TOREX Whitepaper

> Main author: Miao, ZhiCheng <miao@superfluid.finance>

> Co-author(s):
>
> - Francesco Renzi <fran@superfluid.finance>, co-designer the original system.
> - Kaspar Kallas <kaspar@superfluid.finance>, author of the basic liquidity mover.
> - Dietmar Hofer <didi@superfluid.finance>, Daniel Tok<daniel@superfluid.finance>, general involvements in the TOREX design process.

Revision History:

Revised On	Version	Revision Description
@March 4, 2024	1.0	First version for stealth launch.

▼ **Table of Contents**

Abstract

👉 **TOREX** stands for **T**(WAP) **OR**(acle) **EX**(change).

Building open protocols on Blockchain is a compositional process: combining the capability of one protocol with another often solves new problems. One such problem is swapping one asset for another time-continuously at the fairest prices. This whitepaper introduces TOREX which solves this problem by combining the Uniswap V3 protocol as price oracle and the Superfluid Protocol for composable money flows.

Introduction

TOREX is a unique exchange for traders to swap one asset for another using smart contracts, where:

1. The swaps are performed time-continuously. Exchanges that allow such swaps are also known as "reactive exchanges." The appendix "Other Reactive Exchanges" compares it with other reactive exchanges.
2. Using a time-weighted price oracle (TWAP) to regulate the aggregation of liquidity sources gives traders *fair prices over time*.
3. The origins of liquidity sources for the exchanges are agnostic to the exchange. The appendix "Basic Liquidity Mover" provides a reference implementation that uses the liquidity source of a Uniswap V3 pool.

To implement TOREX, two key on-chain innovations are used:

1. The Uniswap implementation of TWAP. The chapter "About Uniswap V3 TWAP" will briefly cover this topic.
2. Superfluid Protocol's implementation of *Semantic Money* allows building on-chain composable money flows. The chapter "About Semantic Money" briefly covers this topic.

The core logic of TOREX allows a competitive and non-stalling marketplace of liquidity providers called *liquidity movers*, detailed in the chapter "Core Logic of TOREX."

Additionally, the core logic of TOREX provides a set of hooks to allow derivative works to extend core functionalities (such as fees, staking, and yield farming). The chapter "TOREX hooks" gives an overview of this topic. Later, the paper also provides an example of using them to implement the fee and the staking mechanism.

Furthermore, because swaps happen time-continuously, traders need not be concerned with the issue of maximal extractable value (MEV) [mev]. However, liquidity movers may still need to worry about MEV to compete for liquidity. The chapter "MEV and TOREX" briefly covers this topic.

About Uniswap V3 TWAP Oracle

To provide traders with fair prices on-chain, TOREX needs an oracle. As its name suggests, TOREX uses a TWAP (time-weighted average price) oracle. Specifically, TOREX uses the TWAP oracles provided by Uniswap V3 pools.

From Uniswap v2 to Uniswap v3

In Uniswap v2, a pool "accumulates this price, by keeping track of the cumulative sum of prices at the beginning of each block in which someone interacts with the contract. Each price is weighted by the time since the last block was updated, according to the block timestamp" [uniswap-v2-whitepaper].

We define P_{t_1,t_2} as the TWAP between time t_1 and t_2. With Uniswap V2 TWAP oracle, we have:

$$P_{t_1,t_2} = \frac{\sum_{i=t_1}^{t_2} p_i}{t_2 - t_1} = \frac{\sum_{i=1}^{t_2} p_i - \sum_{i=1}^{t_1} p_i}{t_2 - t_1} = \frac{a_{t_2} - a_{t_1}}{t_2 - t_1}$$

In Uniswap v3, there are a few changes to the design of the TWAP oracle [uniswap-v3-whitepaper]. For TOREX, the most relevant changes are:

1. "Uniswap v3 brings the accumulator checkpoints into the core, allowing external contracts to compute on-chain TWAPs over recent periods without storing checkpoints of the accumulator value." This change impacts how TOREX interacts with the oracle directly.
2. "... instead of accumulating the sum of prices, allowing users to compute the arithmetic mean TWAP, Uniswap v3 tracks the sum of log prices, allowing users to compute the geometric mean TWAP." This change is necessary to make the change happen in (1), elaborated in the whitepaper.

With Uniswap V3 TWAP oracle, we have instead:

$$log_{1.0001}(P_{t_1,t_2}) = \frac{\sum_{i=t_1}^{t_2} log_{1.0001}(P_i)}{t_2 - t_1} = \frac{a_{t_2} - a_{t_1}}{t_2 - t_1}$$

, or:

$$P_{t_1,t_2} = 1.0001^{\frac{a_{t_2} - a_{t_1}}{t_2 - t_1}}$$

Storage of Price Accumulators

A pool in Uniswap v2 stores only the most recent value of this price accumulator. The external caller is responsible for providing the previous value.

While it is true that in Uniswap v3, the pool can store a list of previous values for the price accumulator, however as its paper suggests, “this array initially only has room for a single checkpoint, anyone can initialize additional storage slots to lengthen the array, extending to as many as 65,536 checkpoints.”

Uniswap v3 is concerned with the public good nature of an “on-chain price oracle for all” paid by market participants, an obligation enforced by the smart contract. However, the users of this public good should be wary of its availability, as we have shown that there is indeed an initial condition (single storage slot to begin with) and a limitation (65,536 maximum checkpoints) built in.

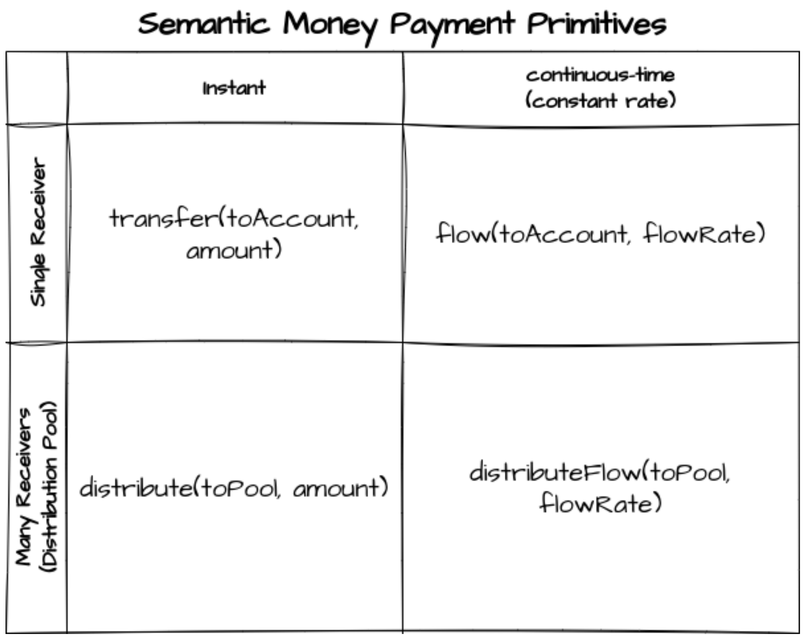
Considering these factors and due to TOREX's core logic requirement, each TOREX maintains its observer of on-chain price accumulators. We shall examine this design shortly.

About Semantic Money

TOREX provides continuous-time swaps. *Semantic money* makes this type of continuous-time "gadgets" composable, meaning with a low amount of capital locked, value moves between these gadgets time-continuously without on-chain transactions.

Beyond copying what we knew about how payment onto the Blockchain world, semantic money expands the modality for payment by having these new "payment primitives" implemented by the Superfluid Protocol:

1. **One-to-one instant value transfer** is how payment works as we all know it.
2. **One-to-one continuous-time value transfer** is a new payment modality, especially as a constant flow of value. Projects such as Superfluid call this "money streaming," which should not be confused with "streaming data" or "streaming music," where semantic money models the payment as a function of time, not discrete data packages in streaming [superfluid-money-streaming].
3. **One-to-many instant value transfer** may be emulated as many one-to-one instant transfers of value. However, semantic money supports an additional modality for efficient on-chain distribution to predefined groups.
4. **One-to-many continuous-time value transfer**, as you may expect, is also supported by semantic money [superfluid-money-distribution].


Semantic Money Payment Primitives

	Instant	continuous-time (constant rate)
Single Receiver	transfer(toAccount, amount)	flow(toAccount, flowRate)
Many Receivers (Distribution Pool)	distribute(toPool, amount)	distributeFlow(toPool, flowRate)

Semantic Money Payment Primitives

Additionally, semantic money is programmable and allows smart contracts to compose all the "payment primitives" with low capital locking requirements, making those mentioned above composable "gadgets" possible [docs-to-super-apps].

Superfluid is an early implementation of semantic money, and its docs explain these concepts in depth [superfluid-docs]. For more curious readers, the semantic money draft yellowpaper offers a more theoretical angle on the topic.

Core Logic of TOREX

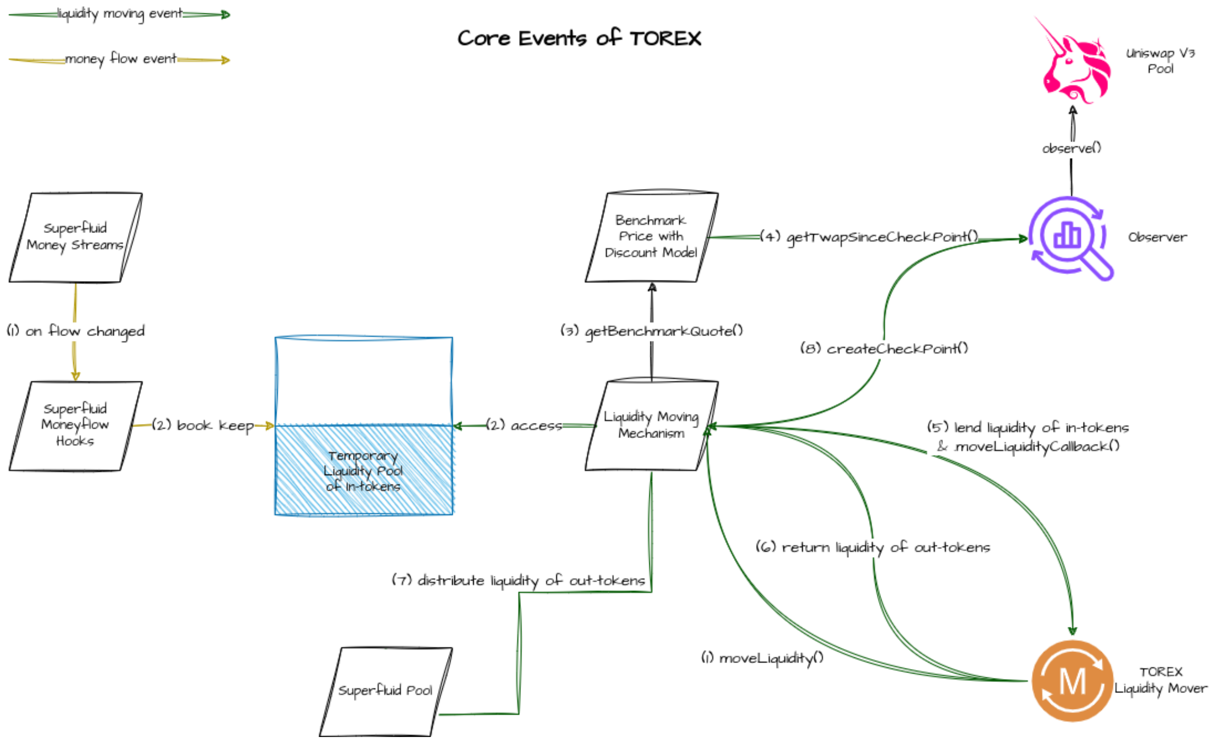
A TOREX swaps liquidity of *in-tokens* for liquidity of *out-tokens* for traders.

To do so, each TOREX book keeps money streams from traders at *money flow events*, maintains a temporary liquidity pool of in-tokens, and offers the liquidity to *liquidity movers* at a *benchmark quote* for a number of out-tokens.

At a *liquidity movement event (LME)*, the TOREX receives the quoted amount of out-tokens and distributes them to the traders through Superfluid money distribution.

Each TOREX is configured with a dedicated TWAP observer, at which TOREX can create checkpoints and query TWAP since the last checkpoint. TOREX then discounts this TWAP through a *discount model* that ensures that TOREX should never stall by offering the liquidity movers ever-greater incentives over time.

One could view TOREX as being driven by these two independent event loops:


liquidity moving event
money flow event
Core Events of TOREX
Uniswap V3 Pool
observe()
Superfluid Money Streams
Benchmark Price with Discount Model
(4) getTwapSinceCheckPoint()
Observer
(1) on flow changed
(3) getBenchmarkQuote()
(8) createCheckPoint()
Superfluid MoneyFlow Hooks
(2) book keep
Temporary Liquidity Pool of in-tokens
(2) access
Liquidity Moving Mechanism
(5) lend liquidity of in-tokens & moveLiquidityCallback()
(6) return liquidity of out-tokens
(7) distribute liquidity of out-tokens
(1) moveLiquidity()
Superfluid Pool
TOREX Liquidity Mover

TOREX Core Events

TWAP Observer

A TWAP observer is the abstraction for TOREX to create checkpoints and obtain TWAP since the last checkpoint TOREX created:

```
interface ITwapObserver {
    function createCheckpoint(uint256 time)
        external
        returns (bool);
    function getDurationSinceLastCheckpoint(uint256 time)
        external view
        returns (uint256 duration);
    function getTwapSinceLastCheckpoint(uint256 time, uint256 inAmount)
        external view
        returns (uint256 outAmount, uint256 duration);
}
```

UniswapV3PoolTwapObserver

It is straightforward to implement it using a single Uniswap V3 pool as the TWAP oracle:

```
/// The Uniswap V3 pool to be used as price benchmark for liquidity moving.
IUniswapV3Pool public immutable uniPool;
/// Uniswap pool is bi-direction but torex is not.
/// If false, inToken maps to token0, and vice versa.
bool           public immutable inverseOrder;

uint256 internal _lastCheckPointAt;
int56   internal _lastTickCumulative;

function createCheckPoint(uint256 time)
    public override onlyOwner
    returns (bool)
{
  _lastCheckPointAt = time;
  _lastTickCumulative = _getCurrentTickCumulative();
  return true;
}

function getTwapSinceCheckPoint(uint256 time, uint256 inAmount)
    public override view
    returns (uint256 outAmount, uint256 duration)
{
  duration = getDurationSinceCheckPoint(time);

  int24 tick;
  if (duration > 0) {
    int56 currentTickCumulative = _getCurrentTickCumulative();
    tick = SafeCast.toInt24((int256(currentTickCumulative) -
                             int256(_lastTickCumulative))
                            / SafeCast.toInt256(duration));
  } else {
    // special case: when duration is zero, returning the current tick directly
    (,tick,,,,,) = uniPool.slot0();
  }

  if (inverseOrder == false) {
    outAmount = OracleLibrary.getQuoteAtTick(tick, SafeCast.toUint128(inAmount),
                                             uniPool.token0(), uniPool.token1());
  } else {
    outAmount = OracleLibrary.getQuoteAtTick(tick, SafeCast.toUint128(inAmount),
                                             uniPool.token1(), uniPool.token0());
  }
}

function _getCurrentTickCumulative() internal view
  returns (int56 currentTickCumulative)
{
  int56[] memory tickCumulatives;
  (tickCumulatives,) = uniPool.observe(new uint32[](1));
  return tickCumulatives[0];
}
```

Other Types of Observers

Notably, this abstraction opens up the possibility for implementations where more than one Uniswap V3 pool was used to calculate the TWAP or even TWAP implementations not using Uniswap.

Liquidity Moving Event

Liquidity Movers

Generally, TOREX facilitates an open marketplace where liquidity movers can compete for liquidity. However, as later chapters uncover shortly, access control to the liquidity roles can be provided by the TOREX hooks and may be desirable to provide consistent liquidity flows to traders.

Concretely, a liquidity mover is an on-chain contract that implements this interface:

```
/**
 * @title Interface for the liquidity mover contract
 */
interface ILiquidityMover {
    /**
     * @notice A callback from torex requesting out token liquidity with in-token
               liquidity transferred.
     * @param inToken      The in token transferred to the contract.
     * @param outToken     The out token that is requested.
     * @param inAmount     The amount of in token that has been transferred to the
                           contract.
     * @param minOutAmount The amount of out token that is requested.
     * @param moverData    The data that liquidity mover passes through
                           `ITorex.moveLiquidity`.
     * @return It must always be true.
     */
    function moveLiquidityCallback(ISuperToken inToken, ISuperToken outToken,
                                   uint256 inAmount, uint256 minOutAmount,
                                   bytes calldata moverData)
        external returns (bool);
}
```

To trigger a liquidity movement event, the liquidity mover contract should call the `moveLiquidity` of TOREX:

```
function moveLiquidity() external returns (uint256 inAmount, uint256 outAmount);
```

The TOREX then:

1. Check the balance of the temporary liquidity pool of in-tokens, as `inAmount`.
2. Call `getBenchmarkQuote(inAmount)`, which `getTwapSinceCheckPoint(inAmount)` from the observer and applies a discount using the discount model.
3. Transfer the in-tokens liquidity to the liquidity mover.
4. Call `moveLiquidityCallback` of the liquidity mover, where liquidity movers should find `minOutAmount` of liquidity of out-tokens.
5. Upon returning from the liquidity mover, the TOREX validates that `minOutAmount` has been deposited to the TOREX. Otherwise, the TOREX aborts the transaction.
6. The TOREX then distributes the liquidity of out-tokens to all traders.
7. Lastly, the TOREX creates a new checkpoint at the observer.

Benchmark Quote

The goal of the benchmark quote is to provide fair prices to the traders. TWAP provides a fair price because it is an average price since the last liquidity moving event.

However, offering fair prices as a goal alone could stall TOREX's ability to continue to attract liquidity movers since the TWAP price could be higher than the spot price for an extended period. A simple thought experiment is when a spot price drops continuously, TWAP may be behind it for a long period. Such stalling would not be ideal for traders,

In the interest of traders, TOREX also adds non-stalling as its goal. A discount model achieves this goal by offering increasing discounts to the liquidity movers over time.

```
function getBenchmarkQuote(uint256 inAmount) public view
    returns (uint256 minOutAmount,
             uint256 durationSinceLastLME,
             uint256 twapSinceLastLME)
{
    (twapSinceLastLME, durationSinceLastLME) = _observer
        .getTwapSinceLastCheckpoint(block.timestamp, inAmount);
    twapSinceLastLME = scaleValue(_twapScaler, twapSinceLastLME);
    minOutAmount = getDiscountedValue
        (_discountFactor, twapSinceLastLME, durationSinceLastLME);
}
```

As demonstrated in the above code, The minimum amount of out-tokens is a discount from `twapSinceLastLME`, taking into account both `_discountFactor` and `durationSinceLastLME`.

Discount Model

The discount model used by TOREX is a *shifted reciprocal function*:

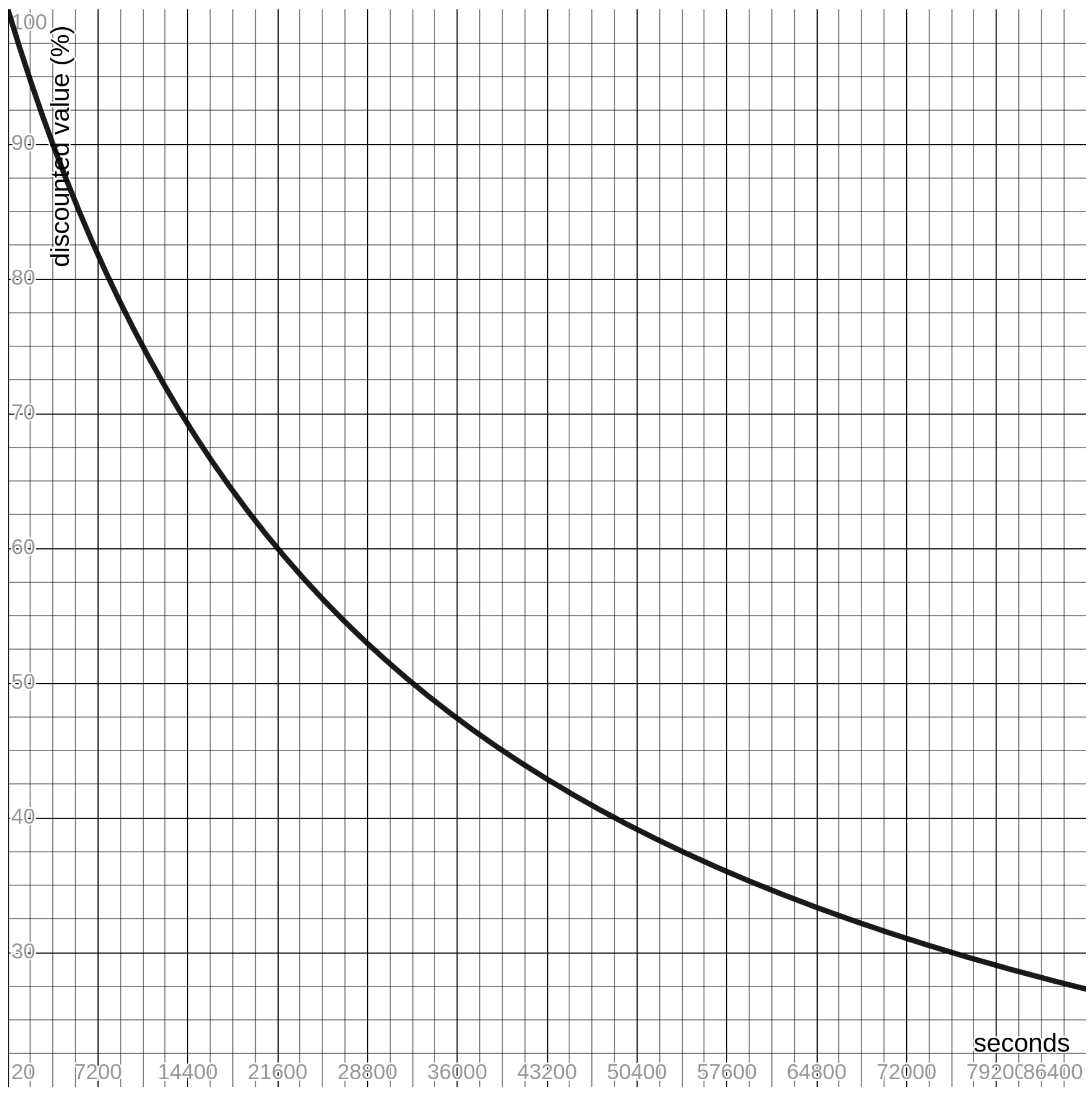
$$f(v,t) = v * \frac{F}{F+t}$$

, where F is the *discount factor*, v is the full value without discount , and t is the relative time since the discount should start applying. As you can notice, when t is 0, it applies zero discount, while at the infinite future, the entire value is discounted.

To choose a discount factor intuitively, it can be re-formulated using τ and ϵ instead:

$$F = \tau\frac{1-\epsilon}{\epsilon}$$

In this formulation, predictably, where at a certain time in the future, say, τ seconds, the discount is ε. As in the following concrete example, perhaps in more extreme settings than in real life, where $\tau = 3600, \epsilon = 10\%$, one should validate from the plot that at exactly 3600 seconds, only 90% of the original value is left.



An example discount factor (desmos link).

Money Flow Event

We haven’t mentioned where the balance of the temporary liquidity pool is coming from, but the readers may have realized that it should come from the traders in money streams.

Indeed, it is. More importantly, to book-keep these events precisely, TOREX uses the money flow hooks provided by Superfluid by being a Super App. These hooks are for when a trader creates, updates, or deletes its money stream of in-tokens to the TOREX.

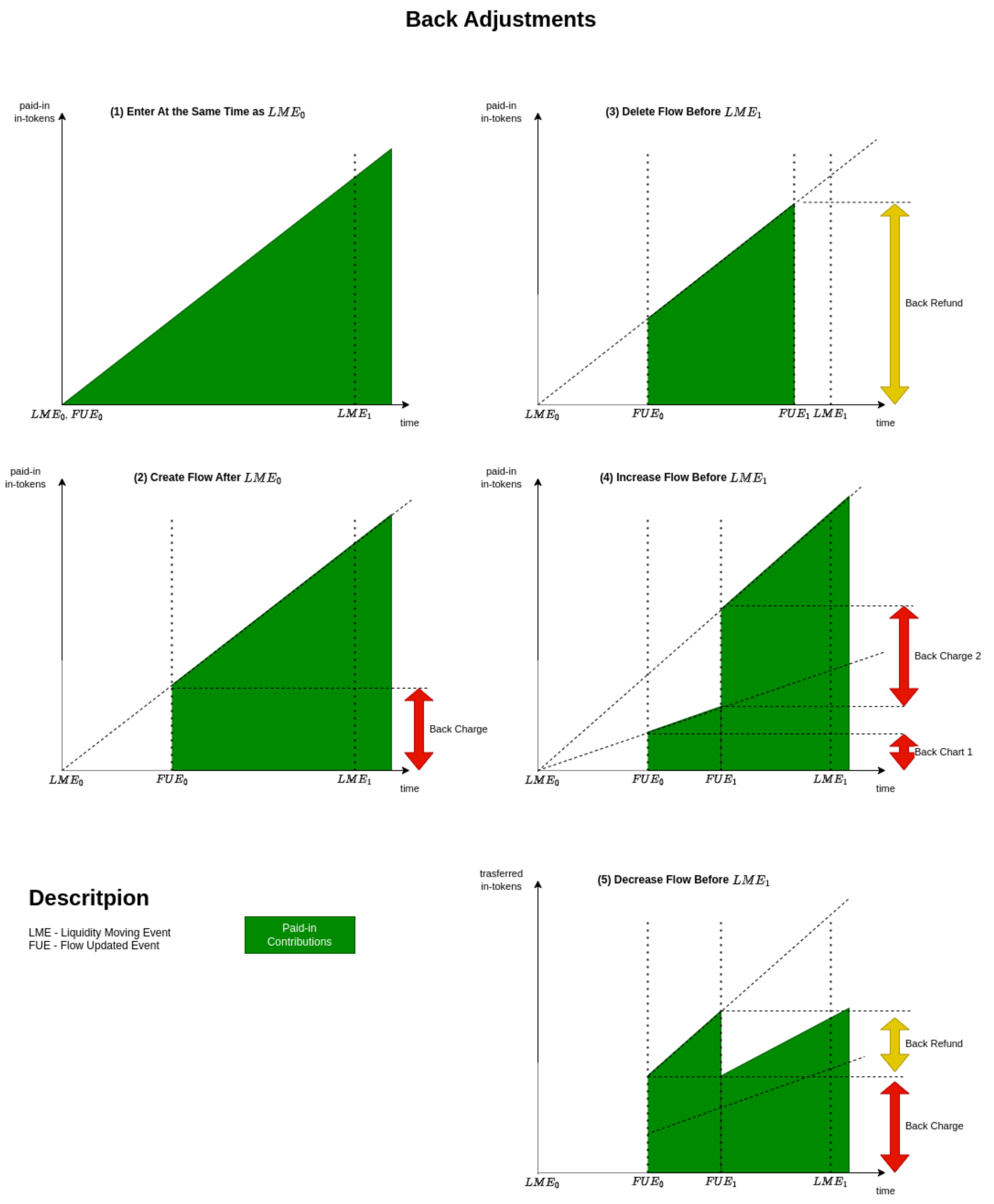
Back Adjustments

Because money flow events and liquidity movement events are independent of each other, new contributions of in-tokens from a trader should only get a portion of the swaps at the next liquidity movement event. However, liquidity movement events deal with all traders simultaneously; it is computationally unscalable to deal with these exceptions on a trader-by-trader basis.

A technique to address this conundrum is called “back charge,” where TOREX requires the trader to pay back the missing contribution amount of in-tokens since the previous liquidity movement event.

In another similar case, a “back refund” technique is provided to refund the trader's portion of in-token contributions since the last liquidity movement event, when the trader stops part or all its money stream to the TOREX.

Both back charges and back refunds are called “back adjustments.” Here is an illustration of several scenarios related to back adjustments:



TOREX Back Adjustment Cases (without fees)

TOREX Hooks

While TOREX's core logic is succinct, it provides additional hooks for extensibility to the design of its applications through the `ITorexController` interface.

TOREX Controller Interface

```
/**
 * @title Interface of torex controllers
 * @dev A Torex controller provides hooks to the Torex so that additional
        functionalities could be added.
 */
interface ITorexController {
    /// Get the type id of the controller.
    function getTypeId() external view returns (bytes32);

    /// A hook for in-token flow changed reported by torex.
    function onInFlowChanged(address flowUpdater,
                             int96 prevFlowRate, int96 preFeeFlowRate,
                             uint256 lastUpdated,
                             int96 newFlowRate, uint256 now,
                             bytes calldata userData) external
        returns (int96 newFeeFlowRate);

    /// A hook for liquidity moved reported by torex.
    function onLiquidityMoved(LiquidityMoveResult calldata result)
        external returns (bool);
}
```

A TOREX controller must implement two hooks `onInFlowChanged` and `onLiquidityMoved`.

Additionally, the TOREX controller is also the admin of the fee distribution pool of TOREX, where the controller can independently control fees in in-tokens to the stakers the controller manages.

Note that, in the production TOREX contract, there is an immutable max-fee setting to prevent the controller from extracting more than a certain percentage of fees.

Implement Permission Controls

With the hooks, the controller may:

- regulate the trading permission for compliance needs,
- or provide access control to the liquidity movement system through auctioning.

Implement Fees & Staking

Through the `onInflowChanged` hook, the TOREX controller may customize fee rules, such as front-end fees, to incentivize building front-ends as distribution channels.

MEV and TOREX

Typically, a swap is instant and discrete in a financial transaction. In blockchains where transactions are not deterministic, such swaps are often subject to a class of game-theoretical attacks known as maximal extractable value (MEV).

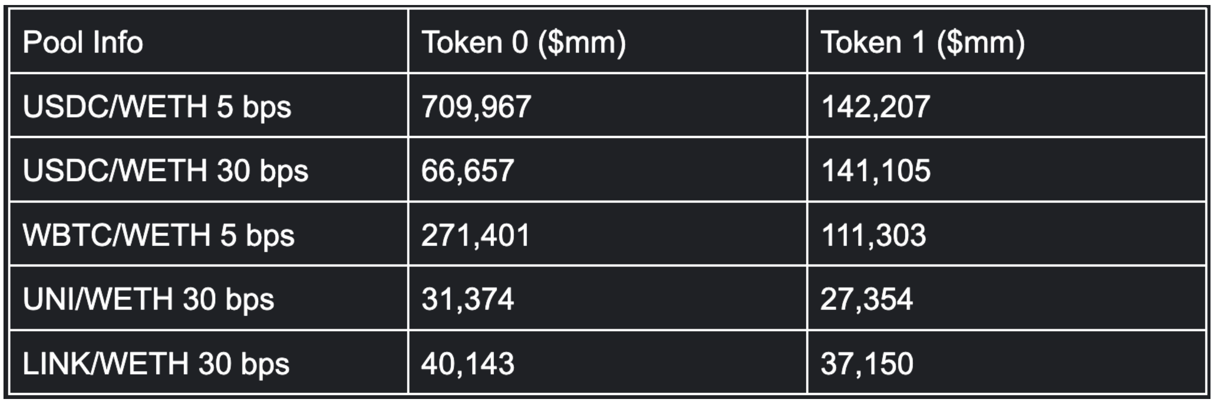
> Maximal extractable value (MEV) refers to the maximum value that can be extracted from block production in excess of the standard block reward and gas fees by including, excluding, and changing the order of transactions in a block. [mev]

Due to the continuous-time nature of TOREX, TOREX provides MEV protection for traders but requires special MEV-awareness for liquidity movers.

MEV Protection For Traders

For a TOREX trader, this attack is minimized because the contributions are paid over time, and the actual swaps of paid-in contributions occur, together with other traders' pooled contributions, in the future by the liquidity movers where prices are time-weighted averages.

Such time-weighted averages are impossible to manipulate in the same block, by definition. However, over time, one may put one's capital at risk to manipulate the average price in future blocks. The longer such delay, the higher one's capital is at risk, while the shorter such delay is, the higher the capital is required to move the price. Such beautiful balance is evident from Uniswap's analysis on "How much does a two blocks 20% manipulation require?"



Pool Info	Token 0 ($mm)	Token 1 ($mm)
USDC/WETH 5 bps	709,967	142,207
USDC/WETH 30 bps	66,657	141,105
WBTC/WETH 5 bps	271,401	111,303
UNI/WETH 30 bps	31,374	27,354
LINK/WETH 30 bps	40,143	37,150

An excerpt from https://blog.uniswap.org/uniswap-v3-oracles

MEV Protection for Liquidity Movers

On the other hand, the swap must happen instantly for liquidity movers at the liquidity moving event.

Liquidity mover's strategy may be at the risk of being snatched by generalized frontrunners.

> Rather than programming complex algorithms to detect profitable MEV opportunities, some searchers run generalized frontrunners. Generalized frontrunners are bots that watch the mempool to detect profitable transactions. The frontrunner will copy the potentially profitable transaction's code, replace addresses with the frontrunner's address, and run the transaction locally to double-check that the modified transaction results in a profit to the frontrunner's address. If the transaction is indeed profitable, the frontrunner will submit the modified transaction with the replaced address and a higher gas price, "frontrunning" the original transaction and getting the original searcher's MEV. [mev]

TOREX's core logic does not provide any MEV protection to liquidity movers. While simple "replacing addresses with frontrunners'" wouldn't necessarily work for TOREX liquidity moves, liquidity movers must be vigilant against the potential of MEV plugins that is TOREX aware.

However, in the future, thanks to the TOREX controller's permission hooks, it is possible to use access control to liquidity movers to limit MEV bots through a prior commitment from earnest liquidity movers.

Appendix A: Other Reactive Exchanges

Modeling a problem domain explicitly with time is also called functional reactive programming (FRP) [frp]. Hence, we call exchanges that do continuous-time swaps "reactive exchanges." [on-reactive-exhanges]

There are several different designs of reactive exchanges. Since each design deserves a paper, this appendix should only cover their essence.

Reactive Constant Function Market Maker

This is a "timely" modification over the Uniswap V1 innovation: Constant Function Market Maker.

This is the sage-math module that adds time to the equation of the CFMM:

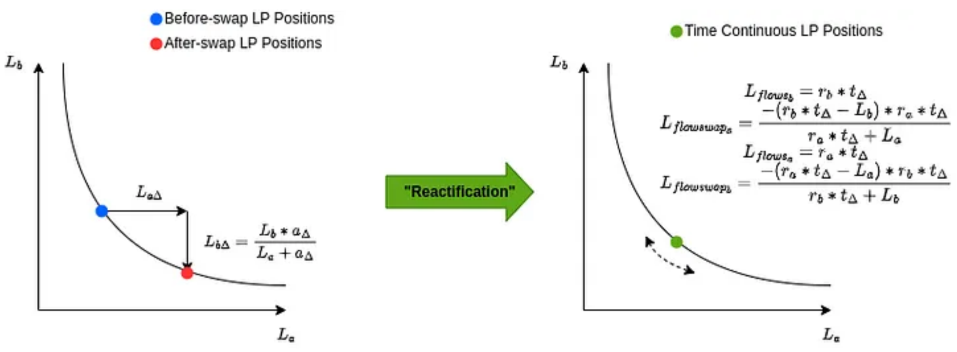
```
from sage.all import var, assume, function, solve

def CLP(x, y, x_prime, y_prime):
    """Constant Liquidity Product Swap"""
    return x * y == x_prime * y_prime
```

```
def solve_rclp_rtb_bidir():
    print("# Solve Reactive CLP rtb_bidir equation\n")
    cf_a = r_a * (t - t_0)
    cf_b = r_b * (t - t_0)

    q = var("q")
    clp = CLP(
        L_a,
        L_b,
        L_a + cf_a + q * cf_b,
        L_b + cf_b + 1/q * cf_a
    )
    sols = solve(clp, q)
    print("L_{flowswap_a} =", (1/q * cf_a).subs(sols[0]))
    print("L_{flowswap_b} =", (q * cf_b).subs(sols[0]))
    print("\n")
```

$$L_{flowswap_a} = \frac{-(r_b * t_\Delta - L_b) * r_a * t_\Delta}{r_a * t_\Delta + L_a}$$
$$L_{flowswap_b} = \frac{-(r_a * t_\Delta - L_a) * r_b * t_\Delta}{r_b * t_\Delta + L_b}$$

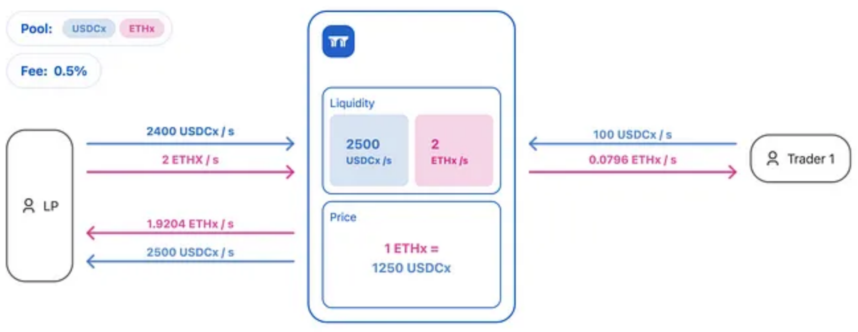

● Before-swap LP Positions
● After-swap LP Positions
L_b
$L_{a\Delta}$
$L_{b\Delta} = \frac{L_b * a_\Delta}{L_a + a_\Delta}$
L_a
"Reactification"
● Time Continuous LP Positions
L_b
$L_{flows_b} = r_b * t_\Delta$
$L_{flowswap_a} = \frac{-(r_b * t_\Delta - L_b) * r_a * t_\Delta}{r_a * t_\Delta + L_a}$
$L_{flows_a} = r_a * t_\Delta$
$L_{flowswap_b} = \frac{-(r_a * t_\Delta - L_a) * r_b * t_\Delta}{r_b * t_\Delta + L_b}$
L_a

From CFMM to Reactive CFMM

This mode of swap requires bootstrapping of liquidity.

Zero-Intermediate-Liquidity Market Maker (ZILMM)

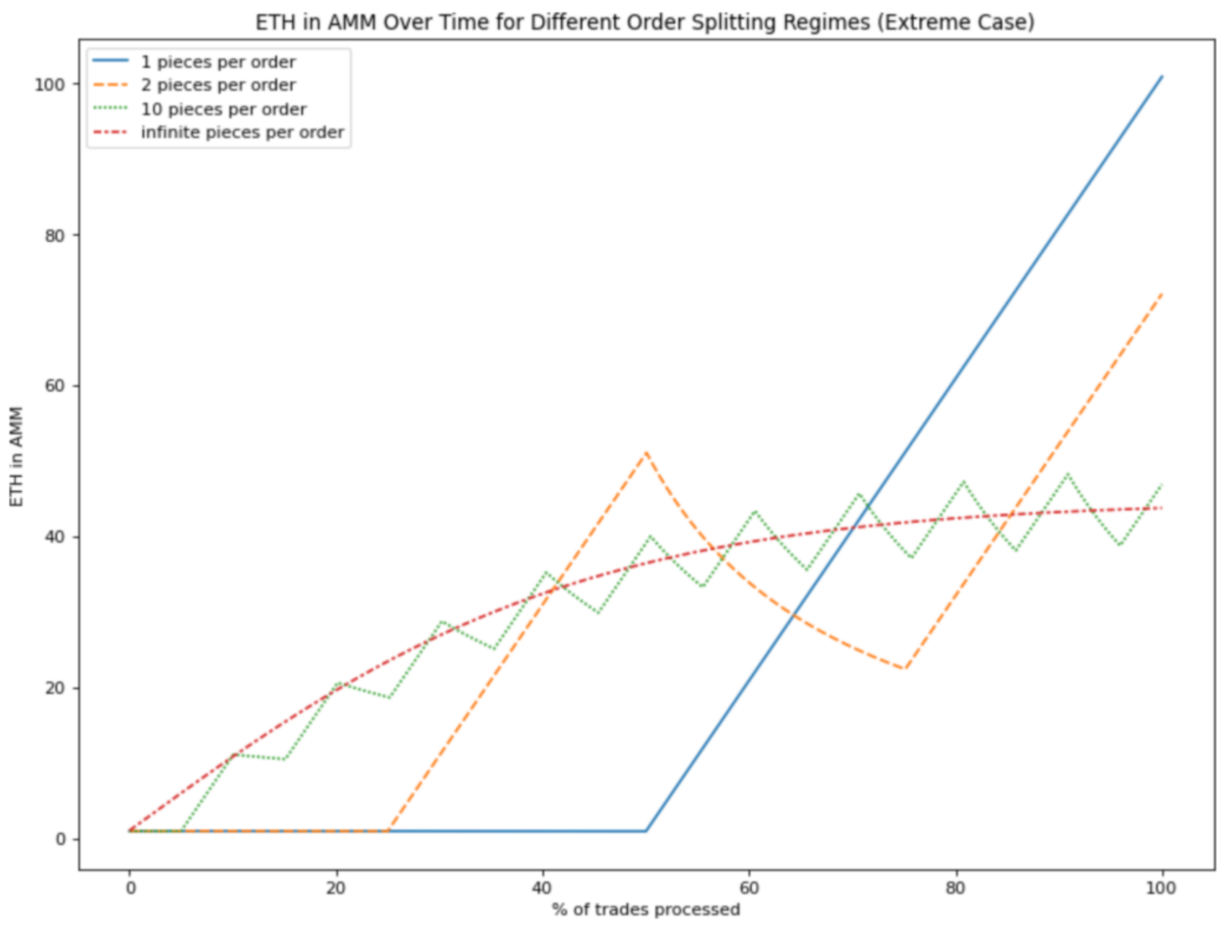
As its name suggests, such a swap mode is notable for its property of zero-intermediate liquidity. The swap is a semantic money “gadget” that routes money flows among liquidity providers and traders:



ZILMM money flow routing.

Reactive TWAMM

A notable prior art that uses TWAP is *time-weighted average market maker*, or TWAMM from paradigm [TWAMM]. Such AMM, when implemented, would split trades into pieces and price them using TWAP.



TWAMM with order splitting regimes.

There was prototypical work from the aqueduct finance team, which combined TWAMM and Superfluid money flow automation [aqueduct-twamm-hook].

Appendix B: Basic Liquidity Mover

👉 Basic liquidity mover is a reference implementation of liquidity mover.

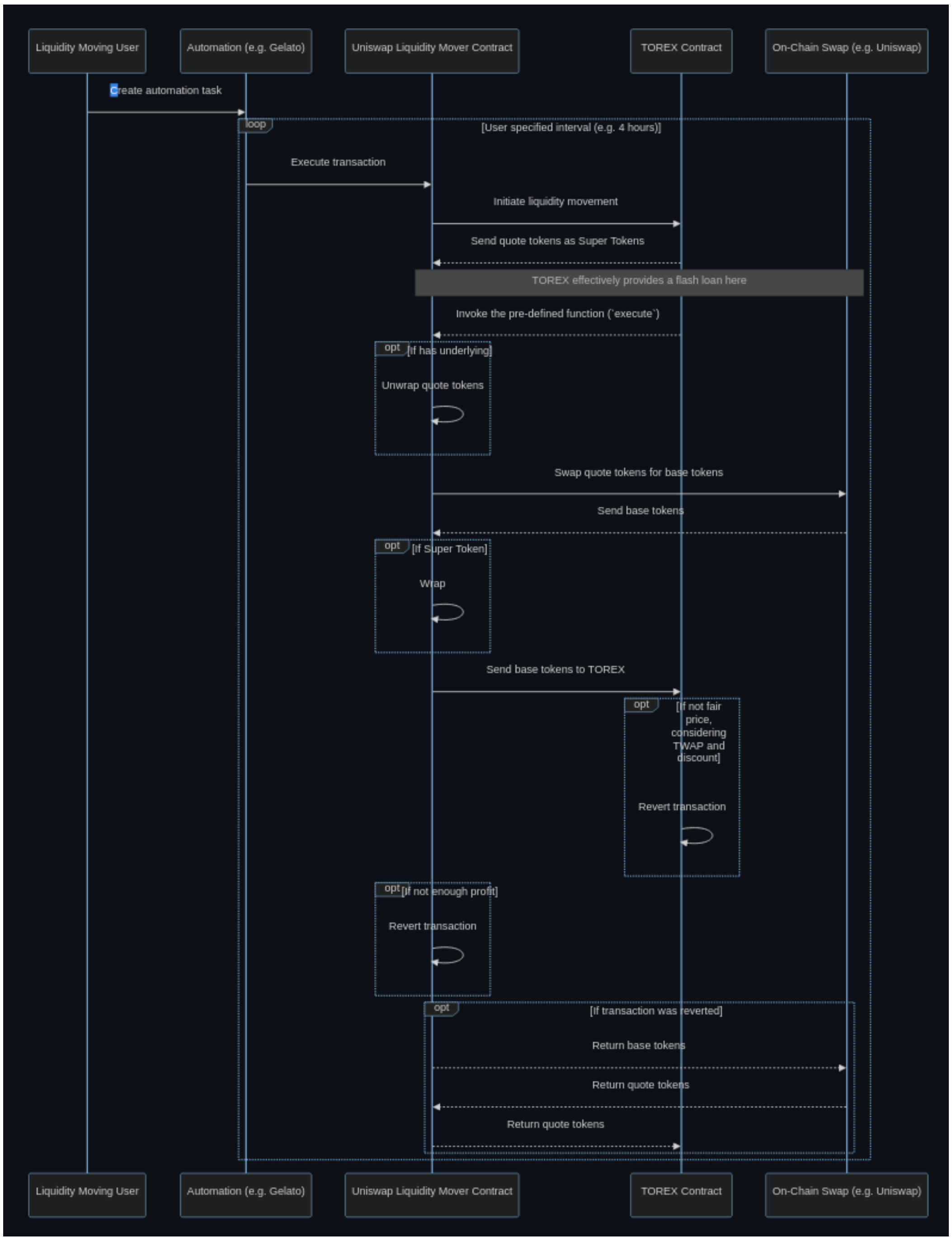
The basic liquidity mover is a fully on-chain implementation of the TOREX liquidity mover.

It aims to address a particular set of requirements:

- An easily deployable solution that can be used as a liquidity mover for various TOREX pools.
- Minimize development and maintenance work by using existing products for off-chain scheduling and transaction execution.
- Have the solution be secure and reliable.
- Make the solution profitable to run.

The solution uses the Uniswap V3 pool as its liquidity source to keep it simple.

This is the sequence for the basic liquidity mover:


Liquidity Moving User
Automation (e.g. Gelato)
Uniswap Liquidity Mover Contract
TOREX Contract
On-Chain Swap (e.g. Uniswap)
Create automation task
loop
[User specified interval (e.g. 4 hours)]
Execute transaction
Initiate liquidity movement
Send quote tokens as Super Tokens
TOREX effectively provides a flash loan here
Invoke the pre-defined function (`execute`)
opt
[If has underlying]
Unwrap quote tokens
Swap quote tokens for base tokens
Send base tokens
opt
[If Super Token]
Wrap
Send base tokens to TOREX
opt
[If not fair price, considering TWAP and discount]
Revert transaction
opt
[If not enough profit]
Revert transaction
opt
[If transaction was reverted]
Return base tokens
Return quote tokens
Return quote tokens

The actual code can be accessed at: https://github.com/superfluid-finance/torex-basic-liquidity-mover/blob/main/src/LiquidityMover.sol.