Blockchains and Distributed Ledger

AKSHAY KANT

s1546937

1 Introduction

Ethereum is an open software platform based on blockchain technology that enable developers to build and deploy decentralized applications. In this report, we will be looking at smart contracts, which are self-executing contracts with term of agreement being directly written into lines of codes. We will be making extensive use of Solidity, a programming language for writing these contracts.

We will be writing, debugging, deploying and optimizing the smart contract. We will also be interacting with other smart contracts using our account. This simple smart contract we are building, enable users to buy tokens against ether, and the smart contract will eventually be initiating a lottery based on some probabilistic result and transferring all the collected ethers back to the winning user.

2 High level description of contract code and its address

In this section we will be describing the implementation of the our smart contract. We are using an interface IERC20-custom, which is being customised according to our contract and contain all the functions which we need to implement in the main contract. We are using this interface to customize code without having to implement everything from scratch. For example, in the case of IERC20-custom, we could change/customize the logic for the functions balanceOf and transfer and still use the rest of the framework.

The main point of using interfaces (or Abstract Contracts) is to provide a customizable and re-usable approach for your contracts.

This interface best describes the contract code from high level.

```
pragma solidity ^0.4.0;

contract IERC20_custom {

    //buy token for ether
```

```
    function buyTokens(uint256 number_of_tokens) payable public;

    //Get the total token sold
    function tokenSold() returns (uint256 tokenSold);

    //Get the ether earned
    function etherEarned() returns (uint256 etherEarned);

    //Get the account balance of another account with address _owner
    function balanceOf(address _owner) returns (uint256 balance);

    //Send _value amount of tokens to address _to
    function transfer(address _to, uint256 _value) private;
}
```

The main functions of the smart contract is divided into these following solidity functions:

- **buyTokens(uint256 number_of_tokens) payable public:** This is a public function which allocates token to the parties based on the ether. This function is being used in two ways - directly and from the fallback function.
  If using directly, it checks for the amount of ether and the number of tokens, then executes the transaction by using the private transfer function. It allocates tokens to the party and keeps the mapping of the party and the number of tokens they hold.
  If used through the fallback function, it accepts only the ether and automatically calculates the number of token user can buy and delegates the task to the buyTokens function.
  In case the amount of ether or the number of tokens are not correct, or there is any fallback in the transaction, the amount is triggerred back to the party's account.
  The buyToken function is responsible for delegating the work of selecting the winner from the stakeholder and updates all the distribution of the parties buying the tokens.

- **tokenSold() returns (uint256 tokenSold):** This function returns the total token sold. The tokens are allocated against the ether.
  For this contract the token value is:
  1 token = 1 finney
  where 1 ether = 1000 finney

  The threshold value at which the lottery will be initiated is set as 0.1 ether.

- **etherEarned() returns (uint256 etherEarned):** This function stores the ether which the smart contract is holding. These ethers are received

by offering tokens to the parties.

- **balanceOf(address _owner) returns (uint256 balance):** This function gives the number of token a party holds. This can be returned against the address of the party. It is stored in a mapping of address of token buyer and the number of token the address holds.
- **transfer(address _to, uint256 _value) private** This is a private function which is called internally from the buyToken function. This is called in any fallback if the transaction is not executed successfully or to transfer the ether to the winning stakeholder.

Address: There are two types of account in this blockchain - one is the normal account which has its own address and ether. Another is the smart contract account, which is identified with its own address and holds ether and token in our case.
Address of our normal accounnt - metamast account -
0x37DBB00D99f816F0D0373BCC768d6b0fD31eb4e2
Address of this smart contract -
0xd2a5db0346fb7b39b4c53b06d38057a04d322782

The address for an Ethereum contract is deterministically computed from the address of its creator (sender) and how many transactions the creator has sent (nonce). The sender and nonce are RLP encoded and then hashed with Keccak-256.

3 How a winning stakeholder is selected

The tokens issued by the contracts are absolute values, which means they are discrete random variables. The distribution is right skewed with long tail, as the tokens will be having only positive values associated with them.

A winning stakeholder is selected from a random distribution of the stakeholders. A single stakeholder buying huge amount of token and reaching threshold is small as the distribution has long tail. On the other hand, a stakeholder buying tokens multiple times has a higher probability of winning.

The winning stakeholder is selected from the distribution when a threshold value of 0.1 ether is reached. As soon as this threshold is reached in this smart contract, the lottery is started which selects one winning stakeholder's address from the distribution of the stakeholder's addresses.

Now let D be the ideal distribution of index selection, and the contract has n tokens in total. The probability distribution of D is:

$$D \xleftarrow{r} \{0, 1, 2, 3, ..., n-1\}$$

Let U be the distribution of index selection in this contract, and g is generator of a cyclic group of order n. The probability distribution function of U is:

$$U = \{x \xleftarrow{r} \{0, 1, 2, 3, ..., 2^{256} - 1\} : g^x\}$$

The statistical distance between D and U is:

$$\Delta(D, U) = \frac{1}{2} \sum_{i=0}^{n-1} |Prob(D = i) - Prob(U = i)|$$

And now consider if 2^{256} modular n is equal to 0, which means that U is uniformly distributed as D.

$$Prob(U = i) = \frac{\frac{2^{256}}{n}}{2^{256}} = \frac{1}{n} = Prob(D = i)$$

which means U distribution is exactly the same as D distribution, and statistical distance between these two distributions is 0.

4 Steps of deployment and engagement with other contract

The smart contract developed using remix was first connected to the blockchain using metamask.

Then the Injected Web3 environment was selected on remix and smart contract was initiated to be deployed on blockchain.

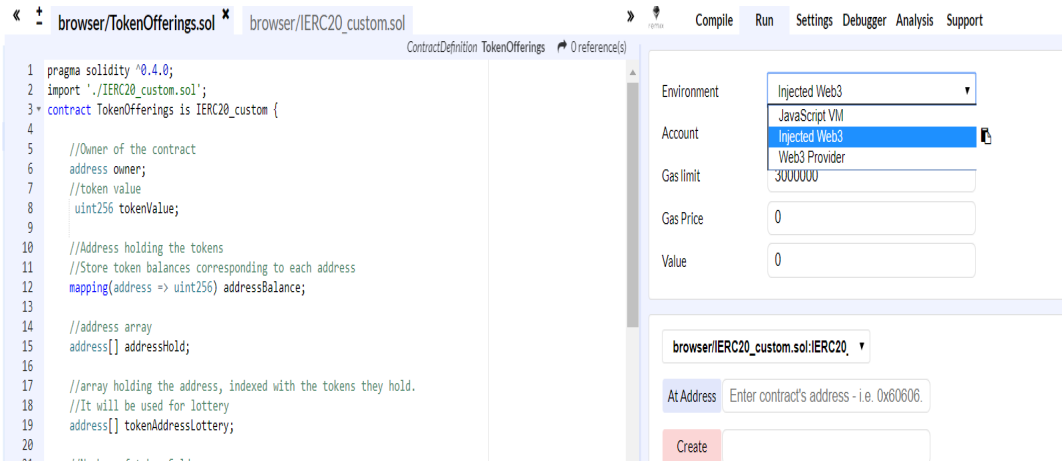

Figure 1: Deploy on blockchain: Remix

Next, Metamask page will appear and by clicking on submit, the contract is send to the blockchain.

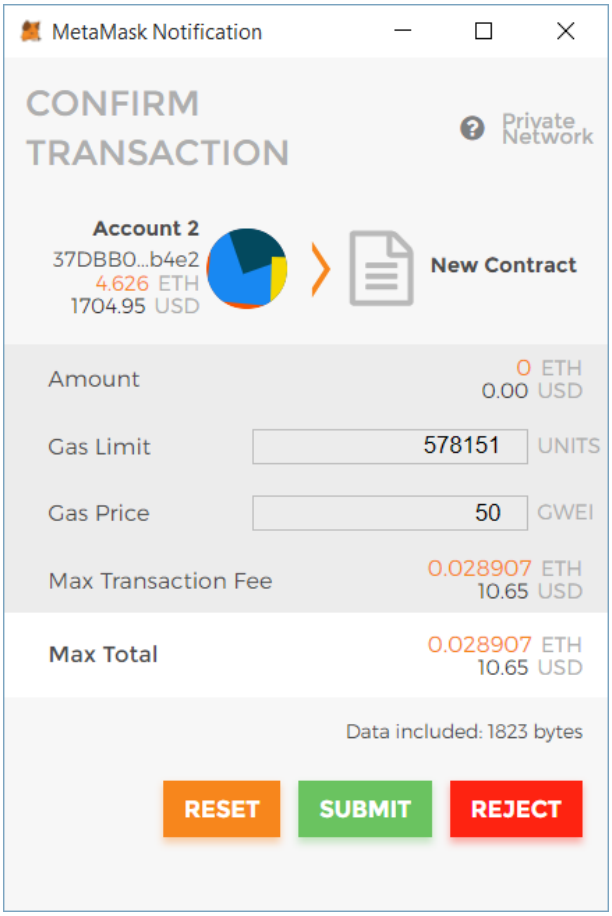

Figure 2: Deploy on blockchain: Metamask

On successful deploying of smart contract, an address is associated with the contract.
The address of this deployed smart contract is:
0xd2a5db0346fb7b39b4c53b06d38057a04d322782

This address is now used to interact with this smart contract.
This is the log from remix which has the account used to deploy the smart contract and the amount of gas consumed in completing the task.

Figure 3: Deploy on blockchain: Deployement Log

Engagement with other contracts
Metamask was used to contact with other contracts on the blockchain. The communication was done using the address of other smart contracts.
For instance the transaction of 0.5 ether was done to this address:
0x00ce06cd61e021d534a64e44b08f80e4c2e102b3,
in order to buy token from this contract.

This was done by selecting the address and the amount needed to buy the tokens. This address is the source of interaction.

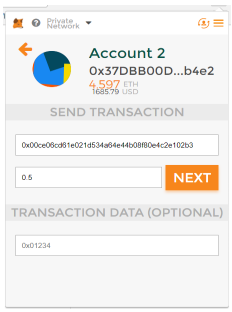

Figure 4: Interaction on blockchain: buy token from other contract

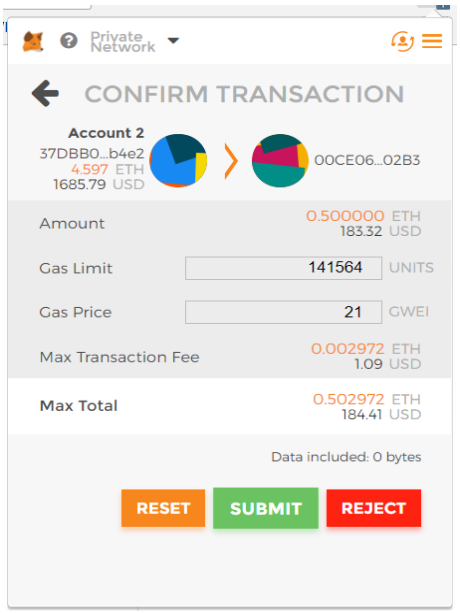

Figure 5: Interaction on blockchain: buy token from other contract

5 Gas Overview

Gas is the name for a special unit used in Ethereum. It measures how much work an action or set of actions takes to perform: for example, to calculate one Keccak256 cryptographic hash it will take 30 gas each time a hash is calculated, plus a cost of 6 more gas for every 256 bits of data being hashed. Every operation that can be performed by a transaction or contract on the Ethereum platform costs a certain number of gas, with operations that require more computational resources costing more gas than operations that require few computational resources.

The reason gas is important is that it helps to ensure an appropriate fee is being paid by transactions submitted to the network. By requiring that a transaction pay for each operation it performs (or causes a contract to perform), we ensure that network doesn't become bogged down with performing a lot of intensive work that isn't valuable to anyone.

]

Type	Gas Consumed	Tokens Issued
Deployment	538151	-
Token acquisition 1	1411125	50
Token acquisition 2	1725537	40
Token acquisition 3	1197274	30

Table 1: Gas consumption overview.

The amount of gas consumed for deployment is 538151 wei of gas.

Figure 6: Deployment of smart contract: gas consumed

As we can see from the table and the token acquisition gas cost, the Token acquisition 1 takes 1411125 wei of gas.

Figure 7: Smart Contract: Token Acquisition 1

It consume this amount of gas as the buyToken function and adding to the distribution will take.
Token acquisition takes 1725537 wei of gas, which is more than the other token acquisition, as it is iterating over the loop and adding to the distribution.

Figure 8: Smart Contract: Token Acquisition 2

Token acquisition takes 1197274 wei of gas, and this initiates the lottery as threshold is reached and computing the winning stakeholder and transferring ether to the winning stakeholder.

Figure 9: Smart Contract: Token Acquisition 3

6 Smart Contract : Fairness

The fairness of smart contract is based on the gas charged by the users to interact with the contract. If the gas charged is same for all the users to interact with the smart contract than the contract is said to be fair.
The current contract is not fair i.e. for different users the gas charged is different. The main purpose of this contract is to issue tokens against ether and run lottery to find the winning stakeholder when threshold is reached. So the amount of gas which can differ from user to user is based on the number of tokens they are buying and whether the threshold has been reached.
The more the token bought, the more will be the gas, and this is fair as the percentage of amount which can be thought as the transaction charge will be similar for all the users. Suppose a user is buying 10 tokens, then the amount of gas consumed is 2000000 wei and other user buying 20 tokens, the gas consumed will be 4000000 wei. So more gas is consumed when more tokens are bought

and the transaction percentage remains same.

Another parameter of gas consumption is selection of winner once the threshold is reached. The user who is initiating the lottery is charged more gas, which is unfair. As we can see from the smart contract, the computation of selecting the winning stakeholder from the distribution and then resetting all the parameters back for the next cycle is been done by the user who initiated it. The approach of solving out this problem and making the smart contract fair by optimizing these and other issue, we will be looking in next section.

7 Smart Contract : Optimization

These are few technique to optimise the code:
- **Rollback Transaction:** This smart contract needs right amount of ether to buy tokens. If someone enters different amount, than the user will lost the ether and cannot buy the tokens. So, if the number of tokens or the ether amount does not match then we need to rollback and transfer the ether back to the user. The line commented as revert back the tokens, shows the code in action.

```
function buyTokens(uint256 number_of_tokens) payable public{
       if(number_of_tokens > 0 &&
           //check amount to buy token
           msg.value != number_of_tokens * tokenValue
           ){
           //revert back the tokens on error
               transfer(msg.sender, msg.value);

           }
           else {
       //Storing the address and token bought
       addressBalance[msg.sender] += number_of_tokens;
       //Storing the eth earned
       ethEarned += msg.value;


       //Storing the address in the array for every input
       //token corresponding to the index
       for(uint256 i = tokensSold; i < number_of_tokens; i++){

           tokenAddressLottery.push(msg.sender);
       }

       //Storing the number of tokens sold
       tokensSold += number_of_tokens;
```

```
        //add the address
        addressHold.push(msg.sender);


        if(ethEarned > ethThreshold){
            runLottery();
        }
            }

    }
```

- **If threshold is not reached:** Currently we are not having an end period for the lottery to reset. We need to set the time frame in which if the threshold is not reached, it will give the ether back to all the participant and reset the scheme.
This can be done by setting the time frame using **block.number**, and once the time is over we need to transfer back the token holding ether to the stakeholders and reset the lottery scheme.

- **Optimising Data Types and Structures:** Currently some of data types and the structures used are consuming a lot of gas. This can be optimised by using strcuts and libraries which can provide a wider range and extending the limit of the current framework.
Resetting the values once the winning stakeholder is selected in also consuming large gas which can be optimised by using auxiliary data (or structure) to track when a value of zero has been explicitly set. A lightweight approach would be to add a bool property to the struct (say named initialized), and to set it to true when the zero is explicitly set. As all bools are false (0) by default, we can check against the value true.

```
contract C {
    uint[] counters;
    function getCounter(uint index)
        returns (uint counter, bool error) {
            if (index >= counters.length) return (0, true);
            else return (counters[index], false);
        }
    function checkCounter(uint index) {
        var (counter, error) = getCounter(index);
        if (error) { ... }
        else { ... }
    }
}
```

- **Optimizing for fairness:** As discussed in the previous part about fairness, we can make our contract fair and solve the problem where a user initiating the lottery leading to compute the winning stakeholder and consuming more

gas. The solution for this problem is once the winning stakeholder is identified, we can give the amount of gas consumed as reward from the winning stakeholder and give it to the user who initiated the lottery and the reaming to the winning stakeholder. This way we can make out smart contract fair for all users.

8 Smart Contract : Code Transaction History

This interface best describes the contract code from high level.

```
pragma solidity ^0.4.0;

contract IERC20_custom {

    //buy token for ether
    function buyTokens(uint256 number_of_tokens) payable public;

    //Get the total token sold
    function tokenSold() returns (uint256 tokenSold);

    //Get the ether earned
    function etherEarned() returns (uint256 etherEarned);

    //Get the account balance of another account with address _owner
    function balanceOf(address _owner) returns (uint256 balance);

    //Send _value amount of tokens to address _to
    function transfer(address _to, uint256 _value) private;
}
```

This is the main smart contract - TokenOfferings which in implementing the IERC20_custom interface.
The global variables are declared and initialised in this block.

```
pragma solidity ^0.4.0;
import './IERC20_custom.sol';
contract TokenOfferings is IERC20_custom {

    //Owner of the contract
    address owner;
    //token value
     uint256 tokenValue;

    //Address holding the tokens
    //Store token balances corresponding to each address
```

```
    mapping(address => uint256) addressBalance;

    //address array
    address[] addressHold;

    //array holding the address, indexed with the tokens they hold.
    //It will be used for lottery
    address[] tokenAddressLottery;

    //Number of token Sold
    uint256 public tokensSold = 0;
    //ethereum earned
    uint256 public ethEarned = 0;

    //Threshold is set to 1 ether
    //Run the lottery after the threshold value is reached
    uint256 ethThreshold;
```

The constructor takes care of assigning the owner at the time of contract deployment. It also set the exchange value against ether and the threshold which will be used to initiate the lottery to select the winning stakeholder.

```
function TokenOfferings() public{
        owner = msg.sender;
        //(1ether = 1000finney)
        //Buying rate of 1 Token is 1000 wei
        //(Solidity converts all unitless values from ethers to wei).
        tokenValue = 1 finney;
        //set the threshold value
        ethThreshold = 0.1 ether;
    }
```

This is the fallback function which will be initiating the transaction incase the payable amount is mentioned and inturn delegating the task to the buyToken function.

```
//fallback function
    function () payable public {
        //since it does not accept any input parameters,
        //so we will make them buy one token
        buyTokens(msg.value/tokenValue);
    }
```

The buyToken is a public function which allocates token to the parties based on the ether. It executes the transaction by using the private transfer function.

It allocates tokens to the party and keeps the mapping of the party and the number of tokens they hold.

```
function buyTokens(uint256 number_of_tokens) payable public{
        if(number_of_tokens > 0 &&
            //check amount to buy token
            msg.value != number_of_tokens * tokenValue
            ){  //revert back the tokens on error
                //owner.transfer(msg.value);

            }
            else {
        //Storing the address and token bought
        addressBalance[msg.sender] += number_of_tokens;
        //Storing the eth earned
        ethEarned += msg.value;


        //Storing the address in the array for every input token
        //corresponding to the index
        for(uint256 i = tokensSold; i < number_of_tokens; i++){

            tokenAddressLottery.push(msg.sender);
        }

        //Storing the number of tokens sold
        tokensSold += number_of_tokens;

        //add the address
        addressHold.push(msg.sender);


        if(ethEarned > ethThreshold){
            runLottery();
        }
            }

    }
```

This function will be initiated to compute and decide the winning stakeholder and transferring the winning ether to their address.

```
 //run the lottery
    function runLottery() private{
```

```
        //access the head of chain
        uint256 blockNumber = block.number;
        //access the hash value of the head of chain
        bytes32  blockHashNow = block.blockhash(blockNumber);

        //Find the lottery by doing the block hash
        //modulus(%) number of tokens sold
        uint256 lotteryWinner = uint256(blockHashNow) % tokensSold;

        address lotteryWinnerAddress = tokenAddressLottery[lotteryWinner];


        //transfer all the ether to the winning address
        transfer(lotteryWinnerAddress, ethEarned);

        //Todo: gas calculation
        //transfer the gas to the address which initiated lottery
        //transfer(msg.sender, ethEarned);

        //Set all variables to null
       tokensSold = 0;
       ethEarned = 0;


       for(uint del1=0; del1 < addressHold.length; del1++ ){
           addressBalance[addressHold[del1]] = 0;
           delete addressHold[del1];
       }

        for(uint del2=0; del2 < tokenAddressLottery.length; del2++ ){

           delete tokenAddressLottery[del2];
       }


    }
```

It returns the number of token sold by the contract.

```
//Get the total token sold
    function tokenSold() returns (uint256 tokensSold){
        return tokensSold;

    }
```

This returns the total number of ether earned by the smart cntract against the

token issued.

```
  //Get the ether earned
     function etherEarned() returns (uint256 etherEarned){
         return etherEarned;
     }
```

This returns the number of token hold by a partiicular address.

```
  //Get the account balance of another account with address _owner
     function balanceOf(address _owner) returns (uint256 balance){
         return addressBalance[_owner];
     }
```

This function is used to transfer ether to other address either once the winning stakeholder is decided or during the transaction failure.

```
  //Send _value amount of tokens to address _to
     function transfer(address _to, uint256 _value) private{
         //check whether the amount to be transfer is available
         //in etherEarned and we are not tranferring 0 value.
         require(
             ethEarned <= _value &&
             _value > 0
             );
         //logging the winner
        TransferLog(_to, _value);

         ethEarned -= _value;
         //transfer ether
         _to.transfer(_value);

     }

     event TransferLog(address _to, uint256 _value);

 }
```

Transaction History

Wallet
Address of Wallet:
0x37DBB00D99f816F0D0373BCC768d6b0fD31eb4e2

Here are the transaction for Wallet

0.5ether worth of token were bought from this smart token from the wallet.
0x00ce06cd61e021d534a64e44b08f80e4c2e102b3

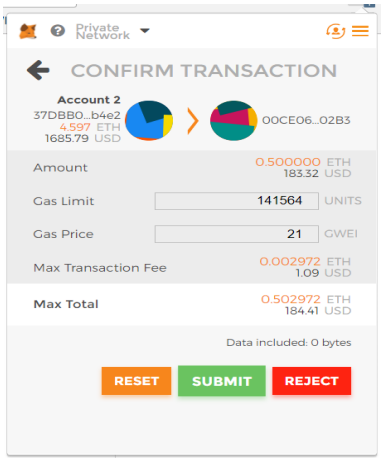

Figure 10: Wallet: Transaction 1

0.7ether worth of token were bought from this smart token from the wallet.
0xaf19eb290071d7ad0c5578296983475c603643e9

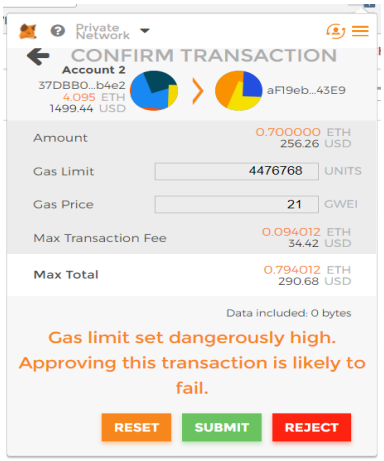

Figure 11: Wallet: Transaction 2

Smart Contract
Address of Smart Contract:
0xd2a5db0346fb7b39b4c53b06d38057a04d322782
Here are the transaction for Smart contract.
This address bought 50 tokens from the smart contract with 0.05 ether.
0x740b4281f99c295ff2c2e44ec2897dc18ccbc51b

Figure 12: Smart Contract: Transaction 1

This address bought 40 tokens from the smart contract with 0.04 ether.
0x4a8cacecc537c0b71b8d86a11c55928e0c8663f8

Figure 13: Smart Contract: Transaction 2

This address bought 30 tokens from the smart contract with 0.03 ether.
0x9cc73217ec996065bf1ccbcf85972e5da702c279

Figure 14: Smart Contract: Transaction 3