Delio

Swap

Q1. 이용자가 ETH에서 DAI로 스왑을 요청했을 때 수수료를 차감합니다. 이때 수수료는 ETH에서 수취합니까? DAI에서 수취합니까?

The fee is subtracted from ETH.

We can find the justification of our answer in both `uniswap-v2-core` and `uniswap-v2-periphery` set of contracts.

Uniswap V2 contracts introduced a separation between the core and periphery contracts, where the core contracts are minimalist in design, so logic related to trader security and ease-of-use are implemented in external helper contracts that can be improved and replace without needing to migrate liquidity.

The repository named `uniswap-v2-core` contains the essential Uniswap V2 contract consisting of

- `UniswapV2Pair.sol`, which implements core swapping and liquidity provisioning functionality
- `UniswapV2Factory.sol`, which deploys `UniswapV2Pair.sol` contracts for any ERC20 token/ERC20 token pair.

The repository named `uniswap-v2-periphery` is set of helper contracts that include among others,

- `UniswapV2Router02.sol` that performs the safety checks needed for safely swapping, adding, and removing liquidity
- `UniswapV2Library.sol` that provides a variety of convenience functions for fetching data and pricing.

Answer 1 (`uniswap-v2-periphery`)

We will first answer the question by looking at the periphery contracts since the contracts there more closely map to human trader activity such as swapping or providing liquidity.

The function `getAmountOut` in `UniswapV2Library.sol` takes as *input* asset amount, and returns the maximum *output* amount of the other asset (accounting for fees) given reserves. We can then infer the exact fee amount and where the fee is taken from.

```
// given an input amount of an asset and pair reserves, returns the maximum output
amount of the other asset
function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) internal pure
returns (uint amountOut) {
    require(amountIn > 0, 'UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT');
    require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library:
INSUFFICIENT_LIQUIDITY');
    uint amountInWithFee = amountIn.mul(997);
    uint numerator = amountInWithFee.mul(reserveOut);
    uint denominator = reserveIn.mul(1000).add(amountInWithFee);
    amountOut = numerator / denominator;
}
```

Derivation of `getAmountOut`

To derive the above function from first principles of $x * y = k$, let
$$(1 - \phi) := \textrm{percentage fee that is paid to liquidity providers}$$
so in Uniswap V2's case of 0.03% fee, $\phi = 0.997$ and represented as 997 in the function body of `getAmountOut`.

Consider the case of swapping token0 for token1. We want to figure out the amount of token1 Δy we can buy from Uniswap by selling Δx of token0. Since we pay a fee of $(1 - \phi)$ to the liquidity providers, the actual amount of token0 that goes into the liquidity pool is $\phi \Delta x$. Then, after the swap the token reserves for token0 should be $(x + \phi \Delta x)$ and $(y - \Delta y)$ for token1.
Now, since the product of token reserves must be constant before and after the swap,

$$
\begin{aligned}
(x + \phi\Delta x) * (y - \Delta y) &= k \\
(x + \phi\Delta x) * (y - \Delta y) &= xy \\
y - \Delta y &= \frac{xy}{x + \phi\Delta x} \\
\Delta y &= y - \frac{xy}{x + \phi\Delta x} \\
\Delta y &= \frac{y\phi\Delta x}{x + \phi\Delta x}.
\end{aligned}
$$

Hence when swapping x for y, we get $\Delta y = \frac{y \phi \Delta x}{x + \phi \Delta x}$. We can now map $\phi \Delta x$ to `amountInWithFee`, $y \phi \Delta x$ to `uint numerator = amountInWithFee.mul(reserveOut)`, $x + \phi \Delta x$ to `uint denominator = reserveIn.mul(1000).add(amountInWithFee)`, and $\Delta y = \frac{y \phi \Delta x}{x + \phi \Delta x}$ to `amountOut = numerator / denominator`. We can do a similar exercise to derive the function body of `getAmountIn`.

Answer 2 (`uniswap-v2-core`)

Fee accounting can also be seen in the `swap` function of the core contract `UniswapV2Pair.sol`. Notice the following lines in the `swap` function body that subtracts 0.03% of `amountIn` as fee before assigning adjusted token balances to `balance{0,1}Adjusted`:

```
uint balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3));
uint balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3));
```

As an aside, we note that Uniswap requires `swap` function callers to specify how many output tokens they would like to recieve via the `amount{0,1}Out` parameters. This is to facilitate flash swap ("flash loan") during which it is possible to withdraw positive amounts of both `token{0,1}`.

Q1-2. 수취된 수수료는 어디에 어떤 방식으로 저장되나요? ex) ETH에서 수취한 수수료는 Pool 내의 ETH Balance를 저장하는 곳에 함께 저장됩니다.

The fee is "held" by the `UniswapV2Pool` contract.

ERC-20 Mechanics

To be more precise in our answer, let's go over some basics of ERC-20 token mechanics. Implementations of ERC-20 tokens have a data structure that records the amount of tokens the various address hold. For example, in OpenZeppelin's implementation of ERC20, there is a member variable `_balances` of type `mapping(address => uint256) private` that maps addresses to positive integer values (balance). The '_' in front of the variable name means that clients should interface with the internal data structure with public or external methods. The most common method of interaction is to ask the token contract for the balance held by an address; as a metter of fact, their implementation of `balanceOf` does a simple lookup from the mapping (table):

```
function balanceOf(address account) public view virtual override returns (uint256) {
    return _balances[account];
}
```

Each implementation of ERC-20 varies and in fact, Uniswap's ERC-20 liquidity pool token implementation is the core contract `UniswapV2ERC20.sol`.

`UniswapV2Pair` member variables

`UniswapV2Pair` contract keeps track of its token reserves and token addresses as member variables:

```
address public token0;
address public token1;
uint112 private reserve0;
uint112 private reserve1;
```

The reserve balances and price oracle data are kept up to date by invoking

```
function _update(uint balance0, uint balance1, uint112 _reserve0, uint112 _reserve1)
private
```

at the end of core state-changing functions such as `mint`, `burn`, and `swap`.

Q2. 스왑시 '유동성 제공자 수수료'라고 표기 되는 값은 어떻게 결정되나요?

[0.03% of `amountIn` obviously?
TODO: But to be absolutely sure look at `uniswap-interface` and corresponding sdk function calls.]

Q3. 스왑요청을 처리하는 방식은 어떻게 되어 있나요? ex) 시간순서대로 먼저 요청이 온것을 처리하는지, 스왑요청 수량이 큰 순서대로 처리하는지, 앞 내용들의 하이브리드 방식인지.

[Is this not a question of gas prices? Won't miners mine transactions with high gas prices first unless you are a miner or have an off-chain agreement with a miner? Will EIP-1559 (London hardfork) change this?]

Q4. 스왑 '경로'는 무엇을 의미 합니까? 스왑 설정에서 확인할 수 있는 '멀티 홉'이 앞에서 질문한 스왑 '경로'를 결정하는데 영향을 미치는 요소 입니까?

[TODO: uniswap-sdk?]

Q4-2. 스왑 '경로' 대상이 선택되는 기준이 무엇입니까? ('Uniswap V3 : Router' 컨트랙트를 참조하세요)

[TODO: uniswap-sdk?]

Q5. 스왑에서 '가격영향'은 스왑의 결과값에 어떻게 영향을 미칩니까?

'Price impact' denotes the difference between mid-price (ratio of current token reserves) and the execution price of a trade.

[Displays the implied change in price according to x * y = k formula?]

Q6. 스왑에서 '슬리피지 허용 오차'는 스왑의 결과값에 어떻게 영향을 미칩니까?

- Swap rate is not locked in. The actual swap rate is the rate at the time that the transaction is included in a block.
- "Slippage" is defined as how much the swap rate has gotten worse between when ir is submitted and when it is executed.
- User can set "slippage tolerance" to cause the transaction to fail if the rate has moved too much.

Q7. 스왑할 때 '최소 수량'은 무엇에 영향을 받아 결정됩니까?

- "Minimum received" is closely related to the concept of "slippage tolerance."
- When interacting with Uniswap's interface, it sets `amountOutMin` in the swap function as in

```
function swapExactTokensForTokens(
    uint amountIn,
    uint amountOutMin,
    address[] calldata path,
    address to,
    uint deadline
)
```

of `uniswap-v2-periphery/.../UniswapV2Router02.sol`. The function checks

```
 require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router:
INSUFFICIENT_OUTPUT_AMOUNT');
```

- [`amountOutMin = (1 - slippage_tolerance) * amountOut` where `slippage_tolerance = 0.003`?]

Q8. 실제 교환비와 스왑 요청 시 받을수 있는 수량비가 다른 이유는 무엇입니까? ex) Uniswap에서 수수료를 부과해서 or 유동성 제공 수수료 때문에 등등...

The answer to this question has more to do with nature of dApps and decentralization; the swap rate is not locked in until the transaction has been included in a block and mined. For more in depth discussions we refer the reader to the official Uniswap V2 audit report: Front-Running and Transaction Reordering.

Q9. 설정에서 '거래 마감 시간'이 스왑요청을 처리하는 방식에 어떤 영향을 미치나요?

`UniswapV2Router02.sol` defines a modifier

```
modifier ensure(uint deadline) {
    require(deadline >= block.timestamp, 'UniswapV2Router: EXPIRED');
    _;
}
```

that decorates most if not all functions contained in the contract.

From Uniswap:

"If a swap takes more than 20 minutes to execute, the router is programmed to fail the transaction. This is to protect the user from extreme swings in prices that can occur while the transaction is pending. If this happens, your tokens will still be in your wallet, but the gas fees paid are not recoverable. To keep this from happening, use a high enough gas price to have your transaction mined in under 20 minutes. This usually falls under “Standard” or “Fast” in most gas price calculators."

Add liquidity

Q13. 유동성 공급 시 설정조건에 '슬리피지 허용 오차'가 들어가 있는 이유는 무엇입니까? 질문의도) 유동성 풀에 입금액은 기존 Pool의 교환비를 반영하는데, '슬리피지 허용 오차' 가 설정 값에 들어가는 이유가 없을것 같음.

Q14. 전문가 모드 전환(높은 가격 영향 거래 허용) on 인 경우 어떤 SC가 관계하는가?

Q14-2. 이때 LP가 제공한 유동성은 자동으로 가격범주가 설정되는데, 위 관계하는 SC 때문인가?

Common

Q16. 유니스왑 거래내역의 총 계산된 fee와 Etherscan 상의 fee는 상이한 부분이 있는가?

Q16-2. Transaction fee, Gas fee, Network fee 등을 각각 어떻게 구분하고 이는 유니스왑과 Etherscan 상 동일하게 표현하고 있는가?