Liquidity Module V1 : Mechanism Explained

1. Context

a. Utility of the Cosmos Hub

- The utility of the Cosmos Hub is to provide core functionalities related to the soon-available IBC(Inter-Blockchain Communication) protocol
- The most important IBC-related functionality is the trading functionality among transferred coins into the Cosmos Hub
- The trading functionality has to safely store and manage a significant amount of external assets to facilitate the utility. Therefore, it is necessary for the Cosmos Hub, who has the largest dPoS security, to take the role to provide this important utility for a broad range of potential users

b. AMM(Automated Market Makers)

- Many DeXs were introduced to the cryptocurrency ecosystem for the past several years, but most of them failed to attract a significant user-base because it was very difficult to provide sustainable high-quality liquidity to the exchanges.
- Difficulties are caused by
    1) Expensive cost to hire professional market makers, because they require highly-trained financial engineers and significant capital investment
    2) Short-term profit oriented and high-return low-risk nature of market makers, which result in unstable and low quality liquidity
    3) Unnecessary large number of onchain transactions for limit orders and cancel orders
- Now, AMM model is introduced which solves most of the problems of traditional orderbook-based DeXs
    1) It lowers the barrier of market maker participation to anyone who does not have financial engineering skills nor significant amount of capital, which results in more efficient competition among market makers, hence cheaper liquidity cost for the exchanges
    2) The mechanism of AMM forces any market makers of AMM to provide liquidity in a pre-defined algorithmic way, so that the provided liquidity becomes having higher quality and better stability
    3) AMM drastically reduces the necessary number of transactions for its functionality, which is more adequate for decentralized environment

c. Interchain AMM as a Vision of the Cosmos Hub

- Most Ethereum-based AMMs are only dealing with ETH and ERC20 coins
- From Binance Exchange, in the top 20 coins, the volume of all ETH and ERC20 coins are occupying only 25%. It means that the Cosmos Hub AMM has potential user base of 3 times more than the potential user base of Ethereum-based AMMs, by connecting different kinds of blockchains via interchain communication technologies
- Therefore, the Cosmos Hub naturally has its vision to become most inclusive multi-chain AMM utility provider

d. **B-Harvest is building Liquidity Module for the Cosmos Hub**
   - B-Harvest is one of the strongest validators and contributors of the Cosmos ecosystem
   - B-Harvest has very skilled Cosmos-SDK engineers who have various experience to research and develop utilities with Cosmos-SDK
   - B-Harvest has strong knowledge/experience background on traditional finance, including stock/derivatives exchanges, market making, algorithmic trading, arbitrage trading, low-latency trading, and asset risk management
   - Above characteristics of the B-Harvest team makes the team the best talents for building AMM for the Cosmos Hub. So the Tendermint team decided to fund B-Harvest to design and implement the production level Liquidity Module, which will serve AMM functionality on the Cosmos Hub.

2. General Philosophy of the Liquidity Module

a. **Stick to the Basic**
   - The basic design of Liquidity module will follow the successful industry design standard of AMM, especially liquidity pool and constant product equation methodology from many current successful AMMs
   - Too unique design of AMM can cause unnecessary design risk which results in failure of mass adoption

b. **Feature Extensibility and Continuous Evolution**
   - We think that every new technology is a dynamically evolving creature
   - Rather than having too specified, narrowed design from the beginning, the Liquidity module has most common and standard design with capability to expand its features in future
   - The module will be continuously evolving to mitigate market demands, to have better incentive system and less technical/financial risks, and to attract more external users from multiple blockchains outside Cosmos ecosystem

c. **Inclusiveness**
   - The Cosmos Hub AMM should have strong philosophy of inclusiveness of users from different blockchains because its prime utility is inter-blockchain communication
   - To possess such characteristics, the Liquidity module should provide most convenient ways for external users to come in and use the services provided by the Cosmos Hub
   - The Liquidity module should not anticipate specific assets, such as Atom, into the process of user-flow in a forced manner. It is repeatedly proved that unnatural anticipation of native coin at unavoidable parts of process resulting in poor user attraction

d. Combination of Traditional Orderbook-based Model and New AMM Model

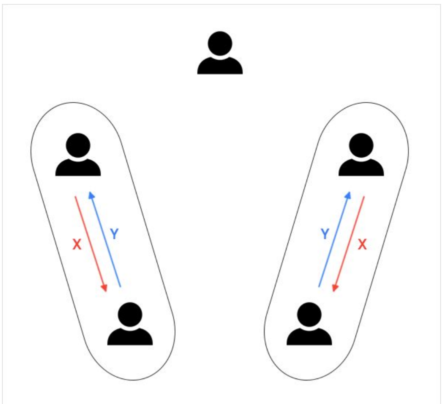


Fig. 1 Order book model

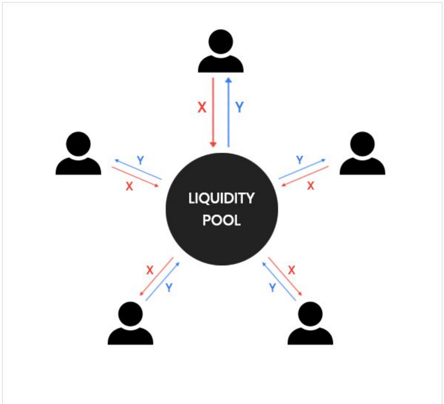


Fig. 2 Liquidity pool model

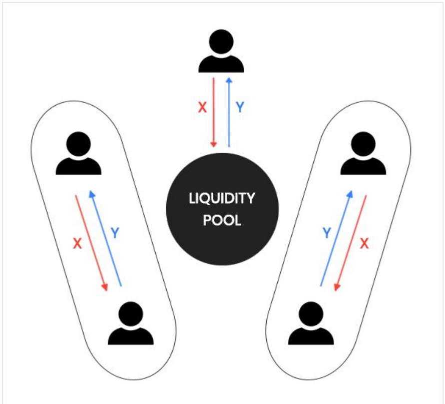


Fig. 3 Hybrid model

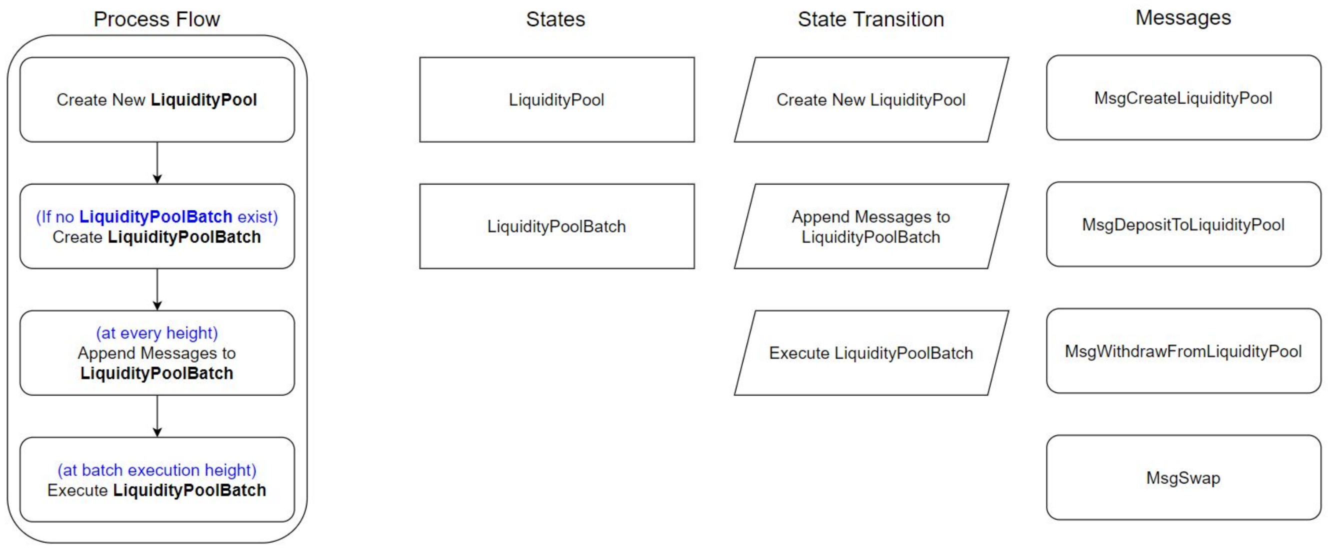
- Although new AMM model has multiple advantages over orderbook-based model, combination of both models will create more enriched utilities for wider potential users
- We re-define the concept of a “swap order” in AMM as a “limit order with short lifetime” in an orderbook-based exchange. Then, two concepts from two different models can be combined as one united model so that the function can provide both ways to participate into the trading and liquidity providing activities
- Although our first version of the Liquidity module will not provide limit order option, but the base structure of the codebase is already anticipating such feature expansion in near future
- Advantages of combined model
    - More freedom on how to provide liquidity : Limit orders
    - Combination of pool liquidity and limit order liquidity provides users more enriched trading environment

e. Fair Trading Environment : Minimizing Low-Latency Competition

- In the traditional financial system, ordinary traders have significant disadvantages against institutional traders because of their inability to trade in a sub-second time frame. Lots of financial institutions around the world are making billions of profit from taking advantage of their experienced technology, better network equipment, and their business position advantage regarding information interception
- This low-latency competition is revisited in most existing AMMs
- We think that the competition of speed is a completely unnecessary characteristic of a DeX, so we adopted the “batch execution” model. Swap executions are processed every end of a batch, which can be of multiple number of blocks, and all messages inside the batch are handled fairly without any prioritized sorting by time dimension
- Prevention of problems
    - Front-running and Manipulated message ordering:
        1) Messages in a batch are handled without any time priority, therefore it greatly reduces the unfairness of execution caused by manipulated sorting of messages
        2) To manipulate batches with multiple blocks, the validator should form a cartel so that all proposers of a batch are members of the cartel. Therefore, batches with more blocks result in more difficult cartelization, hence more difficult front-running
    - Fake orders : Fake orders should have high confidence that they can be cancelled before execution. But, in our model, the batch process is slow, and it is impossible to cancel orders before finalization of the batch process. Therefore there will exist much less fake orders

3. Liquidity Module Mechanism

a. Process Flow, States, State Transitions, and Messages


Process Flow
Create New LiquidityPool
(If no LiquidityPoolBatch exist) Create LiquidityPoolBatch
(at every height) Append Messages to LiquidityPoolBatch
(at batch execution height) Execute LiquidityPoolBatch
States
LiquidityPool
LiquidityPoolBatch
State Transition
Create New LiquidityPool
Append Messages to LiquidityPoolBatch
Execute LiquidityPoolBatch
Messages
MsgCreateLiquidityPool
MsgDepositToLiquidityPool
MsgWithdrawFromLiquidityPool
MsgSwap

b. Liquidity Pool Creation

- Permissionless : Anyone can create a liquidity pool
- Liquidity Pool Type : There can exist multiple types of liquidity pools so that the Liquidity module can provide various liquidity pool models in the future. The Liquidity module version 1 will only have the constant product type with two reserve coins
- Spam prevention : There exists a fee to create a liquidity pool to prevent spams. The fee goes to the Cosmos Hub Community Fund

c. Liquidity Pool Deposit/Withdraw

- Deposit/Withdraw with Pool's Reserve Ratio : Deposits and withdrawals are accepted with pool's reserve ratio, just like Uniswap model
- Pool Token : Depositor receives pool tokens, which represent the right to withdraw a proportional amount of pool's reserve coins. Pool tokens are transferable

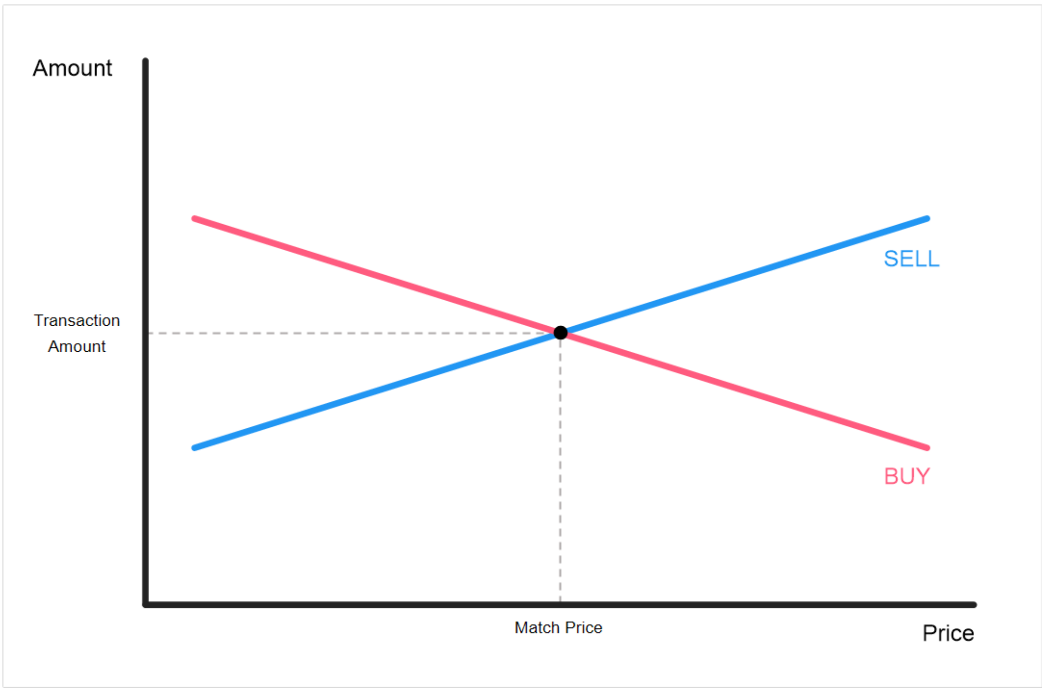
d. Swap Order

- Order Price : The swap order submitter sends the order price to the Liquidity module, which implies the maximum swap price where the submitter can accept for
- If the batch execution result cannot include the swap request with allowed range of swap price by the submitter, the swap order is automatically cancelled
- Compatibility with Orderbook-based Limit Order
  - The swap order interface is completely identical to the limit order of order book-based model. This design will allow us to simply adopt orderbook-based limit orders without any design reconstruction (If the unexecuted swap orders are roll-overed to next batch, this is exactly what limit order is!)

e. Batch Concept

- Swap orders are just limit orders in orderbook-based exchange!
    - Swap from X to Y = Limit order buying Y from X
    - Swap from Y to X = Limit order selling Y for X
    - Order Price : Price of Y over X = Y/X price

- Orderbook-based exchange matching with batch transaction
  (batch transaction : store orders for given period, and match stored orders at once)



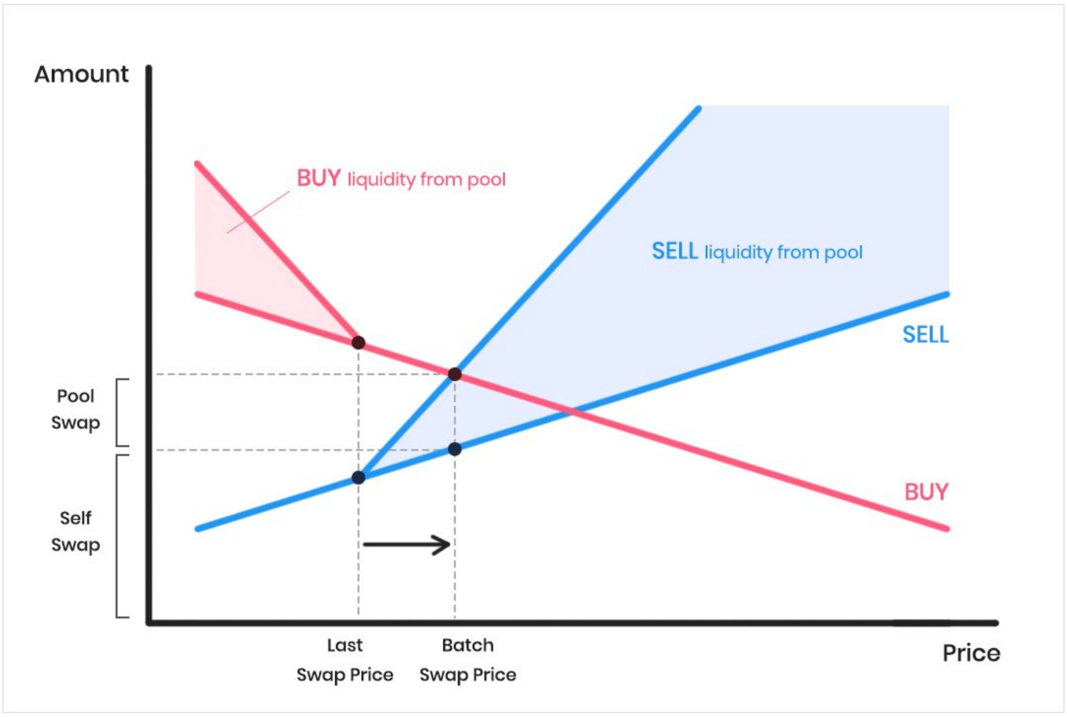
- On **BUY graph**, the y value (Amount) represents all buy order amount which can be executed at given x value (Price)
  → order price equal or higher than the x value (Price)

- On **SELL graph**, the y value (Amount) represents all sell order amount which can be executed at given x value (Price)
  → order price equal or lower than the x value (Price)

- **Match price** : the matching price of this batch for given buy and sell orders

- **Transaction amount** : the amount of buy/sell orders matched in this batch

- The **liquidity pool** participates on the batch as passive buyer/seller



- How the order amount of liquidity pool is determined?
    - Constant Product Equation
    - Available buy/sell orders from liquidity pool are displayed as colored area

- How to find Swap Price ?
    - Variables
        - RX : X amount in the liquidity pool (before this batch)
        - RY : Y amount in the liquidity pool (before this batch)
        - P : The swap price for this batch
        - BX : X amount of buy orders with order price equal or higher than P
        - SY : Y amount of sell orders with order price equal or lower than P
        - PoolX : X amount of liquidity pool orders executable at P, according to the Constant Product Equation
    - Objective : To find the swap price which satisfies below conditions
        - condition 1: All swap orders with executable order prices are matched
          → BX = PoolX + SX = PoolX + SY * P
        - condition 2 : Constant product equation for the liquidity pool executions
          → RX * RY = (RX + PoolX) * (RY - PoolX / P)
    - Uniqueness of the Solution
        - From condition 1
            - PoolX = BX(non-increase) - SY(non-decrease) * P(increase)
            - PoolX is a continuously decreasing function of P
        - From condition 2
            - PoolX = P(increase) * RY(fixed) - RX(fixed)
            - PoolX is a continuously increasing function of P
        - There exists unique P where both condition 1&2 holds because PoolX is decreasing in condition 1 and increasing in condition 2 over P

f. Swap Price Calculation

1) Finding price direction
    - Variables
        - X : Reserve of X coin, Y : Reserve of Y coin (before this batch execution)
        - PoolSwapPrice = X/Y
        - XOverLastPrice : amount of orders which swap X for Y with order price higher than last PoolSwapPrice
        - XAtLastPrice : amount of orders which swap X for Y with order price equal to last PoolSwapPrice
        - YUnderLastPrice : amount of orders which swap Y for X with order price lower than last PoolSwapPrice
        - YAtLastPrice : amount of orders which swap Y for X with order price equal to last PoolSwapPrice
    - Increase : swap price is increased from last PoolSwapPrice
        - XOverLastPrice > (YUnderLastPrice+YAtLastPrice)*PoolSwapPrice
    - Decrease : swap price is decreased from last PoolSwapPrice
        - YUnderLastPrice > (XOverLastPrice+XAtLastPrice)/PoolSwapPrice
    - Stay : swap price is not changed from last PoolSwapPrice
        - when both above inequalities do not hold

2) Increase case
    - Iteration
        - orderPrice(i) : Iterate order prices of all swap orders from low to high
        - EX(i) : All executable orders which swap X for Y with order price equal or higher than this orderPrice(i)
        - EY(i) : All executable orders which swap Y for X with order price equal or lower than this orderPrice(i)
        - PoolY(Y coins provided by the liquidity pool) = Y - X/orderPrice(i)

    - Find the orderPrice(k) where below value has the first negative number
        - EX(k) - EY(k)*orderPrice(k) - PoolY*orderPrice(k)

    - ExactMatch : swapPrice is located between orderPrice(k-1) and orderPrice(k) ?
        - swapPrice = (X + EX(k))/(Y + EY(k-1))
            - orderPrice(k-1) < swapPrice < orderPrice(k)
        - adjPoolY = (Y*EX(k) - X*EY(k-1))/(X + EX(k))
            - adjPoolY >= 0
        - If both conditions are met, swapPrice is the swap price for this batch
            - Amount of X coins matched from swap orders = EX(k)
            - Amount of Y coins matched from swap orders = EY(k-1)
            - Amount of Y coins provided from liquidity pool = adjPoolY
            - Three parts are perfectly matched without fractional match
        - If one of above conditions doesn’t hold, go to next step : FractionalMatch

    - FractionalMatch : swapPrice = orderPrice(k-1)
        - Amount of X coins matched from swap orders :
            - FracEX = min(EX(k-1), EY(k-1)*swapPrice+PoolY*swapPrice)
        - Amount of Y coins matched from swap orders = EY(k-1)
        - Amount of Y coins provided from liquidity pool = PoolY
        - Swap orders which swap X for Y are fractionally matched
            - FractionalRatio = FracEX / EX(k-1)

3) Decrease case

- Iteration
    - orderPrice(i) : Iterate order prices of all swap orders from high to low
    - EX(i) : All executable orders which swap X for Y with order price equal or higher than this orderPrice(i)
    - EY(i) : All executable orders which swap Y for X with order price equal or lower than this orderPrice(i)
    - PoolX(X coins provided by the liquidity pool) = X - Y*orderPrice(i)

- Find the orderPrice(k) where below value has the first negative number
    - EY(k) - EX(k)/orderPrice(k) - PoolX/orderPrice(k)

- ExactMatch : swapPrice is located between orderPrice(k-1) and orderPrice(k) ?
    - swapPrice = (X + EX(k-1))/(Y + EY(k))
        - orderPrice(k) < swapPrice < orderPrice(k-1)
    - adjPoolX = (X*EY(k) - Y*EX(k-1))/(Y + EY(k))
        - adjPoolX >= 0
    - If both conditions are met, swapPrice is the swap price for this batch
        - Amount of X coins matched from swap orders = EX(k-1)
        - Amount of Y coins matched from swap orders = EY(k)
        - Amount of X coins provided from liquidity pool = adjPoolX
        - Three parts are perfectly matched without fractional match
    - If one of above conditions doesn't hold, go to next step : FractionalMatch

- FractionalMatch : swapPrice = orderPrice(k-1)
    - Amount of Y coins matched from swap orders :
        - FracEY = min(EY(k-1), EX(k-1)/swapPrice+PoolX/swapPrice)
    - Amount of X coins matched from swap orders = EX(k-1)
    - Amount of X coins provided from liquidity pool = PoolX
    - Swap orders which swap Y for X are fractionally matched
        - FractionalRatio = FracEY / EY(k-1)

4) Stay case

- Variables
    - swapPrice = last PoolSwapPrice
    - EX : All executable orders which swap X for Y with order price equal or higher than last PoolSwapPrice
    - EY : All executable orders which swap Y for X with order price equal or lower than last PoolSwapPrice

- ExactMatch : If EX == EY*swapPrice
        - Amount of X coins matched from swap orders = EX
        - Amount of Y coins matched from swap orders = EY
        - All two parts are perfectly matched without fractional match

- FractionalMatch
    - If EX > EY*swapPrice : Residual X order amount remains
        - Amount of X coins matched from swap orders = EY*swapPrice
        - Amount of Y coins matched from swap orders = EY
    - If EY > EX/swapPrice : Residual Y order amount remains
        - Amount of X coins matched from swap orders = EX
        - Amount of Y coins matched from swap orders = EX/swapPrice

g. Swap Fee Payment

- Swap fees are calculated after above calculation process
- Swap fees are proportional to the coins received from matched swap orders
    - SwapFee = ReceivedMatchedCoin * SwapFeeRate
- Swap fees are sent to the liquidity pool