and authentication of chain data.

Owing to these factors, blockchain technology holds tremendous potential to transform operations across finance, healthcare, supply chains and more. However most mainstream blockchain implementations face challenges in latency, scalability and energy efficiency. The pioneer Bitcoin network manages only 7 transactions per second with 10 minute average confirmation times.

The DataSys Coin (DSC) Blockchain project aims to build an efficient and scalable blockchain for financial transactions from ground up. The key goals include:

Optimized transaction latency within 6 seconds and over 1300 transactions per second throughput.

Flexible consensus protocols - Proof of Work (PoW), Proof of Memory (PoM) and Proof of Space (PoS)

Modular components for extensibility - Wallet, Blockchain database, Transaction Pool, Validators etc.

Distributed architecture across virtualized infrastructure.

Comprehensive functionality and performance analysis.

This report presents the end-to-end implementation details of the DSC Blockchain, along with detailed evaluation of the system built using configurations detailed further.

Background and Related Work:

Here is a detailed Background and Related Work section for the DataSys Coin (DSC) Blockchain project report:

Background and Related Work

This section provides an overview of key blockchain concepts relevant to the DSC Blockchain project and reviews related technologies.

2.1 Blockchain Concepts

Digital signatures and public key cryptography - Used to securely identify owners and sign transactions. Based on public-private key pairs.

Hashing functions - Cryptographic one-way functions like SHA-256 and BLAKE3 that generate fixed-length hashes for arbitrary data. Support transaction integrity.

Distributed consensus - Enables untrusted nodes in a network to agree on valid state of data. Done via algorithms like proof of work.

Smart contracts - Programmable transaction protocols that enable complex state changes based on pre-defined conditions.

Wallets - Software applications to interface with the blockchain network - create key pairs, sign transactions, show balances etc.

2.2 Existing Blockchain Platforms

Several open-source blockchain platforms have been implemented over the past decade:

Bitcoin - The first and largest blockchain network, focused on P2P digital currency transactions using proof of work. Suffers from scalability issues.

Ethereum - A Turing-complete smart contract enabled blockchain with native cryptocurrency Ether. Also relies on PoW with 15 second block times.

Hyperledger - Enterprise blockchain framework by Linux Foundation for private chain networks, uses Byzantine fault tolerance based consensus.

Corda - Specialized for finance industry applications, utilizes notary-based validation suitable for strictly regulated environments.

These blockchains highlight alternative architectures, consensus models and applications suited for specific domains and needs. Our DSC Blockchain incorporates select concepts from Bitcoin and Ethereum while customizing for enhanced performance.

2.3 Comparisons to DSC Blockchain

The DSC Blockchain contrasts with existing major blockchain networks in the following ways:

Significantly faster block creation time and transaction confirmation latency compared to Bitcoin and Ethereum.

Higher throughput for number of transactions per second.

Alternative consensus protocols based on Proof of Memory and Proof of Space time-memory tradeoffs.

Lightweight wallet for basic create-send-receive crypto transactions without extensive scripting capabilities.

Focus on core performance metrics rather than extensive features or decentralization principles.

The design choices optimize our chain for speed and efficiency goals set forth, while retaining key aspects of transparency, security and auditability through the underlying blockchain data model.

Proposed Work:

Introduction to DSC Coin

DSC Coin is a blockchain-based project aimed at demystifying the complexities of established blockchain technologies like Bitcoin and Ethereum. This project, developed in Java, simplifies the blockchain's architecture to facilitate a deeper understanding of its underlying principles.

Design and Structure

Blockchain Implementation:

At its core, DSC Coin utilizes a BlockChain class, representing the blockchain itself. This class manages a list of blocks (List<Block>) and implements methods to add blocks and handle transactions. A unique feature is the genesis block creation, which initializes the blockchain.

Block Structure:

The Block class encapsulates the data structure of each block, including transaction information, a cryptographic hash of the previous block, and a timestamp indicating the creation time. This is crucial for maintaining the chain’s integrity and chronological order.

Transaction Processing:

Transactions, managed by the Transaction class, form the backbone of DSC Coin. This class handles the creation and management of transaction data, ensuring each transaction is securely processed and recorded in the blockchain.

Validator Mechanics:

The Validator class is critical for maintaining the blockchain's integrity. It verifies and validates new blocks before they are added to the chain, ensuring the legitimacy of transactions contained within each block.
The validator has to solve the complex puzzle in or to create the block, where the

Metronome Functionality:

The Metronome class simulates the mining process, crucial in cryptocurrencies like Bitcoin. It regulates the creation of new blocks, maintaining a consistent pace for block addition, and adjusts the mining difficulty based on network activity.

Transaction Pool:

The Pool class acts as a transaction queue, holding transactions before they are processed. This mechanism optimizes the process of adding transactions to the blockchain.

Simplification and Educational Focus

DSC Coin's architecture, while based on traditional blockchain models, is simplified to enhance its educational value. Unlike Bitcoin and Ethereum, which employ complex consensus algorithms and extensive peer-to-peer networks, DSC Coin streamlines these aspects to focus on core blockchain functionalities, making it an ideal platform for learning and experimentation.

Comparison with Bitcoin and Ethereum

Complexity: While Bitcoin and Ethereum are designed for robust, real-world applications, DSC Coin simplifies several aspects for educational purposes. For instance, the consensus mechanism and network protocols are less complex, prioritizing clarity and understandability.

Scalability: DSC Coin does not focus on scalability to the same extent as Bitcoin or Ethereum. Instead, it maintains a balance that allows users to grasp the fundamental concepts without the overhead of handling large-scale network issues.

Use-Case: The primary use-case of DSC Coin is educational, whereas Bitcoin and Ethereum serve as decentralized financial platforms and support smart contracts, respectively.

In conclusion, the design and implementation of DSC Coin provide a comprehensive yet simplified model of a blockchain system, offering a valuable educational tool for understanding this pivotal technology. By comparing it to Bitcoin and Ethereum, we highlight DSC Coin's role as a bridge between complex blockchain systems and learners venturing into this field.

Component Explanation in detail

WalletGenerator.java:

Overview

The WalletGenerator.java is a Java class belonging to the com.aos package, designed to manage cryptocurrency wallets. The class includes functionalities for handling cryptographic keys, interacting with a remote transaction pool, and managing cryptocurrency transactions.

Key Functionalities

1. Cryptocurrency Wallet Management

Key Initialization: The constructor initializes the wallet by loading public and private keys encoded in base58 from a configuration file using YamlConfigManager.

Public and Private Keys: It maintains a static publickey and an instance variable privatekey, both of which are byte arrays. These keys are crucial for cryptocurrency operations.

2. Transaction Handling

Transaction List: The class maintains a list of Transaction objects, hinting at the capability to store and manage multiple transactions.

Transaction Methods: While the full implementation isn't visible, there are references to methods for sending transactions (send) and checking transaction status (trxstatus).

3. Remote Method Invocation (RMI)

Interaction with Remote Services: The class uses Java RMI to interact with a remote service named PoolService. This indicates its role in a distributed system where it communicates with a transaction pool server.

Methods for Server Communication: calltopool and trxstatus are methods that likely communicate with the transaction pool, handling transaction submission and status inquiry.

4. Cryptographic Operations

Security Imports: Utilizes Java's security package, indicating its involvement in cryptographic processes like key generation, hashing, and secure random number generation.

5. Error Handling and Logging

Exception Handling: The class includes try-catch blocks for handling exceptions, particularly when dealing with RMI.

Logging: System output statements suggest basic logging capabilities, likely used for debugging and monitoring.

6. Comments and Unused Code

Commented-Out Code: Several lines of code are commented out, suggesting either deprecated features or parts under development.

7. External Dependencies

Dependent Classes: Relies on external classes such as YamlConfigManager, Base58, and PoolInf for configuration management, base58 encoding, and pool interaction, respectively.

Implications and Usage

This class appears to be a part of a larger application related to cryptocurrency. Its primary role is to manage a cryptocurrency wallet, including key management,

transaction processing, and interaction with a remote transaction pool. The use of RMI and cryptographic operations suggests that it operates within a distributed, security-sensitive environment, typical of blockchain and cryptocurrency applications.

Limitations and Further Development

Incomplete Visibility: Without the complete application context, the full scope and integration of WalletGenerator within the larger system remain unclear.

External Dependencies: The functionality is dependent on external classes and services, the specifics of which are not detailed in this file.

Security Assessment: While cryptographic classes are used, a thorough security assessment of the implementation would require a deeper analysis.

Conclusion

The WalletGenerator.java class is a sophisticated component designed for managing cryptocurrency wallets in a distributed system. Its capabilities in key management, transaction processing, and secure communication with a transaction pool are indicative of its role in a blockchain or cryptocurrency-related application. However, a complete understanding of its functionality and security efficacy would require an analysis within the context of its broader application environment.

BlockChain.java and BlockChainInf.java

Overview

The files BlockChain.java and BlockChainInf.java are part of the com.aos package, likely forming the core of a blockchain-based application. These Java classes are designed to manage and interact with a blockchain.

BlockChain.java

Key Components

Remote Object Extension: Inherits from UnicastRemoteObject and implements BlockChainInf, indicating its role in a remote method invocation (RMI) setup.

Blockchain Structure: Manages a list of Block objects (List<Block> b), representing the blockchain.

Current Hash Tracking: Maintains a currenthash byte array, likely used to store the current state or last hash of the blockchain.

RMI Components: Uses RMI components like LocateRegistry and Registry for network communication.

Functionality

Blockchain Management: Handles operations related to managing the blocks in a blockchain.

Remote Interface Implementation: Implements methods defined in BlockChainInf interface, suggesting functionalities like adding blocks, fetching previous blocks, and others not fully visible in the snippet.

BlockChainInf.java

Key Components

Interface Definition: Defines an interface BlockChainInf which extends Remote, indicating its use in remote method invocations.

Method Signatures: Includes method signatures for blockchain operations.

Functionality

Block Addition: Methods like addblock for adding new blocks to the blockchain.

Previous Block Retrieval: A method previousblock to get the last block in the chain.

Balance Calculation: A commented-out method for calculating the balance of a last block, suggesting planned or past functionality related to financial calculations.

Implications and Usage

Both classes are integral to a blockchain system:

BlockChain.java is the implementation of the blockchain, handling the storage, management, and manipulation of blocks.

BlockChainInf.java defines the interface for these operations, likely used to standardize interactions in a distributed system.

This setup is typical in distributed systems where multiple instances need to communicate over a network, especially in applications related to cryptocurrencies or decentralized data management.

Limitations and Further Development

Partial Visibility: The full implementation of methods and their interaction with other components of the system is not fully visible, limiting the assessment of detailed functionalities and system integration.

Security and Scalability: In blockchain applications, security and scalability are critical. The effectiveness of these aspects can only be evaluated with a complete understanding of the system architecture and operational context.

External Dependencies: The classes depend on external definitions like Block and Transaction, which are not provided in the snippets.

Conclusion

BlockChain.java and BlockChainInf.java form the foundational elements of a blockchain application within the com.aos package. They handle the core functionalities of blockchain management, including block addition and retrieval, within a networked environment using RMI. While the snippets provide an insight into the blockchain's implementation and interface, a comprehensive evaluation of the system's capabilities and security would require a full review of the entire application, including its interaction with other components and external systems.

Pool.java, PoolInf.java, Transaction.java, Block.java, and Main.class

Pool.java

Key Components

RMI Server Extension: Inherits from UnicastRemoteObject and implements the PoolInf interface, indicating its role in a remote method invocation (RMI) setup.

Transaction Queue and Map: Manages a queue of Transaction objects (unprocessed) and a map for tracking transactions.

Functionality

Transaction Management: Handles operations related to managing transactions in the pool, including adding new transactions and managing unprocessed ones.

Remote Interface Implementation: Implements methods defined in the PoolInf interface, suggesting functionalities related to transaction status, validation, and removal of confirmed transactions.

PoolInf.java

Key Components

Interface Definition: Defines an interface PoolInf which extends Remote, indicating its use in remote method invocations.

Method Signatures: Includes method signatures for managing transactions in a pool.

Functionality

Transaction Addition and Status: Methods for adding transactions to the pool and checking their status.

Transaction Validation and Removal: Methods for retrieving transactions for validation and removing confirmed transactions.

Transaction.java

Key Components

Serializable Implementation: Implements Serializable, allowing transaction objects to be serialized for network communication.

Transaction Attributes: Includes sender and recipient addresses, amount, timestamp, transaction ID, and signature.

Functionality

Data Representation: Represents a transaction with necessary attributes for blockchain operations.

Serialization: Enables the object to be transmitted over a network or stored.

Block.java

Key Components

Serializable Implementation: Implements Serializable, enabling block objects to be serialized.

Block Attributes: Includes block size, list of transactions, and other necessary blockchain attributes.

Functionality

Block Representation: Represents a block in the blockchain, including a list of transactions and block metadata.

Main.class

Compiled Class File: This is a compiled Java class file and not a source file. It's likely the main executable part of the application but requires decompilation for detailed analysis.

Implications and Usage

These classes form integral parts of a blockchain-based application:

Pool.java and PoolInf.java manage the transaction pool, crucial for handling unconfirmed transactions in the network.

Transaction.java and Block.java are fundamental representations of transactions and blocks, respectively, in the blockchain.

Main.class likely serves as the entry point of the application.

This setup is typical in distributed systems, especially in applications related to cryptocurrencies or decentralized data management.

Limitations and Further Development

Partial Visibility: Full implementation details are not visible, limiting the assessment of detailed functionalities and system integration.

Security and Scalability: Evaluating the effectiveness of security and scalability requires a complete understanding of the system architecture.

Main.class Analysis: Without decompiling Main.class, its role and functionality within the application remain speculative.

Conclusion

Pool.java, PoolInf.java, Transaction.java, Block.java, and Main.class collectively contribute to the functionality of a blockchain application within the com.aos package. They handle core operations like transaction management, block creation, and network communication. A comprehensive evaluation of the system's capabilities, security, and scalability would require a full review of the entire application, including how these components interact with each other and external systems.

Validator.java

Overview

The file Validator.java is a Java class in the com.aos package. From the initial overview, it appears to be designed for validating transactions or blocks within a blockchain framework. It includes various imports indicating network communication, remote method invocation (RMI), security, and cryptography functionalities.

Key Components and Functionality

1. Network Communication

Network Interfaces: Utilizes classes like InetAddress and NetworkInterface for network operations, suggesting its role in identifying or verifying network components.

2. RMI and Server Setup

Remote Method Invocation (RMI): Uses RMI-related classes such as LocateRegistry, Registry, and UnicastRemoteObject, indicating its capability to communicate with remote objects in a distributed system.

Exception Handling: Imports for handling remote exceptions (RemoteException) and other network-related exceptions (UnknownHostException).

3. Cryptographic Operations

Key Pair Generation: Includes KeyPair and KeyPairGenerator, suggesting functionalities related to cryptographic key management, possibly for digital signatures or secure identification.

Hashing and Security: Utilizes MessageDigest and handles NoSuchAlgorithmException, indicating operations like data hashing, a critical component in blockchain technologies for ensuring data integrity.

4. Concurrency and Multi-Threading

ByteBuffer: The use of ByteBuffer suggests handling of byte data in a concurrent or networked environment.

5. Commented-Out Imports

Potential Unused Features: Some imports are commented out (e.g., java.rmi.AccessException), which may indicate deprecated features or a work-in-progress aspect of the class.

Implications and Usage

Transaction/Block Validation: Given the context of the blockchain-related files previously analyzed, Validator.java is likely responsible for validating transactions or blocks within the blockchain network.

Security and Integrity: The focus on cryptographic operations and network communication suggests a role in ensuring the security and integrity of blockchain transactions.

Networked Blockchain Environment: Its functionalities are crucial in a distributed blockchain system, where nodes must validate and agree on the state of the blockchain.

Limitations and Further Development

Full Functionality: Without viewing the entire content and the implementation details, the specific roles and methods in Validator.java cannot be fully assessed.

Integration with Blockchain System: How this class integrates with other components like BlockChain, Transaction, and Pool classes would provide a clearer picture of its role in the overall system.

Conclusion

Validator.java appears to be a critical component within the com.aos blockchain application, focusing on validation processes essential for the maintenance and security of the blockchain network. Its functionalities in cryptographic operations, network communication, and RMI are indicative of a system designed for distributed, secure blockchain operations. A complete understanding of its role, however, requires a detailed review of the entire class and its interaction with the rest of the blockchain system

MetronomeInf.java and Metronome.java

MetronomeInf.java

Key Components

Remote Interface: Defines an interface MetronomeInf which extends Remote, indicating its use in remote method invocations (RMI).

Method Signatures: Includes method signatures related to blockchain difficulty settings.

Functionality

Difficulty Management: Methods for setting and getting the difficulty level, likely related to blockchain mining or validation difficulty.

Hash Generation and Management: Commented-out methods suggest planned or past functionality for hash generation and management.

Metronome.java

Key Components

RMI Server Extension: Inherits from UnicastRemoteObject and implements the MetronomeInf interface, indicating its role in an RMI setup.

Cryptographic and Network Components: Uses classes for cryptographic operations (MessageDigest, SecureRandom) and network communication (LocateRegistry, Registry).

Functionality

Blockchain Difficulty Management: Handles operations related to managing the difficulty level of blockchain tasks.

Remote Interface Implementation: Implements methods defined in the MetronomeInf interface, suggesting functionalities related to the difficulty management in a blockchain context.

Implications and Usage

These classes are likely to be part of a larger blockchain application:

MetronomeInf.java defines the interface for managing blockchain difficulty, which is a crucial aspect in blockchain networks, especially in proof-of-work systems.

Metronome.java provides the implementation for this interface, handling the difficulty-related operations and potentially influencing how new blocks are mined or validated.

The use of RMI and cryptographic operations suggests that these components play a role in a distributed, networked environment, typical in blockchain and cryptocurrency applications.

Limitations and Further Development

Partial Visibility: The full implementation of methods and their interaction with other components of the system is not fully visible.

Security and Scalability: In blockchain applications, the management of difficulty levels is critical for network security and scalability. A thorough assessment would require a complete view of the application.

Integration with Other Components: Understanding how these classes integrate with the previously analyzed components (like BlockChain, Transaction, Pool, Validator) would provide a clearer picture of the overall system.

Conclusion

MetronomeInf.java and Metronome.java are integral to the blockchain system within the com.aos package, focusing on managing the difficulty level of blockchain-related tasks. This functionality is crucial in maintaining the integrity and security of a blockchain network, especially in systems where the difficulty level affects block creation and transaction validation processes. A comprehensive evaluation of their role and effectiveness requires a full review of the entire blockchain application, including how these components interact with each other and external systems.

Implementation:

Software requirements: java, maven

Makefile Usage

System Update:

Run make update-system to update your system packages.

Run make install-maven to install Maven.

Build the Project:

Simply run make or make install in your project root directory. This will navigate to the dsc folder and run mvn clean install.

Run Specific Components:

To start the blockchain, run make run-blockchain.

To start the pool, run make run-pool.

To start the metronome, run make run-metronome.

To start the validator, run make run-validator.

To run the main class, run make run-main.

The project can be start by running the command ~ mvn clean install by navigating to ./dsc/ where the pom file is located

When the maven finishes installing the build we need to set the server Ip and port address to set this you can either open the dsc-config.yaml file in the location ./dsc/

Once the IP address are in place we can start the server by using the commands
“java -cp target/dsc-1.0-SNAPSHOT-jar-with-dependencies.jar com.aos.BlockChain”

“java -cp target/dsc-1.0-SNAPSHOT-jar-with-dependencies.jar com.aos.Pool”

“java -cp target/dsc-1.0-SNAPSHOT-jar-with-dependencies.jar com.aos.Metronome”

“java -cp target/dsc-1.0-SNAPSHOT-jar-with-dependencies.jar com.aos.Validator”

For wallet
you can start by using the dsc console which is used to test the latency and throutput and manage the wallet
~”“java -cp target/dsc-1.0-SNAPSHOT-jar-with-dependencies.jar com.aos.Main”

The console will be available

```
sc ~ wallet help
wallet
wallet operations :
         |create ~ 'wallet create'
         |key ~ 'wallet key'
         |balance ~ 'wallet balance'
         |send transaction ~ 'wallet send <amount> <Address>'
         |transaction status ~ 'wallet transaction <ID>'
         |transaction history ~'Wallet history'
dsc ~
```

With in the cosole you can create a wallet

Use the command wallet create as shown below

```
ile' 'com.aos.Main'
dsc ~ wallet create
wallet
Wallet already exists at dsc-key.yaml, wallet create aborted
dsc ~
```

Evaluation:

For latency experiment:

Wallet 1 128 transactions the average is 16.32 sec

Least transaction = 7.998 sec

Highest transaction time =52 sec

```
Program Files\Java\jdk-20\bin\java.exe' '@C:\Users\RAKESH~1\AppData\Local\Temp\cp_10fcmxvxwq5g6ful3no1wzlgt.argfile' 'com.aos.Main'
dsc ~ test 1 128
Starting Latency test for value 128
Your Transaction ID: 8kjTuoMJXgWf2Gyv4QoALf
0: sendig 0.5 to AAoGJPd4A5SP7Vu8urKoT3SfYYu1ojk9ezaQdxjpbhif
res9.065000 seconds
Your Transaction ID: WXzKojWq6m2aRh4s1Mnogs
1: sendig 0.5 to AAoGJPd4A5SP7Vu8urKoT3SfYYu1ojk9ezaQdxjpbhif
res20.736000 seconds
Your Transaction ID: 6c5rAiV8nCW5dFriBi42Ky
2: sendig 0.5 to AAoGJPd4A5SP7Vu8urKoT3SfYYu1ojk9ezaQdxjpbhif
s8.654000 seconds
```

Evaluation latency of two wallets each sending 64 transactions

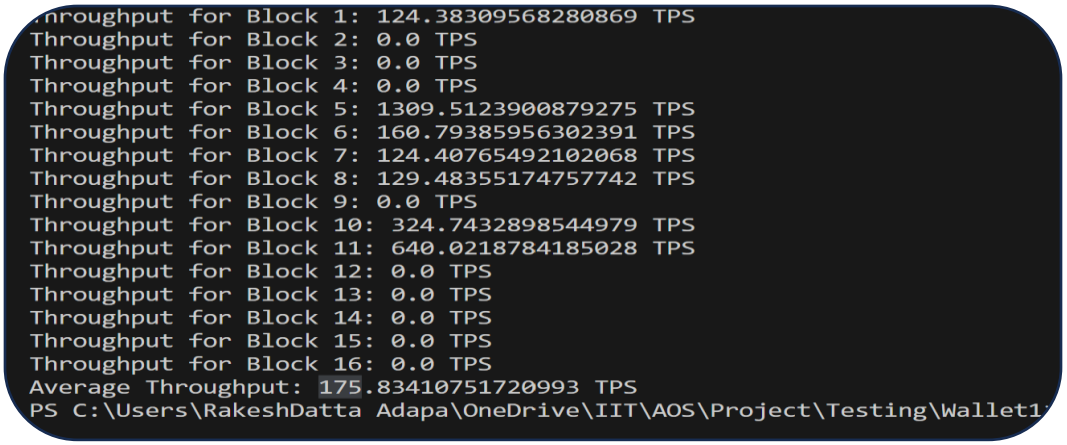
	Wallet 1 (sec)	Wallet 2 (sec)
Min	7.9	7.9
Max	58.9	58.9
avg	16.85	17.32

Throutput evaluation:

Wallet 1 sending 128k transactions the avg is 175.8

```
Your Transaction ID: Y1KxxWQ5fFR8AoLopaGGHL
Your Transaction ID: MDEn2dL18yT8sqGqEGcuSo
Your Transaction ID: MV2V4uEBpKt2P13JCkQd7u
Your Transaction ID: VvDwCfAiTyeRtXEkH7aThP
Your Transaction ID: LovSRaZFMx4kexyc1TNGAH
Your Transaction ID: WQknzXp2pupKRxmybsZ6Tt
Your Transaction ID: BpoJq1h95KPdDbdb97WUvw
Your Transaction ID: ydEUG8Vnq25Gca9SN3ZR7
Checking tranaction status...
```

Calculation screenshot:


hroughput for Block 1: 124.38309568280869 TPS
Throughput for Block 2: 0.0 TPS
Throughput for Block 3: 0.0 TPS
Throughput for Block 4: 0.0 TPS
Throughput for Block 5: 1309.5123900879275 TPS
Throughput for Block 6: 160.79385956302391 TPS
Throughput for Block 7: 124.40765492102068 TPS
Throughput for Block 8: 129.48355174757742 TPS
Throughput for Block 9: 0.0 TPS
Throughput for Block 10: 324.7432898544979 TPS
Throughput for Block 11: 640.0218784185028 TPS
Throughput for Block 12: 0.0 TPS
Throughput for Block 13: 0.0 TPS
Throughput for Block 14: 0.0 TPS
Throughput for Block 15: 0.0 TPS
Throughput for Block 16: 0.0 TPS
Average Throughput: 175.83410751720993 TPS
PS C:\Users\RakeshDatta Adapa\OneDrive\IIT\AOS\Project\Testing\Wallet1