FinalProject Peer2Peer Systems and blockchains

Development of a DAPP for Smart Auctions

Lorenzo Bellomo, 531423

Project Structure

The project is organized as shown in Figure 1, and the key points are:

- *Root folder*: It contains only the report file and a folder DAPP, all the code is in this folder.
- *DAPP folder*: It contains the configurations files needed (*bs-config.json* for lite-server, *package.json* for node, *truffle-config.json* for truffle). It additionaly holds file ropsten.json, which contains the private credentials for ropsten deployment. Then other folders are explained just in the next points.
- *contracts, migrations, test and build*: Truffle related folders. They are compulsory when dealing with truffle projects. Also, at this level, the build folder is generated.
- *node_modules and src*: Folders related to node and the front-end part. The *src* folder contains some *.css* files (bootstrap).
- *index.html*: It only contains the view for the top navbar in the web page, and the two buttons used to choose the auction to render.
- *views folder*: It contains *vickrey.html* and *dutch.html*, which are the files needed to render the auctions.
- *js/App.js*: This is the file that contains the logic for the initialization of web3, the rendering of the two auctions, the handling of all the callbacks and it also manages the interface towards the contracts.
  All the contract calls (setters, getters and events for both the auctions), are handled in this file.
  The file structure is organized in three parts. The top part contains the web3 initialization, which is common to both auctions. The middle part is the Vickrey auction related one. It contains a method (*listenForEventsV()*) for setting up the callbacks for this type of auction, another one for all the calls to the main methods (*callerV()*), another one for the getters (*getterV()*) and some other small utility ones. The third and last part is identical to the former one, but related to the Dutch auction.

Contracts ChangeLog

This section contains all the modifications applied to the delivered Final Term smart contracts. All the modifications have a motivation assigned.

- *Vickrey - Removed file Util.sol*: the only method that was provided in this file (computation of the nonce and amount hash for the Vickrey auction commitments) has been moved to the file Vickrey auction.
- *Both - Added auctioneer role*: the text of the project explicitely asks for an auctioneer role. The owner decides which account is the auctioneer when he calls the newly added method createAuction. If he does not decide an auctioneer, he becomes the one to take the role. As a note, in the Vickrey

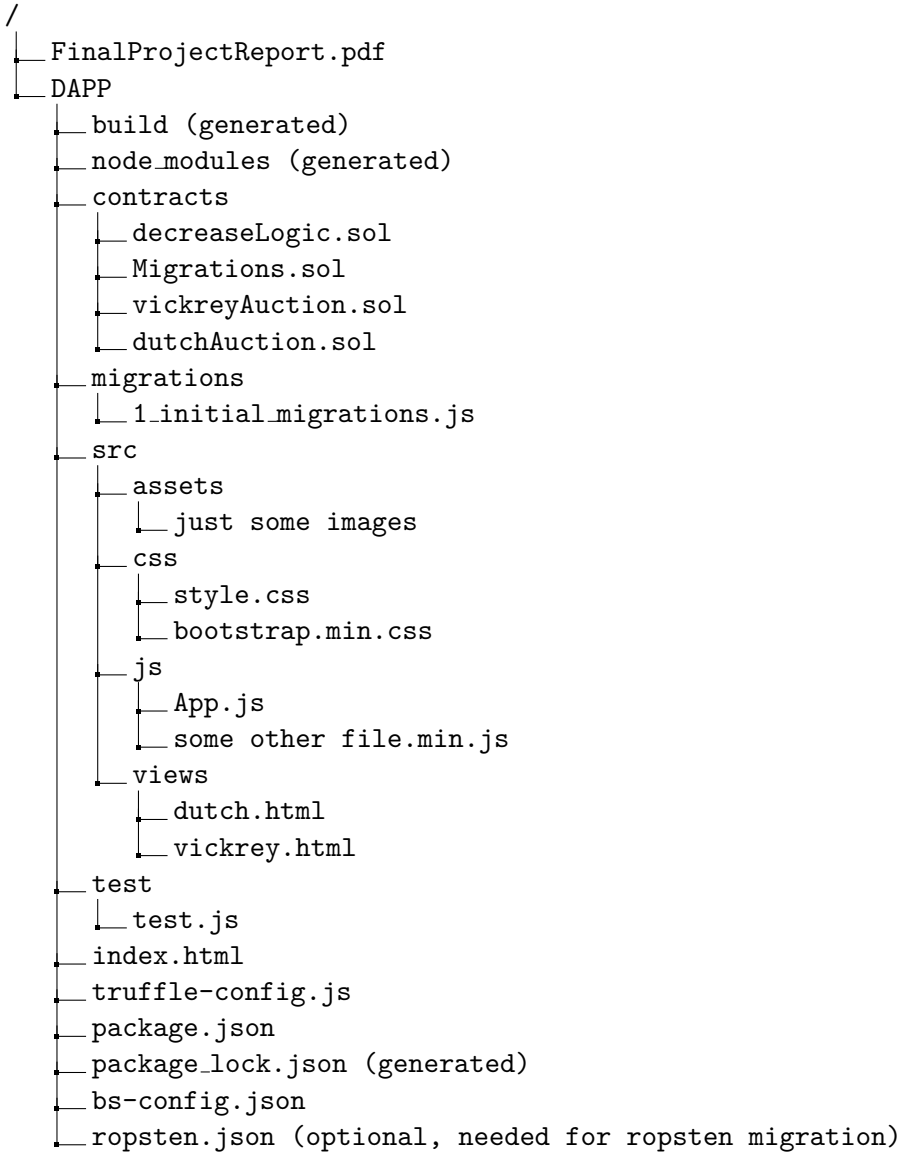

Figure 1: The project *directory* structure

auction, the role of the auctioneer doesn't have a clear purpose. The way it will be used during the tests is to take care of the phase switches, in order to make him pay the gas price of those switches.

- *Both - Moved start time*: Both auctions, in the provided FinalTerm implementation, started in the moment the constructor was called. This was split in two phases. Phase one starts when the constructor is called, and it is the phase in which the auction is on the blockchain, but is not yet active (note, grace period is considered active). The second phase starts when the owner calls the method createAucion. In this method, the grace period begins and an event is emitted.

- *Dutch - Added auctionPhase*: In order to better implement the previous point, the boolean variable ended has been changed to a phase enum in the Dutch Auction. The phases are NEW, ALIVE and ENDED.

- *Dutch - Changed assert to require*: In order to better respect the require-assert definition, the assert in the constructor has been changed to a require.

- *Both - Added some getters*: In order to improve clarity and respect the previous changes in the contracts, some getters have been added.

- *Both - Added suicide operation*: The auction suicides whenever the owner of this auction presses on the red button in the UI.

Main Project Choices

Below are listed the main project choices, together with an explanation regarding the reasoning behind those ones.

- *One HTML view*: The main choice is the one to collapse the three views (auctioneer, owner and bidder) in one HTML page. Technically this is incorrect (because of course every role should have its own customized web view), but this strongly simplifies testing and gives a broader view of the auction as a whole. This choice was taken mainly due to the fact that the setup code is identical for all the roles.

- *address management*: The first choice (collapsing the html views) strongly implies that it is not possible to fetch the MetaMask address while in initialization phase, and then use the same one for the whole interaction. This means that, for each operation that needs to have an address specified, it has to be fetched from MetaMask, resulting in a (very low) delay. The operations affected are only the main contract ones, as getter methods do not need to specify the address they come from, with some rare exception.

- *HTML view organization*: The web index prompts to the choice between the two implemented auctions (Dutch and Vickrey). After deciding which one to view, the web3 initialization process begins, which ends with the rendering of the appropriate view for the two auctions. The web page is organized in three areas as shown in figure 2:

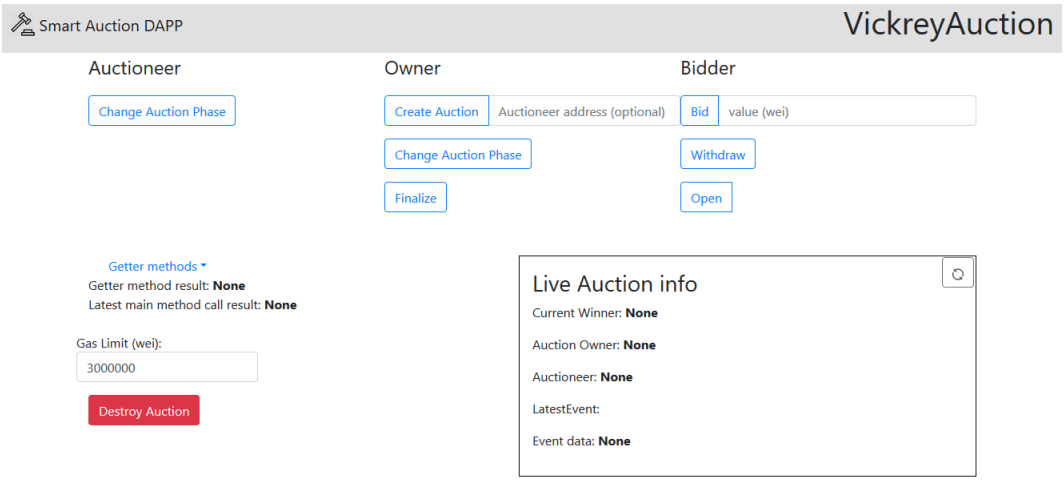

Figure 2: UI snapshot from the Vickrey Auction

  - *Top area*: Provides all the operations for the three roles of the auction.
  - *Bottom Left area*: Provides an interface to call some getter methods from the contract and shows the result of both the main methods from the top area and the getters.
  - *Bottom Right area*: Area with some live information about the auction, together with all the events generated from the smart contract.

- *Dynamic loading*: The whole interaction with the smart contracts is done using one single web page, which uses dynamically *jQuery* in order to change the web view according to the user choices.

- *Vickrey Auction bids*: The Vickrey auction interface provides the ability to bid and open a commitment. The interface does not provide an input field for the nonce and for the hash, but handles those transparently. In addition to that, the only input to be provided for making a bid is the bidding value, and it is provided directly in the bidding phase (in opposition with the contract, where the payment of that amount is delayed to the opening phase). The DAPP will handle the contract payment delay, together with the handling of the hashes and nonces transparently for the user.

- *App object*: The whole contract initialization and interaction is handled by a big js object called *App*, for which the logic is in the *App.js* file. The generals setup call chain (issued whenever the user chooses the auction type) is:

$$init \rightarrow initWeb3 \rightarrow InitX \rightarrow listenForEventsX \rightarrow renderX$$

  Where X is either Dutch or Vickrey (their implementation resides in different functions since they have strongly different events/callbacks). The getter callbacks are under *App.getterX*, while the main methods callbacks are under *App.callerX*.

- *Error Handling*: Web3, truffle-contract and MetaMask do not handle EVM reverts gracefully (revert operations are caused by both assertions and require

clauses). This means that the string of the failing require is not a clear field of the error message. So in order to retrieve the actual error message thrown by the contract some ugly parsing had to be performed. This results in a functional but sketchy way of handling errors.
That being said, in the provided implementation of the Vickrey Auction for the final term, requires and assert were not used in order to force every operation to update the current phase on the blockchain (revert does not allow to write the blockchain status). The original solution was to use legal transaction (which did not fail) and appropriate return codes. But unfortunately web3 does not allow to obtain the return codes from a transaction, which means that this information is lost.
Being committed to the choice of leaving the code with no require operations, the handling of errors in the Vickrey auction has been delegated to doing this:

1. call contract function.
2. call appropriate getter function in order to assert if the contract call was successful.
3. display the result.

Local and Remote setup

Local Testing In order to test the application locally those actions are going to be needed

1. *Install all the needed modules*: Place a terminal in the DAPP folder and run *npm install*. This will generate the *node_modules* folder and the *package_lock.json* file. Expect a wait time around 2 minutes. In case it was not installed earlier, install MetaMask web browser extension.

2. *Run Ganache*: The default port varies. The one that has been set in the project configuration file is 7545. If this is not the case, then just change property *port* under the *development* network in the *truffle-config.json* file.

3. *Migrate*: Do *truffle migrate --reset* in order to compile the project and deploy them on the Ropsten test net. The account that pays the deployment price is the first one in Ganache. This also generates the build folder. The parameters passed to the contract constructors are specified in the file *1_initial_migration.js*. In order to change them, just edit this file. This command will generate the build folder.

4. *Run the server*: Just do *npm run dev*. This starts the lite server.

5. *Browser*: At this point, two things are going to be needed: *MetaMask* and the *web view*. Just open on the browser *localhost:3000*. Now login with your MetaMask with an account and connect it to Ganache. Import the needed amount of accounts from Ganache through their private key and the setup should be complete.

```
{"mnemonic": "twelve metamask words under settings/
   security_and_privacy/reveal_seed_phrase",
"infura_key": "insert infura project ID here"}
```

Figure 3: *ropsten.json* file format

Remote Testing In case of a deployment on the Ropsten test net, then the process is the following:

1. *Install all the needed modules*: In case this step was not performed earlier (during local testing), place a terminal in the DAPP folder and run *npm install*. This will generate the *node_modules* folder and the *package_lock.json* file. Expect a wait time around 2 minutes. In case it was not installed earlier, install MetaMask web browser extension.

2. *Create Infura project*: Infura is used as a provided, so an Infura project is needed. After having created one, copy the Infura project ID.

3. *Create ropsten.json file*: This file is needed from *truffle-config.js* in order to export the project through infura provider. In figure 3, the file structure is shown. The *infura_key* is the previously copied filed, while the seed are shown in metamask by going under settings, security and privacy and then reveal seed phrase.

4. *Migrate*: At this point, issue this command: *truffle migrate --network ropsten*. Every contract migration has to be really mined so around one/two minute wait is to be expected.

5. *Run web app*: Same as before, login with MetaMask and connect it to Ropsten. Get some fake ether on some accounts and then run *npm run dev* in order to run the web page. Visit at this point *localhost:3000* where the empty web view should appear.

Some notes regarding the experience related to the remote

Testing

This section explains some operations that can be done in order to test the application both when it is deployed locally and when it is deployed on the Ropsten Testnet. The only valuable difference between the two cases is that in order to make the time pass on Ganache it is necessary to mine fake blocks, which get mined whenever a *successful transaction* is performed. The easiest way to do it in both auction is to call the phase checker function, which has the only effect of updating the phase in the auction.
If the application was deployed on Ropsten, it is suggested to increase the durations of the various phases in the contract (by at least a 2x or 3x factor). In order to do so, just change the parameters in the file *1_initial_migration.js* under the *migrations* folder.
Another thing worth noting is the way in which it is possible to choose the role of

the caller. Every time that an operation must be issued it is enough to select the issuer address in MetaMask. The front end uses the selected address in MetaMask every time that it issues an operation. So the cycle to make an operation is:

1. Select address in MetaMask.
2. (*optional*) Provide input (if it is needed).
3. Click on the button and issue the operation.
4. MetaMask will prompt a transaction, accept it.
5. Check the result of your operation in the bottom left part of the UI.

Having specified that, some example operations for the Vickrey Auction are listed:

1. *Create Auction*: The owner of the auction (make sure that you have the owner selected as an account under MetaMask) has the ability to start the auction (which starts the grace time).
2. *Commitment Phase*: Place some bids, considering that the value that has to be passed as parameter is not paid straight away, but it is paid by the application whenever the open method is called (so during the opening phase). The application takes care of this process under the hood.
3. *Withdrawal Phase*: Withdraw one bid during this phase (or more).
4. *Opening Phase*: In this phase, open all the bids that were placed. A suggestion would be to open the winning bid not first, in order to see the emission of the event that notifies that a new leader of the auction emerged.
5. *Auction End*: Whenever the opening phase is over (its end gets notified by an event), the auction is officially over. At this point the only thing to see is that the auction is finalized/destroyed correctly. Both these methods are only allowed to the owner of the auction.