COBrA Relationship

ALDO D'AQUINO • A.Y. 2017/18

Table of Contents

1 Introduction

1.1 Aim of the project

COBrA is a university project that aims to implement a decentralized content publishing service.
Some authors will be able to publish their contents on a catalog and users will be able to use them by purchasing them or subscribing to a premium subscription.
Contents can be songs, videos, photos or something else created by the artists and they will be rewarded accordingly to customers' fruition.
The system will have to rely entirely on the blockchain, so as to be completely decentralized and not having to depend on third-party servers that entail high running costs, are single points of failure and can often be unreliable.
The purpose of the project is to deepen the use of blockchain for decentralized and secure systems, other than the mere exchange of money.

1.2 Project specifications

Users of the system are divided into Customers and Authors. An author can also be a customer of the system as he may be interested in content from other authors.

Customers are divided into two types: Standard and Premium.
All the accounts start as Standard accounts: they must buy contents before accessing them and they are entitled to a single fruition of them. For further access they need to buy it again.
Standard account can subscript a Premium access that last for x blocks number on the blockchain. During the Premium period they can watch all the contents for free as many times they want.

1.3 Implementation specifications

The system must be implemented a single Catalog Contract and some Content Management Contracts.
The number of Authors, Customers and Contents is not fixed and can change dynamically.

Each Content Management Contract controls exactly one unique content.
An author submits a new content by deploying a new Content Management Contract that inherit a predefined set of functionalities by extending the Base Content Management Contract. The Content Management Contract must include the name of the content and the content data and may include other features for content management at the discretion of the author.
Once the Content Management Contract is deployed the author has to submit a new content publishing request to the Catalog Contract to link the new contract to the Catalog. This linking is provided by a method call with which the Catalog can find the necessary information about the content from the Content Management Contract.

The Catalog Smart Contract acts as an intermediary between Authors and Customers.
Customers can consult the library of all the contents published in the Catalog by Content Management Contracts.

Customers access the Catalog Contract to request access to contents. To access the content, the Customer must send to the contract an amount of ether equal to the cost of the content, which for simplicity is the same for each content. Premium accounts do not have to pay and can directly access all the contents for free until the Premium subscription expiration. When the access is granted the Customer can consume the content at the Content Management Contract.

All the payments from the users are collected and redistributed among authors by the Catalog Contract, according to the number of views of each content.
For the sake of simplicity Premium Account content fruitions are not considered in the view count.

To ensure decentralization of the system the Catalog Contract deployer cannot receiver any rewards: at the end of the Catalog Contract the remaining budget must be redistributed among the authors according to the number of views obtained by their contents.

Both the Catalog Contract, the Base Content Management Contract and any Content Management Contract must be written in solidity and deployed on the Ropsten network.

It is mandatory to implement at least the following functions:

- Public views
    - GetStatistics(): returns the number of views for each content.
    - GetContentList(): returns the list of contents without the number of views.
    - GetNewContentsList(): returns the list of newest contents.
    - GetLatestByGenre(g): returns the most recent content with genre g.
    - GetMostPopularByGenre(g): returns the content with genre g, which has received the maximum number of views
    - GetLatestByAuthor(a): returns the most recent content of the author a.
    - GetMostPopularByAuthor(a): returns the content with most views of the author a.
    - IsPremium(x): returns true if x holds a still valid premium account, false otherwise.
- Public actions (modify the state of the contracts):
    - GetContent(x): pays for access to content x.
    - GetContentPremium(x): requests access to content x without paying, premium accounts only.
    - GiftContent(x,u): pays for granting access to content x to the user u.
    - GiftPremium(u): pays for granting a Premium Account to the user u.
    - BuyPremium(): starts a new premium subscription.

2 Implementation

2.1 Style

We choose to write all the code in compliance of the official stylish guidelines (1). In particular we refer to the Solidity version 0.4.24 (2), which is the last release at the moment. The Ethereum docs are continuously under construction, their progress as well as the latest version that still under construction (3) can be found in the *docs* path in the GitHub repository (4).
Among others, we consider in particular the following conventions.

2.1.1 Encoding

The stylish guidelines recommend UTF-8 or ASCII. We choose UTF-8.

2.1.2 Indentation

Indentation is 4 spaces wide as suggested; spaces are preferred instead of tabs.

2.1.3 Line breaks

There must be 2 empty lines between contracts, 1 empty line between the functions implementation and 0 between functions and variables declaration.

2.1.4 Spaces

Spaces must be avoided immediately inside parenthesis, brackets or braces, and Immediately before a comma or a semicolon.
Instead there should be a single space between the control structures if, while, and for and the parenthetic block representing the conditional, as well as a single space between the conditional parenthetic block and the opening brace.
Parenthesis should be opened on the same line as the declaration and closed on their own line at the same indentation level as the beginning of the declaration.

2.1.5 Components order

Imports must be placed at the top of the file. Then will come variable declaration, events, modifiers and finally functions.

Variables are usually ordered starting from constants, then runtime variable, the ones that changes during the contract lifecycle, and finally structs and arrays.
Functions have to follow this order: constructor first, then fallback function and then, accordingly with the visibility modifier, external, public, internal and lastly private functions. For the same visibility, priority is given to the functions that modify the status, leaving the views and the pure at the end.
We choose to put the suicide function immediately after the fallback function because we believe that it is one of the special functions.
We have placed the functions with the onlyOwner modifier at the end of the public functions, as their access is not totally public but is restricted to a single person.

2.1.6 Modifiers

There is not a specific order for modifiers, but the visibility modifier for a function should come before any custom modifiers.

2.1.7 Naming

Contract, struct and event names must be in CapitalizedWord style. Variable, modifiers and functions in mixedCase.
For this reason we choose to rename all the mandatory functions from CapitalizedWord to mixedCase, considering the coding language specifications more important than the assignment.

2.1.8 Other

Declarations of array variables should not have a space between the type and the brackets.
Strings should be quoted with double-quotes instead of single-quotes.

2.1.9 Comments and documentation

Although there are no specifications regarding the comments we have noticed that remix supports Doxigen-style (5) documentation and we have therefore decided to adopt this convention.

2.2 Design choices

2.2.1 Suicide function

The suicide function already exists, it was the deprecated version of the selfdestruct function. We consider reasonable to not override a default function, although is deprecated. For this reason the suicide function starts with an underscore.

2.2.2 Parameter names

All the mandatory functions have x as argument name except for GiftContent that has also u that stands for user. For clarity we choose to rename the parameter name choosing a for author, g for genre, leaving u for user and customer and limiting x only for content.
This convention that we adopted helps to keep the code clean and readable. In any case all the parameters of the functions have been documented in the code.

2.2.3 Contract vs storage structures

A fairly common practice is to guarantee access to a resource by deploying a contract. Sometimes contracts are used instead of structs as a method of storing data.
We have found these policies to be improper because they create a large number of contracts and transactions on the blockchain and this helps to create spam on the Ropsten network. In addition, the amount of gas to deploy a contract is much higher than that needed to save a piece of storage: according to the Appendix G of the Yellow Paper (6), a store operation costs 20000 units of gas, a contract deployment 32000.

2.2.4 getContentPremium

We have arbitrarily decided to not implement this function. The reason is that a user could pay only a premium cycle and access all content in the future. even if the premium account is no longer active. The only limitation would be that he can watch it once only. In this case we thought that the premium account would have turned into a "bundle" of content rather than a subscription. Since this is not the expected behavior, we have abolished the function and now a premium user can consume all content without having to first request access to it as long as his premium subscription still valid. When the Content Manager Contract checks if a user has the permission to consume a content, the Catalog will always return true if the user has a Premium subscription.

Access to content from a premium account will not affect previously purchased content. The user can still consume the purchased content (once only) when the subscription has ended, even if that content has been accessed by this user multiple times during his premium account. In addition, a Premium account can also purchase content. They will be consumable (once only) when the premium subscription has ended.
We design the consumable content of a user as an array. We can put elements in the array even if the user is Premium, as long as he pays them, but we remove elements from the array only if the user has not a Premium subscription.

2.2.5 Premium subscription

We decide to use block heights as measuring meter for Premium subscription time.
Every time that a user buys a Premium period we save in a mapping the number of the block after which the Premium subscription expires. When the isPremium function is called we just check if the current block number is greater than the expiration block.
A Premium user can buy another Premium cycle also before the current one is expired, simply the new cycle will start after the current one. This is implemented by adding another Premium cycle length to the mapping value corresponding to the user's address.

2.2.6 Murder function

We implement on the Base Content Management Contract a murder function, that has the same behavior of the suicide function but can be called from the catalog.
This function will be used by the catalog only when it is closed to delete all contents. Any balance will be transferred to the author. Users will be informed with an event.

2.2.7 Suicide function

When the owner calls the suicide function the balance is divided between authors. First, we pay all the unpaid views from non-Premium users. Then the remaining balance, that comes from the Premium subscriptions, is divided between all the authors according to the view numbers.
It is possible that after this operation the balance is not yet zero. This is because we do not consider the remains of the divisions. We thought fair to transfer this little amount to the owner not as an earning but as refund for the big amount of gas used for this operation.

We also took some precautions to limit the consumption of gas. First of all, we call the murder function on all the contents: this generates negative gas to be used for operations. Then we do two different computations to calculate the amount to be transferred to the authors, but we perform one single transaction for each author. Finally, the Catalog is deleted and that generates a lot of negative gas: the owner must provide a big quantity of gas to not run out of gas but not all of this will be spent thanks to the negative gas that lowers the worn one.

Another important aspect of the suicide function is that it leaves the callers of functions on the suicide contract in an undefined state, creating many problems.
To avoid this, we have provided a removesMe function on the Catalog that is called by the Content when committing suicide. The Catalog instead, as already described above, will murder all contents before committing suicide.

2.2.8 Authors payment

We decide to pay the authors after v views among all their contents. We found it more appropriate than paying them after v views on a single content, as they would have been collected after too much time in the case of a lot of content with few views each. Furthermore, this allows us to raise the number of views and make fewer payments, which entail a cost in gas for the author.
We also decide to not trigger the transaction as soon as v views are reached but only emit an event that notice the author that it can withdraw his reward. This is because otherwise when the v-th user consume the content, the transaction starts, and it cost to that user 21000 gas and 0 to the other users. In this case instead we have a fairer solution in which the author pays the gas needed for his withdraw.
Furthermore, the author can decide to not withdraw immediately his reward but to wait to do a bigger transaction instead of lots of small transaction, paying less gas.

2.2.9 Statistics functions

For all the statistic functions, like getLatestBy_ and getMostPopularBy_, there are two opposite kinds of approach. We can keep chart lists updated storing additional information or we can store only needed information and generates the charts each time they are requested. The first solution

increases the gas expenses, the second one leaves all the burden to the consumer calling the methods.
We also have to consider who pays for this operation: in the first case the gas cost falls on the customers that consume the content, that may be not interested in charts and statistics, in the second case there is no extra cost and the burden is up to the user that requested it.
For this reason, we have preferred the second approach, that avoids unnecessary costs in terms of gas and also lightens the blockchain from unnecessary data.

2.2.10 Parameters of the system

The cost of content is a rather generic and difficult to determine, as it depends on the type of content offered, so we are limited to giving an order of magnitude rather than an exact price.
To do this we started from the actual value of the ether, which is around 400€ (7).
We consider 4€, which corresponds to 0.01 ether, a reasonable amount for the individual content, even looking at the current cost in online stores.
As for the Premium subscription, prices range from 10 to 40 euros per month. We therefore considered that 0.1 ether is the right price and we set the premium time to 172800 blocks that are about a month considering that currently mining a block requires 14.99 seconds (8).
As far as the redemption of the rights by the authors, we thought it was right to make the withdrawal possible after 10 viewing. This number is a trade-off to allow authors with few visits not to wait too much and at the same time to ensure that the cost of gas does not affect too much the value taken. Recall that the number of visits is related to all content of the author and not to the individual content.
10 views could be a low number, considering that they do not refer to the single content but to the sum of the visits of all the contents of an author. However, we preferred not to increase the number too much considering that an author can choose to wait more time to withdraw a bigger amount later, paying only one transaction.

2.2.11 Events

The only event that had to be mandatory emitted was the one concerning access granting on a content. Considering the events an important form of logging and aiming to make the behavior of the contract as clear as possible, we decided to issue also other events.

We thought important to communicate not only when a user buy a content, but also when a user becomes premium and when a content is consumed.
Another important event is fired when the Catalog contract is closed, to notice all the interest user that they cannot access theirs account anymore. Similar to this the content deleted event.
Opposite to these, two events are fired when a content is published. The content emits a published event, and the catalog a new content available event.
Finally, it has become necessary to fire an event when an author's payment is available to notice this author that his withdrawals is available.

2.2.12 Content

We have chosen not to store the content in the Content Management contract. This would have implied the input of a large amount of data on the blockchain, which involved various problems, including the cost of gas to deploy the contract.
The task of serving the content to the user is left to others, for example to a DAPP. The only task we have assigned to the Content Management contract is to managing the rights checks for the content access.

2.3 Implementation choices

2.3.1 Assert, require, throw and revert

Assert and require have the same purpose to verify the conditions and stop the execution of the code in case they are not satisfied. However, in the current version of Solidity, the 0.4.24, assert uses throw and consume all the gas, instead require uses revert and will refund all the unused gas.
Therefore, although their behavior is almost equivalent from the point of view of the contract, it is very different from the user's point of view.
Assert vs require is a hotly debated topic about Solidity programming on the web, and we agreed with what Steven McKie wrote in his article (9). The basic idea is that consuming all gas should be a punishment for the user, to be used only when he has done something that he should know he did not have to do. In all the other cases revert should be fine, because means that something went wrong or that the user has done some mistakes in passing variables.
After reflecting on the possible cases of use of an assert we considered that among the functions implemented in this project there was really no wrong action for which it was revealed a need to

punish the user for something. Therefore, we considered it more appropriate to use require everywhere.
The only case in which the assert could be predicted was in the modifier onlyOwner, whose meaning is quite evident, and the user should be aware of the fact that that function is not reserved for him. However, in Remix the modifier is not explicitly visible and therefore the user may not know that this function is reserved. For this reason, we decide to use require also in this case.

2.3.2 Strings

Strings are usually encoded as bytes32. The string type is already available on Solidity but are not fully supported. In particular to be returned by a function call must be enabled the pragma experimental ABIEncoderV2 that, as the name says, is not yet stable.
Another possibility is to accept the strings as a function parameter and transform the outputs into bytes32, but this requires low-level calls that are to be avoided where possible.
We felt that the best solution was to use bytes32 everywhere, because nowadays is a widespread practice and is currently the standard string type.
The use of bytes32 also entails some advantages: strings are considered bytes and therefore arrays of infinite length. This makes it difficult to estimate the amount of gas needed and exposes you to possible attacks aimed at consuming an excessive amount of gas to users. bytes32 instead has a fixed length and therefore guarantees exactly the amount of memory used.
The disadvantage is that bytes32 is an array of bytes of length 32. Therefore, since each letter occupies exactly 8 bits in UTF-8 encoding, it is possible to handle only 32-letter strings. However, we thought it was enough for a content title, also considering that for example “This is a 32 letters long title.”.

2.3.3 Arrays

We always choose static arrays where possible, because they are cheaper then dynamically sized ones.
We also consider different options for storing list of data, but what we notice is that in memory arrays and mappings work behave in similar ways and the gas cost is more or less the same, so we just use the one that is more convenient for the purpose.

For byte arrays we usually use bytes32 as there is no extra cost for allocating empty position in memory without initializing them.

2.3.4 Reentrancy problem

The reentrancy problem (10) is a common bug that is easy to introduce in the code that leads big problems because involves money, and most likely not only our money but also involves other people's as well. In this case a reentrancy bug can allow an attacker to withdraw the entire balance of the contract, damaging all the authors.

A reentrancy attack consists of repeatedly and recursively calling the same function until the entire balance has been taken. Usually is done calling a function, i.e. "withdraw", that start a transaction and in the function that receives this transaction calling again the same "withdraw" function. This will suspend the first call on the transaction and starting another transaction with the second call.

In our case this attack could be done on the function collectPayout. Consider the following example:

```
function collectPayout() public {
    require (authors[msg.sender].uncollectedViews >= payAfter);
    uint amount = contentCost * authors[msg.sender].uncollectedViews;
    msg.sender.transfer(contentCost * authors[msg.sender].uncollectedViews);
    authors[msg.sender].uncollectedViews = 0;
}
```

This function can be exploited with this simple contract:

```
pragma solidity ^0.4.0;
contract CatalogContract {
    function collectPayout() public;
}
contract Exploit {
    function collect(address addr) {
        Catalog memory catalog = Catalog(addr);
        catalog.collectPayout();
    }
    function () payable {
        if (catalog.balance >= msg.value) {
            catalog.collectPayout();
        }
    }
}
```

The collect function calls the collectPayout function. The collectPayout runs until the transfer. The transfer fires the fallback function of the Exploit contract and will return only when the fallback function returns. After the transfer the authors uncollectedViews property is reset. The problem is that the fallback functions does not return but calls again the collectPayout function. The author uncollectedViews property is not yet reset, so the collectPayout function starts another transaction with the same amount. This action is repeated until the Catalog contract has a balance. Then all the recursive fallback functions return and only at this point the collectPayout functions update the uncollectedViews, after have transferred multiples time the amount to the author.

Below you can find the implemented collectPayout function rewritten without the reentrancy bug:

```
function collectPayout() public {
    uint uncollectedViews = authors[msg.sender].uncollectedViews;
    require(uncollectedViews >= payAfter);
    authors[msg.sender].uncollectedViews = 0;
    uint amount = contentCost * uncollectedViews;
    balance -= amount;
    msg.sender.transfer(amount);
}
```

2.4 Gas estimation

We propose below the list of functions with relative cost in estimated gas.

This list can also be useful to better understand the structure of contracts.

Node that we do not provide the cost of functions that can only be called by other functions, such as internal functions or functions that can be called only by a specific function of a specific contract. The gas cost of this functions is included in the gas cost of the calling function.

2.4.1 CatalogContract

```
// gas cost: 2149126
constructor() public;
```

```
// only revert the state
function () public;

// gas cost: at least 17167 (empty contract)
// + 3016 for the first author with his first content (20183)
// + 5014 for each other additional content of any author (25197, 30210, 35224)
// + 5840 for each additional author with his first content (26023)
// + at list 7927 if there is at least 1 view in all the catalog
// (depends on the number of contents in the catalog) (28110)
// + 4028 for each additional author that has at list a visit (49070)
function _suicide() public onlyOwner;

// gas cost: 68902 first time,
// 53902 if the user has already purchased and consumed this content previously,
// reverted if the has already purchased but not consumed this content
// (includes the grantAccess function cost)
function getContent(address x) public payable exists(x);

// gas cost: 55482 (includes the grantAccess function cost)
function giftContent(address x, address u) public payable exists(x);

// gas cost: 63351 (includes the setPremium function cost)
function giftPremium(address u) public payable;

// gas cost: 62937 (includes the setPremium function cost)
function buyPremium() public payable;

// gas cost: 25392
function collectPayout() public;

// included in BaseContentMenagementContract.publish()
function addMe() public;

// included in BaseContentMenagementContract.consumeContent()
function consumeContent(address u) public exists(msg.sender);

// included in BaseContentMenagementContract._suicide()
function removesMe() public exists(msg.sender);
```

```
// view: no gas consumption
function getStatistics() public view returns(bytes32[], address[], uint[]);

// view: no gas consumption
function getContentsList() public view returns(bytes32[], address[]);

// view: no gas consumption
function getNewContentsList(uint n) public view returns(bytes32[], address[]);

// view: no gas consumption
function getLatestByGenre(bytes32 g) public view returns(bytes32, address);

// view: no gas consumption
function getMostPopularByGenre(bytes32 g) public view returns(bytes32, address);

// view: no gas consumption
function getLatestByAuthor(address a) public view returns(bytes32, address);

// view: no gas consumption
function getMostPopularByAuthor(address a) public view returns(bytes32,
address);

// view: no gas consumption
function hasAccess(address u, address x) public view exists(x) returns(bool);

// view: no gas consumption
function isPremium(address u) public view returns(bool);

// included in buyPremium and giftPremium
function setPremium(address u) private;

// included in getContent and giftContent
function grantAccess(address u, address x) private;
```

2.4.2 BaseContentManagementContract

```
// gas cost: 800201
constructor() public;

// only revert the state
```

```
function () public;

// gas cost: at least 35594
// (it depends on how many contents are published in the catalog)
function _suicide() public onlyOwner;

// gas cost: 14472
function murder() public validAddress(catalog);

// gas cost: usually 41292,
// 86292 if it is the first view for this content,
// 42421 if the paymentAvailable event is emitted
function consumeContent() public returns(bytes);

// gas cost: at least 171503 (it depends on the name and genre length)
// If it is the first content published by the author,
// there is an additional expense of 80739 units of gas.
function publish(address c) public onlyOwner validAddress(c);
```

2.4.3 GenericContentManagementContract (is BaseContentManagementContract)

```
// gas cost: 1016403 (includes BaseContentManagementContract.constructor)
constructor() public;

// gas cost: at least 42532 (it depends on the input string)
function setName(bytes32 n) public onlyOwner notNull(n);

// gas cost: at least 42510 (it depends on the input string)
function setGenre(bytes32 g) public onlyOwner notNull(g);
```

2.5 Test suite

To test the work was required a list of functions capable of simulating the system in action.
However, we felt that compiling and deploying contracts one by one was too long and that this would not allow to test the system with an appropriate number of contracts.
For this reason, we have developed a test suite that automatically performs all the necessary operations, logging the information needed to understand its functioning and to check that the system works properly.

The test suite is written in NodeJS. We have chosen this language because in our honest opinion is more flexible than Java, giving us greater freedom. Furthermore, the syntax is more similar to that of Solidity. Finally, from the perspective of a master's degree, we thought it important to learn the use of a language different from those already experimented for other courses.
To run demo are required Node.js (11) and a web3 interface (12) running at localhost on port 8545. You can use geth (13) or RPC-test (14).
To prepare the tests run `npm install` on the root folder of the system. This will also download RPC-test that you can run with `node node_modules/.bin/testrpc`.
To start the demo run `npm start` or `node demo.js`.

3 Works Cited

1. Ethereum organization. **Solidity Style Guide**. *Read The Docs.* [Online] May 09, 2018. http://solidity.readthedocs.io/en/v0.4.24/style-guide.html.

2. —. **Solidity version 0.4.24**. *GitHub.* [Online] May 16, 2018. https://github.com/ethereum/solidity/tree/v0.4.24.

3. —. **Solidity Style Guide**. *Git Hub.* [Online] https://github.com/ethereum/solidity/blob/develop/docs/style-guide.rst.

4. —. **Solidity**. *GitHub.* [Online] https://github.com/ethereum/solidity.

5. Heesch, Dimitri van. **Documenting the code**. *Doxigen Manual.* [Online] May 23, 2018. http://www.stack.nl/~dimitri/doxygen/manual/docblocks.html.

6. Wood, Gavin. **Yellow Paper**. *Ethereum repository on GitHub.* [Online] May 05, 2018. https://ethereum.github.io/yellowpaper/paper.pdf.

7. ***ethereumprice.*** [Online] [Cited: June 13, 2018.] https://ethereumprice.org/.

8. Ethereum organization. ***Etherscan.*** [Online] [Cited: June 9, 2018.] https://etherscan.io/.

9. McKie, Steven. **Solidity Learning: Revert(), Assert(), and Require() in Solidity, and the New REVERT Opcode in the EVM**. *Medium.* [Online] September 27, 2017. https://medium.com/blockchannel/the-use-of-revert-assert-and-require-in-solidity-and-the-new-revert-opcode-in-the-evm-1a3a7990e06e.

10. Ethereum organization. **Solidity re-entrancy**. *Read the Docs.* [Online] May 9, 2018. http://solidity.readthedocs.io/en/develop/security-considerations.html#re-entrancy.

11. **Node.js. Download**. *Node.js.* [Online] https://nodejs.org/it/download.

12. Ethereum organization. **Web3**. *GitHub.* [Online] May 17, 2018.

https://github.com/ethereum/web3.js/.

13. —. **Download.** ***Geth.*** [Online] https://geth.ethereum.org/downloads.

14. —. **rpc-tests**. *GitHub.* [Online] September 11, 2017. https://github.com/ethereum/rpc-tests.