Operating Systems and Distributed Systems 2023 Project Report 2-1

Ruiying Ma, Zhengyuan Su, Haofeng Huang

December 31, 2023

Section 1.

Assumption

We assume that we have a perfect network. There is no netowrk failure and all messages will be received eventually.

Section 2.

Design

We implement a decentralized blockchain network with 5 miner machines and 5 client processes.

Wallet

```
type Wallet struct {
  SK []byte // secret key
  PK []byte // public key
  Address     []byte // address
}
```

A wallet is equivalent to a user in the blockchain network. A real-world person can create multiple wallets. A wallet has a *public key* and a *secret key*. With the public key, the *address* of this wallet can be computed using a specific hash function. Once a wallet is created, the public key and the address of this wallet are broadcast to the network. A created wallet is stored offline.

Transaction

```
type In struct {
  HashTx      []byte  // this income is gained from this tx
  Idx int // this income is gained from this payment
  Amount      int   // the amount of this income
}

type Out struct {
  Amount      int  // the amount of this payment
  Recipient   []byte // the address of the recipient
}

type Transaction struct {
  Initiator   []byte  // the public key of the initiator of this transaction
  Incomes             []In  // the incomes this transaction contains
  Payments    []Out  // the payments this transaction contrains
  IsReward    bool  // whether this transaction is a reward of block mining
  Hash                []byte  // the hash of this transaction
  Signature   []byte  // the signature of this transaction by the initiator
}
```

A transaction is first signed and then hashed. The hash serves as an abstract of this transaction. When we want to identify this transaction, we only need to use its hash. Hashing is performed after signing so that different transactions (including the reward transactions) have different hashes.

When verifying a transaction, we check

- Whether the transaction's incomes are valid (unspent, belongs to initiator). No need to check this for reward transaction.
- Whether the transaction's payments are valid. That is, payments $\leq$ incomes.
- Whether the transaction's signature is valid.
- Whether the transaction's hash is valid.

Block

```
type Block struct {
  Txs []*Transaction  // set of transactions this block contains
  TxHashes    []byte  // the hash of all the transactions
  PrevHash    []byte  // the hash of the previous block
  Time        int64  // the time when this block is created
  Nonce       int
  Height      int  // the position of this block in the blockchain (count from 0)
  IsGenisis   bool  // whether this block is the genisis block (the first block of the blockchain)
  Hash        []byte  // the hash of this block
}
```

When verifying a block, we check

- Whether the block's `TxHashes` is correct.
- Whether there is at most one reward transaction in this block.
- Whether the block's previous hash is correct. No need to check this for the genesis block.
- Whether the block's height is correct.
- Whether the block's transactions are all legitimate.
- Whether the block's nonce is legitimate.
- Whether the block's hash is correct.

Notice that when a miner is going to make a block from a set of transactions, it should ensure that a transaction initiator can only occur once in this block. This aims to defend double-spent attack.

Blockchain

```
type BlockChain struct {
  DB  *bolt.DB  // the offline database that stores the blockchain
}
```

The blockchain is stored in the offline database of each machine. We implement a 1-confirmation blockchain (i.e., when branch occurs, if a branch is longer by 1 block, then this branch is chosen).

When appending a block `B` to the blockchain, we

- Check whether this block is legitimate.
- If the block is legitimate, we update the blockchain using Nakamoto's rules. Suppose the highest block in the blockchain is `B'`. Whether `B` will be appended to the blockchain is determined by the following process:
  - If `B.Height` $<$ `B'.Height`, then `B` won't be appended to the blockchain.
  - If `B.Height` $=$ `B'.Height`, then `B` will be appended to the blockchain if `B` is created before `B'`, that is, `B.Time` $<$ `B'.Time`.

Network Layout

We implement a decentralized network. The network consists of 5 miners. Each miner can

- broadcast a transaction (can consider this functionality as a client process)
- mine and broadcast a block
- receive and handle a transaction
- receive and handle a block

```
type Miner struct {
  BC        *blockchain.BlockChain  // the blockchain of this miner
  MID       string  // the machine id of this miner
  Mempool   map[string]blockchain.Transaction  // the memory pool that stores transactions received by this miner
  Addrs     map[string][]string  // the addresses of all the peers in the network
  bc_lock   chan bool  // the lock of the blockchain
  mem_lock  chan bool  // the lock of the memory pool
  addr_lock chan bool  // the lock of the address map
}
```

At first all miners create their wallets and broacast the public keys and addresses. Each miner stores the addresses and public keys of other miners.

Then, a designated miner will create the genisis block and get the reward. This miner then distributes this reward evenly to all other miners to ensure that all miners have some money before they being to generate transactions.

Now, all miners begin to generate transactions, mine blocks, and handle messages. When a miner receives a transaction, it adds this transaction to its memory pool. If there are sufficiently many transactions in the memory pool, the miner begins to generate a block. Once the block is generated, the miner will broadcast it to all peers. When a miner receives a block, it will check the validity of this block and append the legitimate block to its blockchain.

When handling messages, each miner should make sure that there is no concurrency issue when accessing `Addrs`, `Mempool` and its blockchain `BC`.

Section 3.

Evaluations and Results

We evaluate the disitribution of the time to mine a block. We also evaluate the distribution of the time to verify the validity of a block.

- Time to mine a block

  We can see that the distribution is consistent with the one we've seen in class.

- Time to verify a block

Section 4.

Improvement Proposal

We may implement the following features/improvements later:

- A Merkle tree for transactions in the same block
- A UTXO set to speedup block/transaction verifaction
- k-Confirmation with k greater than one
- Dynamically adjust the difficulty of POW
- Allow a user to create a wallet at any time and use it in the future (i.e., a p2p network such that a user can join/leave dynamically)

Section 5.

References

We referred to a centralized blockchain implementation and the Bolt DB documents.