CN3530 - Computer Networks

A Miniature Version Of Block Chain

Final Project Report

December 4, 2023

Akriti
ES21BTECH11005

Tammineni Sneha
ES21BTECH11034

B Tanushree Singh
ES21BTECH11009

K Vivek Kumar
CS21BTECH11026

Ch Sai Sampath
ES21BTECH11027

P Sriram
ES21BTECH11023

Contents

1 Introduction

Welcome to this report, offering insights into our project—a miniature representation of a blockchain and Bitcoin. Delve into the details of our endeavor, exploring the key elements and outcomes, as we showcase the innovation in blockchain technology within a compact framework.

2 Motivation

Our aim is to construct a working miniature version of the blockchain which highlights on how blockchain's inherent security mechanisms, like block-based storage, mining for validation, public-key cryptography and the change in the cryptocurrency rates, can safeguard data and ensure transparent interactions. This exploration equips us to address critical societal issues and build trust in the digital sphere.

We struggled with real-world challenges, like identifying and rectifying chain branching, through experimentation and innovation. This hands-on approach not only deepens our technical understanding but also cultivates problem-solving skills.

3 Code Requirements

To run the code successfully, make sure to install the following Python libraries:

numpy, pandas, matplotlib, socket, json, cryptography, threading, random, hashlib

You can install these libraries using the following command:

```
$ pip install numpy pandas ...
```

Now you can run the server first, and for a new client just run the client file again and again.

4 Architecture and Working

The implementation were done module wise for easy navigation and readability. We have explained each of them nicely below:

4.1 Client Overview

Client:
In a network, a client is essentially a node that connects to other clients/servers and interacts with them. In our network, a client serves as a participant, joining the network, interacting with other clients, and maintaining a record of network activities.

A client is required to:

- Connect to the server.
- Send and receive transactions.
- Mine blocks (if it's a miner).
- Maintain the local state of the blockchain.
- Display user amounts, check balances, and handle various types of transactions.

Task Execution:
Once a client joins the network, it receives a private key and public key from the server. If the client is already part of the network, the private key is used to log in. Subsequently, the client can send and receive transactions from other clients, and all clients receive messages of any transaction within the network. Clients also play the role of miners for transactions.

Upon receiving a block, a client searches for the chain whose last block's hash value matches the received block's previous hash value, and the new block is connected to that blockchain. The client identifies and removes fake blocks resulting from malpractices, all while continuously updating the account balance.

Key Components of CLIENT Code:

1. **Socket Connection:** Establishes a socket connection with the server when a client joins the network.

2. **Blockchain Initialization and Class Block:** Forms a blockchain and defines the Block class for representing different data fields.

3. **Message Buffer:** Stores transaction messages before they are added to the blockchain.

4. **User Management:** Manages user data and provides keys to new users joining the network.

5. **Mining:** Performs mining to guess a specific hash value and provides proof of work to create blocks.

6. **Checking Buffer and Blockchain:** Checks the message buffer, mines a block, and checks blockchain consistency.

7. **'find_chain' and 'display_blocks':** Functions for finding a specific chain and displaying the blockchain and its branches.

8. **Blockchain Verification:** Normalizes the state of the blockchain, resolves conflicts, and removes shorter branches.

9. **Block Addition:** Adds a received block to the blockchain after verification.

10. **Transaction Verification:** Verifies the integrity of a transaction using a digital signature.

11. **Receive Messages:** Handles incoming messages, including transactions and blocks.

12. **Checking Balance:** Calculates the total balance for a user based on blockchains and the message buffer.

13. **Client Initialization:** Initializes the client by receiving keys, loading the blockchain, and starting a receiving thread.

Modules Used:

1. **Json:** Handles JSON data for users and blockchain information.

2. **Socket:** Establishes socket connections.

3. **Threading:** Creates a separate thread for continuous message reception.

4. **Random:** Generates random nonces for mining.

5. **Hashlib:** Creates hash values, especially using SHA-256, for blocks.

6. **Custom Modules:**

   - **getKeys**
   - **retrieve**

4.2 Mining Overview

Mining:
Mining, a common process in various networks, involves solving mathematical problems, validating transactions, and creating new blocks. The entities performing mining tasks are referred to as miners.

In our network, miners are required to:

- Receive a transaction.
- Guess the hash value puzzle.
- Create new blocks.

Task Execution:
Miners aim to find a random nonce (initially generated in the range [-100000, -1]) that, when combined with other data, produces a hash with a specific pattern (starting with "0000" in this case). If the nonce value is lower than the given difficulty level, the miner is allowed to create a block for the received transaction. This block includes the previous block's hash value, transaction details, and the miner's generated hash value.

Mining Function:
The mining function has been incorporated into our client code, handling the tasks mentioned above when the client functions as a miner. Key components of the mining function include:

1. **Nonce Initialization:** The nonce is initially set to a random integer in the range (-100000, -1). If the stage variable is True (i.e., nonce value $\neq$ required value), the nonce range is adjusted to (-1000000, -100001).

2. **Mining Loop:** The function enters an infinite while loop for the mining process. Inside the loop, it calculates an absolute value based on the nonce.

3. **Data Construction:** The data string represents the new block's data, constructed by concatenating various pieces of information, including the last block's hash value, transaction message, and the absolute value of the nonce.

4. **Hash Calculation:** The SHA-256 hash of the data is calculated using the 'hashlib' module. The resulting hash is stored in the 'current_hash' variable. The current nonce being tried is printed to the console.

5. **Hash Check:** It verifies if the 'current_hash' starts with "0000" and if the nonce is positive. If both conditions are met, the function prints the obtained nonce and hash, and returns them. If not, the nonce is incremented, and the loop continues.

Modules Used:

1. **'random' Module:** Part of the Python Standard Library, used for generating random numbers. The method 'random.randint' generates the random variable nonce.

2. **'hashlib' Module:** Another part of the Python Standard Library, hashlib provides a common interface to secure hash and message digest algorithms. In this code, 'hashlib.sha256(data.encode("utf-8")).hexdigest()' is used to calculate the SHA-256 hash of the data string.

4.3 Storing Clients and Blocks

Our program is designed in such a way that every new client accessing the blockchain initially will be given a public key. Username which can be accessed, viewed by others and a private key which only the client can view initially. When the client wants to enter the block chain, he needs to enter his username and private key which was provided initially, if the entered private key does not match with the initial private key, then the client is not given access to the block chain and an output message 'Key pair verification failed' is printed. This process is done by the program code retrieve.py and functions in it load_blockchain and replace_blockchain using JSON package.

4.4 Login and Re login

We are storing the block chain data in the form of blocks and each block contains the information of 3 transactions and this block chain data is stored in a JSON file. When transactions are successful, they merge to form a block and this block is added to the JSON file. In some cases, if there are any abnormalities in the transactions then the block chain doesn't get updated, and the program will throw an error message 'Error replacing blockchain'. This process is done by the program code getKeys.py and functions in it generate_keys and verify_key_pair using the packages default_backend from cryptography.hazmat.backends, serialization and hashes from cryptography.hazmat.primitives and ed25519 from cryptography.hazmat.primitives.asymmetric which are helpful in generating specific public and private key pair and also for matching a public key with a specific and unique private key.

4.5 Fake Mining

In many cryptocurrencies, miners play a crucial role in creating blocks. However, there is a possibility that a released block might be fake. To address this, the blockchain initiates branching. During branching, blocks are typically connected through the previous hash of the preceding block. In the case of a fake block, it connects using the previous hash of an authentic block in the blockchain.

Subsequently, remaining miners work to remove the fake block by extending the branch of the authentic one. Clients perceive a longer chain as more authentic. The removal of the fake block involves redoing the transactions in reverse, restoring the amounts to their original state. Additionally, any service charge earned through fake mining is also eliminated.

4.6 Hashing

In blockchain systems like Bitcoin, a simple form of proof of work involves finding a nonce (a number used only once) that, when combined with data, produces a hash satisfying certain conditions.

Hashing Data:
In our code, we utilize SHA256 for the hashing process. The line
`hashlib.sha256(data.encode("utf-8")).hexdigest()`
performs the hashing. It takes input data, encodes it in UTF-8 format, and computes the SHA-256 hash. The `hexdigest()` function converts the hash into a hexadecimal string. The resulting hash is stored in the variable `current_hash`.

4.7 Connections

In our system, a block is created in the form
`['TRANSACTION1-TRANSACTION1-TRANSACTION1-PREVIOUS_HASH,HASH']`. Each block is formed with the help of three transactions taken from the transaction buffer. The new block is then created using the `previous_hash` of the current block.

Blocks are stored in a list, and each block is represented as a string. In the event of branching, the blocks that have been printed are stored in a nested list. The smallest list is removed, as discussed.

4.8 Function Calls for a Client

BLK:
This signifies a block in our code. Every block string in our code will start with these letters.

AMT:
Entering this in the terminal reveals the current amount in the account.

BCH:
Indicates that a fake block has been released and is being branched in the blockchain.

WBK:
Denotes the sending of a wrong block.

RAT:
Returns the current rate of Bitcoin currency, utilizing a graph in the code.

TBF:
Used to retrieve the transaction buffer at the moment.

TXN-:
Every transaction happening on the client-side starts with this.

TRANSACTION:
In our code, a transaction has a structure of TXN-client username you are sending amount to-amount that is being sent.

GET:
Provides access to the money present in clients' accounts within the network.

PBK:
Used in the terminal to print the block in our code.

5 Future Scope of Matching Up with Actual Block Chain

The current implementation is a basic blockchain network. To enhance its functionality, several improvements can be made. Firstly we need to strengthen security measures, including secure key management and encrypted communication. We can add a consensus mechanism for decentralized validation of transactions and blocks. We could also bring in smart contracts to make the blockchain more versatile.

Expanding the block structure to include a timestamp, Merkle tree, and nonce for proof-of-work would elevate systems robustness. Storing blockchain data more permanently, improving networking, and making sure checks are solid will help keep everything reliable.

Rigorous testing, comprehensive documentation, and meticulous error handling are vital components for the sustained reliability of the blockchain system, acknowledging the dynamic nature of blockchain technology.

6 Real Life Use Cases of our Project

Our blockchain project isn't just about money; it can do other awesome stuff too! You can use it to keep your health records super safe, make voting fair and clear, and even store your files securely. So, it's not just for buying and selling – it's like a tool for keeping important stuff safe and making sure everything is honest and transparent.

7 Conclusion

While decentralization is not the core focus, our miniature blockchain offers valuable insights into the inner workings of blockchain resilience. We simulate attacks and malpractices, revealing how the system identifies and rectifies chain branching, ensuring data integrity and showcasing the technology's ability to withstand adversity.

This project provided valuable insights into blockchain and cryptocurrencies. The process of translating ideas into actual code was both exciting and challenging. Numerous improvements and better ideation were key contributors to the success of our blockchain project.

Thank You!