DAPP for Smart Auctions

Peer to Peer Systems and Blockchains

Andrea Bruno

Department of Computer Science
University of Pisa
a.bruno25@studenti.unipi.it

1 Introduction

We were asked to develop a Dapp which must include two components: the blockchain-based back-end and the web-based front-end. In particular, by exploiting the contracts previously developed in the final-term project, we have to allow people to interact with their functionalities through a web browser.

2 Architecture

The system can be split into two main parts (Figure 1). The first is the client-side part made with JavaScript [1], while the second part is the blockchain-based back-end, built using Solidity [2] Smart Contracts. These contracts have been deployed to the network using Truffle [3], which is the most popular blockchain development suite.

Once the deploy is complete, a contract cannot communicate by itself, thus, a middleware between the caller and the callee is needed. In this case, almost all component within the UI exploits JavaScript asynchronous functions to retrieve information from the blockchain. In particular, Metamask [4] will act as a wallet by providing an actual address, whereas Web3 [5] will provide the API to interact with the Ethereum Smart Contract.

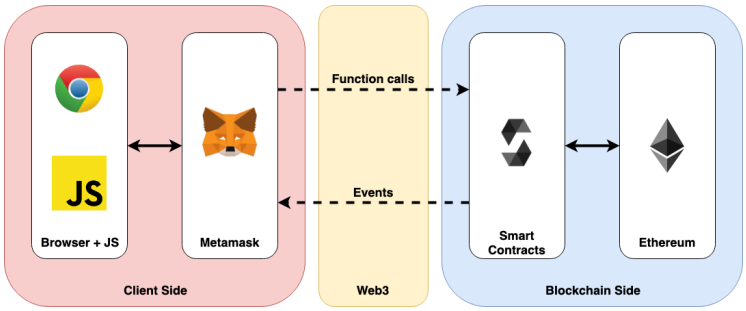

Fig. 1. Overview of the system architecture

3 Blockchain side: Smart Contracts

Due to new requirements, the old smart contracts have been modified. Apart from `AuctionHouse` contract, the figures within the following paragraphs, show only the actual code differences.

3.1 Auction House

The most important novelty is the introduction of the "auction house". This entity implements the so-called Factory pattern, which allows creating new instances (in this case, contracts) without caring about how to initialise them. Sellers can effortlessly create new auctions by calling the methods `newVickrey()` and `newDutch()`, it will be the house in charge of saving all those instances.

Of course, since there is an intermediary between bidders and sellers, there are fees to pay. As you can see from (Figure 1), people willing to sell a good, should pay 0.01 Ether, this because the functions are now called by the auction house, hence all costs (gas) shall be borne by it.

Furthermore, as the assignment requires, an event for every new-born contract get broadcasted. Interestingly, the other events asked, were already thought and written in the past contracts (e.g. contract starting, withdrawal phase...)

```
contract AuctionHouse {

    uint fees = 0.01 ether;

    address[] public auctions;
    uint public num_auctions;

    address payable houseOwner;

    event newAuctionAvailable(string _type, address _address);

    constructor() public {
        houseOwner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == houseOwner, "Sorry, only the owner can call this function");
        _;
    }

    modifier payFees() {
        require(msg.value == fees, "Please send a correct value");
        _;
    }
```

Fig. 2. A snippet of *AuctionHouse.sol*

3.2 Auction

Inside the Auction contract, few lines have been modified. Indeed, there is only the introduction of the modifier `onlyHouse` that allow only the auction house to execute some functions. As far as the description concerns, there are some important considerations to be discussed. First, we added the category of the auction because at run-time is not possible to retrieve. Second, to avoid inefficient computation, we look for events only from the latest, until the block in which the contract was deployed (line 13).

```
8  contract Auction {
9
10     /// @notice This struct incorporates the basic info about the auction.
11     struct Description {
12 +       string category;
13 +       uint deployBlock;
14 +       address payable house;
15         address payable seller;
16         string itemName;
17         uint startBlock;
18         address winnerAddress;
19         uint winnerBid;
20     }
21
22     /// @dev The description is public to allow people to read it without paying any gas.
23     Description public description;
24
25
26 +   /// @dev This modifier allow only the auction house to invoke the function.
27 +   modifier onlyHouse() {
28 +       require(msg.sender == description.house, "Only the auction house can run this function");
29         _;
30     }
31
```

Fig. 3. A snippet of *Auction.sol*

3.3 Vickrey and Dutch

Considering the variation of the roles within our contracts, some noteworthy aspects should be taken into account. First, all the function previously marked with the `onlySeller` modifier, are now signed with the modifier called `onlyHouse`. Moreover, in the Dutch auction (Figure 5), the `state` variable evolves from private to public, to avoid gas consumption. Finally, to let bidders know that the auction is in the validation phase, a new event has been created.

```
83    /// @notice The constructor of the Dutch auction.
84    /// @dev The `msg.sender` will be the seller.
85    /// @param _itemName is a short description of what is going to be sold.
86    /// @param _reservePrice is the reserve price decided by the seller.
87    /// @param _min_deposit is the minimum deposit decided by the seller.
88    /// @param _commitment_len the lenght of the commitment phase.
89    /// @param _withdrawal_len the lenght of the withdrawal phase.
90    /// @param _opening_len the lenght of the opening phase.
91    constructor(
92 +      address payable _houseOwner,
93 +      address payable _seller,
94        string memory _itemName,
95        uint _reservePrice,
96        uint _min_deposit,
97        uint _commitment_len,
98        uint _withdrawal_len,
99        uint _opening_len
100   ) public {
101
102       // Before deplying the contract a few controls are needed.
103       require(_reservePrice > 0, "Reserve price should be greater than zero.");
104       require(_min_deposit >= _reservePrice, "The deposit should be greater than the reserve price");
105       require(_commitment_len > 0, "The lenght of commitment should be greater than zero.");
106       require(_withdrawal_len > 0, "The lenght of withdrawal should be greater than zero.");
107       require(_opening_len > 0, "The lenght of opening should be greater than zero.");
108
109 +     // Set into the description the auction house address,
110 +     description.house = _houseOwner;
111 +     // the address of the seller,
112 +     description.seller = _seller;
113 +     // the name of the item
114       description.itemName = _itemName;
115 +     // the category of the Auction
116 +     description.category = "Vickrey";
117 +     // and the block number at deploy time
118 +     description.deployBlock = block.number;
119
```

Fig. 4. A snippet of *VickreyAuction.sol*

```
30      /// @dev Those are the possible states of the auction.
31      enum State {
32          GracePeriod,
33          Active,
34          Validating,
35          Finished
36      }
37  +   State public state;
38
39  +   /// @notice This event communicate that the auction is in validation.
40  +   event auctionValidation();
41  +
42  +   /// @notice The (ex) constructor of the Dutch auction. See FACTORY PATTERN...
43      /// @param _itemName is a short description of what is going to be sold.
44      /// @param _reservePrice is the reserve price decided by the seller.
45      /// @param _initialPrice is the initial price decided by the seller.
46      /// @param _strategy the address of a `Strategy` contract.
47  +   constructor (
48  +       address payable _houseOwner,
49  +       address payable _seller,
50          string memory _itemName,
51          uint _reservePrice,
52          uint _initialPrice,
53          Strategy _strategy
54  +   ) public{
55  +
56  +       require(_reservePrice > 0, "Reserve price should be greater than zero.");
57  +       require(_initialPrice > _reservePrice, "The initial price should be greater than the reserve price");
58  +
59  +       // Set into the description the auction house address,
60  +       description.house = _houseOwner;
61  +       // the address of the seller,
62  +       description.seller = _seller;
63  +       // the name of the item
64          description.itemName = _itemName;
65  +       // the category of the Auction
66  +       description.category = "Dutch";
67  +       // and the block number at deploy time
68  +       description.deployBlock = block.number;
69  +
70
71          //Set state into "grace period"
```

Fig. 5. A snippet of *DutchAuction.sol*

4 Client Side

4.1 Front-end

The web-based part of the DAPP has been realised using CSS and HTML by starting from the Bootstrap [6] framework. Before interacting with the system, you should log in to MetaMask and give access to the DAPP as Figure 6 shows. Then the UI will be loaded and the list of the auctions will be displayed. Note that to test the DAPP at least 3 accounts should be imported inside MetaMask. In particular one will act as *Auction House*, one as the *Seller* and all the other are *Bidders*.

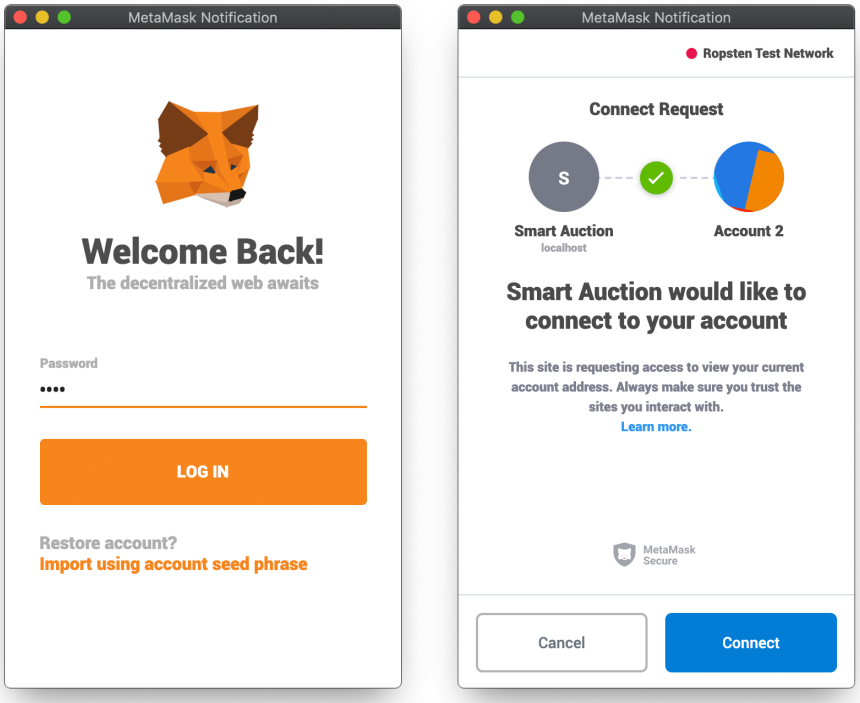

Fig. 6. Login and connection request on MetaMask

4.2 Create auctions

In the centre of the homepage, below the auction list, there is a panel named *"Create a new auction"* containing two buttons *Dutch* and *Vickrey*. For instance, assume we want to create a Dutch auction, as Figure 7 shows.

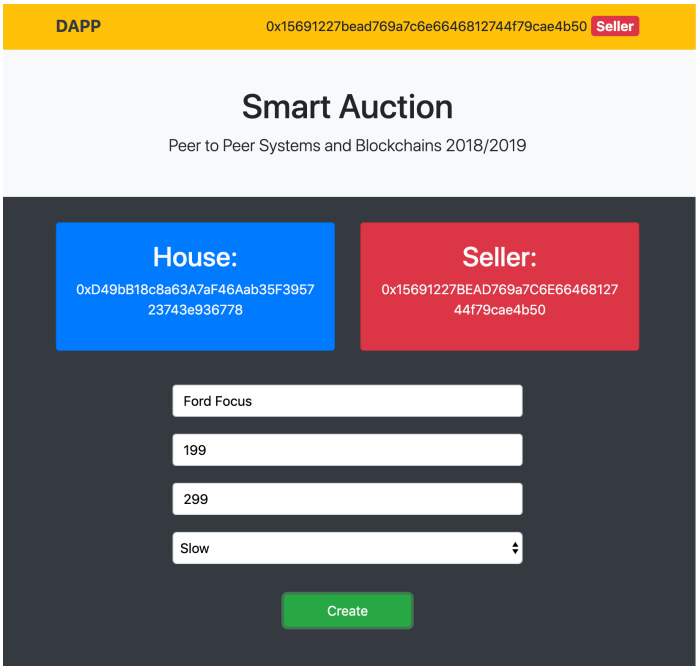

Fig. 7. New Dutch auction UI

By pressing *Create*, if everything was correct, you will display the transaction window, who will ask the confirmation to send Ether to the auction house. Once concluded the transaction, the system will notify the correctness of the creation as Figure 8 shows.

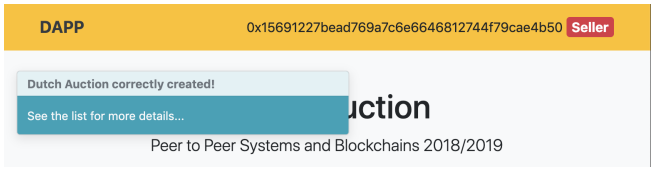

Fig. 8. New auction notification

4.3 Interact with the contract

At the bottom of the page, as Figure 9 shows, there is a panel called *Auction Control Panel*. The first step is to paste an actual address and then press *Interact*. Now, the system will automatically recognise the type of the auction and based on your role (House, Seller, Bidder), it will activate the appropriate functions.

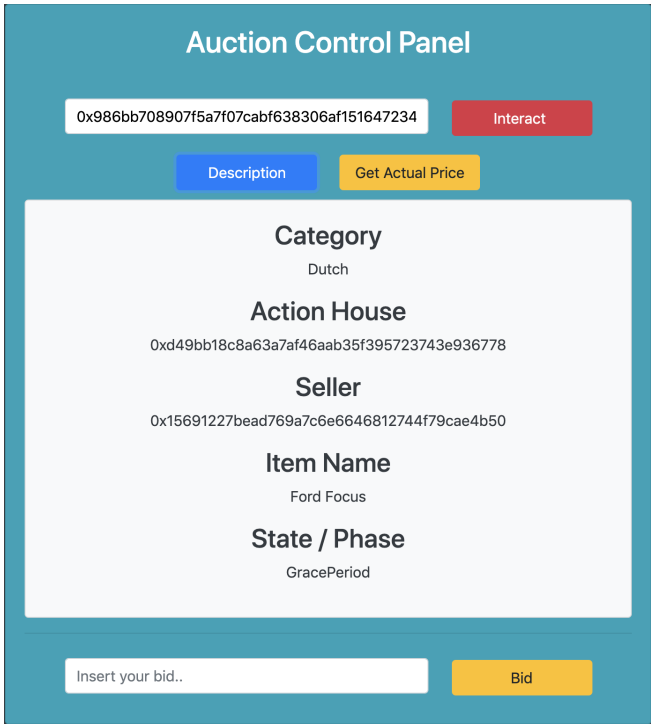

Fig. 9. Contract interaction after pressing *Description*

4.4 Events

As explained in the *Auction House* section, every contract instance is stored inside an array. Unfortunately, in Solidity, there is no way to allow a function return an array, thus although the data structure is public, the only way to retrieve data is a manual iteration. Let's analyse Figure 10.

```
37      App.contracts.AuctionHouse.deployed().then(async (instance) => {
38          let tmp = await instance.num_auctions()
39          numAuction = tmp.toNumber()
40      }).then(async () => {
41          for (var i = 0; i < numAuction; i++) {
42              await App.contracts.AuctionHouse.deployed().then(async (instance) => {
43                  let tmp = await instance.auctions.call(i)
44                  auctAddress.push(tmp)
45              })
46          }
47      }).then(async () => {
48          for (var i = 0; i < auctAddress.length; i++) {
49              var category
50
51              await App.contracts.DutchAuction.at(auctAddress[i]).then(async (instance) => {
52                  var descr = await instance.description()
53                  category = descr[0]
54              })
55
56              if (category == "Dutch") {
57                  await App.contracts.DutchAuction.at(auctAddress[i]).then(async (instance) => {
58                      var tmp = await instance.description()
59                      deployBlock = tmp[1] - 1
60
61                      instance.auctionStarted({}, {
62                          fromBlock: deployBlock,
63                          toBlock: lastBlock
64                          // toBlock: 'latest',
65                      }).watch(async (error, event) => {
66                          Utils.newToast("Dutch Auction started" , event.address)
67                          Listener.eventMap.set(event.address, ["Started", "Dutch"])
68                      })
69
```

Fig. 10. Snippet of *listener.js*

At row 38, we first retrieve the actual number of auctions created until now, then from row 41 to 45, we iterate and store in a temporary data structure, the addresses of those contracts. Afterwards, for each address retrieved, we understand whether this contract is a Dutch one or not, and then we start listening to the event `auctionStarted` (row 61).

5 Deploy on Ropsten

The final step is to migrate the contracts on the real network. Ropsten is a public blockchain network, where developers can test their contracts and DAPPs. Ethers are dummy, but limited as in the real main network. To start communicating with the blockchain, first, we need to create an account on Infura [7]. This website owns a collection of full nodes on the Ropsten network, in which developers can connect conveniently through their interfaces. They provide support for Pub/Sub API, as well as JSON-RPC filter support.

Once registered, you can obtain an Infura API key to an actual end-point. By setting the Truffle configuration file properly (Figure 11, we are now able to migrate our contracts on Ropsten.

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*" // Match any network id
    },

    test: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*" // Match any network id
    },

    ropsten: {
      provider: () => new HDWalletProvider(mnemonic, `https://ropsten.infura.io/v3/${infuraKey}`),
      network_id: 3, // Ropsten's id
      gas: 6000000, // Ropsten has a lower block limit than mainnet
    },
  },
  solc: {
    optimizer: {
      enabled: true,
      runs: 200
    }
  }
}
```

Fig. 11. Truffle configuration file

By executing the command `truffle migrate --reset --network ropsten`, we will get an output similar to Figure 12.

In the attachments of the project, there is a file called *migrationOutput* that contains the full output of the migration on Ropsten.

```
Starting migrations...
======================
> Network name:    'ropsten'
> Network id:      3
> Block gas limit: 0x7a121d


1_initial_migration.js
======================

   Replacing 'Migrations'
   ----------------------
   > transaction hash:    0x97045c35b8a831c7d6a75fdd6fce26534e5c1c22b3d5a0b21f14d4b7a0d94a4f
   > Blocks: 4            Seconds: 19
   > contract address:    0xF1a28162Ad40d013E11c67e3aBF2319eDB478032
   > block number:        5925188
   > block timestamp:     1562254075
   > account:             0x3811A4B4F8cAAB4d16fa123eAA35FD2940595C37
   > balance:             2.40762306
   > gas used:            188634
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.00377268 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.00377268 ETH
```

Fig. 12. Truffle migration on Ropsten

References

1. JavaScript, `https://www.javascript.com`
2. Solidity, `https://solidity.readthedocs.io`
3. Truffle, `https://www.trufflesuite.com`
4. MetaMask, `https://metamask.io`
5. Web3.js, `https://web3js.readthedocs.io`
6. Bootstrap 4, `https://getbootstrap.com`
7. Infura, `https://infura.io`