Blockchains and Distributed Ledgers - Coursework 4

s1732368

January 18, 2021

1 A detailed description of your contract's design

I use variables to store the owner of the contract, owner, and to mark the agree, transfer and pay-out stages the two users are at, agreeStage, transferStage and payOutStage. I also use boolean variables to mark whether the contract needs to return the user's sent tokens, returnTokens, and if the contract needs reset, needReset. timeOut is used to track the last time the contract was interacted with and see if it needs to be reset. I also have a struct, users, that is used to store the user information, such as their address and the tokens they agree to send and receive.

agree() - This function is for the two users to agree to the tokens they are sending and receiving, and for the users to send the address of their Tokens contract. Firstly, it is checked if the contract needs to be reset and, if so, the reset() function is called. Then, it is checked if the swap has timed out (5 minutes) - if so, false is returned and an event is emitted to notify the user. Checks are performed to ensure a swap is not already under way. The users must make a deposit of 1 ether (which would probably be lowered if this were deployed to be used by actual users) which will be used to refund the user who spends more gas on the contract, if they do not send this deposit then the transaction will fail. The user's information is stored in the user struct. Once both users have entered their agreed token transfer, a check is performed to ensure they match and if there is a difference then the swap is marked as failed - the users will be told to advance to the payout stage to receive any tokens that they transferred to the contract. If the two users agree, and they have also both ran transfer transactions, then they may advance to the payOut stage. The gas used by the users is stored in the user struct.

transfer() - Firstly, it is checked if the contract has timed out. Then, various checks are made to ensure this user is able to make a transfer transaction. If the user has not transferred enough tokens to the contract then the swap is marked as failed and the users will be told to advance to the payOut stage to get their tokens returned to them. If enough tokens have been transferred then this is marked in the user struct. If both players have transferred the necessary tokens then they can advance to the payout stage. Gas usage is tracked in this function.

payout() - Similar checks are performed as in the transfer function. If the swap is marked as failed, then the user who performed the payOut transaction would be returned their tokens. If the swap was successful, then the user making the transaction will receive their agreed tokens, along with any extra tokens that they may have sent to the contract. Gas usage is tracked in this function.

checkTimeOut() - This internal function is used to check if the contract has not been interacted with in 5 minutes. If so, the contract is marked as needing to be reset and any transferred tokens are returned to the users.

reset() - Internal function to reset the variables of the contract. This function also calculates the gas difference between the two users and takes this into account when returning the deposits to the two users.

2 A gas and security analysis of your contract

Deployment: 2455979 gas

First Swap						
	A Transaction	A Execution	A Total	B Transaction	B Execution	B Total
agree	205327	182263	387590	175535	152471	328006
transfer	76741	55469	132210	64314	63042	127356
payOut	92306	86034	178340	117666	111394	229060
total	**374374**	**323766**	**698140**	**377515**	**326907**	**704422**

Other Swaps (including resetting contract)						
	A Transaction	A Execution	A Total	B Transaction	B Execution	B Total
agree	88293	153521	241814	175535	152471	328006
transfer	76741	55469	132210	64314	63042	127356
payOut	77306	71034	148340	102666	96394	199060
total	**242340**	**280024**	**522364**	**362515**	**311907**	**674422**

In my contract I used uint256 data types for int storage as they are more gas efficient than smaller uint data types. All functions that are only called from outside the contract are marked as 'external', as this saves on gas when called[2].

When performing checks within functions, I always used require() instead of assert() as the former is more gas efficient[1].

When compiling my contract, I enabled optimisations to hopefully save gas as well.

The SafeMath library is used throughout my contract to avoid overflow attacks. The timeout feature avoids a user creating a denial of service by abandoning the contract mid swap - the timeout feature also returns all tokens to the users. The contract checks to ensure the two users agree on the token trade, and that they have both sent the correct amount of tokens, before allowing either user to receive the other user's tokens - if there is any disagreements or incorrect transactions then the swap is marked as failed, all tokens are returned, and the users have to start the swap over. When returning the users' deposits, the transfer() function is used to send the ether, helping to prevent reentrancy attacks.

Some risks still remain with the contract. If the contract already has a balance in one of the Token contracts that the tokens are being transferred from, then these tokens will be assumed to be from the user interacting with the contract, meaning that this user may have to transfer less tokens than they agreed to.

3 A detailed description of how your contract ensures fairness

For the fair swap to take place, both players must firstly agree to how many tokens they will each be receiving and sending, along with the address of the token contract that they are transferring from. Then, the user must transfer the specified tokens to the contract and then run a transfer transaction with the contract to check that the tokens have been sent. Once both users have successfully run a transfer transaction, they can both run a payOut transaction to receive their tokens from the other user.

To ensure the two users have a fair swap a few checks and functions are in place. Firstly, the two users must make an agree transaction to ensure they agree on the amount of tokens to be sent an received - if they do not agree, then any tokens that have been sent to this contract by the users is returned to them. Secondly, if a user does not send enough tokens to the contract then the swap is considered as failing and all tokens are returned to the users. Also, if a user accidentally sends the contract too many tokens then these will be returned to them when they payout at the end of the swap.

To ensure gas fairness, both players have to run a payout transaction to receive their tokens. Also, gas consumption of both users is tracked and the user who spent more gas is compensated using the deposits the users made in the agree stage. There is still some unfairness as the gas usage tracking of the reset() function is not accurate, meaning the user who executes this function will be spending more gas overall.

4 The transaction history of a successful fair swap between two players on Ropsten

Contract address: 0x382794CB29f0A7AA593Ad32adD87b4cDfE110A77

4.1 User A

```
{
  "accounts": {
    "account{0}": "0x9Df2b63Bb3678761699f7038320BA06C2f702857"
  },
  "linkReferences": {},
  "transactions": [
    {
      "timestamp": 1610884829656,
      "record": {
        "value": "1000000000000000000",
        "parameters": [
          "5",
          "3",
          "0xd0B9571038CcF21cb9f906F390D851f497507F9A"
        ],
        "to": "created{undefined}",
        "name": "agree",
        "inputs": "(uint256,uint256,address)",
        "type": "function",
        "from": "account{0}"
      }
    },
    {
      "timestamp": 1610884864301,
      "record": {
        "value": "0",
        "parameters": [],
        "to": "created{undefined}",
        "name": "transfer",
        "inputs": "()",
        "type": "function",
        "from": "account{0}"
      }
    },
    {
      "timestamp": 1610884883438,
      "record": {
        "value": "0",
        "parameters": [],
        "to": "created{undefined}",
        "name": "payOut",
        "inputs": "()",
        "type": "function",
        "from": "account{0}"
      }
    },
  ],
  "abis": {}
}
```

4.2 User B

```
{
  "accounts": {
    "account{0}": "0x11aB3c123145d85220345B671Cb220E12564eC69"
  },
  "linkReferences": {},
  "transactions": [
    {
      "timestamp": 1610884851781,
      "record": {
        "value": "1000000000000000000",
        "parameters": [
          "3",
          "5",
          "0x77CE9dF241c394e085F6a51021a56C1D036D824E"
        ],
        "to": "created{undefined}",
        "name": "agree",
        "inputs": "(uint256,uint256,address)",
        "type": "function",
        "from": "account{0}"
      }
    },
    {
      "timestamp": 1610884873316,
      "record": {
        "value": "0",
        "parameters": [],
        "to": "created{undefined}",
        "name": "transfer",
        "inputs": "()",
        "type": "function",
        "from": "account{0}"
      }
    },
    {
      "timestamp": 1610884893723,
      "record": {
        "value": "0",
        "parameters": [],
        "to": "created{undefined}",
        "name": "payOut",
        "inputs": "()",
        "type": "function",
        "from": "account{0}"
      }
    }
  ],
  "abis": {}
```

```
}
```

5 The code of your contract.

```
pragma solidity >=0.4.22 <=0.8.0;
// Import the SafeMath library
import "SafeMath.sol";

contract cw3{

        function buyToken(uint256 amount) external payable returns(bool){}

        function transfer(address recipient, uint256 amount) external returns(bool){}

        function sellToken(uint256 amount) external returns(bool){}

        function changePrice(uint256 price) external payable returns(bool){}

        function getBalance() external view returns(uint256){}

        receive() external payable{}
}

contract cw4{

    // Use SafeMath functions for uint256s
    using SafeMath for uint256;

    // Creator of the contract
    address private owner;

    // Variable used to track the stages each user is at
    uint256 agreeStage = 0;
    uint256 transferStage = 0;
    uint256 payOutStage = 0;

    // True when the transaction has failed and the tokens will return be returned to the users
    bool returnTokens = false;

    // Tracks the last time the contract was interacted with
    uint256 timeOut;

    // Tracks if the contract will be reset on the next interaction
    bool needReset = false;


    // Struct to hold player data
    struct User {
        // User's address
        address payable userAddress;
        // Submitted number of tokens they agree to send
        uint256 tokensToSend;
        // Submitted number of tokens they agree to receive
        uint256 tokensToReceive;
        //// True when the user has agreed their tokens to send and receive
        //bool tokensAgreed;
        // True when the contract has recieved the user's tokens
        bool tokensReceived;
        // Address of the token contract they are transferring their tokens
            from
        address payable tokenContractAddr;
        // Token contract object
        cw3 tokenContract;
        // The current gas the user has used on the contract
        uint256 gasUsage;
    }

    // Array of two Player objects
    User[2] users;

    // Event for when the contract times out
    event TimeOut(string message);
    // Event for when gas is refunded to a user
    event GasRefund(address userAddress, uint256 gasAmount, uint256 gasPrice,
         uint256 userIndex);
    // Event for when the swap has failed
    event TokenReturn(string message);

    // Set the owner to the contract's creator
    constructor() public {
        owner = msg.sender;
    }


    function agree(uint256 tokensToSend, uint256 tokensToReceive, address
        payable tokenContractAddr) external payable returns(bool){

        // Track the gas left at the start of the function
        uint256 gasStart = gasleft();

        // First user who call an agree transaction must reset the contract
        if(needReset){
            reset();
        }


        uint256 userIndex = agreeStage;




        // If the contract has timed out then set the contract to be reset
            and return false
        if(agreeStage > 0){
            if(checkTimeOut(block.timestamp)){
                emit TimeOut("Swap has timed out and has now been reset");
                //reset();
                needReset = true;
                return false;
            }
        }

        // Check if both users have agreed their token amounts
        require(agreeStage < 2, "Tokens have already been agreed");

        // Require the two users to have different addresses
        require(userIndex == 0 || userIndex == 1 && users[0].userAddress != msg.sender, "Users must have different addresses");

        // Require the two users to make a depoit for gas refunds
        require(msg.value == 1*(10**18), "Please make a deposit of exactly 1 Ether");

        //require(agreeStage == 0 || !checkTimeOut(block.timestamp), "Swap reset after 5 minutes of inactivity");

        // Set the current time to track timeouts
        timeOut = block.timestamp;

        // Store the user's information in the User struct
        users[userIndex] = User(msg.sender, tokensToSend, tokensToReceive, false, tokenContractAddr, cw3(tokenContractAddr), 0);

        // Advance the agreeStage
        agreeStage = agreeStage + 1;

        // If the users disagree on the tokens they are sending and recieving then set returnTokens to true so the users will receive any tokens they sent to the contract
        if (users[0].tokensToSend != users[1].tokensToReceive && agreeStage == 2 || users[1].tokensToSend != users[0].tokensToReceive && agreeStage == 2){
            returnTokens = true;
            emit TokenReturn("The swap has failed - run payOut transaction to have tokens returned");
        }

        // Store how much gas the user spent on the contract
        uint256 gasSpent = tx.gasprice.mul(gasStart.sub(gasleft()));
        //gasSpent = gasSpent.add(resetGas);
        users[userIndex].gasUsage = users[userIndex].gasUsage.add(gasSpent);

        return true;

    }

    function transfer() external returns(bool){
        // Check if the contract needs reset
        require(!needReset, "Contract has been reset");

        uint userIndex = 0;

        // Track the gas left at the start of the function
        uint256 gasStart = gasleft();

        // If the contract has timed out then set the contract to be reset and return false
        if(checkTimeOut(block.timestamp)){
            emit TimeOut("Swap has timed out and has now been reset");
            needReset = true;
            return false;
        }

        // Check if the swap failed, the tokens have been agreed by a user and that at least one user has not transferred their tokens yet
        require(!returnTokens, "Swap failed, please run a payOut transaction to return any tokens that you have sent");
        require(agreeStage > 0, "Please agree token exchange first");
        require(transferStage < 2, "Tokens have already been transfered");

        // Set the current time to track timeouts
        timeOut = block.timestamp;

        // Check which user is interacting with the contract
        if(msg.sender == users[0].userAddress){
            userIndex = 0;
        } else if (agreeStage > 1 && msg.sender == users[1].userAddress){
            userIndex = 1;
        } else {
            revert("Contract is currently being used by other users or you have not entered your agreed token exchange");
        }

        // If the user has not transferred enough tokens to the contract then the swap has failed and the tokens of both users will be returned
        if(returnTokens || users[userIndex].tokenContract.getBalance() < users[userIndex].tokensToSend){
            returnTokens = true;
            emit TokenReturn("The swap has failed - run payOut transaction to have tokens returned");
        } else {

            // If the correct amount of tokens have been received then store that the user has sent their tokens and advance the transfer stage

            users[userIndex].tokensReceived = true;

            transferStage = transferStage + 1;

            // If both players have agreed and transferred their tokens then advance to the payout stage
            if(transferStage == 2 && agreeStage == 2){
                payOutStage = 1;
            }
        }

        // Store how much gas the user spent on the contract
        uint256 gasSpent = tx.gasprice.mul(gasStart.sub(gasleft()));
        users[userIndex].gasUsage = users[userIndex].gasUsage.add(gasSpent);

        return true;

    }

    function payOut() external returns(bool){
        // Check if the contract needs reset
        require(!needReset, "Contract has been reset");

        uint userIndex = 0;
        // Track the gas left at the start of the function
        uint256 gasStart = gasleft();

        // Ensure both players have transferred their tokens to the contract or that the tokens are going to be returned
        require(returnTokens || payOutStage > 0, "Waiting for both users to transfer tokens");

        // Check which user is interacting with the contract
        if(checkTimeOut(block.timestamp)){
            emit TimeOut("Swap has timed out and has now been reset");
            needReset = true;
            return false;
        }

        // Set the current time to track timeouts
        timeOut = block.timestamp;

        // Check which user is interacting with the contract
        if(msg.sender == users[0].userAddress){
            userIndex = 0;
        } else if (msg.sender == users[1].userAddress){
            userIndex = 1;
        } else {
            revert("contract is currently being used by other users");
        }

        // If the tokens are to be returned, then return the tokens to the user who is making the transaction
        if(returnTokens){
            users[userIndex].tokenContract.transfer(users[userIndex].userAddress, users[userIndex].tokenContract.getBalance());
        } else {
            // Otherwise send the other user's tokens to the user making the transaction along with any extra tokens the user may have sent to the contract
            users[1 - userIndex].tokenContract.transfer(users[userIndex].userAddress, users[1 - userIndex].tokensToSend);
            users[1 - userIndex].tokenContract.transfer(users[1 - userIndex].userAddress, users[1 - userIndex].tokenContract.getBalance());
        }
        // Increment the payout stage
        payOutStage.add(1);

        // If both users have been paid out or if the contract has a balance of 0 in both of the token contracts, then mark the contract to be reset
        if(payOutStage == 3 || users[0].tokenContract.getBalance() == 0 && users[1].tokenContract.getBalance() == 0){
            needReset = true;
        }

        // Store how much gas the user spent on the contract
        uint256 gasSpent = tx.gasprice.mul(gasStart.sub(gasleft()));
        users[userIndex].gasUsage = users[userIndex].gasUsage.add(gasSpent);

        return true;

    }

    // Function to check if the contract has timed out - if so it will return the tokens to the users from the previous transfer
    function checkTimeOut(uint256 currentTime) internal returns (bool){
        if (currentTime-timeOut > 300){
            users[0].tokenContract.transfer(users[0].userAddress, users[0].tokenContract.getBalance());
            users[1].tokenContract.transfer(users[1].userAddress, users[1].tokenContract.getBalance());
            return true;
        }
        return false;
    }

    // Resets the contract - resets the variables and calculates which user spent more gas and returns the users' deposits with this different taken into account
    function reset() internal {
        uint256 eth = 1*(10**18);
        if(users[0].gasUsage > users[1].gasUsage){
            uint256 gasDif = users[0].gasUsage.sub(users[1].gasUsage);
            users[0].userAddress.transfer(eth.add(gasDif.div(2)));
            users[1].userAddress.transfer(eth.sub(gasDif.div(2)));
            emit GasRefund(users[0].userAddress, gasDif, tx.gasprice, 0);
        } else {
            uint256 gasDif = users[1].gasUsage.sub(users[0].gasUsage);
            users[1].userAddress.transfer(1*(10**18) + gasDif.div(2));
            users[0].userAddress.transfer(1*(10**18) - gasDif.div(2));
            emit GasRefund(users[1].userAddress, gasDif, tx.gasprice, 1);
        }
        delete users;
        agreeStage = 0;
        transferStage = 0;
        payOutStage = 0;
        returnTokens = false;
        needReset = false;
    }

    // Function for contract to receive payments
    receive() external payable{

    }
}
```

6 References

[1] https://medium.com/layerx/how-to-reduce-gas-cost-in-solidity-f2e5321e0395

[2] https://medium.com/better-programming/how-to-write-smart-contracts-that-optimize-gas-spent-on-ethereum-30b5e9c5db85

[3] https://eattheblocks.com/how-to-optimize-gas-cost-in-a-solidity-smart-contract-6-tips/