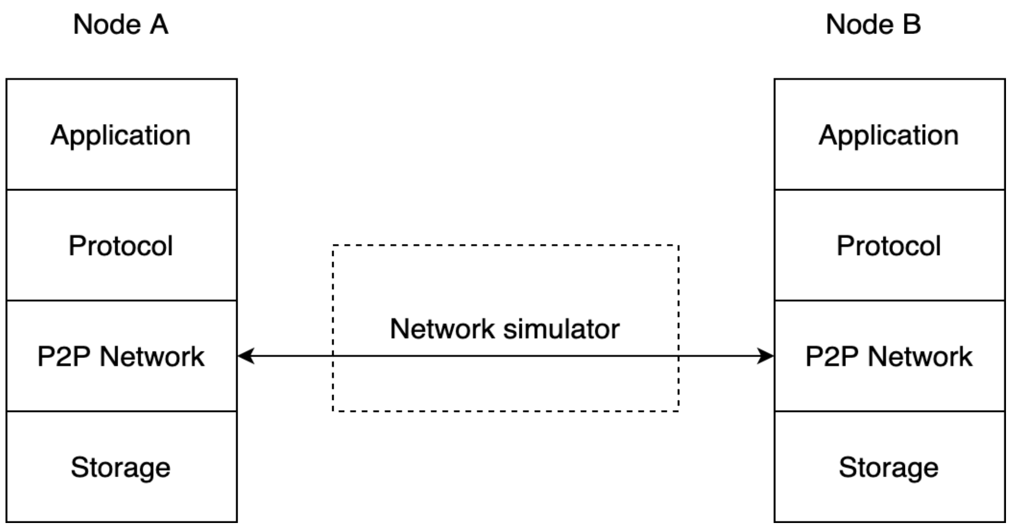
# Blockchain Simulation Report

## Introduction

This project aims to implement a simple blockchain system using the python programming language. Throughout the entire building process, we conducted investigations on real-world blockchain systems and borrowed some interesting ideas from them, besides the investigations, we also learned some programming features in the python libraries. This report is composed of the following sections, in the first section, we would discuss the system design at a high level; then we would discuss some challenges on the way towards a mature design in the section Challengers; we would also elaborate on the detailed implementation of our current blockchain system; finally, we would discuss some improvements that can be achieved in the future. We attached the commands on how to set up our blockchain system and how to interact with it at the end of this report.

## System Design

A traditional blockchain system can be divided into four parts, from a bottom-up point of view, they are storage layer, p2p network layer, consensus/protocol layer, and application layer. The blockchain nodes communicate with each other through the p2p network, specifically, if the node is running in a virtual machine hosted by a physical computer, it requires an internet simulator to set up the p2p network on top of VPN clients. The following graph shows the stack of such a blockchain system.



We break down the above system stack into the following details.

- **Storage Layer**: This layer holds all the data related to the blockchain state, which includes:

    - **Account** holds the mapping from an account to its balance in terms of cryptocurrency, it is worth noting that this account-based design is similar to the Ethereum that each node maintains the balance of all the account, other blockchain systems like Bitcoin, are using UTXO model to record the balance.
    - **Blockstore** stores the entire set of blocks and chains them all together.
    - **Keystore** stores the encrypted account state which contains the public/private key pair associated with the user ID, note this storage is distinct to different block nodes.
    - **TXstore** stores the transaction set through the entire network.
    - **UTXpool** stores pending transactions and waiting to be mined.

- **P2P Network layer:**
    - **P2P on top of TCP**: Similar to the traditional blockchain systems, the P2P network is building on top of the TCP stack, the blockchain node uses this network to broadcast the blockchain state such as the blocks and transactions, through this to reach a synchronized status. Specifically, Ethereum blockchain is using the so-called DEVp2p protocol which can provide encrypted communication.
    - **Internet simulator:** As described at the beginning, if our blockchain system is running atop virtual machines (VM) hosted by different computers, we would need an internet simulator to facilitate the establishment of a P2P network. The reason is the VM uses a private IP address that is only reachable to the host machine or the other VMs that reside in the same host, thus it is impossible for a VM to directly connect another VM that resides outside of the host. The internet simulator solves this problem by setting up tunnels with the VPN client runs on different VMs, and then performs the traffic routing for these VPN clients.

- **Protocol Layer**: In this layer, we implement the basic functionalities related to accounts, transactions, and blocks mining.
    - Account operation:
        - Create accounts will create a public/private key pair for the given user ID and password, and stores the encrypted key pair in one of the Keystore.
        - Unlock accounts will query the account file from the Keystore by the username and decrypt the account using the user-provided password. This step is the preliminary for a user to spend her coins by sending a transaction or doing the mining worker.
    - Transaction operation:
        - Create transaction will generate the transaction using the sender’s public key, and sign this transaction with the corresponding private key. The

        transaction will transfer several coins to the receiver, and the timestamp will be included in the transaction as well.
    - Transaction pool stores the pending transactions that will be included in a block in the future.
    - Mining:
        - When a node starts to do the mining work, it will collect the pending transactions from the transaction pool, then do the verification against the signature and check over the balance to prevent overdraft operations.
        - Block generation: each miner will run the Proof of work consensus, that is, and verify the block by its index, to avoid the collision that other miners already insert a block with that index during the mining process and local miner keep insert it.

- **Application Layer** on JSON-RPC: To enable the interaction between the user and the block node, we incorporate the JSON-RPC into the application layer. A user can send the following commands to the block nodes through HTTP messages.
    - Account related operations such as create accounts, unlock accounts, get balance.
    - Transaction related operations such as submit transactions, get transactions.
    - Block content queries such as get a block.
    - Mining operations: start/stop mining.

## Challenges

In this section, we discuss the challenges we have encountered during the initial stage of system design.

- **P2P layer**. In the beginning, we investigated the P2P module in the Ethereum blockchain since this module is standalone and supports unit test, we hope to take this off-the-shelf module and directly apply it into our system, however, after delving into it, we found it is not suitable for our system due to the following reasons. Given this, we decided to implement the P2P layer by ourselves, to implement the P2P module using python, the main challenge is how to run the code concurrently, on the one hand, a node would listen on a port and wait for connections from other nodes, on the other hand, this node would use an ephemeral port to connect to other nodes as well. This bidirectional communication channel should be handled simultaneously, thus it requires us to build an asynchronous socket programming in python.
- **Twisted library**. During the research on socket programming, we noticed there is a popular library called Twisted which provides asynchronous socket programming and supports user-defined application protocol, this library is really helpful for us to build the P2P protocol. So we spent some effort in learning Twisted and building a toy chatting room program, after gaining hands-on experience, we started to build our P2P protocol.
- **Internet simulator**. While we were building the P2P protocol using Twisted, we encountered a new challenge, that is, how to set up the P2P network on multiple VMs hosted by different computers, a naive way would be setting the port forwarding on the host machine, however, this

doesn't work for computers connected to the home routers, as it requires further port forwarding settings on the router which is quite complex. Instead, we have an alternative design by setting up a VPN network and connect each VM through the VPN tunnel, the idea is that we set up an internet simulator on a VMs, then the VMs will create a TUN interface and assign a private IP to this interface, any packets that sent to this interface will be sent through the VPN tunnel to the public server, after receiving this packet, the public server will decompose the packet and retrieve the destination IP, then write this packet to the corresponding connection from the destination IP address.

- **Protocol layer**:
    - Similar to Ethereum, we recorded each account's balance in the AccountDB in order to prevent overdraft transactions, that is, when receiving a transaction through the RPC or P2P, the node will first check if the signature matches the public key, then check if the transferred amount in the transaction exceeds the account's balance, by this means, a user cannot spend more than he owns.
    - Prevent transactions replay is common adoption in any real-world blockchain systems, for example, in Bitcoin's UTXO model, any transaction output can only be spent once by a successor transaction to prevent double-spending transactions, Ethereum handles this problem by attaching a nonce to each account, the nonce records the total number of transactions that the account has issued so fast, thus, by check if the nonce is monotonically increasing, it can prevent transaction replay. In our system, we solve this problem by searching a transaction from the transactionDB, if the transaction is already included, we will reject this transaction.
    - In addition, we need to prevent blocks replay as well, we solve this problem by checking PoW and the block height, in our blockchainDB, we maintain the current block height, and any lower height blocks will be rejected.

## Implementations

In this section, we first show the structure of our code, then we elaborate on the details of implementation.

### Structure of Code:

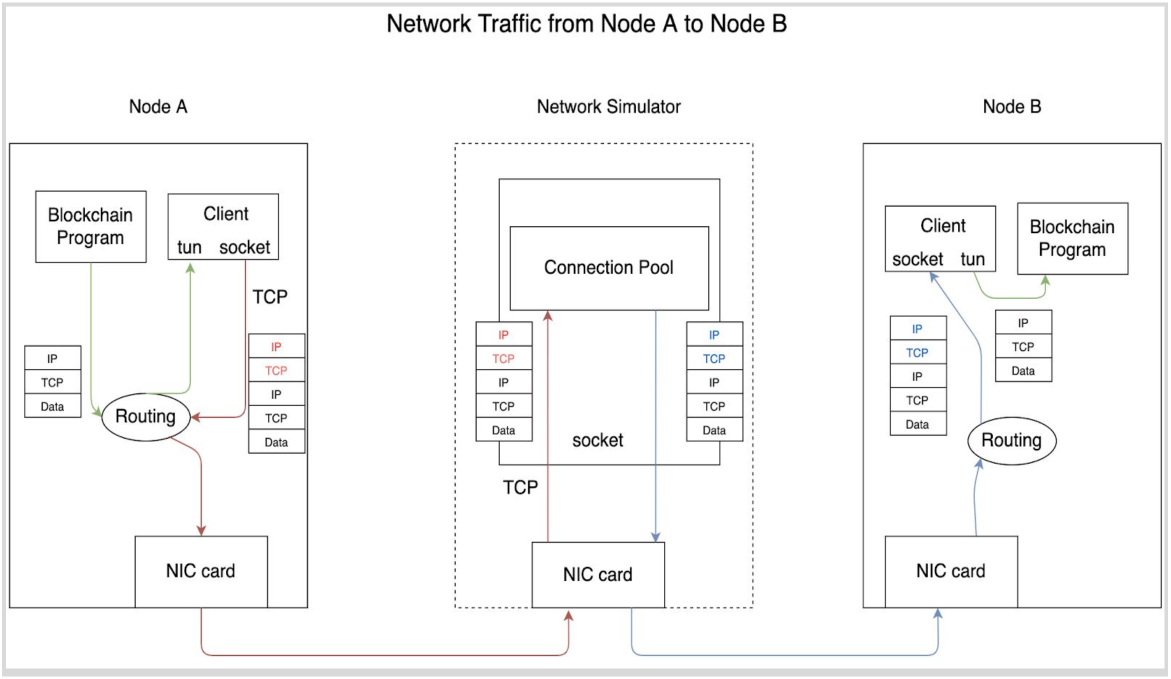
```
src
├── app      // protocol layer, including account, transaction, storage, and mining operations
├── bkc.py  // entrance
├── Data     // blockchain data, includes Account, Keystore, Blockstore, TXstore, UTXpool
├── internet_simulator // internet simulator server
├── Makefile // install python dependency
├── p2p        // p2p network
├── rpc         // JSON RPC server
├── test        // testing scripts
└── vpn        // vpn client
```

- **Storage Layer**

File location: **Data**
Currently, on each node, we create a Data directory to store all the blockchain states, inside this directory, in the design we already show the main purpose for each file.
Here is the structure for each object:

- Account: {public key: balance}; because our blockchain system is an anonymous system, so for each transaction will only include the public key for both sides of the transaction, and the user's username will never be shared with peers.
- Blockstore: {index, hash, tx, nouce, timestamp, miner, previous block hash}
- Keystore {public key: private key} and each file named by username, and encrypted by user's password, and files in this directory will not be shared.
- Txpool and UTXpool all store the transaction object: {hash, sender, timestamp, amount, receiver, signature}

- **P2P network layer**
  File location: **p2p, internet_simulator, vpn**
  - We use a python-based VPN client code to create a TUN interface and assign it with the user-provided private IP address. We also implemented the internet simulator on python, we drew the following diagram to show the packet flow among the blockchain nodes and internet simulator.



  - We implement the P2P protocol under the Twisted framework, and we start a TCP server by listening on the private IP and using the user-provided port (default is 5555). The protocol supports several types of messages, such as HELLO, PING/PONG, ADDR, TX, and BLOCK. Specifically, when a connection is established between a pair of peers, they both send the HELLO message to move forward the connection to READY state, without this type of message, the connection will be considered invalid and they will immediately drop the connection. After this, they will send ADDR messages to exchange peer info that they have

connected so far, this is quite similar with the P2P discovery mechanism. The PING/PONG messages are used to check the liveness of a connection, by default, every 20 seconds a node will initiate a PING message. TX, BLOCK messages are initiated by the consensus layer, when a block or transaction is created or received.
    - A P2P network is used to broadcast the messages and make each node to reach a synchronized state, to do the broadcast, each node when received a message, it should send it to all connected peers, except the one where it receives the message from. Excluding the original sender from the broadcast is quite important, as this can prevent infinite transactions/blocks propagation cycle. We are doing so by creating an array for each connection and record all the BLOCK/TX messages that have been received or sent, before the broadcasting, for each connection, we check whether this message has been sent to this connection, if not, we then send it.

- **Protocol layer**
  File location: **app**
    - In this layer, we are using python ECDSA library to generate a pair of public/private keys for the user and encrypt the keys by the user-provided password, only using the password can decrypt the account file for operations related to balance change like mining and sending transactions.
    - For blocks mining, the cores of this part are Proof of Work and block verification. Specifically, for Proof of Work, the miners will try their best to find a nonce, then concatenate the nonce with the block content and compute a hash over it, if the hash begins with “0x0000”, such nonce will be considered as a the solution to this puzzle. Here one can change the difficulty by adding more ‘0’s to the target hash; For the block verification, two problems have to be solved, one is the check if the transactions included in the block are valid, we solve this problem by verifying the transactions’ signature over the public keys. The other problem occurs when multiple nodes start to mining simultaneously, it is possible that the miners may find the same solution and broadcast their mined blocks to the network, how to decide the winner is tricky. To solve this problem, we proposed to decide the winner by selecting blocks with the smallest timestamp, in case that the timestamps are the same, blocks with larger nonce will be selected.

- **Application layer**
  File location: **rpc**
    - Similar to how Ethereum supports decentralized applications (Dapps), we support user’s application through JSON-RPC. This layer requires to install the **python-jsonrpc** library on the blockchain node. Specifically, on the blockchain node, this module will create a new TCP server to accept the HTTP JSON requests sent from the users, and after receiving the JSON requests, our application handler module will translate the messages to commands, and pass the commands to the blockchain protocol layer to execute the corresponding functions, such as create accounts, unlock accounts, send transactions, mining and so on.

## Future improvements

- **Storage Layer**
  - In our current design, we are storing all the accounts in one file, and so does for the block contents, and transactions. The design works well for a small blockchain where the state is relatively small. However, when the blockchain system is deployed on a large scale, this design has a significant drawback, as it requires every node to load and write to these files, whose size will become larger and larger as time goes by.
  - To mitigate the above problem, a favorable design would be storing the blockchain state in SQL, we can create several databases to store the accounts, blocks, and transactions, the advantage of using SQL is it supports read and write quite efficiently.
- **P2P network layer**
  - Unlike real-world blockchain systems, e.g., Bitcoin and Ethereum, our current design doesn't limit the number of connected peers, that's to say, we allow as many connections as possible on one node. This design is not suitable for a large blockchain network, suppose the network size increased to 1000 nodes, using our design, every two nodes mutually have a connection, then they have to maintain 1000 connections and read from or write to those connections will be quite cumbersome and un-efficient.
  - To solve the problem, on the P2P layer, we can add a hard limit to the total live connections with different peers, so that if the number of connections exceeds this limit on one node, it forbids any connections from other nodes.
- **Protocol layer**
  - Fast catch-up. In the current design, we require all blockchain nodes joining the network almost at the same time, and only after all nodes are properly started, then we start mining. However, in real world, a blockchain node may join the network after the mining, or exit the network for a while then rejoin. For such nodes, they should catch up with other running nodes by downloading the missed blocks, therefore, supporting block downloading is necessary for a newly joined blockchain node..
- **Application layer**
  - So far, our application handler module supports the basic functionalities as described in the implementation, in the future, we could add more features into this module to support more rich functions, such as some aggregation queries (get all transactions sent by one account), manage connected peers (reconnect, disconnect and so on).
  - Another big improvement would be the support of smart contracts, quite like Ethereum blockchain, it supports executing users' deployed code in a so-called Ethereum Virtual Machine. This feature can greatly enlarge the categories of Dapps.

## Test the program

In this section, we illustrate how to run our programs on the testing platform.

### 1. Preparation

- We are using **Fresh SEED Ubuntu 16.04 VM** (32-bit) as the testing platform. To scale up the blockchain network, you may run multiple VM instances in different hosts. We have tested with 5 VMs that run as blockchain nodes, and 3 of them can do mining simultaneously.
- Find a **public server**. We have to run the internet simulator on a public server. In case you have difficulty accessing a public server, we have a long-run internet simulator running on **128.230.208.73:55555**, you can directly use it.
- If you are using our public server and you host the VMs on Mac OS, you need to do additional settings on the Virtual Box dashboard, you need to set the network adapter to **Bridged Adapter** mode, and then make sure you have your computer connected to SU VPN server. These additional settings are not required for Window OS.

### 2. Download the code from Github.

- If you want to run your own public server, use the following commands to start.
  - *git clone 'https://github.com/likai1993/blockchain_simulation'*
  - *cd blockchain_simulation/src/internet_simulator*
  - *./server.py 55555*
- On each SEED VM, run the following commands.
  - *git clone 'https://github.com/likai1993/blockchain_simulation'*
  - *cd blockchain_simulation/src/*
  - *make*
  - *(optional) make clean: if want to clean old data and start from fresh*

### 3. Start the blockchain nodes.

- For each VM, you need to select a private IP address, we limit that all the private IPs should use '**192.168.**' as the prefix. For example, you can select **192.168.53.5** as the bootstrap node's IP.
- Start the bootstrap node on the first VM.
  - *sudo ./bkc.py --listen  '192.168.53.5'*
- Start the other nodes on other VMs.
  - *sudo ./bkc.py --listen  '192.168.54.5'*
  - *sudo ./bkc.py --listen  '192.168.55.5'*

    - *sudo ./bkc.py --listen  '192.168.56.5'*
    - *sudo ./bkc.py --listen  '192.168.57.5'*
- Note that start the nodes with the above commands, they will connect to our Internet Simulator (IM), if you are running yours, then you should append the *--imAddr, --imPort* options.
    - *--imAddr 'your IM IP'  --imPort 'your IM port'*
- After starting the nodes, check the output and see if they are mutually connected, you are expected to see **Connected clients**, and **PING/PONG** messages in every 20 seconds.

## 4. Create accounts.

- Open a new terminal in the first VM and execute the following commands.
    - *cd blockchain_simulation/src/test*
    - *./createAccount.sh 'test1' '123' '53.5'*
    - *./createAccount.sh 'test2' '123' '54.5'*
    - *./createAccount.sh 'test3' '123' '55.5'*
- The above commands will create one account on the first three VMs, the arguments are **user ID**, **password**, and the **IP suffix** you configured for each VM. In the response message, you will see **each user's public key,** denoted as *PUBLICKEY1*/*PUBLICKEY2*/*PUBLICKEY3*. Remember the public keys and they will be used in the following test.

## 5. Start mining.

- Use the same terminal to execute the following commands.
    - *./mining.sh 'test1' '123' '53.5'*
    - *./mining.sh 'test2' '123' '54.5'*
    - *./mining.sh 'test3' '123' '55.5'*
- The above commands will start the mining worker on the first three VMs, the arguments are **mining reward receiver**, **password**, and the **IP suffix** you configured for each VM.
- After starting mining, you are expected to see the first three nodes can mine some blocks, it is very likely some of them may mine the same block, our system will select the block with the earliest timestamp. In other nodes, you can see logs of broadcast blocks and receive blocks.

## 6. Stop mining.

- After mining for 2 minutes, execute the following commands in the same terminal.
    - *./stopMine.sh '53.5'*
    - *./stopMine.sh '54.5'*
    - *./stopMine.sh '55.5'*

- The above commands will stop the mining worker on the three VMs, you are expected to see no more blocks will be mined.

## 7. Check mining rewards.

- Execute the following commands in the same terminal to check the balance.
  - *./getBalance.sh 'PUBLICKEY1' '53.5'*
  - *./getBalance.sh 'PUBLICKEY2' '54.5'*
  - ...
- In the above commands, you can pass any one of the 3 public keys, and execute the above commands in any one of the 5 IP suffixes, all the 5 nodes should return the same balance for the same public key, that means all nodes are synchronized.

## 8. Send a transaction.

- Execute the following commands in the same terminal and it will return a transaction HASH.
  - *./sendTx.sh 'test2' '123' 'PUBLICKEY3' '20' '54.5'*

  You can check this transaction with its hash:
  - *./getTx.sh 'HASH' '53.5'*
- In the above commands, the second account/PUBLICKEY2 will transfer 20 coins to the third account/PUBLICKEY3.
- Check their balance.
  - *./getBalance.sh 'PUBLICKEY2' '53.5'*
  - *./getBalance.sh 'PUBLICKEY3' '54.5'*
  - ...
- You will see that the balance of each account is the same as step 7. That is because no one is mining, thus the transaction is not finalized in any blocks, it is a pending transaction at this point.
- Start mining again.
  - *./mining.sh 'test1' '123' '53.5'*
- After one block is mined, stop mining and check the balance again.
  - *./stopMine.sh '53.5'*
  - *./getBalance.sh 'PUBLICKEY2' '53.5'*
  - *./getBalance.sh 'PUBLICKEY3' '54.5'*
- At this point, you will find the second account has decreased 20 coins, and the third account has increased 20 coins compared to step 7.

## Demo video

https://drive.google.com/file/d/1iIF6T5D96Gn9TJIjCErYG6vNi9BQjtSU/view?usp=sharing