Blockchains & Distributed Ledgers - Coursework 4

s1519734

April 18, 2025

1 Contract design

1.1 Additional assumptions

1. We assume that users A and B know each other's identities (addresses) on the blockchain.
2. It is assumed that the token contracts C1 and C2 follow the same implementation of the "transfer" function as the one presented in my submission for CW3. This means that the "transfer" function detects whether the recipient is a contract; if it is, then it has to adhere to a certain interface ("IERCiobReceiver"). This means that it needs to implement a "token accepter" function, which will be called by the transfer function.

1.2 Requirements

1. After the swap, either both users receive the tokens they wanted or neither user's balance on the two contracts are reduced.
2. The contract should be as fair and secure as possible.
3. The contract must implement only one fair swap at a time.

1.3 Design

1.3.1 Design choices

My design differentiates between the two swap participants: the one who initiates the swap will be the "initiator" and the participant that accepts the swap will be the "accepter". The initiator wants to swap x of his tokens on contract C1 with y of the accepter's tokens on contract C2.

The C1 and C2 contracts have the same API as the token we had to implement in Assignment 3. This means that there is no way for our Assignment 4 fair swap contract to directly withdraw funds from the participants' contract balances and then transfer them (nor is there a way for our contract to even check the participants' balances). As a direct consequence of this fact, it will be necessary for the participants to first transfer the necessary funds to our contract, prior to the swap being made. This comes with a problem: how do we identify that the swap participants made the necessary transfers to our contract? For this, we assume that the token contracts' transfer functions follow the same implementation as the contract we have submitted for CW3. In practice, this means that when somebody transfers C1 or C2 tokens to our contract, the transfer functions will detect that the recipient is a contract and call its "receiveERCiobTokens" functions, with the sender and the amount as arguments. This will allow our CW4 contract to identify exactly who transferred tokens to it, and also identify the type of token sent (C1 or C2).

With the help of two dictionaries, our contract stores how many tokens of each type each user sent to the contract. The users are free to deposit and withdraw tokens whenever they want (even when they are involved in a swap).

In order to initiate a swap, the initiator must provide the amount of C1 tokens he wants to trade (x), the address of the user he wants to trade with and the amount of C2 tokens he wants to receive in return (y); an event with the details of the swap is also emitted. While the swap is active, nobody else can initiate another swap, which means that the details of the swap remain fixed for this period (the addresses of A and B, and the values of x and y).

Once the initiator has successfully activated the swap, the accepter has a limited period of time in which to accept the swap. He also needs to confirm the details of the swap: the address of the initiator, and the x and y values. If these details match with the details provided by the initiator, then this constitutes an agreement to the terms of the swap. Next, the contract checks whether the accepter's balance of C2 tokens and the initiator's balance of C1 tokens suffice. If they do, then the swap is executed internally: y C2 tokens are added to the initiator's (internal) balance and x C1 tokens are added to the accepter's balance. If the details do not match, or the funds of the two parties are insufficient, then the transaction is reverted and the "accept" fails. Retries are allowed as long as the swap remains active.

Once the window in which the accepter can accept passes, the swap is considered to have failed; the accepter can no longer accept the swap (unless the initiator initiates the swap again).

1.3.2 Contract variables

- **address tokenInit**: Address of C1.
- **address tokenAccept**: Address of C2.
- **address initiator, uint256 initiatorStake**: Address of the the current initiator, and the number of C1 tokens he wants to swap (x).
- **address accepter, uint256 accepterStake**: Analogous to initiator, initiatorStake.

- **acceptTokenBalance, initTokenBalance**: Dictionaries mapping addresses to their balance of C2, C1 tokens (respectively) on the contract.
- **uint256 lockTime**: The time period (in number of blocks) in which the swap remains active. Should allow sufficient time for the accepter to accept the terms of the swap. Default value is 50 blocks (which corresponds to 10 minutes, assuming that 5 blocks are mined per minute).
- **uint256 initiateBlock**: The block in which the last swap was initiated.
- **event SwapInitiated(address initiator, address accepter, uint256 x, uint256 y)**: Event which contains the details of the swap. Emitted once a swap has been initiated; signals that the accepter should accept the swap.
- **event SwapFinished**: Same parameters as above; is emitted once a swap has finished successfully; indicates that a new swap can be initiated.

1.3.3 Interface

The "FairSwap" contract implements the "IERCiobReceiver" interface, which defines a single function, "receiveERCiobTokens"; this allows it to receive tokens from "ERCiobTokens" contracts (such as C1 and C2): they will call this function in transfers in which our contract is the recipient.

1.3.4 Contract functions

- **constructor(address initContract, address acceptContract)**: Constructor of the contract. The arguments are the addresses of C1 and C2, "tokenInit" and "tokenAccept" respectively. It also initialises the "lockTime" variable. Once initialised, these variables cannot be changed (the assignment statement does not require it). The constructor enforces that C1 and C2 are different: if the contracts are the same, it is more efficient to just use the "transfer" function. The constructor also initialises the swap variables ("initiator", "accepter", "initiatorStake", "accepterStake") and the "initiateBlock" variable with dummy values, for gas fairness considerations (so that the first call to "initiateSwap" is not as expensive).
- **receiveERCiobTokens(address sender, uint amount)**: Function belonging to the "IERCiobReceiver" interface. Will be called when somebody calls the "transfer" function of an ERCiobToken contract with this contract as the recipient. The function only accepts tokens from C1 and C2 and will revert the transaction if "msg.sender" is neither of those. When receiving tokens from those two contracts, the function will increment the balance of the "sender" by "amount" (it will increment either "acceptTokenBalance[sender]" or "initTokenBalance[sender]", depending on which of the two contracts called it).
- **initiateSwap(uint256 tokens, address swapPartner, uint256 partnerTokens)**: Initiates a swap, with "msg.sender" as the "initiator", "tokens" as "initiatorStake",

"swapPartner" as "accepter" and "partnerTokens" as "accepterStake". Also sets "initiateBlock" to the current "block.number". The swap can only be initiated if another swap is not currently active (if "block.number" - "initiateBlock" > "lockTime").

- **acceptSwap(uint256 tokens, address swapPartner, uint256 partnerTokens)**: Can be called by the "accepter", within "lockTime" blocks from "initiateBlock" (will revert otherwise). Will check that the details of the swap match: "msg.sender" must be equal to "accepter", "tokens" to "accepterStake", "swapPartner" to "initiator" and "partnerTokens" to "accepterStake" (if they do not match, then the transaction will be reverted). If these details match and the participants' funds suffice, then the swap will be resolved, and the "initiator" and "accepter"'s intra-contract balances will be updated; "initiateBlock" will also be reset to 1 (a non-zero value, for fairness considerations), in order to prevent a double "acceptSwap" attack (more in Section 3).
- **withdrawTokens(address tokenAddress)**: "tokenAddress" must be one of C1 or C2, otherwise the transaction will be reverted. Calling this function will make the contract transfer the remaining balance of "msg.sender" (of "tokenAddress" tokens) to him. The initiator of an active swap can withdraw his C1 tokens from the contract - this is permitted by design. Doing so will make the "acceptSwap" function fail and is essential for fairness (as I perceive it): the accepter can change his mind and never call "acceptSwap" and the initiator can also change his mind and effectively cancel the swap by withdrawing his C1 tokens.

1.3.5 Normal execution

A normal scenario would look like this:

1. The initiator and accepter transfer the tokens they want to swap to the contract.
2. Once no swaps are active, the initiator will initiate the swap with the details he agreed to with the accepter.
3. The accepter will accept the swap within the required time window and will validate the details provided by the initiator. If this validation is successful and the participants have sufficient balances then the swap will be made internally. Another swap can be started from this moment.
4. The accepter and initiator withdraw their funds, ending their swaps.

1.3.6 Special cases

1. The initiator can withdraw his C1 tokens before the "acceptSwap" function is called. Doing so will effectively guarantee that the swap will not go through. The accepter is able to withdraw the C2 tokens he sent to the contract whenever he wants.
2. The accepter can change his mind and never call "acceptSwap". The initiator is able to withdraw the C1 tokens he sent to the contract at a time of his choosing.

A swap will go through if and only if the C1 tokens balance of the initiator and C2 tokens balance of the accepter suffice at the moment "acceptSwap" is called (and the swap is active, naturally).

2 Gas analysis

All the gas costs presented in this section have been estimated using Remix and Solidity version 0.7.4. For each transaction, Remix emits an execution cost and a transaction cost. The execution cost represents the gas cost of running the code executed in the transaction, while the transaction cost = "execution cost + gas cost of sending the necessary data to the blockchain (function arguments) + 21k gas (fixed fee) - any refunded gas" (taken from my CW3 submission). For considerations on gas fairness, only the transaction cost matters (as that represents the fees the users will end up paying in the end).

"Two calls to the same function can incur different gas costs. The main reason for these discrepancies in our contract is variables' state change. In Solidity, changing a (storage) variable from zero to a non-zero value incurs a fixed 20K gas cost. Changing it from a non-zero value to another value incurs a 5K gas fee. "Deleting" a variable (changing its non-zero value to zero) refunds 15k gas cost, with the special mention that the refund cannot exceed half the gas used for the current context. Source: `https://ethereum.github.io/yellowpaper/paper.pdf`." (also taken from my CW3 submission).

Constructor
Transaction cost: 1619725 gas
Execution cost: 1204789 gas.
The constructor sets 8 variables (5 of them for gas fairness reasons, as we will see later), incurring a cost of 160k gas only for that. Also, the deployment cost is directly proportional with the size of the bytecode generated by the EVM from our contract code.

receiveERCiobTokens
The first two sets of costs presented here are the costs of calling the "transfer" functions of the token contracts. As our "FairSwap" contract owns no tokens after deployment, these costs will be inflated by 15K gas (the cost of setting the contract balance to a non-zero value in the token contracts = 20K - the cost of changing a nonzero value to another nonzero value = 5K).

Case 1: Call comes from C1.
Transaction cost: 79657 gas.
Execution cost: 56785 gas.

Case 2: Call comes from C2.
Transaction cost: 79652 gas.
Execution cost: 56780 gas.

Case 3: Call comes from an address who is neither C1 nor C2. The function is called directly in this case, and the transaction will fail.
Transaction cost: 25099 gas.
Execution cost: 2227 gas.

initiateSwap
Transaction cost: 57927 gas
Execution cost: 34863 gas
The transaction gas cost of initiateSwap will remain the same across multiple calls (with the mention that the cost will be roughly 5k gas cheaper per swap field that remains the same, relative to the previous swap).

acceptSwap
Transaction cost: 59097 gas
Execution cost: 66033 gas
The transaction gas cost of this function is dependent on the balances of the initiator and accepter on the "FairSwap" contract. It is assumed that in the general case, before the swap, the balance of C1 tokens of the initiator and the balance of C2 tokens of the accepter will be x and y respectively and that the balance of C1 tokens for the acceptor and C2 tokens for the initiator will be 0 (again, before the swap).
The best case scenario is when before the swap, "initTokenBalance[initiator]" is x (10k gas refunded for updating it inside the swap), "acceptTokenBalance[acceptor]" is y (10k gas refunded), "initTokenBalance[accepter]" is different from 0 (which means that adding x to it will incur a 5k gas cost instead of 20k) and the same for "acceptTokenBalance[initiator]". In this case, the transaction cost will be roughly 30k gas cheaper than the general case presented here.
The worst case scenario is when before the swap, "initTokenBalance[initiator]" is greater than x (requiring 5k gas to change it), "acceptTokenBalance[acceptor]" is greater than y, and "initTokenBalance[accepter]" and "acceptTokenBalance[initiator]" are zero. In this case, the transaction cost will be 30k gas more expensive than the general case presented here.

withdrawTokens
Case 1: msg.sender has some tokens on the contract.
Transaction cost: 36273 gas
Execution cost: 43593 gas

Case 2: msg.sender has no tokens on the contract. Transaction cost: 38673 gas
Execution cost: 15993 gas The cost of this function will stay the same across multiple calls, since it empties the balance of "msg.sender".

3 Security analysis

3.1 Identities of swap participants

The identities of the swap participants are represented by the addresses they use for transferring tokens to the "FairSwap" contract and for initiating or accepting a swap. It is known that "msg.sender" cannot be forged in Solidity. This fact is fundamental to the security of the contract and is used to identify the contracts sending tokens to the contract, the initiators and the accepters and the users wanting to withdraw tokens from the contract. Assuming that the swap participants know each others' addresses (which they do according to the problem statement), a third party cannot assume either participant's identity.

It is necessary that C1 and C2 are trusted. Ideally, their code would be public and verified (on Etherscan for instance). "FairSwap" relies on the fact that the "transfer" functions of those two contracts will not provide fake values for the "sender" and "amount" arguments of "receiveERCiobToken". Providing a higher value for "amount" than the number of tokens sent to "FairSwap" will essentially allow the "sender" to steal tokens from another party (by using the "withdrawTokens" function). If C1 and C2 are not trusted, then this will be clearly be a security vulnerability.

3.2 Integer underflows and overflows

The only arithmetic operations used by the "FairSwap" contract are additions and subtractions. In order to prevent underflows and overflows when performing these operations, we replaced them with SafeMath's equivalents. This way, if an underflow or overflow happens in a transaction, it will be automatically reverted.

By design, an underflow will happen in the "acceptSwap" function if the initiator has withdrawn his C1 tokens prior to the accepter accepting the swap. This will lead to the transaction being reverted by SafeMath, which is the desired outcome.

Using SafeMath for additions may have been overkill. The only way for an addition to overflow in our contract is if a user owns more than $2^{256} - 1$ tokens, which is currently impossible. Assuming a current ether supply of 72M ETH (see `https://etherscan.io/stat/supply`) and knowing from our CW3 implementation that the minimum price of a C1 or C2 token is 1 Wei, a user can own at most $72 * 10^6 * 10^{18} = 72 * 10^{24} = 2^{86}$ tokens (if the contract is deployed on the main net), which is not enough to cause an overflow in our contract. However, if our contract is deployed on another Ethereum blockchain in which the Ether supply is much higher, then the precaution of using SafeMath for additions may become necessary.

3.3 Checks - Effects - Interactions pattern

We employed this pattern in every one of our functions:

- For receiving tokens, we verify that either C1 or C2 are the msg.sender and only then update the balance of the sender.
- For accepting and initiating a swap, we first check that these functions can be called and that the details match in the case of "acceptSwap" and only then update the balances of the initiator and the accepter. In "acceptSwap" we also make sure that the participants' funds suffice before updating their balances.
- For withdrawing tokens, we first check if the provided contract address matches the addresses of either C1 or C2. Then, we update the internal balance of msg.sender and finally send him the tokens. This acts like a safeguard for re-entrancy attacks. May be unnecessary if C1 and C2 are trusted, since "withdraw" uses their "transfer" functions.

3.4 Double accepting a swap

We recognised the hazard that an accepter may accept the same swap twice, which is contrary to the initiator's intent. This is why after a swap, the "initiateBlock" variable is reset to 1. **It is assumed that the current block.number - lockTime is greater than 1**, which is a given if the contract is deployed on a blockchain with at least "lockTime"+1 blocks. Otherwise, this attack will be made possible.

3.5 Griefing by/as the contract initiator

The initiator can always put "0x" as the accepter address (or another address which is not in use), essentially locking the contract for other users, for a period of "lockTime" blocks at a time. We do not have an answer to this which does not punish legitimate initiators as well. However, this attack costs the attacker gas, so he will not be able to block the contract forever. Also, this method of griefing is unreliable: the attacker would need to make sure that his "initiateSwap" calls are always preferred by the blockchain and that a legitimate user's call doesn't pass through. This can be assured by increasing the gas cost of his calls, but will lead to a faster depletion of his funds.

It would be more feasible for an attacker to grief a particular initiator this way, however the fact remains that he can do this only as long as his funds suffice. We consider that this hazard is a consequence of the requirements of this assignment (specifically, of the requirement that our contract implements only one fair swap at a time).

4 Fairness

4.1 Swap fairness

We consider that a swap begins when a participant calls the "initiateSwap" function or when a participant deposits some tokens in the "FairSwap" contract, whichever happens first. We consider that a swap ends when both participants have withdrawn all their tokens from the contract.

Under this interpretation, the basic requirement of this assignment is met. The swap will be successful if and only if both participants have sufficient tokens when "acceptSwap" is called (and the swap details match). Both participants are free to transfer tokens to the contract and withdraw their tokens from the contract at any time. The accepter is free to refuse a swap by not calling the "acceptSwap" function (as he is the only one who can successfully call it). The initiator is free to refuse a swap by withdrawing his funds before "acceptSwap" is called or not sending them in the first place. If any of the participants refuse the swap, the swap simply does not happen and they are free to withdraw their funds from the contract. Thus, their token balance will be the same as before the swap. If the swap is successful, then the participants will have the tokens they wanted after they withdraw their funds (aka after the swap is over).

4.2 Gas fairness

There are two cases to consider:

1. Both parties want to see the swap through.
2. One of the parties changes their mind or acts in bad-faith.

In case 1, perfect gas fairness is impossible. This is because the operations the two parties (the initiator and the accepter) have to perform are asymmetrical: one of the parties will always have to store the details of the swap in the contract and the other party will have to validate them. One of the parties will need to update the internal token balances if the details match.

In our implementation, in case 1, both parties will have to transfer tokens to the contract and withdraw tokens. The costs of these operations will be the same for both parties. The asymmetry in costs for the two parties will come from the "initiateSwap" and "acceptSwap" functions, since they each need to be called by a different party only once. This is why we tried our best to balance the costs of these two functions. Under normal circumstances, calling "acceptSwap" is estimated to cost 2k gas more than calling "initiateSwap". This small difference is to be expected, since the responsibilities of the two functions are different: "initiateSwap" saves the swap details to storage, while "acceptSwap" validates these details and updates the balances of the two participants. We managed to reduce the initial cost (due to storage values initialisations) of "acceptSwap" by initialising the swap variables in the constructor. A further discrepancy between the two functions was resolved by resetting "initiateBlock" to 1 instead of 0 when "acceptSwap" is successful. The cost of "acceptSwap" can fluctuate based on the balances of the participants; the gas cost we presented here corresponds to the most common case (see the "Gas Analysis" section for details), when the participants have exactly the tokens they need for the swap on the contract. Why do we believe this to be the most common case? Because the participants have no tangible benefit if they keep tokens on the "FairSwap" contract and do not trade them.

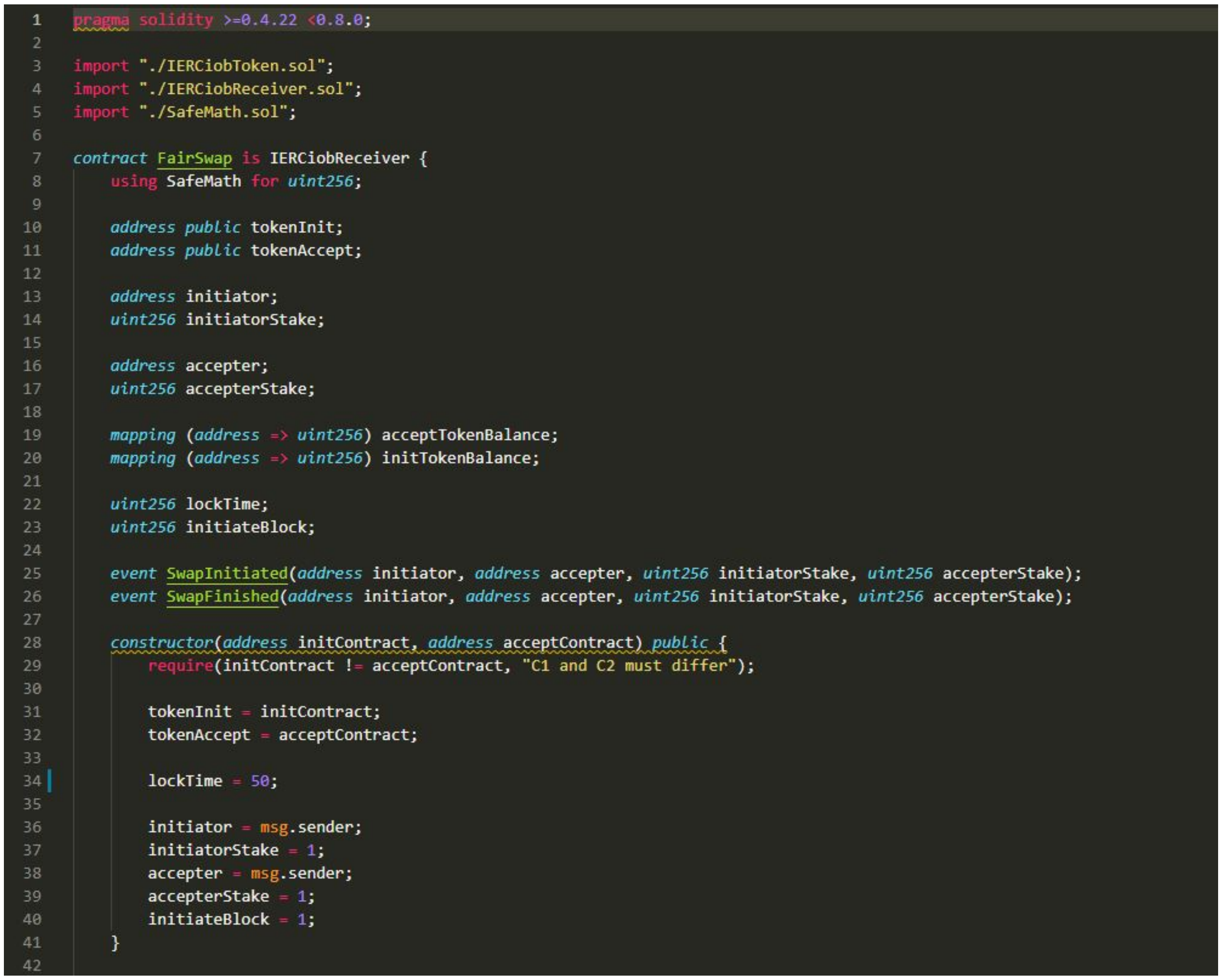
In case 2, gas fairness cannot be ensured. The party who is interested in seeing the swap through is favored to waste more gas than the party who changes its mind or refuses the swap (since the honest party is likely to transfer tokens to the contract and then will have to withdraw them). Also, the initiator stands to waste more gas then the accepter, since he needs to initiate the swap.

5 Transaction history

The necessary contracts were deployed on the course blockchain:
C1 address: 0xDB76e99021B361d326CC792CA56f7562116f8d4e
C2 address: 0xc503191918F9F8DC52db8Eb79DAf70Dc15bE4076
FairSwap address: 0x3C855384F3002a38B7CCB4EAe9fAA125b9EfE381

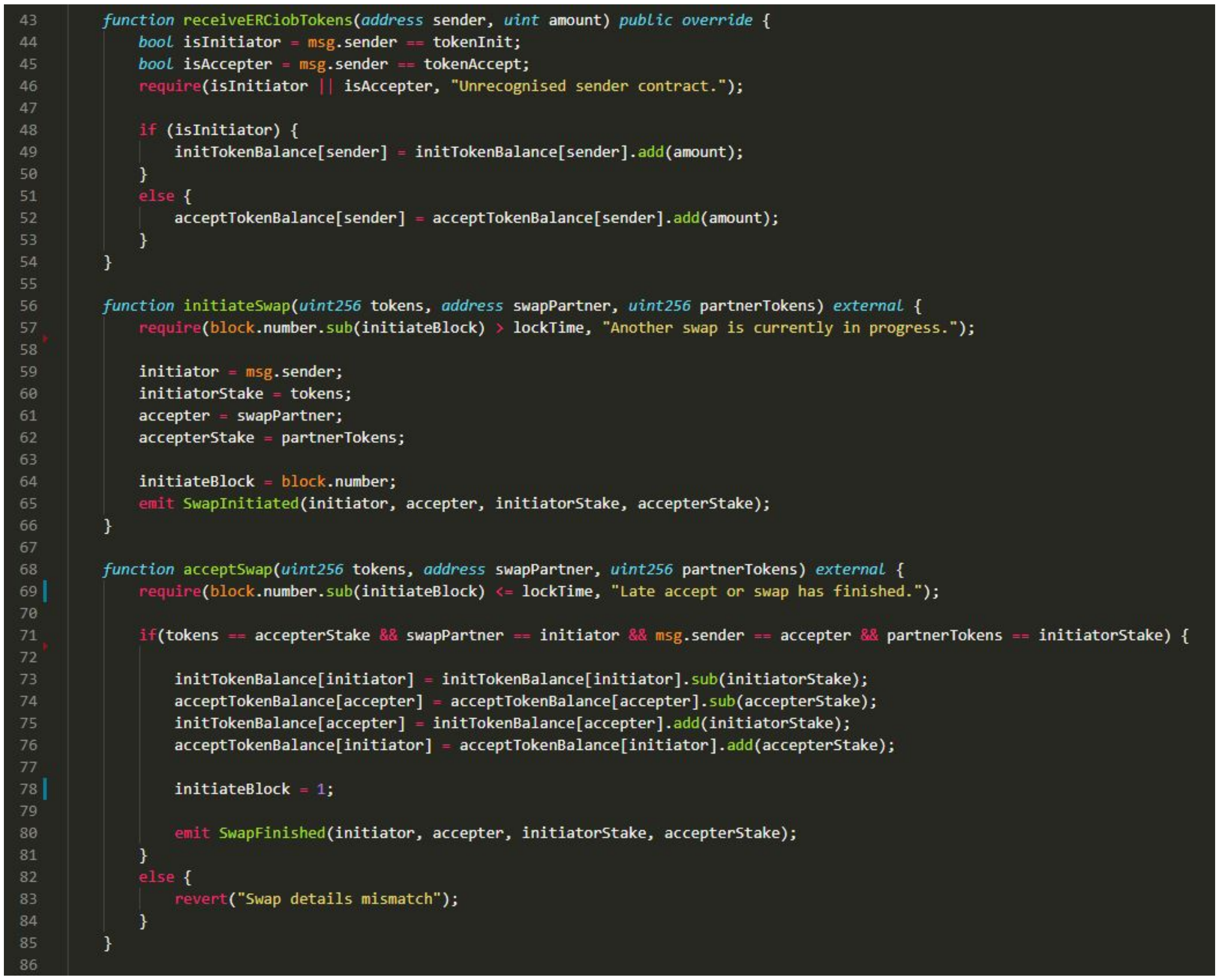
My address: 0x2fda762f105934037f5Aa67827d8733BFa6ae4AE. It will act as both participants. I will swap 2 C1 tokens with 3 C2 tokens using "FairSwap".
Transferring 2 C1 tokens to "FairSwap" (tx hash):
0x1fea02be9af29ec1356032889b4600f6b7fd08aa7ee1ee477680695972206951
Transferring 3 C2 tokens to "FairSwap" (tx hash):
0xec37f1a23fa29737bc22aad97165c5ce6719e3688fb4fb8be1d9a99fb066178b
Initiating swap:
0x5b275bd7ebc9e4f2d80520a6b6fcf47e279b2d6d96530a72e34a0aeaa556cc6a
Accepting swap:
0x46e0a88d5f89b285bc4c73e8e85e1b8f28672f8eb3fe3c346be050a7b5d4644c
Withdrawing C1 tokens:
0x467a56dc7c24e0e64b1cd446349f01c0b1a2e34fa84a854d5a997fa53dc75fa5
Withdrawing C2 tokens (swap is over):
0x7f06a6779a6659f4c79bb055f2555dc3241f2bd34760d27afef12fec6a8124d8

6 Contract code



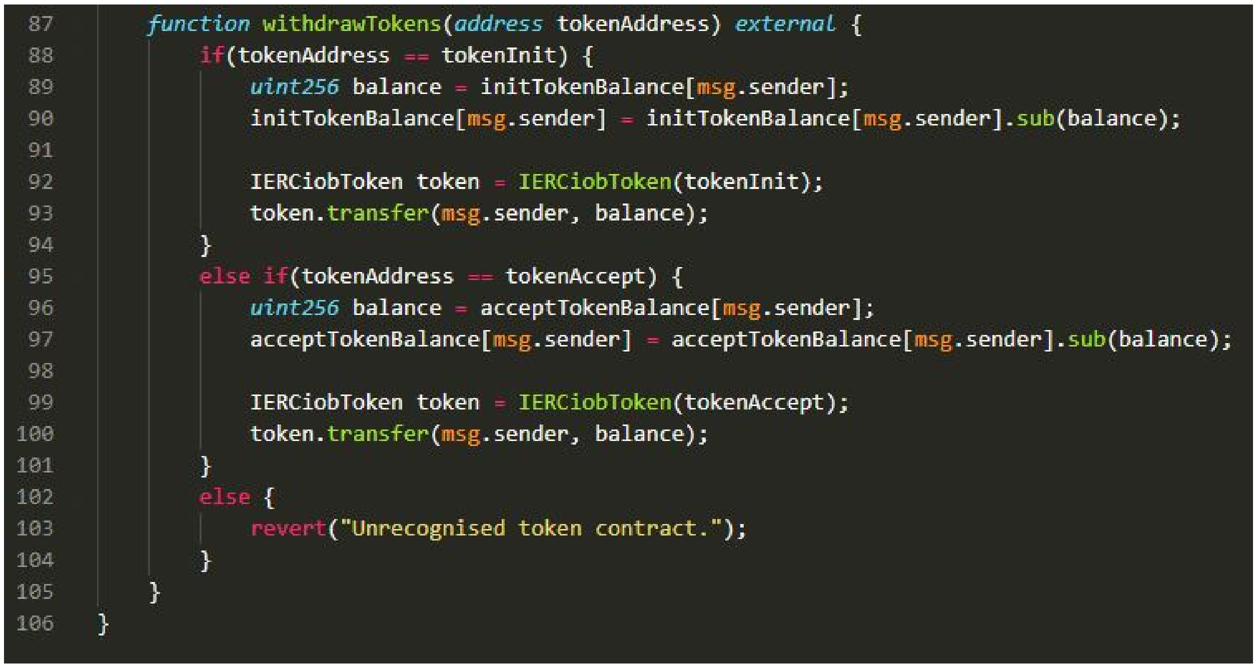
```
pragma solidity >=0.4.22 <0.8.0;

import "./IERCiobToken.sol";
import "./IERCiobReceiver.sol";
import "./SafeMath.sol";

contract FairSwap is IERCiobReceiver {
    using SafeMath for uint256;

    address public tokenInit;
    address public tokenAccept;

    address initiator;
    uint256 initiatorStake;

    address accepter;
    uint256 accepterStake;

    mapping (address => uint256) acceptTokenBalance;
    mapping (address => uint256) initTokenBalance;

    uint256 lockTime;
    uint256 initiateBlock;

    event SwapInitiated(address initiator, address accepter, uint256 initiatorStake, uint256 accepterStake);
    event SwapFinished(address initiator, address accepter, uint256 initiatorStake, uint256 accepterStake);

    constructor(address initContract, address acceptContract) public {
        require(initContract != acceptContract, "C1 and C2 must differ");

        tokenInit = initContract;
        tokenAccept = acceptContract;

        lockTime = 50;

        initiator = msg.sender;
        initiatorStake = 1;
        accepter = msg.sender;
        accepterStake = 1;
        initiateBlock = 1;
    }

```

Figure 1: Contract constructor and storage variables.



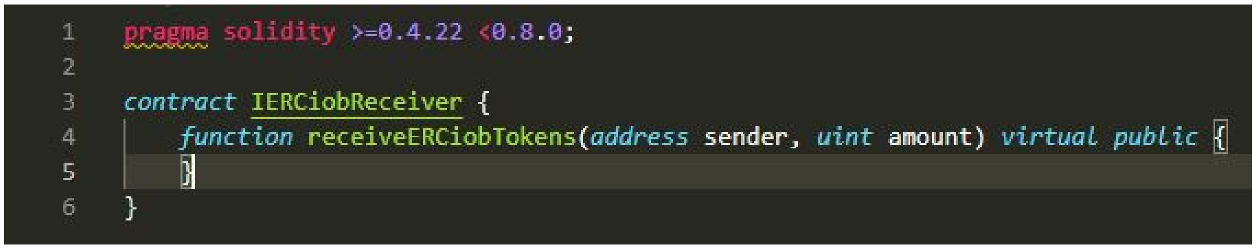
```
    function receiveERCiobTokens(address sender, uint amount) public override {
        bool isInitiator = msg.sender == tokenInit;
        bool isAccepter = msg.sender == tokenAccept;
        require(isInitiator || isAccepter, "Unrecognised sender contract.");

        if (isInitiator) {
            initTokenBalance[sender] = initTokenBalance[sender].add(amount);
        }
        else {
            acceptTokenBalance[sender] = acceptTokenBalance[sender].add(amount);
        }
    }

    function initiateSwap(uint256 tokens, address swapPartner, uint256 partnerTokens) external {
        require(block.number.sub(initiateBlock) > lockTime, "Another swap is currently in progress.");

        initiator = msg.sender;
        initiatorStake = tokens;
        accepter = swapPartner;
        accepterStake = partnerTokens;

        initiateBlock = block.number;
        emit SwapInitiated(initiator, accepter, initiatorStake, accepterStake);
    }

    function acceptSwap(uint256 tokens, address swapPartner, uint256 partnerTokens) external {
        require(block.number.sub(initiateBlock) <= lockTime, "Late accept or swap has finished.");

        if(tokens == accepterStake && swapPartner == initiator && msg.sender == accepter && partnerTokens == initiatorStake) {

            initTokenBalance[initiator] = initTokenBalance[initiator].sub(initiatorStake);
            acceptTokenBalance[accepter] = acceptTokenBalance[accepter].sub(accepterStake);
            initTokenBalance[accepter] = initTokenBalance[accepter].add(initiatorStake);
            acceptTokenBalance[initiator] = acceptTokenBalance[initiator].add(accepterStake);

            initiateBlock = 1;

            emit SwapFinished(initiator, accepter, initiatorStake, accepterStake);
        }
        else {
            revert("Swap details mismatch");
        }
    }
```

Figure 2: ”receiveERCiobTokens”, ”initiateSwap” and ”acceptSwap”



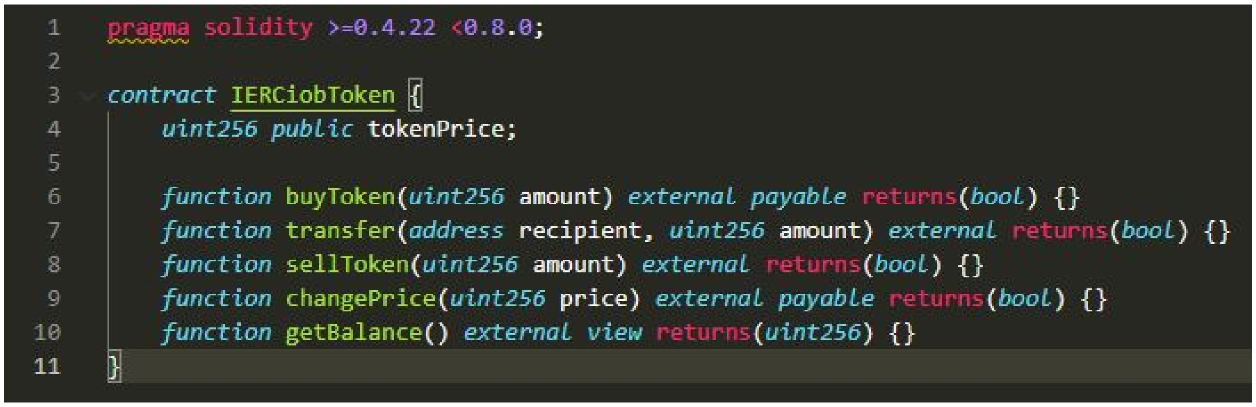
```
    function withdrawTokens(address tokenAddress) external {
        if(tokenAddress == tokenInit) {
            uint256 balance = initTokenBalance[msg.sender];
            initTokenBalance[msg.sender] = initTokenBalance[msg.sender].sub(balance);

            IERCiobToken token = IERCiobToken(tokenInit);
            token.transfer(msg.sender, balance);
        }
        else if(tokenAddress == tokenAccept) {
            uint256 balance = acceptTokenBalance[msg.sender];
            acceptTokenBalance[msg.sender] = acceptTokenBalance[msg.sender].sub(balance);

            IERCiobToken token = IERCiobToken(tokenAccept);
            token.transfer(msg.sender, balance);
        }
        else {
            revert("Unrecognised token contract.");
        }
    }
}
```

Figure 3: ”withdrawTokens”



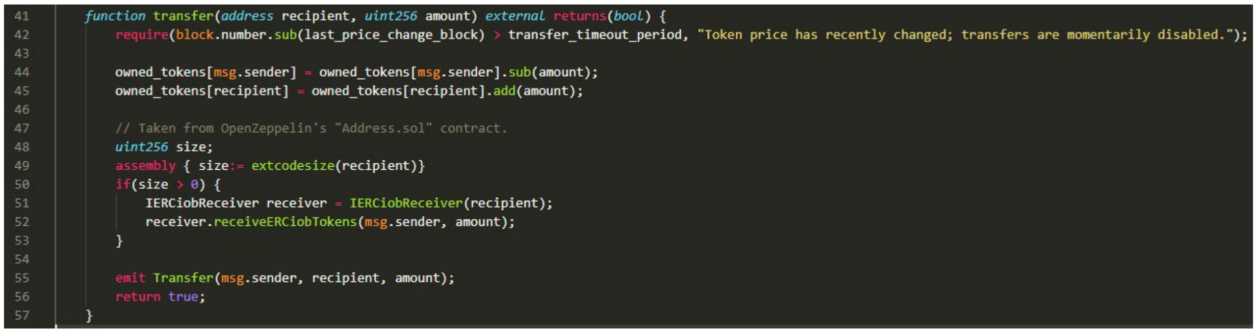
```
pragma solidity >=0.4.22 <0.8.0;

contract IERCiobReceiver {
    function receiveERCiobTokens(address sender, uint amount) virtual public {
    }
}
```

Figure 4: ”IERCiobReceiver” interface



```
pragma solidity >=0.4.22 <0.8.0;

contract IERCiobToken {
    uint256 public tokenPrice;

    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) {}
}
```

Figure 5: ”IERCiobToken” interface



```
function transfer(address recipient, uint256 amount) external returns(bool) {
    require(block.number.sub(last_price_change_block) > transfer_timeout_period, "Token price has recently changed; transfers are momentarily disabled.");

    owned_tokens[msg.sender] = owned_tokens[msg.sender].sub(amount);
    owned_tokens[recipient] = owned_tokens[recipient].add(amount);

    // Taken from OpenZeppelin's "Address.sol" contract.
    uint256 size;
    assembly { size:= extcodesize(recipient)}
    if(size > 0) {
        IERCiobReceiver receiver = IERCiobReceiver(recipient);
        receiver.receiveERCiobTokens(msg.sender, amount);
    }

    emit Transfer(msg.sender, recipient, amount);
    return true;
}
```

Figure 6: My implementation of the ”transfer” function from CW3