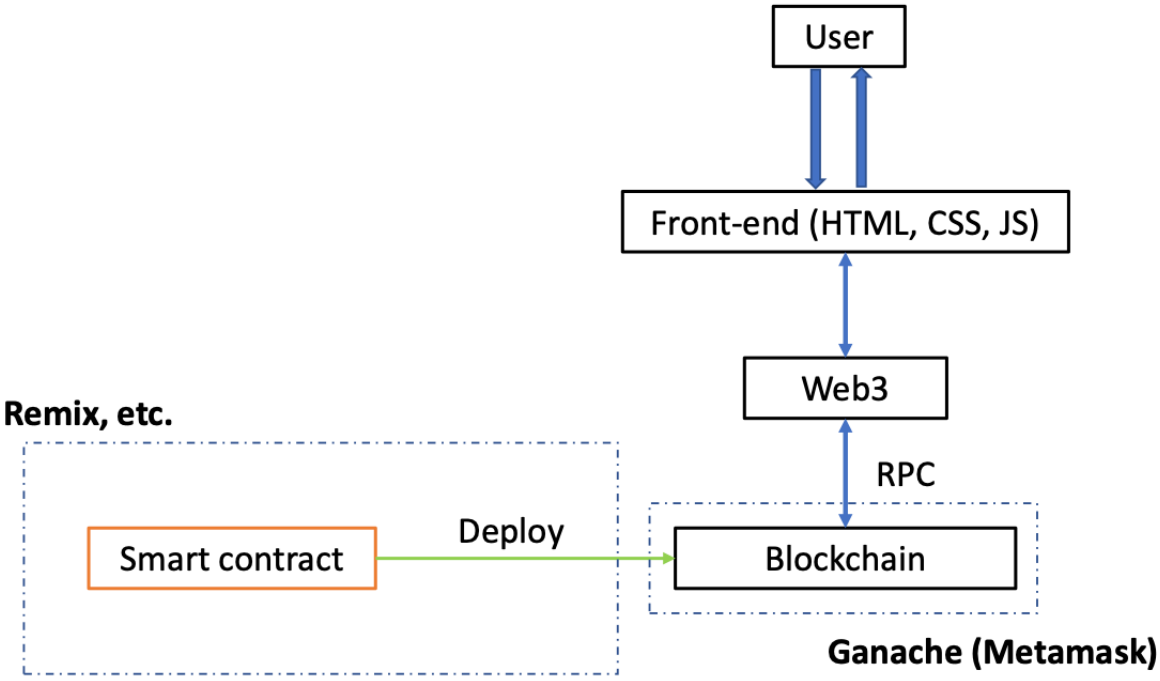
# Design Document of Assignment 2

## Machine Problem: Distributed Coin Flipping Game Using Ethereum

Jiabao LIN, 3035673521

### Requirements Analysis

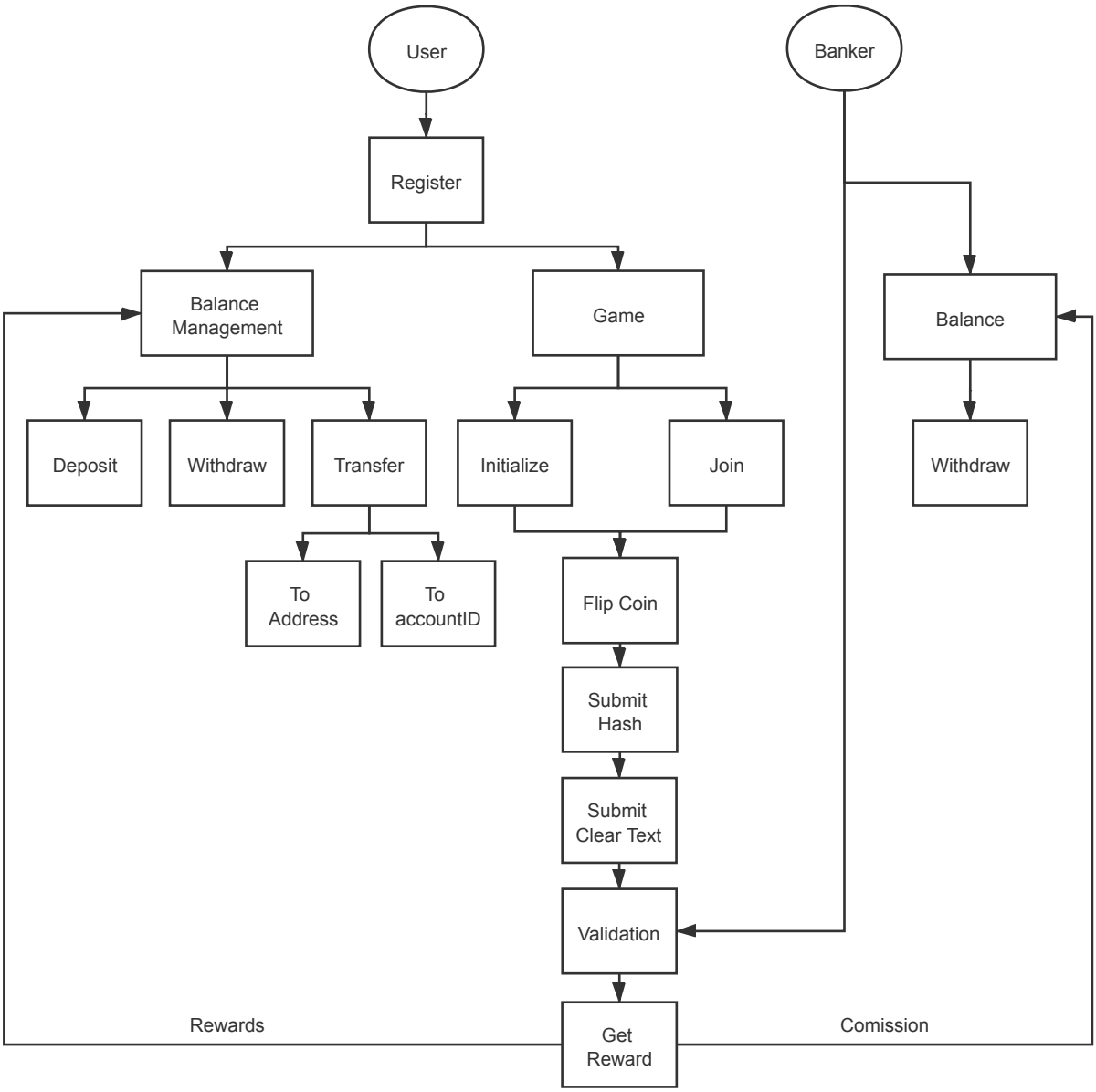
As mentioned in the requirements, we need to design a platform that people could register as a user, deposit/withdraw ETH to/from this platform, do in-game transfer to other registered users, and play coin flipping game with each other. In the meantime, a banker is required to perform as a dealer, which will prove the players are not cheating and get commission for this service.




As a DApp, the backend should be the pure Ethereum Network, no database and backend engine is needed, and I need to write a contract to satisfy all the requirements. The user connects to the Ethereum Network to interact with the contract through web3. As long as the computer is conntected to the Internet and accessible to the Ethereum Network, the user could play this game. No backend server is needed, and I can even upload and host the static site on some third-party service like GitHub Pages.

### Design

In my consideration, I will provide 4 functions, namely, Resigtration, Balance Management, Dealer (Banker) and Coin Flipping Game.




## User Registration

User opens the website. If the user is not registered yet, the site will not showing other contents, and it should register with a proper `accountID` first.

For each registered user, it will have a `User` struct maintained in the contract.

```
struct User {
    address bindAddress;
    string accountID;
    uint balance;
    uint lastGameID;
    uint[] transactionRecord;
}
```

`accountID` should be identical to each other, and should not start with `0x`, as it is reserved for addresses.

- Because users can transfer their balance to other users with the corresponding address/accountID, the website need to tell whether the input is an address or an accountID, so the accountIDs are limited to not start with `0x` to distinguish with addresses.

## Balance Management

### Introduction

For each user, it owns an Ethereum account, which is its private wallet. For this platform, it initialize an `User` struct for each user and maintaining their balance. Therefore, user does not need to do an transaction every time when it would like to initialize/join a game. As the `accountID` and `address` are all identical to each other, only the user itself can manage its balance.

Once the user has registered successfully, it could perform the Balance Management. In my consideration, Balance Management should cover the following functions.

### Deposit

User can deposit its Ether (ETH) into the contract, and the contract will modify the corresponding `balance` for the user.

Once the user deposit an amount of ETH into the contract through `deposit()` function, the `balance` will increase by the corresponding amount in wei.

### Withdraw

User can withdraw its Ether (ETH) from the contract, and the contract will modify the corresponding `balance` for the user.

Once the user withdraw an amount of ETH from the contract through `withdraw()` function, the `balance` will decrease by the corresponding amount in wei.

### Transfer

User can transfer its Ether (ETH) within the contract to another registered user by its address or accountID.

Once the user transfer an amount of ETH to another user, the `balance` of the user will decrease by this amount and the `balance` of the target will increase by this amount.

### Transaction Histories

All the balance change will be record by a `Transaction` struct.

```
struct Transaction {
    uint _transactionID;
    uint _time;
    string _from;
    string _to;
    uint _amount;
    string _type;
}
```

Once the user deposit, withdraw, transfer, receive transaction, bet in a game, win the game, a `Transaction` instance will be initialized and the `transactionID` will be appended to user's `transactionRecord`. Therefore, the user can check all the transaction records of him.

As required, the user can see its transaction records within 1 day, so a `require()` is appended on the `transactionCheck()` function to check the time. Also, a criterion is implemented in the JavaScript to prevent the user request the outdated transaction records.

## Dealer (Banker)

As mentioned by the requirement, the dealer should be a third-party banker. However, if a third-party banker is a real person, it will suffer the following problems.

1. It might not notify the players to reveal the numbers in time,
2. It might not pick out the winner in time,
3. It might make some mistakes when calculating the `mod` and pick the winner.

In the meantime, as the TA have answered on Moodle, I make implement the banker in the contract, and the person who deploy this contract can claim the banker's balance (commission earnings) because its address will be bind to banker, which is saved in `_bindAddress`.

Therefore, I implement a `Banker` struct in my contract.

```
struct Banker {
    address payable _bindAddress;
    uint balance;
    uint gameDeposit;
}
```

Once the contract is deployed, the `constructor()` will initialize the `banker` instance automatically.

```
Banker internal banker;

constructor() public {
    banker = Banker({
        _bindAddress: msg.sender,
        balance: 0,
        gameDeposit: 0
    });
}
```

A qualified banker should have the following functions.

**Collect Hash Values and Clear Text**

In each game, the participants will submit the hash values and random numbers in clear text to the contract.

The hash values are stored in a temporary parameter `submittedHashValue`, which will be cleaned after each game. The clear text are stored in permanent parameter `submittedClearText`, which is a parameter of each `Game` struct (see Coin Flipping Game).

**Validate Hash Values and Clear Text**

Once all the hash values and clear texts are received, the banker (contract) will start validating the values. (see Round 4 in Coin Flipping Game).

**Find out the Winner**

If there is no cheater detected, the banker (contract) will pick out the winner automatically (see Round 4 in Coin Flipping Game).

The banker will transfer 95% of the game deposit to the winner, and transfer 5% of the game deposit to the banker's own balance as the commission fee. After that, the banker will reset the `gameDeposit`, waiting for the next round of the game.

**Withdraw Commission**

As the banker keeps earning commissions in the contract if users play games on the platform, it will have some deposit in the `balance`. The person who deploy this contract will be able to withdraw this commission earnings by calling `withdrawBanker()` function, and the required amount of ETH will be transferred to the person's Ethereum account from the contract.

## Coin Flipping Game

For each game initiated, a `Game` struct will be initialized in the contract and stored permanently.

```
struct Game {
    uint ID;
    uint status;
    address[2] player;
    uint betValue;
    address winner;
    mapping (address => uint) submittedClearText;
}
```

To take this game apart, we can easily find out there are several rounds for the game.

**Round 1**

A user initialize a game on the platform, set the bet value. Other users will see the game on the platform, if other users think the bet value is proper, they can click `Join Game` button to join the game.

Once the user participates in the game, the bet value will be deducted and transferred to the banker's game deposit to make sure the player have enough balance and will not withdraw the money before the game ends.

**Round 2**

After the table is full, each user can click the `Flip the Coin` button to generate a random number on its machine (browser), the random number is a private variable in a function, so the user cannot see the value from the console.

The random number is concatenated with the users address (see Answer the Question 2.), which will be look like $Address||Random\ Number$, and fed into a SHA3-256 function. After that, the hash value is sent to the banker, and the banker will keep the hash value for validation.

**Round 3**

Once the banker receive hash values submitted by all the users, a parameter `bothSubmitHash` will be changed from `false` to `true`. In the frontend, the JavaScript keeps checking this value (once per second). Once the JavaScript find this value has been changed to `true`, the function will automatically send the random number to the banker (reveal the clear text).

The participants do not need to do anything else, all they need to do is to confirm the transaction pop-up in MetaMask, which is the transaction of sending the random number to banker.

**Round 4**

Once the banker receive all the clear text, it will start the `validation()` function, which is to check whether all the users faithful in the process. If all the participants are loyal, the banker will get the sum value of the clear text, calculate the remainder with the `addmod()` function provided by Solidity. This remainder will be the index of the participants in the `player` list.

If at least 1 user is detected cheating, the game will end automatically and the deposit will be refund to the participants account. I do not implement a punishment here because of the following reasons:

- The random number is a private value in the `flipCoin()` funtion in JavaScript, a normal user cannot read this value in the browser. If the attacker would like to modify the value, it need to capture the browser's traffic and manipulate the hexadecimal transaction data sent by MetaMask.
- As the attacker can capture the traffic and modify the transaction data on its local machine, it can also capture the traffic of other participants. If an attacker capture other participants package and change the transaction data, and leads to a loss for those participants, that will be quite unfair for them.

**Round 5: House Clearning**

After each game, the contract will clear up the unnecessary parameters to save gas, and to change the game status, increase the `gameID` by $1$.

# Answer the Question

Please explain/prove whether this protocol actually works or not, in term of fairness, and what are the possible attacks (cheating) among the three parties. In your implementation, how have you solved them?

For the given protocol, overall speaking, this outline will work, but still have some problems.

1. Random number generation. For most of the cases, the random number generator just provide pseudo-random.

   **Solution**

   - As I generate the random number in the website, I do not use my own random number generator, but use the `Math.random()` provided by the browser. As I found here, this random number is not generated by the JavaScript but generated by the browser with **xorshift128+** algorithm. This algorithm takes a long time to repeat, and the game only last for very short period of time, so the random number generation will be secure enough.
   - Make sure the possibility of odd/even number is the same. Because the `Math.random()` generate the random number within $[0, 1]$, we need to scale it up to make the most use of it. However, even if its decimal precision is to the 18th decimal places, it is of high frequency that the last character is 0, so the occurance of odd/even number is not of same possibility. Therefore, I only scale it up to $10^{16}$ to make sure the odd/even number occurs with the same possibility.
2. Potentially reversible hash function. Basically speaking, the hash funtion is not reversible. However, as mentioned by D Kaovratovich [1], the SHA-2 family is still acdamically crackable. If the input is not long enough (the random number is not large enough), the attack might brute-force or use collision method to crack the clear text. For example, on this site, it maintains a dictionary of the cracked hash values and corresponding clear text.

   **Solution**

   - Add salt for each submit value. Because JavaScript cannot hold interger larger than $9007199254740992 = 2^{53}$, which is only 53 bits and shorter than the output of `keccak256()` function (SHA3-256 with 256 bits output). Therefore, I used the user's address, which is 20 bytes/160 bits long, as the salt. The concatenation of the address

and the random number will be look like $Address||Random\ Number$, which is $160 + 53 = 213$ bits long, and is difficult to collision and brute-force attack. As the attacker cannot see the counterparties' addresses on the website, the attacker can only get their addresses from the Ethereum blockchain, and the attacker need to generate and maintain a table for each user it would like to attack, which will be much more difficult. Once the user thinks the address is compromised, it can easily register another account and transfer all its balance to the new account.

3. Unreliabile third-party banker. If the banker is a real man, then he might cheat and pick out the loser as the winner.

   **Solution**

   - As mentioned in Dealer (Banker) part, I implement the whole banker within the contract. Once the contract is deployed to the Main Ethereum Network, everyone can real the contract and the code, make sure I am actually not cheating at all. There is no real third-party banker involved in this game, the winner is picked out by the contract automatically. Therefore, we do not need to worry about the banker cheats or quits his job as long as the Main Ethereum Network functions. In the meantime, the banker will not maintain the transaction records or game histories, all these things are maintained in the contract. (see Balance Management and Coin Flipping Game)

# Measurement and Analysis

As I suppose I do not fully understand this requirement, I will discuss this topic from my own points of view.

## Evaluation of the Ethereum Network

I will compare the performance of Ethereum Network with other mainstream networks supporting DApp, namely Tron and EOS.

| | **Ethereum** | **EOS** | **Tron** |
|---|---|---|---|
| **Block Time (per block)** | 10 to 20 seconds | 0.5 second | 0.5 second |
| **TPS** | 10 | Not limited, max 3996 | Not limited, max 748 |
| **Time Consumption (per Game)** | 50 seconds | 2.5 seconds | 2.5 seconds |

As the consensus algorithm of Ethereum Network is Proof of Work (PoW) and the consensus algorithm of EOS and Tron is Delegated Proof of Stake (DPoS), the Block Time for EOS and Tron will be significantly reduced because DPoS does not require the worker to do hash value calculation for each block, and the worker can focus on maintaining the blockchain.

As has been mentioned by Vitalik Buterin on Ethereum GitHub Repository and other news, the Ethereum might switch to Proof of Stake (PoS) consensus algorithm in the future, which will lead to higher block time and larger TPS with less cost.

### Block Time of Ethereum Network

As the block time of Ethereum Network is 10 to 20 seconds per block, if we would like to run such a game on Ethereum Network, users will have to wait for at least 10 seconds for the next movement, which will be quite time-consuming.

For each round of the game, it will take at least 5 block time for my contract (see Block Time of the Contract). Therefore, if the contract is deployed on Ethereum Network, it will take at least 50 seconds for each game. However, if the game is deployed on EOS or Tron Network, it will only take 2.5 seconds for each game.

### Block Size and TPS of Ethereum Network

As of April 27, 2020, the unconfirmed transactions on Ethereum Network is $73028$. That is to say, even if the block time is 10 to 20 seconds per block, it will actually take a long time for the block to confirm (much longer than 10 to 20 seconds, because our transaction is waiting in queue).

## Evaluation of the Contract

### Gas Cost of the Contract

1. I have tried my best to cut down the deployment fee and the gas cost of the contract. I am not comparing my results with others, but I think with the same functions achieved, my contract will cost much less than others. Here is the point:
    - Use `mapping` instead of `array`. As mentioned on Stack Exchange, in Solidity, the `array` is built based on `mapping`, so manipulating the `array` will cost more gas than `mapping`.
    - Initialize an instance if the object is called many times in a function. For example, if the parameters in a certain `Game` struct needs to be modified many times, I will initialize a `Game storage game` instance for this maninpulation, and consequently the cost will be lower.

      **Evidence**
2. As what I have implemented, users on my platform need to deposit ETH into the contract before initialize/join a game, and this method will save $21000$ gas in each transaction comparing to a contract that users need to transfer ETH every time they initialize/join a game.
    - Deposit before playing: the contract will maintain a `balance` for this user, and every time the user interact with the contract, the contract is only manipulating the `balance` for this user, and transaction of ETH will not actually happen, which will save $21000$ gas for the action of sending ETH transaction.
    - Transfer every time: if there is no `balance` for the user, the users need to do a transaction every time they participate in a game, an a pure transaction of ETH will cost $21000$ gas. If the user be the winner, it will cost another $21000$ gas for receiving this transaction.

### Block Time of the Contract

As we are deploying this contract on Ganache, which will do auto-mining for all the blocks and the block time is not defined (no latency). However, in the actually Ethereum Network, the block time is 10 to 20 seconds per block. Therefore, the speed of the game will be significantly influenced by the block time.

**Comparision**

- Traditional (with real-person banker) 2-player minimum number of blocks for one round: at least 6 blocks/60 seconds
    - Initialize the game (1 block)
    - Join the game (1 block)
    - Generate hash values and submit (1 block)
    - Banker confirms the receiving of hash values, informs the participants to submit random number in clear text **(at least 1 block)**
        - As the banker need to send a transaction to the contract to confirm the receiving of hash values mannually, it will at least need 1 block for this transaction to be recorded in the blockchain. If the banker does not confirm in time, there might need more time for the confirmation.
    - Submit random number in clear text **(at least 1 block)**
        - As the participants needs to check the banker's information from time to time, they might not be able to submit the clear text once the banker inform them to submit. Therefore, it will cost at least 1 block time to submit.
    - Pick out the winner and house cleaning (1 block)
- My 2-player minimum number of blocks for one round: 5 blocks/50 seconds (2 players)
    - Initialize the game (1 block)
    - Join the game (1 block)
    - Generate hash values and submit (1 block)
    - `bothHashSubmitted` changed to `true` once all hash values received **(0 block)**
        - As the banker is implemented in the contract, once both hash values are received, the contract will automatically turned `bothHashSubmitted` to `true`, which is in the same block of the previous one.
    - Submit random number in clear text automatically **(1 block)**
        - As the JavaScript will listen to the change of the `bothHashSubmitted` parameter, once the parameter is changed, the JavaScript will send out the clear text automatically. Therefore, it will always take only 1 block time for this process.
    - Pick out the winner and house clearning (1 block)

# Reference


1. Khovratovich, D., et al. (2012). Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 Family, Berlin, Heidelberg, Springer Berlin Heidelberg.