School of Informatics
Blockchains and Distributed Ledgers

Assignment 3

Erodotos Demetriou (s2187344)

Part 1

For Part A of this assignment we are required to interact with a smart contract deployed on the *Ethereum Ropsten network*. In more detail, we should call the *register* method, giving as input arguments a secret key and our student number. Since *Ethereum* is a public Blockchain, everything can be accessed on chain. Hence, the following script was used to retrive the secret key from the smart contract's storage.

```
const Web3 = require('web3');

var web3 = new Web3(new Web3.providers.HttpProvider('https://
    ropsten.infura.io/v3/0ab1814012ad4231965d67bf98a40b1a'));

var contract_address = '0xde3a17573B0128da962698917B17079f2aAbebea';

web3.eth.getStorageAt(contract_address, 1).then(result => {
    console.log(web3.utils.hexToString(result));
});
```

The found key is : *actually...;)*. We imported the smart contract source code and deployed address into remix and finally called the *register* function with input *actually...;)* and *s2187344* as inputs.

Transaction Hash : 0xc6ebbfea38258b115d07d5826667dfdd2445a5cbe277f0c3e18fa5a294ef3d32

Part 2 A

Smart Contract High-Level Description

Part 2 A of this assignment regards implementing and deploying a custom token. Its requirements include allowing users to exchange ETH for tokens, sell their tokens back for ETH, and transfer tokens between accounts. Also, the users can view their balance by calling the smart contract. Besides that, the token owner can change the token's price if there is enough balance in the smart contract.

To achieve that, we introduced the concept of liquidity providers. Liquidity providers are individuals willing to deposit their ETH to the smart contract and receive tokens as rewards whenever a new buyer purchases the token in the future. Next, we provide a real-world example of using this mechanism to understand its functionality better.

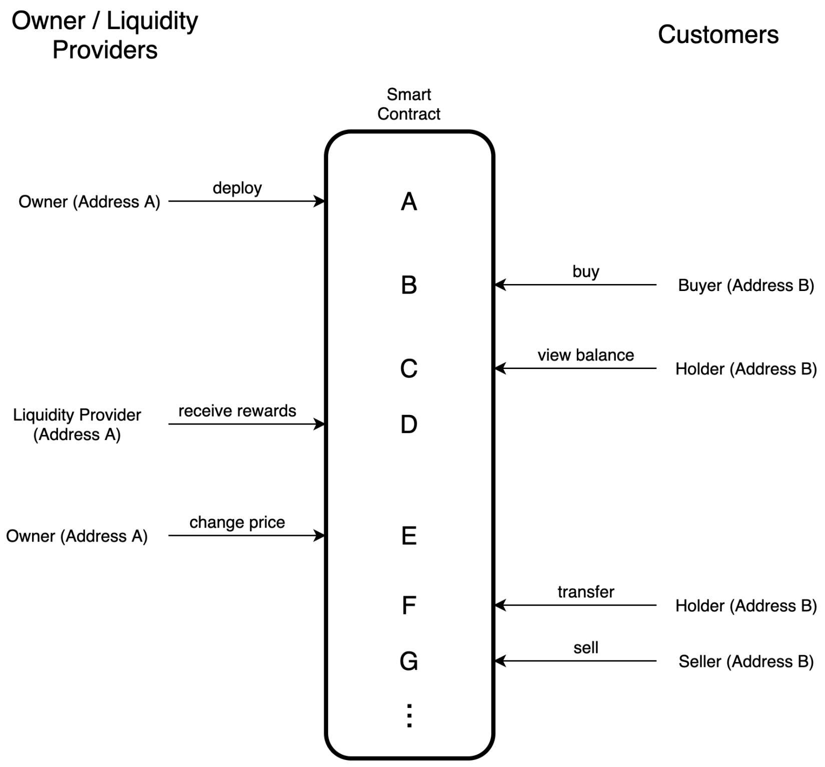

Figure 1: Smart Contract Execution Workflow Example

Figure 1 illustrates the smart contract's life cycle. In step A, the owner (address A) deploys the smart contract providing 100 Wei. This automatically registers him as the first liquidity provider. Next, in step B, we can observe a buyer (address B) conducting a buy request to the smart contract. For instance, he wants to buy 30 tokens priced at ten Wei each. Hence, he has to send 300 Wei to the smart contract and provide the number of tokens as an input parameter. We check for the validity of the input data, and if everything is accurate, we proceed. 10% of the purchased tokens is contributed into a rewards pool while the customer receives 90%. We have to mention that for simplicity objectives in calculating the percentage, we allow purchases of multiples of 10. Later, in step C the buyer can view its balance by calling the getBalance() method of the smart contract. At this point (D), a liquidity provider can receive rewards from the rewards pool. He will get a portion of tokens according to the fraction of liquidity he provided to the total liquidity. In other words, each provider can increase the rewards he receives by offering more liquidity.

Nonetheless, we have to note that because solidity does not support floats, we restrict the amount a liquidity provider can offer to multiples of 100 Wei. Further, if there is only one token in the rewards pool and two liquidity providers own 50% of it each, the smart contract will not allow them to receive rewards because of an impossible division of 1/2. They can receive their rewards later when the pool balance allows a fair distribution.

Now, suppose we want to change the price of our token. The owner can allow this action in step E. The only constrain of this transaction is that the contract's balance should be greater or equal to the product of circulating tokens and the new price.

Moreover, a token holder can transfer its tokens to another address. This is depicted in step F, where the user calls the smart contract providing the receiver's address and the transfer amount of tokens.

Finally, any token holder may sell his tokens to the smart contract for ETH considering the current token price in Wei. The sell token function does not directly transfer ETH to the user. Instead, a custom library deployed on the Ropsten network is used. One of the main challenges of this assignment was to link an already deployed library into our smart contract. This was impossible to achieve through remix online editor. Thus we used Truffle smart contract development framework to achieve the desired outcome. Practically, we compiled the customLib.sol locally to generate a reference ABI and afterward import the library into our Token.sol contract. Since we needed to use an already deployed instance of the library, we specified in the migrations javascript file its address and used the link method to reference it. There follows the deployment and link script for the operation as mentioned earlier.

```
const Token = artifacts.require("Token");
const CustomLibrary = artifacts.require("customLib");

module.exports = function(deployer, accounts) {
    CustomLibrary.address = "0xc0b843678E1E73c090De725Ee1Af6a9F728E2C47"
    deployer.link(CustomLibrary, Token);
    deployer.deploy(Token, { value: "100" })
};
```

There follows a detailed list and description of smart contract's variables, functions and events.

Variables

address owner: This variable is set on deployment, and stores the smart contract's owner address.

uint256 contractBalance: This variable stores the number of *wei* deposited into the smart contract.

uint256 circulatingTokens: This variable stores the number of circulating tokens.

mapping balances: This is a mapping variable maintaining a one to one correspondance between a user address and the number of tokens it holds.

mapping liquidityProviders: This is a mapping variable maintaining a one to one correspondance between a liquidity provider user address and the amount of liquidity it provided in *wei*.

uint256 numberOfProviders: This variable holds the number of total liquidity providers.

uint256 totalLiquidity: This variable holds the amount of total liquidity provided during the complete life cycle of the smart contract.

uint256 rewardsPool: This variable stores the amount of tokens that will be available to liquidity providers as rewards for their services.

uint256 tokenPrice: This variable stores the current token price in *wei*.

Events

event Purchase: Whenever a token purchase is fulfilled, an event stating the buyer's address and the purchased amount, is emitted.

event Sell: Whenever a token holder sells its tokens back to the smart contract, a sell event is emitted. This event states the seller address and the amount of tokens sold.

event Transfer: Whenever a token holder transfers its tokens to an other address a transfer event is emitted. This event states the sender and receiver adddresses, as well as the transfered amount of tokens.

event Price: This event is emitted whenever the smart contract owner changes the token price.

Functions

constructor(): This is the smart contract constractor. It is called whenever the code is deployed for the first time, and sets the contract owner, as the initiator of the transaction. The smart contract constructor requirement is that the owner provides 100 *wei* as initial liquidity. Thus he becomes the first liquidity provider.

buyToken(): Whenever a buyer wants to exchange his ETH for our token has to call this function. In more detail, he needs to provide as input the number of tokens that he desires to purchase and send the proportional amount of ETH, considering the token price which is in *wei*.

transfer(): This function can be called by a token holder that desires to transfer a specific amount of tokens to an other address.

sellToken(): This function enables a token holder to exchange his tokens with ETH. He can provide the number of tokens he wants to exchange and subsequentyl he will receive the according value of ETH considering the current token price.

changePrice(): This function can be called only by the smart contract owner, to change the token price. Its execution will only succed if there is enough liquidity to pay token holders if they decide to sell their token simultaneously.

getBalance(): This function returns the balance of tokens that are in the posetion of the fucntion's caller.

provideLiquidity(): This is an internal function that can be triggered through the fallback function. It receives an address and the amount of provided liquidity. Subsequentyl, it registers this user as a liquidity provider.

getReward(): This is an internal function that can be trigered through the fallback function and only if the caller is a registered liquidity provider. Its execution results into receiving a proportion of the reward tokens from the reward pool.

fallback(): This function can be used with different arguments and can serve different purposes. If it is called providing some ETH, registers you as a liquidity provider; Else, it can be used by liquidity provider to receive its rewards.

Gas Costs Evaluation

This section measures and evaluates the gas cost we expect to have when deploying and interacting with the Smart Contract.

Gas costs appear in the following table, and they are unquestionably high. The Smart Contract owner has to pay 1,801,345 gas units to deploy the code, which is approximately $995 (price on 26/10/21). This is definitely a high amount of gas which makes the Smart Contract not so gas efficient.

Gas Costs

Contract Owner Fees	
Contract Deployment	1,801,345

Customer Fees	
Contract Deployment	1,801,345

Liquidity Provider Fees	
Contract Deployment	1,801,345

Potential Hazards and Vulnerablities

Developing a Smart Contract can always be challenging. This is because, on a public-permissionless blockchains, everything is observable by everyone. This fact makes it difficult when it comes to securing users' data. Besides that, a developer has to be alert to write code that is attack-resistant. Smart Contracts expose a broad spectrum of vulnerabilities, enabling an adversarial entity to exploit them for his interest.

When developing the Matching Pennies game, we took into consideration possible attacks. The following list presents vulnerabilities and mechanisms to moderate them.

DoS(Denial of Service) - Griefing: An attacker attempts to make a Smart Contract get stuck when executed. In the case of our implementation, a player might grieve and stop playing to halt the Smart Contract or make his opponent lose money. This is not wanted since the other player's ETH will get stuck, and no one else will be able to play the game.

Mitigation: To counter this type of attack, we implemented a time limit mechanism. When a player interacts with the Smart Contract, a timer is initiated. The other player has only 10 minutes to play his move. If the time limit expires, the last player can request a refund and cancel the game. The funds of the lapsed player will be kept as punishment. If a single player tries to halt the program (i.e., only one player joins the game and his opponent refuse), the Smart Contract owner can join, to force the first player to play. If the first player denies playing, he will lose his money since the contract owner can request a refund, which will reset the game. Using this technique, the Smart Contract can never halt.

Re-Entrancy: An attacker might try to take advantage of the Smart Contract withdraw function by executing a re-entrancy attack. In more detail, when he invokes a withdrawal transaction, he can drive his transaction to a malicious fallback function on another Smart Contract that can recursively call again and again the withdraw function, trying to get more ETH.

Mitigation: In order to avoid such unpleasant attacks, we execute the code of the Smart Contract in a particular way. When there is a withdrawal invocation, we check some constraints to ensure that the transaction sender is allowed to withdraw ETH. After that, we will pass any updates to the state of the Smart Contract and eventually make the call that sends the requested ETH to the recipient. It is important to make the ETH transfer after changing the Smart Contract state. If a reentrancy attack occurs on the next transaction, it will be stopped because function constraints will evaluate the transaction according to the updated state.

Front-Running: This attack happens on the Miner level. An attacker might clone your transaction and put a much higher gas limit on it. This results in the inclusion of his transaction to the next block instead of yours. This is inconvenient since another player might steal your spot in the game.

Mitigation: There is no straightforward solution since the problem lies at the transaction mining level. For the Matching Pennies game, front-running will not have a significant effect, except that a player might steal the position of another one. The disfavored player will have the opportunity to play in another moment.

Overflow/Underflow: Matching Pennies game does not face this problem since it does not receive any integer value from the transaction sender.

Randomness Source Exposure: Matching Pennies game does not face this problem since it does not use any random value during the code execution.

Delegation: Matching Pennies game does not face this problem since it does not use code from other Smart Contracts or Libraries.

Other good practices:

- Use *call()* instead of *transfer()* or *send()*. Using *call()* might be insecure, but *transfer()* and *send()* can forward only 2300 gas. In the future gas costs might change, and 2300 gas might not be enough. As a result using *call()* properly is the right approach. Below we can see an example of using *call()* function and handling its outcome correctly.

```
        (bool success, ) = msg.sender.call.value(amount)("");
        require(success, "Transfer failed.");
```

- If you need to have a callback function, keep it simple.
- Any public-permissionless blockchain reveals transaction details to the ledger. Due to this fact, in data-sensitive applications such as the Matching Pennies game, we need to hide users' data. We can do this by following a two-phase process: committing a value obscured by hashing the bet and some salt and then exposing the real value.

Smart Contract Deployment and Execution History

Owner Address: 0x79BE6e946368520419BF4A20aC45b28fd3a5b2bA

- **Smart Contract deployment**

  TX Hash:
  Contract Address: 0xF87a42464eEf144fF0C81cE8d5E927548CF4695D

- **Buy Tokens**

  TX Hash:

- **Get Rewards**

  TX Hash:

- **Change Price**

  TX Hash:

- **Transfer Tokens**

  TX Hash:

- **Sell Token**

Implementation Code

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.00 <0.9.0;

import "./customLib.sol";

contract Token {
    address private owner;
    uint256 private contractBalance = 0;
    uint256 private circulatingTokens = 0;

    mapping(address => uint256) private balances;
    mapping(address => uint256) private liquidityProviders;
    uint256 private numberOfProviders = 0;
    uint256 private totalLiquidity = 0;
    uint256 private rewardsPool = 0;

    uint256 public tokenPrice = 10;

    event Purchase(address indexed buyer, uint256 amount);
    event Sell(address indexed seller, uint256 amount);
    event Price(uint256 price);
    event Transfer(
        address indexed sender,
        address indexed receiver,
        uint256 amount
    );

    constructor() payable {
        require(
            msg.value == 100,
            "As the contract owner you are required to provide 100 wei
               liquidity"
        );
        owner = msg.sender;
        provideLiquidity(owner, msg.value);
    }

    function buyToken(uint256 amount) public payable returns (bool) {
        require(
            msg.value == amount * tokenPrice,
            "Your funds are not sufficient"
        );
        require(
            amount % 10 == 0,
            "The purchased quantity should be multiple of 10!"
        );
        circulatingTokens += amount;
        contractBalance += amount * tokenPrice;
        balances[msg.sender] += (amount * 90) / 100;
        rewardsPool += (amount * 10) / 100;
        emit Purchase(msg.sender, amount);
        return true;
    }

    function transfer(address recipient, uint256 amount) public returns (
        bool) {
        require(balances[msg.sender] >= amount, "Not enough balance");
        balances[msg.sender] -= amount;
        balances[recipient] += amount;
        emit Transfer(msg.sender, recipient, amount);
        return true;
    }

    function sellToken(uint256 amount) public returns (bool) {
        require(amount >= 1, "Provide positive amount of tokens!");
        require(amount <= balances[msg.sender], "Not enough balance!s");
        require(amount > 0, "Not enough balance!");
        circulatingTokens -= amount;
        balances[msg.sender] -= amount;
        contractBalance -= amount * tokenPrice;
        bool success = customLib.customSend(amount * tokenPrice, msg.
           sender);
        require(success, "Transfer failed!");
        emit Sell(msg.sender, amount);
        return true;
    }

    function changePrice(uint256 price) public returns (bool) {
        require(
            msg.sender == owner,
            "This function is restricted to the contract's owner"
        );
        require(
            contractBalance >= circulatingTokens * price,
            "Not enough liquidity"
        );

        tokenPrice = price;
        emit Price(price);
        return true;
    }

    function getBalance() public view returns (uint256) {
        return balances[msg.sender];
    }

    function provideLiquidity(address provider, uint256 amount) internal {
        liquidityProviders[provider] += amount;
        totalLiquidity += amount;
        contractBalance += amount;
        numberOfProviders += 1;
    }

    function getReward() internal {
        require(
            liquidityProviders[msg.sender] >= 100,
            "You are not a liquidity provider"
        );

        require(
            rewardsPool % numberOfProviders == 0,
            "You can not receive rewards now (unfair division)"
        );

        uint256 allowance = (liquidityProviders[msg.sender] /
           totalLiquidity) *
            100;
        balances[msg.sender] += (rewardsPool * allowance) / 100;
        rewardsPool -= (rewardsPool * allowance) / 100;
    }

    fallback() external payable {
        if (msg.value > 0) {
            provideLiquidity(msg.sender, msg.value);
        } else {
            getReward();
        }
    }
}
```

Part 2 B