Implementation of a Decentralized Auction Platform in the Ethereum Blockchain

Blockchain and Overlay Networks
Challenge Task 2020 - dbay

1. Introduction

Online auction platforms are a popular tool to sell used items. Well-known services like Ricardo or Ebay are built in a client-server architecture. Within the challenge task of the course Blockchain and Overlay Networks 2020 at the University of Zurich, a similar system was developed on the technology of distributed systems. This distributed application (dapp) enables the creation of an auction platform on the basis of a smart contract on the ethereum blockchain. This platform has the functionality to create and bid for time-limited offers. If the item is sold after this period, the money of the winner of the auction is transferred to the seller. We called the developed system dbay. It can be accessed via https://bcoln-litecoin.web.app/.

Section 2 explains the technologies used and shows the architecture of the system. The implementation of the dapp is explained in section 3. This section deals with the structure of the Smart Contract. Section 4 shows the functionality of the dapp. The core functionalities of the frontend as well as a quick installation guide are covered in section 5. Lastly, section 6 looks back at the development process. It also selects which functionalities remain open and can be implemented in further development.

2. Architecture

Dbay consists of several components. The central element is the smart contract on the Ethereum blockchain. Such an auctionhouse, as we call it, manages various auctions, which are also separate smart contracts but all managed by the auctionhouse. All smart contracts were developed with the language Solidity.

To interact with this auctionhouse, we have developed a web application as frontend. It uses the framework Angular from Google. With the help of this web application, you can communicate with the blockchain via a well-structured UI. It is also possible to change the address of the contract via the UI, if you want to auction offers on your own platform (auctionhouse).

The webapp requires that the user has installed the browser plugin metamask. This is a widely used tool to communicate with an ethereum blockchain via the browser. Web3.js is used to communicate from the webapp to the blockchain via metamask.

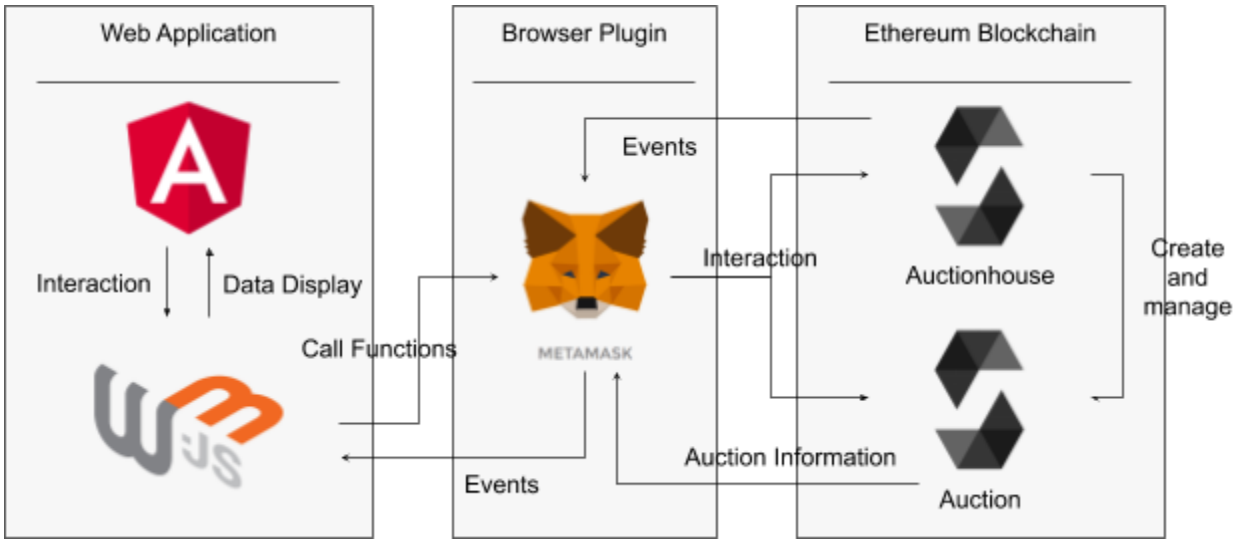

Figure 1: Architecture

3. Smart Contract

As already described in section 2, the main contract with which the webapp interacts is the auctionhouse. Via the auctionhouse, the user can create new auctions, see where they placed any bids and bid on auctions.

3.1 Auctionhouse

The auctionhouse stores all auctions and makes interacting with them easier. It contains a list of all auctions and various getter functions that filter the list by various criteria. For example all auctions that are running or all auctions where the asking address placed a bid at some point. In addition to that, the auctionhouse has functions to create new auctions and interact with them. There's a create function for every auction type (described below), and pass-through functions for all interactions with the individual auctions such as bid() or close().

The auctionhouse is technically not necessary to run auctions. Bidders can just send their bids to the auctions themselves in order to bid. However, keeping track of all the auctions and managing the different addresses is quite tedious, so the auctionhouse allows a more convenient interface to do that. Of course, using the auctionhouse for that incurs slightly higher transaction costs, so an implementation that does more by itself would be slightly cheaper to use.

3.2 Auction

Auction is an abstract contract that specifies the interface every auction has to adhere to. It has functions such as bid() and close(), but also public variables such as current_max_bidder or min_bid. Additionally, it also contains default function modifiers that simplify writing the actual functions. For example, there is a modifier auction_running that is used for the bid() function that checks that the function only runs when the auction has not ended yet.

An auction usually follows the pattern bid - delivery - close. During the bid phase, only bids can happen. After the bidding has ended, the auction winner has to acknowledge that they received the item they bought by calling the mark_as_item_received() function. Then, anyone can call the close() function that pays the winning bid to the seller.

3.3 Auction types

The FirstPriceAuction is what everyone thinks about upon hearing the term ‘auction’: The maximum bidder pays as much as they bid. It is intended that most auctions will be first price auctions as this format makes most sense if all bids are openly visible, as is the case with any scheme on a public blockchain.

A nice feature of the first price auction is that it’s really easy to implement. There’s only the need to keep track of the current highest bidder and their bid. In any other somewhat normal bidding format more information has to be stored.

The SecondPriceAuction is mostly a proof of concept because implementing the first price auction was so straight forward. Actually using the second price auction ends up in utilizing more gas than in the first price auction and has almost the same result unless a bidder makes a mistake and overbids. In a second price auction, the winner of the auction pays only as much as the next highest bid was. In a setting with sealed bids, this has the advantage that the optimal strategy is just bidding whatever the bidder values the item to be received at. In the first price auction, a lot of strategizing happens. However, on a public blockchain the second price auction does not make much sense to use because the seller can just underbid the buyer by a tiny amount and they will have to pay roughly the amount they bid.

Adding more auction types would be an interesting area to explore. Since all information is public, many schemes do not make sense, but there are a few that might be interesting, such as the dutch auction. In a dutch auction, bids instantly buy the item, but the strike price slowly decreases over time. On the other hand, this could lead to inefficiencies because blocks are mined too slowly (probably not a problem on the ethereum blockchain) or because discoverability is rather limited.

3.4 Further Improvements

As with any place where auctions are conducted, the auction format is a constant point of dissatisfaction. There are many different auction schemes, but most of them require some behaviour to get the best results. Especially on the blockchain where all information is public, the only place to hide information is outside of the blockchain and therefore inaccessible for any smart contract enforced bidding. We are currently not aware of any auction scheme that fulfills all the desired properties of strategy-proofness (the optimal bid is to bid as much as you value the item) and leaving no room for exploitation by the seller (the seller can easily impersonate as a bidder on public blockchains and therefore drive the price upwards). In addition, on the blockchain every single instruction has a cost, so in the optimal case every bidder has to bid at most once.

Lots of consideration can go into what capabilities an auction should have. On the blockchain, the winner could receive the item they bid for in the same transaction as the seller receives the payment. This however requires the capability to call arbitrary functions and thus complicates not only the evaluation of the item to be bid for, but also the interface to create new auctions. Especially in a GUI, we considered this task to be impossible to do in any user friendly way. While adding an address to be called upon sale is not that hard, it still requires competency within the blockchain by the seller, and the buyer has to be capable of examining the code to be executed. This is especially difficult because the code on the blockchain is only visible as assembler-like code.

As a last suggested improvement we consider modular auction contracts: Since every capability of a contract requires some code, every capability costs a certain amount of gas to store on the chain. To minimize the auction creation costs, there could be a way to enable or disable certain features of the auction. However, we don’t think this is the most urgent topic to be addressed.

4. DAPP Functionality

In this section we will have a look at the dapp’s functionality. There are three views, namely “Open Auctions”, “My Bidding” and “My Selling” whose features are explained below.

4.1 Open Auctions

Before being able to load the open auctions, one first has to set the auction house address to a valid contract address. Afterwards, the open auctions are shown. The open auction view includes all items that still can be interacted with. This are items that are still open for bidding, as well as items that are already sold but await confirmation of reception or confirmation of payment. Items of the latter two cases are only shown in open auctions if the currently selected account in metamask is either the buyer or the seller.

Figure 2: Open auction available for bidding

4.2 My Bidding

In the “My Bidding” view, items are listed that you have already placed a bid on. This includes items where the user has been outbidden, items that wait for reception, as well as a history of items successfully bought.

Figure 3: Item successfully bought. Confirming reception of the item.

Figure 4: Buying history

4.2 My Selling

In the “My Selling” view, items are listed that you are selling. This includes items which you have already sold, items that wait for the money to be collected, as well as items that have been sold in the past. Furthermore, new auctions can be started.

Figure 5: Waiting for buyer to confirm reception of item

Figure 6: Claim Money after buyer has confirmed reception of the item

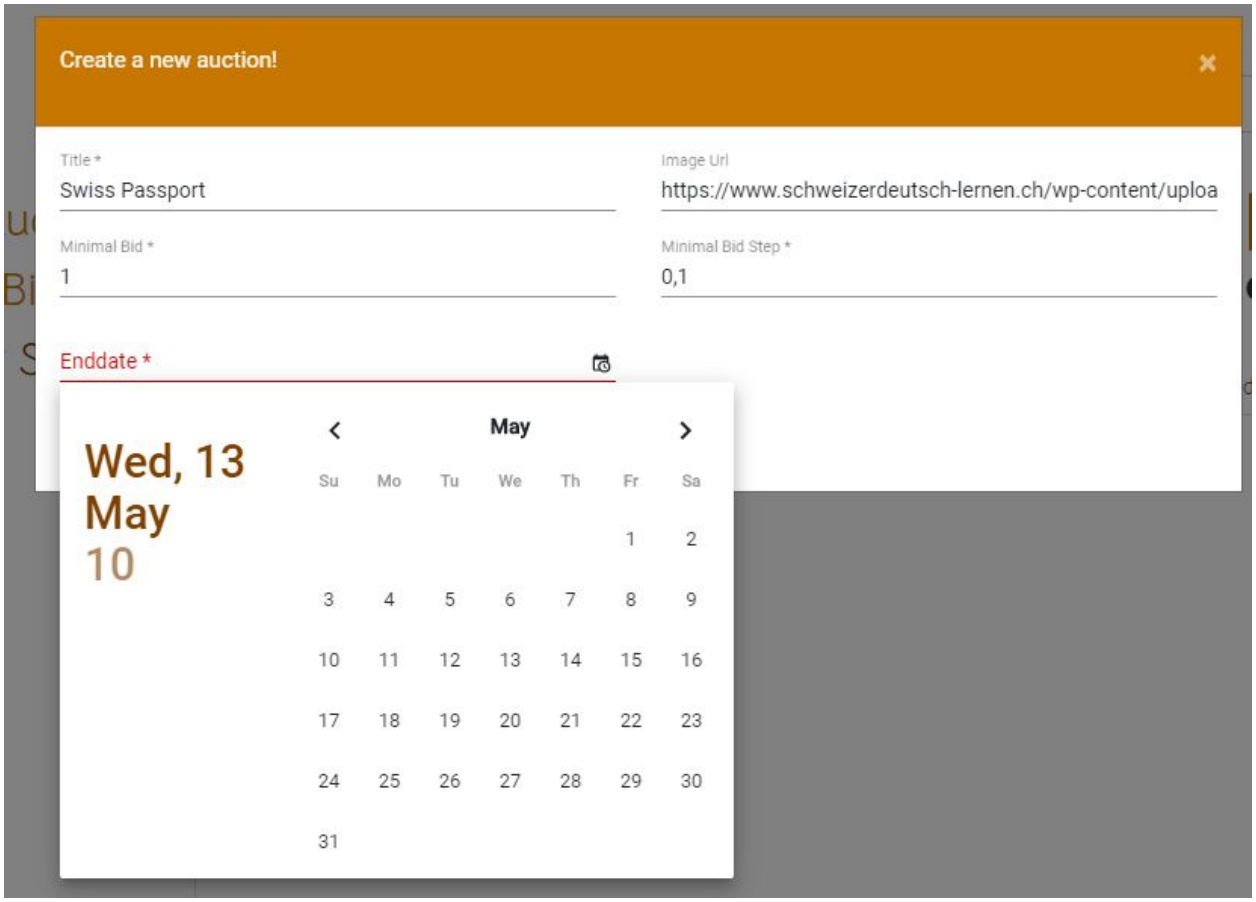

Figure 7: Create new auction

5. DAPP Source Code

In this section we discuss the core components of the frontend as well as a setup-guide.

5.1 Core Components of the Frontend

The dapp's frontend is implemented using the Angular Framework (Version 9). We make use of several different display components, one for each of the main views (open auction, my selling, my bidding) as well as a wrapper component and an item displayer component. These components include various different utility functions like converting ether to wei, displaying and formatting currencies etc.
The core of the frontend are two services. There is the auctionService that is used by all display components. The interface includes all the functions needed to cover the full functionality of the dapp (creating new auctions, placing bids, claiming items, claiming money, etc.). The auctionService itself connects to the metamaskService. The metamaskService makes use of the web3.js plugin and collects all the data stored in the metamask plugin. Furthermore, it contains

the ABI of the smart contract and includes all the functionalities that are needed to interact with an auctionHouse smart contract. The combination of those two services allow us to fully interact with the smart contract using the Angular GUI.

5.2 Frontend installation guide

In order to use the frontend in a local environment, the following prerequisites need to be fulfilled:

- Local instance of Ganache running
- Git installed
- Node.js version 12 or higher installed
- Metamask plugin
- Angular CLI 9 installed `npm install -g @angular/cli`

Follow these steps to run the gui locally:

1. Open the git bash in the folder you want to install the gui
2. Clone the git repository
   `git clone git@github.com:sesi200/bcoln-challenge-task.git`
3. Quick-Start a ganache environment
4. Open http://remix.ethereum.org/ and choose a Solidity workspace
5. In the installation folder (from github) you will find the following file:
   bcoln-challenge-task\contract\auctionhouse.sol
   Copy this file into the remix IDE, compile the contract (with compiler version 0.6.1)
6. Switch to the deployment tab, choose a Web3 Provider Environment with the Ganache url: HTTP://127.0.0.1:7545
7. Make sure in the contract selection drop-down, the AuctionHouse Contract (Second Entry) is selected. Deploy the smart contract.
8. Write down the address of the deployed contract.
9. In the git bash, navigate to bcoln-challenge-task\webapp\ in your installation folder.
10. Run the command `npm install` followed by `npm start`
11. After the app has started up, the local frontend should be available at http://localhost:4200
12. Make sure you have selected HTTP://127.0.0.1:7545 as network in metamask and import a few accounts from ganache
13. Enter the auctionHouse contract address from step 8 in the corresponding field and refresh the page

6. Conclusion

We agreed on an architecture that uses the browser plugin Metamask only during the development. At first, we wanted to access the blockchain directly. However, Metamask allows

the user to get a good overview of the transactions made. It is also a very popular and widely used tool that most users of an Ethereum blockchain have already installed. Communication via metamask makes it easier to connect and use dbay.

The current version of dbay contains the core functionalities of an auction platform. Thanks to the structure of the system, the system can easily be extended. Many online auction platforms also offer additional information such as item location or the possibility of an immediate purchase price. This is not yet available in the current version. With smart contracts, profiles can also be created and saved for the users. The smart contract already differentiates between two different types of auctions. In order not to confuse users with too much choice we agreed to keep this first version of dbay simple in the frontend. Therefore the first type of auction is chosen as default. Due to the structure of the system a change to the other auction type only requires the change of one line of code in the frontend.