Peer to Peer Systems and Blockchains
Final Project: TRY DApp

Matteo D'Onofrio, 561901

22/06/2022

1 DApp Structure

The TRY DApp is composed by a Front-end and a Back-end. The Back-end is the business logic code, implemented through a set of smart contracts written in Solidity; the Front-end is a web interface that allows the user to interact with the Application.

The folder provided, containing the whole application, is structured as follows:

- *contracts*: a folder containing three smart contracts;
- *migration*: a folder containing the file *1_initial_migration.js*, used to migrate the Dapp on the running Blockchain.
- *src*: a folder containing the whole Front-end. We can find *index.html* as the unique *HTML* web page; in addition there are two more folders inside: *src/css* containing the file *Style.css* for the page style and the folder *src/js* containing the file *app.js*, which implements all the functions that run in order to allow the web page to be interactive.
- *test*: a folder containing the file *Lottery.js*, a short code provided to run a simple test of the smart contracts functionalities.

In the following we'll describe firstly the Back-end showing some differences with respect to the Final Term, then the Front-end and finally we'll show the DApp interface functionalities.

2 Back-End

The Back-end is implemented by three smart contracts: *Collectibles.sol*, *newNFT.sol* and *Lottery.sol*.

The smart contract *Collectibles.sol* is exactly the same of the Final Term. It defines the structure of each collectible: an integer $id \in [1, 8]$ and a short string describing it; for example the collectible with id = 1 has a description

"*Hello_Kitty_1*". This smart contract has a function *buy_collectible*() that allows the Lottery Manager (Operator) to buy all the collectibles he need.
The smart contract *newNFT.sol* is the one used to mint and transfer the ownership of a NFT. This is a derived contract but differently from the Final Term, here the contract inherits the base contracts named *Ownable.sol* and *ERC721.sol* [link at: [1]], which are two of the most used *superclasses* inherited for handling the NFTs. This choice was done since the contract *ERC721.sol* allow a simpler and more clear managment of the NFTs. In the *newNFT.sol* contract, we use a struct called *NFT_info* which allow to associate to each NFT minted these information: the address of the owner *nft_owner*, the integer *token_id* that uniquely identifies an NFT, the string *description_collectible* which contains the description of the collectible used to mint it and finally the integer *class* which store the class of minted NFT.

About the methods, the *newNFT.sol* contract implements the following: *mintToken*() which in turn invokes the $super-class$ method *_safeMint* in order to mint a new NFT; *give_to_winner*() which in turn invokes the $super-class$ method *_transfer* in order to transfer a NFT ownership. Other functions are defined in order to get some specific NFT information.

The most relevant is contract *Lottery.sol*, the one that handles the whole lottery.

The main difference with respect to the Final Term lies in the controls that are done in order to guarantee the correct flow execution of the DApp. In the Final Term we used a set of boolean variables, one for each different phase of the lottery, whose value set to true or false define the ongoing phase; in addition all the modifiers were based on these variables. In the Final Project instead all these boolean variables are replaced by a single string called *lottery_phase_operator*. The value of that string changes among: "*Not_created*", "*Created*", "*Init_phase*", "*Buy_phase*", "*Extraction_phase*", "*Compute_prizes_phase*" and "*Give_prizes_phase*". Accordingly all the function modifiers that check for the correct execution of the DApp, do checks on this string value. This choice was done since simplify both the functions complexity and the readability of the code, in addition allow to use fewer variables while obtaining the same result.

About the *Lottery* contract attributes, the most important are the following:

- *collectibles_bought*: a mapping ($uint8 \implies string$) associating for each collectible *id* its respective string containing the description. Used to store the collectibles bought by the lottery manager (operator).
- *NFT_Minted*: an array containing the integer *token_id* of each NFT minted so far, used to keep track of all the NFT minted during the lottery.
- *drawn_numbers*: an array of 6 integers containing the numbers drawn randomly in each round of the lottery.

[1] https://docs.openzeppelin.com/contracts/4.x/api/token/erc721

- *Ticket*: a struct used to define a ticket bought from a user; each ticket has the attributes: address *owner* and a set of 6 *chosen_numbers*.
- *tickets_sold*: an array of struct *Ticket* containing all the tickets bought from the users in a single round.

The main functions that implements the business logic of the lottery are the following and are all invoked by the lottery manager (operator).

- *create_lottery*(): used to create the lottery and notify the event to all the users;
- *set_balance_receiver*(): used in order to provide the account address to which send the smart contract balance at the end of each round.
- *buy_collectible*();
- *start_new_round*():
- *draw_numbers*(): used in order to extract the 6 numbers randomly.
- *compute_prizes*(): used in order to compare the drawn numbers with the numbers provided by the users within the ticket, based on this check the winners will be identified and their prizes with them;
- *give_prizes*(): used in order to assign the prizes to the users; in this phase either a NFT ownership is transferred from the operator to the winner user (in case the set of 8 initially minted NFTs are not assigned yet) or a new NFT is minted.
- *close_lottery*();

The only function that a user can invoke is *buy_ticket*();

The price for both a collectible and a ticket is fixed to 1 ether. The length of a round is defined by the variable M, set by default to 5 which means that a round can have at most 5 tickets bought.

3 Front-End

The Front-end is composed by the Web page that acts as an interface that allows the Lottery Operator and the Users to interact with the smart contracts. This web page is developed using *HTML*, *CSS*, *JavaScript*, *JQuery*, *Bootstrap* and *Web3.js*.
The *HTML* web page is unique but inside it contains two different interfaces, one for the Lottery Operator and one for the Users. Based on which account address is connected to MetaMask, the page shows a different interface with the help of *JavaScript* and *JQuery* functions.

Lottery Operator and Users interact with the web page through a set of buttons and input text fields. The web page takes the inputs provided and invoke the smart contracts functions (both for transactions and simple get_data functions). Each event generated by the smart contracts is notified to the web page through the *Events*. The events generated are the following:

- *lottery_created*;
- *balance_receiver_initialized*;
- *collectible_bought*;
- *ticket_bought*;
- *phase_change*;
- *values_drawn*;
- *prize_assigned*;
- *NFT_minted_now*;
- *NFT_transfered*;
- *lottery_closed*;

4 How to Run the DAPP

To run the DApp you need first to install the following:

- *nodeJS*(version 12 or more) (sudo apt install nodejs) and *npm* (sudo apt install npm) on the computer. After that you need to install all the dependencies of *npm* used to create a $lite - server$;
- npm init (in the project folder);
- npm install –save lite-server (in the project folder);
- npm install –save web3 (in the project folder);
- npm install –save @truffle/contract (in the project folder);
- install *Truffle* which will be used to compile the smart contracts, to make test on them and finally to migrate/deploy the smart contracts on a running blockchain ((sudo) npm install -g truffle);
- finally Metamask must be installed on the browser.

Now in order to execute the DApp run the following commands:

- run ganache-cli -l 100000000 -m "fly harvest slam field nuclear leave decorate manage easily neither garlic venue" (in the project folder). This command start the blockchain, creating 10 account addresses each with 100 ether. The parameter $-l$ allow to set a specific $gasLimit$, the parameter $-m$ allow to define the mnemonic word which allow to get always the same set of addresses.

- now on the command line you can see the set of 10 accounts created, each account has its own $private-key$. Since for the DApp we need at least 4 accounts, we can take the $private-keys$ of the first 4 accounts and import them on Metamask, choosing as test net the $localhost8545$. If the account were already used, is needed to reset them going to: Metamask settings, advanced, reset account. This will reset the $nonce$ of all the account to 0.

- run truffle migrate –network development –reset (in the project folder) will compile and migrate/deploy the smart contracts on the blockchain. This command will automatically set the first account of the provided set as a Lottery Operator.

- run npm run dev (in the project folder). Will start the $lite - server$ running on the browser at $http : //localhost : 3000/$.