TRY

Antonio Pace

April 2023

1 Introduction

In the Section 2, are described the two smart contracts that implement TRY, a blockchain based NFT lottery. In the Section 3 there is a brief description of Cryptoducks, the collectibles given as a prize in form of NFTs for the lottery. In the Section 4 there is a description of the dApp for the TRY lottery.

2 Smart Contracts

For the implementation of TRY, I created two smart contracts: one for the NFT and one for the lottery.

2.1 Cryptoducks.sol

The NFT contract is written starting from `https://github.com/nibbstack/erc721/blob/master/src/contracts/tokens/nf-token-metadata.sol` that is an implementation of ERC-721 standard, with methods for mint, transfer, set metadata and check the ownership of NFTs.

Cryptoducks.sol is a subclass of this implementation, with only two methods: one for minting the NFT linking a metadata to it, and another for transfer the NFT from one account to another one, checking that the operation is allowed.

2.2 lottery.sol

lottery.sol is the implementation of the lottery logic. It contains all the methods required by the assignment, plus getters for some useful values and some auxiliary methods:

- **random(int range, int seed)**: (*private*) generate a random number from 1 to *range*.
- **checkDuplicates(int a, int[] b, int i)**: (*pure, private*) checks if integer *a* is present in the array *b[1..i]*. I need it for checking that the random winning numbers generated are all different.
- **findCommonElements(int [] c, int [] d)**: (*pure, private*) finds common elements between arrays *c* and *d*.
- **withdraw(address addr)**: (*public*), withdraws the revenue of the lottery sending it to *addr*. Can be called only by the lottery manager.
- **mintOnDemand(address to, uint tokenId, string uri)**: (*private*) for minting an NFT when necessary, i.e if the NFTs minted at the beginning of the lottery are already assigned. The difference from **mintDKS()** is that here the token ids are not stored for later assignment but just minted on the fly.
- **transferDKS(address to, uint tokenId)**: (*private*) for transfer an NFT from one address (the manager that minted the NFTs at the beginning) to another one.

2.3 Implementative choices

Players' address and tickets are stored in an array of *struct Player*, that is cleared every end of a round.

For the NFTs, there is a mapping between the classes and the NFTs (*struct DKStoken*) minted at the beginning of the lottery, that gives the informations about the ID, the URI and if the NFT is already assigned. This is important for decide if transfer the existing NFT or minting a new one on the fly, and for knowing the metadata URI for every class. Another mapping is used for linking users and their prizes.

There is a variable for counting the ID of the tokens that have to be minted on the fly. So it's very important minting the NFTs always progressing in an increasing way.

The length M of each round is set at the the creation of the lottery.

There are three round states: *active* when users can buy tickets, *unactive* when M blocks are passed and *finished* when the prizes are given and it's possible to start a new round. When the lottery is closed, players are refunded if necessary (round not finished), and a boolean variable is set at false (*lotteryUp*), blocking the possibility to start a new round of the lottery.

A round should be active for M blocks. I check for this in the **buy(...)** function, deactivating the round if M blocks are passed. I tested the application in local, so there is one block for every transaction and there are only blocks relative to the lottery. In a deployment over a blockchain, there are many different blocks, so it can happens that the function doesn't detect immediately the M blocks passed. Moreover, if nobody buys the ticket, the check doesn't take place.

2.4 Execution of TRY

I will present some steps for a simulation of the lottery:

- Deployment of Cryptoducks.sol.
- Deployment of lottery.sol, the constructor take as arguments the address of Cryptoducks contract (for calling mint and transfer).
- The lottery manager creates the lottery deciding the duration (in number of blocks), and the first time that this happen, the first 8 NFTs (one for each class) are minted and assigned to himself.
- The lottery manager calls **startNewRound()**. The round is now active.
- Now, players can buy a ticket for 610'000 gwei (as tx value) calling **buy(...)**, choosing 5 different numbers from 1 to 69 and one powerball from 1 to 26 (the last one). An event shows the ticket.
- When M blocks are passed, the round is not active anymore, and the lottery manager can call **drawNumbers()**. An event shows the 6 winning numbers.

- The lottery manager can now call **givePrizes()** for distribute the prizes to the winning players, according to the guessed numbers. An event show the address of the winner, the tokenId and the class of the prize. The first time that someone win a prize of a class c, receives the one previously minted. Then following prizes are minted on the fly. The round is now finished and the data structure containing the tickets is cleared.

- The lottery manager can **withdraw(...)** the revenues of the tickets on an address of his choice (when round is finished).

- The lottery manager can now starts a new round.

- The lottery manager can close the lottery when he want. If the round is not finished, players are refunded.

2.5 Gas

Surely, the transaction that cost more gas is **givePrizes()** if the tickets are more than a couple. That's because the function has to check for every ticket, how many number the player has guessed, if he guessed the powerball and which prize should be assigned, and that means nested loops. So for a big number of tickets the gas price is high, while the other function's cost are almost constant. With 10 tickets, Remix says that the function call costs 420787 gas, that is almost the double of the **drawNumbers()** cost (212863 gas), which has to generate 6 random numbers, checking that they are different.

2.6 RNG

For extracting the winning numbers, there is a need for a random sequence. I used an hash of the timestamp of the block with an index for changing the sequence. Due to the deterministic nature of the method, this is a pseudo-random generator. For a true random number generator we may need an oracle. In fact, knowing these information, everyone can reconstruct the sequence. However, the winning numbers are generated when is not possible anymore to buy the tickets, so it's impossible to predict in advance the right sequence. The real problem can be the manipulation of the block by miners, but this is an issue only if the prize of the lottery is valuable enough.

3 Cryptoducks

Cryptoducks are a batch of collectibles, designed specifically for this assignment (special thanks to Lisa Queirolo). In Figure 1, you can see a Cryptoduck for every class. These collectibles are intended to be uploaded on IPFS or a standard image-hosting website and set as metadata for the NFT prizes of TRY.

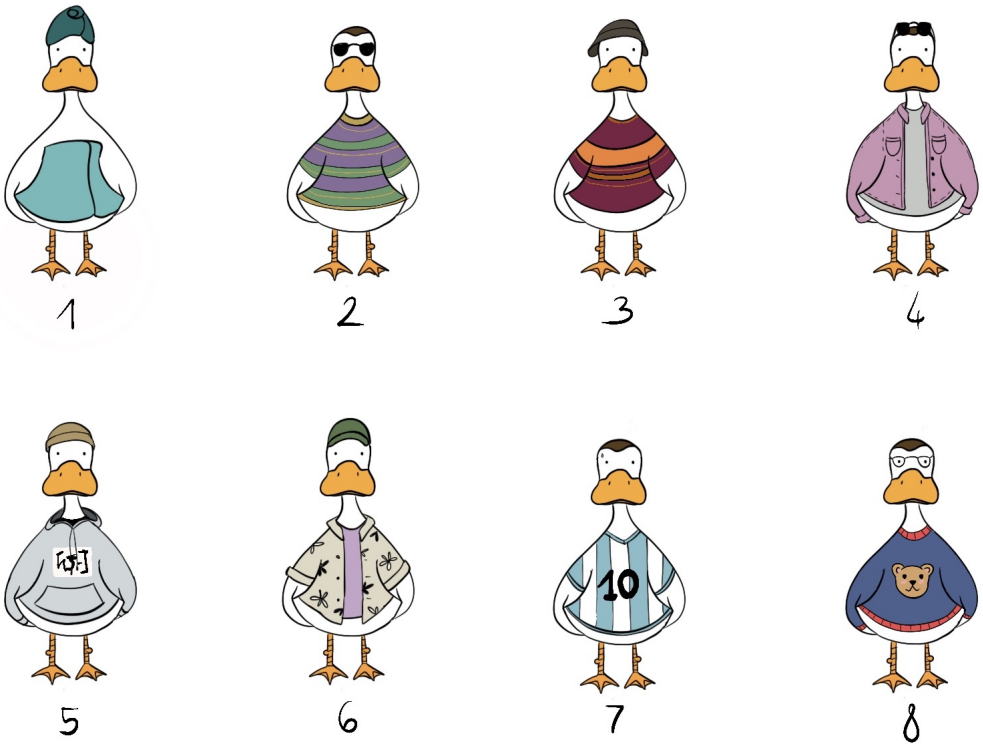

Figure 1: Cryptoducks

4 dApp

The TRY dApp is made up by two components: the blockchain-based backend implemented through the smart contracts presented before and a web-based frontend written in web3 Javascript (app.js). I deployed the smart contracts on a local blockchain on Ganache using Truffle, managing the accounts that interact with the dApp with Metamask.

4.1 Implementation

After initializating web3 and the contracts in the dApp, the event listener is started, sending an alert for:

- Creation/termination of the lottery
- Change on round's state
- Emission of the winning ticket
- Prize won by a player
- Revenues of the lottery after the withdraw

The first three events are seen only by players, the prize won is shown also to the manager and the winner player has a personalized alert with the tokenID of the prize, while the last event is shown only to the manager.

Most of the functions are for rendering information about the player and the lottery with getter methods of the contract (like round state, number of ticket sold...) or just for calling the lottery methods.

Two functions are a bit more particular, *getTickets()* and *renderPrize()*, one for getting the list of tickets of a player, and the other one for rendering the list of prizes won by that player. They call a first lottery method for retrieving the number of items, and then loop for that number of times retrieving the i-th item each iteration. This is due to the fact that we can only retrieve a fixed-size array from the contract. The flow of execution of TRY dApp is the same described in 2.4.

4.2 Interface

The dApp present one interface for the manager and one for the players. In both of them, the following values retrieved from the contract are shown: the state of the lottery, the state of the current round, the number of tickets sold, and the winning numbers.

4.2.1 Manager interface

The lottery manager can create the lottery in the homepage deciding the duration.

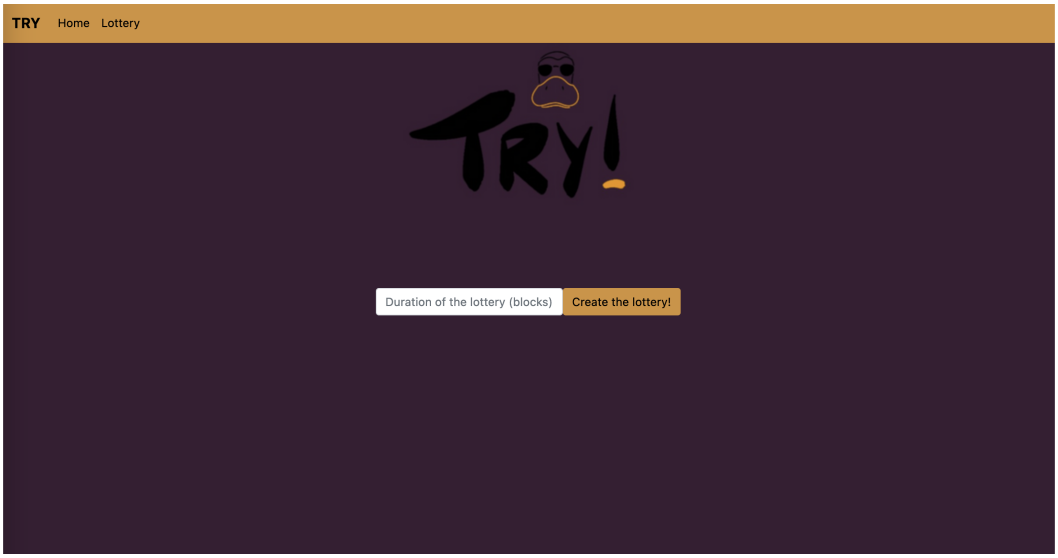

Figure 2: Manager homepage

In the lottery page there are 5 different function buttons for: starting a new round, drawing the winning numbers, giving the prizes discovering the winning players, closing the lottery and withdrawing the revenues.

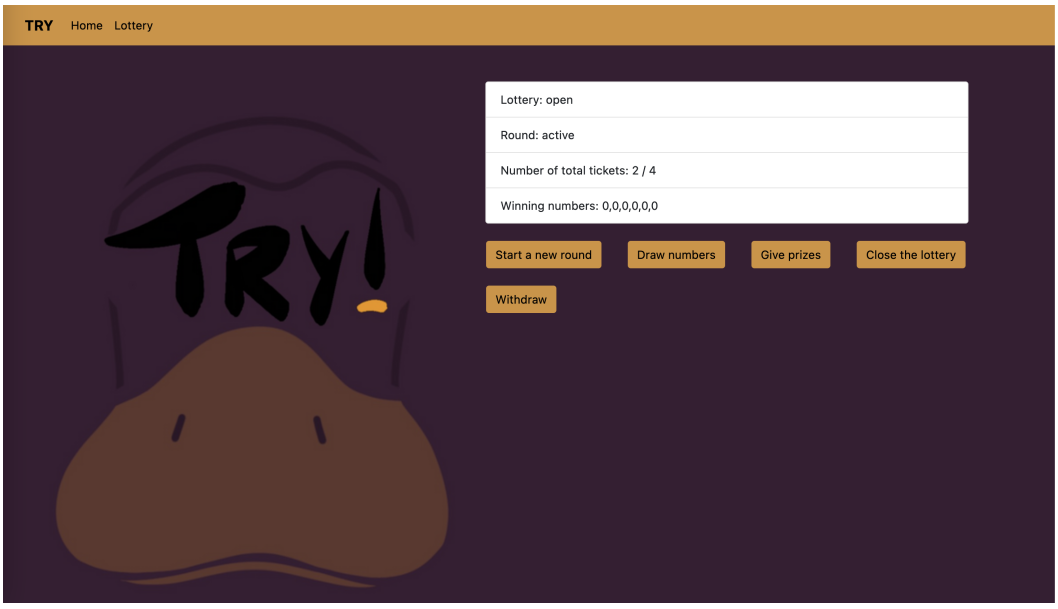

Figure 3: Manager lottery page

4.2.2 Players interface

The players, instead, can see the bar for buying a ticket and the list of all his tickets. The prizes that a player won, are shown above, one after the other. In the homepage the player is just informed if the lottery is open or closed.

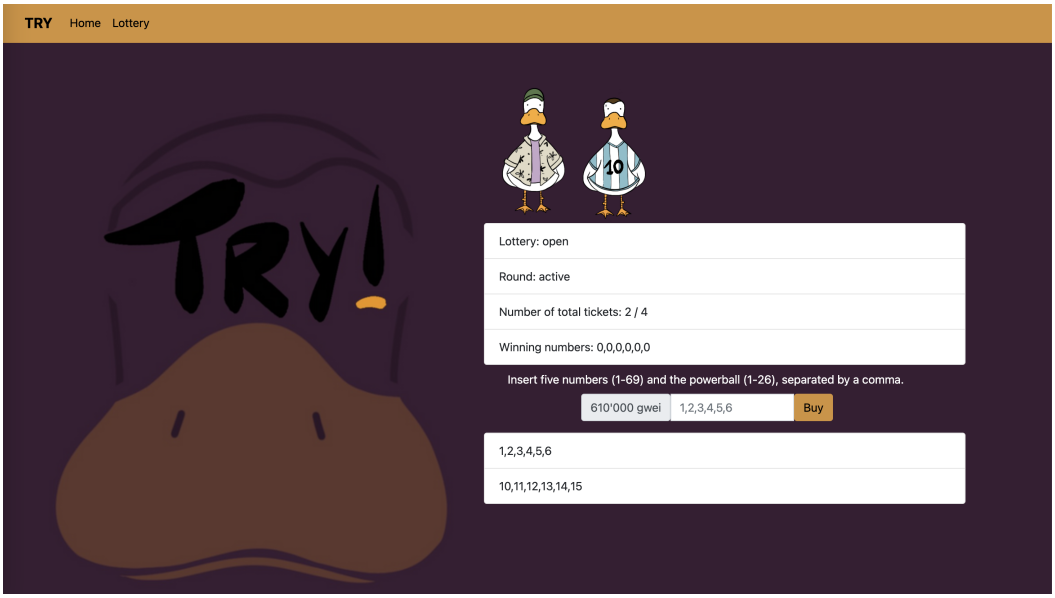

Figure 4: Player lottery page

4.3 Run the dApp

For running this project, you need:

- NodeJS
- Truffle
- Connection to an Ethereum network (Ganache, Infura...)
- Metamask

First, compile and migrate the contracts on a ethereum network (indicate target network in *truffle-config.js*) executing *truffle migrate –reset –network development*. I used a local network created with Ganache on localhost:8545. Import accounts on Metamask. Then run the dApp with *npm run dev* for running on localhost:3000.