

In this project, we implemented a block chain, realizing its functionalities and simulating its working process using our self-defined API. We mainly focus on Blockchain mining and reaching consensus. Things related to P2P transactions are not implemented in our code.

We use Python as our development language. Using the “*Flask*” package to develop the front end to interact with users: each node has its own ip address and port. We also define specific urls to call corresponding functions. For example, if we have a node running at 15.232.64.13: 9871. We can call url: 15.232.64.13: 9871/mine. Then the node will do mining until it finds a qualified block or be broadcasted a block mined by neighbors. For hashing, we use python standard library: hashlib.sha256 to make a hash. In order to enhance users’ experience, we write some html decorated with css in those interfaces.

In the real blockchain systems, there are some routers running so each node can learn about its peers through these routers. We do not make a routing system since there are usually no more than 10 nodes during our development and usage. Instead we use a registration mechanism to let a node learn about its neighbors. Details are described in the “Interaction API” section. Using our provided urls, users can mine a new block, see the current locally stored copy of the blockchain in each node, make a new transaction with provided sender, recipient and amount, resolve conflicts with neighbors, check if a transaction has been confirmed.

Code Usage Instructions:

All needed code is in `block_chain.py`. To run a node, use “*python block_chain.py -port your_port*”. It will launch a node. To register another node, call url: node’s ip: node’s port number/register, e.g. 15.123.41.11:9871/register. Enter another node’s url in the window and submit.

To set up the environment, please install a python 3.6 version. Then “*pip install -r requirements.txt*” to install the required packages. Then it is ready to run. We recommend using conda to create a 3.6 version python and install packages.

Part 1: Data Structures

1. **Transaction**: We use a simple dictionary to represent a transaction. Since we did not focus on P2P transactions, in our code we use clear text instead of encrypted transactions. Each transaction has a sender field, which is the address of the sender, a recipient field which is the address of the receiver and the amount of bitcoin in this transaction, which is a float variable. Each node has an unique address. Transaction also has a timestamp and a hash, which acts as its id. We rely on this id to find and confirm a transaction.
2. **Block**
Each block has the following members:
 - a. **Index**: Each block has its own index, also acting as its height. The index indicates its location in the blockchain. Besides, the height information helps us decide if a transaction is confirmed: We can get the index of a block containing a specific transaction and then acquire the current height of the blockchain. We can easily find out if this transaction has been followed by more than 5 blocks.

- b. **Transactions:** a list of transactions. The transactions in the list are the transactions that have been added to the block. Once the node decides to mine a new block, it will collect some transactions from its mem pool. It is noteworthy that we have to decide which transactions will be added to the block before we do proof of work since the hash of a block will change if we change the transactions in it.
- c. **Timestamp:** A timestamp indicating the exact time when the block is mined.
- d. **Previous hash:** This is the hash of the last block in the current blockchain. It ensures one block in the chain can not be easily replaced since it will impact the next block's previous hash and all the blocks followed.
- e. **Difficulty:** This is the level of difficulty of the block. We store it in the blockchain because the difficulty of a blockchain will change in the future. For the already created block, we store their difficulty to help us validate the block.
- f. **nonce :** This is an integer we adjust to create a hash has specified leading zeros (same as the block's difficulty).
- g. **Merkle hash:** This is a hash generated through merkle tree. It is generated using all the transactions in the block. Hence if someone change a transaction in the block, the new merkle hash can not match the existing merkle hash. Details are explained in the Interaction *API* section.
- h. **Block hash:** A block's hash is computed using SHA256 using its index, timestamp, merkle hash, previous hash and nonce.

3. Blockchain

Each blockchain has the following members

- a. **Current_transactions.** This is a list of transactions that has not been added to a block. It acts like a mem pool. When mining a new block, the block will retrieve some transactions from this pool. The block will prioritize those transactions with a higher amount. (This is not a standard design for all blockchain implementations). If a block is mined and not yet added to the chain, and at the same time a block mined from other nodes is added to the chain, the node will give up this node, releasing all the transactions in the node back to the mem pool (except those confirmed transactions)
- b. **Chain.** This is a list containing all the added block.
- c. **Nodes:** A python set object containing all other nodes' ip and port number. As mentioned before, we use registration instead of routing to learn about a node's peers.
- d. **Genesis_block.** This is the first block. Its index is 1 and its previous hash is 100.
- e. **Level of difficulty:** Current difficulty of blockchain. New block's hash must have this amount of leading zeros in order to be added to the blockchain.

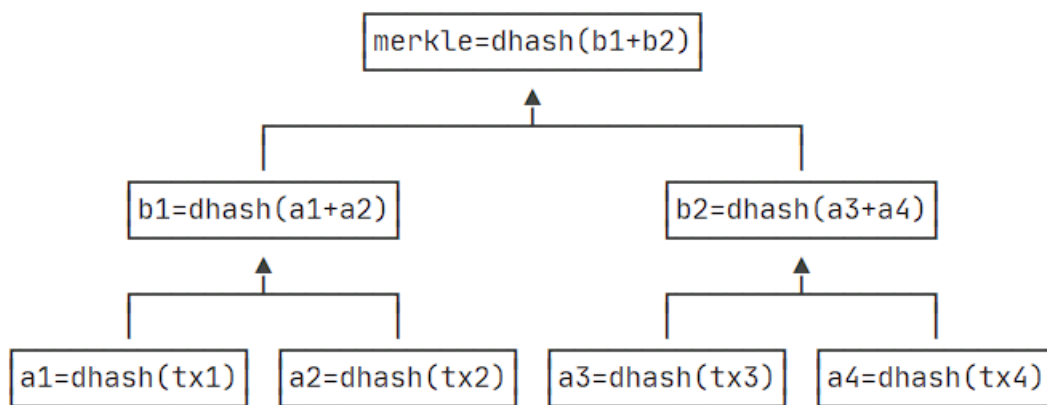
Part 2: Interaction APIs

1. Mine

Mining function mine() is the major part of block chain implementation where a new block is forged by the user and added to the chain if checked to be valid. This function is

triggered when a 'GET' request is made to the /mine endpoint. The implementation can be found in Fig. 2.

Specifically, first, the new block will gather several transactions from its node's mem pool (current_nodes list). These transactions and another transaction sending from "0" to the current node, acting as a mine reward, will be stored in the new block. Then the miner will calculate the merkle hash of the new block, Merkle hash is a way to calculate hash. All data is organized into a tree structure and all non-leaf node represents its child nodes' hash. This method ensures a small change in the leaf data will result in significantly different root hash value. Hence keeping transactions information safe. The following graph shows its basic mechanism.



Then we will do proof of work. The miner will constantly increase the nonce from zero by one until the SHA256 hash of the block has leading 0s equals to the level of difficulty. The block's hash is related to all its members: index, transactions, timestamp, previous_hash, nonce and merkle hash.

Once a new block is mined, it will broadcast to other nodes. Another node will check its validity. If it is valid, it will add to their local copy of blockchain. If they are mining a block and not finish yet, they will give up and start mining from the beginning.

It is noteworthy that for demonstration, we separate mining, broadcasting to other nodes, other nodes reaching consensus so you can see the result at each step clearly. However, in working scenarios, these three steps will be completed in one go.

2. New_transaction

The new_transaction() function is for processing a new transaction for the user. This function is triggered when a 'POST' request is made to the /transaction/new endpoint through the transaction.html form submission (will be introduced later). It checks if all the required fields ('sender', 'recipient', 'amount') are present in the received data. If not, there will be an error message. If all the required data is present, it proceeds to create a new transaction by calling the new_transaction() in the 'blockchain' object. Then, the

function prepares a response message indicating the transaction is added to a specific block (identified by 'index'). Success message will show up in the front end. The 'transaction.html' is designed to display the outcome of the process of creating a new transaction.

3. Full_chain

The full_chain() function is for displaying the entire chain of the blocks. It displays all the blocks in the blockchain in a sequential way. This function is triggered when a 'GET' request is made to the /chain endpoint. It retrieves data by iterating over each block in the blockchain.chain. For each block, it retrieves a list of attributes using the save_list() method of the 'block' object. This method would return key details of a block, such as its index, transactions, timestamp, previous hash, difficulty, nonce and Merkle hash. The retrieved data will be added to the result_chains list. The function then prepares a response dictionary that includes the entire list of blocks (result_chains) and the length of the blockchain. For the displaying hashing part, the function truncates longer hashes (previous hash and Merkle hash) to their first 6 characters, which simplifies the display of hash values. Correspondingly, the "chain.html" visualizes the blockchain data to give a comprehensive view of the blockchain.

4. Register_nodes

We use the flask route decorator, "@app.route('/register', methods=['GET','POST'])", to specify that this function "def register_nodes" should be executed when a request is made to the URL path /register. It accepts both GET and POST requests. In this function, if the user is submitting data through forms. Then we retrieve the value of the 'url' field from the submitted form data defined in our "register.html". Then we check if the 'url' field is empty. If it is, it returns an error message with a 400 status code. Otherwise, we call the blockchain method "register_node" that registers a new node in the blockchain network using the supplied URL. After it's registered, we return a message for confirmation.

As described in the data structure section, we do not use routing to find peers because we have relatively small nodes in our system so we simplify this step by using registration instead. However, it is not how the real blockchain system works.

5. Consensus

The consensus function handles GET requests to the '/nodes/resolve' endpoint, as indicated by the @app.route decorator. It calls the blockchain.resolve_conflicts() method to attempt to resolve conflicts among nodes in the blockchain network.

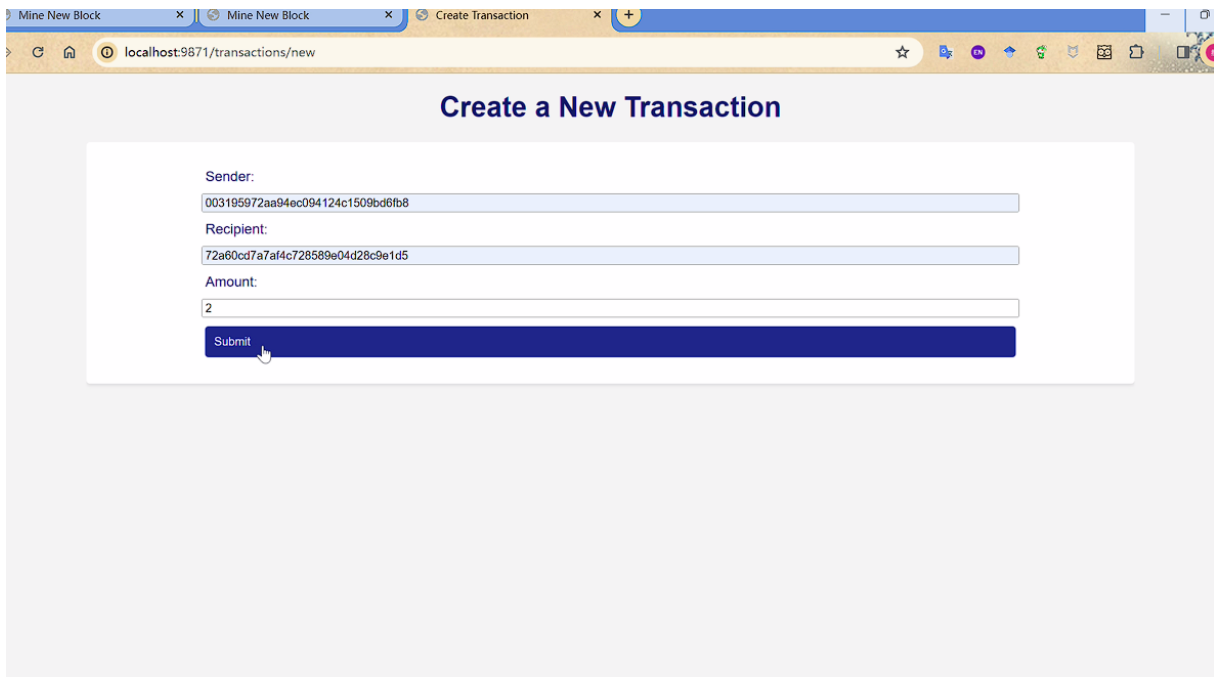
The resolve_conflicts method is a part of a blockchain consensus algorithm that aims to resolve conflicts by replacing the current chain with the longest valid chain found among the neighboring nodes in the network. Specifically, the method iterates through each neighbor node's address, and then sends an HTTP GET request to the neighbor node's /jsonify_chain endpoint to request their blockchain data. If the HTTP response status

code is 200, indicating a successful response, it extracts the received chain from the JSON response. This chain data is a list of blocks. Then the method checks if the received chain's total difficulty is larger than current difficulty and if the received chain is valid by calling the `self.valid_blockchain(chain)` method. This function will verify a block by first checking its previous hash is correct, then check its merkle hash to ensure the transactions in it are correct, then check if its hash has the same leading zeros as its difficulty. If both conditions are met, it regards the block as valid .

After iterating through all neighboring nodes, the method will keep the valid chain with the largest sum of difficulty. It replaces the current chain with the new, longer, and valid chain (`new_chain`), and returns `True` to indicate that the chain was successfully replaced. Otherwise, this method returns `False` to indicate that the current chain remains unchanged.

Finally, it returns an HTML template named 'conflict.html' using `render_template`. This template is used to display information about the consensus resolution process to users, including details about the resolved chains and whether any replacements occurred. Also, it will display the latest block chain to the user.

Appendix



The screenshot shows a web browser window with three tabs: 'Mine New Block', 'Mine New Block', and 'Create Transaction'. The address bar shows 'localhost:9871/transactions/new'. The page title is 'Create a New Transaction'. The form contains the following fields:

- Sender: 003195972aa94ec094124c1509bd6fb8
- Recipient: 72a60cd7a7af4c728589e04d28c9e1d5
- Amount: 2
- Submit button

Figure 1. New Transaction. Input a transaction's sender and receiver with the amount of bitcoin. It will be stored in node's mem pool.

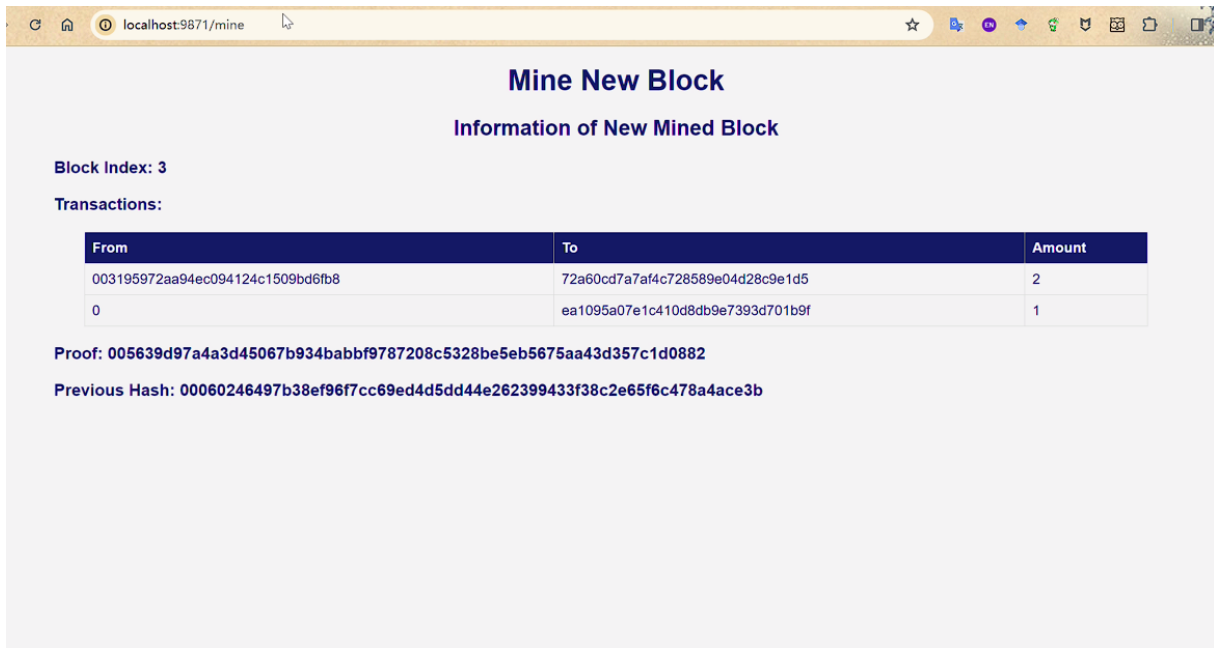


Figure 2 . Mine. The transaction added above will first be stored in a mem pool. Then when there is a new mined block, it will be moved to the block as shown in figure 2.

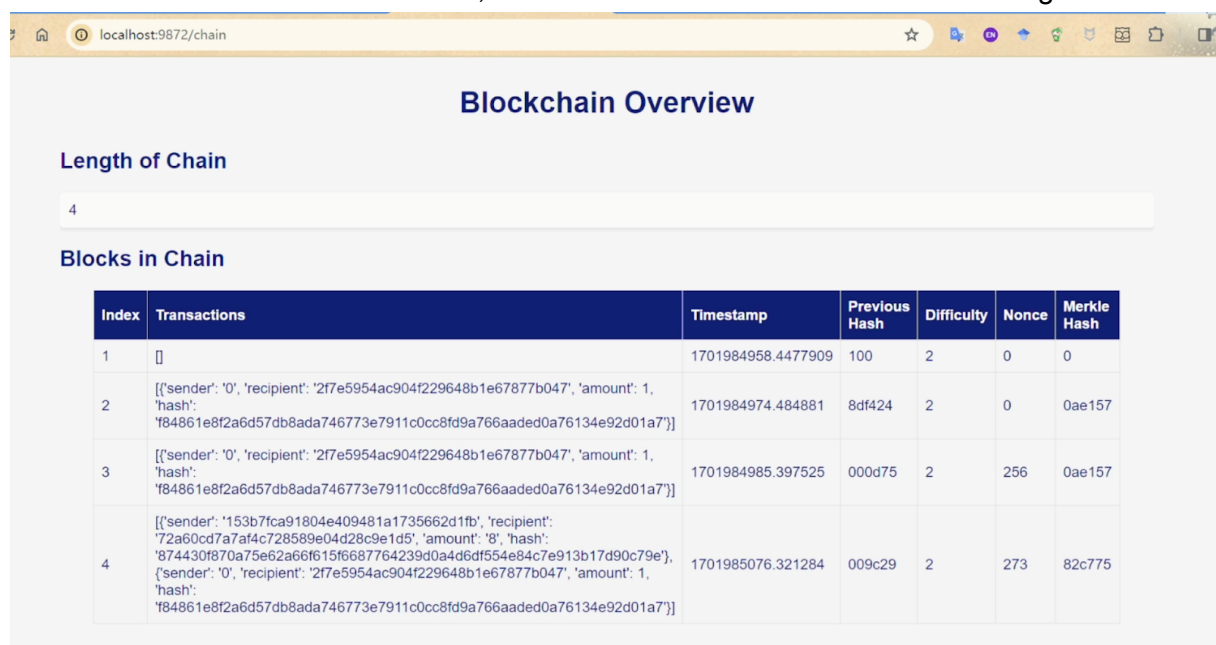


Figure 3. Full_chain() of presenting all block information, including each block's index, transactions, timestamp, previous hash, difficulty, nonce and Merkle hash

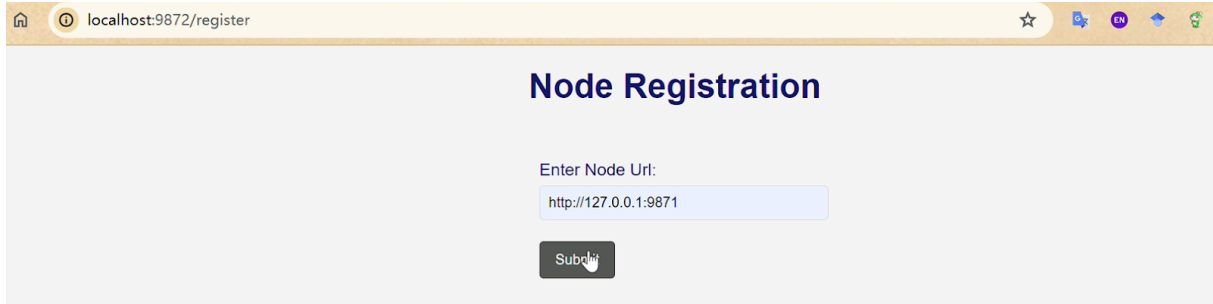


Figure 4. Register. Enter the target url and node 9872 will add 9871 to its neighbor set.

| Index | Transactions | Timestamp | Previous Hash | Nonce | Merkle Hash |
|-------|---|--------------------|---------------|-------|-------------|
| 1 | [] | 1701984960.763532 | 100 | 0 | 0 |
| 2 | [{"amount": 1, "hash": "d3310ff7f60b8f31b6e5adc04aec74b494425bbd02a41e046aae81e4ed9e6f8f", "recipient": "ea1095a07e1c410d8db9e7393d701b9f", "sender": "0"}] | 1701984966.6764872 | d8837b | 159 | 06e745 |
| 3 | [{"amount": "2", "hash": "f7c81926880483f685055809927da23ac60fe0d6d94c87d8fc54009b35504ec5", "recipient": "72a60cd7a7af4c728589e04d28c9e1d5", "sender": "003195972aa94ec094124c1509bd6fb8"}, {"amount": 1, "hash": "d3310ff7f60b8f31b6e5adc04aec74b494425bbd02a41e046aae81e4ed9e6f8f", "recipient": "ea1095a07e1c410d8db9e7393d701b9f", "sender": "0"}] | 1701985039.6672785 | 000602 | 18 | 4445a8 |
| 4 | [{"amount": "4", "hash": "f14e155007fe7abd04ce88f0c286b5040ff8fce39837b0d2ab236b5847b01311", "recipient": "72a60cd7a7af4c728589e04d28c9e1d5", "sender": "4202ea1b315a43bd8f7fe0c205a657b0"}, {"amount": 1, "hash": "d3310ff7f60b8f31b6e5adc04aec74b494425bbd02a41e046aae81e4ed9e6f8f", "recipient": "ea1095a07e1c410d8db9e7393d701b9f", "sender": "0"}] | 1701985053.390282 | 005639 | 522 | 389366 |
| 5 | [{"amount": 1, "hash": "d3310ff7f60b8f31b6e5adc04aec74b494425bbd02a41e046aae81e4ed9e6f8f", "recipient": "ea1095a07e1c410d8db9e7393d701b9f", "sender": "0"}] | 1701985105.118654 | 003c66 | 12 | 06e745 |

Figure 5. Resolve. If a node sees its neighbor has a more difficult block, it will replaced its blockchain with its neighbor, and displaying this new chain.

Those transactions that are given up are recycled, if they are already in the new chain, we will abandon them, if not, we will put them into mem pool for further mining.

More details can be found in the [video](#) (same as the one used in presentation), better use 1080p resolution.

a.