Title Goes Here

No Institute Given

Abstract. Abstract goes here

1 Introductory Remarks

2 Order Books

Order books are data structures that maintain lists of bid and ask orders for various assets (*e.g.,* currencies, stocks, bonds *etc.*) in specific markets. The most common version of order books is what we call a *double auction*, where market participants submit their bid and ask orders and the market clearing price will be calculated as the average between the best bid and best ask prices. Order books often sort the orders based on their price and submission time (this order allocation technique is called price-time priority) [**?**], where orders are prioritized from highest price to lowest and given any two orders with the same price, they will be sorted based on their submission timestamps.

Looking more closely, order books are rather electronic ledgers that get updated over time. Given the definition of the Etehreum blockchain (*i.e.,* a distributed ledger), the idea of implementing an order book in the form of a smart contract will seemingly resolve the existing issues with centralized exchanges. However, this design is not feasible due to some crucial challenges that exist within blockchains:

- **Speed.**
- **Front-running and Censorship.**
- **Enforcing Time.**

3 Clearing Mappings

To facilitate a safe exchange among buyers and sellers, we implement the CallMarket smart contract in the form of a *collateralized*; for market participants to be able to send bid and/or ask orders, they have to first supply assets (depending on what asset they aim to trade) as collaterals by calling any of the `DepositToken()` or `DepositEther()` methods. The collateralized CallMarket acts as a payment guarantees and market participants cannot default on payment or delivery of their assets.

To maintain the collateral balance of each market participant, we use two Solidity type one-to-one mapping that map Ethereum addresses to 256 bits unsigned integers; `TotalBalance` and `UnavailableBalance`. Once market closes

and orders are matched, the `UnavailableBalance` needs to be cleared. However, since it is not possible to delete the entire mapping without knowing the keys [1], clearing the `UnavailableBalance` mapping remains a challenging issue to solve . Here we provide a landscape of solutions for that.

1. **Creating a New Mapping Every Time the Market Opens.** Instead of clearing the `UnavailableBalance` of traders at the end of a matching process, we could create a new mapping every time the market opens. Note that using this solution, traders can only claim their funds (using the `ClaimEther()` and/or `ClaimToken` methods) only when the market is in state `Closed`.
2. **Creating Custom Keys for the Mapping.** We can create custom keys for the mapping by defining a counter as a global variable inside the CallMarket smart contract. This counter is incremented at the end of the matching process. So instead of clearing the mapping, we only use another portion of it every time the market opens.
3. **Storing the Mapping in a Data-Contract.** Another design proposal is to create a smart contract every time the market opens, this data contract only stores the `UnavailableBalance` mapping and will be killed at the end of the matching process.
4. **Storing the Mapping keys in an Array.** We can create an array that stores the traders' addresses (*i.e.,* the mapping keys). Knowing the keys enable us to iterate over the mapping and delete individual keys and what they map to at the end of the matching process. This is a gas-costly design pattern and would cause a high probability of exceeding the gas limitation inside the `Match()` function.

4 Who Pays the Cost for the `CloseMarket()` Function?

Ethereum contracts can only run when a function is called. So if no one calls the `CloseMarket()` function at the end of the trading period, this function cannot self-execute to modify itself. In addition, upon closing the market the `Match()` function will be executed which consumes a significant amounts of gas, and the person who closes the market must have enough incentives to do so. We think it is useful to explore the landscape of possible designs for closing the market.

1. **Using the Meta Transactions.** Meta transactions enable users to execute Ethereum functions without paying the gas. Rather than spending gas, users sign their intended action using their private keys and broadcast it to the network with no cost. A third party process (*a relayer*) then crafts the actual transaction on user's behalf, sends the transaction to the Ethereum blockchain, and charges the base contract with the associated fees (see Figure **??**). The required gas to pay for the `Match()` function could be collected as fees. So market participants are charged with certain amounts of fees every time they submit an order, these fees are accumulated in the CallMarket contract and will be used to pay for executing the `CloseMarket()` function.

[1] https://solidity.readthedocs.io/en/v0.5.12/security-considerations.html

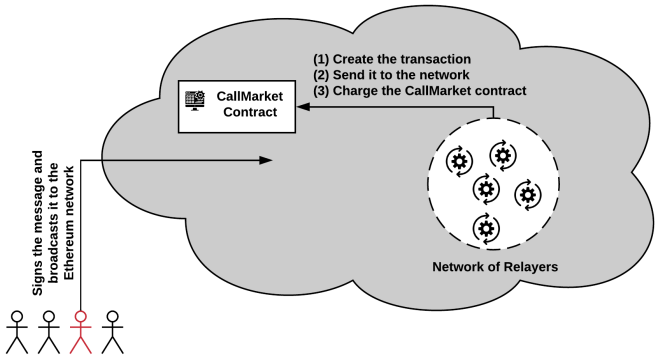

Fig. 1:

2. **Using the "Contract Pays" Model.** An alternative solution is to design the market such that the last person to submit an order calls the `CloseMarket()` function, but in contrast to a normal transaction (where the person initiating the transaction must pay the fee), here the CallMarket contract pays the cost for closing the market and matching the orders respectively. To enforce this design we can use Solidity function modifiers; every time a new order is submitted, a function modifier checks whether (i) the auction period has to end and/or (ii) the maximum number of total orders has reached. If any of these two conditions are met, the `CloseMarket()` will be called. Again, market participants are charged with certain amounts of fees every time they submit an order, these fees are accumulated in the CallMarket. Once the `CloseMarket` is successfully executed and orders are matched, the contract transfers its funds to that person. Note that here the person must still have enough gas to cover the execution of the transaction as the funds will be only transferred after the transaction is fully executed. However, market participants are incentivized to do so as they may receive more ethers than they have spent.

5 Unit Testing the Priority Queues

Here we execute the same JavaScrip test on the five priority queues with an end goal of unit testing them. We enter 50 unsigned integers to the priority queues in random ordering. To do so, we use JavaScript `Math.random()` function to generate pseudo-random integers between 1 and 200. Figure **??** shows the gas cost variations for entering 50 unsigned integers in the five data structures. The x-axis is the place in line (*e.g.,* the10th number entered in the priority queue) and the and y-axis is the cost of that transaction in gas.

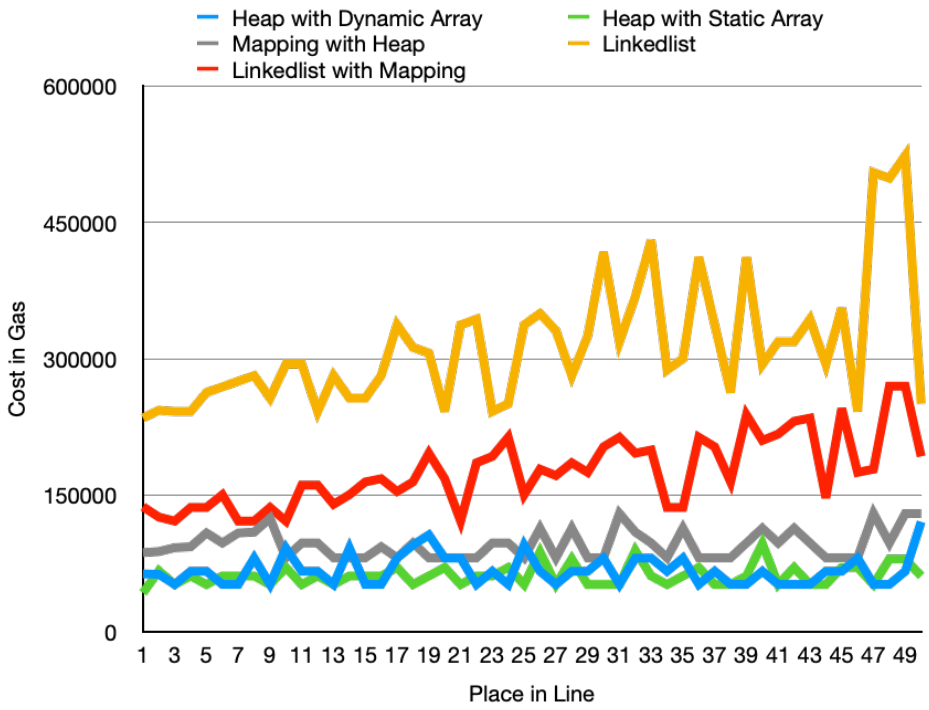
Heap with Dynamic Array
Heap with Static Array
Mapping with Heap
Linkedlist
Linkedlist with Mapping
600000
450000
300000
150000
0
Cost in Gas
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
Place in Line

Fig. 2:

Then, we call the `Dequeue()` function which iteratively removes the maximum value of the priority queue (until the data structure is empty). The computational costs for dequeuing 50 unsigned integers in each priority queue are outlined in Table **??**. The tests are performed using the current Ethereum gas metrics (block gas limit =11,741,495 and 1 gas = 56 gwei) [2]. The second column of the table shows the net gas consumption (the gasUsed value derived from transaction receipts) for removing 50 integers from each priority queues.

At the time of this writing, Ethereum transaction receipts only contain the net gas consumption and not the total gas consumption (total gas consumption is defined as $gasrefunded + gasUsed$) and we cannot find out the value of the EVM's refund counter from inside the EVM.

So in order to account for refunds inside each priority queue smart contract, we can calculate them manually; first we figure out exactly how much storage is being cleared when dequeuing the max integers and then we could multiply the number of storage slots cleared by 15,000 (see the last column of Table **??**).

Another way to know the amount of refund in each priority queue is to use the `estimateGas API` which provides a rough idea about the total amount of gas that is required for a transaction to go through. The `web3.eth.estimateGas` pretends the transaction is included in the block and its functions (with the parameters passed) will be executed on the Ethereum blockchain. Doing so, it provides us an estimate of how much gas is needed to be sent with the transaction. The second and third columns of Table **??**) show the total amount of gas required for dequeuing 50 integers from each priority queue (provided by estimateGas) and the amount of gas refund ($TotalGasConsumption - gasUsed$) respectively.

Note that in order to urge miners to process smart contract with refunds, the accumulated gas refund can never exceed half the gas used up during computation [**?**]. So at the end of a successful transaction, the amount of gas in the refund counter (capped at half the net gas used) is returned to the caller. For example, the amount of gas that has been used when dequeuing 50 integers from the linkedlist with mapping data structure is 731,514 and since $3,000,000 > 731,514/2$, the amount of refund returned to the caller is $731,514/2 = 365,757$.

6 Experiments

Our application was developed in Solidity using the Truffle development framework and deployed on Ganache-CLI. We used Javascript for testing by leveraging the Mocha testing framework. Followings outline the results of different tests we performed.

[2] https://ethstats.net/

Priority Queue	Net Cost in Gas	Total Cost in Gas (from estimateGas)	Gas Refund (from `estimateGas`)	Gas Refund (Manually Calculated)
Heap with Dynamic Array	2,547,031	3,312,378	765,347	750,000
Heap with Static Array	1,324,856	2,090,182	765,326	750,000
Mapping with keys stored in Heap	2,863,239	4,378,584	1,515,345	1,500,000
Linkedlist	557,085	1,772,085	1,215,000	1,200,000
Linkedlist with Mapping	731,514	3,731,514	3,000,000	3,765,000

Table 1:

6.1 Experiments on the `Match()` Function

We executed the same test on the the five different versions of the CallMarket we implemented using five priority queues to examine the cost of the `Match()` function as well as the maximum pairs of bid and ask orders it can handle in each case. The `Match()` function's computational cost and the maximum number of orders it can execute in each case (before running out of gas) are outlined in Table **??**. Note that this is a *worst case matching* test where all bids and asks are submitted as marketable limit orders with specified prices that would be filled undoubtedly, performed using the current Ethereum gas metrics (block gas limit =11,741,495 and 1 gas = 56 gwei) [3]. The last column of Table **??** shows the gas cost of matching 1000 pairs of bids and asks for each priority queue for which we set the block gas limit to the maximum of 2^{53} (the Javascript's max safe integer).

7 Concluding Remarks

[3] https://ethstats.net/

Priority Queue	**Maximum Number of Matched Orders**	**Net Cost in Gas**	**Net Cost in Gas for 1000 Pairs of Orders**
Heap with Dynamic Array	26 pairs	3,274,994	457,326,935
Heap with Static Array	28 pairs	3,107,527	333,656,805
Mapping with keys stored in Heap	40 pairs	5,414,803	319,481,722
Linkedlist	90 pairs	3,279,847	35,823,601
Linkedlist with Mapping	130 pairs	3,297,717	25,095,370

Table 2: