Amongus DApp Project

Pietro Bonazzi, Ettore Tancredi Galante, Junwoo Hwang, and Can Inan

Universität Zürich, Rämistrasse 71, 8006 Zürich, Switzerland
https://www.uzh.ch/

Repository:https://github.com/ToreGore/AmongusFinance/

Keywords: Smart Contracts · Tokens · DApp

Abstract. The project here proposed consists of a Decentralized Application (DApp) that has been written with the idea of implementing a simple farming application that can be used by issuing instances of a particular stable token called StableToken to the smart contract in order for a user to receive, every fixed amount of time, some interests in the form of a token called AmogusToken.
This is done via four smart contracts that can be interacted with via a simple front end interface.

1 Main Idea

The main idea of this project is to develop a simple DeFi app where users can depose a stable coin to get in return a different token as interest. The Decentralized Finance (DeFi) movement has been at the leading edge of innovation in the blockchain space. DeFi applications are unique in the sense that they are permissionless. This means that everyone with an internet connection and a supported wallet can interact with them. In addition they are trustless, implying that they generally don't require trust in any middleman like a bank or insurance company.
The concept of deposing cryptocurrencies and getting rewarded is called yield farming or liquidity mining. Yield Farming involves:

- Liquidity providers;
- Liquidity pools.

Liquidity providers provide funds to the liquidity pool, mostly, but not only, in the form of stable coins pegged to a real world currency. Some of the most used stable-coins used in DeFi are the USDT, DAI, BUSD and USDC.
In order to provide liquidity to the liquidity pool, the users need to get new tokens as interest back. In this scenario, the user deposes a mock stable-coin called Stable Token in order to get the Amogus Token in return. In the following chapters is described how said smart contracts are implemented and the user interface acting as front end that allows a smooth interaction between users and said smart contracts.

1.1 Stakeholders

There are two major stakeholders in this model. One is the **Yield Farm**, which is the Smart Contract managing the whole interest and token management. In the demo demo, the first account (index 0) is connected with this contract. The other one is the **Investor**, which is the account that will depose the Stable Token into the Yield Farm, and receive the Amogus Token interest.

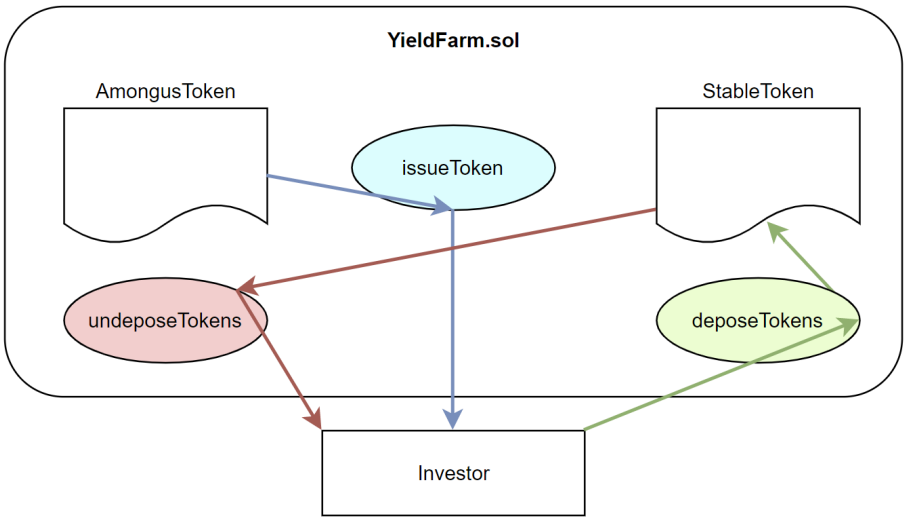

Fig. 1. Stakeholders diagram

In Figure 2, three functions, issueToken, undeposeTokens, deposeTokens, are shown as oval shapes, and the transferring of tokens are depicted as arrows.
In the YieldFarm contract, there are stored both the Amogus Tokens needed to provide an interest to the users participating in the liquidity farm and the amount of Stable Tokens deposited by them. The deposing and undeposing only occurs with the Stable Token (the arrows going to / from the investor). It can be noted that the interests are given in form of a AmogusToken (blue arrow).

2 Smart Contracts: the back end

The three smart contracts that work as back end of the Amogus DApp have been written in the Solidity language and run on top of the Ethereum blockchain, in our setting emulated in a test scenario thanks to the Ganache framework. Following, a brief explanation of the code of the smart contracts implemented.

2.1 StableToken.sol

This is the code representing a stable coin implemented on top of the Ethereum blockchain. In a real case scenario it could be intended as DAI tokens or USDT tokens. There is a total supply equal to $1e24$ units and 18 decimals.
Two events are associated to the contract, a Transfer event and an Approval event. The contract helds two maps, one called "blanceOf", keeping the mapping from address to amount of tokens held, and a map called "allowance."
The transfer function takes as parameters the recipient address and the value to transfer, triggering the Transfer event. It can be seen in Listing 1.1.

```
function transfer(address _to, uint256 _value) public returns
    (bool success) {
        require(balanceOf[msg.sender] >= _value);
        balanceOf[msg.sender] -= _value;
        balanceOf[_to] += _value;
        emit Transfer(msg.sender, _to, _value);
        return true;
    }
```

Listing 1.1. Transfer function

The approve function in Listing 1.2 emits an Approval event after modifying the allowance of the spender.

```
function approve(address _spender, uint256 _value) public
    returns (bool success) {
        allowance[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
        return true;
    }
```

Listing 1.2. Approval function

The transferFrom function in Listing 1.3 emits a Transfer event after checking that the "from" address has enough liquidity and allowance to perform said operation. The balance and allowance of the spending address is decreased and the receiver increases its funds.

```
function transferFrom(address _from, address _to, uint256
    _value) public returns (bool success) {
        require(_value <= balanceOf[_from]);
        require(_value <= allowance[_from][msg.sender]);
        balanceOf[_from] -= _value;
        balanceOf[_to] += _value;
        allowance[_from][msg.sender] -= _value;
        emit Transfer(_from, _to, _value);
        return true;
    }
```

Listing 1.3. TransferFrom function

2.2 AmongusToken.sol

This smart contract implementing works in the same way as StableToken.sol, it has the same event and function but, in the scenario here presented, is meant to represent the token that gets handed to a user that employed its own coins in a farming operation.

2.3 YieldFarm.sol

The smart contract at the core of this project. It holds a public array called "deposers" which holds the addresses of all the users currently taking part into the farming operation. It presents three mappings, one from address to unsigned int representing the deposited balance of an address and the other two from address to boolean value, clarifying if a user has deposited some StableTokens and if it is currently engaged in a farming activity.
The constructor takes as parameters stableTokens, AmongusTokens and the address of the sender which is then set as the owner of the contract. The contract presents three functions:

- deposeTokens
- undeposeTokens
- issueToken

DeposeTokens In order for the function in Listing 1.4 to act effectively, the user obviously needs to depose more than zero stableTokens. The contract then transfers from the address of the sender to its own address the amount of stableTokens specified, which are then added to the deposing balance of the sending user. If the user has not already deposed, then its added to the "deposers" array. Then its status as currently deposing is updated.

```
1  function deposeTokens(uint _amount) public{
2              // To depose, you need to depose more than 0
3
4              require(_amount > 0, "amount cannot be 0");
5
6              // Trasnfer stable tokens to this contract for
                   deposing
7              stableToken.transferFrom(msg.sender, address(this
                   ), _amount);
8
9              // Update deposing balance
10             depositingBalance[msg.sender] = depositingBalance[msg
                   .sender] + _amount;
11
12             // Add user to deposers array *only* if they
                   haven't deposed already
13             if(!hasDeposited[msg.sender]) {
14                 deposers.push(msg.sender);
```

```
            }

            // Update depose status
            isDeposing[msg.sender] = true;
            hasDeposited[msg.sender] = true;
        }
```

Listing 1.4. DeposeTokens function

UndeposeTokens This function in Listing 1.5 fetched the balance in farming and, after checking that the user balance is greater than zero, transfers the stableTokens from the sender to the user balance of the contract. Then it resets the deposing balance and signs the status of the sending address as non deposing.

```
function undeposeTokens() public {
            // Fetch staking balance
            uint user_balance = deposingBalance[msg.sender];

            // Can't have less than 0 staked tokens
            require(user_balance > 0, "Kinda SUS, where is
                your balance?");

            // Transfer stable tokens to this contract for
                staking
            stableToken.transfer(msg.sender, user_balance);

            // Reset staking balance
            deposingBalance[msg.sender] = 0;

            // Update staking status
            isDeposing[msg.sender] = false;
        }
```

Listing 1.5. UndeposeTokens function

IssueToken The function in Listing 1.6 can be called only by the contract's owner. It works by calculating the total amount of the deposited balance among all users in order to calculate an interest of what a single user should receive. This is given by the formula:

$$\frac{balance}{overallBalance} \times \frac{balance}{10}$$

where the amount of tokens issued to a user are equal to the tenth part of the ratio between the squared balance over the total deposited balance. This formula, since there a calculation of the Annual Percentage Yield has not been implemented for complexity reasons, allows for a user to earn at most 10% of the amount of Stable Tokens deployed to the Yield Farm, in case it is the only actor in the farming pool.

```
function issueToken() public {
            //only owner can call this function
            require(msg.sender == owner, "caller must be the
                owner");

            uint total_deposited_balance = 0;

            //calculate total amount of deoposited balance of
                 all users together
            for(uint i=0; i < deposers.length; i++) {
                total_deposited_balance =
                    total_deposited_balance + deposingBalance
                    [deposers[i]];
            }

            // send intrest tokens (amonugsTokens) to the
                users that deposited stable tokens
            for (uint i = 0; i < deposers.length; i++){
                address recipient = deposers[i];
                uint balance = deposingBalance[recipient];

                //calculate intrest to send to the different
                    users
                uint intrest = ((balance /
                    total_deposited_balance) * balance ) /
                    10;

                if(balance >0){
                amongusToken.transfer(recipient, intrest);
                }
            }
        }
```

Listing 1.6. issueToken function

3 Front end

The front end has been implemented using ReactJS and the styled component library. Once the web page has been loaded, users are able, just by clicking the buttons on the right, to invest and divest their tokens. In Figure 2 the user has a balance of 100 stable tokens and he is about to depose 50 of these tokens into the liquidity pool.

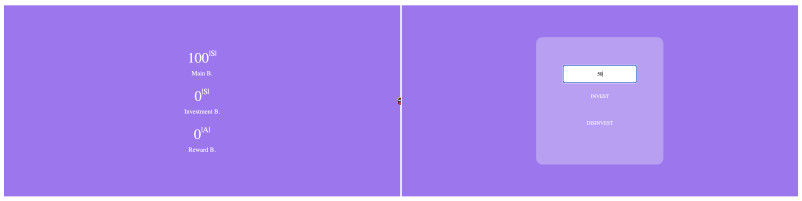

Fig. 2. FrontEnd GUI View not invested

In Figure 3 you can see that the user has invested 50 stable tokens into the application and the main balance of stable tokens got reduced and the investment balance got increased by 50 tokens.

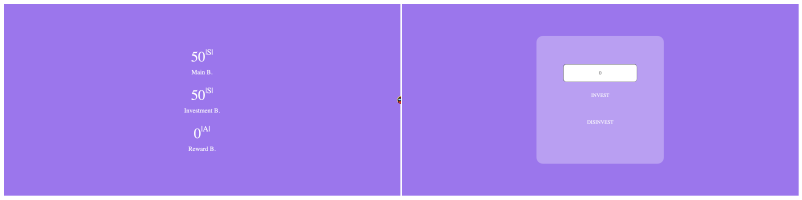

Fig. 3. FrontEnd GUI View invested

After a certain time, the user gets rewarded in form of the new Amongus Token. In Figure 4 one can see that the reward balance increased by 5 Amongus tokens.

3.1 Functions and Workflow

From the page called index.html, the body tag containing the div with id = root is invoked. ReactJS is loaded inside this script tag using ReactDOM, which handles routes in a web app. The workflow of ReactJS starts in index.js. Here the App component is imported from App.js and invoked.
The async function "componentDidMount()" is the first function invoked just before mounting occurs. It is called before render(). In itself, it first waits for

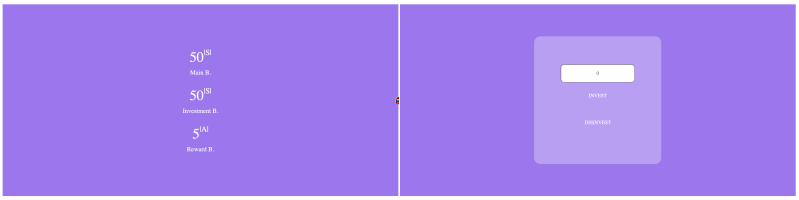

Fig. 4. FrontEnd GUI View invested

execution of loadweb3(), which loads web3, and then for loadBlockchaindata(). The latter function loads the data from the network for the specific account ID. Especially loads the balances amounts of the three tokens defined in section 2. Once these pieces of information are collected, the loading state of the App component is set to false and the Main components are rendered.

```
1     async loadBlockchainData() {
2
3         //open a window in web 3 that can communicate with
              the Ethereum network
4         const web3 = window.web3
5
6         //get the accounts from the Ethereum network
7         const accounts = await web3.eth.getAccounts()
8
9         //insert them in the component's state
10        this.setState({ account: accounts[0] })
11
12        //insert them in the component's state
13        const networkId = await web3.eth.net.getId()
14
15        //load stableToken
16        //get a reference to the contract in the network
17        const stableTokenData = StableToken.networks[
              networkId]
18        if(stableTokenData) {
19
20            //create a new contract object with the same json
                  interface of the respective smart contract
21            //this allows us to interact with smart contracts
                  as if they were JavaScript objects.
22            const stableToken = new web3.eth.Contract(
                  StableToken.abi, stableTokenData.address)
23
24            //pass the new object to the component state
25            this.setState({ stableToken })
26
```

```
27              //load the balance of the the account using the
                    method "balanceOf" defined in the SC.
28              let stableTokenBalance = await stableToken.
                    methods.balanceOf(this.state.account).call()
29              this.setState({ stableTokenBalance:
                    stableTokenBalance.toString() })
30
31          } else {
32              window.alert('StableToken contract not deployed
                    to detected network.')
33          }
34
35          // Same procedure for Amongus Token
36          const amongusTokenData = AmogusToken.networks[
                networkId]
37          if(amongusTokenData) {
38              const amongusToken = new web3.eth.Contract(
                    AmogusToken.abi, amongusTokenData.address)
39              this.setState({ amongusToken })
40              let amongusTokenBalance = await amongusToken.
                    methods.balanceOf(this.state.account).call()
41              this.setState({ amongusTokenBalance:
                    amongusTokenBalance.toString() })
42          } else {
43              window.alert('AmongusToken contract not deployed
                    to detected network.')
44          }
45
46          // TokenFarm
47          const tokenFarmData = YieldFarm.networks[networkId]
48          if(tokenFarmData) {
49              const tokenFarm = new web3.eth.Contract(YieldFarm
                    .abi, tokenFarmData.address)
50              this.setState({ tokenFarm })
51              let deposingBalance = await tokenFarm.methods.
                    deposingBalance(this.state.account).call()
52              this.setState({ deposingBalance: deposingBalance.
                    toString() })
53          } else {
54              window.alert('YieldFarm contract not deployed to
                    detected network.')
55          }
56
57          this.setState({ loading: false })
58      }
```

4 Conclusion

In this paper, as well as in the implementation, a proof of concept for a Yield Farming DeFi Application has been implemented. The users are allowed to invest their mock Stable Tokens and get rewarded with an interest in the form of the here implemented Amongus Tokens.
Future developments should in first place rework the equation to calculate the rewarded tokens in order to allow the employment of the Annual Percentage Yield or, even better, a variation capable of reflecting the volatile nature of cryptocurrencies, such as a Weekly or Daily Percentage Yield. It should also be implemented an internal Ethereum Wallet before an eventual market launch.
Last but not least, for this DeFi application to reach a much broader audience, different stable-coins should be accepted and achieve some sort of interoperability on blockchain and tokens.