

E-payment Project Report: Bitcoin

TANG Haikuo, XIU Zixing, LI Shangying, AO Xiaochuan, GUO Yujia, LIU Shuting

October 2023

Contents

1	Blockchain Design	2
2	Mining and PoW	2
2.1	Implementation of PoW	2
2.2	Dynamic Difficulty Design	3
3	Transaction and Verification	4
3.1	Transaction Class Design	4
3.1.1	P2PKH	5
3.1.2	Coinbase Transaction	6
3.1.3	Signature and Verification	7
3.2	UTXO Implementation	7
4	Network	8
4.1	TCP connection and Data Transfer	8
4.2	Get Block	8
4.3	Broadcast	9
4.3.1	Transaction Broadcast	9
4.3.2	Block Broadcast	9
4.4	Block Data Verification	10
5	Storage	10
5.1	Disk Storage	10
5.2	Memory	11
6	Wallet	11
6.1	Key Pair	11
6.2	User's Wallet	12
7	Demonstration	12
7.1	Environment	13
7.2	Command Line Interface	13
7.3	Multi-node simulation	14

1 Blockchain Design

The Blockchain prototype we designed is basically consistent with Bitcoin. We have implemented the basic functions of Bitcoin by implementing the following classes.

Class name	Parameter	Functions
Block	Block information and transactions	Generate block, Encode & Decode a block
Blockchain	Database	Maintain the blockchain
Proof of Work	A block	Calculate and verify the nonce
Transaction	Txid, input and output	Create, sign, verify, encode & decode Tx
Merkle Tree	Merkle node	Build and maintain a merkle tree, verify Tx
UTXO	Blockchain pointer	Browse and locate UTXO
Wallet(s)	Wallet(sk, pk, address)	Generate and maintain key

Figure 1: Class Design

Among them, block contains the following parameters.

Parameter	Type
Timestamp	int64
Transactions	[]*Transaction
PrevBlockHash	[]byte
Hash	[]byte
Nonce	int
Height	int
MerkleRoot	*MerkleTree
Difficulty	int

Figure 2: Block Design

Because when storing, we only store block information. This means that our block class needs to contain all the information required for the blockchain to run. The following is an introduction to the parameters in the block.

- **Timestamp** The creation time of the block.
- **Transaction** The transactions in the block, which can be verified through Merkle Tree.
- **PrevBlockHash** The hash of previous block.
- **Hash** The hash of current block.
- **Nonce** The nonce of current block.
- **Height** The height of current block, begins at 0.
- **MerkleRoot** The Merkle tree of current block, maintained by recording a pointer to the merkle root.
- **Difficulty** The difficulty of current block, that is the number of 0-bits at the beginning of current block hash.

2 Mining and PoW

2.1 Implementation of PoW

We implement the mining and verification of blocks by constructing a PoW class. PoW contains two parameters, namely the pointer of the block to be calculated or verified by PoW, and the target difficulty. PoW class contains the following functions.

- **NewProofOfWork** Initialize PoW.

- **prepareData** Combine all the data that needs hash calculation into a byte string.
- **MineBlock** Find a legal nonce.
- **Validate** Validate the nonce.

```

type ProofOfWork struct { 7个用法
    block *Block
    target *big.Int
}

// NewProofOfWork builds and returns a ProofOfWork
func NewProofOfWork(b *Block) *ProofOfWork {...}

func (pow *ProofOfWork) prepareData(nonce int) []byte {...}

// MineBlock performs a proof-of-work
func (pow *ProofOfWork) MineBlock() (int, []byte) {...}

// Validate validates block's PoW, only validate the nonce, not the transaction
func (pow *ProofOfWork) Validate() bool {...}

```

Figure 3: PoW Design

We use the large integer type provided by goLang to simplify coding. Assuming that the target difficulty is 24, we first calculate a binary large integer A containing 256 ones, and then move A to the right by 24 bits. In this way, A is the largest integer that meets the target difficulty, and the calculated Hash cannot be greater than A.

Algorithm 1 Mine a block

```

Difficulty := Target
A := {1}256
A := A >> Difficulty
BlockHash := Hash(Block)
while BlockHash > A do
    Change(nonce)
    BlockHash := Hash(Block)
end while

```

The verification process is similar. You need to calculate the large integer A first, and then compare A with the Hash of the received block. The Hash function we use is SHA256. Since we doesn't design the transaction fee, the output of the coinbase transaction is actually equal to the reward of mining the block, which is, as we designed, 1000 tokens.

2.2 Dynamic Difficulty Design

We adjust the mining difficulty by examining the mining time of the latest 10 blocks. If the height of the block is less than 10, no adjustment will be made. Otherwise, we first implemented a block iterator, took out the timestamps of the first 10 blocks, calculated the average mining time using the difference-by-difference method, and then compared it with the target time (10 minutes, the same as Bitcoin). Then based on the comparison results, adjustments are made based on the difficulty of the latest block.

This algorithm enables dynamic adjustment of difficulty. As shown in the figure, after miners quickly mined several blocks, the difficulty began to rise.

Algorithm 2 Dynamic Difficulty Compute

```
Block := Lastblock
if Block.Height < 10 then
    return(16)
else
    TargetTime := 600
    Ave :=  $\frac{\sum_{i=6}^{10} T_i - \sum_{i=1}^5 T_i}{5^2}$ 
    if Ave > TargetTime × 1.5 then
        return(Block.Difficulty - 1)
    else if Ave < TargetTime × 0.75 then
        return(Block.Difficulty + 1)
    else
        return(Block.Difficulty)
    end if
end if
```



Figure 4: Dynamic Difficulty Design

3 Transaction and Verification

In this chapter, we describe our design of Transaction in detail. Our design ideas are completely consistent with Bitcoin, but there may be differences from Bitcoin in the design process of P2PKH. Overall, in our system, users use UTXO for transactions. All users do not need to pay transaction fees, so all used coins will be completely converted into UTXO and entered into the payee's account. Miners can only obtain tokens from mining itself, and recording transactions will not bring them any profit. (We understand that this is not feasible in practice, but we did not have sufficient time to design this detail.) The output of a Coinbase transaction is designed to be 1000 tokens.

3.1 Transaction Class Design

The following figure 5 is our specific design of Transaction:

```
type Transaction struct { 39 个用法
    ID    []byte
    Vin   []TXInput
    Vout  []TXOutput
}
type TXInput struct { 9 个用法
    Txid    []byte
    Vout    int
    Signature []byte
    PubKey  []byte
}
type TXOutput struct { 15 个用法
    Value    int
    PubKeyHash []byte
}
```

Figure 5: Transaction Design

Each transaction contains a unique ID, which is the hash value of the transaction, and several inputs and outputs.

Each transaction input contains the corresponding Txid, input amount, address (public key) and signature for verification.

Each transaction output contains the output quantity, and PublicKeyHash.

As shown in figure 6, we have implemented the creation, signing, and verification of Transactions (including all two types of transactions: UTXO transactions and coinbase transactions), and established functional interfaces for the implementation of subsequent wallet functions. In addition, we have also implemented some auxiliary functions, such as calculating transaction hash, transaction serialization and deserialization (for storage and reading), transaction information visualization, etc. The following is the transaction function:

```
// IsCoinbase checks whether the transaction is coinbase
func (tx Transaction) IsCoinbase() bool {...}

// Serialize returns a serialized Transaction
func (tx Transaction) Serialize() []byte {...}

// Hash returns the hash of the Transaction
func (tx *Transaction) Hash() []byte {...}

// Sign signs each input of a Transaction
func (tx *Transaction) Sign(privKey ecdsa.PrivateKey, prevTXs map[string]Transaction) {...}

// String returns a human-readable representation of a transaction
func (tx Transaction) String() string {...}

// TrimmedCopy creates a trimmed copy of Transaction to be used in signing
func (tx *Transaction) TrimmedCopy() Transaction {...}

// Verify verifies signatures of Transaction inputs
func (tx *Transaction) Verify(prevTXs map[string]Transaction) bool {...}

// NewCoinbaseTX creates a new coinbase transaction
func NewCoinbaseTX(to, data string) *Transaction {...}

// NewUTXOTransaction creates a new transaction
func NewUTXOTransaction(wallet *Wallet, to string, amount int, UTXOSet *UTXOSet) *Transaction {...}

// DeserializeTransaction deserializes a transaction
func DeserializeTransaction(data []byte) Transaction {...}
```

Figure 6: Transaction Implementation

3.1.1 P2PKH

We implemented Pay-to-Public-Key-Hash. The following is the P2PKH workflow we designed.

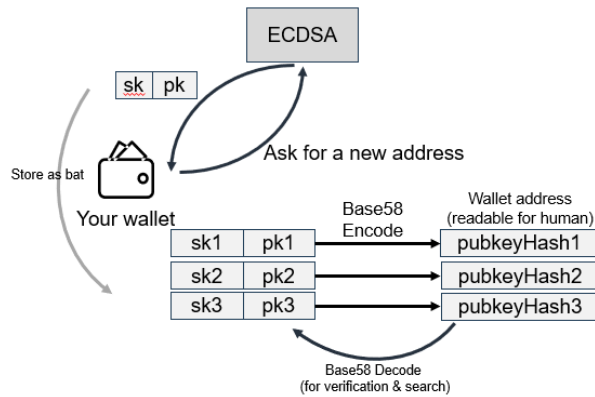


Figure 7: Pubkeyhash Derivation

As shown in figure 7, when a user creates an address through the wallet, the program calls the API of the ECDSA library to generate a key pair, including a public key and a private key, which is saved in the user's local bat file. The program then uses the hash of the private key to generate a Base58 encoding, which is

presented to the user as an address in the wallet. This code will also be saved in the bat file for query and verification.

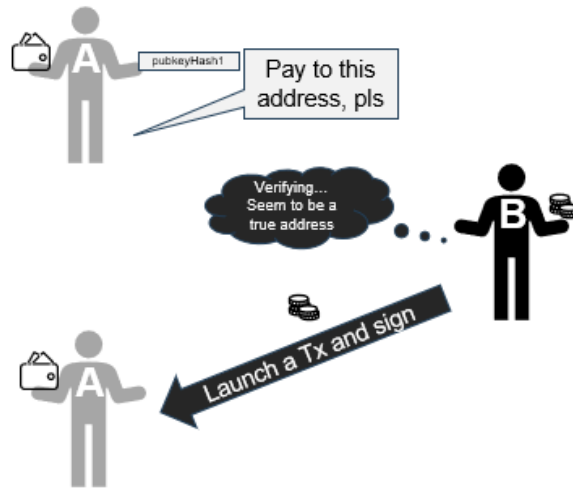


Figure 8: P2PKH Demonstration

As shown in figure 8, when conducting a transaction, the payee first needs to provide its own pkh (pub-keyhash) as the payment address. The payer will then Base58 decode the address to verify the validity of the address. After passing the verification, the payer will initiate a transaction request and take out his private key to sign the transaction, so that the transaction will take effect. After the miner receives and verifies the transaction and mines a block containing the transaction, the payee can confirm the transaction.

3.1.2 Coinbase Transaction

We implement the coinbase transaction using a special create transaction function.

```
// IsCoinbase checks whether the transaction is coinbase
func (tx Transaction) IsCoinbase() bool { 6个用法
    return len(tx.Vin) == 1 && len(tx.Vin[0].Txid) == 0 && tx.Vin[0].Vout == -1
}

// NewCoinbaseTX creates a new coinbase transaction
func NewCoinbaseTX(to, data string) *Transaction { 4个用法
    if data == "" {
        randData := make([]byte, 20)
        _, err := rand.Read(randData)
        if err != nil {
            log.Panic(err)
        }
        data = fmt.Sprintf("#%x{randData}")
    }

    txin := TXInput{Txid: []byte{}, Vout: -1, Signature: nil, PubKey: []byte(data)}
    txout := NewTXOutput(subsidy, to)
    tx := Transaction{ID: nil, Vin: []TXInput{txin}, Vout: []TXOutput{*txout}}
    tx.ID = tx.Hash()

    return &tx
}
```

Figure 9: Coinbase Transaction Implementation

As shown in figure 9, in our design, since there are no transaction fees, the coinbase transaction output is constant and the reward is 1000 tokens. The output address is a wallet address specified by the miner and controlled by itself. When starting the miner node, our CLI will ask the miner to specify an address as

the address to receive the Thousand Years reward. Unlike Bitcoin, we stipulate that the number of coinbase transaction outputs is 1, which means that this reward cannot be distributed to multiple addresses.

3.1.3 Signature and Verification

We have implemented the signature and verification of transactions. The following algorithm is our design idea.

Algorithm 3 Signature

```

if tx.IsCoinbase() then
    return
end if
for vin.iterator := tx.Vin.begin() to tx.Vin.end() do
    if prevTXs[vin.Txid].ID == nil then
        return(ERROR)
    end if
end for
txCopy := tx.Copy()
for InID, vin.iterator := tx.Vin.begin() to tx.Vin.end() do
    prevTx := prevTXs[vin.Txid]
    txCopy.Vin[InID].Signature = nil
    txCopy.Vin[InID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash
    dataToSign := Tostring(txCopy)
    r, s, err := ecdsa.Sign(rand.Reader, &privKey, []byte(dataToSign))
    signature := append(r.Bytes(), s.Bytes()...)
    tx.Vin[InID].Signature = signature
    txCopy.Vin[InID].PubKey = nil
end for
return

```

When the client attempts to conduct a transaction, it will first verify whether all input sources of the transaction are legal. If there are illegal inputs, an error will be thrown, otherwise, continue. The client will copy the input, try to sign all copied inputs with a known private key, and throw an error if an unknown address is found. If all signatures are completed, the transaction is successfully signed and the client broadcasts the transaction.

When the miner node receives the transaction, it will perform a process similar to signing the transaction and verify the signature of the transaction. Only transactions that pass all verifications will be recorded by the miner, otherwise the transaction will be considered illegal and abandoned.

After the miners dig out the block, they will broadcast the new block, and all nodes will receive the new block. Similarly, newly started nodes will also ask other nodes to synchronize the blockchain state for them, thus receiving many blocks. All nodes will verify all transactions in the block when receiving it, and will also verify the nonce after all transactions pass verification. Only blocks that pass both of these two items will be considered legal blocks and entered into the local database.

The specific implementation of the signature algorithm is shown in the figure 10, and the verification algorithm is also implemented. We will not go into details here.

3.2 UTXO Implementation

Our Blockchain, like Bitcoin, uses the UTXO-based model described above. But in the CLI, we did not design a command for users to specify UTXO for transactions. Therefore, in the design of UTXO, we only provide the function of scanning all UTXO. This function traverses the blockchain, finds the UTXO owned by the local wallet, and stores it in memory for subsequent transactions.

UTXO class mainly provides several functions:

- **FindSpendableOutputs** It finds and returns unspent outputs to reference in inputs. Use this function to show all UTXO in the blockchain.

```

// Sign signs each input of a Transaction
func (tx *Transaction) Sign(privKey ecdsa.PrivateKey, prevTXs map[string]Transaction) {
    if tx.IsCoinbase() {
        return
    }

    for _, vin := range tx.Vin {
        if prevTXs[hex.EncodeToString(vin.Txid)].ID == nil {
            log.Panic(fmt.Sprintf("ERROR: Previous transaction is not correct"))
        }
    }

    txCopy := tx.TrimmedCopy()

    for inID, vin := range txCopy.Vin {
        prevTx := prevTXs[hex.EncodeToString(vin.Txid)]
        txCopy.Vin[inID].Signature = nil
        txCopy.Vin[inID].PubKey = prevTx.Vout[vin.Vout].PubKeyHash

        dataToSign := fmt.Sprintf("#%s\n", txCopy)

        r, s, err := ecdsa.Sign(rand.Reader, &privKey, []byte(dataToSign))
        if err != nil {
            log.Panic(err)
        }
        signature := append(r.Bytes(), s.Bytes()...)

        tx.Vin[inID].Signature = signature
        txCopy.Vin[inID].PubKey = nil
    }
}

```

Figure 10: Signature Implementation

- **CountTransaction** It returns the number of transactions in the UTXO set. Use this function to show transaction which includes some UTXOs.
- **FindUTXO** It finds UTXO for a public key hash. Remember that it can be done by everyone who learns the pubkeyhash. We use this function to find UTXOs owned by us so as to spend them.

4 Network

As mentioned earlier, we use TCP for inter-node communication. We use several terminals with different ports to simulate different users.

We first created a node with a fixed port number (we call it the founder node) and used this user to build the blockchain. Remember that in our design, one blockchain corresponds to one database. Manually share this database file (containing only the genesis block) to other users for user initialization. Every new user only knows the founder's port (that's why the founder's port must be fixed). They need to communicate with the founder to learn the existence and addresses of other nodes to build a decentralized network.

After that, we first implemented the underlying TCP connection establishment, and the sending and receiving of byte string data. On this basis, we implemented version query, block transmission, and transaction transmission between nodes. After realizing the transmission of these data types, we also implemented the broadcast and verification of blocks and transactions.

4.1 TCP connection and Data Transfer

As mentioned above, we first implement the basic TCP connection establishment function and byte string data sending and receiving function.

After the node starts, a TCP connection needs to be established first. Subsequently, the receiver needs to send the cmd field to make a request to the data sender. The cmd type is as shown below.

4.2 Get Block

We implemented the getblock function.


```

// UTXOSet represents UTXO set
type UTXOSet struct { 13个用法
    Blockchain *Blockchain
}

// FindSpendableOutputs finds and returns unspent outputs to reference in inputs
func (u UTXOSet) FindSpendableOutputs(pubkeyHash []byte, amount int) (int, map[string][]int) {...}

// FindUTXO finds UTXO for a public key hash
func (u UTXOSet) FindUTXO(pubKeyHash []byte) []TXOutput {...}

// CountTransactions returns the number of transactions in the UTXO set
func (u UTXOSet) CountTransactions() int {...}

// Reindex rebuilds the UTXO set
func (u UTXOSet) Reindex() {...}

// Update updates the UTXO set with transactions from the Block
// The Block is considered to be the tip of a blockchain
func (u UTXOSet) Update(block *Block) {...}

```

Figure 11: UTXO Design

After the node starts, the node will request their version from other nodes, where version refers to the height of the blockchain. When it is found that the blockchain height of a node is higher than its own local chain, the node will issue a synchronization request and use the getblock function to ask other nodes to send blocks to it. The process is as follows.

- **NODE A** Send "Get Version"
- **NODE B** Send Version
- **NODE A** Handle Version, find that local chain is outdated.
- **NODE A** Send "Get Blocks"
- **NODE B** (Serialize Blocks)
- **NODE B** Send Serialized Blocks
- **NODE A** Handle Serialized Blocks
- **NODE A** (Deserialize and verify blocks)

4.3 Broadcast

4.3.1 Transaction Broadcast

When a new block is generated, the running miner node will automatically broadcast the block. The miner node sends the serialized block to every known node. Each receiving node will verify the block, and if the verification passes, it will be recorded locally. We included this in a later test video.

4.3.2 Block Broadcast

When a new transaction is generated, the node will automatically broadcast the transaction. The client node sends the serialized transaction to each known node. Each receiving miner node will verify and record this transaction. After the number of recorded transactions reaches a threshold (the threshold we set is 2), the miner node will try to mine a block, and this block will contain all these transactions. We included this in a later test video.

```

switch command {
case "addr":
    handleAddr(request)
case "block":
    handleBlock(request, bc)
case "inv":
    handleInv(request, bc)
case "getblocks":
    handleGetBlocks(request, bc)
case "getdata":
    handleGetData(request, bc)
case "tx":
    handleTx(request, bc)
case "version":
    handleVersion(request, bc)
default:
    fmt.Println(a...: "Unknown command!")
}

```

Figure 12: TCP CMD list

4.4 Block Data Verification

As mentioned before, we have implemented block verification. Each node will verify all received blocks, and if the verification passes, it will be recorded locally. Verification consists of three parts:

- **Verify every transaction** Nodes will verify the signatures of these transactions and the legality of inputs and outputs. The output token should be less than or equal to the input one.
- **Verify Merkle Tree** Recalculate the Merkle Tree and compare the locally calculated Merkle Root with the Merkle Root in the received block.
- **Verify Nonce** Create a PoW class and verify whether the nonce matches the difficulty of the block.

5 Storage

5.1 Disk Storage

The blockchain we implemented uses BoltDB that comes with go for data persistence.

BoltDB is an embedded, pure Go language key-value store database. It is designed as a high-performance, reliable, and easy-to-use database engine suitable for various applications and use cases. We use BlockHash as the key, the serialized Block as the Value, and save it to BoltDB. No other storage is required beyond this.

Here are some key features and advantages of BoltDB:

- **Embedded database:** BoltDB is an embedded database to Golang, which means it can be directly embedded within an application without the need for a separate database server. What's more, the version maintaining is not a problem to our project.
- **High performance:** BoltDB is designed to have excellent performance. It uses a technique called MVCC (Multi-Version Concurrency Control) that allows multiple read operations to execute concurrently while maintaining consistency for write operations. This makes BoltDB perform well under high-concurrency loads and provides low-latency characteristics.
- **Simple and easy to use:** We do need a simple sample DB for our project. BoltDB has a simple and intuitive API that is easy to use and understand. It provides basic key-value store operations such as insert, update, delete, and query. Additionally, BoltDB supports the concept of buckets, which allows data to be organized in a hierarchical structure for better organization and management.

wallet_1000.dat	12/1/2023 10:48 AM	DAT File	2 KB
wallet_1001.dat	12/1/2023 10:49 AM	DAT File	2 KB
wallet_1002.dat	12/1/2023 10:49 AM	DAT File	2 KB
blockchain_1000.db	12/1/2023 12:04 PM	Data Base File	64 KB
blockchain_1000.db.lock	12/1/2023 11:27 AM	LOCK File	0 KB
blockchain_1001.db	12/1/2023 12:04 PM	Data Base File	64 KB
blockchain_1001.db.lock	12/1/2023 12:02 PM	LOCK File	0 KB
blockchain_1002.db	12/1/2023 12:07 PM	Data Base File	64 KB
blockchain_init.db	12/1/2023 11:17 AM	Data Base File	32 KB

Figure 13: Disk Storage

In addition, in addition to storing the blockchain, users also need to properly keep their public and private keys. We use the client to create an encrypted bat file for each user to ensure security. All persistent storage files are shown in the figure.

5.2 Memory

We also need to use Memory to store some temporary data to speed up calculations.

After the miner node is started, the verified transaction information will be temporarily stored in memory.

6 Wallet

Regarding the wallet, we designed two classes.

Class "wallet" represents a (sk, pk) key pair, and its corresponding concept is a wallet address in bitcoin, which has functions such as generating, verifying, and exporting pubkeyhash.

Class "wallets" contains several "wallets", and its corresponding concept is a real user's wallet, which provides the functions of importing files, exporting from files, browsing all wallet addresses, and interacting with each wallet address. In our design, the program saves the user's wallet address in units of "wallets". We have created a "wallets" for each user, and users can add, manage, and use the wallet addresses in this wallet. The program will save all wallet addresses of each user in the local "wallet_username.dat" file for permanent storage.

6.1 Key Pair

As mentioned before, we use the ECDSA cryptosystem to generate a key pair as the user's wallet address.

The wallet address we designed is as follows:

```
// Wallet stores private and public keys
type Wallet struct { 11个用法
    PrivateKey ecdsa.PrivateKey
    PublicKey []byte
}

// NewWallet creates and returns a Wallet
func NewWallet() *Wallet {...}

// GetAddress returns wallet address
func (w Wallet) GetAddress() []byte {...}

// HashPubKey hashes public key
func HashPubKey(pubKey []byte) []byte {...}

// ValidateAddress check if address if valid
func ValidateAddress(address string) bool {...}

// Checksum generates a checksum for a public key
func checksum(payload []byte) []byte {...}

func newKeyPair() (ecdsa.PrivateKey, []byte) {...}
```

Figure 14: "wallet" Key Pair Implementation

As shown in the figure 14, the program generates pubkeyhash through the ECDSA key pair. We can use the pubkeyhash string as an index to find the user's ECDSA key pair.

6.2 User's Wallet

The wallet class we designed contains a mapping from pubkeyhash to key pairs. Its implementation is shown in the figure:

```
// Wallets stores a collection of wallets
type Wallets struct { 11个用法
    Wallets map[string]*Wallet
}

// NewWallets creates Wallets and fills it from a file if it exists
func NewWallets(nodeID string) (*Wallets, error) {...}

// CreateWallet adds a Wallet to Wallets
func (ws *Wallets) CreateWallet() string {...}

// GetAddresses returns an array of addresses stored in the wallet file
func (ws *Wallets) GetAddresses() []string {...}

// GetWallet returns a Wallet by its address
func (ws *Wallets) GetWallet(address string) *Wallet {...}

// LoadFromFile loads wallets from the file
func (ws *Wallets) LoadFromFile(nodeID string) error {...}

// SaveToFile saves wallets to a file
func (ws *Wallets) SaveToFile(nodeID string) {...}
```

Figure 15: "wallets" Implementation

We can test these functions through our command line interface.

First, we create some wallet addresses for user with $ID = 1000$. See figure 16. We name the user by its export node ID.

```
D@DESKTOP-2IBJMTB MINGW64 /d/G01
$ ./G01 createwallet
Your new address: 1HLFWiZL8PR5Rb6NEWhiH7x3JBnT9WiEFQ

D@DESKTOP-2IBJMTB MINGW64 /d/G01
$ ./G01 createwallet
Your new address: 12C1WgP6E8xvWFmyNtSE8CTKkov8tonrXo

D@DESKTOP-2IBJMTB MINGW64 /d/G01
$
```

Figure 16: Create Wallet Address

After creating a Genesis Block using the first wallet address, the address will automatically receive the Genesis Block's coinbase reward. Remember that the coinbase reward we designed is 1000 tokens. We can use wallet to query the balance of the wallet address. As shown in the picture, the balance of this address is 1000 tokens.

We can use this address to transfer money to other people. As shown in the picture 17, we transfer 50 tokens to the second wallet address we just created.

The input of this transaction is the previous coinbase transaction, which is legal, so the input verification passes. Since the current user is the owner and creator of the wallet, he knows the private keys of all addresses, so the signing proceeds normally. In this way, the transaction is effective.

We can also set the wallet address as the address to receive mining rewards. As shown in the figure 18, when the miner node starts, it needs to provide a wallet address to receive rewards.

We can view all wallet addresses of the current user. This command reads the wallet.dat file under the username and finds the addresses that can be used. As the figure 19 shows.

7 Demonstration

After completing the project compilation, we launch multiple terminals to simulate different users. In fact, since we use TCP to transmit data, these users can be distributed in different machines. But this may require

```

Value: 1000
Script: b32663d23aab9150c90375f5dae67cb2d24f1b78

D@DESKTOP-2IBJMT MINGW64 /d/G01
$ ./G01 getbalance -address 1HLFWiZL8PR5Rb6NEWhiH7x3JBnT9WiEFQ
Balance of '1HLFWiZL8PR5Rb6NEWhiH7x3JBnT9WiEFQ': 1000

D@DESKTOP-2IBJMT MINGW64 /d/G01
$ ./G01 send -from 1HLFWiZL8PR5Rb6NEWhiH7x3JBnT9WiEFQ -to 12C1WgP6E8xvWFmyNtSE8CTKkou8to
nrXo -amount 50 -mine
a1c7bc113910139941f0e9789e7f068c1f6db84b036366feb69e9091c4cabf0

Success!

```

Figure 17: Check Balance and Transaction

```

D@DESKTOP-2IBJMT MINGW64 /d/G01
$ ./G01 startnode -miner 1LEPuj pgxeaMzc35uJLpN9sAH3mCRKoupc
Starting node 1001
Mining is on. Address to receive rewards: 1LEPuj pgxeaMzc35uJLpN9sAH3mCRKoupc

```

Figure 18: Miner Address for Reward

us to start our initial node on a machine node with a public IP. Therefore this is currently not possible.

In any case, we have implemented data transmission using TCP and communication between different types of nodes. The implementation of these functions ensures the reliability of this blockchain. Our blockchain is available, trusted, synchronized, and distributed. Additionally, outside of the blockchain database, each user also has an encrypted dat file that holds their own keys.

7.1 Environment

Our program is written in go language, and the go version is 1.18.19.

The IDEA we use is golang 2023.2.3.

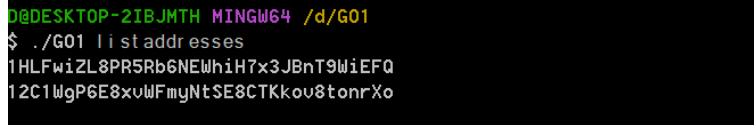
Use the ECDSA library that comes with go to implement key pair related work. Package ecdsa implements the Elliptic Curve Digital Signature Algorithm, as defined in FIPS 186-4 and SEC 1, Version 2.0.

7.2 Command Line Interface

In the CLI, we have implemented the following commands.

- **createwallet** Generates a new key-pair and saves it into the wallet file
- **createblockchain** Create a blockchain and send genesis bloc reward to ADDRESS
- **send** Launch a transaction
- **getbalance** Get balance of address
- **printchain** printlocal blockchain
- **startnode** Start a node with ID specified in NODEID. Var -miner enables mining, a miner's address is needed to receive reward
- **reindexutxo** Rebuilds the UTXO set
- **listaddresses** Lists all addresses from the wallet file

In subsequent demonstrations, we will demonstrate the usage of these commands.



```

D@DESKTOP-2IBJMT MINGW64 /d/G01
$ ./G01 listaddresses
1HLFWiZL8PR5Rb6NEWhiH7x3JBnT9WiEFQ
12C1WgP6E8xvWfmyNtSE8CTKkou8tonrXo

```

Figure 19: Browse Address

7.3 Multi-node simulation

We recorded a video containing almost all functional tests and attached it to the submitted zip package. You can also view this video via the link in the attachment [1].

As mentioned before, we opened three terminals and defined three different ports, namely:

- **Export NODEID=1000** use this port to simulate the creation node.
- **Export NODEID=1001** use this port to simulate the miner node.
- **Export NODEID=1002** use this port to simulate a lightweight client node.

Here I'll talk about our pilot process. You may want to read against the video to understand our testing process.

Before starting to run the blockchain, you need to test the wallet's address generation function. We create a wallet and two wallet addresses for each user.

First, test the prototype of the block. We use the founder node to create a blockchain with a genesis block and receive the coinbase reward of the genesis block. Manually transfer the database corresponding to the blockchain with only the genesis block to others. In Bitcoin, this step corresponds to Bitcoin users downloading a client from the official website. The blockchain inside only contains the genesis block, and the customer needs to connect to the Internet to obtain more blocks. At the beginning, clients only know one founder node. He will ask the founder node for the existence of other nodes in order to build a decentralized network. After receiving the address information of other full nodes, the new client will try to communicate with them to synchronize the blockchain state. Of course, the client needs to verify each received block.

Next, we use the founder node to initiate transfers for each user and mine some blocks to record these transactions. This step is to facilitate subsequent experiments. At this point, we can already view the blockchain status on the founder node. However, due to the failure to synchronize the latest transfer information, the miner node did not know that the founder transferred money to him. Therefore, the miner's balance displayed on the miner node is 0.

Next, we need to start the miner node and client node. It can be found that after starting the node, the new nodes immediately sent a query version request to the only founder node they knew. The version here refers to the height of the blockchain. If the other party's version number is higher than ours, we will send a synchronization request to the other party. After the other party receives and agrees to the synchronization request, blocks will be sent to us. We receive and verify these blocks and record them in the local database. It can be found that at this time, the miner received and verified the block containing the transaction "The founder transferred 50 tokens to the miner", so the miner's balance was updated.

At this time, a decentralized Bitcoin network has taken shape. We want to test the functionality of the client. In reality, lightweight clients are usually only responsible for initiating transactions and not mining. We test this. We used 1002 users, which are client users, to initiate two transactions. It can be seen that after the transaction is sent, the miner node receives the broadcast about the transaction, and after recording 2 transactions (we set the miner node to start trying the mine block after receiving 2 or more transactions). Soon, the miner mined a block and broadcast it. The founder node received the broadcast of this block and verified that it was recorded.

The client came online again and requested the latest block from all nodes, and the founder node responded. After receiving the latest block recording its transactions, the client user's balance is updated. The functional test of the client passed.

There are also some small features shown at the end of the video, which we won't go into details here.

References

- [1] HAIKUO, T. Build a bitcoin blockchain, 2023. https://www.bilibili.com/video/BV1Cg4y1f7A6/?spm_id_from=333.999.0.0vd_source=3fbf4ee4a6b73aaa7103702a680186a2.