FarFarAway Swap

Team of Kryptosphere INSA Lyon

March 20, 2022

Abstract

This whitepaper explains how FarFarAway Swap works, key features and principles of a project. We will cover why FarFarAway can be profitable for its users, how the brand new combined swap algorithm works and our researches, all the technologies that power the project.

1 Introduction

Most DEX platforms use two models of exchange: automated market maker (AMM) like liquidity pools and order book based exchanges. Both models have some disadvantages [**sotatek**]:

- Illiquid market makes traders have difficulty in finding matching orders due to large bid-ask spread and low trading volumes. If you want to have successful orders, you have to make sure that your highest bid is lower than the lowest ask.
- AMM causes high slippage for large orders: Slippage relies on the liquidity pool's size of a certain trading pair. The liquidity pool needs to be 100x greater than the size of the order to keep the slippage rate under 1%.

These problems could be solved by implementing both models at the same time. Now it's a good moment to present FarFarAway Swap. Let's talk a little bit of the original idea of this project. The idea of FarFarAway Swap was just to use the next token price difference (like on schema) between pool and order book price and choose whether we buy it from pool or from order book.

While developing this idea, we found out that implementing both on the same platform would be a great idea. So here came the combined swap.

FarFarAway Swap is decentralized crypto exchange platform (DEX) meant to be your money saver when buying crypto. Powered by Near, we provide 1000x lower transaction fees, scalable and secure swaps. Our key feature is our **combined swap algorithm** that solves two mentioned model's inefficiencies. It allows users to get the best token prices by buying one part from pool and another part from the order book. Which part? Only algorithm knows.

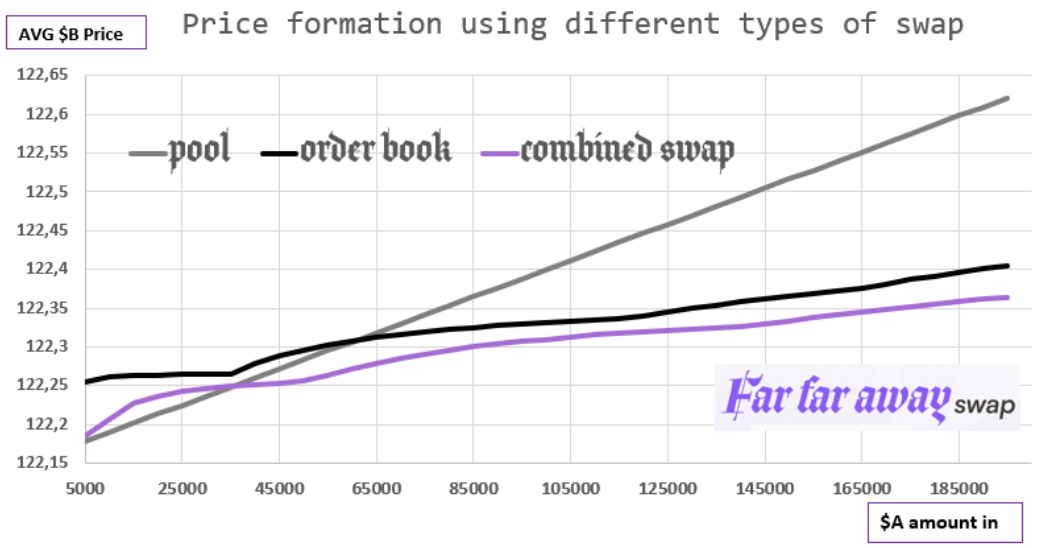
2 Researches

2.1 Price formation

Our goal is providing the best price by using combined swap. To prove the the utility of these swaps, we present you our simulations of market price formation when buying in three different ways:

- from pool only
- from order book only
- combined swap

Figure 1: $B reserves 215K, $A reserves 26M



Here is the chart explaining the price formation.

As you see, when we use combined swap, at one moment we reach a better price compared to order book and pool. Now let's talk about algorithms behind these simulations and how they could work in production.

To make this very beautiful chart we had to simulate the one-moment **abstract market state**. On this market we have the total amount $B = N and $A = M and 0.3% transaction fee. How did we estimate N and M constants? Easily, this data was downloaded and parsed from Binance, so once we reach their turnover, we know how to deal with it.

When buying

- from pool only: we have pool reserves $A=M and $B=N
- from order book only: we use Binance's 98K limit sell orders when in total people sell N $B starting with token price = M/N
- combined swap: we have pool reserves $A=M/2 and $B=N/2 and we use the same order book, but each $B amount is divided by 2

How do our pool works? Exactly the same as on Uniswap : Check here

2.2 Combined swap : pool ratio estimation

The idea of combined swap is just swapping tokens from pool and order book at the same time. **How to estimate which part must be taken from pool?** For all the simulations we used a very simple python script that iterates trough all possible ratios and chooses the one who gives us the best $B amount out.

```
def get_optimal_ratio(
        amount_inA: float,
        orders: list,
        reserves_in,
        reserves_out,
        step=100) -> list:
    """
    calculation of $A ratio sold in pool to get maximum of $B
    param orders: test order book data
    param step: the precision of algorithm depends on this param

    returns: [
        <ratio of $A sold in pool to get the max of $B>,
        <average $B price>
        ]
    """

    amounts_out = []
    steps = numpy.arange(step, amount_inA, step)

    for amount_in_pool in steps:
        test_orders = orders.copy()
        pool_amount_out = get_amount_out_pool(
            amount_in_pool, reserves_in, reserves_out
        )
        ob_amount_out = get_amount_out_ob(
            amount_inA - amount_in_pool, test_orders
        )
        amounts_out.append(pool_amount_out + ob_amount_out)

    max_amount_out = max(amounts_out)
    max_profit_a_pool = list(steps)[amounts_out.index(max_amount_out)]
    max_profit_a_pool_ratio = max_profit_a_pool/amount_inA
    token_b_price = amount_inA/max_amount_out

    return [max_profit_a_pool_ratio, token_b_price]
```

You can see that this algorithm is not efficient at all - O(n) complexity and too much memory usage. **However, in production we can avoid using it. How? (next page)**

Depending on reserves in pool and order book data, the best ratio curve can take these forms.

- X-axis: total $A we are swapping.
- Y-axis: percentage of $B we buy from pool

As you can see, starting from 10'000 $A swapping, these charts all look like $y = k/x^n$. This allows us to say that depending on liquidity in pool and order book, we will be able to predict the percentage of tokens we swap in pool due to mathematical model. **In this case pool ratio calculation will have O(1)complexity.** We will talk about it in roadmap section.

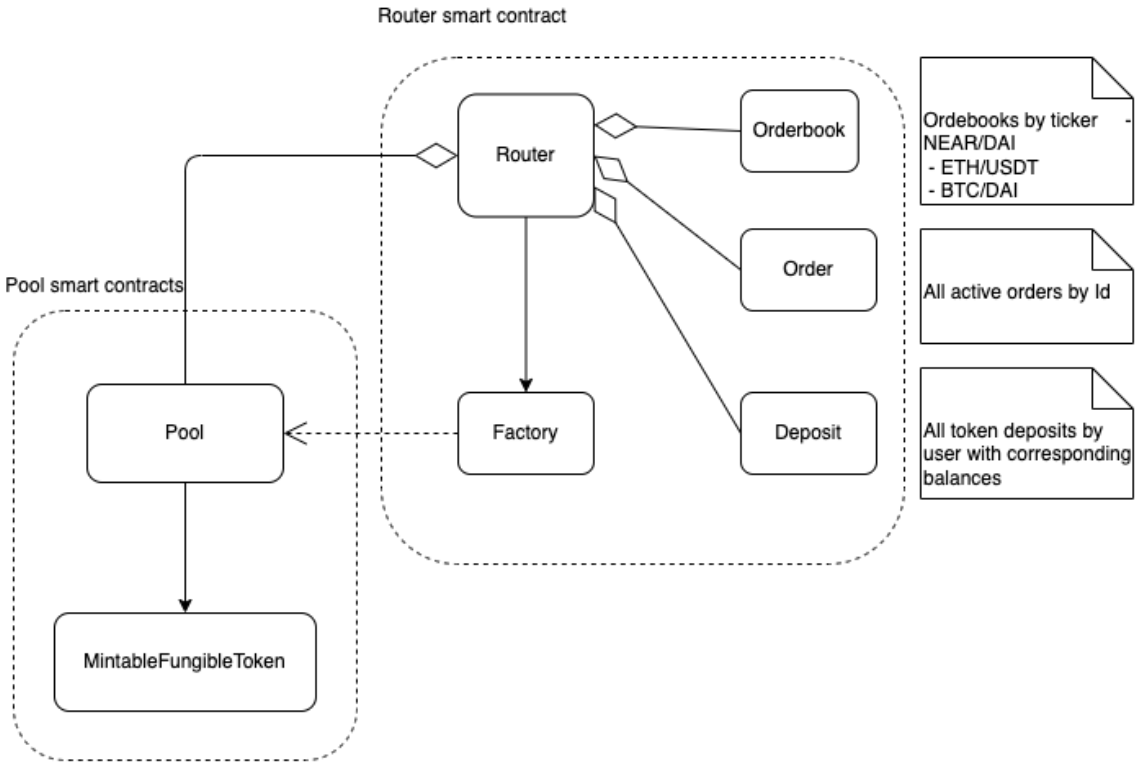
3 Implementation

3.1 Under the hood

FarFarAway Swap is built on Near, providing security, cost-efficiency and scalability. For Front-end part we use Svelte, Typescript, JS. Back-end is on Rust.

There is four main smart contracts:

- Router
- Factory
- Orderbook
- Pool



3.2 Data structures

deposits - This structure keeps all user's deposits mapped by the user account_id. The struct Deposit contains all tokens by their token_id and balances.

order_books - Orderbooks are stored into the UnorderedMap mapped by ticker (e.g. "BTC/DAI"). It has been chosen because it supports iteration over keys and values (tickers and orderbooks).

orders_by_account - This structure allows to instantly retrieve (O(1) complexity) all order_ids by user account_id.

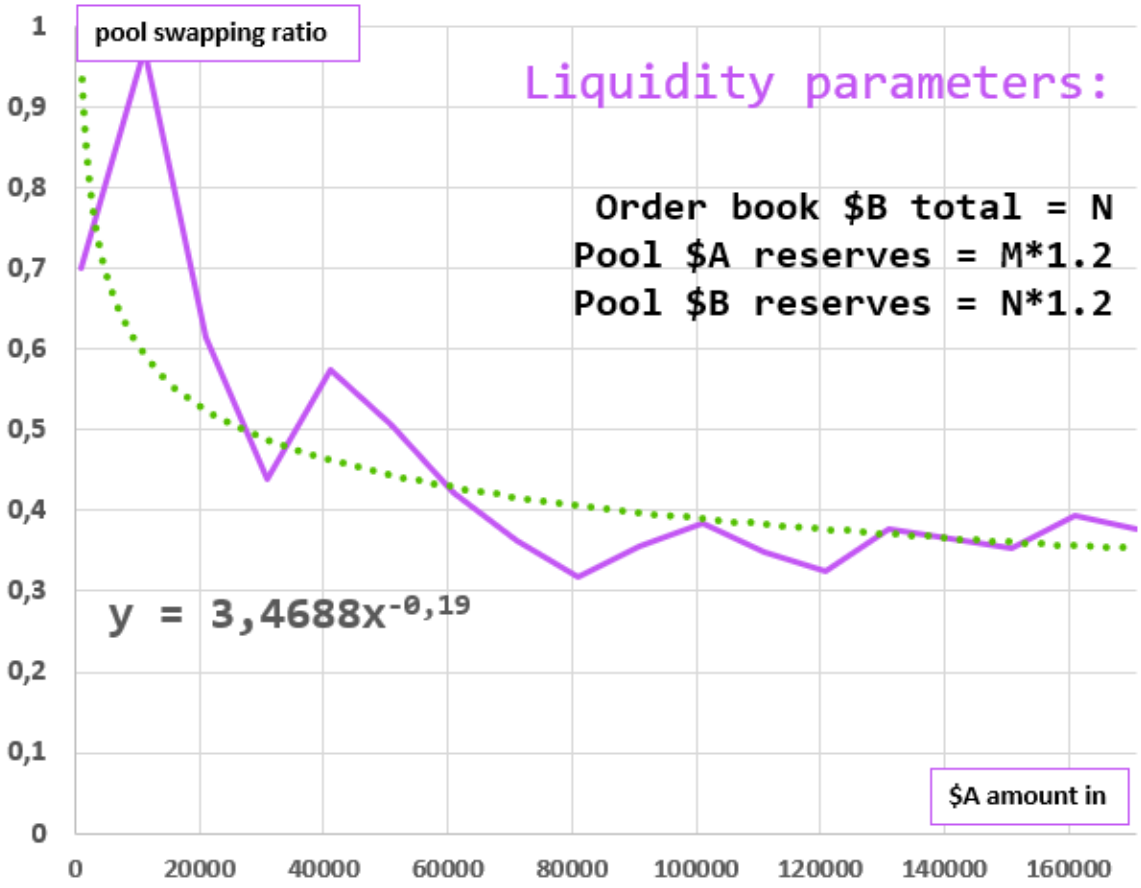
orders_by_id - To fetch order details with a constant complexity (O(1)), we use UnorderedMap mapped by order_id and Order struct as a value.

pools - Once a new pool has been created, it's smart contract account_id is stored into this structure in order to call it later.

4 Roadmap

4.1 Combined swap algorithm improvement

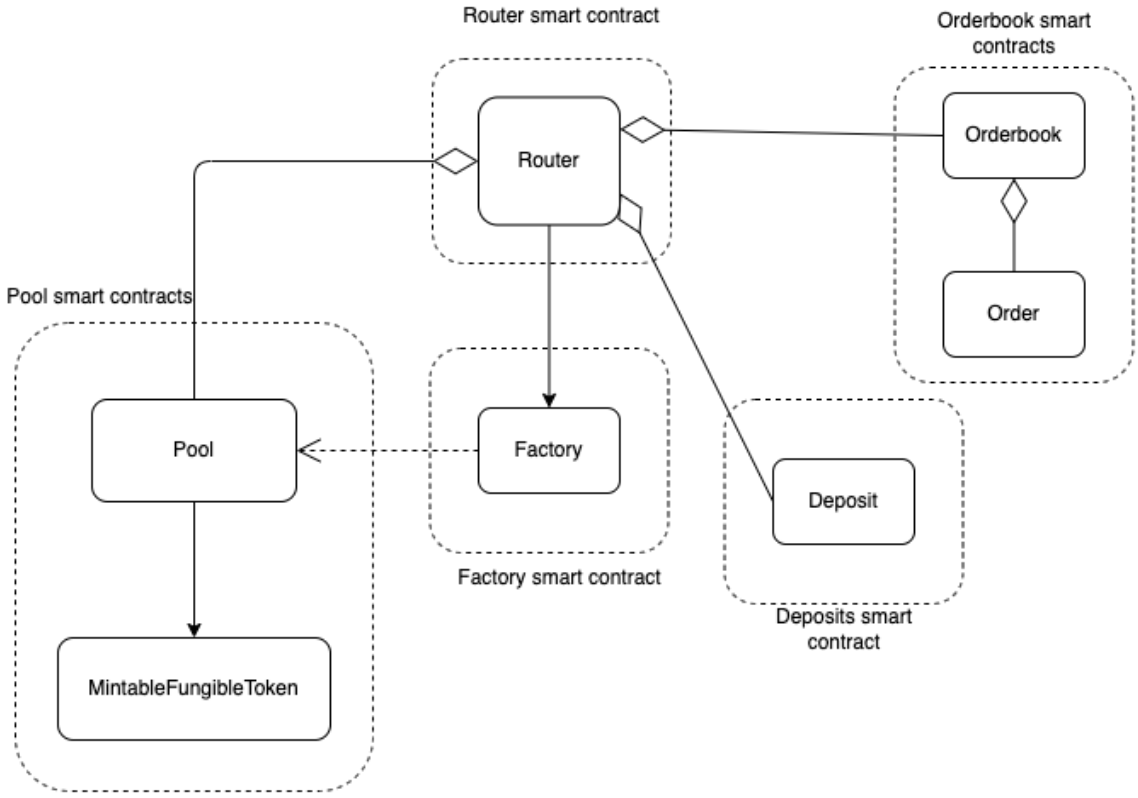
As you could see previously, on all the ratio / swap amount curves we can apply a power trendline $y = k/x^n$ like here:



In the future we will find the mathematical models for k and n depending on pool and order book liquidity. **This will allow us to provide O(1) complexity for pool ratio calculations for every pair of tokens.**

4.2 Security & optimisation

In the future we will use our V2 architecture which is a little bit different:



In order to optimize and make more scalable our DEX, we will implement a sort of "sharded app design" inspired by META Near.

References

[1] *sotatek.com, WHICH ONE IS MORE EFFECTIVE FOR DEX DEVELOPMENT: ORDER BOOK OR AMM?*

[2] *order book vs AMM which will win*