Second MidTerm P2P Report

Remo Andreoli

June 2019

Contents

Preface

Implementation of the English and Vickery Auction using Solidity.

Chapter 1

Smart Contracts

Both implementations are located in `smart_auction.sol`, along with the base contract, called **smartAuction**.

1.1 smartAuction

It is the abstract contract which acts as a base for the English and Vickery Auction. The idea was to define a fixed main design and to provide a set of useful methods, in order to simplify the implementation of custom auctions and to ease the burden of security.

smartAuction is defined as a set of phases/states that sequentially changes over time, making the contract difficult to break, if the programmer is able to correctly define the conditions of the main functionalities. The possible states are the following: preBidding (phase before any bid can be made), Bidding (phase where bids can be made), postBidding (phase after any bid can be made) and End (when the auction is completely over). The auction passes sequentially from a phase to another, as shown in Figure 1.1, until the state becomes End. It is possible to dinamically change the length of each phase, in order to move the auction closer or further away from the End state.

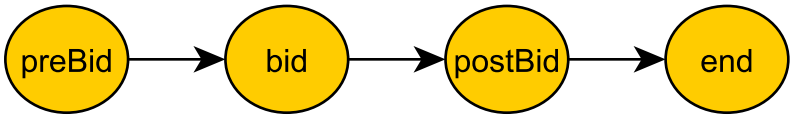

Figure 1.1: State transaction

The provided set of methods is described below:

- `getCurrentPhase()`: Used to determine the current state of the auction, by measuring the time in blocks.
- `bidConditions()`: Internal method that specifies the conditions under which a bid can be made. In particular, it checks if the current state is the bidding phase. It is recommended to use it when implementing the bid method of a child contract.
- `withdrawConditions()`: Exactly as before, but with the withdraw's conditions. By default, you can withdraw only during postBidding phase or when the auction has ended.
- `finalizeConditions()`: As before, but with the finalize's conditions. It checks if the auction has ended and if the auctioneer has already collected the winning bid.
- `finalize()`: The actual public method that finalizes the auction. It is defined and not implemented for the solely reason of making the contract abstract. Therefore, the child contract is required to implement it.
- `refundTo(bidder, amount)`: Internal method that refunds a certain amount to a specified bidder, collecting the remains as a fee. Currently, the fees simply are left into a limbo, since its handling is out of the term's scope.

Other simple getter methods are not described, but are still listed in the gas consumption tables (Chapter 2).
The base contract also provides a series of useful events:

- `newHighestBidEvent(bidder, bid)`: It notifies when a new winning bid is made, showing the winner's address also.
- `finalizeEvent(bidder, price)`: It notifies when an auction end, who is the winner and how much it paid.
- `noWinner()`: As before, but used if no one won the auction.
- `refundTo(bidder, amount)`: It notifies when someone get a refund
- `logEvent(str)`: debug purpose.

In order to correctly create an auction contract using **smartAuction**, the programmer is required to define the length of each phase (In particular, the bidding phase period needs to be higher than 0) and the reserve price (in wei), and to implement the `finalize()` function.

The next sections describe the two implementation and show how each phase matches with the final term's requirement.

1.2 englishAuction

The parameters of the **englishAuction** are: the reserve price (in wei), the buy out price (in wei), the unchallenged period length and the required minimum increment from the latest winning bid (in wei).
Note: big numbers need to be wrapped in double quotes to avoid encoding errors.

Regarding the association with the **smartAuction**'s phases, the preBidding phase matches with the grace period (which is more or less 20 blocks), the Bidding phase matches with the buy out and bid period, and the postBidding phase doesn't not exist. In particular, the bidding phase length is initially equal to the unchallenged period length, then it is dinamically increased with every new winning bid or zeroed if someone buys out.

In the following, the methods that makes the auction:

- `finalize()`: It simply makes the money movement, if there is a winner.
- `buyOut()`: If there is a buy out price and no one has already started bidding, it is possible to instantly win the auction. This is done by zeroing the Bidding phase length, ending the auction immediately, because there is no postBidding phase.
- `bid()`: if the value sent is higher than the winning bid, it refunds the previous winner and extends the bidding phase length. The latter is not statically increased by the unchallenged period length, but depends on the current bidding length and on the distance between the block on which the contract is defined and the block on which the bid is stored. This way, every winner has the same length of unchallenged period, even if the bidding phase is close to the end or close to the start.
  Let's see an example in the next page to better consolidate the idea.

Example:
The contract is deployed on block 0. if $unchallengedLength = 3$, then $biddingPhaseLength$ will be initially set to 3. If the bid transaction is stored on block 1, we can't simply do:

$$biddingPhaseLength \mathrel{+}= unchallengedLength$$

Because it is not fair for the winning bidder: instead of having 3 time block worth of unchallenged period, it has 4! (Figure 1.2)

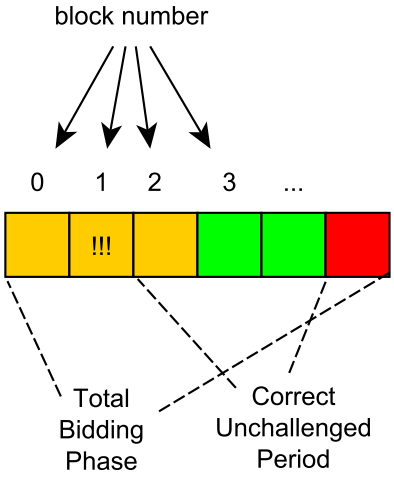

Figure 1.2: The unchallenged period is longer than expected

In order to achieve the correct unchallenged period length, depicted in the figure, the formula is the following:

$$biddingPhaseLength \mathrel{+}= unchallengedLength - X$$

$$X = biddingPhaseLength - ((bidBlockNum - deploymentBlockNum) + 1)$$

1.3 vickeryAuction

The parameters of the **vickeryAuction** are: the reserve price (in wei), the minimum deposit required (in wei), the bid commitment phase length, the bid withdrawal phase length and the bid revealing length.
Note: big numbers need to be wrapped in double quotes to avoid encoding errors.

Regarding the association with the **smartAuction**'s phases, the preBidding phase matches with the grace period, the Bidding phase matches with the bid commitment and withdrawal periods (hence here the Bidding period is split up), and the postBidding phase matches with the bid reveal period.

In the following, the methods that makes the auction:

- `finalize()`: As for englishAuction, It simply makes the money movement, if there is a winner.
- `bid(hash)`: It takes a `bytes32` hash, created from a `uint32` nonce and a `uint256` bid amount. In order to be accepted, the transaction also requires an amount of wei to be trasferred equal to the minimum deposit.
- `simple_bid(nonce, bidAmount)`: Handy method used to generate an hash, using a nonce and the bid commitment (in wei), and pass it to `bid(hash)` method.
  Note: the simulations have been made using this method.
  Note: big numbers need to be wrapped in double quote to avoid encoding errors.
- `withdraw()`: It refunds the deposit to the bidder wishing to withdraw the commitment, holding half of the value as a fee.
  Note: It doesn't use the `withdrawConditions()` method described in section 1.1, because the withdraw is done during the second part of the Bidding phase!
- `reveal(nonce)`: It checks if the hash between the nonce and the amount of wei sent with the message is equal to the hash of the commitment. If so, the method dinamically set the current winner and the current price of the good, refunding the previous winner if needed.

Chapter 2

Estimated Gas Consumptions

Below the estimated gas consumption of the getter methods, which are shared between the child contracts. Note that the actual consumption may vary. For example, `getCurrentPhase()`'s gas consumption, being the method a series of if-else, slightly changes depending on the branch taken.

Gas Consumption	**success**	**failed**
`getCurrentPhase()`	1499*	x
`getAuctioneer()`	626*	x
`getPreBiddingLength()`	512*	x
`getBiddingLength()`	402*	x
`getPostBiddingLength()`	490*	x
`getAuctionLength()`	858*	x

* Cost only applies when called by a contract

The only important value is the `getCurrentPhase()`'s cost, since this getter is basically used in every function implemented by the child contracts.

2.1 englishAuction

Below the estimated gas consumption table of the englishAuction, showing the costs of the typical usage.

Gas Consumption	success	failed
Deployment/init cost	1013959	429
`bid()`	60745	1861
`buyOut()`	34387	2214
`finalize()`	33371	1554

2.2 vickeryAuction

Below the estimated gas consumption table of the vickeryAuction, showing the costs of the typical usage.

Gas Consumption	success	failed
Deployment/init cost	1312281	429
`bid(hash)`	43156	2832
`simple_bid(nonce, bidAmount)`	43227	3095
`withdraw()`	21167	3225
`reveal()`	42602	3426
`finalize()`	48612	2699

Chapter 3

Simulations

A list of operations temporally ordered, used to simulate the system.
This operations are executed by those actors that want to trigger a certain action within the smart contract.
In the following simulations we are going to use 4 actors:

- auctioneer (or seller), funds = 0.
- bidder1, funds = 100 wei.
- bidder2, funds = 100 wei.
- bidder3, funds = 100 wei.

Actor and smart contracts communicate via transactions. A transaction invokes an auction's method in order to trigger an action, such as an exchange of money or a change of state in the auction. The only attribute required by the simulation is `value`, because it defines the amount of wei to be transferred to the contract. Hence, a transaction can be represented as:

$$senderActor(value = X) \rightarrow contract.method(params = values)$$

Where *method* might be the constructor call, named *init* in the simulations, or a contract's function, listed in Chapter 1.
Note: If the parameters' ordering is obvious, they won't be passed as keyword arguments for space's reasons.
Note: When an exchange of money is made, the funds' array is also displayed near the transaction

$$[auctioneerFund, bidder1Fund, ...]$$

Transaction's Example: englishAuction just started and bidder1 bids 10 wei:

$$bidder1(value = 10) \rightarrow englishAuction.bid()[0, 90, 100, 100]$$

3.1 englishAuction

Let's see a possible simulation of the englishAuction, recalling that the constructor is the following:
Constructor:
$englishAuction(reservePrice, buyOutPrice, unchallegedLen, increment)$

1. $auctioneer \rightarrow englishAuction.init(9, 20, 3, 2)$
2. wait grace period (use `wait()`)
3. $bidder1(value = 11) \rightarrow englishAuction.bid()$ [0, 89, 100, 100]
4. $bidder2(value = 20) \rightarrow englishAuction.buyOut()$ **REVERTED!**
5. $bidder2(value = 20) \rightarrow englishAuction.bid()$ [0, 100, 80, 100]
6. $bidder3(value = 21) \rightarrow englishAuction.bid()$ **REVERTED!**
7. $bidder1(value = 40) \rightarrow englishAuction.bid()$ [0, 60, 100, 100]
8. no other bids for 3 time blocks (use `wait()`)
9. $auctioneer \rightarrow englishAuction.finalize()$ [40, 60, 100, 100]

The gas consumption for the auctioneer is **1062571**, for bidder1 is **121490**, for bidder2 is **62959** and for bidder3 is **1861**. Hence the total gas consumption, without considering the `wait()` cost, is equal to **1248881** gas.

3.2 vickeryAuction

Let's see a possible simulation of the vickeryAuction, recalling that the constructor is the following:
Constructor:
$vickeryAuction(reservePrice, deposit, CommitLen, WithdrawLen, OpeningLen)$

1. $auctioneer \rightarrow vickeryAuction.init(20, 20, 5, 5, 2)$
2. wait grace period (use `wait()`)
3. $bidder1(value = 3) \rightarrow vickeryAuction.simple\_bid(1, 25)$ [0, 97, 100, 100]
4. $bidder2(value = 3) \rightarrow vickeryAuction.simple\_bid(2, 40)$ [0, 97, 97, 100]
5. $bidder3(value = 3) \rightarrow vickeryAuction.simple\_bid(3, 30)$ [0, 97, 97, 97]
6. $bidder2(value = 3) \rightarrow vickeryAuction.withdraw()$ **REVERTED!**
7. 1 block before withdraw phase (use `wait()`)
8. $bidder2(value = 3) \rightarrow vickeryAuction.withdraw()$ [0, 97, 98, 97]
9. 2 block before opening phase (use `wait()`)
10. $bidder1(value = 25) \rightarrow vickeryAuction.reveal(1)$ [0, 75, 98, 97]
11. $bidder3(value = 30) \rightarrow vickeryAuction.reveal(2)$ [0, 100, 98, 70]
12. $bidder3 \rightarrow vickeryAuction.finalize()$ [30, 100, 98, 70]

The gas consumption for the auctioneer is **1312281**, for bidder1 is **85829**, for bidder2 is **67619** and for bidder3 is **134441**. Hence the total gas consumption, without considering the `wait()` cost, is equal to **1600170** gas.