Distributed System: Project 2 Report

1 Part1: Implementation Details

1.1 Client

The code for `Client` is mainly in *client.go*, which is acting like an test file. In that file, you will create some `wallet` and define their actions like "A transfer AMOUNT to B". Since our implementation is based on UTXOSet, the transaction should includes multiple `TxIn` and multiple `TxOut` (explained later). In our implementation, a wallet will not maintain a `UTXOSet` it self, he needs to get one from one `Miner` and then create corresponding `Transaction`s (find enough his spendable `TxOut` in the UTXOSet for the AMOUNT it needs). Also, he need to sign the inputs of the transaction. After that, he needs to broadcast all the transactions to all the miners.

1.2 Wallet

The code for `Wallet` is mainly in *wallet.go*. A wallet consists of a pair of (PublicKey, PrivateKey). And we identify each wallet by the hash of its PublicKey.

1.3 TxIn

The code for `TxIn` is mainly in *TxIn.go*. It is just a struct containing the signature and PublicKey (used to verify the legal wallet), the `TxOut` it is using.

1.4 TxOut

The code for `TxOut` is mainly in *TxOut.go*. It contains the value of this output and the target wallet's PublicKeyHash (used to specify to whom).

1.5 Transaction

The code for `Transaction` is mainly in *Transaction.go*. There are basicly two types of transactions. One is called "CoinBaseTransaction", which has no input and is used to generate some initial `TxOut`. Another is the normal transactions that are raised by wallets to spend their spendable outputs. When creating a normal transaction, a wallet need to sign on every input of the transaction using its PublicKey and specify the receiver by putting the receiver's PublicKeyHash in the output.

1.6 Block

The code for `Block` is mainly in *block.go*. For each block, it contains "Header": the hash value of the transactions, "PrevHash": the hash value of the previous block, "ID": the height of this block, "TimeStamp": the time when this block is created, "Nonce": the result of ProofOfWork, "Transactions": the list of all the transactions kept in this block. The block need to make sure that the hash of the contatenate of the hash of this block (all the stuff, including the nounce) and the nounce should have enough number of leading zeros.

1.7 BlockChain

The code for `BlockChain` is mainly in *blockchain.go*. It contains an array of blocks, making sure that they are linked by PrevHash.

1.8 UTXOSet

The code for `UTXOSet` is mainly in *UTXOset.go*. It contains a map with key being the PublicKeyHash, representing each wallet and value being a list of his spendable `TxOut`.

1.9 Miner

The code for `Miner` is mainly in *miner.go*. It contains one `UTXOSet`, one `Blockchain` and several channels used to transmit information. We are making sure that the Blockchain and the UTXOSet it contains are consistent, i.e. all the spendable TxOut in the UTXOSet have not been used in the Blockchain or the set of transactions it is working on, and you can find those TxOut in the Blockchain or the set of transactions it is working on. Below shows our method.

1.9.1 Receive a transaction from the client:

When receiving a transaction from the client, the miner first check that the validation of the transaction (signature, amount ...) and all the inputs are in UTXOSet. Then he interrupt the mining process, update the UTXOSet by the content of this transaction, put this transaction into the block he's mining and restart the mining process. After mined out one block, it will broadcast the block to all miners.

1.9.2 Receive a block from another miner:

When a miner gets a new block and send it to me, I will first check the validation of this block (if invalid, just ignore it) and then check whether the PrevHash of the block is the hash of the tailblock of my blockchain.

If yes, it means that our blockchains are the same (except for the sent block), hence this block is also legal in my blockchain. Then I will undo the update of UTXOSet of the current transactions I am working on, update all the transactions in this block and then reverify the transactions I was working on and update the UTXOSet by the content of those legal transactions (ignore the illegal transactions).

If not, it means that our blockchains differ. We should adopt "Longest Chain Rule". If your block's ID is smaller than or equal to mine, I will just ignore this block. If your ID is greater (your chain is longer), I will send a request for your whole chain. By scaning over the PrevHash in chain, we can find the first different block and then check whether your blocks are valid (PrevHash correct, PoW correst). If your chain is indeed valid, I will undo the update of UTXOSet of my blocks and the transactions I am current working on. Append your blocks to my chain and update the UTXOSet by the contents. Then we will redo all the transactions undone just now (verify whether they are legal according to current UTXOSet and decide whether to work on it) and further keep on mining.

2 Part1: Attack and Defense

There are several kinds of attack we can defend:

- Wallet fake on the amount of Transactions. We will verify the amount of the transaction when the miner get the transaction from the client. If sum(input) < sum(output), it will output "Not Enough Inputs! Fake Client !!!" and ignore the transaction.
- Wallet fake on the signature. Verified when the miner get the transaction from the client, output " Wrong Signature! Fake Client !!!" and ignore the transaction
- Miner fake on ProofOfWork. When a miner get a block from another miner, it will verify its PoW, if fake, raise "Invalid Block" and ignore this block.

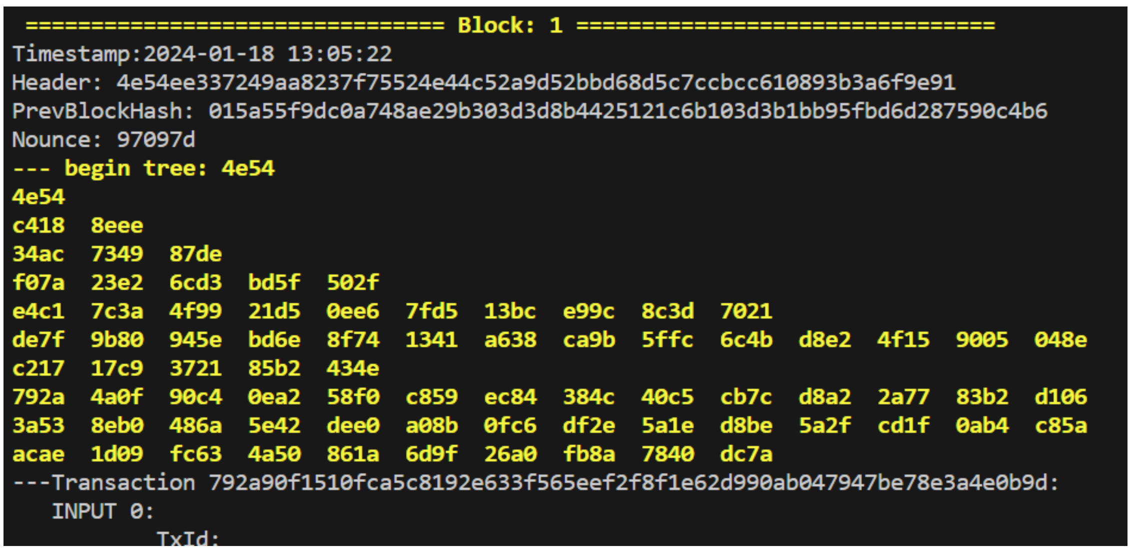
- Miner fake on PrevHash. When a miner get the blockchain from another miner (want to adopt "longest chain rule"), it needs to verify that the chain is valid on the PrevHash pointers. If we find that the blockchain is invalid, we will just take the miner as a lyer and ignore his blockchain and keep working on our own work.

Notice that there is another kind of attack: a miner fake on transactions it have mined out (only change the content of the transaction but the PrevHash, PoW are correct). Up to now, we assume that our miner will only put the transactions permitted by his own UTXOSet onto his blockchain. In this setting, this attack will not be raised so we cannot handle this so far. We may fix this problem in Part2.

3 Part2: Improvements

3.1 Merkel Tree

We change the structure of a block, and set the header of block to be the root of merkel tree built from transactions in it. When the block is printed out in terminal, the tree will also be printed out like below:



```
================================ Block: 1 ================================
Timestamp:2024-01-18 13:05:22
Header: 4e54ee337249aa8237f75524e44c52a9d52bbd68d5c7ccbcc610893b3a6f9e91
PrevBlockHash: 015a55f9dc0a748ae29b303d3d8b4425121c6b103d3b1bb95fbd6d287590c4b6
Nounce: 97097d
--- begin tree: 4e54
4e54
c418  8eee
34ac  7349  87de
f07a  23e2  6cd3  bd5f  502f
e4c1  7c3a  4f99  21d5  0ee6  7fd5  13bc  e99c  8c3d  7021
de7f  9b80  945e  bd6e  8f74  1341  a638  ca9b  5ffc  6c4b  d8e2  4f15  9005  048e
c217  17c9  3721  85b2  434e
792a  4a0f  90c4  0ea2  58f0  c859  ec84  384c  40c5  cb7c  d8a2  2a77  83b2  d106
3a53  8eb0  486a  5e42  dee0  a08b  0fc6  df2e  5a1e  d8be  5a2f  cd1f  0ab4  c85a
acae  1d09  fc63  4a50  861a  6d9f  26a0  fb8a  7840  dc7a
---Transaction 792a90f1510fca5c8192e633f565eef2f8f1e62d990ab047947be78e3a4e0b9d:
   INPUT 0:
           TxId:
```

Figure 1: figure for section 2

3.2 Dynamic List of Miner

We add two grpc protocols to achieve this. First each miner maintains a list of address which it can communicate with. When a miner is added, it needs to know the address of at least one miner. This is achieved in function 'GetAddrList'. Then it will call these existing miners to get their address list and set its own list to be the union of these lists. After this, it will send its own address to the address in its own list to let these miners add its address to their list. This is achieved in function 'UpdateAddrList'.

3.3 Disable the attack of fake transactions

In the Attack and Defense part we mentioned that we assume the miner will not include transactions that can not be done in its own UTXO set.
Now we remove the assumption.
When a miner receives a block, we need to simulate the transactions of incoming block in its own UTXO set. If one of the transaction cannot be done, then we will undo the previous simulated transactions and reject the block. So is for the case when miner gets a chain from other miners, but in this case we will first undo the blocks different from the incoming blockchain, then simulate the different part of incoming chain,

if simulation fails (one transaction cannot be done), then we need to undo the simulated blocks and redo the differnet blocks of my own chain.

3.4 Exponential search for faster blockchain merge

In part1, when there is conflict of blockchain, the miner with shorter chain will fetch the whole longer chain from other miner. This can be very inefficient when the blockchain is very long and two chain differs by a small fraction (say one is of length 10000002 and the other is 10000000). A natural idea is to search for the first different position of two chain and only transfer the different part. Using exponential search in class, this can be done efficiently.
Suppose I am the miner with shorter blockchain and I received a block with ID larger than my length, then I will send an request to the owner of received block to fetch block[my_length - 1], then compare it with my block[my_length - 1]. If they don't match, I will call the owner again and fetch block[my_length - 3], then compare it with my block[my_length - 3] and so on until I the two matches. In the i-th time we fetch block[my_length - 2^{i-1}].
One example:

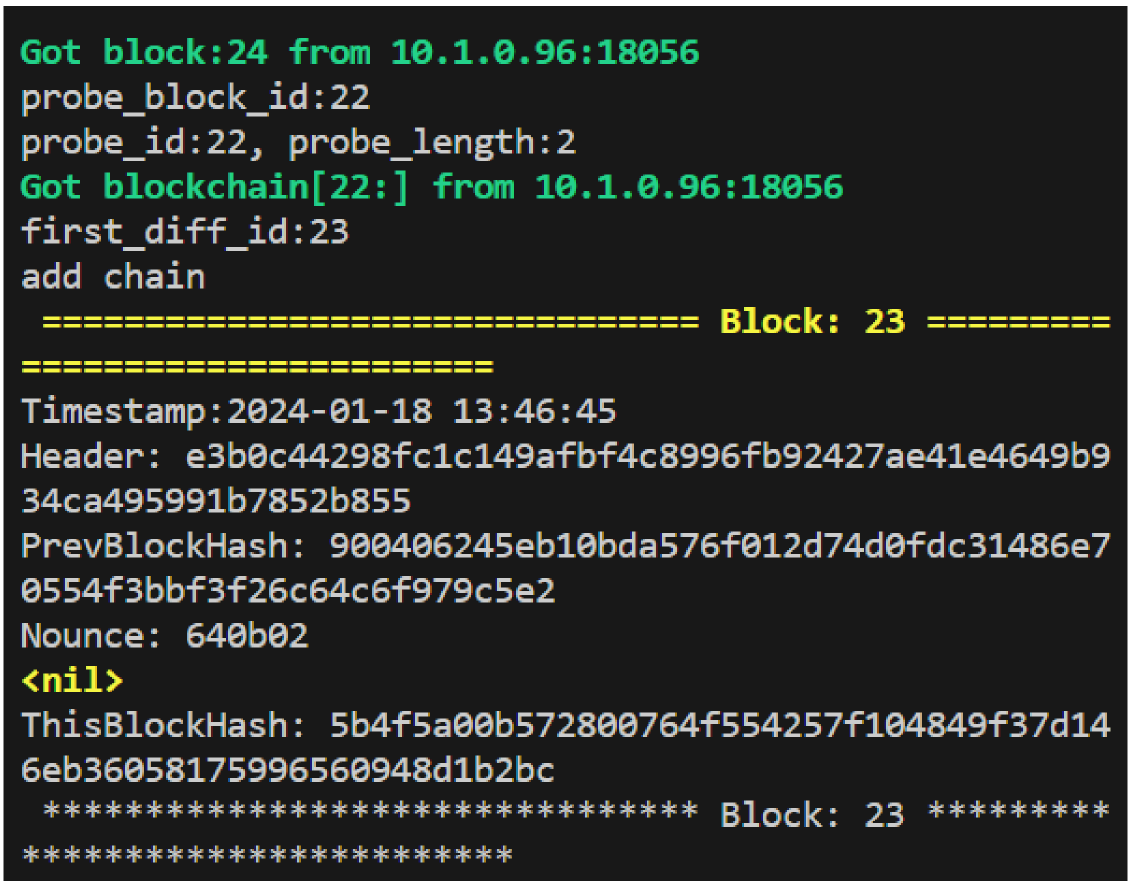


Figure 2: figure for section 2