Universitá di Pisa

Peer to Peer Systems and Blockchains

Final Term

COBrA: Fair COntent Trade on the BlockchAin

Author:
Carmine Caserio

Contents

1 Describing the implementation

The smart contracts implementation has been developed by using the Remix IDE. The implementation involves basically, the `CatalogContract`, the `BaseContractManagementContract` and each one of the `ContentManagementContracts`.

1.1 CatalogContract

This smart contract is the pivot of the entire decentralised content publishing service. The data structures involved in this, are the following ones:

- the structure `Customer`, which takes into account informations related to the user that wants to use the service in order to get all the items to which he is interested, so: the `contents` and the `gifts` that he can receive by other users, together with the `registered` and `premium` flags that represent respectively, whether the user is registered and whether the user has an active premium account, that will expire after a constant number of operations, and the number of operations that he has already done with his premium account;
- the structure `ContentsInside`, useful for knowing both the author's address and the item's name;
- the structure `Author`, in which there are some informations related to the `contents` published by the author and their length, the number of occurrences useful for computing the most popular item searched by the customers that won't be never zeroed, the number of occurrences needed for keeping track of the payment the author deserves, that will be zeroed after having update his balance, the `payment` that will be accredited to him and the address of the author necessary for accreditation;
- the structure `PossibleGenres`, that is used for keeping track of the most popular item relative of a specific genre, so inside it has a mapping between the genre and the number of views related to that genre;
- the mappings related to the `contents`, the `customers`, the `authors` and the `genres`, together with some other auxiliary mappings such as the `nameAuthors` used to know the authors that participate in the service, the `latestAuths` and the `latestGens` for keeping track of the latest item relative of an author or a genre, etc;

Besides, many of the view functions does not require that an user is registered as customer, because, in this way he can see the items offered by this service, but when a transaction or when the informations of the customers are involved, the customer needs to be authenticated.

In all the functions developed, for efficiency reasons, there would be too much overhead by using the fields of a smart contract inside this `CatalogContract`, so we save them as `memory` variables inside the function if they are used several times inside it.
In the *getContent()*, *getContentPremium()* and *giftContent()* functions there is a call to the *searchContent()* function that iterates among all the items of the contents for searching the chosen one; this means that there is a direct proportionality between this function cost and the number of items. This happens in the *updateMaxViewsForGenres* too, in

which there is a call to the *searchGenre* even if, in this case, we expected to have too less iterations, because the genres are not so much, normally.

The *distributeFeeds()* function is called in two cases. The first case is when the customer is consuming the content, using the *consumeContent()* function, and in this case the function checks if the number of occurrences exceeds the limit beyond which the redistribution actually happens; in case of positive answer, after the redistribution phase, in which the system takes into account the amounts to pay to each author, there is the actual sending of the wei they are entitled. The second case is when the owner of the `CatalogContract` want to close the contract, and calling the *closeContract()* function, it'll call this function and will redistribute all the fees gathered up to the *closeContract()* function call.

The *getContent()* function is only for the non-premium customers, so that, whether a customer receive a premium account gift from another customer, the `onlyForStandard()` modifier in the `Vm error: revert` message, it'll communicate him that he's premium.

The premium account can be bought by using the *buyPremium()* function, or by being the recipient of premium gift from another customer. The *getContentPremium()* function checks whether the customer is already a premium one and possibly updates the premium validity, so that the next time, the customer will use the same system functionality, he'll see his premium account expired. The *giftContent()* function requires always a wei transfer because there could be the situation in which a standard customer gets along with a premium account to give him all the items he's interested in, without paying (obviously, supposing that the premium account costs less than the sum of the costs of all those items).

The total amount gathered by all the payable functions when called by the customers, is then redistributed onto all the authors that have published at least an item, proportionally to the number of views they received for each item. The functions that have the task to gather the total amount that has to be divided between the authors are the following ones: the *buyPremium()*, the *closeContract()*, the *consumeContent()*, the *getContent()*, the *giftContent()* and the *giftPremium()*.

1.1.1 System parameters

Before proceeding with the description of the other smart contracts, we pause on the choice of the premium duration and cost, and the limit of occurrences beyond which the *distributeFeeds()* function is called: the premium duration used *for testing* is very low for allowing that certain situations occur rapidly, after just some transaction and the chosen value is 3, but for real cases, the value to consider could increase, so the value could be around 15. The same considerations apply for the cost: for being fair, the value used in the tests is about a fifth of normal, so since the cost that I've decided in the real case is the equivalent of about 5€, that is $0,01$ ether, in the case test it is about 1€, that is $0,002$ ether. The limit of occurrences beyond which the *distributeFeeds()* function is called, used for the *test cases*, has been setted to an height of 2 blocks, for the same reason of the premium duration; in the real cases, it depends on the usage level of the system, because if this is setted to a too low level, there would be too much transactions

all about at the same time. This means that for a real case, the block height to reach before triggering the function call can be setted to an high value such as 100 or even more, depending on the usage level of the system and on the number of authors that have published some item.

1.2 BaseContentManagementContract

This smart contract is the one that is inherited by all the `ContentManagementContract` ones. This provides all the methods useful for handling an item, indeed, it provides all the view methods for obtaining the name, the genre, the author, the cost, the statistic (that is, the number of views), the time (that is, the block number), the address of the owner and the balance.

Besides, there are:

- the *addContent()* function that calls the public *addContent()* function of the `CatalogContract`;
- the *grantAccess()* function that sets to true the access of the address passed as parameter;
- the *consumeContent()* function that checks whether the address has the access granted and then increments the view count.

1.3 ContentManagementContract

Each `ContentManagementContract` only calls the constructor of the `BaseContentManagementContract` with the parameters asked in it. Since it inherits from the `BaseContentManagementContract`, all the methods declared into the parent are available in these one.

The given example is about the handling of songs, for this reason in the constructor of the bunch of `ContentManagementContract`s given below of the definition of their parent concern the name of songs that are given with the name of the corresponding author and all the informations needed.

the situation in which two `ContentManagementContract`s are deployed by two different owners, but they refers to the same author (in a Youtube fashion) has not been handled, giving preference to an other kind of situation in which if two `ContentManagementContract`s refers to the same author, then they have the same address (in a Spotify fashion).

2 Important events generated by contracts

In the development of all the smart contracts, the decision of triggering events has been delegated to the only CatalogContract, which is the main one and the only one for which makes sense to trigger events.

The triggered events in the various functions are:

- `contentAdded`, which is triggered in the *addContent()* function when an author publishes a content;
- `giftContentSent`, triggered in the *giftContent()* function when a customer makes a present to another one which consists of an item;
- `giftPremiumSent`, triggered in the *giftPremium()* function when a customer gives to another a premium account;
- `grantContentAccess`, which is triggered when a customer has asked to get a content both in the premium and in the standard cases, so it's called both in the *getContent()* and in the *getContentPremium()* functions and also when a customer that has received a present consisting of an item from another customer; in this case, obviously, the access is granted for the recipient of the gift and the function in which the event is triggered is the *getFromAGift()* function;
- `boughtPremium`, triggered when a customer buys a premium account, so in the *buyPremium()* function;
- `distributeTotalFeeds`, triggered when the *distributeFeeds()* function has been called, that is, either when a customer wants to obtains the access to a content and, using the *getContent()* function, he calls the *redistribution()* private function called in the *distributeFeeds()*, or when the contract is about to be closed, in which there is the redistribution of the feeds gathered, up to the closing time. This distribution has to be applied taking into account the views of the items published by the authors, that are the recipients of the redistribution.

3 Meaningful list of operations

The cases exposed in this section start from the most trivial one and they become more and more valuable up to the most significant one. In the project has not been taken into account the situation in which a customer can not buy the same item twice.

3.1 Standard case - publish&buy

In the most trivial case, the list of operations consist of just the publishing of an item by the author and the buying of this item by the customer. In more detail:

t the `CatalogContract` is deployed;

$t+1$ the author deploys the `ContentManagementContract`, with a given cost;

$t+2$ the author publishes an item by using the *addContent()* function that creates a transaction between the `ContentManagementContract` and the `CatalogContract`; here, the author has to specify the address of the `CatalogContract` and the `CatalogContract` updates all the informations relative of the publishing author, together with the genre information which is updated with the item's genre and the last item published by the author which is updated with this one;

$t+3$ the customer registers by using the *registerAsCustomer()* function of the `CatalogContract` that creates a not payable transaction between the customer and the `CatalogContract`;

$t+4$ the customer obtains the item's name by using the view function *getContentList()* of the `CatalogContract`;

$t+5$ the customer obtains the item's cost by calling the view function *getCostOfContent()*;

$t+6$ the customer uses the *getContent()* function for buying the item published by the author; this function creates a payable transaction between the customer and the `CatalogContent`; afterwards, the `CatalogContent` updates:

- the number of views of the content previously published by the author;
- the total amount of ether it has to redistribute among all the authors that participates;
- the customer's content data structure with the requested content.

Later on, the `CatalogContract` emits the *grantContentAccess* event, grants the customer access to the chosen item and updates the maximum views both for the author and for the genre data structure.

$t+7$ the customer uses the not payable transaction *consumeContent()* function between the customer and the `CatalogContract` for consuming the item, and after this execution, the list of contents of the customer remains unchanged because, if it wants to consume in a time $t+k$ (where $k \geq 6$) he can still do it.

3.2 Standard case - publish&gift

In this case, there is the analysis of another trivial case, in which two customers are registered to the system, an author publishes an item and afterwards, one of the two customers makes a present to the other one, by sending him the content published by the author. In more detail:

t the `CatalogContract` is deployed;

$t+1$ the author deploys the `ContentManagementContract`, with a given cost;

$t+2$ the author publishes an item by using the *addContent()* function that creates a transaction between the `ContentManagementContract` and the `CatalogContract`; as before, the author has to specify the address of the `CatalogContract` and the `CatalogContract` updates all the informations relative of the publishing author, together with the genre information which is updated with the item's genre and the last item published by the author which is updated with this one;

$t+3$ the customer a registers by using the *registerAsCustomer()* function of the `Catalog-Contract` that will create a not payable transaction;

$t+4$ the customer b registers by using the same function used by the customer a;

$t+5$ the customer a calls the view function *getCustomersList()* that can be called only by registered users, and select the address of the customer to whom send the gift, so in this case the customer b;

$t+6$ the customer a calls the view function *getCostOfContent()* obtaining the item's cost;

$t+7$ the customer a calls the *giftContent()* function that takes in input the address of the customer which is the recipient of the gift (the address of the customer b) and the identifier of the item (in *Remix*, this field has to be filled with the identifier surrounded by the double quotation marks) so that the customer a pays for the chosen item on behalf of the customer b, afterwards the customer a guarantees the access to the customer b and the `CatalogContract` adds the selected item in the list of gifts of the customer b; this function creates a payable transaction between the customer a and the `CatalogContract`;

$t+8$ the customer b can call the view function *getCustomerGiftList()* that shows the gift list length, in this way he knows whether he has received presents;

$t+9$ the customer b calls the *getFromAGift()* function that creates a not payable transaction with the `CatalogContract`; it updates the contents and gifts arrays by, respectively, adding and deleting the item. This can be observed by using the view functions *getCustomerContentList()* and *getCustomerGiftList()* that return respectively a 1 and a 0 because the gift has been moved from the gifts array into the contents array;

$t+10$ the customer b calls the *consumeContent()* function that creates a not payable transaction with the `CatalogContract`. This function calls the *consumeContent()* function of the `ContentManagementContract` of the item and it firstly requires that the access to the item is granted and then it updates the view count.

3.3 Standard case - publish&publish&

In this case, two different trivial situations has been analysed: an author publishes two different items and the customer buys or gifts the newest one. In more detail:

t the `CatalogContract` is deployed;

$t+1$ the author deploys the `Content1ManagementContract`, with a given cost;

$t+2$ the author publishes an item i_1 by using the *addContent()* function that creates a transaction between the `Content1ManagementContract` and the `CatalogContract`; the author specifies the address of the `CatalogContract` and the `CatalogContract` updates all the informations relative of the publishing author, together with the genre information which is updated with the item's genre and the last item published by the author which is updated with this one;

$t+3$ the author deploys the `Content2ManagementContract`, with a given cost;

$t+4$ the author publishes another item i_2 by using the *addContent()* function that creates a transaction between the `Content2ManagementContract` and the `CatalogContract`; again, the author specifies the address of the `CatalogContract` and the `CatalogContract` updates all the informations relative of the publishing author, together with the genre information which is updated with the item's genre and the last item published by the author which is updated with this one;

$t+5$ the customer a registers by using the *registerAsCustomer()* function of the `CatalogContract` that will create a not payable transaction between the customer a and the `CatalogContract`;

$t+6$ the customers a uses the view function *getAuthorList()* for obtaining the identifier of the authors, actually, there will be only one available identifier;

$t+7$ the customer a calls the *getLatestByAuthor()* view function that takes in input the author's identifier and returns i_2;

$t+8$ the customer a gives in input the item's identifier to the *getCostOfContent()* view function and it returns the cost of i_2;

3.3.1 buy_latest

In this case, the situation ends within two other steps:

$t+9$ the customer a calls the *getContent()* function that creates a payable transaction between the customer a and the `CatalogContract`, in which the value of the message has to be equal to the cost obtained by the previous call to the *getCostOfContent()*. As explained in the first trivial case, this function also considers the redistribution of the feeds, but in this case, there are no other transactions and so the limit of occurrences is not reached;

$t+10$ the customer a has the content and can decide to consume it, by calling the *consumeContent()* function that creates a not payable transaction with the `CatalogContract`. Afterwards, it calls the *consumeContent()* function of the `ContentManagementContract` of the item, requiring that the customers has the access right for that item, and then it updates the view count of that item.

3.3.2 gift_latest

In this case, we have that the customer b registers and the customer a makes a present to the customer b. More in detail:

$t + 9$ the customer b registers by using the *registerAsCustomer()* function of the `CatalogContract` that will create a not payable transaction between the customer b and the `CatalogContract`;

$t + 10$ the customer a by using the view function *getCustomerList()* obtains the address of the customer b, recipient of the present he wants to make;

$t + 11$ the customer a calls the *giftContent()* function that takes in input the address of the customer b and the identifier of the item so that the customer a makes a present with the chosen item to the customer b, afterwards the customer a guarantees the access to the customer b and the `CatalogContract` adds the selected item in the list of gifts of the customer b; this function creates a payable transaction between the customer a and the `CatalogContract`;

$t + 12$ the customer b can call the view function *getCustomerGiftList()* that shows the gift list length, in this way he knows whether he has received presents;

$t + 13$ the customer b calls the *getFromAGift()* function that creates a not payable transaction with the `CatalogContract`; it updates the contents and gifts arrays by, respectively, adding and deleting the item. This can be observed by using the view functions *getCustomerContentList()* and *getCustomerGiftList()* that return respectively a $+1$ and a -1 resulting values because the gift has been moved from the gifts array into the contents array;

$t + 14$ the customer b calls the *consumeContent()* function that creates a not payable transaction with the `CatalogContract`. This function calls the *consumeContent()* function of the `ContentManagementContract` of the item and it firstly requires that the access to the item is granted and then it updates the view count.

3.4 Premium case - publish&

This situation is too similar to the previous one; the only difference is in the buying and selling of premium accounts. Two different situations has been analysed: an author publishes an item and the customer a, in the first case, buys premium account and gets the item published by the author; in the second case, another customer, called b, registers and the customer a gives the premium account to b. In more detail:

t the `CatalogContract` is deployed;

$t+1$ the author deploys the `ContentManagementContract`, with a given cost;

$t+2$ the author publishes an item by using the *addContent()* function that creates a transaction between the `ContentManagementContract` and the `CatalogContract`; here, the author has to specify the address of the `CatalogContract` and the `CatalogContract` updates all the informations relative of the publishing author, together with the genre information which is updated with the item's genre and the last item published by the author which is updated with this one;

$t+3$ the customer a registers by using the *registerAsCustomer()* function of the `CatalogContract` that creates a not payable transaction between the customer a and the `CatalogContract`;

$t+4$ the customer a obtains the item's name by using the view function *getContentList()* of the `CatalogContract`;

3.4.1 buy_premium&get_item

The first sub-case is the one in which the customer a buys the premium account and gets the item published by the author. In detail:

$t+5$ the customer a uses the *buyPremium()* function that creates a payable transaction between the customer a and the `CatalogContract`. It is executed only whether the customer is registered and is not already a premium user and whether the value in the message is the same of the `premiumCost` variable; as said before, the function keeps track of the premium duration by saving the block number of the transaction, so that a limited number of transactions is guaranteed; the function triggers the `boughtPremium` event already described before; gathers the cost of the transaction and ends;

$t+6$ the customer a uses the *getContentPremium()* function that takes in input the content to access and it creates a not payable transaction between the customer a and the `CatalogContract`. This function can be called only by premium customers and checks inside whether this is the last transaction that the customer can make. Afterwards, it guarantees the access to the content and updates the maximum views for author and for genre data structures;

$t+7$ the customer a consume the content by using the *consumeContent()* function that creates a not payable transaction between himself and the `CatalogContract` and it calls the *consumeContent()* of the `ContentManagementContract` of the item that updates the view count, after being sure that the user has the right access.

3.4.2 gift_premium&get_item

The second sub-case is the one in which another customer, called b, registers and receives a premium account present by the customer a. Afterwards, b gets the item published previously by the author. In more detail:

$t + 5$ the customer b registers by using the *registerAsCustomer()* function of the `CatalogContract` that creates a not payable transaction between the customer b and the `CatalogContract`;

$t + 6$ the customer a uses the *getCustomerList()* that can be called only by registered customers and that returns the address of all the customers;

$t + 7$ the customer a gives the address of the customer b as input for the *giftPremium()* function as well as the premium cost. This function creates a payable transaction between the customer a and the `CatalogContract` and emits a `giftPremiumSent` event;

$t + 8$ the customer a then checks whether the customer b results to have a premium account, by using the view function *isPremium()*;

$t + 9$ the customer b afterwards tries to get a content by using the *getContent()* function, but his request is going to be rejected because he has a premium account;

$t + 10$ the customer b then uses the *getContentPremium()* to get the content to which he's interested; the function creates a not payable transaction between the customer b and the `CatalogContract`. It guarantees the access to the content and the update of the maximum views for authors and for genres;

$t + 11$ the customer b consume the content by using the *consumeContent()* function that creates a not payable transaction between b himself and the `CatalogContract` and through the call of the *consumeContent()* of the `ContentManagementContract` of the item, it checks the right access and, in case, it updates the view count.

3.5 Premium case - publish&buy_premium&gift_item

The following situation may occur when a customer want to avoid to pay for an item and so he tries by using the buy-premium-&-gift-item trick (hoping that for when a premium account is active it doesn't pay not even the gifts). This situation has been handled by paying the gift to the premium account. In more detail:

t the `CatalogContract` is deployed;

$t+1$ the author deploys the `ContentManagementContract`, with a given cost;

$t+2$ the author publishes an item by using the *addContent()* function that creates a transaction between the `ContentManagementContract` and the `CatalogContract`; here, the author specifies the `CatalogContract`'s address and such contract updates all the informations of the publishing author, together with the genre information which gets updated with the item's genre and the last item published by the author which gets updated with this one;

$t+3$ the customer a registers by using the *registerAsCustomer()* function of the `CatalogContract` that creates a not payable transaction between the customer a and the `CatalogContract`;

$t+4$ the customer a obtains the item's name by using the view function *getContentList()* of the `CatalogContract`;

$t+5$ the customer a uses the *buyPremium()* function that creates a payable transaction between himself and the `CatalogContract`. It is executed only whether the customer is registered and is not already a premium user and whether the value in the message is the same of the `premiumCost` variable; the function keeps track of the premium duration by saving the block's number of the transaction, so that a limited number of transactions is guaranteed; the function triggers the `boughtPremium` event and it gathers the cost of the transaction before ending;

$t+6$ the customer b registers by using the *registerAsCustomer()* function of the `CatalogContract` that will create a not payable transaction between the customer b and the `CatalogContract`;

$t+7$ the customer a chooses b as recipient of the gift by using the view function *getCustomerList()* and obtaining the address of b;

$t+8$ the customer a calls the *giftContent()* function that takes in input the address of the customer b and the identifier of the item so that the customer a makes a present with the chosen item to the customer b, afterwards the customer a guarantees the access to the customer b and the `CatalogContract` adds the selected item in the list of gifts of the customer b; this function creates a payable transaction between the customer a and the `CatalogContract`;

$t+9$ the customer b can call the view function *getCustomerGiftList()* that shows the gifts' list length, in this way he knows whether he has received presents;

$t+10$ the customer b calls the *getFromAGift()* function that creates a not payable transaction with the `CatalogContract`; it updates the contents and gifts arrays by,

respectively, adding and deleting the item. This can be observed by using the view functions *getCustomerContentList()* and *getCustomerGiftList()* that return respectively a $+1$ and a -1 resulting values because the gift has been moved from the gifts array into the contents array;

$t + 11$ the customer b calls the *consumeContent()* function that creates a not payable transaction with the `CatalogContract`. This function calls the *consumeContent()* function of the `ContentManagementContract` of the item and it firstly requires that the access to the item is granted and then it updates the view count.

3.6 Common ending

For all the lists of operations (in which there is also a bit of description of what happens after the functions have been called), the common ending is the closing of the contract by using the *closeContract()* function, that can be called only by the owner of the `CatalogContract`. This function provides to redistribute all the ether that has been collected by the `CatalogContract` fairly to the authors, by using the same function triggered after v views (where in the test cases $v = 2$).

4 Gas estimation of non-trivial operations

The estimation of non trivial operations has been achieved by analysing some of the functions that belong in the previous list of operations section. The cost paid is the transaction cost that includes both the execution cost and the cost of sending data to the blockchain, so the results shown below take into account only the transaction cost.

4.1 *registerAsCustomer()* gas estimation

In this function, `customers[msg.sender]` is created together with `addressCustomers[noCustomers]`, so, according to [1], the cost is 2 $\mathcal{G}_{sset}$ so in this case, it corresponds to $2 \times 20.000 = 40.000$; after that, the cost of all the settings of the fields of the `customers[msg.sender]` can be approximated by 4 $\mathcal{G}_{sreset}$, so: $4 \times 5.000 = 20.000$. This means that the execution cost is about $40.000 + 20.000 = 60.000$ gas spent.

Besides, the function creates a transaction after the *Homestead* transition, so the cost is about $\mathcal{G}_{txcreate} = 21.000$; this means that the transaction cost is about $60.000 + 21.000 = 81.000$. This theoretical approximation can be proved by applying the function *registerAsCustomer()* and seeing that the transactional cost obtained in this way is about 83.800 whilst the execution cost is about 62.500; this means that the approximation computed above it's a lower approximation of the actual cost, because it does not take into account also the gas spent for the contract loading, for finding the correct function to execute and all the intermediate steps that cause something more in the gas value spent.

The first time, actually, the gas spent is 15.000 more than the showed one, that is, about 98.800. The reason for this difference between the first and the second execution of the same function is due to the storage cleaning operation done initially and not redone after. The extra gas spent for the storage cleaning operation is refunded when the transaction is going to be closed, for this reason there is a difference between the first and the second execution.

4.2 *getContent()* gas estimation

For this function, the first time it's executed it costs about 140.500, but the second time about 110.500. This 30.000 missing gas is due as before to the storage cleaning operation done two times, for the execution of both the `updateMaxViewsForAuthor` and the `updateMaxViewsForGenre`. The cost of 110.500 can be explained by going to analyse all the instructions of the function: the first cost to consider, there is the creation of the transaction, that costs 21.000; after that, there is a $\mathcal{G}_{sset}$ in the *getContent()* function (cost: 20.000) when the content is actually added into the array of the contents belonging to the customer. Furthermore, there is a $\mathcal{G}_{sset}$ and a possible $\mathcal{G}_{sreset}$ in `updateMaxViewsForAuthor` (with a cost, resp., of 20.000 and 10.000) and a $\mathcal{G}_{sset}$ in `updateMaxViewsForGenre` (with a cost of 20.000); this means that the total cost of the update functions is the sum of all of them, that is, 50.000.

Summing up, the total cost of gas spent is:

$$21.000 + 20.000 + 50.000 = 91.000$$

In the ending part of the function the item's function *grantAccess()* is called, in which there is a *sset*, so, this means that the cost from 91.000 becomes

$$90.000 + 20.000 = 111.000$$

and as before it is very close to the actual one (that in this case is 110.500).

References

[1] Ethereum: A secure decentralised generalised transaction ledger Byzantium version e94ebda - 2018-06-05, `https://ethereum.github.io/yellowpaper/paper.pdf`