FinalTerm Peer2Peer Systems and blockchains

Smart Auctions

Lorenzo Bellomo, 531423

Attached Files Structure

Attached with the project comes the auction files. The chosen auction is the Dutch one.
The final term code consists of four files:

- *dutchAuction.sol*: This file contains the Dutch auction logic. More details are explained in section Dutch Auction.
- *decreaseLogic.sol*: This file, to be associated with *dutchAuction.sol*, contains the *DecreaseLogic* interface contracts, and the 3 implemented logics: *Linear*, *Exponential* and *Logarithmic*.
- *vickreyAuction.sol*: This file contains the logic of the Vickrey Auction. Further details are explained in section Vickrey Auction.
- *util.sol*: This file contains one function to compute the *Keccak256* function needed for the Vickrey auction.

Notes Common to Both Auctions

Handling of Time Time was handled in the same way between the two auctions. All the durations (except for the one of grace time) are expressed in *number of blocks* and are passed as parameter to the constructor of the auction. Grace time, instead, is constant and "hard coded" in the contract. To respect the specifications of the assignment this value should be around 20/23 in order to respect the request of a time close to 5 minutes for the grace time. This is computed assuming an average mining time of 13/15 seconds per block on the Ethereum blockchain (according to the values in `https://etherscan.io/chart/blocktime`). However, in the actual implementation, the constant value for graceTime is set to 2 in order to ease the burden of testing.
The time passage is implemented in a *lazy* way. What this means is that every time a major function, like *bid()*, is called, the contract checks the current block number, and by confronting it to the first one (the block number when the auction was generated), and by checking the various durations specified, it computes the current phase. So no central entity is required to synchronize the auction (no auctioneer is required to dictate the passing of time).
The main pros of this approach (lazy way), with respect to the one with the auctioneer that dictates the passage of time are:

- *Completely decentralized*: No central entity (auctioneer) is required to make the time pass.
- *Guaranteed time correctness (on average)*: The auctioneer model requires the this entity to make the time flow. This means that he might be late while calling the phase switch (but not early supposing that the contract checks the correctness of the auctioneer calls). This is not the case of the lazy implementation, where the checks are made for each call by the contract.
  The time is guaranteed to be respected on average, with possible fluctuations

given by the variability in the mining process. This could have been avoided by using the timestamp of the block of the function call transaction, but this hypothesis was discarded in order to stick to the project requirements and implement time with respect to number of blocks.

Instead the main cons of this choice are:

- *Events variability*: Both of the auction implementations emit events related to the passage of time. Those events are fired whenever a phase switch is recognized. Since the phase switch process is lazy, the events may fire late with respect to the actual time in which the phase switched. This issue is inevitable in this kind of implementation, and it can be eased by providing a special method that checks the current phase of the auction and updates the state of the contract if enough time has passed. This method is *checkIfAuctionEnded()* for the Dutch Auction and *updateCurrentPhase()* for the Vickrey one.
- *Events cost*: The cost of those events emissions (which is very high with respect to normal EVM instructions) are in the general case at the expenses of the users (the bidders), which fire those events by executing the main methods of the contracts. The same "solution" of the first point is adopted, but this problem is evident, particularly when dealing with "almost empty auctions".

Compilation and Remix Deployment : Both the auctions have been compiled, during the test phase with the compiler version 0.5.1+commit.c8a2cb62 with enabled optimizations. In case optimizations are disabled a gas cost increase will be accounted, corresponding to around 30% with respect to the one with optimizations.

Dutch Auction

Main Functions The file has two main methods:

- *bid()*: This method simply allows a user to make a bid if everything respects the correctness parameters.
- *checkIfAuctionEnded()*: This method is only allowed to auction owner and allows to check if the auction has ended with no bidder.

In addition to these two the method provides some getter functions, in the form of *Solidity View Methods*.

Events The only event that is generated by this auction is the one emitted when it ends. This is generated when either the owner discovers that no offers were given or when someone tries to bid but discovers that the limit time was reached.

Decrease Logic The three implemented decrease logics are:

- *Linear*: This contract simply provides the linear decrease logic, which goes linearly with the passing of time from *startPrice* to *reservePrice*.

- *Exponential*: This contract provides the exponential decrease logic, which means that the first price drops are huge only to later decrease the price drops in order to reach the reservePrice
- *Logarithmic*: This contract is the opposite of the exponential one, meaning that the first price drops are low, only to increase with time.

The $\lceil \log_2 x \rceil$ function, used in order to implement both the logarithmic and the exponential decrease logic, was copied from a forum online. It uses inline assembly code and has a fixed cost of 757 *wei*.

Testing

In this section the *Remix* deployment and the *gas evaluation* will be addressed for the Dutch Auction. The evaluation will be done with the JavaScript VM with the *listen on network* box unchecked, in order to only consider the transactions made by the user when mining blocks.

Test Setup : In order to test the Dutch Auction, the subsequent actions will be performed:

1. *Activate compiler and deployer*: If on the new *Remix* environment, the two components to activate are the *Solidity compiler* and the *Deploy and Run Transaction* components.
2. *Compile dutchAuction.sol*: This will also compile the file *decreaseLogic.sol* because of the direct dependency. As a note, the compiler outputs warnings for the $\lceil \log_2 x \rceil$ implementation provided in the file *decreaseLogic.sol*, but they can be overlooked. The compiler version used in this examples is 0.5.1+commit.c8a2cb62 with enabled optimizations.
3. *Deploy a decrease logic*: In order to instantiate the auction contract, a decrease logic will be needed. The data following this section will be provided using a *linear decrease logic* contract.
4. *Deploy the Dutch Auction*: At this point the contract can be deployed (passing as parameter, where the decrease logic is expected, the address of the previously deployed linear logic). For example, it is possible to use 1000 as start price, 100 as reserve price and 10 as duration.
5. *Make the grace period expire*: The grace period duration has been lowered to 2 blocks in order to ease testing. It is useful to consider the fact that remix virtually mines a block for each transaction. The first grace period block is burnt the moment the auction is created, so it is only necessary to call once one of the main methods (*bid()* or *checkIfAuctionEnded()*) in order to effectively end the grace period. As a note, calling any of the getters (view methods), do not burn a block number, so it is possible to always check the contract status.
6. *Try some losing bids*: Now is the time to try some failing bids, before making the winning one, with any account. At this point any non view method call burns a block (and decreases the current price by 100 in the example).

7. *Win the auction*: At this point, one can either make an account win the auction, or let it end with no winner.

Gas Evaluation The cost of the various points has been computed as:

- The cost of deploying the two contracts (Decrease Logic and Dutch Auction) are respectively 88 735 and 650 446 gas.
- The cost of skipping the grace period is 830 gas and will be neglected.
- The cost of every losing bid is 4687 gas and it is fixed (since they all fail at the same instruction).
- The cost paid by the winner is fixed and it is 41 619 gas. If instead the auction ends without a winner (the duration expires), then the cost is around 23 651 gas.

Taking those values, then the final cost can be approximated as:

$$totalCost \approx 88\,735 + 650\,446 + (4\,687 * \#losingBids) + lastEventCost$$

Neglecting the losing bids, the auction owner pays around 740 000 gas, while the winner pays around 40 000 gas.

Vickrey Auction

Main Functions The main functions and modifiers provided by this file are:

- *changeAuctionPhase(blockNumber)*: This is a modifier that checks the current time (current block number) with respect to the parameters passed when constructing the auction contract. This modifier is used before any of the following methods are called.
- *bid(hash)*: This function takes a commitment as parameter and tries to submit it if all the parameters are respected (correct time range for making bids, paid deposit...).
- *withdraw()*: This function, if all the parameters are respected, undoes the commitment of the function caller.
- *open(nonce)*: This function is both used to open the envelopes of the bids, and to refund losers immediately. The case of a non valid commitment (which means that the commitment hash and the hash computed with the received nonce do not coincide), is handled as a non valid bid, which means that the deposit fund is not refunded. This is the same treatment of an opening which reveals that the original payment was below the reserve price threshold.
- *finalize()*: This function can only be called by the owner of the auction, and only when the auction state is *ENDED*. This method simply refunds all the not yet refunded accounts.

- *changeAuctionTime()*: This method simply calls the modifier specified to change the auction phase.

One important thing regarding this auction should be noted. Every call to any of the main functions potentially updates the state of the contract, because of the lazy check on the current phase. This means that it is not possible to use the *require* keyword, which implies a *revert* call on the EVM, which would make the call to the phase modifier useless. The checks are performed, and a return code is returned. The return code is an integer, and it is:

- 0 for and ok call
- 1 for a call made in the wrong phase
- 2 for a call made with insufficient funds
- 3 for a not found commitment
- 4 for a not valid call
- 5 for an valid bid, but which has surely lost the auction
- 6 for a not allowed method call

What this means is that, even wrong calls (those ones that do not satisfy the method specifications), are used to update the contract state (in particular, the auction state). This is to be considered as an incentive to make right calls.

Events The events that are generated by this auction are:

- *Time related events*: Those are simple events that follow the passing of time (phases). They are *GraceTimeOver*, *CommitmentOver*, *WithdrawalOver* and *AuctionEnded*. All those events carry some informations, like the number of current bidders (so the number of valid commitments that were submitted), and the duration of the next phase. As discussed in section Notes Common to Both Auctions, all those events are emitted in a *lazy* way.
- *Informative*: Those events (*NewCommitment* and *NewWithdrawal*) are emitted every time a new valid commitment is made or withdrawn. An additional event, *NewLeader*, informs that a new opening has been performed and it became the current winner of the auction.

Testing

In this section the *Remix* deployment and the *gas evaluation* will be addressed for the Vickrey Auction.

Test Setup : In order to test the Vickrey Auction, the subsequent actions will be performed:

1. *Activate compiler and deployer*: As for the Dutch one, compiler and transaction deployer are both needed

2. *Compile util.sol and vickreyAuction.sol*: The compiler version used in this examples is 0.5.1+commit.c8a2cb62 with enabled optimizations. Contract *util.sol* should be deployed before compiling, since for some reason compiling another file makes the executable disappear. It is a fake contract, and its cost should be neglected. It is only useful for the user to provide an easy way to compute the *keccak256* function of the nonce and the amount, being those two elements correctly placed in a bytes32 buffer as the contract expects. Otherwise, below this section there will be a table with a set of precomputed *keccak256* outputs.

3. *Deploy the Vickrey Auction*: At this point the contract can be deployed. It is recommended to use low durations for the various periods (like, for example 5 blocks), because to perform a phase switch, a main function is going to be called.

4. *Make the grace period expire*: In a similar way with respect to the Dutch auction, just a main method call is going to be enough to perform the phase switch.

5. *Commitment phase*: Now it is time to perform some bids (any other method, except for *checkAuctionPhase()*, is locked). Consider that the expected input of a bid is a hash with type *bytes32*. This input type is very strict, and any non matching function call is refused. This means that it is not possible to make fake bids with parameters like *0x0, 0, 0x1111111*, because they will all get refused by the EVM. Invalid bids should be created ad hoc with the *util.sol* contract (for example by using 0 as amount, or invalid nonces...). In order to perform a commitment it is just needed to use the function *getKeccak(nonce, amount)* from the util contract. Compute the value and using it as parameter. Remember to keep track of the nonces used and the amount sent. Perform those commitments with different accounts.

6. *Withdrawal phase*: In this phase only two methods are allowed (*withdraw()* and *checkAuctionPhase()*). Remember anyway that any call (for example to *bid()*) will virtually mine a block, so always consider the durations passed as parameters when calling any of the main methods.

7. *Opening phase*: At this point the first bids can be opened. A suggestion would be to open the winning bid early in order to test that the later losing accounts get refunded during this phase.

8. *Finalize*: This is the end of the auction, so the *finalize()* method will have to be called.

Table 1 containing some precomputed *keccak256* values. The nonce used in this examples is always 123.

amount [wei]	keccak256 [bytes32]
100	0xa7a5334673a8a584579f0818e23518ff24d11104b4dd0e048de82cc6277519a8
200	0x20b2cd778463c40ec253a86b7d3d1a90a12df6e4e76ccc01714e119ae42a9e93
500	0xa6708b3b03fd25062084301805a60c65263a30bacbec7835a1c607b0b3193bd7
1 000	0xbe1e9df8fd1dfa794e3e308e8eaeb0c81bd5a71b7bfadaa1ac364f1634f7349b
1 500	0xc51ac2e1ebd9e01d7eb5e75c5da3f3514003d70e6ae3ebf376c4668de170095f
2 000	0x2116bcf3740f445e8d3d99fad343b14699444eeaab657400157d6dae4659c4dd
3 000	0x34a3d2153b1546c2defe31082602ce6674c70fdc3f44a16ab67fbda6d16f4f7f
5 000	0x88f4b0a9c6a6c5001b0c264116c231b40d0ac7c6c3b02ead4774c61e0d591300
10 000	0x99ab724b476b98e1a5135ae0c80639dcbff4faf19c5b90fe3f7bbbfcce827244
50 000	0xb4b69dbb8aa3a48628b3f00bce86d21524da5c915032ee9403b56df5cead52bd

Table 1: Some Keccak function outputs. The nonce in each one of those output is fixed to 123.

Gas Evaluation An important note regarding gas evaluation should be taken into account. Being that events are fired every time that a phase switch occurs, the cost of those events relies on the shoulder of the unlucky participants. To evaluate the gas cost of these methods, a perfect auction owner assumption will be made, which means that the owner will perform a *changeAuctionPhase()* every time that a phase switch is submitted. The cost of the events emitted by the phase switch relies solely on the owner of the auction.
The cost of the various points has been computed as:

- The cost of deploying the auction contract is 1 880 553 gas.
- The cost of a phase switch (assumed on the shoulders of the auction owner) is around 51 000 gas totally (considering every phase switch).
- *bid()*: A successful commitment costs around 100 000 gas (the first one costs a little more, the following ones cost less), while instead it costs around 2 500 for a failing one.
- *withdraw()*: A successful withdraw costs 20 328, while a failing one costs around 3 000.
- *open()*: An opening which might still win costs 57 650, while a losing but correct commitment opening costs 23 421. An opening which is discovered to have paid a lower price with respect to the reserve price costs around 15 000 gas.
- *finalize()*: The cost of the finalize function is very dependant on the number of participants to the auction, but, averaging a number of runs, its cost has been estimated to be around 20 500 for each participant.

Neglecting the losing bids, the auction owner pays something less than 2 000 000 gas + the cost of *finalize()*, which has been estimated around $20\,500 * \#numberOfBidders$, under the assumptions described before. The honest bidders, instead, pay something around 200 000 gas.