Simple Blind Auction

Blockchain Homework

Assignment

Zsombor Zsáli

A6HFRW

Table of Contents

Design decisions

The smart contract for the homework assignment is implementing a Simple Blind Auction described in the task description. After consideration I decided to implement the phases of this first-price sealed-bid auction in a way, that after deployment the deployer can start the auction specifying the minimum price for the ECR20 tokens in auction, and a duration for the bidding and the revealing period.

The process is as follows:

1. The seller has some amount of ECR20 tokens – ZsTokens in our example – he wants to sell.
2. The seller deploys a smart contract on the blockchain specifying:
    a. The amount of tokens he wants to sell.
    b. The minimum price.
    c. The duration of the bidding and revealing periods.
    d. The address of the ZsToken smart contract.
3. The seller transfers the amount of ZsTokens to the auction smart contract.
4. The seller starts the auction.
5. The bidders can submit their bids during the bidding phase specifying:
    a. A hash made with the keccak256 algorithm from their bids value and a random nonce,
    b. A choice of value in Ethereum, transferred to the auction smart contract, greater than or equal to the reserve price, and of course their bids value.
    c. It is highly recommended for the bidders, to transfer a higher value than their bid, for the sake of secrecy.
6. After the bidding phase the bidders can reveal their bids specifying:
    a. The value of their bids,
    b. The nonce they earlier used.
    c. The keccak256 hash of their revealed value and nonce must be equal to their earlier submitted one, making it impossible to cheat.
7. After revealing their bids, the highest value bidder becomes winner.
8. The winner can claim his token and gets back the amount paid above their bids value.
9. The others can withdraw all their paid Ethereum.

The phases are implemented by functions in the smart contract that are described in detail in the API section of this document.

Data model

The bids structure

```
struct Bid {
        bytes32 hashedBid;
        uint256 valuePayed;
        uint256 value;
        bool revealed;
    }
```

The Bid is the essential data structure of the smart contract specifying:

1. The hashed form of the bid, submitted in the bidding stage,
2. The value paid by the bidder in the bidding stage,
3. The value bid by the bidder in the revealing stage,
4. A bool variable representing the state of the bid.

The nonce is not required to be saved.

ZsToken

```
contract ZsToken is ERC20 {
    constructor(uint256 initialSupply) ERC20("ZsToken", "ZS") {
        _mint(msg.sender, initialSupply);
    }
}
```

The contract of the ZsToken mocking an actual ERC20 contract. It inherits all the functionality of the implementation of the ERC20 token by openzeppelin. The only modification I made is in the constructor, where the deployer – in our case the Seller – gets an initial supply of the token binded to their address.

API of the Smart Contract

Variables

```
address public seller;
```

Address of the seller deploying the contract.

```
uint256 public highestBid;
```

The highest bid submitted.

```
address public highestBidder;
```

Address of the bidder with the highest bid.

```
uint256 public reservePrice;
```

Minimum price for the tokens given by the seller.

```
uint256 public biddingDuration;
```

Duration of the bidding phase given by the seller.

```
uint256 public revealingDuration;
```

Duration of the revealing phase given by the seller.

```
uint256 public auctionEndTime;
```

Ending time of the auction calculated from the starting time, the bidding duration and the revealing duration.

```
ZsToken public token;
```

The address of the token in auction.

```
uint256 public tokenAmount;
```

Amount of tokens the seller is offering.

```
struct Bid {
      bytes32 hashedBid;
      uint256 valuePaid;
      uint256 value;
      bool revealed;
   }
```

Data structure for the bids specifying:

- The hashed form of the bid, submitted in the bidding stage,
- The value paid by the bidder in the bidding stage,
- The value bid by the bidder in the revealing stage,
- A bool variable representing the state of the bid.

```
mapping(address => Bid) public bids;
```

Mapping between the bidders addresses and their bids.

Events

```
event AuctionStarted(
      address indexed seller,
      uint256 reservePrice,
      uint256 biddingDuration,
      uint256 revealingDuration,
      address indexed token,
      uint256 tokenAmount,
      uint256 time
   );
```

Event emitted when the auction has started specifying:

- The sellers address,
- The minimum price of the token in auction,
- Duration of the bidding phase,
- Duration of the revealing phase,
- Address of the ERC20 tokens smart contract,
- The **block.timestamp** when the event has happened.

```
event BidPlaced(address indexed bidder, bytes32 hashedBid, uint256 time);
```

Event emitted when a bid has been placed specifying:

- The address of the bidder,
- The hashed bid,
- The **block.timestamp** when the event has happened.

```
event BidRevealed(address indexed bidder, uint256 value, uint256 time);
```

Event emitted when a bid has been revealed specifying:

- The address of the bidder,
- The bid value,
- The **block.timestamp** when the event has happened.

```
event AuctionEnded(
    address indexed highestBidder,
    uint256 highestBid,
    uint256 time
);
```

Event emitted when the auction has ended specifying:

- The address of the winner,
- The value of the highest bid,
- The **block.timestamp** when the event has happened.

Modifiers

```
modifier onlySeller()
```

Modifier for functions callable only by the seller.

```
modifier onlyBeforeAuctionStarted()
```

Modifier for functions callable only before the auction has started.

```
modifier onlyDuringRevealingPhase()
```

Modifier for functions callable only during the revealing phase.

```
modifier onlyAfterAuctionEnded()
```

Modifier for functions callable only after the auction has ended.

Functions

```
constructor(
    address _token,
    uint256 _reservePrice,
    uint256 _biddingDuration,
    uint256 _revealingDuration,
    uint256 _tokenAmount
  )
```

Constructor of the contract called at deployment specifying:

- The address of the token in auction,
- The minimum price of the token in auction,
- Duration of the bidding phase,
- Duration of the revealing phase,
- The amount of tokens in auction.

```
function startAuction() public onlySeller onlyBeforeAuctionStarted
```

Function starting the auction callable only by the seller before the auction has started.

Emits **AuctionStarted** event.

```
function placeBid(
      bytes32 _hashedBid
   ) public payable onlyDuringBiddingPhase
```

Function callable by the bidders to place their bids during the bidding phase.

_hashedBid must be the bidders bid and their nonce hashed by the keccak256 algorithm.

It is recommended for the transferred value to be bigger than the exact value of the bid to hide reserving the secrecy of the auction.

A bidder can only place one bid.

The paid value must be greater than or equal to the **reservePrice.**

The Bid added to the mapping will have the **hashedBid** specified as parameter, and the **valuePaid** specified by the transferred value.

Emits **BidPlaced** event.

```
function revealBid(
      uint256 _value,
      uint256 _nonce
   ) public onlyDuringRevealingPhase
```

Function callable by the bidders to reveal their bids during the revelation phase.

_value is the exact value of the bid.

_nonce is the exact nonce of the bid.

valuePaid earlier must be greater than or equal to the exact value of the bid.

The **value** of the bid is specified as parameter.

The bids state changes to revealed.

Emits **BidRevealed** event.

```
function getTokenBalance() public view returns (uint256)
```

Function callable by the bidders to check if the contract has the amount of tokens specified in the auction.

```
function claimToken() public onlyAfterAuctionEnded
```

Function callable by the winner after the auction has ended.

Transfers the amount of tokens specified in the auction.

Refunds any excess funds to the winner above the exact value of their bid.

Emits **AuctionEnded** event.

```
function withdrawExcessFunds() public onlyAfterAuctionEnded
```

Function callable by the losing bidders.

Transfers the paid value of the losing bidders back to them.

Important implementation details

Modifiers

The mostly used requirements of functions are implemented in a form of modifiers specified at the end of function declarations.

Functions

All of the functions have some other *require()* statements for the sake of expected functioning. All these requirements are tested described in the Test Section.

Testing

The logic of the smart contract and the Sealed Blind Auction relies in the process of the auction. Therefore, it would not be suitable to build a new process for every testcase. The testcases implemented are in the order of the process testing every single point of failure. And achieving the expected results.

The implementation is JavaScript based with truffle initial support for mocha-like testing. The testcases use the mocha assert function for checking variables, return results and errors.

For simulating time, and the mining of the blocks I used openzeppelins test-helpers package.

For interacting with the blockchain and retrieving the values of variables in the contract I used the web3 package on the development test network described in the truffle-config.js. For detailed description of the test environment and running the testcases see Next Section.

For testing purposes, I used truffles default addresses as my seller and bidders. The seller is the 0. account, and the bidders are accounts 1-5. The 5. bidder is added to simulate a bad bidder, who tries to bid more than he pays for. The token and the auction are deployed on the network from the migration file with initial values:

- ZsToken
  - 2000 token balance for the seller.
- SealedBidAuction
  - address of the token deployed,
  - 100 for the minimum price for the tokens,
  - 120 for the duration of the bidding phase,
  - 120 for the duration of the revealing phase,
  - 1000 for the amount of tokens on auction.

Testcases

```
it("Check initialization", async () => {}
```

Checking the initial token balances of the seller and the auction contract.

```
it("Check for error thrown if the contract has not enough tokens to start the auction", async () => {}
```

Expecting an error if the auction contract has not yet received the expected amount of tokens.

```
it("Check state after tokentransfer", async () => {}
```

Checking the token balances of the seller and the auction after the transfer.

```
it("Check for error thrown if someone tries to start the auction who is not the seller", async () => {}
```

Expecting an error if someone other than the seller tries to start the auction.

```
it("Check for error thrown if value is less than reservePrice", async () => {}
```

Expecting an error if a bids value is less than the minimum price.

```
it("Check for error thrown if seller tries to start the auction again", async () => {}
```

Expecting an error if the seller tries to restart the auction.

```
it("Check state after bids were placed", async () => {}
```

Checking the balance of the auction after the bids have been placed.

```
it("Check for error thrown if bidder has already placed a bid", async () => {}
```

Expecting an error if a bidder tries to "rebid".

```
it("Check for error thrown if bidding phase has ended", async () => {}
```

Expecting an error if a bidder tries to bid after the bidding phase has ended.

```
it("Check for error thrown for bad hash", async () => {}
```

Expecting an error if a bidders hash does not match the hashed result of their bid value and nonce.

```
it("Check for error thrown if bidder has not placed a bid", async () => {}
```

Expecting an error if a bidder tries to reveal their bid without submitting a hash in the bidding phase.

```
it("Check for error thrown if bidder has not deposited enough for paying their bid", async () => {}
```

Expecting an error if the bids value is larger than the value paid by the bidder in the bidding phase.

```
it("Check state after bids were revealed", async () => {}
```

Checking for the winner and the winning bid after all the bids were revealed.

```
it("Check for error thrown if bid has already been revealed", async () => {}
```

Expecting an error if a bidder tries to “rereveal”.

```
it("Check for error thrown for claiming the token before auction end", async () => {}
```

Expecting an error if the winner tries to claim his token before the auction has ended.

```
it("Check for error thrown if revealing phase already ended", async () => {}
```

Expecting an error if a bidder tries to reveal his bid after the revealing phase has ended.

```
it("Check for error thrown if token claim is not from winner", async () => {}
```

Expecting an error if someone other than the winner tries to claim the tokens.

```
it("Check for error thrown if winner wants to withdraw", async () => {}
```

Expecting an error if the winner tries to withdraw his funds.

```
it("Check results of withdraws", async () => {}
```

Checking the balance of the auction contract after the losing bidders have withdrawn their funds.

```
it("Check results of claim", async () => {}
```

Check the token balances of the winner and the auction and checking the balance of the auction after the winner has claimed his tokens and excess funds.

```
it("Check for error thrown if winner tries to claim tokens again", async () => {}
```

Expecting an error if the winner tries to “reclaim”.

Setup to run the project and the testcases

For running the commands, you need to have Node.js v16, and npm installed and a working internet connection!

The project relies in a standard truffle project with a package.json in the root directory.

Open a terminal!

Windows – powershell (cmd is also acceptable, but it calls to powershell)
Linux – bash/sh/zsh

Clone the repository!

git clone https://github.com/ZsZs88/SimpleBlindAuction.git

It clones the project from github in the working directory.

Go to the projects root directory!

cd SimpleBlindAuction

It changes the working directory to SimpleBlindAuction, which is the root directory of the project.

To install the requirements for the tests, in the root directory run the following command!

npm install

It installs the necessary packages – specified in the package.json as dependencies – into the projects node_modules directory.

```
"dependencies": {
   "@openzeppelin/contracts": "^4.9.0",
   "@openzeppelin/test-helpers": "^0.5.16",
   "truffle": "^5.9.2"
  }
```

To run the tests run the following command!

Windows: npm run test-ps
Linux: npm run test-linux

The scripts are specified in the package.json:

```
"scripts": {
   "test-ps": "@powershell -NoProfile -ExecutionPolicy Unrestricted -Command
$GANACHE=ganache-cli -i=8484 -D -p 8484; truffle test; ganache instances stop
$GANACHE; Remove-Variable GANACHE",
   "test-linux": "GANACHE=$(ganache-cli -i=8484 -D -p 8484) && truffle test; ganache-cli
instances stop $GANACHE",
   "docify": "node docify.js"
  }
```

The commands process is as follows:

- starts a ganache development network in the background on localhost:8484 with a networkId 8484,
- runs the truffle test command, which deploys the smart contracts on the network and runs the test cases from TestBid.js,
- stops the ganache network.

The test network and required solidity compiler version is specified in truffle-config.js:

```
compilers: {
                solc: {
                        version: "0.8.0",
                },
        },
…
networks: {
                development: {
                        host: "127.0.0.1", // Localhost (default: none)
                        port: 8484, // Standard Ethereum port (default: none)
                        network_id: 8484, // Any network (default: none)
                },
}
```