Given the recent interest of people building their own solvers for CoW Protocol and the recurring question “Am I allowed to use my own, private market maker liquidity?”

(to which the answer is yes), I wrote a proof of concept to showcase how one might integrate one’s own liquidity into a settlement.

This post is meant to serve as a starting point to discuss design improvements and alternatives to this admittedly very hacky implementation. I assume you are roughly familiar with the solver competition architecture and format (for more background check out our solver workshop in Amsterdam).

The Goal

Let’s say we have a batch with a single user order, which is willing to trade a token pair at some price (e.g. sell 1000 DAI for at least 980 USDC). Let’s assume this is a price at which you would be happy to match the trade using your own inventory. Given the fierce competition between solvers (and taking the potential of the 100 CoW reward into account) you may be inclined to offer the user 1000 USDC, effectively matching them at a 1:1 exchange rate.

So, how would you do that? How can you match this trade at 1:1 given your own liquidity? I was able to think of three possibilities (there might be many more):

1. Underwrite a “just-in-time” 0x Limit Order and encode it in the “interaction_data” section of your solution.json (example tx).
2. Add a signed native CoW Protocol order (same format as regular CowSwap users use) in the solution.json (example tx).
3. Encode interactions that directly send the sell token amount from the settlement contract into your EOA and transfer the buy token amount from your EOA into the settlement contract (example simulation).

All of these approaches have their caveats. First, here is how they compare in terms of gas costs:

[

Points scored

1200×742 16.8 KB

](https://europe1.discourse-cdn.com/business20/uploads/cow/original/1X/ca7270ffa533bae682b78c1386da2f5443d98bcf.png)

0x Orders

The main issue with 0x orders is their high gas cost. Currently, solvers are picked based on an objective criterion which aims to maximize the total user surplus (difference between limit price and executed price) discounted by the estimated gas cost of the settlement. The intuition behind that choice was that the protocol doesn’t want to incentivize a $20 more expensive settlement to achieve only a $10 price improvement. Therefore, competitive solvers need to care about settling trades as gas-efficiently as possible. All prices even, a solver that can settle a trade cheaper is preferred over a slightly less gas efficient solver.

While 0x orders work perfectly fine and may be an easy starting point if you already have a working RFQ system that can emit calldata for their system, they may not be the best choice in the longer term. In our example a single trade was matched for 260k gas units.

Native CoW Orders

Signing native CoW Protocol orders has been the longest supported form for market makers to match user orders at the orderbook level (even before a batch is cut and sent to solvers). The basic idea is that by looking at the orderbook you can find orders you like and issue a “perfect counter-order” against them.

While onboarding market makers, we learnt that the long duration of “execution uncertainty” is an issue. Since your offer is competing with other on-chain liquidity, it can take up to ~30s before the protocol’s solvers have decided if they want to make use of your quote or not (in practice even longer as we don’t broadcast the matched orders before they are mined).

By becoming a solver themselves, market makers can add their liquidity after the batch has been cut, in the last possible moment before the winner of the competition is chosen. This dramatically reduces the trade uncertainty period. The caveat here is that adding liquidity in the form of native CoW orders is currently not supported by the solution json format (we should be able to add it quite easily if requested).

In my experiments the costs for settling a trade using two native CoW orders (1 user, 1 matching market maker) was ~200k gas units (~25% improvement compared to the 0x order format).

Transfer Interactions

The most gas efficient way is option number 3 - directly accessing funds from the market maker. However, for this to work the EOA would have to give an allowance to the settlement contract permanently (approving it on each settlement would be costly). Giving this allowance is risky. Without any checks, other solvers could abuse a market maker's allowance to match other user orders at arbitrarily low prices.

Therefore, to avoid abuse, we need some intermediate proxy contract that can verify some logic - the fact that the market maker is indeed authorizing this transfer - before sending the market maker's funds into the protocol.

Market Maker Intent Verification

Cryptographic signatures are usually used for this purpose. The easiest approach would be - in some intermediate contract - verify an off-chain signed intent from the market maker before admitting the transfer. However signatures need replay protection (to prevent a malicious solver from reusing a previously granted trade). Since there is no reliable protocol provided "per-batch" information available inside an interaction, I couldn't come up with a reliable way to prevent replay without requiring expensive storage writes inside the intermediate contract (basically invalidating a nonce).

The solution I eventually arrived at uses the fact that each solver has their own dedicated submission EOA. I implemented a wrapper contract which considers the requested fund transfer legit if and only if the transaction originated from a specific EOA (the solver's submission address).

Currently this key is managed by the "driver" component and therefore not under control of the solver directly (it requires some trust in the CoW development team). However, we are looking to decentralize the driver component soon and aim to have it be colocated with the solver implementations.

Implementation

I deployed an example instance of such an allowance manager on Gnosis Chain (the code has not been reviewed or audited, please don't use it blindly).

The creator of the contract instance is considered its owner and can change both the "sender" (the market maker's hot wallet address) as well as the "allowed origin" (the solver submission address your solver ends up being assigned).

Afterwards, the market maker's hot wallet needs to approve this allowance manager with all the tokens they want to market-make on. Then, in order to match the desired user order inside our solver, all we need is to encode two interactions in the settlement:

1. sell_token.transfer(market_maker, sell_amount)
2. allowance_manager.send(buy_token, proposed_buy_amount)

The first interaction sends the user's sell token amount into the market maker's wallet, the second will transfer the buy token amount from the market maker's wallet into the settlement contract, which will then send it to the user. The intermediate component (SolverAllowanceManager checks the tx.origin and reverts unless it's the expected one).

My simulation shows that this approach uses the least gas (183k, ~10% less than native orders).

Example code for how to implement this logic in ourpython solver framework can be found in this commit (which should allow reproducing the simulation).

To summarize the approach visually:

[

1600×825 118 KB

](https://europe1.discourse-cdn.com/business20/uploads/cow/original/1X/5914ffe9bd4a3cc9880cc427916bceea6e00d1e3.jpeg)

Any thoughts, comments or suggestions on how to improve private market making on CoW Protocol? Let's discuss…