**COMP5521**

**Distributed Ledger Technology, Cryptocurrency and E-payment**

**Group Project Report**

**Group 1**

**20001138G Cheuk Man Ting**

**21000018G Li Yat Long**

**21006322G To Ching Wai**

**21014947G Yu Hin Chung Nikko**

**21045374G Zhang ZhanLei**

1. **Introduction**

In order to have an in-depth understanding on how a blockchain system works, we developed a cryptocurrency blockchain system referred to Naivecoin. The programming language is Typescript. In this project, we implemented and enhanced the functions of Naivecoin among several parts - blockchain architecture and design, mining procedure, transactions, networking and storage. More details are introduced and explained in the sections below.

1. **Blockchain Design**

The structure of our blockchain prototype is mostly based on the open source project – Naivecoin, with some modification to achieve the goals of the project [1].

**2.1. Block Structure**

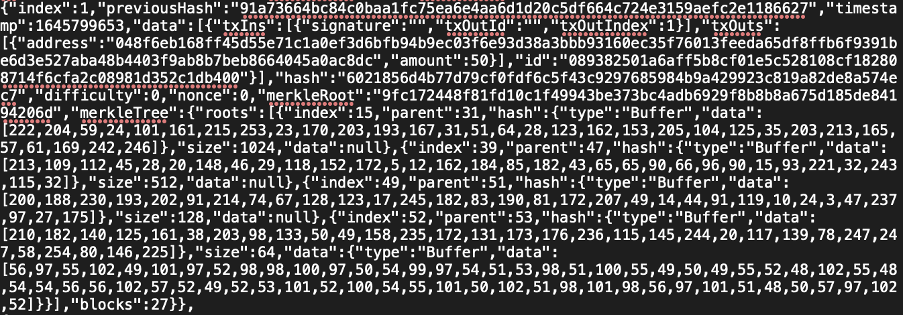


Fig. 1. Structure of each block

The above Fig. 1 shows the basic structure of each block which includes index, previousHash, timestamp, data, hash, difficulty, nonce and merkleRoot.

**2.1.1 Index**

Index of each block represents the height of the current block which starts from index 0 from the genesis block. Through the mining process, a new block will be generated, and the index number will be the last block in the chain +1.

**2.1.2 PreviousHash**

PreviousHash stores the hash of the last block which is the block with index -1 of the current block.

**2.1.3 Timestamp**

Timestamp, which is recorded in the format of the number of milliseconds since 1970 00:00:00 to now divided by 1000, stores the date and time when each block is successfully mined [2].

**2.1.4 Data**

The data section of each block stores the transaction input, transaction output and the transaction ID information.

For the transaction input, the information of the previous transaction ID, the block index of the transaction output, together with the sender’s signature generated by the sender's private key are recorded, indicating that the previously obtained coins on the sender side are unlocked to be used [1].

For the transaction output, the address of the receiver, amount of transaction and the transaction output ID are recorded.

The transaction ID is recorded by calculating the hash of all the content within the transaction excluding the sender’s signature [1].

**2.1.5 Hash**

This field is calculated by hashing the whole content of the block like all transaction data, difficulty, timestamp and previous hash, which modifying the content of the block will result in a different hash value. This is significant to allow the blocks to verify if the previous block has been modified [1].

**2.1.6 Difficulty**

The difficulty field stores the value of difficulty requirement when generating that specific block, which is a part of proof-of-work. The details of proof-of-work is mentioned in 2.2.1 of this report.

**2.1.7 Nonce**

The nonce field stores the number of nonces tried to successfully generate that specific block which, similar to the difficulty field, is a part of proof-of-work.

**2.1.8 Merkle Root**

Merkle tree is a hash-based data structure. Each leaf node is a hash of data, and each non-leaf node is a hash of its children. Merkle root, the root hash of the merkle tree, can help to verify the transaction and that block cannot be modified.

A stream `merkle-tree-stream '' is added in our Naivecoin project from an open source project [3]. It generates a merkle tree according to the incoming data.

First, the base merkle tree instance is defined when the program is initiated. The first root hash is generated in the genesis block. The genesis block also stores the base merkle tree instance so other new blocks can reference the base merkle tree instance to generate a new hash base on the merkle tree root. Each new block will store the latest block of the merkle tree.

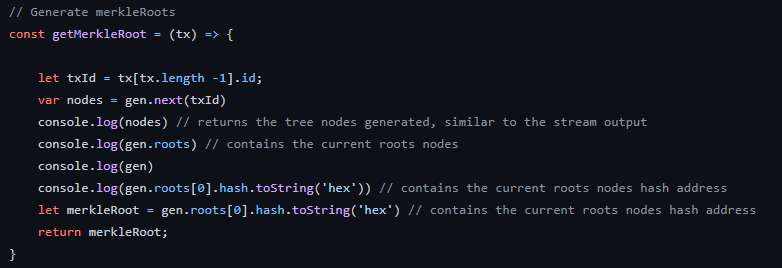


Fig. 2. Generate merkle tree hash.

**2.2 Mining**

**2.2.1 Proof-of-work**

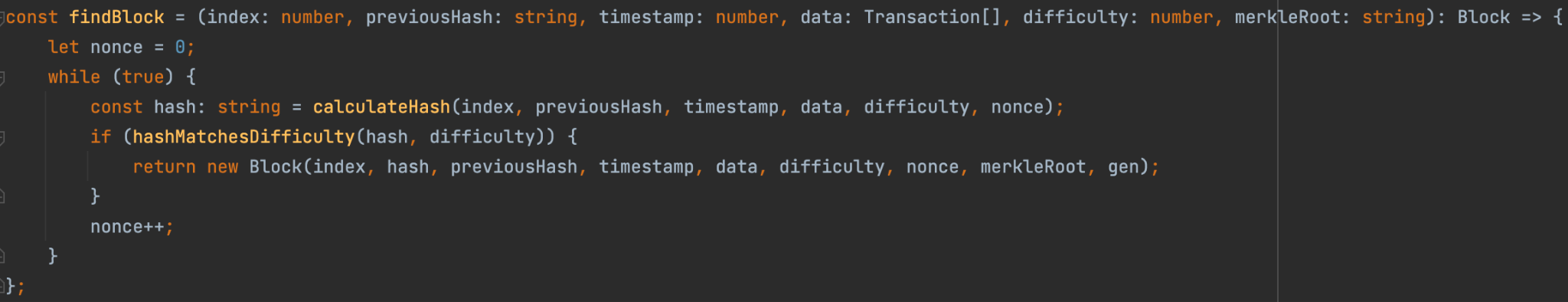
****

Fig. 3. FindBlock function

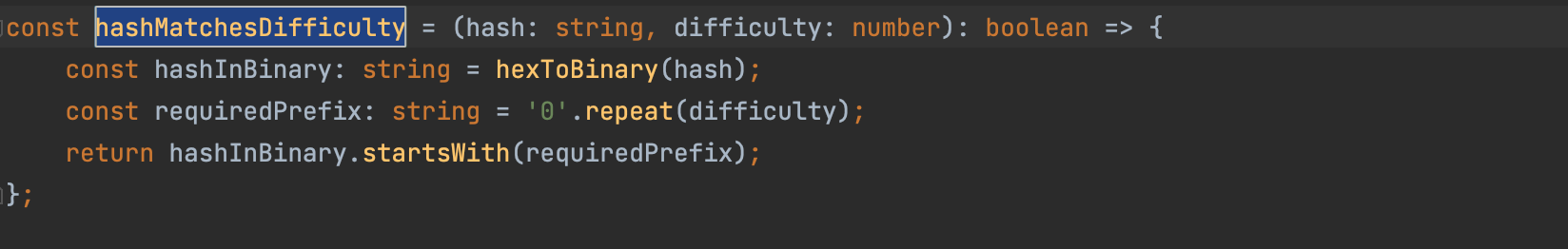


Fig. 4. HashMatchesDifficulity function

Proof-of-work is a function that ensures the blocks generated are valid. Only the valid block will be added to the blockchain, which is identified by the number of prefixes of the hash that matches the difficulty. For example, if the number of difficulty is 4, the first four digits of the binary hash should be zero [1].

The above Fig. 3 and 4 shows how the feature was implemented. If the block generated does not meet the required prefix of the difficulty, the value of nonce which starts from 0 initially will be increased by 1. As the hash of a block is calculated by all the content of the block including nonce, the hash generated each time will be different by increasing the value of nonce. The process will end if the generated hash matches the difficulty.

**2.2.2 Dynamic Difficulty**

****

Fig. 5. GetAdjustDifficulity function

To ensure the block time interval of generating each block is controllable, which prevents the blockchain from having too many or too few blocks compared to the prediction initially, the adjust difficulty function had been introduced. As mentioned in the last section, difficulty controls the number of prefixes needed to be 0 in the binary hash of each block. When the difficulty increases, the average attempts of nonce to fulfill the difficulty also increases, resulting in an averagely longer mining time for each block. Therefore, by adjusting the difficulty dynamically, an averagely even time interval of generating each block can be obtained.

Currently, the difficulty is evaluated every 10 blocks (DIFFICULITY\_ADJUST\_INTERVAL =10) and the target time interval of generating each block is 10 seconds (BLOCK\_GENERTION\_INTERVAL =10). If the time taken is larger than twice of BLOCK\_GENERTION\_INTERVAL \* DIFFICULITY\_ADJUST\_INTERVAL, the difficulty will be decreased by 1 [1].

Otherwise, if the time taken is half the twice of BLOCK\_GENERTION\_INTERVAL \* DIFFICULITY\_ADJUST\_INTERVAL, the difficulty will be increased by 1 [1].

**2.3 Transaction**

**2.3.1 Pay-to-Public-Key-Hash (P2PKH) (Try to implement but unsuccessful)**

In the original code, the transaction between two accounts is based on sending the public key as an address. This method does not satisfy the definition of P2PKH.

There are many kinds of attempts to change the transaction method. The basic idea in P2PKH is sending the script which includes the hash of the public key [4]. For the hash function, to create the hash of the public key, using SHA256 and RIPEMD160 at the same time to ensure the security of the transaction.

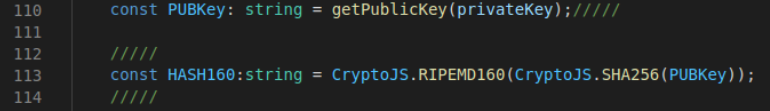


Fig. 6. Transform public key to hash

The second step is the form of the script. the code from [5] was found and used the bitcoin library.

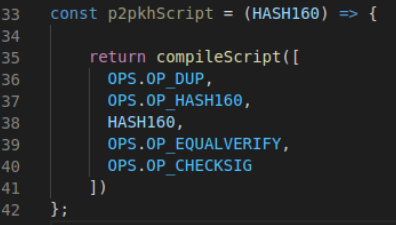


Fig. 7. The simple script including the hash of the public key

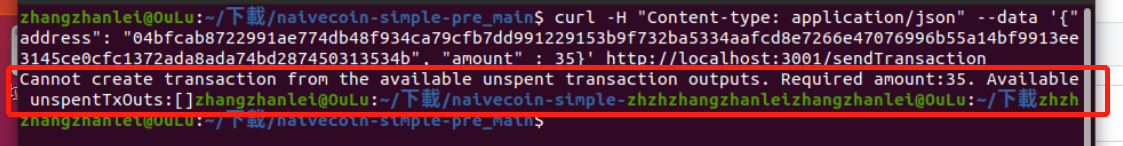
The third step is verification. But in this step, the transaction is unsuccessful. The transaction cannot be created by sending a script.

Fig. 8. The error message about the unsuccessful transaction

After the script method is unsuccessful, it is believed that the transaction can just send the hash of the public key to replace the public key as the address. It is the same as the script method, there are many challenges in the verification step. To avoid some verification, some verifying functions needed to be removed using the public key as address.

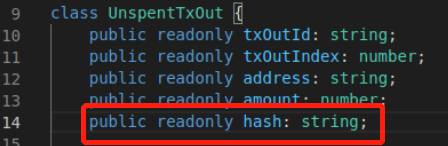


Fig. 9. Add the hash value in the USTO

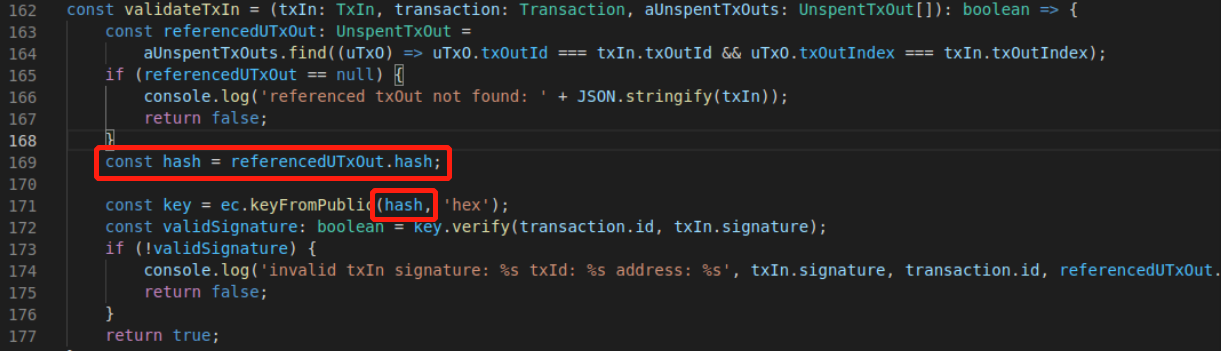
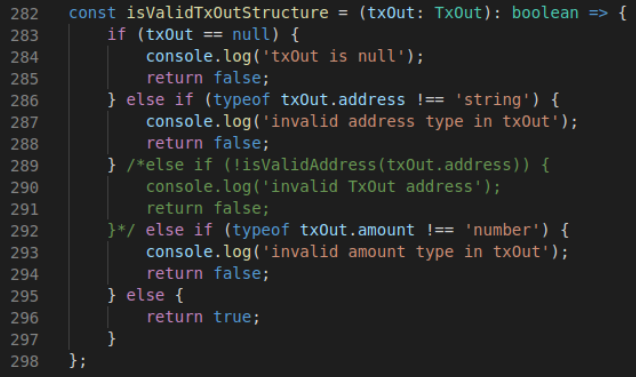
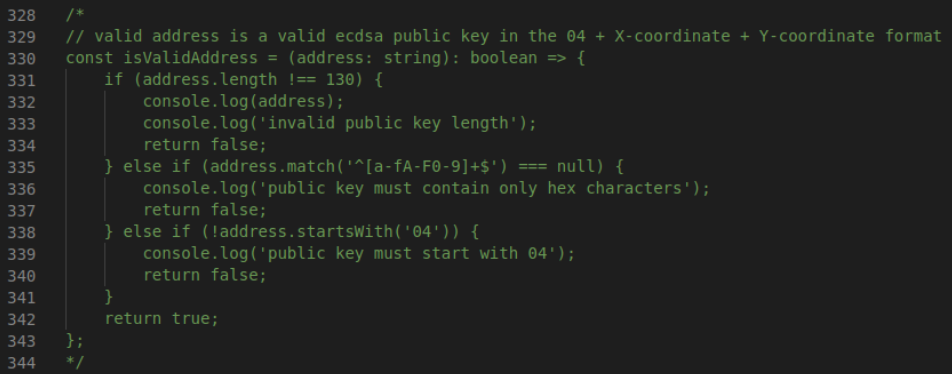


Fig. 10. In verification process, comparing the hash value with the reference’ hash

Unfortunately, the modifying process is unsuccessful until the whole system cannot run. Our team gave up on changing the transaction method to P2PKH.

****

(a) (b)

Fig. 11. The verification process in original code basing on comparing the public key as address

**2.3.2 Digital Signature and Signature Verification**



Fig. 12. Signature included in the block

In our Naivecoin system, each user owns a set of unique private and public keys.

The signature in Naivecoin is used for signing the transaction input with the sender's private key in ECDSA. The credibility of the coins in the transaction input can be verified with the public key of the sender.

Meanwhile, the public key is used as the address of the user during the transaction.

**2.3.3 Coinbase Transaction**

****

Fig. 13. Constant COINBASE\_AMOUT sets to 50

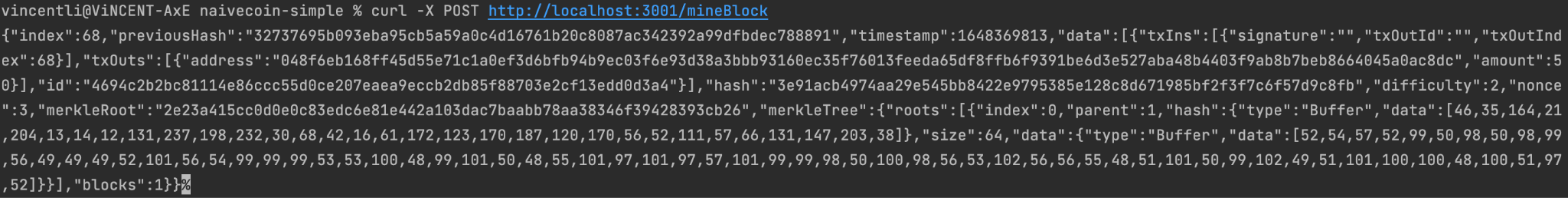


Fig. 14. Screenshot of obtaining coinbase transaction

In order to encourage miners mining the unconfirmed transaction in the transaction pool to activate the transaction, a coinbase transaction which only includes output is introduced to reward miners 50 coins per each block mined. As miners usually obtain multiple coinbase transactions, to prevent the transaction ID being always the same due to the same amount of the coins and the same output address, block height (txOutIndex) is introduced to the input of coinbase transactions as shown in Fig. 14. [1].

**2.4 Network**

**2.4.1 Peer to Peer (P2P) Connection**

****

Fig. 15. Screenshot of connecting two Naivecoin programs by using websockets

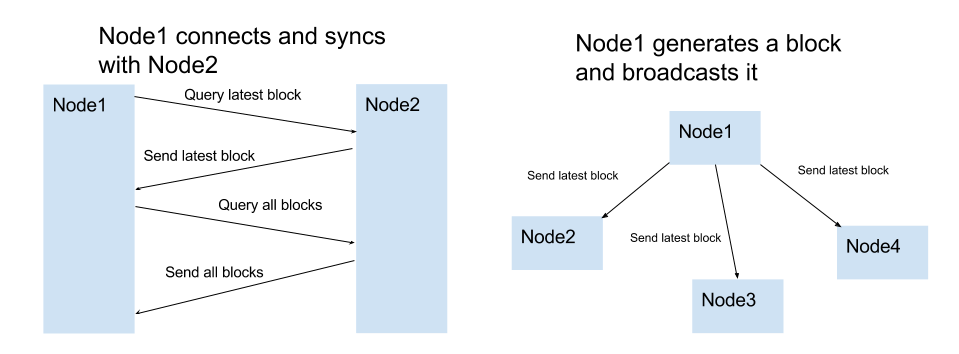
****

Fig. 16. Structure of P2P in Naivecoin [1]

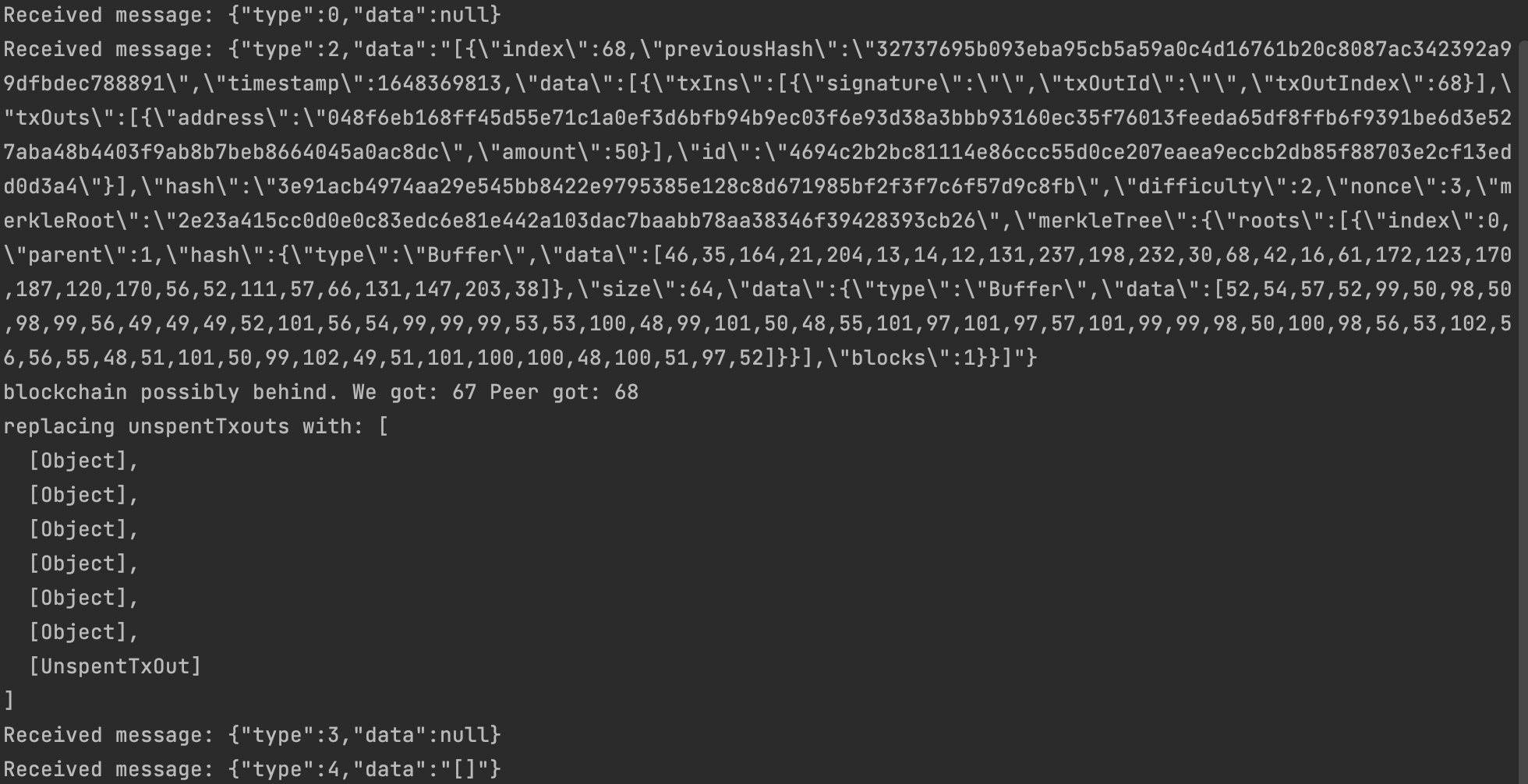


Fig. 17. Node A received P2P message from Node B

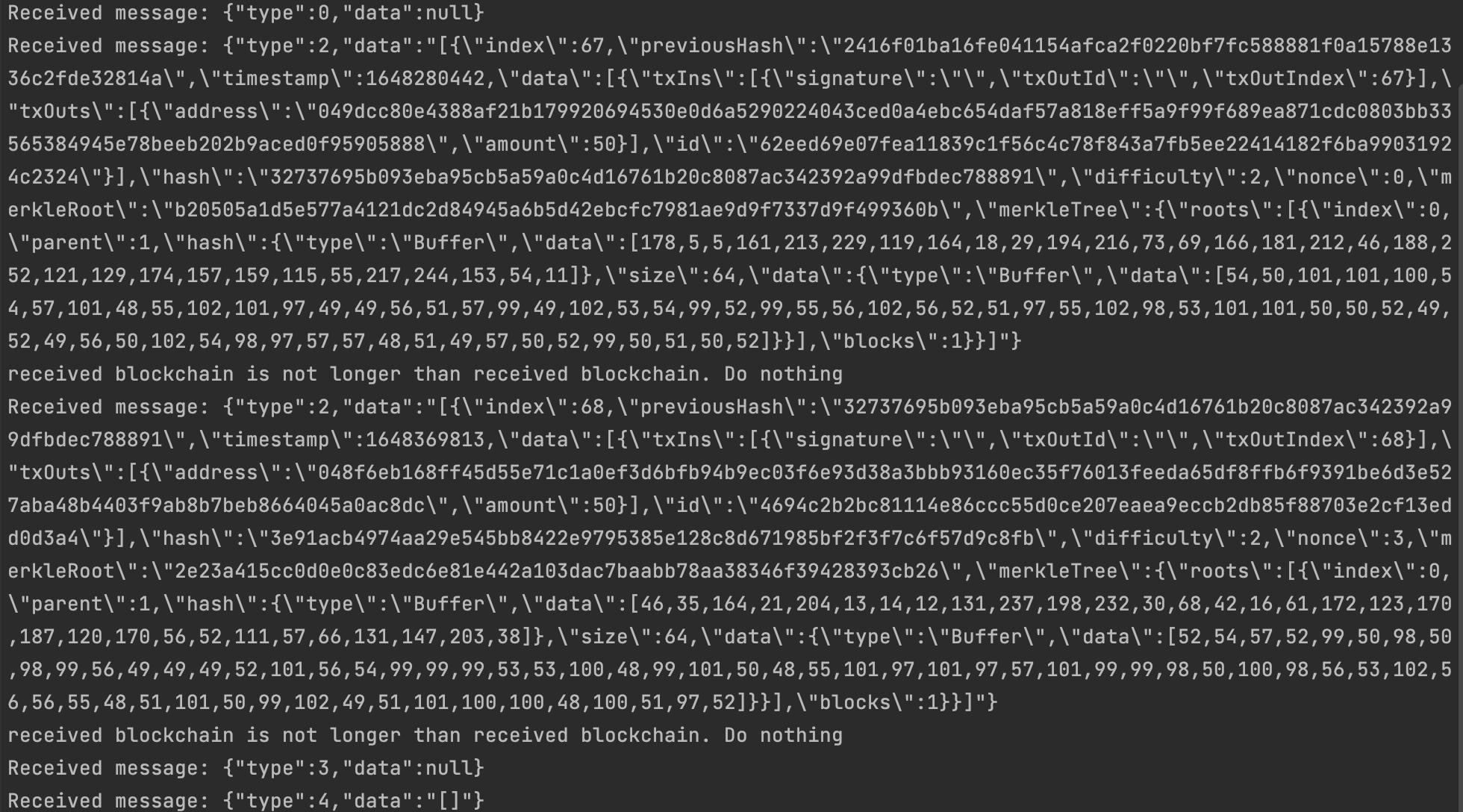


Fig. 18. Node B received P2P message from Node A

By forming a Naivecoin network using websockets, users are able to exchange coins worldwide through connecting the network. Localhost environments with two Naivecoin programs were installed on the same computer to simulate the connection. When users call the add peer API similar to Fig. 15, the P2P connection will be established, and the latest blocks will be synchronized. Three rules are established to ensure P2P connection works [1]:

* Once a new block is mined, the node will be broadcasted to all peers connected to the network
* Once a new peer is being added to the network, the nodes will communicate to the new node and query the latest block similar to Fig. 16
* When nodes receive blocks with higher index number than its current block list, it will append the block to the list or query the whole blockchain from the network similar to Fig. 17

The details of validation during synchronization will be discussed in part 2.4.2 and 2.4.3 of this paper.

**2.4.2 Validating the received Blocks**

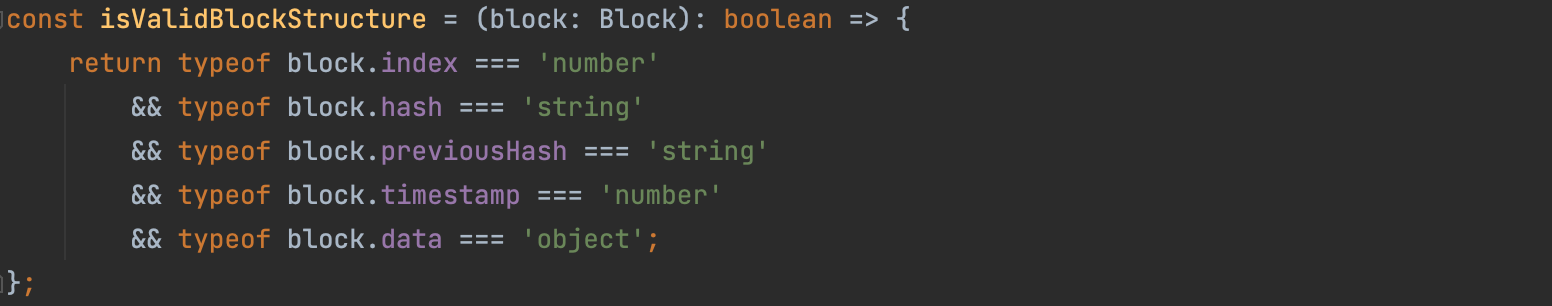
****

Fig. 19. Function validating the block structure

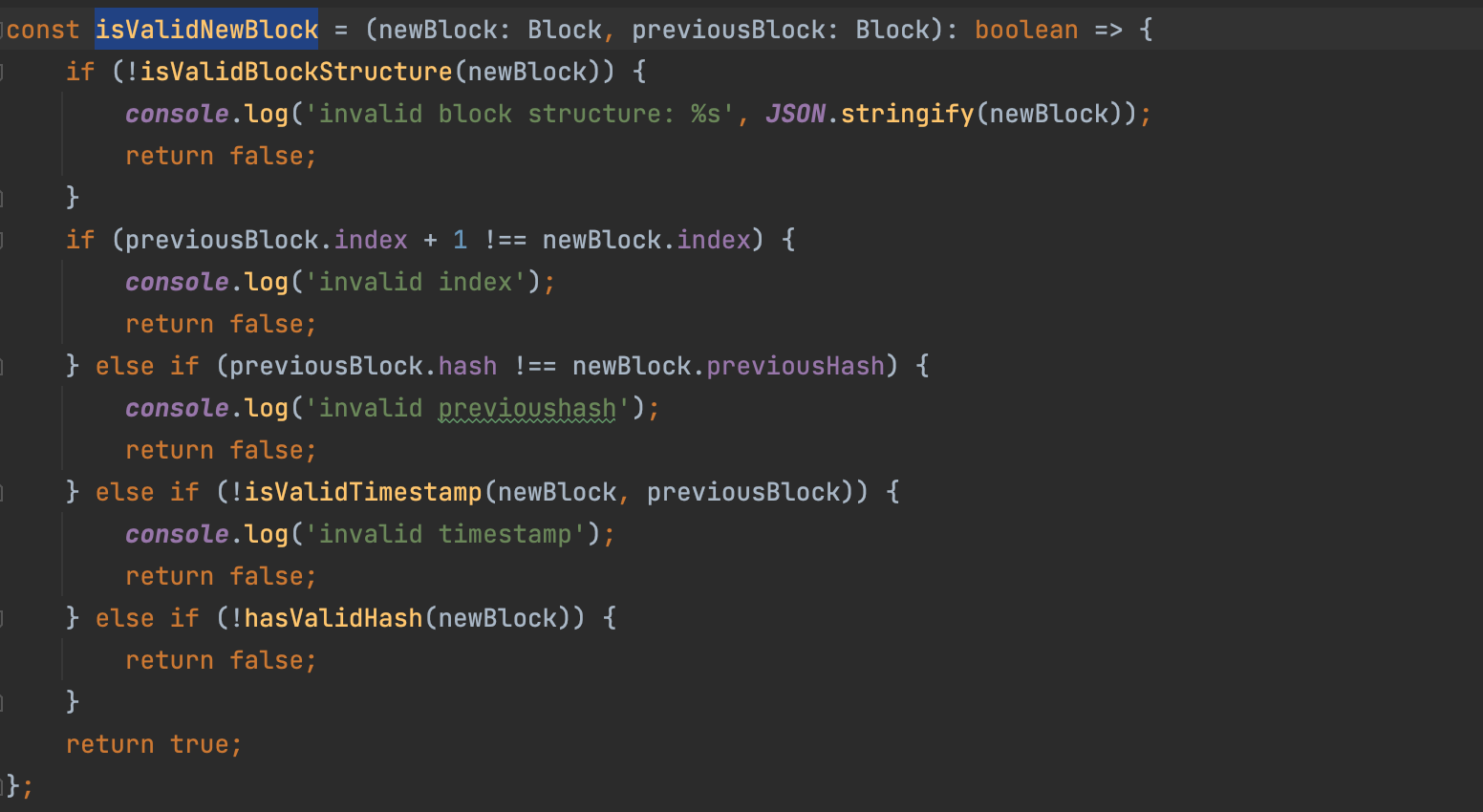
****

Fig. 20. Function validating the new block

Before putting a new block into the chain, validation is required. It is checked that the new block’s index should be one more than the previous block index. Also, the previous block hash must be the previous hash in the new block. At last, it checks the format of timestamp and hash data.

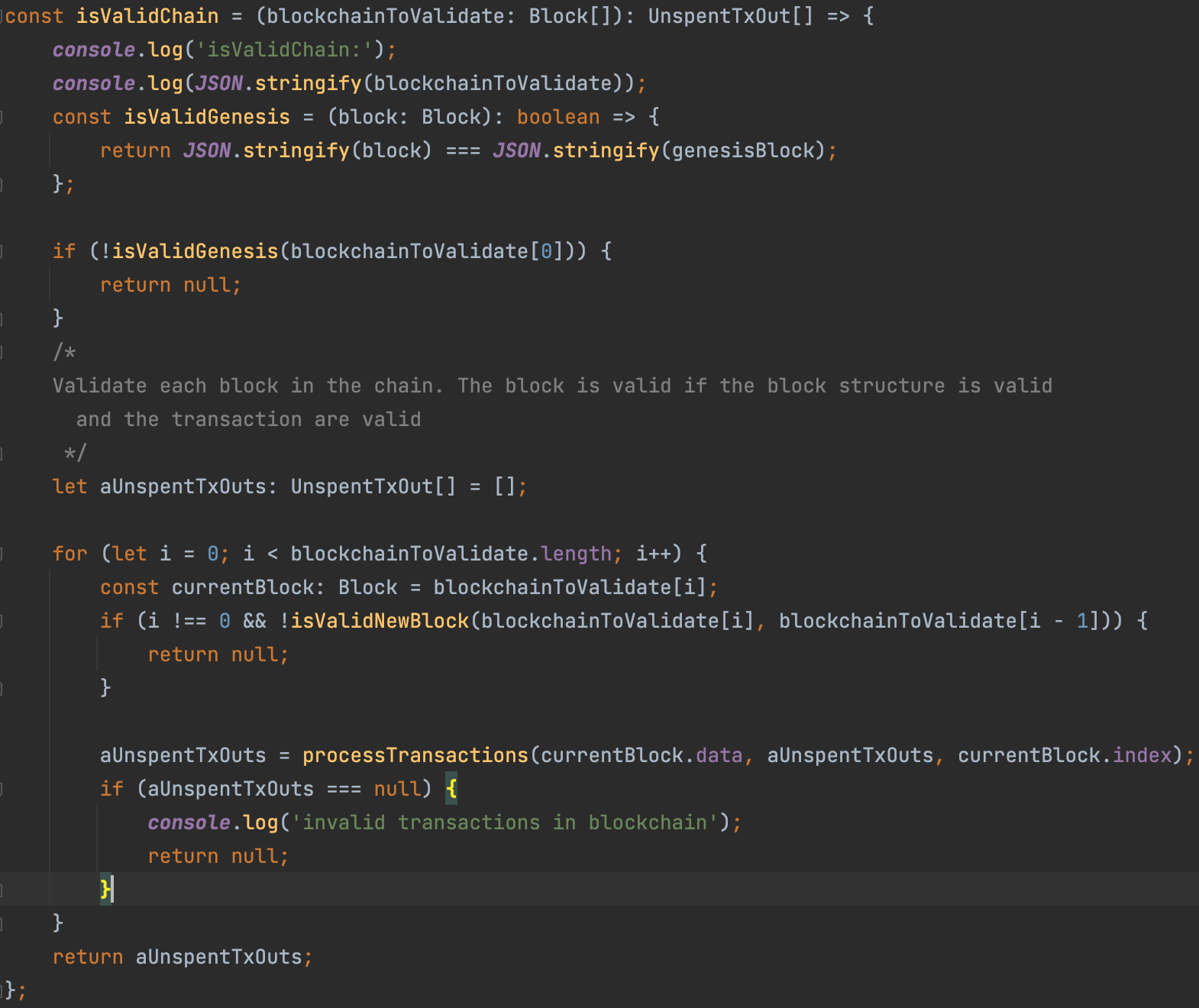
****

Fig. 21. Function validating the blockchain

To verify the blocks in the chain, it needs to validate whether the block structure and transactions are valid. It checks if it starts from the genesis block. Also, it checks if the unspent transaction output is empty.

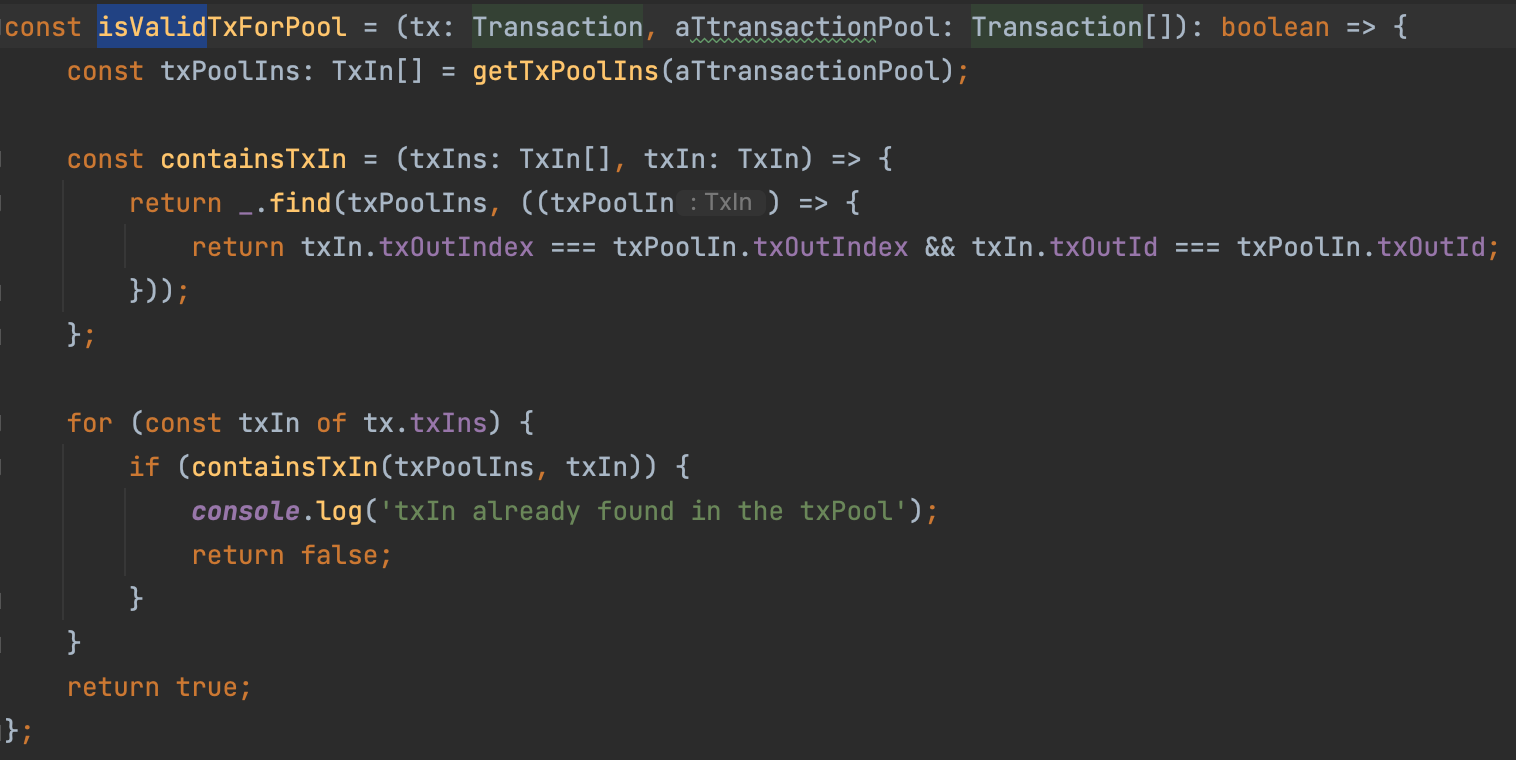


Fig. 22. Function validating the transaction pool

It checks whether the transaction already existed in the pool. It uses index and transaction ID to search.

**2.4.3 Longest Chain Rule**

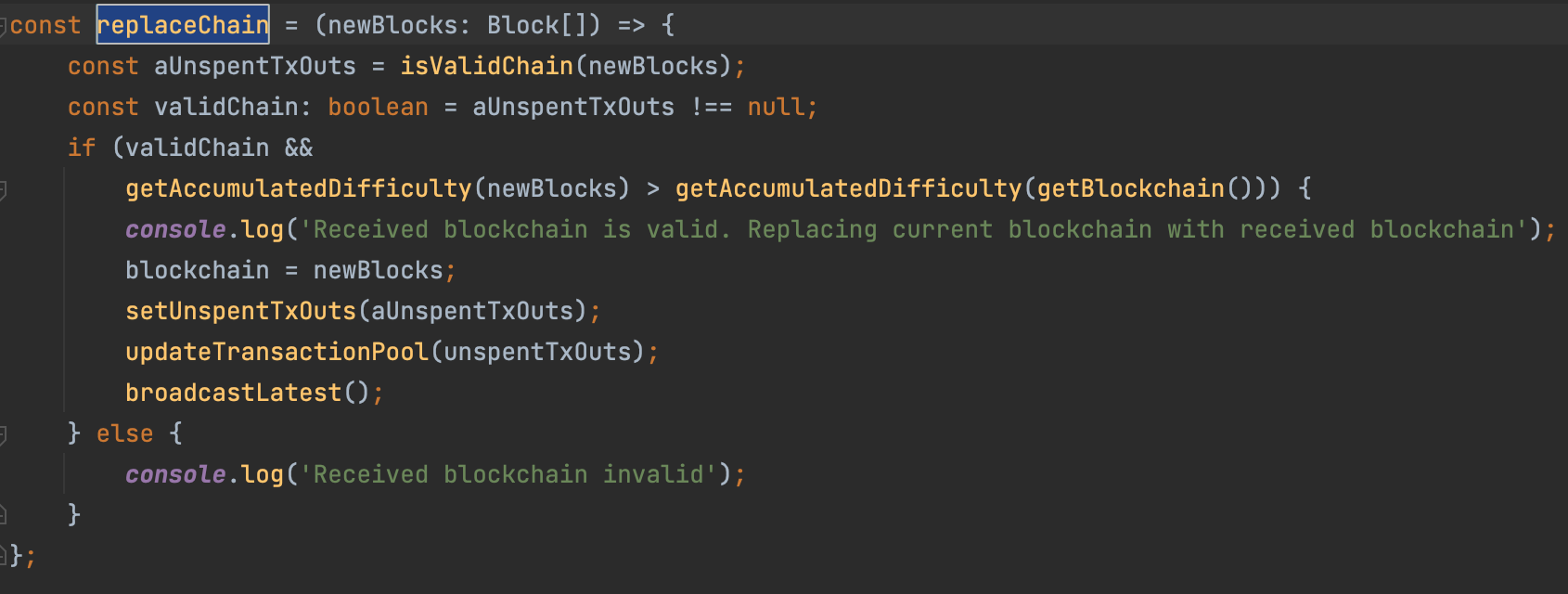
****

Fig. 23. Function replacing the shorter blockchain with longer blockchain

In the blockchain system, it is normal to generate a fork when the transaction is still in unspent transaction output (UTXO) and not yet confirmed. Hence, the longest chain rule is applied to solve the fork problem and help miners to pick the longest one to work on it. In Naivecoin, the existence of unspent transaction output is used to verify the validity of the longest chain, as shown above Fig. 23.

**2.5 Storage**

**2.5.1 Storage raw Blockchain data on Disk**

****

Fig. 24. JSON file stores all the blocks

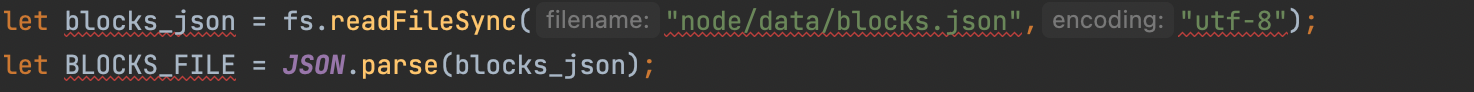


Fig. 25. Blocks in JSON files load to the list in the program file when booted (part 1)

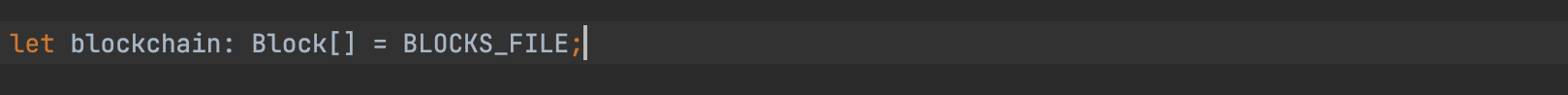


Fig. 26. Blocks in JSON files load to the list in the program file when booted (part 2)

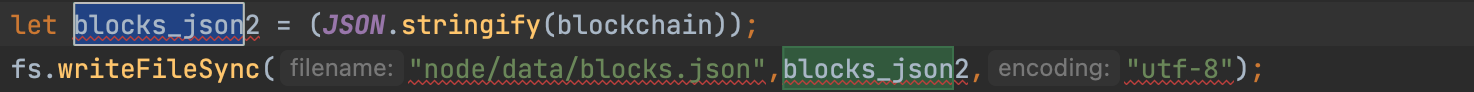


Fig. 27. store the new generated blocks to JSON file

All the raw data of the whole blockchain are stored in a JSON File locally as shown in Fig. 24. The block history will be read from the file to a list when the Naivecoin program runs and each newly mined block will be added to the JSON File, ensuring the data in the JSON File is up-to-date.

**2.5.2 Storing unspent transaction (UTXO) on disk**

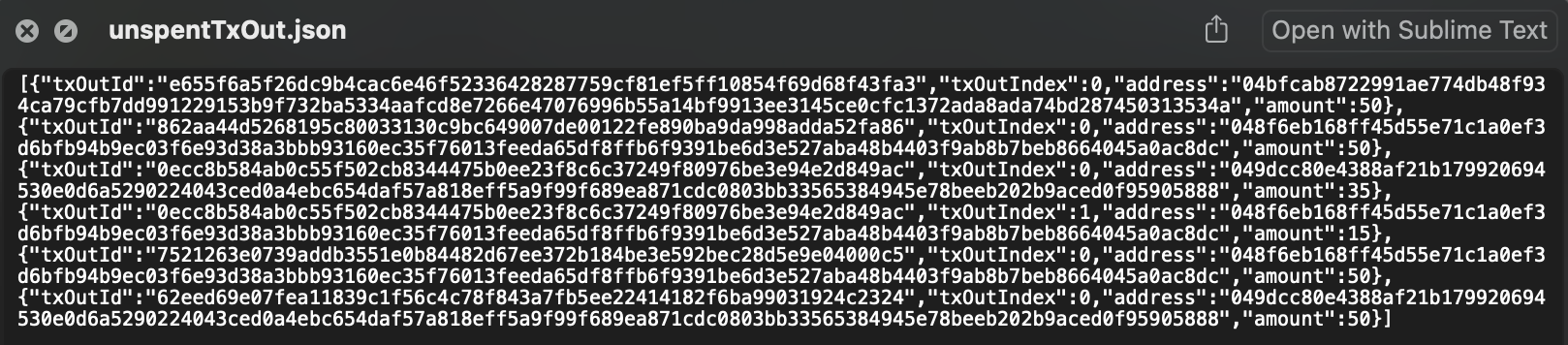
****

Fig. 28. JSON file stores all unspent transaction (UTXO)

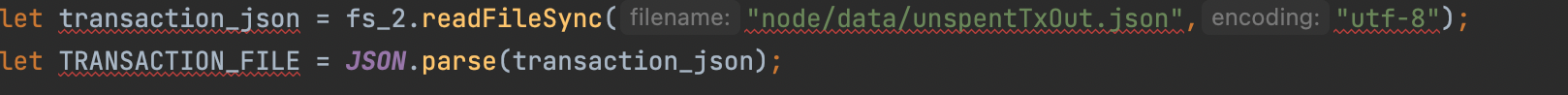


Fig. 29. UTXO in JSON files load to the list in the program file when booted (part 1)

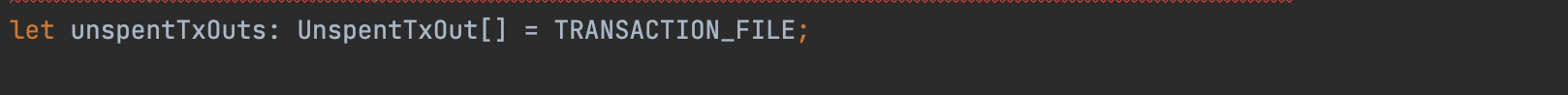


Fig. 30. UTXO in JSON files load to the list in the program file when booted (part 2)

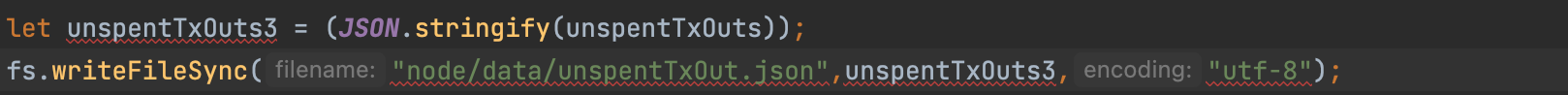


Fig. 31. Latest UTXO updated to JSON file

All the unspent transactions are stored in a JSON File locally as shown in Fig. 28. Similar to storing raw Blockchain data, the unspent transactions will be read from the file to a list when the Naivecoin program runs and the latest unspent transactions list will be updated to the JSON file once the unspentTxOuts variable is updated (new blocks are mined), ensuring the record is up-to-date.

**2.5.3 Transaction Pool**

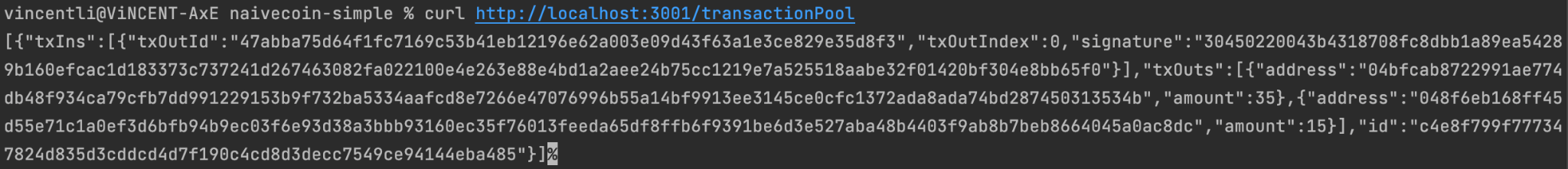
****

Fig. 32. Unconfirmed transaction stored in transaction pool

A transactionPool.ts file was created to store all new transactions. All newly generated transactions are classified as unconfirmed transactions, which are waiting for others to be mined to confirm the transaction is valid. Therefore, only transaction input and output data are stored in the transaction pool.

1. **User Manual**

****

Fig. 33. Screenshot of inputting ‘npm start’ command

Before mining any blocks or performing transactions, command ‘npm start’ needed to be typed on the terminal to activate the Naivecoin program. Then, both http port number and p2p port number will be displayed like Fig. 33, which are compulsory in the following sections.

**3.1 Connect Nodes (P2P connection)**

****

Fig. 34. Screenshot of connecting two nodes and checking which nodes are connected

Currently, peer to be added manually by typing the following http request to the terminal:

curl -H "Content-type:application/json" --data '{"peer" : "WebSocketAddress"}' http://localhost:HttpPortNumber/addPeer, where target node’s web socket address and the http port number shown in the last section should be included. If the command is valid, the nodes will be connected automatically, and the user of the target node is not required to type any command to accept the connection.

To check which nodes are currently connected, the user can type the following command:

curl http://localhost:HttpPortNumber/peers.

**3.2 Mine Block**

Blocks in our Naivecoin need to be mined manually by inputting the following command to the terminal: curl -X POST http://localhost:HttpPortNumber/mineBlock. By typing the command each time the program will go through the proof-of-work process automatically until the obtaining a hash of the block matches the difficulty.

**3.3 Transaction**

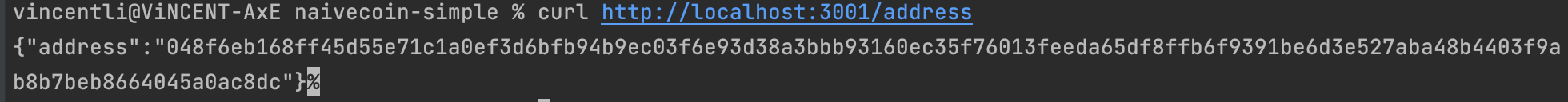
****

Fig. 35. Screenshot listing the address of the user

To transfer your coins to others, the user needs to know the receiver’s address which is the same as the receiver’s public key. The user’s own address can be enquired by: curl http://localhost:HttpPortNumber/address.

After knowing the address of the receiver and deciding the amount needed to be transfered, the user can type the following command to submit the transaction: curl -H "Content-type: application/json" --data '{"address": "ReceiverAddress", "amount" : NumberOfCoinsToBeTransfered}' http://localhost:HttpPortNumber/sendTransaction.

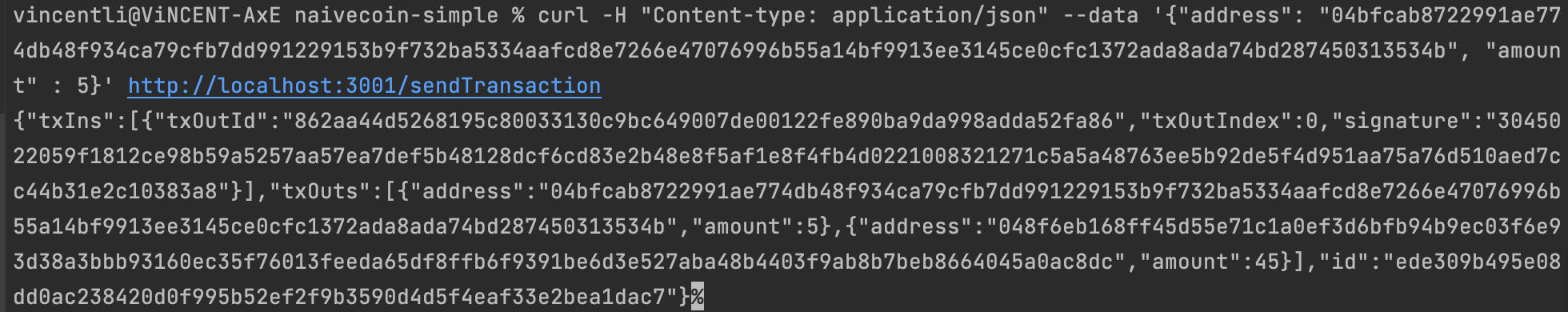


Fig. 36. Screenshot of valid transaction



Fig. 37. Error message when the available of unspent transaction is not enough

If the transaction is valid, the details of the unconfirmed transaction will be displayed like Fig.36. The transaction will be executed when someone mined the unconfirmed transaction. However, if the transaction amount is larger than the balance of the user, an error message will be displayed like Fig. 37, which lists all the unspent transactions of the user.

**3.4 Wallet Balance**

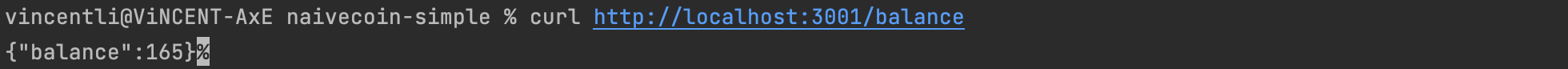
****

Fig. 38. Screenshot listing the balance of the user

To check the balance of the account, the user can type the following command: curl http://localhost:HttpPortNumber/balance, which sums all the unspent transactions in the unspent transaction list where the address equals the user’s public key.

**3.5 Other Functions**

|  |  |
| --- | --- |
| **Function** | **Command** |
| List all blocks in blockchain | curl http://localhost:HttpPortNumber/blocks |
| List all unconfirmed transaction | curl http://localhost:HttpPortNumber/transactionPool |
| List all unspent transaction | curl http://localhost:HttpPortNumber/unspentTransactionOutputs |
| List all unspent transaction which the address equals the public key of user | curl http://localhost:HttpPortNumber/myUnspentTransactionOutputs |

1. **Code**

<https://github.com/AbitraryYu/naivecoin-simple.git>

**\*IMPORTANT\***

There are several branches in GitHub, download and run our project in the **production** branch in GitHub. The P2PKH branch is based on the production branch but it also includes the attempted implementation of P2PKH.

Our Naivecoin project is divided into several sub-programs.

In /src/,

* main
* blockchain
* p2p
* transaction
* transactionPool
* util
* wallet

The JSON files in /node/data are the log files for storing the transactions in the entire Naivecoin blocks and UTXO.

**References**

|  |  |
| --- | --- |
| [1] | L. Hartikka, “Naivecoin: a tutorial for building a cryptocurrency,” n.d. [Online]. Available: <https://lhartikk.github.io/> [Accessed Feb 10, 2022]. |
| [2] | W3schools, “JavaScript Date getTime(),” 2022. [Online]. Available:  https://www.w3schools.com/jsref/jsref\_gettime.asp [Accessed Mar 19, 2022]. |
| [3] | Mafintosh, “Mafintosh/Merkle-tree-stream: A stream that generates a Merkle tree based on the incoming data,” 2021. [Online]. Available: https://github.com/mafintosh/merkle-tree-stream [Accessed Feb 22, 2022]. |
| [4] | G. Walker, “P2PKH,” 2020. [Online]. Available: <https://learnmeabitcoin.com/technical/p2pkh> [Accessed Mar 27, 2022]. |
| [5] | B. Mancini, “Bitcoin P2PKH Transaction Building with Node.js,” 2019. [Online]. Available: <https://www.derpturkey.com/bitcoin-p2pkh-exploration/> [Accessed Mar 27, 2022]. |

**Appendix – Contribution Table**

|  |  |
| --- | --- |
| **Groupmate Full Name** | **Contribution** |
| 20001138G Cheuk Man Ting | Program testing, Documentation (report section 1, 2.4.3, 4), presentation slides |
| 21000018G Li Yat Long | JSON file storage implementation (raw block, UTXO), Program testing, Documentation (report section 2, 3), presentation slides, Video recording for presentation |
| 21006322G To Ching Wai | Program testing, Documentation,  presentation slides |
| 21014947G Yu Hin Chung Nikko | Merkle Tree implementation, GitHub creation, report (report section 2.1.8, 4), presentation(storage) |
| 21045374G Zhang ZhanLei | P2PKH research, testing & implementation, Documentation (report section 2.3.1) |