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 orders based on their price and submission time (this order allocation technique is called price-time priority) [3], 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 Ethereum blockchain, that is 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 additional Array.** Another common pattern is to create an additional array on top of the mapping and iterate over that. This array (*e.g.,* address[]) stores the traders' addresses and enables us to iterate over the mapping and delete individual keys and what they map to at the end of the matching process. Not that this is a gas-costly design pattern as we would need to maintain a secondary data structure.

4 Design Details of PQs

4.1 Mapping with Keys Stored in a Heap

To store the orders, here we use Solidity mapping that map orders' ids (256-bit integers) to order structs with their variables of different types. The mapping keys are then stored in a sorted heap that is implemented with a dynamic storage array. Every time a trade happens; (i) the mapping keys associated with the best bid and ask orders are deleted from the heap and the heap is re-sorted, and (ii) the mapping elements containing the best bid and ask orders are deleted from storage completely which refund 15000 gas [4]. Deleting the mapping elements is yet a gas-costly operation which lowers down the maximum number of orders the `Match()` function could handle in this case. [2] However, we clear the mapping element once a trade happens in our design as leaving unnecessary data is not a proper design practice.

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

[2] Every order structs contains multiple variables of different types and removing them from storage is identical to setting multiple variables to zero.

4.2 Linkedlist

Every time a trade happens, we use the `self-destruct` operation to remove the nodes (*i.e.,* smart contracts) containing the best bid and ask orders from the Ethereum blockchain. When a node is deleted, it transfers its funds (if any) to the payable address of the CallMarket contract that has been previously passed to it as a constructor argument. Also, removing a smart contract from the Ethereum blockchain refunds 24000 gas to the caller.

4.3 Linkedlist with Mapping

Every time a trade happens when the `Match()` function is executed, the best bid and ask orders need to be removed from the data structures. We could do this in two different ways; (i) update the head and tail pointers of the linkedlist and/or (ii) removing the mapping elements that contain the best bid and ask orders from storage completely as well as updating the pointers of the linkedlist.

For the same reason we mentioned in 4.1, we clear the best bid and ask order structs from the mapping every time a trade happens as well as updating the pointers.

5 Closing and Reopening the Market

At the end of the trading period, the market needs to be closed and reopened. However, there is no automatic process to called an Ethereum function and contracts can only run when a function is called.

5.1 Enforcing Time on the Blockchain

5.2 Who Pays the Cost for Closing and Reopening the Market?

We think it is useful to explore the landscape of possible designs for closing the market.

1. **Miners Close and Reopen the Market.** The difference between the best bid and ask prices is called the *bid-ask spread*. In our design, when the trade occurs between the the highest bid (the highest amount a buyer is willing to pay for an asset) and lowest ask (the lowest amount a seller is willing to accept for an asset) orders, the bid-ask spread is paid to the miner. There are possibilities that (i) no trade occurs or (ii) the bid-ask spread is zero (*i.e.,* the best bid and best ask prices are identical). So there is enough economic incentive for the the miner to execute the `CloseMarket()` function and get refunded as the refund amount could be potentially higher than the bid-ask spread.

2. **Processing Orders in Groups.** Another solution pattern is to process certain number of bid and ask orders upon every execution of the `CloseMarket()` function rather than treating them as one substantial transaction. A market participant P_i would process n orders from the previous market (`CloseMarket(n)`) when sending new orders to the current market. This process continues until all the orders in the previous markets are processed.
3. **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 1). 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.

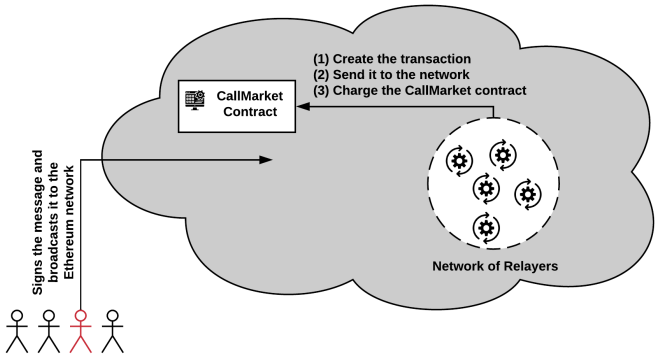

Fig. 1:

4. **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. **Using Rollups.** Rollup is a scaling method that moves the storage and computation of the smart contracts off-chain while maintaining the transaction data on the main chain as call-data. In this technique, any Ethereum user can act as a validator; they can execute the `CloseMarket()`function and only post the new state of the contract (the updated balance of traders) in the form of *assertions* to the main chain. Rollups improve scalability, provide faster and cheaper execution of the contracts, and eliminate the gas limit as the contract is no longer executed on-chain. In the followings we briefly discuss different rollup proposals and techniques. Each approach uses a different method to ensure correction of assertions:
   - **Non-interactive Rollups.** In this rollup technique, assertions are posted together with a validity proof that would be later used by validators to check if the `CloseMarket()` function has been executed correctly. ZK-Rollup scheme is one of the solutions that uses ZK-SNARKs to prove the validity of the assertions in zero-knowledge. ZK proofs are quick and cheap to verify but they are expensive and time consuming to generate. These proofs could be generated (i) for free or (ii) the CallMarket contract could collect proportional fees for every trade that is successfully executed.
   - **Optimistic Rollups.** In this scheme, assertions are assumed to be valid if there is no dispute posted about them with a certain window of time (a.k.a. "the challenge period'). Here, dispute resolution is a gas-costly method as the CallMarket contract would have to emulate the transaction on-chain to ensure the correctness the assertion. This scheme introduces a tradeoff between privacy and performance as all the assertions are publicly available and accessible. However, here the new state only reflects the updated balances of traders and no secret is involved.
   - **Multi-round Interactive Rollups.** In this design paradigm, *pending assertions* are posted on-chain and they are open to dispute. Once the challenge period is over and no challenge is submitted, the assertion is confirmed and the CallMarket contract transitions to the new state (*i.e.,* updates traders' balances). This scheme takes the overhead for the CallMarket contract to execute the `Close()` on-chain by using rounds to the dispute resolutions. The two parties (asserter and challenger) must run an interactive protocol and the CallMarket smart contract would have to act as a referee and decides which party's claim is true. Arbitrum is an example of multi-round interactive rollups that uses an efficient challenge-based protocol to penalize the dishonest parties [2].

6. **Using Trusted Execution Environments.** Another way of achieving execution of the `CloseMarket()` function is incorporating the Ethereum blockchain into the Trusted Execution Environments (TEEs) and decoupling the contract execution from consensus mechanism. TEEs enable secure execution of applications in an isolated processing environment called the *enclave*. Here, the enclave could execute the `CloseMarket()` function off-chain in TEEs and publish an on-chain attestation Quote to the CallMarket contract. The contract then verifies the correctness of the Quote and if validated correctly, it transitions to the new state. Ekiden is an example that uses Intel SGX to solve the scalability and confidentiality issues with the smart contract execution [1]. A drawback of this scheme is in order to achieve confidentiality-preserving smart contracts we have to trust a trusted party in the form of the hardware manufacturer (*e.g.,* Intel).

6 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 2 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.

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 2. The tests are performed using the current Ethereum gas metrics (block gas limit =11,741,495 and 1 gas = 56 gwei) [3]. 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 2).

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

[3] https://ethstats.net/

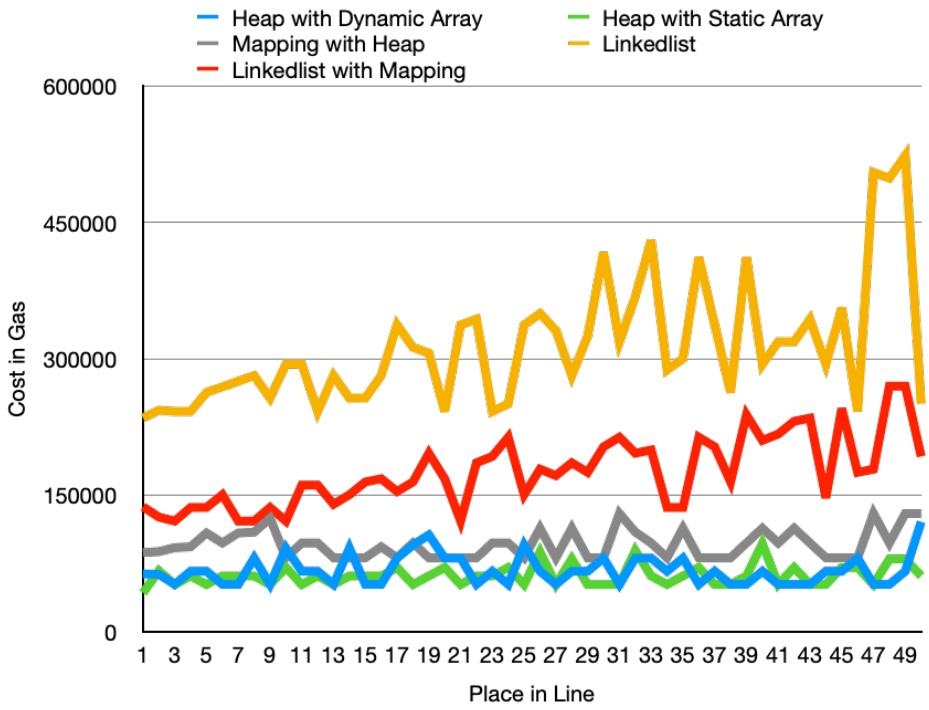
Heap with Dynamic Array
Heap with Static Array
Mapping with Heap
Linkedlist
Linkedlist with Mapping
600000
450000
300000
150000
0
Cost in Gas
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
Place in Line

Fig. 2:

provides us an estimate of how much gas is needed to be sent with the transaction. The second and third columns of Table 2) 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 [4]. 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 $731,514/2 = 365,757$.

Priority Queue	**Net Cost in Gas**	**Total Cost in Gas (from estimateGas)**	**Gas Refund (from `estimateGas`)**	**Gas Refund (Manually Calculated)**
Heap with Dynamic Array	2,575,997	3,349,746	773,749	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: PQUnitTests with 50 Random Integers between 1 and 200

7 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.

7.1 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()`

Priority Queue	Net Cost in Gas	Total Cost in Gas (from estimateGas)	Gas Refund (from `estimateGas`)	Gas Refund (Manually Calculated)
Heap with Dynamic Array	2,394,202	3,159,551	765,349	750,000
Heap with Static Array	1,317,922	2,083,240	765,318	750,000
Mapping with keys stored in Heap	2,629,191	4,144,536	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 2: PQUnitTests

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 3. 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) [4]. The last column of Table 3 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).

8 Concluding Remarks

References

1. R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.
2. H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. Arbitrum: Scalable, private smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1353–1370, 2018.
3. T. Preis. Price-time priority and pro rata matching in an order book model of financial markets. In *Econophysics of Order-driven Markets*, pages 65–72. Springer, 2011.
4. G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[4] 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	38 pairs	5,372,679	457,326,935
Heap with Static Array	42 pairs	5,247,636	333,656,805
Mapping with keys stored in Heap	46 pairs	5,285,275	226,499,722
Linkedlist	152 pairs	5,495,265	35,823,601
Linkedlist with Mapping	86 pairs	5,433,259	62,774,170

Table 3: