Francesca Sbolgi 554920
francesca.sbolgi@gmail.com

Peer to Peer Systems and Blockchains Final Term

COBrA: Fair COntent Trade on the BlockchAin

Academic Year 2017/2018

1 Introduction

COBrA is a decentralized service that allows the authors to publish their creations (songs, videos, photos, ...) and the customers to consume them by paying some kind of fee. Later, the authors are rewarded accordingly to customers' use of the content.

1.1 Structure of the Project

In this implementation we decided to create just two contracts:

- the *Catalog*, that keeps track of all the contents submitted and handles the communication from customers to authors;
- a *BaseContentManager* that saves the data and provides the actual fruition of a specific content.

Another possible way to structure the problem would have been to make the *BaseContentManager* abstract and let the author implement a contract that suits his necessities. Instead, we decided to create just one kind of content manager contract that will fit the requirements of most contents. This way an author can easily publish his creation without having to implement a new contract. However, if an author wants to add more functionalities, for example to give more detailed information about the content, he can always create a new contract that extends the *BaseContentManager* and enrich it with new functions. [1]

2 Implementation

2.1 Base Content Manager

Data Stored
Each *BaseContentManager* keeps a set of data relative to the contract itself and to the content it's managing.

[1] http://solidity.readthedocs.io/en/v0.4.24/

The variables that are independent from the content are necessary to prevent malicious behaviours. The owner and the contract address are useful to check that the EOA submitting the content to the catalog is the one that deployed the contract. Moreover, by knowing the owner address is possible to reward directly him for the content fruition and avoid to write a fallback function. Another variable used to write specific access modifiers is the address of the catalog. Instantiating it in the *BaseContentManager* constructor has a double purpose: remove the possibility of changing the catalog address once the content is created and use it to enforce that some functions can be called only by the catalog. The presence of just one these conditions is not sufficient to prevent view tampering.
On the other hand the variables relative to the content are title, author and genre. We assumed the title to be an unique identifier of the content. The author in our perspective is not linked to the owner of the contract account, indeed we believed that a person may have many pseudonyms and at the same time the same author may have many addresses. The genre we intended to be encoding both the category of the content and the actual genre, so it can be either a string such as "song:rock" or simply the genre "rock" alone that implies the category.
All three variables are public since they are necessary to surf the catalog. They are saved as bytes32 since strings are hardly handled in solidity: they have mutable length so it's not possible to return a dynamic array of string and saving strings in a contract instead of bytes32 imply a higher gas expense. [2] The conversion from bytes32 to string, and viceversa, in order to make the Dapp more user friendly, will be done in the front-end. By pushing forward this computation we can save the gas of the translation or, if the output in bytes32 is used later as input for something else, the conversion may not even take place.
We have also inserted three variables to manage the views: one that counts the number of fruitions by standard customers, another that keeps track of how many of those views have already been paid by the catalog and a variable, common with the catalog, that establish the minimum number of views to require a payment.
In the end, the content keeps two mappings, one from standard customers to a boolean value that stores the access right and a second one from premium customers to an integer number. This value keeps the block number [3] representing the expiration date of the premium subscription. Indeed we believed that it would be fairer to allow premium users unlimited access to contents only within their subscription time, otherwise, a premium customer may spam a lot of access requirements and later decide whether to consume the content after the expiration date was long gone.

Events Triggered
An event is triggered every time a function that modifies the state is called by an EOA. So, for example, every time a new *BaseContentManager* is created or once it's destroyed, when a customer consume the content and when v visualizations are reached.

Functions
The functions of the *BaseContentManager* can be called each by a specific entity. Only the catalog can call *Authorize*, *AuthorizePremium* and *Payed*. The first one takes as input a

[2] https://ethereum.stackexchange.com/questions/11556/use-string-type-or-bytes32
[3] http://gavwood.com/paper.pdf

standard customer and stores that he is allowed to consume once this content, the second one does the same but for premium accounts and the third simply increase the number of views that have been payed.
The *ConsumeContent* function can be called only by customers authorized by the functions above and simply mark the content as consumed by that customer. Moreover, if it's a standard customer, it increases the total views count and checks whether a multiple of v views are reached.
KillContent allows the owner to destroy the content. There may be an issue if this function is called before a customers can consume it. However, the owner will not benefit from this, since he can earn money only after the consumption. We thought therefore that this function can't be exploited to enable malicious behavior and that it will be used in very rare cases.

2.2 Catalog

Data Stored
The *Catalog*, like the *BaseContentManager*, saves its address and its owner. The catalog's address is the natural counterpart of the one stored in the content, it's needed to check that a content has the right catalog variable. Once the catalog is deployed, the owner can no longer interfere: he will not gain anything from providing this service and the only thing he is allowed to do is to destroy the catalog.
The *Catalog* must handle in an efficient way the data relative to the contents and to the customers. Indeed it's fundamental to reduce as much as possible the number of "write in the storage" since this is the most gas consuming operation. We optimized the gas consumption at the expense of the computational time, we thought anyhow that, when running a program on blockchains, the few milliseconds necessary to scan an array are neglectable comparing to the transaction time.
So, to store the contents' list, we thought of using just an array of *BaseContentManager* that allows to keep track of all the contents added to the catalog. This, however, was not enough since it does not allow direct access to a content given a title.[4] To face this problem we used also a mapping from a content's title to the position p in the array. Well, actually, to the $p + 1$ to avoid mistaking an element in position 0 from one that is not present.
While standard customers does not require to be stored, for premiums is necessary to check the validity of the subscription. We decided to measure time as block height since the time for mining a block may vary according to many factors. So, a measure relative to the blockchain evolution we thought would lead to fairer results.
The subsequent problem was to estimate a possible duration for the premium subscription. The current average transaction time for Ethereum is about 14 seconds. Even though this value is quite stable, we didn't want to choose a value that may cause issues if the transaction time changes a lot. We decided to set the premium duration as 40000 blocks. As for the current transaction time this is equivalent to about a week ((60 seconds*60 minutes*24 hours*7 days)/(14 seconds) = 43200). This way, even if the transaction time doubles or is halved, the users have the time to consume the contents they payed for.

[4] https://ethereum.stackexchange.com/questions/13167/are-there-well-solved-and-simple-storage-patterns-for-solidity

But how much would it be fair to pay for a content or for a week of unlimited accesses? The current exchange rate is 415 Euro for 1 Ether. As above, we wanted to choose a value that would still be fair even with big exchange rate fluctuations. So we set the price of a content to 0.002 Ether (about 80 cents) and the price of a premium subscription to 0.03 Ether (about 12.5 Euro). This way it's convenient to buy a premium account if you consume more than two contents each day.
The last parameter to set is the number of visualizations that must be reached before being able to require a payment. If this is set too low the cost of the transaction would exceed the money payed but, on the other hand, if set too high small authors may never be able to require their reward. The compromise we took was to set $v = 100$ but allowing also to wait more to require the reward. In this way contents with a lot of visualizations can wait as long as they want and obtain the reward in just one transaction.

Events Triggered
The events triggered in this contract follow the same principle as *BaseContentManager*. Customers who wants to be notified when something changes in the the catalog can simply listen to the event instead of having to query periodically the contract. Moreover they can be used as a further confirmation that a transaction went well.
The events *content_acquired* and *premium_acquired* may be useful to the EOA calling the corresponding functions in order to store locally some values or, for instance, to an external entity collecting some statistics. For the same reason the event *new_publication* can be useful or simply for a customer who wants to be updated with the newest release.
author_payed can be useful to keep track in an easy way of the rewards given to authors.
At last we triggered an event when the catalog is created and once it's deleted.

Functions
The functions in the *Catalog* can be divided in two categories: view functions that return some kind of values and transaction functions that modify the state of the contract.
View functions, if called from an EOA, does not consume gas and therefore can be slightly more computationally complex.
First of all, we have written three utility view functions *GetAuthor*, *GetGenre* and *GetAddress*, that simply given the title of a content returns respectively its author, its genre and its address. We thought indeed that the title would be the best thing to identify a content from a user's point of view.
The function *GetStatistics* returns two arrays, one of titles and one of views where the corresponding values are written in the same position. At first we thought of returning an array of couples but structures cannot be returned in solidity. Moreover the decoupling of the two arrays may simplify some operations later on.
GetContentList and *GetNewContentsList* simply scan the array of *BaseContentManager* and returns the list of contents required. The elements in the *GetNewContentsList* are ordered from newest to older. *GetMostPopularByGenre* and *GetMostPopularByAuthor* take into account the number of views in order to retrieve the most popular title of a given genre or author. The functions *GetLatestByGenre* and *GetLatestByAuthor* both go backwards in array until they find the content required.

The last view function is *isPremium*. Since the catalog keeps the expiration date of each subscription we just have to check whether we already reached that block height. Once the subscription expires the customer is not removed from the premium list. Indeed this would not free some space in the blockchain but it will require to overwrite the value, which will consume gas and occupy more space.
The state modifying functions, since they require gas to do it, do not contain cycles.
The *AddContent* function insert a content in the catalog. *GetContent*, *GetContentPremium* and *GiftContent* perform almost the same operation: they call the function of a *BaseContentManager* for authorizing an EOA to a content. The difference is that the first requires an exact payment of the content price, the second just checks if the sender has a valid premium account and the third instead of authorizing the caller of the function take the address given as input. The same principle applies to *GiftPremium* and *BuyPremium* for purchasing a premium account.
A little less trivial function is *PayAuthor*. It can be called by whoever and, given a content's title, it checks whether the number of views that still have to be paid are more than v. In that case, pays the owner the price of the content for each view. We didn't restricted the access to this function as maybe, in a future implementation, it can be created a contract that handles the payments in a more sophisticated way. For example, for small authors it calls *PayAuthor* immediately after v views while for bigger ones it will do it less frequently.
At last the function *KillCatalog* has to perform a lot of actions before calling *selfdestruct*. It has to calculate the total number of views and the ones that still have to be paid. This is done by the function *GetTotalViews* by scanning the catalog's list. Even if it's a view function it will still be paid since it's necessary to execute a transaction. Once recovered these two results, the catalog can proceed paying each content the remaining views and, proportionally to the view count, the money of the premium subscriptions. This function is very expensive since it has to scan twice the catalog list and perform a money transaction to each content. However, we thought it would be better to make *KillCatalog* heavy rather than the others since most likely it will never be called or at most one time.

3 Execution

3.1 System Simulation

In this section we will simulate a possible execution.
First of all, we must deploy the *Catalog* contract and some *BaseContentManager* that, once created, must be added to the catalog. Then, for instance, a possible customer can request to the catalog the list of the contents, select the first title from the ones returned and look for the most popular content of that author, pay to get access to the content and, in the end, ask the relative *BaseContentManager* to consume the requested content.
The sequence function calls needed in order to execute these actions are written below. For simplicity the EOAs address will be referred just as capital letters.

deployment:
1) A: < deploys > c = Catalog();
2) B: < deploys > bcm1 = BaseContentManager(c, t1, a1, g1);
3) C: < deploys > bcm2 = BaseContentManager(c, t2, a2, g2);
4) [...];

5) B: < calls on c > AddContent(bcm1);
6) C: < calls on c > AddContent(bcm2);
7) [...]

execution: 8) D: < calls on c > list = GetContentList();
9) D: < calls on c > a1 = GetAuthor(t1) with t1 = list[0];
10) D: < calls on c > t2 = GetMostPopularByAuthor(a1);
11) D: < calls on c > GetContent(t2) paying 0.002 ether;
12) D: < calls on c > bcm3 = GetAddress(t2);
13) D: < calls on bcm3 > ConsumeContent();

3.2 Gas Estimations

Let's look at an estimation of the gas cost required to execute the sequence above.
According to Remix the cost of creating the *Catalog* and the *BaseContentManager* is about 2.89M and 705K gas each. This estimation is confirmed by the actual deployment results of 2.89M and 675K.
Once deployed, we can add a content to the catalog. Remix is not capable of estimating this cost so let's take the function *AddContent* in detail to understand how to compute it.

```
1     function AddContent (BaseContentManager _c) external {
2         BaseContentManager cm = _c;
3         require (cm.owner() == msg.sender && cm.catalog() == ...
              catalog_address);
4         contents_list.push(cm);
5         position_content[cm.title()] = contents_list.length;
6         emit new_publication (cm.title(), cm.author(), cm.genre());
7     }
```

The assignment at line 2 does not write in storage, so it consumes just 210 gas. The *require* calls two functions that cost respectively 552 and 442 and two comparisons that cost 220 each. On the fourth line there is a write in the array that costs 40784. Then you retrieve the content length, 410 gas, you compute the title, 478 gas, and you write the result in the map, 20339 gas. In the end, the event emission requires 1879 gas and 478, 588 and 434 for the getters. Summing up we obtain a result of 66814 which is quite good estimation since the actual cost we paid is 75949. As we can see the value is approximately given just by the write in storage operations and the event triggered. From now on we will consider just these operation to give an estimation of the gas consumption. The view functions *GetContentList*, *GetAuthor*, *GetMostPopularByAuthor* and *GetAddress* does not consume gas as they are called from an EOA.
The *GetContent* function performs just two heavy operations: it calls the function *Authorize* that costs 20918 gas and emits and event with 1937 gas. The *ConsumeContent*, in the worst scenario, increases the view count, 20370 gas, it emits an event, 1258 gas, it deletes an element from a mapping, 5296 gas, and send another event, 1276 gas. So the computation of these two functions require about 22855 and 28200 gas confirmed by the actual execution of 26022 and 28493.