Final Project

Corso P2P System & Blockchain

Davide Morucci - mat. 548058

1 Introduction

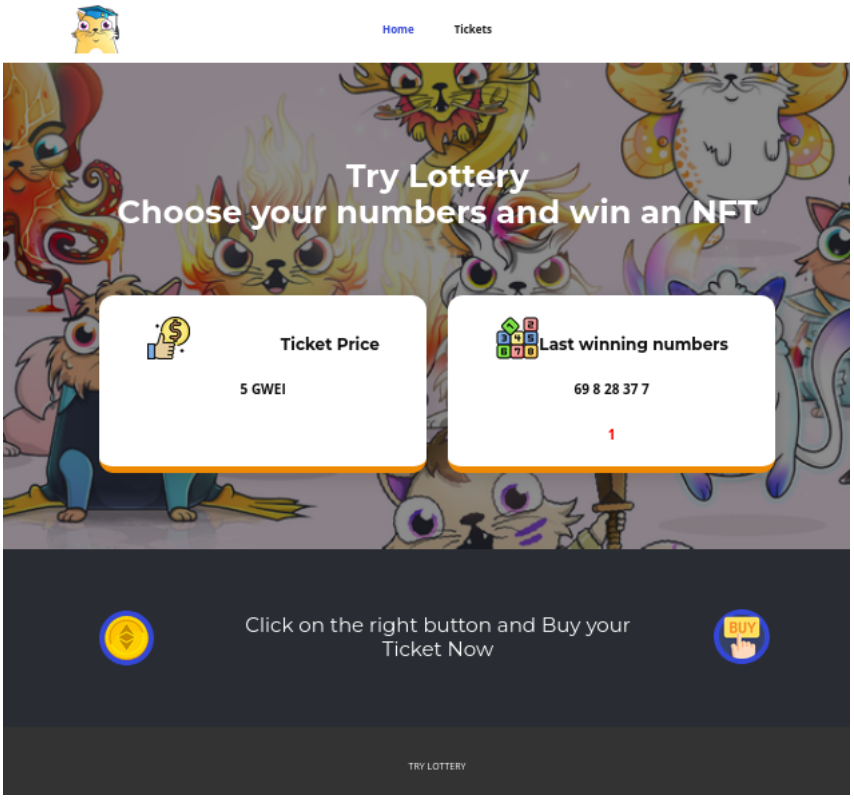
What is presented below is the Dapp implementation of the Try Lottery smart contract, implemented for the final project of the P2P System and Blockchain course.

1.1 General Structure

The Dapp developed basically consists of three different interfaces that are used to manage all the tasks provided by the lottery. In particular, these three interfaces are called *Home*, *Tickets* e *Manager*.

1.1.1 Home

The Home page is the main view of the Dapp, that is used to present the application to the users, giving some usefull information about the lottery. In this page we can see the ticket price for the current lottery, we have a link that will bring the user to **Account page**, where tickets can be bought, and also, this is the page where the extracted numbers will appear when the lottery operator calls the **drawNumbers** function: such numbers will be visibible after the extraction and until a new round or a new lottery is started.



1.1.2 Tickets

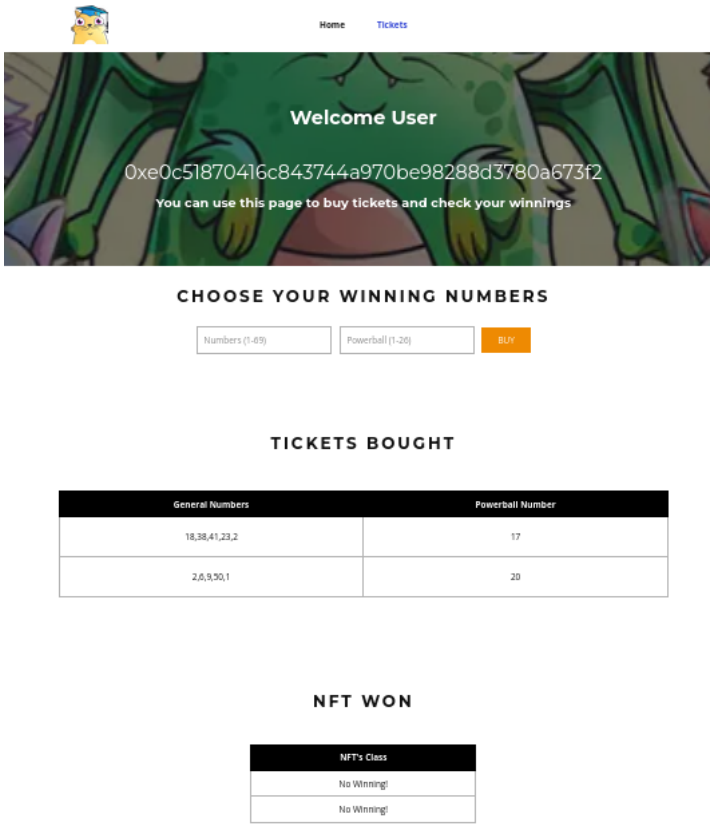
The tickets page represents a sort of user account page (we are not talking about a real account, since there is no form of login or similar). What the Dapp does in this case is to retrieve the user's status using the address from which this is currently connected (ex. *the Metamask account*).

On this page the user can buy a new ticket, can view those already purchased and finally also see the NFTs won up to that moment.

To be more precise:

- The form on the page allows the user to buy a new ticket: it checks live if the numbers played by the user are valid (both the normal numbers and the powerball) and in case of an error, notifies the user that the ticket is not valid. This control at the frontend level avoids any interaction with the backend and with the smart contract, in case of error. If instead the numbers are correct, the **BUY** function of the smart contract is called. If everything is ok, the user receives feedback informing him that the purchase was successful: this can be done by listening to the **newTicketBought** backend event, which informs the user that the purchase functions were executed with success. If something goes wrong in the backend, it will be caught in the frontend to warn the user. For example when the lottery is closed and the user tries to buy a new ticket, an alert will suggest to the user that he can buy a new ticket only at the start of a new round.

- The list of won NFTs is updated using the description associated with the new NFT (for semplicity, just the class of the NFT). The list of NFTs won is designed to last even when a new round or lottery begins; this is possible because, as described below, the backend has been modified and the **nftKitty** class (the Solidity class that represents the NFTs) which is detached from the single instance of the Lottery, contains an address mapping of the won NFTs, which is then used to display NFTs won on this page.
- There is a list of tickets bought by the user that is updated at every new purchased ticket, and it is designed to persist for the entire duration of a round.



1.1.3 Manager

This page is reserved for the *lottery manager*. As mentioned earlier, when the front end of the app loads, it loads a certain type of information or not, based on the account (address) from which the user is connecting. If the account is recognized as the lottery manager, the link to this page will appear in the nav bar of the interface (for a common user this will not be visible). This page is used by the lottery manager to interact with the lottery and in particular to call up a whole series of functions present in the back end. For example, the lottery manager can *start a new round, draw numbers or close the lottery.* More importantly, the lottery manager has the ability to create a new lottery

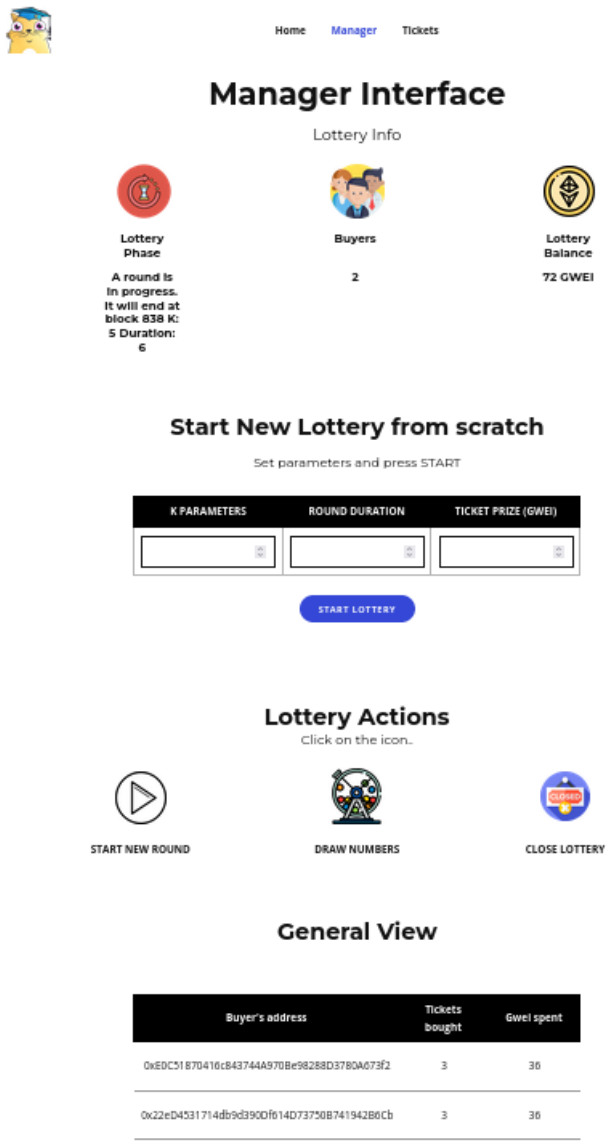
(how, will be discussed later), defining some important parameters such as the **number of rounds** of the lottery, the **parameter K** used for the randomness during the drawing and the new **ticket price**.

Also the lottery manager within this page has the ability to view some important statistics related to the lottery. Particularly:

- It is shown the **current lottery balance** retrieved from backend.
- There is a **Status** section that indicates the current status of the lottery (e.g. if the lottery is active or if a round is started or if winning numbers have been extracted...).
- A **Buyers** section indicates the total numbers of buyers for this round of the lottery.
- A final section where the addresses of all users who have bought a ticket to participate in the current round are indicated, the total number of tickets that each user has bought and the total gwei that each user has spent in this round. This information remains present until the round ends.

Also, whenever the lottery manager interacts with the lottery:

- When a new lottery is started, two transactions are generated to first end the previous lottery (this is because only one lottery can be there at a time), then a second transaction is sent to create a new lottery instance, which is then loaded from the frontend. When a new lottery starts, an event is issued by the backend in order to notify users.
- When a new round starts, all connected users will receive an alert notifying the start of a new round.
- When it decide to extract winning numbers or close the lottery, a notification through alert is sent to connected users.



1.1.4 General information

- Management functionalities like *startNewLottery, startNewRound, drawNumbers, closeLottery* can be called only via the Manager Dapp interface, available only to the lottery manager. Normal users cannot access this page.
- Dapp has been developed with *JS and Web3Js, HTML and CSS,* using some tools that allow an easier management of HTML and CSS files. In particular the Home and Tickets page are managed by the *app.js* file and the Manager interface by the *lotteryManager.js* file.
- Each JS files embed an App object used to initialize Web3, initialize the

contracts (as explained later), the event listener and finally to render the page and its elements.

2 Backend Structure

The implementation of the Dapp required several changes to the original smart contract project. Due to the fact that it is necessary to istantiate, on demand, a new istance of the Lottery smart contract, and not rely on a single istance of the Lottery contract, a **button** has been added which allows you to instantiate a new lottery.

The approach that was used for the development of the Dapp is called **Factory Pattern**.

The main idea of this pattern is to have a contract, hypothetically called "factory", which is responsible for creating other contracts.

In other words we can say that the *Factory contract* is used to create and deploy child contracts.

Why this pattern? The main motivation for this pattern in class-based programming comes from the single-responsibility principle (a class does not need to know how to create instances of other classes), and this pattern provides an abstraction for the constructor.

This type of design pattern is very useful in Soldity and Dapp development: for example if we want to create multiple instances of the same contract (as in this case), this design pattern helps us to keep track of these "child" contracts(in our case, the new *Lottery* istance) and makes it easier their management.

There are different types of Factory Patterns: in this case the one used is called **Normal Factory Pattern**, where simply, the contract called *LotteryFactory* (our "factory"), is implemented with a function that manages the creation of a child contract, by means of the use of the *new* keyword.

The *newLottery()* function in the *Factory* contract takes care of what has just been said, in particular by instantiating a new *Lottery contract*, with the parameters defined by the lottery operator (so round duration, K and price). In order to fetch the new lottery from the frontend, we use another function, *at()*, provided by the *TruffleContract factory*: this function allows to fetch a contract thanks to its address, so that the frontend can report the new instance created.

In our case it is assumed that only and only one Lottery can be active at a time.

Another major improvement concerns the **nftKitty** contract used to manage NFTs.

Thanks to the improvements introduced with the factory schema, it has been revisited and is now totally detached from the *Lottery* and *Factory* contracts, but above all, it has been implemented to be able to work with a configurable Lottery instance. This was obviously necessary due to the changes to which the Lottery contract is subject during its execution, in particular due also to the fact that the *nftKitty* contract can no longer assume a static address for the lottery. On the other hand it is then necessary to configure a new instance of the lottery within the *nftKitty* contract, in order to maintain the specification that only the lottery can invoke the functions of the *nftKitty* contract. Whenever a new *Lottery* is deployed, it receives the address of the *nftKitty* contract to

allow communication, and at the same time, the *nftKitty* contract must also be configured to allow the same thing: all this is done by the *Factory* which communicates to the contract *nftKitty* the new lottery address.

However, the factory pattern has some problems in terms of cost and gas spent, which in normal cases is high. In our case, given the simplicity and low complexity of our implementation, the extra costs in terms of gas that are added, compared to the solution with a single contract, can be neglected. From this point of view, various functions in the backend have been reworked and optimized, in order to use little gas anyway.

3 Configuration

Project has been developed using Truffle and Ganache, and has been tested in Ubuntu 22.10. The code is contained in the zip file associated to this submission. Smart contracts are in the contracts/ folder and frontend code is in src/ folder.