New trends in decentralized exchange applications; exploring the design space

This article aims to raise awareness in the blockchain ecosystem about the ways in which decentralized exchanges (DEX) in Ethereum can evolve in the near future. It focuses on presenting design ideas for applications, some of which are new. With the intention of being as concrete as possible, we introduce a demo smart contract called "Dealer" which contains several convenient features that might be considered in subsequent community discussions. We illustrate each of these features with examples. Finally we discuss possible SUAVE applications that may play an important role in the path that connects user's action with on-chain execution.

Many thanks to Sarah Allen for her valuable comments and ideas to improve the article. Also thanks to Christoph Schlegel and Fred Hjalmarsson for their great comments.

Introduction

Up to this day, decentralized exchange activity in Ethereum is dominated by automated market makers (AMMs). Since their introduction, they have proven to be a powerful tool for both liquidity providers and traders. As a consequence of their wide adoption, we saw the rise of MEV, which is by now a well-known phenomenon. During the last few years, Flashbots has provided many interesting products for both preventing MEV and democratizing access to MEV. Other teams have focused on developing DEX applications of a new type based on smart contracts, which also offer MEV protection. Some of these are CoWswap, 1inch Fusion and UniswapX. To operate these apps, end users have to sign trade orders that are not blockchain transactions but messages following a predetermined format. At a later step, these orders get executed by the app's smart contract when included in a transaction sent by another actor, called "solver" or "filler". We want to illustrate the advantages of this mechanism in the following sections. We refer to the UniswapX whitepaper and CowSwap introduction for other perspectives on the same general picture.

The distinctive feature recently introduced by UniswapX is to make the filler role permissionless. This article strongly subscribes to this approach, since it will enable a higher degree of decentralization and efficiency.

Arguably, this type of system deserves a name. We will call them "Integral DEX applications" (IDEX). The name comes from the fact that they are able to batch arbitrary users' orders jointly with AMMs swaps and any other source of liquidity on the same chain, all in a single transaction. This is to be contrasted with a scenario where users only swap at AMMs. In such a scenario, we have larger gas costs, pool fees for every swap, and the outcomes strongly depend on the order of execution, leading to front-running attacks and outcome uncertainty. We would like to boost the development of the IDEX family, since we understand that there is a lot to gain from possible improvements and adoption. In the following sections, we want to illustrate many things that can be done.

Dealer smart contract

As a means to exhibit several significant use cases in the most concrete way, we present an IDEX smart contract called "Dealer" with its corresponding test cases. This smart contract has not been audited. It currently only serves the purposes of research and dissemination of ideas. The prototype has been useful to obtain the gas costs for each of the examples below. Moreover, we encourage the technical reader to explore the code. Its direct style makes it easy to read, so it can help some readers to understand how IDEX applications work in general.

The main guiding principles of this smart contract are flexibility and efficiency. Here, flexibility means that it should support a very general class of trading operations. As we will see, in many cases IDEX smart contracts are expected to work in combination with other systems running off-chain or on a different chain. Thus, flexibility is related to adoption through multiple routes. Efficiency means to optimize gas costs. The operative cost will determine the minimum viable trade size and therefore influence the ecosystem scalability. For this reason, we understand that saving an amount as low as 10K gas units per trade is significant and a priority.

We first present the smart contract design and then highlight some expected use cases. We chose this logical order of exposition, though it is also reasonable to read the use cases first.

An "order

" is a user generated piece of data (sometimes called "message") to be executed by the contract's main function.

Order

structure

allowedTokens

: a list of ERC20 tokens that the user allows the contract to transfer from the user when executing the order.

inequalities

: this is a finite set of[linear inequalities](#) that the balances of the user's tokens has to satisfy. The variables are token balances at both the beginning and the end of the order execution. This allows the user to impose conditions on the prices and amounts of trades.

conditions

: this field allows the user to express any other custom conditions. A condition takes the form of any contract function, since they can perform arbitrary verifications and revert whenever the condition is not met.

expirationBlock

: the block number at which the order expires.

Main function flow

The main function is called fillOrders

. The filler

is the agent calling this function. The input consists of a set of orders generated by users but chosen by the filler, their signatures, a set of transfers from users, a set of transfers from the filler (both specified by the filler) and a set of other functions to be called after the transfers, which may include transfers back to the users.

Visualization of an example input for Dealer smart contract's main function

The flow of "fillOrders

" can be summarized as follows.

1. Verify signatures and retrieve users' addresses.
2. Record balances before execution.
3. Execute transfers from users and transfers from the filler.
4. Call smart contract functions specified by the filler. These are any external or internal functions, with the exception of those whose names are "transferFrom" and "burnFrom".
5. Check balances inequalities, expiration time and other arbitrary conditions as specified by the orders.

Each of these steps is indicated with a comment in the code. Let us highlight some salient features emerging from this design.

- The orders do not specify transfer amounts nor recipients. This flexibility property allows partial execution and arbitrary routing of funds.
- The same order might be executed multiple times until the expiration time. To save gas, the contract does not store any data about the orders.
- Linear inequalities on user's balances also allow to specify minimum execution amount. Thus an order can forbid partial fills for itself.
- Other contracts functions can be called directly from this smart contract by the filler. This may include AMMs swaps and the transfers back to the users or the filler.

Notable use cases

Here we exhibit several important cases of varying degrees of complexity.

(1) Minimal case: maker-taker trade

A user wants to swap 5000 units of token A to 10000 units of token B. More precisely, the user signs an order to swap token A to token B at a rate at least 1:2, to obtain at most 10000 units of token B. A second user likes this price and takes the order. The second user is therefore the filler. The Dealer contract performs two transfers between these two users. This operation only consumes 105K gas units (using OpenZeppelin implementation of ERC20). The users benefit from the absense of any intermediaries other than the blockchain. If there were AMMs operating the pair A/B, whose spot price was 1:2, then none of the users could have obtained a better deal by swapping at the AMM. It would be worse for them because of higher gas cost, pool fees and non-zero price impact.

We emphasize that in this case the filler is not an intermediary.

(1') Partial fill

In many cases the filler user may want to take only a portion of the order. In this way, the full order can be processed by different fillers after some time. The ability to take partial orders is important, since there will typically be actors operating amounts with different orders of magnitude.

(2) Walrasian auction

Let us restrict our attention to a single trading pair A/B that is highly liquid. In this context, agents participating in a Walrasian auction submit to the auctioneer their offer/demand amounts at each price (i.e. as a function of the price). In an ideal Walrasian auction, the auctioneer finds the equilibrium price and clears every order at this price. Next we describe a system design approximating this ideal auction, that can be built on top of an IDEX.

Consider the following plausible scenario. There is an AMM of the constant product

type available, operating A/B with a 0.3% pool fee. There is a public entity, the auctioneer, that receives orders privately and generates fair market executions. Assume that at some point in time, there are 10M units of token A and 20M units of token B at the AMM. The filler entity (auctioneer) has received five orders:

(a) Sell 80K token A at a price at least 1.99.

(b) Sell 5K token A at a price at least 1.985.

(c) Sell 1K token A at a price at least 1.984.

(d) Buy 55K token A at a price at most 2.

(e) Buy 21K token A at a price at most 1.993.

In this situation, considering the orders and the AMM, the auctioneer's task is to find a fair execution based on a uniform price as follows. To each order, it will charge the gas cost, assumed to be 2 units of B (in current tests this cost is twice the cost of an ERC20 transfer), and then apply a small fee of 0.05%. To obtain a convenient AMM swap, it considers a gas cost of 3 units of B and also applies a 0.05% fee (i.e. the filler interprets AMM swaps in this way to solve the system). With these premises, the filler finds the solution price 1.9908 and the following execution:

(a) 80,000 ---> 159,182 (b) 5,000 ---> 9,947 (c) 1,000 ---> 1,987 (d) 55,500 <--- 110,542 (e) 21,000 <--- 41,825 (AMM) 9,500 <--- 18,925 (the AMM receives 9,500 of A and gives back 18,925 B). (F) The auctioneer keeps a surplus of 176 units of token B.

The left column indicates amounts of token A, while the right one indicates amounts of token B. Each line indicates the swap for the corresponding user, AMM or filler. Arrows from left to right multiply by 1.9898 and substract 2 units. Arrows from right to left substract 2 units (except the AMM case, 3 units) and then divide by 1.9918.

The total cost for this transaction is 456K gas units. For this test we used OpenZeppelin ERC20 and a Uniswap V2 pool. Notice that users receive an execution that is better than their signed limit price. This contrasts with the fact that orders sent to public mempools will probably get executed at the worst possible price. Thus, the advantage of such "walrasian auction solver" for the users is that, by signing a suboptimal price, they get both a good probability of execution and a fair price.

(3) Temporary AMMs

Any user can build a simple AMM by signing a pair of orders. Imagine a user that starts with 600K units of token B. The user signs:

(a) buy any amount of token A at a price at most 1.98 B/A.

(b) sell any amount of token A at a price at least 2 B/A.

Assume that the price starts at 1.99. If at a later time this price goes below 1.98, the order (a) will be filled by one or more actors, as soon as it is profitable to arbitrage this against any other order or source of liquidity, or maybe before that happens. If later on the price goes above 2, upon full execution of order (b) the user will make a profit of 6062 units of token B. The same orders can be executed again and again until the expiration time. By replacing inequalities by a condition pointing to another function, it is possible to set up AMMs defined by arbitrary curves. This could be practical for temporary liquidity providers, since in this way they avoid the cost of joining and exiting pools.

(4) Many tokens clearing

A batch of orders operating more than two tokens can be cleared at the same transaction. To our knowledge this is not possible to do in traditional exchanges. The simplest case would be a triangle, whose execution could look something like:

user1 ---1000A---> user2 user2 ---2000B---> user3 user3 ---3000C---> user1 user3 ---- 30C---> filler

where each user signed one order involving only one pair. In cases like this it is again crucial the ability to execute orders partially, since for almost all cases the amounts will not fit exactly.

(5) Swap fee paid with a different token

When a user wants to swap two rare tokens, it is possible that fillers do not support those to cover the gas cost. In this case, users may indicate a third token for that purpose. To do so, they have to include this third token in allowedTokens

, and add an inequality that bounds the fee amount according to the execution amount.

Limitations of linear inequalities

As we can see, linear inequalities as a standard condition format allow the most frequently needed order types. At the same time, their verification is simple and efficient. However, using linear inequalities and expiration times exclusively as order conditions might not be enough for non-technical users. The reason is that token balances may change if the user makes a transfer before the order's expiration time, leading to unwanted replays. There are different ways to mitigate this without adding other conditions, but it is possible that they do not fully solve the issue, depending on some hard to predict future tendencies. For the moment we prefer to limit the claim: for specialized users, it seems to be useful to express their orders in terms of linear inequalities and expiration times exclusively.

Private vs. public orderflow

In example (2), we saw that users may benefit from sending their orders to private entities. These can offer different types of protection, regarding price or expiration times. This includes small deviations of price and also large deviations in case of errors. They have the ability to delay or cancel orders. However it is not always better to use private channels. Public orders enjoy higher execution probability, simply because they are visible to anyone. Therefore we can expect to have many public orders as well, and this public information will be valuable to others.

Applications on top of IDEX smart contracts

Every blockchain smart contract can be operated from different interfaces. This is a powerful feature of decentralized systems. One class of applications to interact with IDEX smart contracts is that of user interfaces (UIs). They would not only encode orders to be signed, but also assist the users in decisions such as price and expiration date, with suggestions and warnings. They can also suggest different destinations for the orders, either private or public. Moreover UIs can show pending orders that users can fill directly or use as a reference.

Another interesting class of applications are those lying at an intermediate layer between the UI and the settlement smart contract. In some cases, applications of this type will need to handle orders in a predetermined way, sometimes managing their privacy status. We envision SUAVE as a natural platform for hosting this type of programs as MEVM smart contracts. One example is the auctioneer from the above use case (2). It might be possible to generalize this to many tokens instead of only one pair. Another example is shield applications aimed at protecting users from errors in their signed orders. This would mainly be composed by smart contracts that perform security verifications on the orders, preserving their privacy when needed, and possibly pushing them forward if they are correct. We expect this class of applications to be an active field for research and development in the years to come.