Development of a Đapp for COBrA

Peer to Peer Systems and Blockchains

Master's Degree in Computer Science and Networking
Academic Year 2017/2018

ORLANDO LEOMBRUNI

17-01-2018

Contents

1. Introduction

This project concerns the development of a complete Đapp for a fair content trading system.

The Solidity backend was first implemented as an assignment for the second midterm (end-term); in this project, the existing Solidity contracts are expanded as requested and slightly tweaked for their use in the frontend.

The application frontend has been instead fully implemented for this project and features a responsive webapp that can work using a multitude of Ethereum clients.

1.1. Content of the submission

The submission `zip` file contains the project source files and a number of bookkeeping entities generated by various tools, necessary to run or build the application.

Source files

The `src` folder contains all relevant source files, i.e. the ones written by the student:

- the `asset` folder contains the media assets, in this case two versions of the COBrA kai logo;
- the `styles` folder contains files that define the graphical style of the app:
  - `global.css` is a global stylesheet for the application;
  - `login.css` is a specific stylesheet for the login page;
  - `MaterialCustomStyles.js` is a collection of Material UI style objects, one for each Component;
- the `solidity` folder contains the Solidity contracts, both in source and compiled formats:
  - `compiled.json` contains both contracts in their compiled (ABI + BIN) form;
  - `Catalog.sol` contains the Solidity contract for the Catalog;
  - `ContentManagementBase.sol` contains the Solidity contract for managing a single Content;
- the `components` folder contains the React Components that make up the webapp:
  - `App.js` is a "wrapper" component for the entire application; as such, it manages the connection to a Web3 instance and shows to the user the desired interface (Consumer or Creator);
  - `AppDrawer.js` is a Material UI component for a top bar and left-hand side drawer menu;
  - `BuyGiftDialog.js` is a component that shows a dialog to allow consumers to buy or gift a Content;
  - `BuyGiftPremiumDialog.js` is a component that shows a dialog to allow consumers to buy or gift a Premium subscription (or extending an existing one);

- `CatalogManager.js` is a simple page that allows to create or close Catalogs;
- `Charts.js` is a component that shows various information about trending (top rated) Contents;
- `ContentList.js` is a component that shows a list of Contents, allowing the user to perform one or two actions on a desired Content (e.g. buy it);
- `CreatorApp.js` is the main portion of the app dedicated to creators; it allows them to create new ContentManagementBase contracts and publish them to a Catalog, request monetization or close own ContentManagementBase contracts;
- `FeedbackForm.js` is a form to ask feedback from an user about a consumed Content;
- `LoginPage.js` manages the login (unlocking of EOA) if needed;
- `NewContentForm.js` is a form used by creators to input the data for a new Content contract;
- `TransactionConfirmation.js` is a simple dialog popup shown when the user is about to execute a transaction, and sums up the transaction's info (expected gas consumption, total gas cost and EOA balance);
- `UserApp.js` is the main portion of the app dedicated to consumers; it allows to buy and gift Contents or Premium subscription, to consume bought Content and leave feedback for them;
- `Web3LoginForm.js` is a form to unlock EOAs using Web3, if needed.

Also, the `report` folder contains this report in both PDF and Markdown formats.

Bookkeeping files

- The `package.json` and `package-lock.json` files, created by the `create-react-app` and used by the Node.js system to manage dependencies;
- the `build` folder, which contains the compiled application ready to be run in a browser.

2. The Solidity backend

The Solidity contracts' implementation has been already discussed in the report of the endterm project; the main implementation choices remain the same.

2.1 The rating system

In accordance with the assignment, a rating system has been implemented into COBrA: after consuming a Content, a customer can leave feedback about their experience. In particular they will be asked three

questions, to which the user can answer with a number between 0 (worst) and 5 (best)[1]:

1. How much did you enjoy the Content? (shorthand: `appreciation`)
2. Do you think that the price was fair? (shorthand: `fairness`)
3. How likely are you to recommend this Content to others? (shorthand: `suggest`)

The Catalog proceeds to re-calculate the average for each category every time a feedback is sent. This is needed for a fair payment system: in the previous version of COBrA, creators were paid a sum that only depended on the number of views for that Content (i.e. `amount = content.views * content.price`), whereas now the Content's rating affects the amount of ether paid by the system to the creator (the formula is now `amount = content.views * content.price * (content.avg_rating/5)` since 5 is the max rating.).

Gaming the system

The rating system can be exploited by a malicious Creator to either increase its revenue (by increasing ratings of their own Content) or decrease competitors' revenue (by decreasing their ratings). It can be shown, however, that *without Premium subscriptions* this is not economically feasible.

To increase their own Content rating, a malicious Creator has to repeatedly buy that Content, consume it and leave feedback. This is automatically a losing game: each transaction costs some gwei in gas, and more importantly the payout is always less than or equal the sum paid for gaming the system (there's a factor ≤ 1 in the calculation of the amount).

A similar argument can be made for decreasing the competitors' rating. Frequent purchases and consumption of a Content (for the sake of leaving negative feedback) also mean that the competitor will receive equally frequent payouts, and if the Content's rating was high this effectively means that the first payouts are close to the Content's price (the factor is near 1).

However, Premium subscriptions allow for a basically infinite amount of "free" feedback leaving that will never cause a payout. In the current system this is very effective for rigging the ratings. There are some steps that can help prevent this issue:

- increase the Premium subscription cost to the point that it's not feasible anymore to purchase it only for the sake of rigging the system (i.e. the malicious Creator will always lose money)
- modify the rating system to completely separate "Premium" feedback from "regular purchases" feedback. Both types will be considered in a "global" rating for charts and queries by Consumers, but only "regular purchases" feedback will be considered when paying a Creator.

[1] This number is actually multiplied by 100 in the `Catalog` contract in order to avoid working with floating-point numbers.

2.2 Events

In order to let the frontend know about the effects of transactions, Solidity contracts need to emit Events. The assignment requested three types of events:

- Feedback activation (event `CanLeaveFeedback` of the `ContentManagementBase` contract);
- Access grant (event `GrantedAccess` of the `Catalog` contract);
- New content publication (event `NewContentAvailable` of the `Catalog` contract).

In addition to those events a few others were implemented, either because they were deemed necessary or because their implementation simplified bookkeeping in the frontend:

- `PaymentAvailable` is emitted by the `Catalog` wherever a Content has passed the view threshold for monetization;
- `GotPremium` is emitted by the `Catalog` when an user purchases (or is gifted with) a Premium subscription;
- `ProvideContent` is emitted by the `ContentManagementBase` contract and simulates the actual provisioning of the Content to an user.

2.3 Other

This section details some changes (with respect to the COBrA version that was implemented as the endterm project) that haven't been discussed in the above sections.

- The `MusicContent` contract (extension to `ContentManagementBase`) has been deemed redundant and it's not provided anymore. The endterm version *should* still work, but hasn't been tested.
- Both `Catalog` and `ContentManagementBase` contracts contain a dummy function (`isActiveCatalog` or `isActiveContent`) that always returns `true`, to allow the frontend to check whether a contract has been successfully deployed or closed.
- The `Catalog` now contains a `grantsAvailable` function that returns a consumer's list of currently-available grants (accesses to Contents), in order to avoid losing information when the consumer logs out of the webapp.
- The `Catalog` now contains a `getCreatorContentList` function that returns the list of Contents published by the requesting Creator.
- The `Catalog` now contains a `getPremiumPrice` function that returns the price of a Premium subscription (in wei).
- The `Catalog` now contains a `remove_content` function that effectively removes a Content from the Catalog and allows to close the Content contract.
- Creators can now set a price for their Contents, instead of being a set parameter of the Catalog.

- The `ContentManagementBase` contract now contains a `closeContract` function that allows the creator to remove it from the Catalog and close the associated contract.

3. The webapp frontend

The frontend portion of the application is called *COBrA kai* (where "kai" stands for *kai application interface*). It's a single-page webapp built in Javascript using mainly the *React* and *Material UI* frameworks; the focus was on easiness of use and a responsive design that could be used on a wide variety of screens and devices, even if this came at the cost of a increase in boilerplate code.

The application is logically divided in 3 parts:

- an *user* interface, from where consumers can buy and gift access to content or Premium subscriptions, consume content and leave feedback;
- a *creator* interface, from where creators can manage their contents (create new ones, publish them to the Catalog, request monetization, close them);
- a "hidden" *catalog manager* interface for easy creation and deletion of Catalog contracts. This can be reached by appending `/catalog` at the end of the webapp URL.

3.1 Frameworks and libraries

The project was implemented using the following technologies:

- Javascript (ECMAScript 6)
- React 16.6.0
- create-react-app 2.0.4
- Web3.js 1.0.0-beta.36
- Material UI 3.3
- Solidity 0.4.24
  - Please note that versions >= 0.5.0 are unsupported.

Other dependencies (various Javascript libraries used e.g. for notifications and handling of big numbers) are listed in the `package.json` file.

Testing and debugging of the frontend was done on the Google Chrome browser with the *Metamask* and *React Developer Tools* extensions installed. Development was done using WebStorm 2018.2.5 and Visual Studio Code, running on macOS 10.13.6 High Sierra.

3.2 Connecting to the Ethereum backend

COBrA kai requires a connection to a Web3 instance, which in turn manages the connection to the Ethereum network. Two types of Web3 connections are possible:

- "provided" Web3, which has to be connected to an Ethereum client in order to work (e.g. `geth`, `parity`, Infura nodes, …);
- "injected" Web3, which relies on the browser for a "client-less" connection to the Ethereum network (either via native support, like the Mist browser, or with extensions like Metamask on common browsers like Firefox and Chrome).

Moreover, there are three ways to perform transactions on behalf of an EOA:

1. use `Web3.personal` to unlock the account using its passphrase, then send transactions without having to explicitly sign them;
2. sign every transaction "offline" separately, using the EOA's passphrase every time;
3. leave all EOA and transaction signing management to the browser/extension.

"Provided" Web3 instances can only work with methods #1 and #2. "Injected" Web3, on the other hand, used to be exactly the same as "Provided" Web3 instances until November 2nd, 2018[2], when major privacy-conscious Ethereum browsers implemented a big change and implemented permissions to access user EOA/Wallet data; now they only work with method #3.

COBrA kai provides support for methods #3 and #1, in this order: if there's no support for method #3 (asking browser/extension for permission to use an EOA), then it falls back to method #1. Method #2 is not supported, since using it would mean either a) to ask the user the EOA passphrase at every transaction or b) to ask it once, when opening the webapp, and then store it somewhere in plain text in order to reuse it later. Neither solution is elegant and the second one could also lead to security issues.

Moreover, it can be argued that, from now on, method #3 should be the standard for widely distributed Đapps, while #1 is a well-documented fallback in case the user prefers to run its own Ethereum node, a scenario in which leaving the account unlocked for the duration of the utilization of the Đapp is not a security concern.

A note on the Parity client: `parity` supports the `personal` package of Web3, but will automatically re-lock the EOA after every transaction. This could be avoided by the `--unlock [EOA]` flag, but it hasn't been tested with COBrA.

[2] Breaking Change: No Accounts Exposed By Default, https://medium.com/metamask/https-medium-com-metamask-breaking-change-injecting-web3-7722797916a8

3.3 Online version

A fully-functional version of COBrA kai has been deployed and published on Heroku and can be found at https://cobrakai.herokuapp.com. Just like the local version (the one provided in the zip file) it can connect either to a "provided" Web3 instance or an "injected" one. This allows the webapp to be used, for example, on smartphones and tablets, simply by visiting the above URL with an Ethereum-enabled browser.

3.4 Local version

An optimized, minified build of the app is available in the submission's `build` folder; just open the `index.html` file in a browser. It's also possible to run the development version of the app using `npm install` and then `npm run start` inside the submission's main folder. Both local versions and online version is functionally equivalent, but the optimized local version is the fastest.

4. Tests, bugs, other considerations

Thorough tests have been conducted using the development version of the app and a local testnet created using `geth`. Unfortunately, I've been unable to run an Ethereum Ropsten Testnet node on my computer since the university's Internet access at the Student house where I live does not allow direct connections to peer-to-peer networks, causing the syncronization with other nodes to fail. Due to this unconvenience, all tests on Ropsten were made using the Chrome browser (via Metamask) on my computer or the *Status* Ethereum browser on my Android smartphone.

A deployed and active Catalog can be found on the Ropsten network at the address

`0x6Ec52ae221937408066e06dd2089882136eB89E3`

This can be input e.g. in the online version of COBrA kai for quick evaluation of functionalities, since it is pre-loaded with some Content. [3]

There's a bug that arises sporadically when using Metamask to connect to Ropsten: sometimes the frontend receives multiple copies of the same event (e.g `NewContentAvailable`), thus causing minor visual glitches sometimes. I could not reproduce this issue constantly, nor it seems to be present when connected through the `geth` client.

[3] For testing purposes, also a closed and inactive Catalog exists on the same network, at address `0x1B57d629C6dB795D746337750919bEc56376D8dC`.