Messages

Deposit

Overview

In Neutron DEX's concentrated liquidity model, liquidity providers (LPs) can provide liquidity to a specific trading pair by depositing tokens at a specific price into one or both sides of the pair in "a liquidity pool". When depositing into the pool LPs must choose a fee for their deposit. This fee will be paid every time a trader swaps through a liquidity pool. When a trader wants to buy or sell one of the tokens in the pair, they must first pay the pool fee, which is a percentage of the trade value, to the liquidity pool. This fee is then shared among the LPs in proportion to their share of the liquidity in the pool. By providing liquidity, LPs earn a portion of the transaction fees and may also earn incentives for depositing in certain liquidity pools. In a concentrated liquidity DEX, LPs can set their own price range for each token in the pair. This means that they can set a higher price for selling their token and a lower price for buying the other token, or vice versa. This allows LPs to earn more fees by capturing the spread between the buying and selling prices. However, providing liquidity in a concentrated liquidity DEX can also carry risks, such as impermanent loss, where the value of the tokens in the pool may diverge from their initial ratio due to price fluctuations. LPs may also be exposed to potential losses if the DEX is hacked or suffers from smart contract vulnerabilities. It is important for LPs to carefully consider these risks before providing liquidity.

Deposit Mechanism

When depositing into an LP position (PoolReserves) a user specifies amounts of Token0 and Token1 as well as a TickIndex and a fee. The liquidity is added to the reserves for the respective ticks. Token0 will be deposited into thePoolReserves struct with the matching fee atTickIndex - fee and Token1 will be deposited into thePoolReserves atTickIndex + fee .

In the most basic case, when depositing into a pool, the ratio of Token0 to Token1 will be preserved. If a user does not provide tokens in the same ratio, then only a portion of their total deposit will be used so as to maintain the pool ratio.

t r u e 0 = m i n (a m o u n t D e p o s i t e d 0 , e x i s t i n g R e s e r v e s 0 · a m o u n t D e p o s i t e d 1 e x i s t i n g R e s e r v e s 1) true_0 = min(amountDeposited_0, \frac{existingReserves_0 \cdot amountDeposited_1}{existingReserves_1}) t r u e 0

= min (am o u n t De p os i t e d 0 ,

e x i s t in g R eser v e s 1

e x i s t in g R eser v e s 0 · am o u n t De p os i t e d 1) t r u e 1 = m i n (a m o u n t D e p o s i t e d 1 , e x i s t i n g R e s e r v e s 1 · a m o u n t D e p o s i t e d 0 e x i s t i n g R e s e r v e s 0) true_1 = min(amountDeposited_1, \frac{existingReserves_1 \cdot amountDeposited_0}{existingReserves_0}) t r u e 1

= min (am o u n t De p os i t e d 1 ,

e x i s t in g R eser v e s 0

e x i s t in g R eser v e s 1 · am o u n t De p os i t e d 0)

NOTE : Most pools will only have Token1 OR Token0, so most deposits will only be providing one tokens to one side of the pool.

In return for depositing tokens into a pool the user is issued PoolShares corresponding to the specific liquidity pool in which they have deposited. These can be used in the future to withdraw the user's pro-rata share of the PoolLiquidity in the future. The PoolShares are also fungible denoms that can be bought, sold, and traded.

The amount of pool shares issued is calculated using the following formula:

v a l u e D e p o s i t e d = t r u e 0 + p (i) · t r u e 1 valueDeposited = true_0 + p(i)\cdot true_1 v a l u eDe p os i t e d

= t r u e 0

+ p (i)

· t r u e 1

n e w S h a r e s = v a l u e D e p o s i t e d · t o t a l S h a r e s v a l u e T o t a l newShares = \frac{valueDeposited \cdot totalShares}{valueTotal} n e wS ha res

= v a l u e T o t a l

v a l u eDe p os i t e d · t o t a lS ha res

Autoswap

By default theautoswap option is enabled, which allows use to deposit their full deposit amount. Autoswap provides a mechanism for users to deposit the entirety of their specified deposit amounts by paying a small fee. The fee for performing an autoswap is deducted from the total number of shares the the user is issued. When calculating share issuance the same formula as above is used for the balanced portion of the deposit, with the following formula use to calculate the shares issues that would unbalance the pool:

a d d i t i o n a l S h a r e s = l e f t 0 · p (– f e e) + l e f t 1 · p (i – f e e) additionalShares = left_0 \cdot p(-fee) + left_1 \cdot p(i-fee) a dd i t i o na lS ha res

= l e f t 0

· p (– f ee)

+ l e f t 1

· p (i

– f ee)

Deposit Message

message

MsgDeposit

{ string creator =

1 ; string receiver =

2 ; string token_a =

3 ; string token_b =

4 ; repeated

string amounts_a =

5

[(gogoproto . moretags)

=

"yaml:\"amounts_a\"" , (gogoproto . customtype)

=

"github.com/cosmos/cosmos-sdk/types.Int" , (gogoproto . nullable)

=

false , (gogoproto . jsontag)

=

"amounts_a"] ; repeated

string amounts_b =

6

[(gogoproto . moretags)

=

"yaml:\"amounts_b\"" , (gogoproto . customtype)

=

"github.com/cosmos/cosmos-sdk/types.Int" , (gogoproto . nullable)

=

false , (gogoproto . jsontag)

=

"amounts_b"] ; repeated

int64 tick_indexes_a_to_b =

7 ; repeated

uint64 fees =

8 ; repeated

DepositOptions options =

9 ; }

MsgDeposit

Feild Description Creator string (sdk.AccAddress) The account from which deposit Tokens will be debited Receiver string (sdk.AccAddress) The account to which PoolShares will be issued TokenA string Denom for one side of the deposit TokenB string Denom for the opposing side of the deposit AmountsA []sdk.Int Amounts of tokenA to deposit AmountsB []sdk.Int Amounts of tokenB to deposit TickIndexesAToB []int64 Tick indexes to deposit at defined in terms of TokenA to TokenB (ie. TokenA is on the left) Fees []uint64 Fees to use for each deposit Options []DepositOptions Additional deposit options

DepositOptions

Field Description Autoswap bool Toggle to use autoswap (default true)

MultiHop Swap

Overview

Multihop swap provides a swapping mechanism to achieve better prices by routing through a series of pools.Rather than swapping directly from TokenA to TokenB a user may be able to get a better price if they swap from TokenA to TokenC to TokenD to TokenB. When performing a multohop swap the user provides an array of Denoms that they would like to swap through. For example if the user supplied the following array of Denoms[“TokenA”, “TokenC”, “TokenD”, “TokenB”], the following swaps would be performed: Swap TokenA for TokenC; Swap TokenC for TokenD; Swap TokenD for TokenB.

The underlying swaps (hops) within a Multihop Swap are performed using the same mechanism as the basic Swap function. Unlike the the basic swap however, the complete amount of specified byAmountIn will always be used. If there is insufficient liquidity in a route to swap 100% of theAmountIn the route will fail. Additionally, rather than supply an explicit argument forTokenIn the first denom in eachRoutes array is used as theTokenIn.

MultihopSwap also allows users to set anExitLimitPrice. For a route to succeed the final conversion rate for the entry token and ExitToken must be less than the ExitLimitPrice. For a Multihop swap to succeed the following test must be satisfied:

E x i t L i m i t P r i c e < = A m o u n t O f E x i t T o k e n A m o u n t I n ExitLimitPrice <= \frac{AmountOfExitToken}{ AmountIn} E x i t L imi tP r i ce

<= A m o u n t I n

A m o u n tO f E x i tT o k e n

Multihop swap also allows users to supply multiple different routes. By default, the first route that does not run out of liquidity, hit theExitLimitPrice or return an error will be used. Multihop swap also provides aPickBestRoute option. WhenPickBestRoute is true all routes will be run and the route that results in the greatest amount of the resulting TokenOut for at final hop will be used. This option to dynamically pick the best route at runtime significantly reduces the risk of front running.

Multihop Swap Message

message

MultiHopRoute

{ repeated

string hops =

1 ; }

message

MsgMultiHopSwap

{ string creator =

1 ; string receiver =

2 ; repeated

MultiHopRoute routes =

3 ; string amount_in =

4

[(gogoproto . moretags)

=

"yaml:\"amount_in\"" , (gogoproto . customtype)

=

"github.com/cosmos/cosmos-sdk/types.Int" , (gogoproto . nullable)

=

false , (gogoproto . jsontag)

=

"amount_in"] ; string exit_limit_price =

5

[(gogoproto . moretags)

=

"yaml:\"exit_limit_price\"" , (gogoproto . customtype)

=

"github.com/neutron-org/neutron/v2/utils/math.PrecDec" , (gogoproto . nullable)

=

false , (gogoproto . jsontag)

=

"exit_limit_price"] ; // If pickBestRoute == true then all routes are run and the route with the // best price is chosen otherwise, the first succesful route is used. bool pick_best_route =

6 ; }

MsgMultiHopSwap

Field Description Creator string (sdk.AccAddress) Account from which TokenIn is debited Receiver string (sdk.AccAddress) Account to which TokenOut is credited Routes []MultiHopRoute Array of possible routes AmountIn sdk.Int Amount of TokenIn to swap ExitLimitPrice sdk.Dec Minimum price that that must be satisfied for a route to succeed PickBestRoute bool If true all routes are run and the route with the best price is used Multihop Route

Field Description Hops []String Array of denoms to route through

Place Limit Order

Overview

Limit orders provide the primary mechanism for trading on the Neutron DEX Dex. Limit orders can provide liquidity to the Dex (“Maker LimitOrders”) and/or can be used to trade against preexisting liquidity (“Taker Limit Orders”). All limit orders provide aTickIndex which serves as a limit price for which the order will be executed at.

Maker limit orders provide new liquidity to the dex that can be swapped through by other traders (either via Multihop Swap or a Taker Limit Order.) The liquidity supplied by a maker limit order is stored in aLimitOrderTranche at a specific tick. Once the tranche has been fully or partially filled via another order the user can withdraw the proceeds from that tranche. Maker limit orders can also be cancelled at any time. Maker only limit order’s are created with the following order types: GOOD_TIL_CANCELLED (will first try to satisfy the order via a taker limit order and only create an maker order if there is insufficient liquidity available above the providedTickIndex) JUST_IN_TIME GOOD_TIL_TIME

Taker limit orders do not add liquidity to the dex, instead they trade against existingTickLiquidity . Taker orders will either fail at the time of transaction or be completed immediately. Successful taker orders will deposit the proceeds directly back into the receiever’s address.

Rather than supplying a limit price, limit orders take aTickIndex as an argument. For maker limit orders this is the tick that theLimitOrderTranche will be placed at. For taker limit order will only trade through liquidity at or above theTickIndex. A specific price can be converted to aTickIndex using the following formula:

T i c k I n d e x = l o g 1.0001 (p r i c e) TickIndex = log_{1.0001}(price) T i c k I n d e x

= l o g 1.0001 (p r i c e)

Order types

FILL_OR_KILL

Fill-or-Kill limit orders are maker limit orders that either successfully swap 100% of the suppliedAmountIn or return an error. If there is insufficient liquidity to complete the trade at or above the suppliedTickIndex a Fill-or-Kill order will return an error ofErrFoKLimitOrderNotFilled.

IMMEDIATE_OR_CANCEL

Immediate-or-Cancel limit orders are maker orders that will swap as much as of theAmountIn as possible given available liquidity above the suppliedTickIndex . Unlike Fill-or-Kill orders they will still successfully complete even if they are only able to partially trade through theAmountIn at theTickIndex or better.

GOOD_TIL_CANCELLED;

Good-til-Cancelled limit orders are hybrid maker and taker limit orders. They will attempt to trade the suppliedAmountIn at theTickIndex or better. However, if the totalAmountIn cannot be traded at the limit price they remaining amount will be placed as a maker limit order. The proceeds from the taker portion are deposited into the user’s account immediately, however, the proceeds from the maker portion must be explicitly withdrawn via WithdrawLimitOrder.

GOOD_TIL_TIME;

Good-til-Time limit order function exactly the same as Good-til-Cancelled limit orders first trying to trade as a taker limit order and then placing any remaining amount as a maker limit order. However, the maker portion of the limit order has a specifiedExpirationTime. After the ExpirationTime`the order will be cancelled and can no longer be traded against. When withdrawing a Good-til-Time limit order the user will receive both the successfully traded portion of the limit order (TokenOut) as well as any remaining untraded amount (TokenIn).

JUST_IN_TIME;

Just-in-Time limit orders are an advanced maker limit order order that provides tradeable liquidity for exactly one block. At the end of the same block in which the Just-in-Time order was submitted the order is canceled and any untraded portion will no longer be usable as active liquidity.

PlaceLimitOrder Message

message

MsgPlaceLimitOrder

{ string creator =

1 ; string receiver =

2 ; string token_in =

3 ; string token_out =

4 ; int64 tick_index_in_to_out =

5 ; string amount_in =

7

[(gogoproto . moretags)

=

"yaml:\"amount_in\"" , (gogoproto . customtype)

=

"github.com/cosmos/cosmos-sdk/types.Int" , (gogoproto . nullable)

=

false , (gogoproto . jsontag)

=

"amount_in"] ; LimitOrderType order_type =

8 ; // expirationTime is only valid iff orderType == GOOD_TIL_TIME. google . protobuf . Timestamp expiration_time =

9

[(gogoproto . stdtime)

=

true , (gogoproto . nullable)

=

true] ; string max_amount_out =

10

[(gogoproto . moretags)

=

"yaml:\"max_amount_out\"" , (gogoproto . customtype)

=

"github.com/cosmos/cosmos-sdk/types.Int" , (gogoproto . nullable)

=

true , (gogoproto . jsontag)

=

"max_amount_out"] ; }

MsgPlaceLimitOrder

Field Desciption Creator string (sdk.AccAddress) Account from which TokenIn is debited Receiver string (sdk.AccAddress) Account to which TokenOut is credited or that will be allowed to withdraw or cancel a maker order TokenIn string Token being “sold” TokenOut Token being “bought” TickIndex int64 Limit tick for a limit order, specified in terms of TokenIn to TokenOut AmountIn sdk.Int Amount of TokenIn to be traded OrderType orderType Type of limit order to be used. Must be one of: GOOD_TIL_CANCELLED, FILL_OR_KILL, IMMEDIATE_OR_CANCEL, JUST_IN_TIME, or GOOD_TIL_TIME ExpirationTime time.Time Expiration time for order. Only valid for GOOD_TIL_TIME limit orders

Cancel Limit Order Message

Overview

Standard Taker limit orders (Good-til-cancelled & Good-til-Time) can be canceled at any time if they have not been completely filled. Once a limit order is canceled any remaining “TokenIn” liquidity is returned to the user.

NOTE: Canceling a partially filled limit order does not withdraw the traded portion. A separate call must be made toWithdrawFilledLimitOrder to withdraw any proceeds from the limit order

Cancel Limit Order Message

message

MsgCancelLimitOrder

{ string creator =

1 ; string tranche_key =

2 ; }

MsgCancelLimitOrder

Field Description Creator string (sdk.AccAddress) Account which controls the limit order and to which any untraded amount is credited TrancheKey string TrancheKey for the target limit order

Withdraw Filled Limit Order

Overview

Once a limit order has been filled – either partially or in its entirety, it can be withdrawn at any time. Withdrawing from a limit order credits all available proceeds to the user. Withdraw can be called on a limit order multiple times as new proceeds become available. Withdraw Filled Limit Order Message

Withdraw Filled Limit Order Message

message

MsgWithdrawFilledLimitOrder

{ string creator =

1 ; string tranche_key =

2 ; }

MsgWithdrawFilledLimitOrder

Creator string (sdk.AccAddress) Account which controls the limit order and to which proceeds are credited TrancheKey string TrancheKey for the target limit order

Withdrawal

Overview

Withdraw is used to redeem PoolShares for the user’s pro-rata portion of tokens within a liquidity pool. Users can withdraw from a pool at any time. When Withdrawing from a pool they will receive Token0 and Token1 in the same ratio as what is currently present in the pool. When withdrawing the users PoolShares are burned and their account is credited with the withdrawn tokens.

Withdraw Message

message

MsgWithdrawal

{ string creator =

1 ; string receiver =

2 ; string token_a =

3 ; string token_b =

4 ; repeated

string shares_to_remove =

5

[(gogoproto . moretags)

=

"yaml:\"shares_to_remove\"" , (gogoproto . customtype)

=

"github.com/cosmos/cosmos-sdk/types.Int" , (gogoproto . nullable)

=

false , (gogoproto . jsontag)

=

"shares_to_remove"] ; repeated

int64 tick_indexes_a_to_b =

6 ; repeated

uint64 fees =

7 ; } Field Description Creator string (sdk.AccAddress) The account from which the PoolShares are removed Receiver string (sdk.AccAddress) The account to which the tokens are credited TokenA string Denom for one side of the deposit TokenB string Denom for the opposing side of the deposit SharesToRemove []sdk.Int Amount of shares to remove from each pool TickIndexesAToB []int64 Tick indexes of the target LiquidityPools defined in terms of TokenA to TokenB (ie. TokenA is on the left) Fees []uint64 Fee for the target LiquidityPools

UpdateParams

Overview

Used to update module's params

UpdateParamsMsg

message

MsgUpdateParams

{ option

(amino . name)

=

"dex/MsgUpdateParams" ; option

(cosmos . msg . v1 . signer)

=

"authority" ;

// Authority is the address of the governance account. string authority =

1

[(cosmos_proto . scalar)

=

"cosmos.AddressString"] ; // NOTE: All parameters must be supplied. Params params =

2

[(gogoproto . nullable)

=

false , (amino . dont_omitempty)

=

true] ; }

message

Params

{ option

(gogoproto . goproto_stringer)

=

false ; repeated

uint64 fee_tiers =

1 ; string max_true_taker_spread =

2

[(gogoproto . moretags)

=

"yaml:\"max_true_taker_spread\"" , (gogoproto . customtype)

=

"github.com/neutron-org/neutron/v2/utils/math.PrecDec" , (gogoproto . nullable)

=

false , (gogoproto . jsontag)

=

"max_true_taker_spread"] ; } Field Description FeeTiers uint64 Fee tiers is the list of allowable fees that can be used for LP deposits MaxTrueTakerSpread string max true taker spread is the maximum difference in price between the stated tick price and the actual price (after rounding) given to the taker. Swaps in pools that result in a price difference greater thanmax_true_taker_spread will be aborted.

Gas Estimates

Below are basic gas estimates for various Dex operations. Depending on the exact state of the dex and the inputs of the message being sent real gas costs can vary substantially. For operations that touch multiple ticks the gas cost can be estimated using the formula:

g a s U s e d = f i x e d G a s C o s t + p e r T i c k G a s ∗ n T i c k s gasUsed = fixedGasCost + perTickGas*nTicks g a s U se d

= f i x e d G a s C os t

+ p er T i c k G a s

∗ n T i c k s

These estimates only consider the gas costs at the message server level and below; application level (ie. AnteHandler) gas costs are not included.

Deposit

CASE FIXED_GAS PER_TICK_GAS New pools 21825 48076 Adding to existing pools 19069 39100 *assumes single-sided deposits

Withdrawal

FIXED_GAS PER_TICK_GAS 21825 32215

PlaceLimitOrder

CASE FIXED_GAS PER_TICK_GAS Taker only limit order 44213 7779 Maker only GTC 31095 N/A Maker only JIT 48095 N/A Maker only GoodTil 49107 N/A

WithdrawLimitOrder

FIXED_GAS PER_TICK_GAS 24992 N/A

CancelLimitOrder

FIXED_GAS PER_TICK_GAS 25451 N/A Previous Fees Next Client