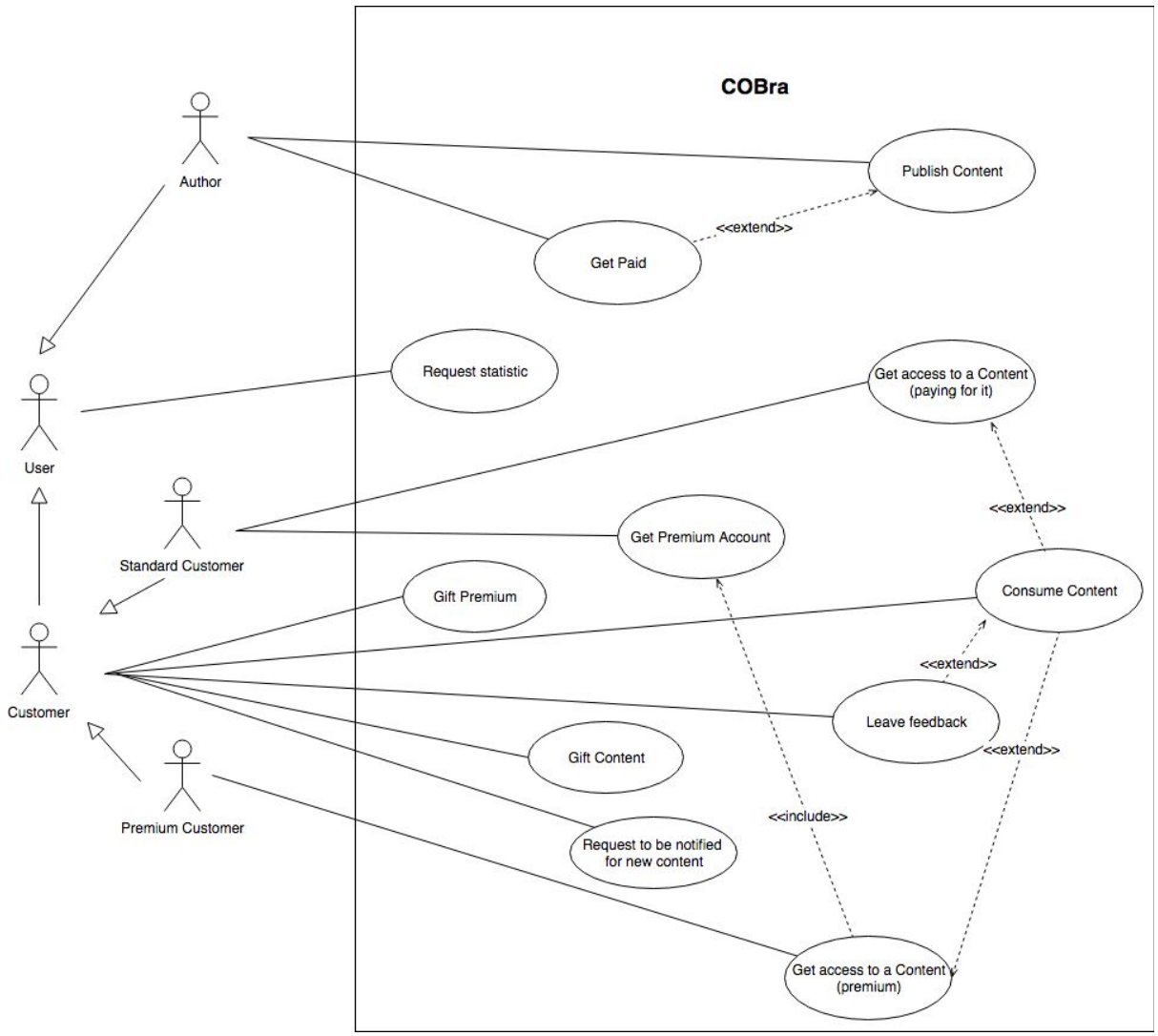
Peer to Peer System and Blockchain Final Project

COBrA Dapp Report

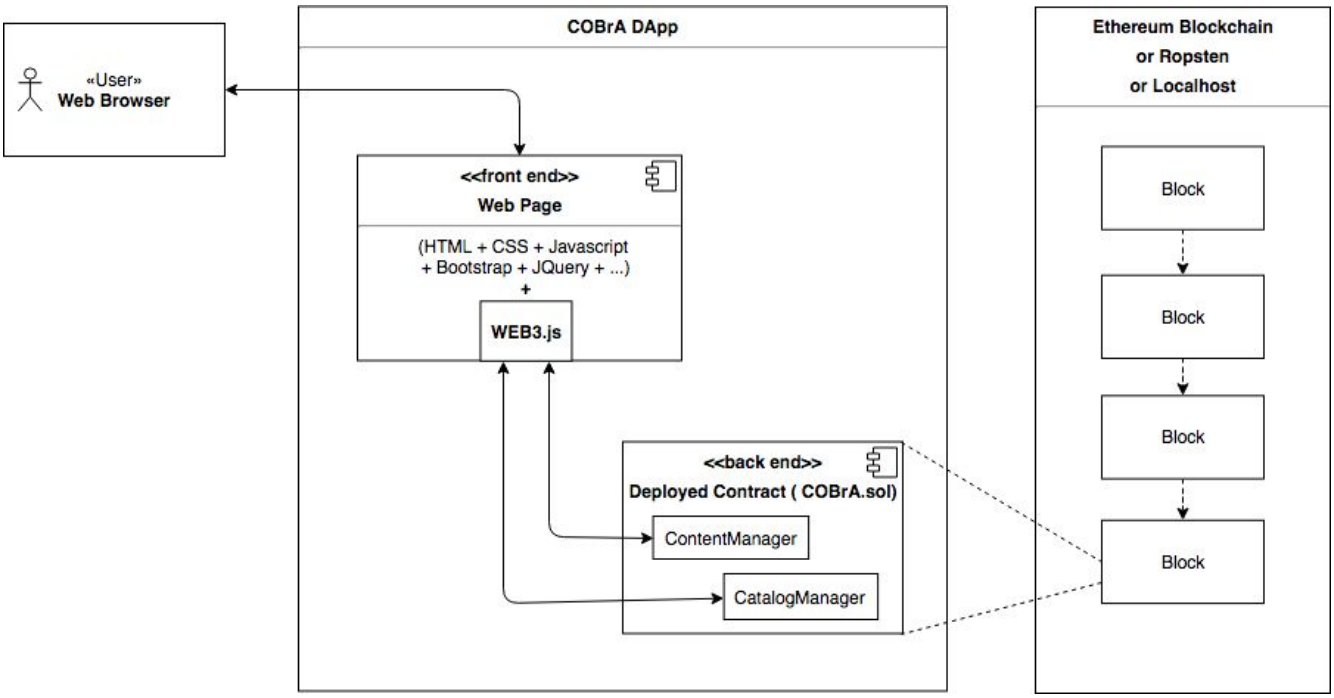
Antonio Boffa 520931

COBra Dapp:

Use case diagramm :



Organizzazione generale:



Come si può notare dal diagramma (non UML e non tecnicamente preciso) qui sopra la Dapp sviluppata prevede per quanto riguarda il front end una pagina web con la quale l'utente può interagire. È stata usata la libreria Javascript web3.js (versione 1.0.0 beta 33) per interagire con un nodo Ehtereum attraverso RPC, e quindi permette di inviare transazioni ai Contract Account. Questa libreria, essendo ancora in sviluppo presenta diversi problemi su alcune funzionalità, nonostante ciò questa implementazione di COBrA è funzionante e rispetta le specifiche. Si è cercato di seguire il più possibile i principi per un buon design delle Dapp (https://medium.com/@lyricalpolymath/web3-design-principles-f21db2f240c1).
Per quanto riguarda il backend vi sono due contratti: CatalogSmartContract.sol e ContentManagementContract.sol.

Scelte principali:

Dalla traccia del progetto e dal buon senso è chiaro che:

- ➔ il Catalog deve gestire:
  - i pagamenti per i conenuti
  - i pagamenti per diventare Premium
  - i pagamenti verso gli artisti
  - il calcolo delle statistiche

➔ il Content Manager deve:

- contenere i dati relativi ad un singolo contenuto

È stato lasciato allo studente il compito di assegnare le altre competenze necessarie quali:

- i dati relativi ai diritti che gli utenti hanno dei contenuti
- i dati aggiuntivi sui contenuti (data di publicazione, visualizzazioni, feedback ecc…)

Per assegnare questi compiti alle entità del sistema (Catalog e Content Manager) è stato considerato maggiormente il principio della equa distribuzione di essi in modo da non sovraccaricarne nessuno. Quindi le competenze sopra elencate sono state assegnate al ContenManager. L'utente dell'applicazione comunicherà con entrambi i contratti e modo molto trasparente. Questa implementazione di COBrA è quindi organizzata nel seguente modo:
Il Catalog ha il compito di :

- gestire i pagamenti
- contenere solo gli indirizzi e i nomi dei Content e nient'altro
- essere un intermediario tra gli utenti e i contenuti
- richiedere dati per le statisciche
- notificare all'utente i vari eventi

Il ContentManager ha i compiti di:

- contentere le autorizzazione per accedere ai contenuti e per recensire i contenuti
- contenere altre informazioni aggiuntive (date, visualizzazioni, feedback ecc…)
- fornire i contenuti agli utenti che lo richiedono e che ne hanno diritto
- registrare i feedback degli utenti che hanno i diritti per lasciarli
- fornire i giusti dati al Catalog per le statistiche

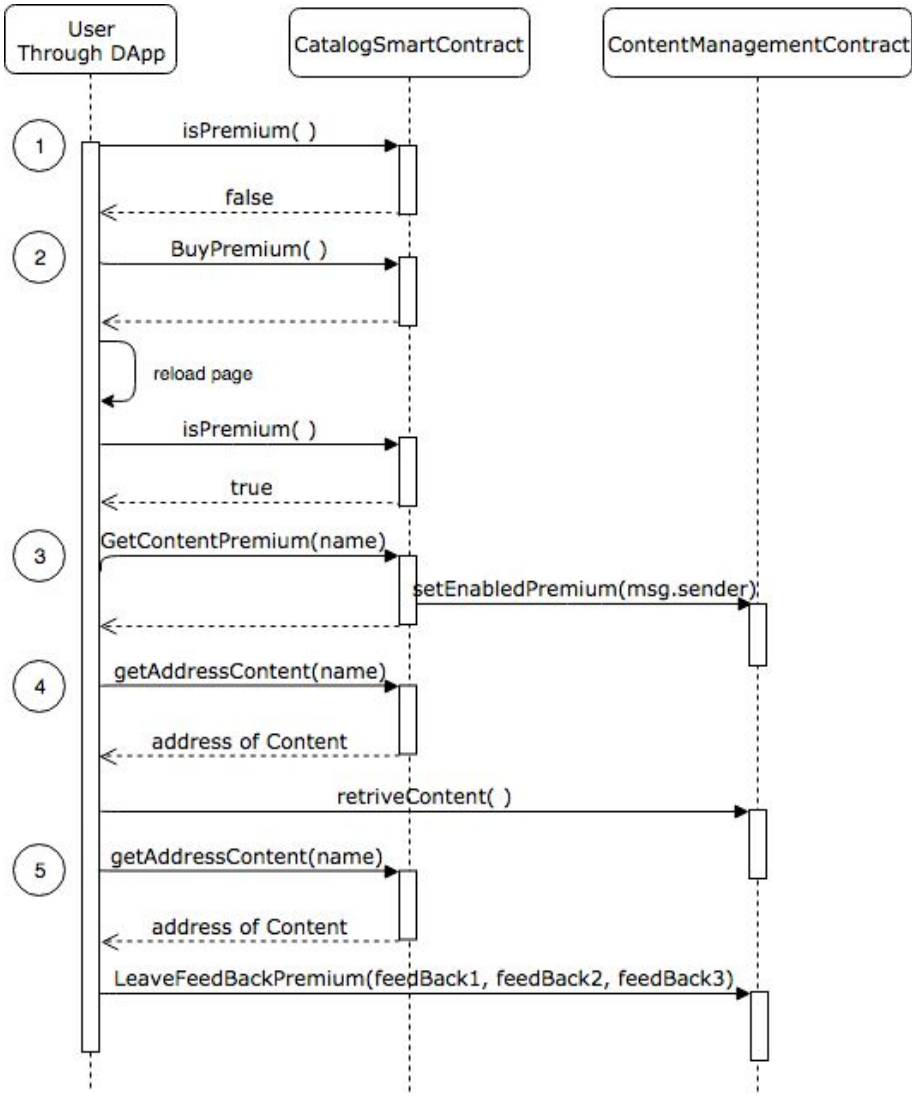
La scelta di far mantenere al Catalog anche una mappa nome – indirizzo del contenuto è dovuta al seguente fatto: gli Utenti molto spesso cercheranno i contenuto in base al loro nome, sia se per esempio vogliono ottenere l'accesso al contenuto o sia se vogliono regalarlo, e viene garantito loro un basso consumo di Gas. Infatti la ricerca del giusto indirizzo del ContentManagementContract associato al nome sarà l'immediata ricerca in una mappa.

Per avere l'unicità di nome dei contenuti e anche l'unicità dei nomi degli artisti si è scelto di appesantire la funzione di creazione di una nuovo contenuto con una ricerca che coinvolge tutti gli altri contenuti. Ciò avvienre per due motivi : poiché l'unicità suddetta è molto importante per avere un sistema serio di publicazione di contenuti; e perché scoraggia (dato il costo in gas) eventuali utenti malevoli che vogliono sfruttare il sistema a publicare contenuti spam, ovvero contenuti di bassa qualità che nessuno mai visualizzerà.

Scelte specifiche attraverso esempi e diagrammi UML :

In questa sezione del report si espone il funzionamento di alcune operazioni critiche della Dapp. Per capire il resto delle funzionalità e delle scelte implementative si rimanda al codice.

Un utente diventa premium, ottiene un contenuto e rilascia un feedback:



(1) Al suo caricamento, la pagina web in automatico chiama il metodo view isPremium del Catalog per sapere se l'utente che sta servendo è premium o meno in modo da organizzare la UI nel modo giusto.

(2) Nell'esempio un utente non premium richiede di diventare premium premendo l'apposito bottone e la Dapp chiama il metodo BuyPremium del Catalog, al suo completamente con esito positivo (l'utente ha abbastanza Ether) la pagina si ricarica e l'utente avrà davanti a se la UI per gli

utenti premium. Esso ha quindi speso 1 Ether (costo per diventare premium) e lo sarà per il "tempo" di 10 blocchi.
(3) Esso decide poi di ottene l'accesso ad un certo contenuto X, allore egli inserisce il nome di X nel giusto form, conferma e la Dapp chiama il metodo GetContentPremium del Catalog che durante la sua esecuzione ricerca l'indirizzo del ContentManagementContract e chiama il suo metodo setEnabled, notificando così al Content che l'utente in questione ha il permesso di accedere al contenuto.
In realtà vi sono due versioni del metodo setEnabled, una per gli utenti Premium e una per i Standard, ciò perché ci sono due diverse strutture dati che mantengono le informazioni degli utenti, come si vede nella seguente figura:

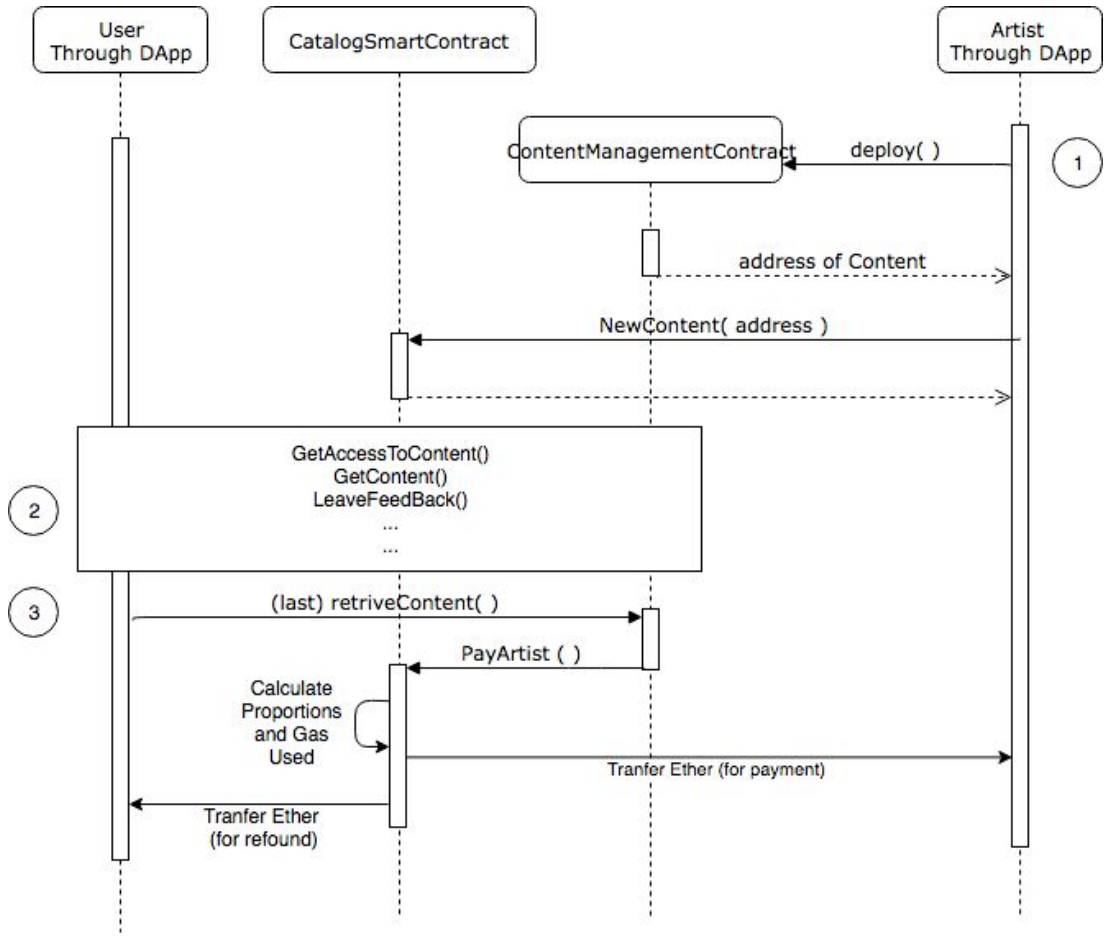
```
struct StandardRight{
    bool isAuthorized;
    bool canLeaveAFeedBack;
}

struct PremiumRight{
    bool isAuthorized;
    uint lastblockvalid;
    bool canLeaveAFeedBack;
}
mapping (address => PremiumRight) public AuthorizedPremiumCustomers;
mapping (address => StandardRight) public AuthorizedStandardCustomers;
```

Gli utenti standard posso solo essere autorizzati a ottenere un contenuto o essere autorizzati a lasciare un feeback, mentre i premium hanno anche una "scadenza", infatti nel campo lastblockvalid c'è il numero dell'ultimo blocco nel quale gli utenti in questione sono considerati premium. Gli utenti premium ai quali è scaduto l'abbonamento perderanno ogni diritto aquisito sui contenuti. Ovvero se un utente ha ottenuto la possibilità di accedere ad un certo numero di contenuti, una volta scaduto l'abbonamento egli non potrà visualizzarli, ma dovrà o pagare per ottenerli singolarmente o ridiventare premium.
(4) L'utente ora ha la possibilà di inserire nel giusto form il nome del contenuto X e quindi confermare la volontà di ottenerlo, quindi la Dapp chiama il metodo view del Catalog getAddressContent(X) per ottenere l'indirizzo del Content e chiama il suo metodo retriveContent( ).
Ne segue dal precedente discorso sulle due diverse strutture dati che vi sono due implementazioni del metodo retriveContent, una per gli utenti Premium e una per i Standard, da notare che la seconda aumenta il contatore delle visualizzazioni, e ha la possibiltà di lanciare il pagamento agli artisti. Nel prossimo esempio si parla proprio di questo.
Il Content in questione quindi aggiorna il suo stato interno per memorizzare che l'utente ha ottenuto il contenuto, e che ora egli può rilasciare un feedback.

(5) L'utente infine vuole recensire il contenuto, inserisce il nome di X nel giusto form e inserisce tre interi compresi tra 1 e 5 (modificando la posizione di tre appositi slider) che rappresentano numericamente quanto egli ha apprezzato il contenuto, l'equità del prezzo e l'originalità del contenuto. Egli conferma e la Dapp, come prima, chiama il metodo view del Catalog getAddressContent(X) per per ottenere l'indirizzo del Content e chiama il suo metodo LeaveFeedBackPremium. Anche in questo caso si hanno metodi diversi per operare su strutture dati diverse.

Un artista crea un contenuto e viene pagato:

Come detto nelle specifiche di questo progetto ogni qualvolta un contenuto raggiunge un certo numero *n* di visualizzazioni l'artista viene ricompensato con una quantitativo di Ether proporionale al prezzo da lui deciso, ad *n* e al rapporto tra la sua media di feeback e quella ricevuta dal contenuto con le migliori recensioni. In questa implementazione si è scelto $n = 3$ (quindi un valore basso per motivi di debug). Per quanto riguarda queste funzionalità, ecco un diagramma UML:



(1) Un utente che vuole publicare un contenuto (nel grafico chiamato Artist) deve compilare il giusto form, aggiungere nome, genere, il suo nome da autore e prezzo. Poi deve selezionare il contratto da usare, se sceglie di usare quello di defualt può benissimo selezionare il file ContentManagementContract.json fornitogli. Se ne vuole usare un altro deve scrivere uno smart contract che estende ContentManagementContract e complilarlo con:
solc <nome>.sol --combined-json abi,asm,ast,bin,bin-runtime,clone bin,devdoc,interface,opcodes,srcmap,srcmap-runtime,userdoc > <name>.json
(NB: Il nome del file .sol deve avere lo stesso del contratto che vi si sta definendo dentro)
L'artista deve poi confermare e in automatico la Dapp crea il contratto, fa il depoy, e chiama il metodo NewContent( ) del Catalog in modo da avvisarlo del nouvo contenuto. Il Catalog prima di accettare il nuovo contenuto (come preannunciato prima nella sezione Scelte Generali) controlla che non ve ne siano già con lo stesso nome, e se esistono già contratti con lo stesso nome da artista controlla che sia sempre lo stesso indirizzo a publicarli (se così non fosse, qualcuno sta cercando di appropriarsi il nome da artista che qualcuno in precedenza aveva già scelto). Se tutto ciò è andato a buon fine il contenuto sarà regolarmente publicato e accessibile agli utenti della Dapp.
(2)Nel tempo quindi i vari utenti richiederanno diverse volte l'accesso al contenuto in questione, lo otterranno, lo rencensiranno, ecc… .
Si chiarisce che con: "il contenuto ha raggiunto la soglia giusta di visualizzazioni per procedere al pagamento dell'artista" si intende quando *views % n == 0* (questo calcolo non viene ovviamente fatto quanto view è zero). Da qui in poi ci si riferisce all'utente che rende vera la formula precedente con: "l'utente n-esimo" anche se non è una dicitura propriamente adeguata.
(3) Quando il contenuto raggiunge la suddetta soglia viene chiamto il metodo PayArtist( ) del catalog (durante l'esecuzione della retriveContent da parte dell'n-esimo utente). Questo metodo prima trova il contenuto con la media delle recensioni più alta poi calcola la proporzione tra media delle recensioni del contenuto in questione e quella del contenuto meglio recensito e procede al pagamento dell'artista che ha publicato il contenuto. Il metodo non finisce qui poiché per eseguirlo è stato usato il gas dell'n-esimo utente. Ciò non è giusto quindi si è scelto di rimborsarlo con la giusta quantità di Ether. Di ciò se ne discute nella prossima sezione.

Stima Gas usato da n-esimo utente:

Si può dividere il calcolo di quanto Ether rimborsare all'n-esimo utente in due parti, una variabile e una no. La parte variabile viene calcolata a runtime, ovvero il metodo retriveContent( ) del ContentManagementContract, appena entra nel blocco di codice necessario a pagare l'artista, registra il gas rimasto alla transazione, e passa questo numero al metodo PayArtist( ) che una volta eseguito pagato l'artista, registra il gas rimasto alla transazione e può sapere quanto gas è stato speso. La parte fissa è quella relativa al semplice rimborso all'utente, che ho precisamente calcolato

essere pari a 7910 gas. Le due parti vengono sommate e viene tranferito dal bilancio dell catalogo al bilancio dell'n-esimo utente una quantità Ether pari a : gas usato * prezzo del gas da lui deciso.

Gestione notifiche:

Un utente viene notificanto quando:

1. gli viene regalato un contenuto
2. gli viene regalato un abbonamento premium
3. se ne fanno richiesta, alla publicazione di nuovi contenuti di un particolare genere e/o di un particolare artista.
4. ha la possibilità di rilasciare un feedback

Per quanto riguarda le notifiche 1 e 2, il client che serve un utente Standard è sempre in ascolto per gli eventi ContentAccessGifted e PremiumGifted in modo tale che se il regalo è prorpio rivolto all'utente che si sta servendo si possa lanciare una notifica.
Per quanto riguarda le notifiche 3, quando l'utente aggiunge un autore o un genere di cui vuole essere notificato, lo aggiunge ad un array interno (followingGenre o followingArtists) e rimane in ascolto dell'evento NewContentEvent in modo tale che se l'autore o il genere del nouvo contenuto sono uguali ad uno o più elementi degli array sopra citati allora viene lanciata una notifica.
Infine per quanto riguarda le notifiche 4, il catalog fa "l'emit" dell'evento ma in realtà la notifica arriva all'utente appena il metodo GetContent( ) si conclude con successo. Ciò poiché è ovvio che l'utente ha appena consumato il contenuto e quindi può lasciare il feedback.

Chiusura del catalog:

La chiusura del catalogo è permessa solo a colui che lo ha creato. Egli può chiamare il metodo close( ), esso calcola la giusta percentuale di Ether disponibile da dare ad ogni artista (in base alle visualizzazioni ottenute dai contenuti), procede nel trasferire queste somme di Ether, chiama il metodo close( ) di ogni contenuto (si esegue solo content.selfdestruct( )) ed infine chiama su se stesso selfdestruct( ).

Deploy su localhost:

Nello sviluppare e testare questa Dapp è stata utilizzata una blockchain personale fornita dal programma Ganache. È stato usato Remix per il deploy del contratto Catalog su di essa. In questa configurazione la Dapp funziona perfettamente poiché si utilizza il protocollo websocket, infatti all'inizio nel codice della pagine web si può notare:
var web3 = new Web3(new Web3.providers.WebsocketProvider('ws://localhost:8545/'));

Così dovrebbe funzionare anche con geth ma come si vede nella prossima sezione non mi è stato possibile testare.

Deploy su Ropsten:

Per quanto riguarda il deploy su una vera blockchain, è stato utilizzato il programma Parity per ottenere un nodo di Ethrereum funzionante colegato alla rete Ropsten, per sincronizzarsi ha impiegato una sola notte. Non è stato utilizzzato geth –testnet –syncmode “fast” … poiché avrebbe impiegato troppo tempo.
Per far sincronizzare parity è stato utilizzato il comando:
parity --chain ropsten --mode active --tracing off --pruning fast --db-compaction ssd --cache-size 2048
Per utilizzare il nodo sincronizzato è stato rilanciato parity con il comando:
parity --chain ropsten --ws-interface all --ws-apis web3 --jsonrpc-hosts all --ws-origins all --ws-hosts all --jsonrpc-interface all --jsonrpc-cors null
La scelta di utilizzare Parity (v1.11.10-20180830 unstable) però ha portato lo svantaggio di poter utilizzare solo il protocollo http (ha problemi con websocket) quindi in questo caso nella pagina web bisogna utilizzare questo comando:
var web3 = new Web3(new Web3.providers.HttpProvider(‘http://localhost:8545/’));
e ciò porta a dei problemi con web3 che in versione 1.0.0 ha deprecato *HttpProvider.*
In particolare, in questa configurazione, ci sono problemi con la ricezione degli eventi, ma tolto ciò la Dapp funziona perfettamente.