TRY: a nfT lotteRY

Midterm for Peer to Peer & Blockchain 21-22

Chenxiang Zhang
Department of Computer Science
University of Pisa
c.zhang4@studenti.unipi.it

1 Introduction

In this didactic blockchain project, we create a simple *Animal Lottery* where the game's mechanics resembles the classic Powerball lottery game. Briefly the lottery has multiple rounds, where each round allows users to buy tickets containing 5 whiteball numbers and a powerball number. The objective is to guess the numbers by matching as many numbers as possible with the winning numbers presented at the end of each round. Each user that matches at least one number will be rewarded with a unique NFT, whose rarity increases with the number of matched numbers.

2 Lottery Architecture

We present the Animal Lottery, whose architecture is composed by two contracts interacting among each other: *Lottery* and *NFTAnimal*.

2.1 NFTAnimal

The NFTAnimal inherits from the standard ERC721 contract. We define each NFT with a triple of *(id, class, metadata)*, where *id* represents the unique token's identifier, the *class* indicates the rarity class of the token, and the *metadata* is a textual description representing the token. The NFTAnimal contract provides two main functions that are used by the Lottery contract during its execution:

- *mint.* This function is the core of the NFT application. It mints a new NFTs of a specific class rarity directly to the address of the contract's owner. Since each NFT is represented by a unique identifier, we use a simple *uint* variable abstraction saved in the storage and incremented every time the *mint* function is called to represent the ID of a NFT.
- *transfer.* This function is used by the lottery whenever there's a winner and we need to transfer one NFT of a specified class rarity to the winner's address. The choice of which NFT of the specific class will be transferred is arbitrary. Since we store the NFTs in an array, for simplicity we transfer the last element of the array. If the NFT of that specific class is not available, then we can call the *mint* function to create a new one and transfer it.

The NFTAnimal contract implements additional utility functions that are helpful to generate random numbers and random metadata descriptions. The generation of random numbers relies on the *keccak256* hashing of three elements (seed, block.number, block.timestamp), where the seed is a user-parameter value. This randomness solution uses the blockchain itself as the source of randomness and can be subject to vulnerabilities (discussed in Subsection 3.2). The creation of random metadata simply picks and concatenates few random elements from different array of strings. Some example of the generated metadata are: (White Bear), (Yellow Bird), (Black Shark)...

2.2 Lottery

The Lottery represents the main contract of the system. It presents different interfaces allowing the admin (operator) to manage the state of game. All the states of the game and their relationship is represented in the Figure 1. We describe the main states of the Lottery:

- *Inactive (Prize Assigned).* The first state when deploying a new contract. During this phase the round is not started and the previous round's prize have already been assigned. The only allowed main operation is to *startNewRound* which transitions the state to *Active*.
- *Active.* During this phase the round is active and it will last for K blocks (K = ROUND_DURATION) before transitioning automatically to the next state. This state allow users to buy the tickets.
- *Inactive (End Purchase).* In this phase it, users are not able to buy tickets anymore. The only allowed main operation is to *drawNumbers* which draws the winning numbers of this round and transitions the state to the next state.
- *Inactive (Winning Numbers Drawn).* This is the last phase of a complete round cycle, the winning numbers are drawn and the operator has to assign the prizes to the winners.

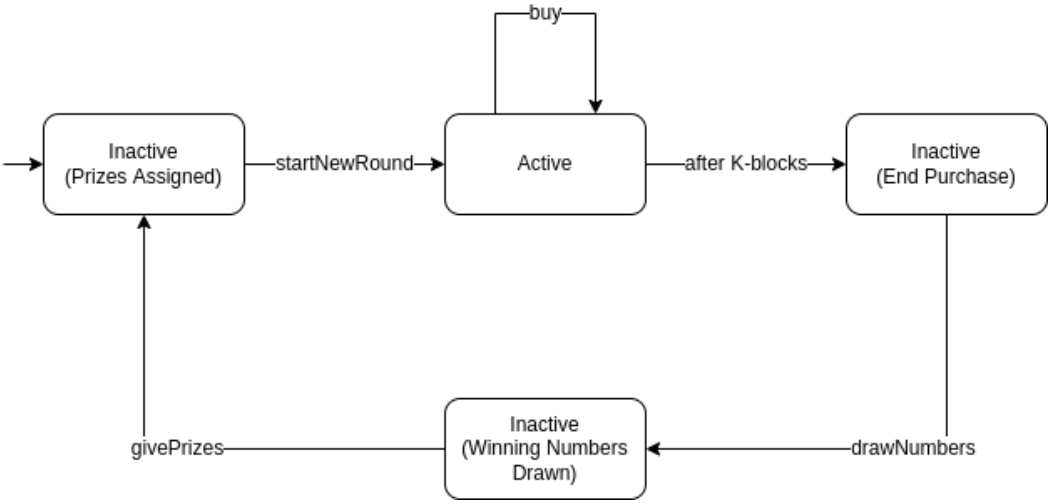

Figure 1: Lottery's state machine representing the different states of the lottery system and the transition relationships among the states.

An overview of all the main Lottery's operation in relationship with the NFTAnimal contract can be seen in the Figure 2. The operator (admin) can interact with all the functions in the Lottery contract, which in turn calls the functions in the NFTAnimal contract, which inherites the standard ERC721 contract. In order to understand more the logic behind each state transition operation, we describe in detail the main interfaces and their exact functioning:

- *startNewRound.* This operation is usable only by the admin and when the game state is in *Inactive (Winning Numbers Drawn)*. It allows the admin to start a new round of game. It resets the states of a game such as the winning ticket numbers of the previous round to zero, and sets the startingBlock to the current global variable *block.number* to keep track of the number of blocks elapsed from the start of the round.
- *buy.* This operation is usable only when the lottery is in the state *Active*. It allows a user to buy a ticket by paying the ticket price and inserting the five whiteballs and one powerball numbers, which are validated through different checks such as: range validity check and

duplicates check. The ticket price is also checked, and if any of the checks fail, the state is reverted.

- *drawNumbers.* This operations is usable only by the admin and when the lottery is in the state *Inactive (End Purchase)*. It draws the winning ticket numbers using the random generator function described in the Subsection 2.1. We ensure the absence of whiteballs duplicates by changing the seeds of the random generator when a duplicate is found and redrawing a new number. Furthermore, this function can only be called once per round, preventing the misbehaviour from the admin to re-sample the winning numbers.

- *givePrize.* This operations is usable only by the admin and when the lottery is in the state *Inactive (Winning Numbers Drawn)*. The lottery stores all the tickets purchased in a round during the *Active* phase. We compare each ticket with the winning ticket and assign the prizes accordingly to the provided rules of the game. For every winner, we transfer the NFT prize to it. If there's no prize of that rarity class left, we mint a new NFT of that class and transfer it ensuring that the winning prize will always be distributed.

- *mint.* This operations is usable only by the admin. It allows to mint a new NFT of a specific class that will be used as a prize for the lottery.

- *closeLottery.* This operations is usable only by the admin. It deactivates the lottery and can be used during any state of the round. Its behaviour depends in the state of the game: if the lottery state is not *Inactive (Prize Assigned).* then the lottery has to refund the ticket price to all the current round's players before deactivating the lottery.

At the creation of the Lottery contract, its constructor will create a NFTAnimal contract, which will initially mint one NFT for each class rarity. For all the presented main operations, we notify the outcome of the operations by emitting a proper event. If an operation fails due to an unsatisfied requirement, then the state is reverted to the initial state using the *require* keyword.

The Lottery contract implements additional utility *getters* functions that helps the admin to visualize the state of the round such as, visualizing the number of purchased tickets, the number of total available NFTs, and other useful debugging informations.

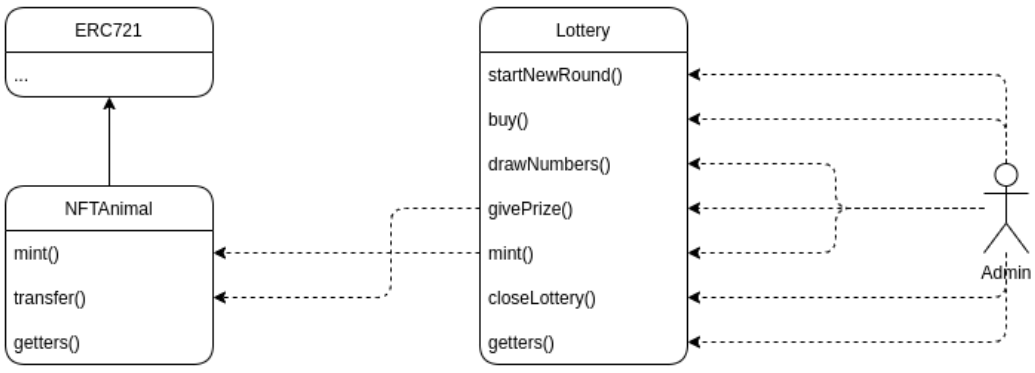

Figure 2: Animal Lottery's complete architecture. NFTAnimal inherites from ERC721, while Lottery creates an instance of NFTAnimal and use its interfaces.

3 Miscellaneous

3.1 Gas Estimation

We estimate the Gas cost of two non trivial functions: *startNewRound* in the Table 1 and *buy* in the Table 2. We do so by estimating approximately the Gas cost of each operation that make up the function. We divide the whole set of operations into categories:

- *Global* = {msg.sender, block.number...} Indicating the global variables accesses
- *Comparisons* = {GT, SL, ...}
- *Simple Arithm.* = {ADD, SUB, GT, LT, AND...}
- *Complex Arithm.* = {MUL, DIV, MOD...}
- *Logging* is a one time cost for each logging event
- *Logging Data* is a cost paid for each byte containing in the logging event

Operation	**Times**	**Gas**
Global	3	2
Simple Arithm.	5	3
Complex Arithm.	0	5
Memory Load / Store	5	3
Storage Load	6	200
Storage Store	6	20.000
Logging	1	375
Logging Data	47	8

Table 1: *startNewRound* function Gas estimation. Since we are resetting the states of the round, we have to access to many storage variables. We can observe that the cost mainly is derived from the Storage Store operations. The Storage Store operations makes up for almost all the cost of the function.

Operation	**Times**	**Gas**
Global	3	2
Simple Arithm.	48	3
Complex Arithm.	0	5
Memory Load / Store	70	3
Storage Load	6	200
Storage Store	5	20.000
Logging	1	375
Logging Data	26	8

Table 2: *buy* function Gas estimation. We can observe that also in this function the majority of the cost is taken by storing the data on the storage memory, specifically here we update the current round's array of tickets by adding the new bought ticket utilizing various *Storage Store* operations: one for adding the struct element (4 SSTORE) and one implicit for updating the dynamic array's length (1 SSTORE).

3.2 Security Issues

We highlight the most common vulnerability and how we handle them in the Animal Lottery.

- *Re-entrancy.* This vulnerability is avoided by using the .transfer() when sending currency in the *closeLottery* in case we have to reimburse the players
- *Arithmetic flow.* This vulnerability is avoided by the new releases of Solidity v0.8.0 [1]

[1] https://docs.soliditylang.org/en/v0.8.0/080-breaking-changes.html

- *Front running.* This vulnerability is not an issue, we just need to set a proper upper Gas limit when deploying the Lottery contract
- *Miners attack.* The random number generation used in our implementation uses the keccak256 hash of three different elements (seed, block.number, block.timestamp). Since the miner knows the block.number and can also adjust the block.timestamp, this random number generation can be exploited to attack the contract by the miners. The solution would be to use an external source to generate the random number instead of using a source from the blockchain itself
- *Phishing attack.* We use the msg.sender instead of tx.origin to avoid this vulberability

3.3 Other Issues

We would like to always ensure at least one NFT available per class by minting a new NFT immediately when the NFT of that class is transferred as a prize. However, this caused the JavaScript VM to continuously crash. This issue can be solved by deploying on a more stable test environment instead of using a VM.

4 Simulation

We provide a chronological list of operation that allow to run a meaningful simulation of the system on Remix IDE [2]. Before starting the simulation, you need to create a new workspace and upload the contracts *NFTAnimal.sol* and *Lottery.sol*. Then compile the *Lottery.sol* and deploy it using a JavaScript VM. For simplicity, you should limit the constant variables MAX_POWERBALL in *Lottery.sol* defining the range of the powerball numbers to win more easily the lottery. Finally, in order to simulate a meaningful round of lottery you need to:

1. Start a new round by clicking on *startNewRound*
2. Buy tickets by inserting the correct TICKET_PRICE (=1) and the proper currency (=Wei). We suggest to change the Account while buying the tickets for a more meaningful simulation
3. After having purchased ROUND_DURATION (=5) tickets, click on *drawNumbers*
4. Distribute prizes by click on *givePrizes*

We want to inform you that due to the unstability of a VM simulation, Remix IDE may sometime crash and you will need to restart the simulation again. One thing that may improves the stability is to compile the code using by enabling the optimization.

[2] https://remix.ethereum.org/