CS 436 – Project1: Network programming to implement Bitcoin
100 points + 10 points extra credit

1. Bitcoin:

Bitcoin is a cryptocurrency. It is a decentralized digital currency without a central bank or single administrator that can be sent from user to user on the peer-to-peer bitcoin network without the need for intermediaries. In this project, we will implement a simplified Bitcoin system.

Bitcoin is used as a currency. The users can buy and sell things using their own bitcoins. How do the users initially get some bitcoins? There are companies where a user can buy some bitcoins to start with. Then the user can start making transactions and pay bitcoins to buy things. Note that bitcoin is not a shopping system. It is only used to send and receive bitcoins using transactions. It is similar to the concept of your credit account in a bank system. However, there is a difference between bank system and bitcoin system. The bank system is centralized. All transactions are stored in the bank database and centrally managed by the bank system. But the bitcoin system is distributed. The transactions are stored in the blockchain, and all full nodes have a copy of that.

The bitcoin system contains a group of full nodes. Each full node stores a copy of the blockchain. The users use a client program to record and keep track of their transactions. Each client connects to one or more full nodes. Each full node also connects to some other full nodes. In this way, each full node connects to all other full nodes, either directly or through intermediate full nodes.

Each full node stores a list of temporary transactions. Once a user records a new transaction, its client program sends the transaction to its full node(s). Then the full node stores that in its list of temporary transactions and forwards the transaction to its neighbor (directly connected) full nodes. They also do the same thing. Shortly, the transaction will be distributed all across the full node network and stored in the list of temporary transactions of all full nodes. But note that the transaction is not confirmed yet. A transaction could be confirmed once it is added to the blockchain. Someone should put the transaction in a block of transactions, mine the block, and add it to the blockchain.

Each full node, potentially, is a miner. When there is enough number of blocks in the list of temporary transactions, a miner puts them together, and starts the mining process. Mining in bitcoin takes almost 10 minutes. The stronger miners are able to do the mining process faster than others and win the competition. As soon as a miner finishes the mining process, it distributes the mined block to the full node network. When other miners receive this block, they stop their mining process and store it as the new block in blockchain. Then a new round starts.

What is the miner's incentive to do all this hard work? The rewards. If the miner wins the mining competition, it will be rewarded by a mining fee. Also, each client who makes a new transaction pays a transaction fee. The transaction fees of all transactions in a block will be collected by the miner. Where does the mining fee come from? Those are the new minted bitcoins in the system.

2. Python3

You will implement this project in Python3. How to learn Python3? There are many great tutorials on the Internet. The following is a link to a very easy and helpful tutorial for beginners with hands-on examples in the most commonly used PyCharm IDE. Study the first 3 hours of the video.
https://www.youtube.com/watch?v=_uQrJ0TkZlc&t=16696s

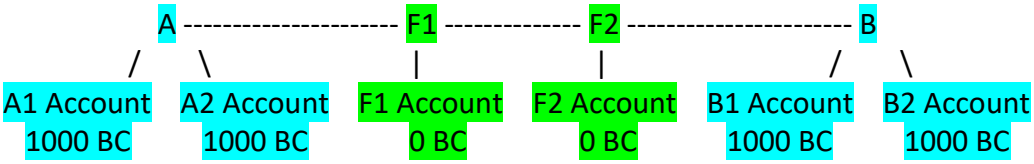
In addition to that, you need to learn some special topics in Python3, e.g. read and write to files. Then, the playlist tutorial by Corey Schafer is great to learn those topics with examples.
https://www.youtube.com/watch?v=YYXdXT2l-Gg&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU
Files tutorial: https://www.youtube.com/watch?v=Uh2ebFW8OYM
OS Module tutorial: https://www.youtube.com/watch?v=tJxcKyFMTGo

3. Socket programming in Python

Refer to the socket programming lecture and video. Also, a simple socket programming example is given in the project section of the course. Download the attached folder. I have also provided an instruction to run the files in the project section. Implement this project with **UDP**.

4. Implement the bitcoin system.

In this project, we assume there are two full nodes (F1 and F2) and two clients (A and B). Each client connects to one full node: A connects to F1, and B connects to F2. Each client has two accounts and each full node has one account. The initial balance of each client account is 1000 BC (bitcoin) and the initial balance of each full node account is 0 BC (bitcoin).



Client A can record a transaction with either of its accounts. Then A sends its transaction to its full node F1. Then F1 stores that in its list of temporary transactions and also sends it to F2. Then F2 also stores that in its list of temporary transactions. When there are 4 transactions in the list of temporary transactions, a full node will remove them from the list and mine a block with them. Here, for simplicity, we don't implement competition between miners. F1 mines the odd blocks and F2 mines the even blocks. Once mined, the miner full node stores the block in its blockchain and sends the mined block to the other full node. It also sends the corresponding transactions to its client. The corresponding transactions are the ones in which the client is a payer or a payee. This lets the client know that these transactions are confirmed now. Once the other full node received the block, it stores that in its blockchain and sends the corresponding transactions to its own client. When a client receives the confirmed transactions, it first makes sure that the received transaction is the same as the transaction that the client has recorded earlier. Then it appends it to the its list of confirmed transactions and updates its balance.

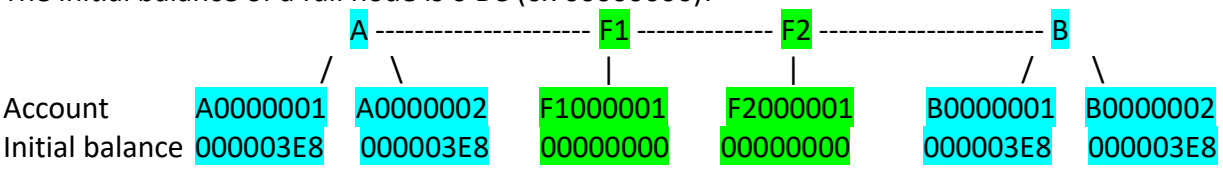
HEX (base 16)

A HEX number is a sequence of digits. Each digit is 4 bits in the range {0, 1, 2, …, 9, A, B, …, F}. Thus, each byte (8 bits) is two HEX digits. For example, 0x 4A93B5FC is a 4-byte HEX number. The notation used to demonstrate a HEX number is 0x.

The structure of an account and its balance:

An account is a 4-byte integer in HEX format. We predefine the accounts for nodes.
Client A has two accounts: 0x A0000001, 0x A0000002.
Client A has two accounts: 0x B0000001, 0x B0000002.
Full node F1 has one account: 0x F1000001.
Full node F2 has one account: 0x F2000001.

The balance of an account is a 4-byte integer in HEX format.
The initial balance of a client account is 1000 BC (0x 000003E8). Thus, the total initial balance of a client is 2000 BC (1000 BC for each account).
The initial balance of a full node is 0 BC (0x 00000000).



The structure of a Transaction

In real bitcoin system, a transaction could be very complex. In our bitcoin system we define a very simple structure for a transaction. A transaction (Tx) is created by a client. It contains 3 fields.

1. Payer account
2. Payee account
3. Transaction amount (Tx_amount)

Each of these fields is a 4-byte integer in HEX format. Thus, a transaction is a 12-byte HEX. For example, A0000001B00000010000000E is a Tx created by client A. The payer is A's first account (A0000001), the payee is B's first account (B0000001), and the Tx amount is 14 BC (0000000E).

We don't explicitly mention the transaction fee (Tx_fee) in the Tx. However, a Tx_fee of 2 BC (00000002) will be paid from the payer account (A0000001) to the miner once a block containing this Tx is mined by a miner, which will be either F1 or F2.

In our bitcoin system, a client never pays itself. It means in one Tx, the Payer and Payee accounts cannot be the same or belong to the same client.

Each full node account has a balance, which is initialized to 0x 00000000. Once a full node mines a block, two types of rewards will be payed to its account's balance:

1. 30 BC mining fee (0x 0000001E).
2. A total of 8 BC Tx_fee, which contains 2 BC Tx_fee for each of the 4 Tx of the block.

The client files

The client stores its Tx and balance information in files. The reason is we will run two different programs on the client machine, and they both need to access and manipulate the same values. Therefore, we store them in files to maintain consistency between the two programs.

The client Tx files

The client stores its transactions in two files:

1. Unconfirmed_T. txt: To store the unconfirmed transactions.
2. Confirmed_T.txt: To store the confirmed transactions.

The balances of a client account

Each client has two accounts and each client account has two balances:

1. Unconfirmed_balance: To store the balance after all transactions.
2. Confirmed_balance: To store the balance after all confirmed transactions.

The confirmed transactions are the ones that have been added to the blockchain.

All client balances are initialized to 1000 BC (0x 000003E8).

1. Unconfirmed_balance

This is used for new transactions that have not been entered to the blockchain yet.

- When a user enters a new Tx, the client takes the following steps.
  1. It checks whether {Tx_amount + Tx_fee} would not exceed the current value of Unconfirmed_balance, then reduces the Unconfirmed_balance by {Tx_amount + Tx_fee}.
  2. It appends the Tx to a file containing its unconfirmed transactions (Unconfirmed_T. txt).
  3. It sends the Tx to the full node.
- When a client receives a confirmed Tx from its full node:
  1. If the Tx Payee is one of the client's accounts, it adds the Tx_amount to the account's Unconfirmed_balance.

2. Confirmed_balance

This is used for confirmed transactions, which have been entered to the blockchain.

- When a client receives a confirmed Tx from its full node:
  - If the Tx Payer is one of the client's accounts:
    1. It will reduce the {Tx_amount + Tx_fee} from the account's Confirmed_balance.
    2. It will remove the Tx from the file containing its unconfirmed transactions (Unconfirmed_T. txt).
    3. It appends the Tx to a file containing its confirmed transactions (Confirmed_T. txt).
  - If the Tx Payee is one of the client's accounts:
    It will add the Tx_amount to the account's Confirmed_balance.

The client balance file

The client stores the current values of its balances for each of its two accounts in a file named balance.txt. The values in the files are initialized at the beginning and updated while running the program. I suggest the following structure for the file. You are free to select another structure.

Account1:Unconfirmed_balance:Confirmed_balance
Account2:Unconfirmed_balance:Confirmed_balance

For example, for Client A, the file will be initialized as following. We store the values in HEX.

A0000001:000003E8:000003E8
A0000002:000003E8:000003E8

The client implementation

The client runs two programs:

1. Client_send.py: To send Tx to the full node.
2. Client_receive.py: To receive Tx from the full node.

1. Client_send.py

In this program, the client is in the roll of a client. It connects (as a client) to its full node (as a server). You should bind the full node program to a port number, e.g. 10000 for F1.
In an infinite loop, the program displays a menu and asks the client to enter a choice:

1. Enter a new transaction.
2. The current balance for each account.
3. Print the unconfirmed transactions.
4. Print the last X number of confirmed transactions (either as a Payee or a Payer).
5. Print the blockchain (Get the blockchain from the full node and print it in a structured format block by block. Separate the fields of each block as well: **10 points Extra credit**).

For entering a new transaction:
The Payer could be one of the client's accounts. The Payee could be one of the other client's accounts. At each step, provide a list of possible accounts to the user and ask them select the accounts from the list. Once the user entered a new Tx, the client takes the following steps.
a. It checks whether {Tx_amount + Tx_fee} would not exceed the current value of Unconfirmed_balance, then reduces the Unconfirmed_balance by {Tx_amount + Tx_fee}.
b. It appends the Tx to its Unconfirmed_T. txt
c. It sends the Tx to the full node.

2. Client_receive.py

In this program, the client is in the roll of a server. The full node (as a client) connects to this program (as a server). You should bind this program to a different port number, e.g. 20000 for A. (Note: You might be able to combine these 2 programs with multi-threading and signal handlers, but it is beyond the scope of this course).

In an infinite loop, the program listens to the full node. Once it received a confirmed Tx, it checks whether one of its accounts is a Payer or a Payee in the Tx.

a. If it is a Payer:

1. It makes sure that the Tx is available in its Unconfirmed_T. txt
2. It reduces the {Tx_amount + Tx_fee} from the account's Confirmed_balance.
3. It removes the Tx from its Unconfirmed_T. txt
4. It appends the Tx to its Confirmed_T. txt

b. If it is a Payee:

1. it adds the Tx_amount to the account's Confirmed_balance and Unconfirmed_balance.
2. It appends the Tx to its Confirmed_T. txt

The full node files

A full node has two files.

1. Temp_T.txt to store the list of temporary transactions.
2. Blockchain.txt to store the blockchain

The full node implementation

The full node runs one program, which is a server. It binds its program to a port number, e.g. 10000 for F1 and 11000 for F2.

Recall that the full nodes take turns to mine blocks. F1 mines odd blocks and F2 mines even blocks. For this purpose, define a variable named "turn", and initialize it to 1 in F1 and 2 in F2.

Each full node has a balance, which is initially 0 BC. When a full node mines a block, it gets rewarded by the Tx fees and mining fee, which will be added to the full node balance.

The full node program works as following:

In an infinite loop, the program listens to the requests for connections, which could be from its client or the other full node.

1. If the requester is its client, the received message is a Tx, run the **TTT instructions** given below.
2. If the requester is the other full node, the received message is either a Tx or a block.
    1. If the message is a Tx, run the **TTT instructions** given below.
    2. If the message is a block:
        a. Append the block to the blockchain.txt file.
        b. Remove the 4 Tx of the block from Temp_T.txt
        c. Check the 4 Tx of the block and send the Tx where its client is a Payer or Payee to the client. The purpose is to let the client know these Tx are confirmed.

TTT instructions:

1. Receive the Tx and append it to the Temp_T.txt.
    - If the requester is the full node's client, send the Tx to the other full node as well.
2. If the number of Tx in Tempt_T.txt has reached 4, increment "turn" by 1.
    1. If (turn%2==1), it is the other full node's turn to mine, exit.
    2. Otherwise, it should mine the new block.
        1. Remove the 4 Tx of the block from Temp_T.txt and mine a block with them.

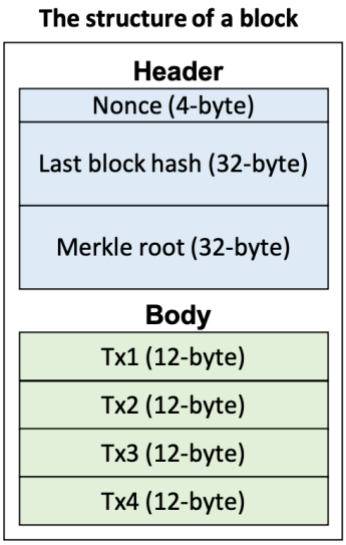
2. Add the mining fee (30 BC) and the total Tx_fee (8 BC) to the full node balance.
3. Append the block to its blockchain.txt
4. Check the 4 Tx of the block and send the Tx where its client is a Payer or Payee to the client. The purpose is to let the client know these Tx are confirmed.
5. Send the block to the other full node.
6. Print the block on the screen and exit.

The structure of a block

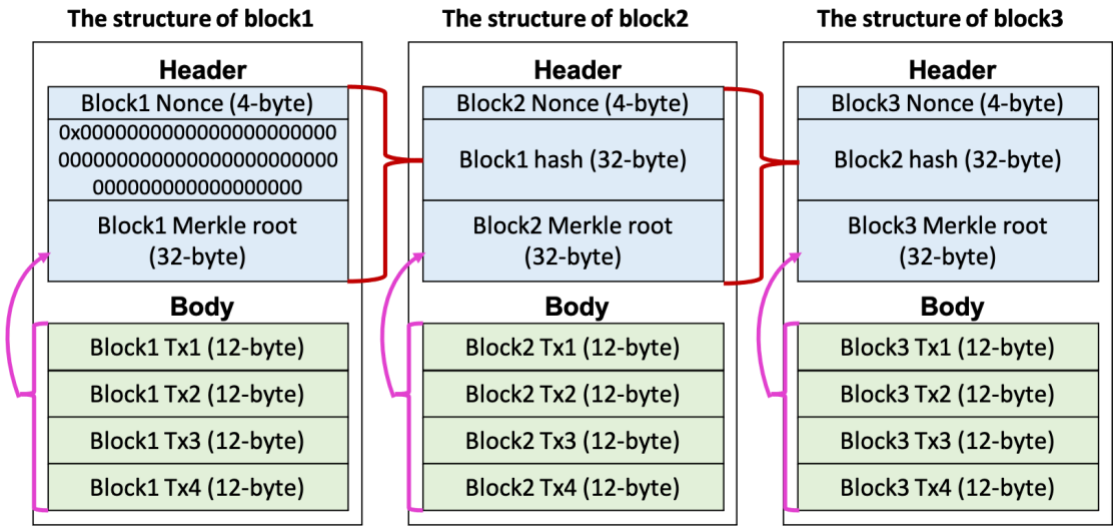
A block contains a Header (68-byte) and a Body (48-byte).

1. Header: a. Nonce (4-byte); b. Last block hash (32-byte); c. Merkle root (32-byte).
2. Body: 4 Tx, 12-byte each.

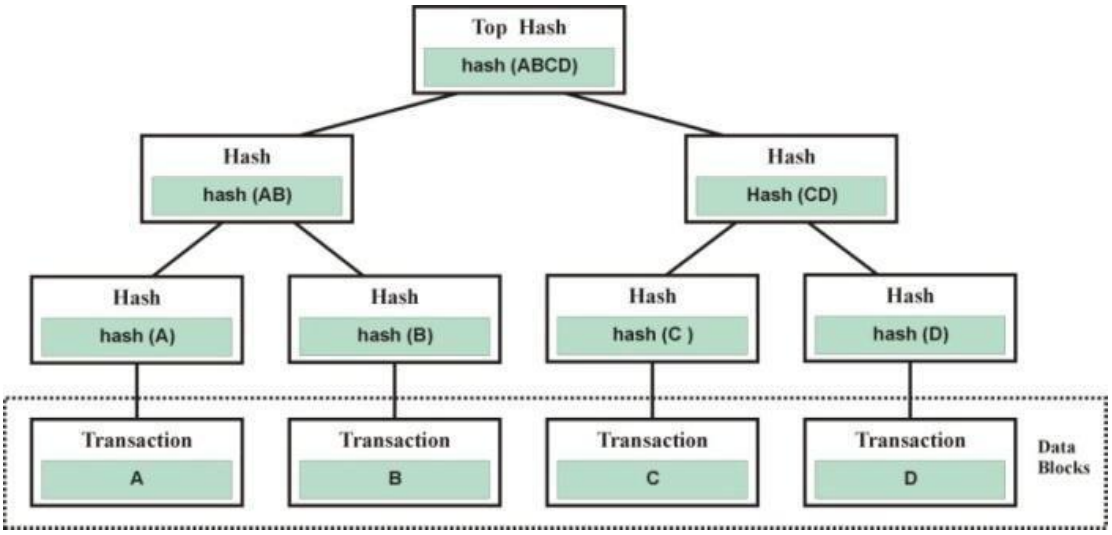
All values are stored in HEX format. The entire block is stored as a 116-byte HEX (232 digits).



Blockchain



Merkle root



Each block contains 4 Tx: A, B, C, D. We use a Merkle tree to hash the block. In the given figure:

In the first step, hash each Tx to create the leaves of Merkle tree.
hash(A) = Hash (Tx A)
hash(B) = Hash (Tx B)
hash(C) = Hash (Tx C)
hash(D) = Hash (Tx D)

In the next step, we concatenate two hashes and hash the result. Use + operator to concatenate.
hash(AB) = Hash (hash(A) + hash(B))
hash(CD) = Hash (hash(C) + hash(D))

In the final step, we concatenate the two new hashes and hash the result to get the Merkle root.
hash(ABCD) = Hash (hash(AB) + hash(CD))

The Merkle root is the calculated hash(ABCD).
We use SHA256 hash function in Python3. The result will be 256 bits, which is 32-byte HEX.

Use the following commands to calculate SHA256 hash of a string containing “message”. You can try other strings as well. Observe that the hash function always returns a 32-byte result. Also, for the same string, the hash result is always the same.

```
$ python3
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update("message".encode("utf-8"))
>>> m.hexdigest()
'ab530a13e45914982b79f9b7e3fba994cfd1f3fb22f71cea1afbf02b460c6d1d'
>>> exit()
```

Mining a block

To mine a block, you need the last block hash

1. If this is the first block, the "Last block hash" is a 32-byte HEX all zeros. Otherwise, take the last block from the blockchain and hash its header. This will be the "Last block hash".
2. Take the 4 Tx from Temp_T.txt and find their "Merkle root".
3. Find a correct "Nonce".
4. Concatenate the header and Body fields and store the block as a 106-byte Hex.

What is a correct Nonce?

Concatenate all fields of the block header and hash it. This is called the "block hash". We do not store it in the block. We can calculate it anytime. The block hash has a requirement. It should be less than a given number. In real bitcoin system, this number changes over time, so that the average time to mine a new block always remain 10 minutes despite the advances in processors speeds. In our bitcoin system, the requirement is the "block hash" should start with 2 byte 0's, which is 0x 0000. For example, if we found a Nonce that results in a block hash as following, that will be a correct Nonce.

0x 0000e0920bf43f0c9e3a0184a48a75d75165fc35a9443e47943106d0432a0e5f

Watch the linked video and refer to the linked demo webpage to learn more about blockchain.

I wrote the following simple program, which calculates a correct Nonce for some imaginary block. Note that we hash the block header, not the entire block. You can use the structure of this program in your project. Here, I simply entered some strings for the last block hash and Merkle root. In your program, you should replace these with real values of your block.

```
import hashlib

hashHandler = hashlib.sha256()
nonce = 0
while True:
    block_header = str(nonce) + "Last block hash" + "Merkle root"
    hashHandler.update(block_header.encode("utf-8"))
    hashValue = hashHandler.hexdigest()
    print('nonce:{0}, hash:{1}'.format(nonce, hashValue))
    nounceFound = True
    for i in range(4):
        if hashValue[i]!='0':
            nounceFound = False
    if nounceFound:
        break
    else:
        nonce = nonce + 1
```

The result is 49737.

```
nonce:49737, hash:0000e0920bf43f0c9e3a0184a48a75d75165fc35a9443e47943106d0432a0e5f
```

Some useful commands to convert strings and integers to HEX and vice versa are given below.

```
>>> "hello".encode("hex")
'68656c6c6f'
>>> "68656c6c6f".decode("hex")
'hello'
>>> int('0x10AFCC', 16)
1093580
>>> hex(1093580)
'0x10afcc'
```

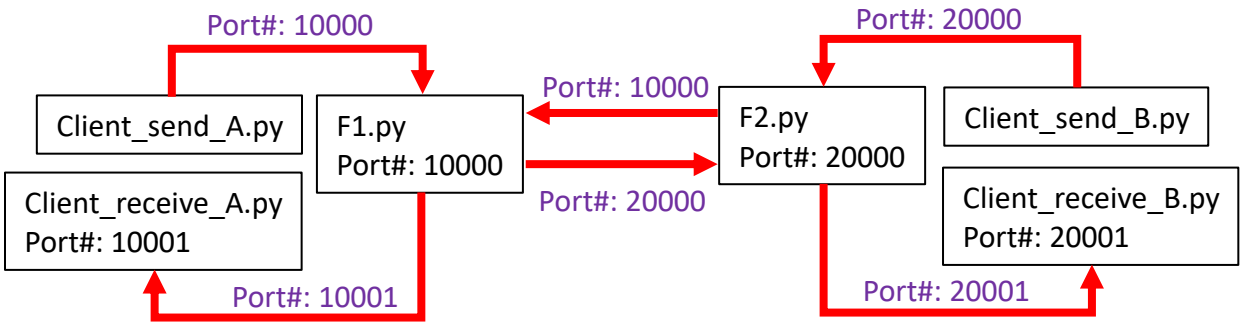
Notes about mining implementation:

Implementing the mining part of the project is 15 points. If you struggle implementing that, first focus on other parts of the project and then try to implement the mining part.

Notes about running and testing your program:

Working with Virtual Machines could be difficult because you may not have access to a convenient IDE to work with Python. You can run all programs on your own machine (localhost) without any virtual machine. In that case, make sure that you enter appropriate port numbers for client and server programs.

Notice that you define a port number for a server program (not a client program). A client program is the program who initiates a connection with a server program. Let's assume you define the following port numbers for the server programs:
Port number 10000 for F1.py
Port number 10001 for Client_receive_A.py
Port number 20000 for F2.py
Port number 20001 for Client_receive_B.py
When A (client) connects to F1 (server), A sends its messages to F1, port 10000.
When F2 (client) connects to F1 (server), F2 sends its messages to F1, port 10000.
When F1 (client) connects to A (server), F1 sends its messages to A, port 10001.
When F1 (client) connects to F2 (server), F1 sends its messages to F2, port 20000.



Test your program

Step 1. Run the programs of full nodes F1 and F2.
Step 2. Run the two programs of A client and the two programs of B client.

Display a menu to the client as following. Take a screenshot.

```
Please make a choice from the following selection:
 1: Enter a new transaction.
 2: The current balance for each account.
 3: Print the unconfirmed transactions.
 4: Print the confirmed transactions.
 5: Print the blockchain.
 6: Exit.
 Choice:
```

Now, the user starts entering the transactions. You should provide a menu of possible Payers and Payees and ask the user to enter a choice. The user enters the amounts in decimal. You should convert them to HEX in your program. A sample run is as following. Take a screenshot.

```
Select the Payer:
1. A0000001
2. A0000002
Choice: 1
Select the Payee:
1. B0000001
2. B0000002
Choice: 1
Enter the amount of payment in decimal.
207
Tx: A0000001 pays B0000001 the amount of 207 BC.
```

If the user enters some value above the Payer's balance, the program should immediately display a message and exit the transaction process. Take a screenshot.

```
Enter the amount of payment in decimal.
1500
Insufficient funds.
```

Step 3. The user at A client enters the following transactions. After you entered the 3rd transaction and before you enter the 4th one, run choices 2, 3, 4 to print the current balances, the unconfirmed and confirmed transactions. Take screenshots. Now enter the 4th transaction.

```
Tx: A0000001 pays B0000001 the amount of 207 BC.
Tx: A0000001 pays B0000002 the amount of 193 BC.
Tx: A0000002 pays B0000001 the amount of 161 BC.
```

Tx: A0000002 pays B0000002 the amount of 239 BC.

We expect that F1 mines a block at this point and sends it to F2. Notice that F1 mines odd blocks and this is block number 1, thus it is expected to be mined by F1.

Step 4. The users at A and B clients request the following reports and the programs print them.

1. The current balance for each account. Take a screenshot.
2. Print the unconfirmed transactions. Take a screenshot.
3. Print the confirmed transactions (either as a Payee or a Payer).
4. Print the blockchain (Get the blockchain from the full node and print it in a structured format block by block. Separate the fields of each block as well: **10 points Extra credit**).

The printed blockchain should look like the following. Take a screenshot.

Block: 1
Nonce (4-byte): 0003835
Last Block hash (32-byte): 00000000000000000000000000000000000000000000000000000000000000000
Merkle root (32-byte): ed0635432ab365cd35ab335f388532a32455d35c5a3583f345e3523c235a3cef
Tx1 (12-byte): A0000001 paid B0000001 the amount of 207 BC.
Tx2 (12-byte): A0000001 paid B0000002 the amount of 193 BC.
Tx3 (12-byte): A0000002 paid B0000001 the amount of 161 BC.
Tx4 (12-byte): A0000002 paid B0000002 the amount of 239 BC.

Step 5. The user at B client enters the following 4 transactions.

Tx: B0000001 pays A0000001 the amount of 135 BC.
Tx: B0000001 pays A0000002 the amount of 115 BC.
Tx: B0000002 pays A0000001 the amount of 138 BC.
Tx: B0000002 pays A0000002 the amount of 112 BC.

We expect that F2 mines a block at this point.

Step 6. The user at A client enters the following 2 transactions.

Tx: A0000001 pays B0000001 the amount of 47 BC.
Tx: A0000002 pays B0000002 the amount of 35 BC.

Step 7. The users at A and B clients request the following reports and the programs print them. Take screenshots.

1. The current balance for each account. Take a screenshot.
2. Print the unconfirmed transactions. Take a screenshot.
3. Print the confirmed transactions (either as a Payee or a Payer). Take a screenshot.
4. Print the blockchain containing the two blockchains. Take a screenshot.

Step 8. The user at B client enters the following 4 transactions.

Tx: B0000001 pays A0000002 the amount of 71 BC.
Tx: B0000002 pays A0000001 the amount of 29 BC.

We expect that F1 mines a block at this point.

Step 9. For client A, print the confirmed transactions (either as a Payee or a Payer). Take a screenshot.

Submission

Create a report file named YourTeamMembers. Enter the names of your team members at the beginning. Run the test steps given above. At each step, take a screenshot of the outputs and add them to the report. Enter the step numbers for each screenshot.

Put all your Python files in a folder named YourTeamMembers. Compress it in a zip file.
Submit both the report and the zip file.