tl;dr

: we explore a theoretical

Ethereum CLOB DEX protocol that leverages Flashbots to enable traders to place orders that live off-chain and pay no gas fees unless the order trades and settles on-chain.

The issue with gas-expensive blockchains and CLOB

Today, most DEX protocols operate with passive liquidity provision and explicit AMM pricing curves (xyk, stableswap). This design allows for market makers to be entirely hands-off while providing liquidity and facilitating price discovery. However, it suffers from a number of drawbacks, including impermanent loss, poor capital efficiency, and high slippage.

On the other hand, TradFi markets predominantly use a central limit order book (CLOB) design. Within a CLOB, market makers submit bids and offers to provide highly targeted liquidity at select price points. Unlike depositing tokens into an automated market maker (AMM), placing a bid or offer expresses a view that the intrinsic price of the asset is above the bid or below the offer. Thus, a market maker needs to respond quickly to market moves and shift their orders around in real-time. Professional electronic market makers can easily submit millions of adds/cancels per day in their quest to tighten spreads and keep markets efficient.

The sheer magnitude of on-chain transactions and computation required for a CLOB makes implementation prohibitively expensive on a gas-expensive chain like Ethereum. Replicating a CLOB on a blockchain is possible, but currently only on gas-cheap chains (e.g., Serum built on Solana) or Layer 2s (e.g., dYdX).

Existing solutions

Uniswap V3

- Uniswap V3 and similar protocols partially bridge the gap between AMMs and CLOBs by allowing LPs to provide concentrated liquidity across a user-defined price range.

0x: Decentralized order book

- 0x provides decentralized liquidity through an order book or RFQ offering, in which professional market makers provide competitive pricing through the (centralized) 0x API, which can be settled on-chain. Orders can only be executed by explicitly setting an allowance of tokens with the 0x Exchange Proxy.

dYdX: Decentralized perpetuals

- dYdX maintains an order book and matching engine off-chain with settlement happening on-chain. Derivatives trading takes place on a Layer 2 protocol using a custom StarkWare integration, enabling trade settlement via a batched set of STARK proofs. Similar to 0x, the state of the order book is only visible through the dYd API.

CowSwap: Meta DEX Aggregator

- CowSwap is a DEX aggregation protocol that seeks to optimize trade execution by batching orders together and running competitive auctions to settle them. Traders submit orders (signed intent-to-trade messages) to CowSwap's off-chain service. Orders are collected to be settled in batches. For each batch, a limited set of searchers compete to find the best execution by matching orders against each other or against external liquidity.

We propose an alternative decentralized matching engine that can fill the orders without relying on any external liquidity.

Searcher-maintained Limit Order Book

Lending protocols like Aave use the concept of user-maintained liquidations. In case of a default, anyone can invoke the smart contract to liquidate the underwater loan and receive a bounty. Thus, protocol risk management is delegated to a searcher community that monitors for and liquidates bad loans. We propose a similar system in which anyone can match compatible orders and receive a "matcher reward."

In our proposed protocol, there are two types of transactions: order transactions and match transactions. Order transactions are submitted by traders and represent signed limit orders. They specify the intent

to trade at a side (bid or ask), price, size, matcher fee, and deadline. Match transactions can be submitted by anyone and are attempts to match outstanding orders in hopes of claiming matching bounties (fees specified by matched orders). They take in a set of orders and a settlement price. They then verify that the proposed match is valid (price_bid ≥ price ≥ price_ask), that sizes match (bid_size = ask_size), and that all orders are still live (time_now < deadlines). If all checks pass, the trade is settled at the specified price, tokens are transferred, and the fee bounty is transferred to the matcher.

Here's a concrete example using BTC-DAI:

1. Trader 1 submits an on-chain bid

Price - 45,000 DAI

Size - 1 BTC

Fee - 0.001 WETH

Deadline - Now + 24 hours

1. Shortly after, Trader 2 submits an on-chain offer.

Price - 44,500 DAI

Size - 1 BTC

Fee - 0.002 WETH

Deadline - Now +6 hours

1. The two orders are compatible because there is a valid match price that the matcher can choose (price_bid ≥ price ≥ price_ask), both orders are still live (time_now < deadlines), and sizes match. A matcher observes this and invokes the smart contract to match the orders, claiming the fees (0.001 + 0.002 WETH) allocated by the orders and clearing the trade at a price they choose.

Matching is fully permissionless and driven by matchers' self-interest. Matching is profitable if fees accrued overcome gas fees of performing the match operation. Fees are paid in WETH for convenience to matchers who balance trading fees against native gas fees. To claim these fee bounties, matchers must monitor for incoming orders (either on-chain or in the mempool), maintain an internal view of the "order book," and settle profitable matches on-chain. Through this open competition for matching bounties, the protocol delegates the computation ordinarily done by a matching engine to a community of searchers that can run it completely off-chain.

A few notes:

- Traders should provide an allowance beforehand to the DEX smart contract for the tokens they intend to trade. The funds will be transferred only in case of a successful match. Until then, the funds are not locked and traders can still use them in any way they want. They should also provide the allowance for WETH (used for fees).
- As currently presented, the design would have separate books for different quantities of the traded asset. Instead, we allow users to specify a quantity of any power of two. We then provide multi-match functionality, where the matcher can match several asks/bids as long as a single price can be used to satisfy all of them and the total size of asks is equal to the total size of bids.

This protocol works, but at great expense because each order submission is an on-chain transaction. Even when their order may never be filled, traders need to pay the requisite gas fee.

Enter Flashbots

We can improve this design with a critical tweak: leveraging Flashbots for bundled execution.Flashbots allows users to

submit atomic bundles of transactions directly to miners, where all or none of the transactions will be executed. Crucially, it allows for a bundle's gas to be paid altogether. Therefore bids/asks do not need to land on-chain when they are placed. Instead, they can specify a minimum gas price and be kept off-chain (in the mempool) until a match is found. They are then bundled together alongside the match transaction and submitted (and paid for) at once by the matcher.

In this example, four traders submit orders to the match,

and the matcher leverages Flashbots for bundled execution.

1. Traders submit orders to the mempool

Trader 1 - bid for 2 BTC for 84,000 DAI, with a 0.001 WETH fee

Trader 2 - bid for 2 BTC for 84,030 DAI with a 0.001 WETH fee

Trader 3 - bid for 4 BTC for 170,000 DAI with a 0.001 WETH fee

Trader 4 - ask for 8 BTC for 330,000 DAI with a 0.002 WETH fee

1. Matcher scans the mempool for a valid match - potentially involving multiple orders on each side.
2. Matcher submits a bundle [addBuy0, addBuy1, addBuy2, addSell0, multiMatch(ids, price)]

through Flashbots along with the requisite gas fee/tip. Note that the traders who submitted the initial orders do not pay anything beyond the minimum gas fee.

1. Flashbots submits the bundle to MEV-geth per the typical Flashbots workflow. If profitable for the miners, the bundle (and order matches) will be brought on chain in the prescribed order. The traders get filled and the matcher receives the WETH fee (in this case 0.001 + 0.001 + 0.001 + 0.002 = 0.005 WETH).

Note that in the above simplistic example, any trade price between 41,250 (the effective sell limit price) and 42,000 (the effective bid limit price) would satisfy the match requirements. The matcher can also take the other side and extract any crossing surplus by submitting their own matching trades. In the above example, the matcher can submit the opposite orders for matchable orders at the trader's reserve price, i.e.

1. Submit an offer of 2 BTC for 84,000 DAI
2. Submit an offer of 2 BTC for 84,030 DAI
3. Submit an offer of 4 BTC for 170,000 DAI
4. Submit a bid for 8 BTC for 330,000 DAI

In this way, the matcher earns the 0.005 WETH fee and

the 8,030 DAI surplus from crossed orders. As matching becomes more competitive, an increasing proportion of matcher profits will be split to miners.

Fee pricing

Matching is profitable whenever the total matcher reward plus potential arbitrage surplus exceeds the fee/tip required to be accepted by Flashbots. In the simplest case of no arbitrage surplus and a single bid and ask being matched, each order would be roughly responsible for paying half the base fee of the match operation, plus any additional priority fee/tip needed to be competitive in the Flashbots auction. Note that matching requires a constant number of instructions for every filled order, so the gas consumed does not depend on the size of the orders. Of course, higher fees would imply higher priority and vice versa.

Limitations

EVM transaction nonces create UX challenges

To prevent double-spending, the Ethereum Virtual Machine (EVM) keeps track of the nonces for each address (transaction signer). The nonce functions as a transaction counter. A transaction from a given sender is mine-able if and only if the nonce of the transaction is incremented by one from the nonce of the sender's previous transaction.

This complicates order management for traders attempting to maintain multiple outstanding orders. If the orders are specified with the same nonce, n

, then any fill invalidates all the remaining orders because they are using a now-invalid nonce. To refresh the orders' liveness, the trader would need to resubmit all orders with nonce n+1

. If the orders are specified with different nonces, e.g. n, n+1, n+2, ...

, then the orders must be filled in the sequence of the nonces. Both cases are unintuitive for traders accustomed to path independence for limit orders.

A few imperfect fixes:

1. Submit all orders with the same nonce. Once one of the orders gets pushed on-chain and the nonce is invalidated, resubmit all open orders with an incremented nonce.
2. Dummy wallets to distribute the transactions across several wallets.
3. For i

orders, submit all i^2

combinations of valid orderings. For example, if the next valid nonce is n

, submit each order with the next i

possible nonces (n, n+1, ..., n + i - 1)

.

No gas-free cancellations

Once an order transaction is signed and broadcasted, there is no way to recall it. The only way to "cancel" the order is to land a transaction that invalidates the nonce of the order transaction, but this requires paying gas and is not guaranteed to land before a matcher posts the trade. In the absence of such cancellation, the deadline parameter can be set to the desired expiration of the order. This mechanism is similar to how "time in force" on most TradFi exchanges defines the duration of orders.

No defined order priority

CLOBs typically operate with price-time priority. Orders at better prices will execute before orders at worse prices, and orders placed first (at the same price) will execute before orders placed later. However, because orders are matched by searchers as opposed to a deterministic matching engine, there are no priority guarantees. Consequently, bid and ask orders can be crossed.

Conclusion

The proposed theoretical design enables traders to leverage Flashbots and the existing searcher network to emulate an order book matching engine. Due to the numerous limitations described above, there are significant tradeoffs relative to alternative solutions involving centralized servers or specialized networks. Nevertheless, we believe SLOB represents an interesting thought experiment on how to incentivize existing infrastructure to provide additional functionality.

Acknowledgments

Thanks to Kevin Liu (@_kevinliu),

Mike Setrin (

@msetrin_), Ben Huan (@bhuan_), and Spence Pitts, Valentin Von Albrecht (@valentalb) for their feedback and discussion!

Share