A Netting Protocol for Liquidity-saving Automated Market Makers

Margherita Renieri[1],*, Letterio Galletta[1], Alberto Lluch Lafuente[2] and James Hsin-yu Chiang[3]

[1]*IMT School for Advanced Studies Lucca, Italy*
[2]*Technical University of Denmark, Denmark*
[3]*Aarhus University, Denmark*

Abstract

Automated Market Makers are one of the most used Decentralized Finance services. They allow users to exchange crypto-assets without a third party. Current protocols have strong constraints related to the liquidity level that users' balances must satisfy for each transaction. In this paper, we propose a liquidity-saving mechanism that aims at reducing the required amount of liquidity in an AMM service. We provide an operational semantics of such a mechanism that precisely characterizes the interactions between users and AMMs and the conditions when the liquidity-saving mechanism is triggered. Our mechanism collects the proposed transactions in a finite queue, providing a global perspective of all users' actions. Starting from the queue, it finds a feasible transaction sequence that satisfies the users' balances. Finally, it performs these transactions on the blockchain atomically, reaching a state where all liquidity constraints are met. By doing so, the mechanism allows for novel liquidity saving behavior for multi-party exchange and multi-AMM arbitrage with less upfront liquidity as usually required.

Keywords

Formal methods, Operational semantics, Liquidity-Saving Mechanism, Netting, AMMs

1. Introduction

In Decentralized Finance (DeFi) systems, Automated Market Makers (AMMs) are decentralized exchanges (DEXs) that provide users with liquidity using different reserves, enabling them to trade different crypto-assets, i.e., tokens. Users participating in these token exchanges pay fixed trading and gas fees charged by the Ethereum network. Similarly to traditional financial systems, users need to provide an upfront amount of crypto-assets as collateral to operate with such decentralized financial services. Liquidity is essential for AMMs to work properly, otherwise, it is possible to incur scenarios that hinder their economic performance and limit user involvement.

AMMs rely on users who act as liquidity providers by depositing a certain amount of tokens into AMM smart contracts that can be used for trades. AMM protocols implement *liquidity-saving mechanisms* (LSM) that aim at reducing liquidity costs by minimizing the required liquidity for transaction settlement. DeFi protocols have introduced various approaches to achieve this goal, e.g., flash loans [1, 2] and flash swaps [3]. These mechanisms allow users to temporarily loan tokens with no collateral but require them to pay back their debt within a single transaction. One of their limitations is that they do not easily permit scenarios with more users involved or trades requiring more than one transaction to be completed.

In this paper, we propose the design of an LSM for AMMs that offers the advantages of flash loans and flash swaps but overcomes their limitations. In particular, our mechanism permits users to operate on the AMM without upfront balance by aggregating multiple actions over a specific time frame. Our aggregation mechanism is inspired by *netting* [4, 5, 6], an LSM used in traditional finance: netting creates payment instruction queues among banks, offsets the value of multiple payments, and finds the most

DLT 2024 : 6th Distributed Ledger Technology Workshop, May 14-15, 2024, Torino, TO
*Corresponding author.
margherita.renieri@imtlucca.it (M. Renieri); letterio.galletta@imtlucca.it (L. Galletta); albl@dtu.dk (A. Lluch Lafuente); jachiang@cs.au.dk (J. H. Chiang)
0000-0001-6987-1881 (M. Renieri); 0000-0003-0351-9169 (L. Galletta); 0000-0001-7405-0818 (A. Lluch Lafuente); 0000-0002-5126-9494 (J. H. Chiang)

convenient payment combination to eliminate stunting periods. We provide here a formalization of our mechanism in terms of a labelled transition system (LTS) that precisely characterizes the interactions between users and AMMs and the conditions when the netting mechanism is triggered.

Intuitively, our mechanism works as follows. First, we equip AMMs with a finite queue to store trading transactions (swaps) proposed by users. When a user performs a transaction that would result in a balance overdraft, the transaction is stored in the queue. Otherwise, if the transaction has the effect of setting all balances of users having pending transactions in the queue to positive, the transaction queue is executed atomically, resulting in a liquidity-saving sequence of swaps. If the queue reaches its maximum capacity where some users' balances are still negative, a netting algorithm selects a subset of the stored transactions that meet the liquidity constraints for each user, if any, and atomically performs them. In such a way, users capitalize on market inefficiencies, facilitating the trading of assets more efficiently and cost-effectively. We discuss the scenarios where our proposed netting-based AMM and traditional AMM differ in the performed swap transactions. In particular, we show examples of multi-party trading operations that are not executed in standard AMMs due to lack of liquidity but are settled by ours.

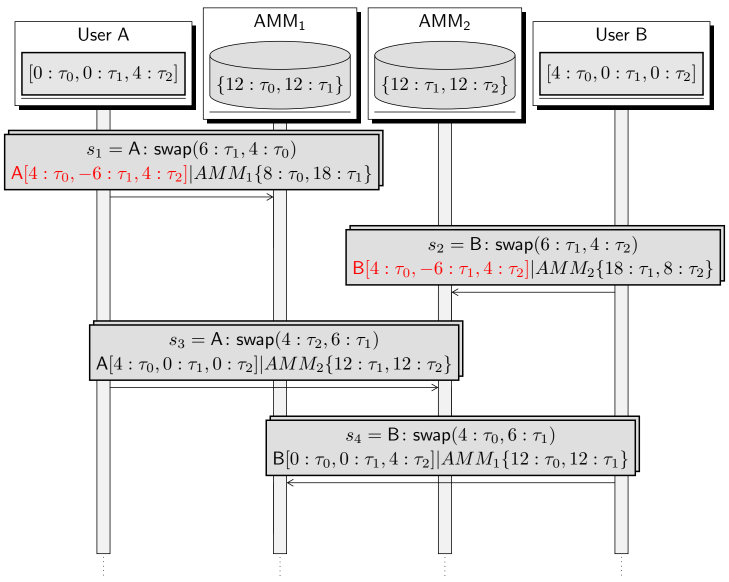
In summary, our main contributions are:

- We provide an LSM that settles AMM trades through a netting algorithm and allows us to implement liquidity-optimized AMM. We illustrate examples that allow users to perform multi-party exchanges and multi-AMM arbitrage without the upfront liquidity that would be required in traditional AMMs.
- We formalize our mechanism by adopting the operational semantics of the interactions between users and AMMs proposed by Bartoletti *et al.* [7]. We adapt their semantics by removing the liquidity constraints on users' balances when performing swap actions and by appending swap transactions in a queue until a liquidity-balanced state is reached.
- We provide a lightweight netting algorithm that can run on the blockchain as part of a smart contract to select a subset of swaps that can actually be performed without violating any liquidity constraints.

In the rest of the paper we proceed as follows. We provide a high-level overview of our approach in Section 2. Section 3 presents the operational semantics of our LSM mechanism, together with relevant scenarios comparing it with standard AMMs. Section 4 discusses related literature, and Section 5 concludes and discusses some future work.

2. Overview of the approach

In current DeFi protocols, such as Uniswap [3], a user interacts with the AMM reserve through single actions. She can deposit, redeem, or swap crypto-assets in change of others in the AMM. However, she can perform the transaction only if she has the needed capital upfront to cover it. For instance, consider the example of Figure 1 where there are two users A and B that interact with two AMMs performing a sequence of swap actions $s_1 s_2 s_3 s_4$. Initially, user A owns 4 tokens of type τ_2, no tokens of types τ_0 and τ_1 (denoted with the notation $\mathsf{A}[0 : \tau_0, 0 : \tau_1, 4 : \tau_2]$ in the figure); whereas user B has 4 tokens of type τ_0, no tokens of types τ_1 and τ_2. The AMMs have 12 tokens type of τ_0, τ_1 and τ_1, τ_2, respectively (denoted with $\{12 : \tau_0, 12 : \tau_1\}$ and $\{12 : \tau_1, 12 : \tau_2\}$ in the figure). The main idea is that the sequence of transactions would allow A and B to exchange their tokens at 1-to-1 rate, via the two AMMs. In standard protocols, some transactions are rejected because users' balances cannot cover their fulfillment: A cannot perform s_1, namely swapping 6 tokens for at least 4 tokens of type τ_0, in symbols $s_1 = \mathsf{A}\colon \mathsf{swap}(6 : \tau_1, 4 : \tau_0)$, and B cannot perform s_2, in symbols $s_2 = \mathsf{B}\colon \mathsf{swap}(6 : \tau_1, 4 : \tau_2)$. Our LSM overcomes the previous scenario. It allows users to instantaneously perform transactions whenever they lead to a state where the overall balance is positive. Otherwise, these transactions are delayed and stored in a queue. Once the queue is full, the protocol executes a netting procedure to discard the transactions that make balances negative. Thus, the protocol reaches a state where no


User A
$[0:\tau_0, 0:\tau_1, 4:\tau_2]$
AMM$_1$
$\{12:\tau_0, 12:\tau_1\}$
AMM$_2$
$\{12:\tau_1, 12:\tau_2\}$
User B
$[4:\tau_0, 0:\tau_1, 0:\tau_2]$
$s_1 =$ A: swap$(6:\tau_1, 4:\tau_0)$
A$[4:\tau_0, -6:\tau_1, 4:\tau_2] \| AMM_1\{8:\tau_0, 18:\tau_1\}$
$s_2 =$ B: swap$(6:\tau_1, 4:\tau_2)$
B$[4:\tau_0, -6:\tau_1, 4:\tau_2] \| AMM_2\{18:\tau_1, 8:\tau_2\}$
$s_3 =$ A: swap$(4:\tau_2, 6:\tau_1)$
A$[4:\tau_0, 0:\tau_1, 0:\tau_2] \| AMM_2\{12:\tau_1, 12:\tau_2\}$
$s_4 =$ B: swap$(4:\tau_0, 6:\tau_1)$
B$[0:\tau_0, 0:\tau_1, 4:\tau_2] \| AMM_1\{12:\tau_0, 12:\tau_1\}$
Figure 1: Interaction between two Users and two AMMs.

liquidity constraints are violated, and transactions can be safely performed. The underlying idea of our mechanism is to accept a momentary deficit in users' balances as long as they may be covered by subsequent transactions, for instance, a swap in a different direction made by the same user or an update to the token reserve. Back to the example of Figure 1, our approach allows the sequence of the actions $s_1 s_2 s_3 s_4$ (for each action in the figure, we report how it affects the user balances and AMM reserves) to be executed since the s_4 yields a state where all balances are positive. It is not relevant if there are intermediate states where user balances are temporarily negative (colored in the figure). In this way, our LSM permits a multi-party trade between A and B, even when they do not have sufficient funds. In contrast, traditional AMMs reject all four swap actions due to lack of liquidity.

3. Liquidity-saving mechanism

This section formalizes our LSM, illustrates our netting algorithm, and shows how our approach differs from the standard one through some examples.

3.1. Formal model of Automated Market Makers

We formalize the interaction between users and AMMs as a labelled transition system (LTS). Our semantics is based on those of Bartoletti *et al.* [7]. Intuitively, LTS states represent the system configurations that store each user's token supplies and the AMMs. Transitions represent transactions performed by users, which may result in an update of token supplies. For the time being, we focus only on swap actions and neglect redeem and deposit. We also assume that swap actions require no fees and that there are no transaction fees.

3.1.1. AMM basic definitions

We assume a set $\mathbb{T}$ of *atomic token types* (ranged over by τ, τ') representing native cryptocurrencies and application-specific tokens. In general, we denote with r, r' real numbers ($\mathbb{R}$), and we write $r\colon \tau$ to denote r atomic tokens of type τ ($\tau \in \mathbb{T}$).

We also assume a set of *users* $\mathbb{A}$ (ranged over by A, A'). The *wallet* of a user A is denoted by A$[\sigma_{\mathsf{A}}]$, where the partial map $\sigma_{\mathsf{A}} \in \mathbb{T} \rightharpoonup \mathbb{R}$ represents A token balances, i.e., $\sigma_{\mathsf{A}}(\tau)$ denotes the amount of τ owned by A.

We model the AMM as a reserve of $r_0 : \tau_0$ and $r_1 : \tau_1$ where $\tau_0 \neq \tau_1$, written as an unordered pair $\{r_0 : \tau_0, r_1 : \tau_1\}$.

The *states* Γ, Γ' of the protocol are finite non-empty compositions of wallets and AMM. Formally, they are defined as follows:

$$\Gamma = [\mathsf{A}_1[\sigma_{\mathsf{A}_1}] | \cdots \mid \mathsf{A}_n[\sigma_{\mathsf{A}_n}] \mid \{r_0 : \tau_0, r_1 : \tau_1\} \mid \cdots \mid \{r_w : \tau_w, r_k : \tau_k\}] \tag{1}$$

and for the sake of simplicity of our formalization, we assume the following conditions: for all $i \neq j$ (i) each user has a single wallet ($\mathsf{A}_i \neq \mathsf{A}_j$); (ii) distinct AMMs cannot hold exactly the same token types ($\{\tau_i, \tau_i'\} \neq \{\tau_j, \tau_j'\}$).

The labels of LTS are swap transactions that are terms of the form

$$\mathsf{A}\colon \mathsf{swap}(x : \tau_0, y^* : \tau_1) \tag{2}$$

meaning that user A transfers $x : \tau_0$ to the AMM $\{r_0 : \tau_0, r_1 : \tau_1\}$ specifying that she wants to receive back at least y^* units of token τ_1 in return. The amount of token y^* must meet the constraint $0 < y^* \leq y$ where $y = x \cdot SX(x, r_0, r_1)$ (see below for SX) to preserve the *constant-product swap rate* of the reserve. In our mechanism, y^* represents a minimum acceptable threshold for the amount of token a user expects to receive from an exchange with an AMM reserve. A transaction will be rejected if the swap rate does not ensure the user receives at least this amount. Users submit their actions without knowing whether they will be executed at the resulting swap rate. In our formalization, we adopt the *Constant Product Market Maker* (CPMM) which is one of the most popular forms of DEX that considers a trade valid if the value of the reserve before and after (with an additional amount for fees) is constant or simply the same. More precisely, we consider the *constant product swap rate* that is the most commonly implemented in Uniswap [8], SushiSwap [9], and Curve [10] protocols. We use the swap rate function in all instances throughout this paper. In particular, when a user swaps a token τ_0 with an AMM $\{r_0 : \tau_0, r_1 : \tau_1\}$, the actual amount y of τ_1 is calculated using the following formula:

$$SX(x, r_0, r_1) = \frac{r_1}{(r_0 + x)} \tag{3}$$

This rate ensures that the evolution and the update of an AMM from $\{r_0 : \tau_0, r_1 : \tau_1\}$ to $\{r_0 + x : \tau_0, r_1 - y : \tau_1\}$ preserve the ratio between the reserve maintaining a constant level of tokens:

$$r_0 \cdot r_1 = (r_0 + x) \cdot (r_1 - x \cdot \frac{r_1}{r_0 + x}) = r_0 \cdot r_1 \tag{4}$$

The formula above is derived by substituting the definition of $SX(x, r_0, r_1)$ from the equation (3) in place of y, then, by performing some algebraic simplification to obtain again $r_0 \cdot r_1$. The actions that result in overdrafts on users' balances are not immediately executed but are stored in a queue Λ. Formally, a queue is a term obtained by the following grammar:

$$\Lambda := \emptyset \mid \Lambda \circ s \tag{5}$$

where $\emptyset$ denotes an empty queue with no pending actions, and $\Lambda \circ s$ a queue Λ where its head is the action s. In the semantic rules below, we use $\Lambda \circ s$ to represent when an action s is added to the queue Λ. Moreover, we write $|\Lambda|$ to indicate the size of the queue Λ. For instance, $\emptyset$ is an empty queue with no pending action; while the term $\emptyset \circ s1$ denotes a queue with a single action s_1 that is equal to $\mathsf{A}\colon \mathsf{swap}(6 : \tau_1, 4 : \tau_0)$; and the term $\emptyset \circ s1 \circ s2$ denotes a queue with two actions: the tail is $s_1 = \mathsf{A}\colon \mathsf{swap}(6 : \tau_1, 4 : \tau_0)$ and the head is s_2 that corresponds to $\mathsf{B}\colon \mathsf{swap}(4 : \tau_2, 6 : \tau_1)$. The LTS states are *configurations* defined as a 3-tuple $\langle \Gamma, \Gamma', \Lambda \rangle$ where the first component is a state Γ of the form (1), the second one Γ' is the last simulated state (see the next subsection), and the third one is a swap action queue Λ of the form (5) of size ℓ. Given a state Γ we define $\langle \Gamma, \Gamma, \emptyset \rangle$ as an *initial configuration.*

3.1.2. AMM semantics

Transitions $\langle \Gamma, \Gamma', \Lambda \rangle \xrightarrow{s} \langle \Gamma'', \Gamma''', \Lambda' \rangle$ from a configuration to another one are triggered when a user submits a transaction s, and are defined by the inference rules below.

It is convenient to introduce some notation for the semantics. We say that a state Γ is `green` when the token supply of each user is non-negative, formally:

$$\forall\, \mathsf{A} \in \mathbb{A} \cap \Gamma, \tau \in \mathbb{T}, \text{ s.t. } \sigma_{\mathsf{A}}(\tau) \geq 0$$

where we denote with $\mathbb{A} \cap \Gamma$ the set of users occurring in the configuration Γ. While we say it is **`red`** otherwise, namely if

$$\exists\, \mathsf{A} \in \mathbb{A} \cap \Gamma, \tau \in \mathbb{T}, \text{ s.t. } \sigma_{\mathsf{A}}(\tau) < 0$$

Moreover, we denote with $\Gamma \overset{s}{\Rrightarrow} \Gamma'$ a simulation of the transaction s in the state Γ using the semantic rules of [7] without constraints on users' balances. This is why we want to maximize the number of settled actions performed by users. Differently, we maintain checks to impose non-negativity constraints on the AMM reserve. This means that some user balances could be negative in the resulting state Γ' but not AMM reserves. Given a sequence of transactions Λ, we denote with $\Gamma \overset{\Lambda}{\Rrightarrow} \Gamma'$ the simulation of the actions in Λ. Also, we assume that when Λ is the empty sequence ϵ, the simulation results in an unchanged state, i.e., $\Gamma \overset{\epsilon}{\Rrightarrow} \Gamma$. When a transition is triggered by an action s in a configuration $\langle \Gamma, \Gamma', \Lambda \rangle$, we have three possible scenarios.

The first scenario arises when the simulation of s in Γ' reaches a `green` state Γ''. In this case, we apply the following rule:

$$\frac{\Gamma' \overset{s}{\Rrightarrow} \Gamma'' \qquad \Gamma''\texttt{green}}{\langle \Gamma, \Gamma', \Lambda \rangle \;\xrightarrow{s}\; \langle \Gamma'', \Gamma'', \emptyset \rangle}\ \textsc{Cover}$$

The rule Cover performs all the pending actions in Λ (if any), and the resulting configuration consists of the `green` state, Γ'' (in the first two components), and the emptied queue.

The second scenario happens when the simulation of s in Γ' reaches a **`red`** state Γ'', and the length of Λ is less than ℓ (max queue size, a parameter of our mechanism). In this case, we apply the following rule:

$$\frac{\Gamma' \overset{s}{\Rrightarrow} \Gamma'' \qquad \Gamma''\texttt{red} \qquad |\Lambda| < \ell \qquad \Lambda' = \Lambda \circ s}{\langle \Gamma, \Gamma', \Lambda \rangle \;\xrightarrow{s}\; \langle \Gamma, \Gamma'', \Lambda' \rangle}\ \textsc{Overdraft}$$

The rule Overdraft enqueues s in Λ, thus, the resulting configuration consists of the current state Γ, the **`red`** state Γ', and Λ' that is Λ extended with the incoming action s.

The last scenario occurs when the simulation of s leads to a **`red`** state Γ'', and the length of Λ equals ℓ. In this case, we use the netting procedure (denoted by the function net in the following rule) to perform the *settlement* from the state Γ and the actions of Λ plus s.

The netting procedure identifies and returns a (possibly empty) sequence of feasible actions Λ'. Such sequence is a subset of the input sequence Λ. The obtained subsequence Λ' is executed to obtain the resulting state Γ^*. All the other actions of Λ that are not selected by the procedure are discarded.

The resulting configuration is a 3-tuple with the state Γ^* (for the first two components) and the empty queue for the last one (the queued actions were carried out or discarded). Formally, we apply the following rule:

$$\frac{\Gamma' \overset{s}{\Rrightarrow} \Gamma'' \quad \Gamma''\texttt{red} \quad |\Lambda| = \ell \quad \Lambda' = net(\Gamma, \Lambda \circ s) \quad \Gamma \overset{\Lambda'}{\Rrightarrow} \Gamma^*}{\langle \Gamma, \Gamma', \Lambda \rangle \;\xrightarrow{s}\; \langle \Gamma^*, \Gamma^*, \emptyset \rangle}\ \textsc{Netting}$$

It is worth remarking that the mechanism provided in this section is agnostic with respect to the netting procedure that we invoke in a black-box manner. For example, a simple procedure could be to discard all enqueued transactions. The next section introduces a more meaningful procedure. Also, we remark that when we apply the above rule, the queue Λ contains a prefix of actions leading to a **red** state that is not balanced by other actions in the queue. Otherwise, the rule Cover would be applicable. The rule invokes the netting procedure to execute a subset of actions that allows the AMM to progress.

Finally, note that our configurations and semantic rules could be written without Γ' (the 2nd component of the configuration recording the simulated/speculative final state) and replacing the premise $\Gamma' \overset{s}{\Rightarrow} \Gamma''$ with $\Gamma \xRightarrow{\Lambda \circ s} \Gamma''$. This approach requires re-computing the final state every time a new action is performed. These re-computations are not efficient in an implementation where it is more convenient to store the intermediate state. We decided to follow this approach to make our formalization coherent with the implementation.

3.2. Netting problem

Here, we provide an algorithm for the netting procedure *net* invoked in Netting rule above. The netting problem is defined as follows: Given as inputs a state Γ, a queue Λ, find a subset Λ' of Λ, whose actions lead to a green state Γ^* from Γ. Recall that the *liquidity constraints* required to each user are satisfied in a green state. Ideally, netting aims at finding an optimal solution, maximizing a specific parameter. For example, one may want to maximize the size of Λ', the number of users affected, or the amount of tokens involved. We formalize the netting as an optimization problem that, given as inputs a state Γ of the LTS and a queue Λ, maximizes the size of Λ', a queue, whose actions bring to a valid state Γ^* that meet the liquidity constraints:

$$\max |\Lambda'| \tag{6}$$

$$\text{subject to}$$

$$\Gamma \xrightarrow{\Lambda'} \Gamma^* \tag{7}$$

$$\Gamma^* \ \texttt{green} \tag{8}$$

Note that the *liquidity constraints* (8) are satisfied if the final state Γ^* is a green (there is no overdraft in users' wallets). The problem in financial literature is known as *bank payment clearance*, which is *NP-hard* [4, 11], so we adopt a heuristic algorithm to implement the *netting* procedure that avoids enumerations and tries to maximize the number of transactions performed.

Since we need an algorithm that can run on a smart contract (thus incurring affordable gas expenses), we adopt a heuristic approach that sacrifices optimality for efficiency, and we propose *Algorithm 1* that runs in polynomial time (quadratic in the size of the queue, at worst).

Algorithm 1 Heuristic netting procedure implementation

```
Input: $\Lambda = [s_1, s_2, \dots, s_l],\ \Gamma$
Output: $\Lambda_r$
Initialization: $\Lambda_r \leftarrow \Lambda,\ \Gamma_0 \leftarrow \Gamma$
1: $\Gamma_0 \xRightarrow{\Lambda_r} \Gamma_l$                                  ▷ Starting simulation
2: while $\Gamma_l$ red $\wedge\ \Lambda_r \neq \emptyset$ do                  ▷ Final state has overdraft
3:     select min $i$ s.t. $s_i \in \Lambda_r$ where $\sigma_A(\tau) < 0$   ▷ Select action that overdraft
4:     $\Lambda_r \leftarrow \Lambda_r - s_i$                                  ▷ Update actions queue
5:     $\Gamma_i \leftarrow \Gamma_{i-1}$                                      ▷ Update last green state
6:     $\Gamma_i \xRightarrow{s_{i+1}} \Gamma_{i+1} \xRightarrow{s_{i+2}} \Gamma_{i+2} \cdots \xRightarrow{s_f} \Gamma_f$
7:     $\Gamma_l \leftarrow \Gamma_f$                                          ▷ Update the variable $\Gamma_l$
8: end while
9: return $\Lambda_r$
```

Intuitively, *Algorithm 1* simulates the swap actions ignoring the liquidity constraints. It starts by initializing Λ_r with Λ, the initial queue of pending actions, and Γ_0 with Γ, the initial state.

It then starts a loop (line 2), where it simulates action execution until either the final state Γ_l is not **red** or the queue Λ_r becomes empty, namely, until there are still actions to be processed and overdrafts in the system. During each iteration, we select from Λ_r the first action s_i that makes the balance of some account A become negative, i.e., the execution of s_i leads Γ_l to **red** state (line 3). Then, the algorithm removes s_i from Λ_r (line 4) and updates the simulation state Γ_i (line 5) by reverting s_i. We achieve that by simply considering the previous state Γ_{i-1} that is the last green state. After we recover to the last green state, the simulation is run again but from Γ_i using the actions following s_i until all balances are non-negative. As a result of this last simulation, we obtain the green state Γ_f (line 6). Finally, Γ_l is updated with the final state Γ_f (line 7) and the loop starts again.

These steps are iterated until we obtain a green state or empty Λ_r. When this happens, the algorithm returns Λ_r. In the following section, we provide an example of execution of *Algorithm 1*.

3.3. Liquidity-saving behavior

Our mechanism enables liquidity-saving behavior that is not possible in ordinary AMMs. An example is the simultaneous change of tokens through AMMs that we presented in Section 2. Another interesting behavior enabled by our mechanism is the ability to perform arbitrage on multiple AMMs simultaneously and with no liquidity.

3.3.1. Liquidity-saving arbitrage Example

Assume to have three AMMs with the following token reserves

$$\{8 : \tau_0, 18 : \tau_1\}, \{8 : \tau_1, 18 : \tau_2\}, \{8 : \tau_2, 18 : \tau_0\}$$

If we assume, a 1-to-1 exchange rate, we could see three arbitrage opportunities, only available to users with sufficient funds. Now suppose that a user A has no tokens but wants to perform the following actions $s_1 s_2 s_3$:

$$\mathsf{A}\colon \mathsf{swap}(4 : \tau_0, 6 : \tau_1), \mathsf{A}\colon \mathsf{swap}(4 : \tau_1, 6 : \tau_2), \mathsf{A}\colon \mathsf{swap}(4 : \tau_2, 6 : \tau_0)$$

When considering AMMs without our mechanism, all the actions are discarded because A does not have an upfront balance. Figure 2 shows a flow diagram of the actions performed adopting our mechanism. Now $s_1 s_2$ are enqueued because they cause an overdraft that is then covered by s_3. The execution of s_3 results in the following green state:

$$[\mathsf{A}[2 : \tau_0, 2 : \tau_1, 2 : \tau_2]|\{12 : \tau_0, 12 : \tau_1\}|\{12 : \tau_1, 12 : \tau_2\}|\{12 : \tau_2, 12 : \tau_0\}]$$

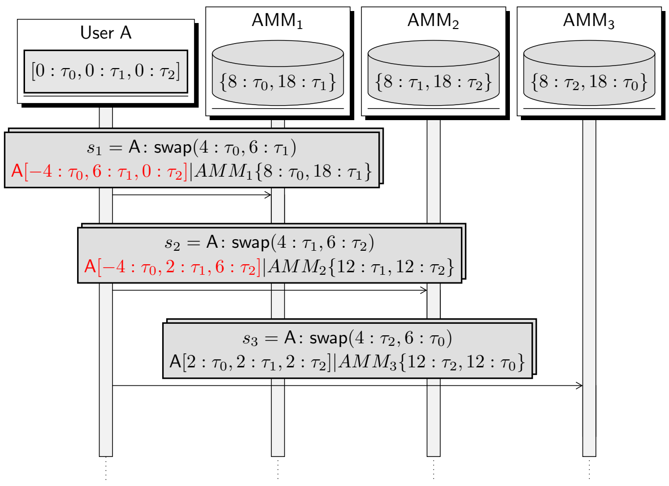
The above scenario has similarities with the traditional finance scenario known as *gridlock* [6] where banks cannot settle their payments individually due to their insufficient liquidity. Through *netting*, each bank submits its payment instructions to designated queues, performing multilateral settlement exclusively for the net obligations. Back to our example, user A overcomes the stuck state with possible no evolution of the system of traditional AMMs protocol where individual swap incurs in an overdraft, enqueuing her actions and atomically perform them when reaching a positive balance.

The approach we developed may settle transactions on the blockchain differently than the ones used by standard AMM protocols: netting can select transactions that the standard approach discards and vice versa. This difference may affect users' rewards and lead them to apply new/different market strategies. The following scenario exemplifies a case in which the netting of transactions could prevent swaps that would otherwise be executed.

3.3.2. Two Users scenario Example

Consider a scenario where there are two users A and B and two AMMs with the following wallets and reserves:

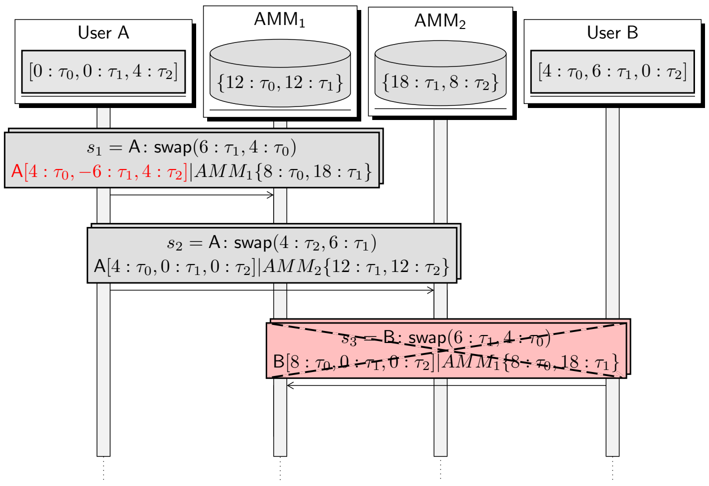
$$[\mathsf{A}[0 : \tau_0, 0 : \tau_1, 4 : \tau_2]|\mathsf{B}[4 : \tau_0, 6 : \tau_1, 0 : \tau_2]|\{12 : \tau_0, 12 : \tau_1\}|\{18 : \tau_1, 8 : \tau_2\}]$$



User A
$[0:\tau_0, 0:\tau_1, 0:\tau_2]$
AMM$_1$
$\{8:\tau_0, 18:\tau_1\}$
AMM$_2$
$\{8:\tau_1, 18:\tau_2\}$
AMM$_3$
$\{8:\tau_2, 18:\tau_0\}$
$s_1 =$ A: swap$(4:\tau_0, 6:\tau_1)$
A$[-4:\tau_0, 6:\tau_1, 0:\tau_2] | AMM_1\{8:\tau_0, 18:\tau_1\}$
$s_2 =$ A: swap$(4:\tau_1, 6:\tau_2)$
A$[-4:\tau_0, 2:\tau_1, 6:\tau_2] | AMM_2\{12:\tau_1, 12:\tau_2\}$
$s_3 =$ A: swap$(4:\tau_2, 6:\tau_0)$
A$[2:\tau_0, 2:\tau_1, 2:\tau_2] | AMM_3\{12:\tau_2, 12:\tau_0\}$

Figure 2: Flow diagram of the Example 3.3.1 among User A and three AMMs.

Assume that user A wants to perform two actions. The first one, $s_1 =$ A: swap$(6 : \tau_1, 4 : \tau_0)$, on the first AMM, $\{12 : \tau_0, 12 : \tau_1\}$, and the second one, $s_2 =$ A: swap$(4 : \tau_2, 6 : \tau_1)$ on the second AMM, $\{18 : \tau_1, 8 : \tau_2\}$. User B wants to perform an action $s_3 =$ B: swap$(6 : \tau_1, 4 : \tau_0)$ on the first AMM.



User A
$[0:\tau_0, 0:\tau_1, 4:\tau_2]$
AMM$_1$
$\{12:\tau_0, 12:\tau_1\}$
AMM$_2$
$\{18:\tau_1, 8:\tau_2\}$
User B
$[4:\tau_0, 6:\tau_1, 0:\tau_2]$
$s_1 =$ A: swap$(6:\tau_1, 4:\tau_0)$
A$[4:\tau_0, -6:\tau_1, 4:\tau_2] | AMM_1\{8:\tau_0, 18:\tau_1\}$
$s_2 =$ A: swap$(4:\tau_2, 6:\tau_1)$
A$[4:\tau_0, 0:\tau_1, 0:\tau_2] | AMM_2\{12:\tau_1, 12:\tau_2\}$
$s_3 =$ B: swap$(6:\tau_1, 4:\tau_0)$
B$[8:\tau_0, 0:\tau_1, 0:\tau_2] | AMM_1\{8:\tau_0, 18:\tau_1\}$

Figure 3: Flow diagram of the Example 3.3.2 among two Users and two AMMs.

When considering an AMM without our mechanism, s_1 and s_2 are discarded because A does not have enough balance, whereas s_3 is performed reaching the configuration:

$$[\mathsf{A}[0 : \tau_0, 0 : \tau_1, 4 : \tau_2] | \mathsf{B}[8 : \tau_0, 0 : \tau_1, 0 : \tau_2] | \{8 : \tau_0, 18 : \tau_1\} | \{18 : \tau_1, 8 : \tau_2\}]$$

Differently with our mechanism s_1 is enqueued, and when A performs s_2 on the second AMM, both actions are settled because s_2 covers the overdraft on A's wallet (see Figure 3). While s_3 is not executed on the first AMM because the execution of the previous actions changes the ratio in the reserve, and hence the swap rate. Once $s_1 s_2$ are performed, the ratio of the first AMM is updated and s_3 is discarded (as the red and crossed box denotes in Figure 3) because user B can swap 6 tokens of τ_1 with 2 tokens of τ_0 that is a different swap amount compared to what she wants to perform (6 tokens of τ_1 with 4 tokens of τ_0).

This example highlights that the standard mechanism and ours generally settle different transactions, so they are incomparable. However, the behavior is not uncommon from what happens in ordinary AMMs where a user (in this case B) would typically decide to trade at a swap rate based on his local view or speculation on the state of AMMs, which can of course change if transactions of other users (in this case A) are executed.

3.3.3. Netting scenario Example

Consider a scenario where there is user A and three AMMs with the following wallet and reserves:

$$[\mathsf{A}[0 : \tau_0, 0 : \tau_1, 4 : \tau_2]|\{12 : \tau_0, 12 : \tau_1\}|\{18 : \tau_1, 8 : \tau_2\}|\{12 : \tau_2, 12 : \tau_0\}]$$

Assume user A wants to execute three actions sequentially: s_1, s_2, and s_3. With s_1 user A swaps 6 tokens of type τ_2 for 4 tokens of type τ_0 on the third AMM, in symbols $s_1 = \mathsf{A}\colon \mathsf{swap}(6 : \tau_2, 4 : \tau_0)$. With s_2 user A swaps 6 tokens of type τ_1 for 4 tokens of type τ_0 on the first AMM, namely $s_2 = \mathsf{A}\colon \mathsf{swap}(6 : \tau_1, 4 : \tau_0)$. Finally, with s_3 user A exchanges 4 tokens of type τ_2 for 6 tokens of type τ_1 on the second AMM: $s_3 = \mathsf{A}\colon \mathsf{swap}(4 : \tau_2, 6 : \tau_1)$.

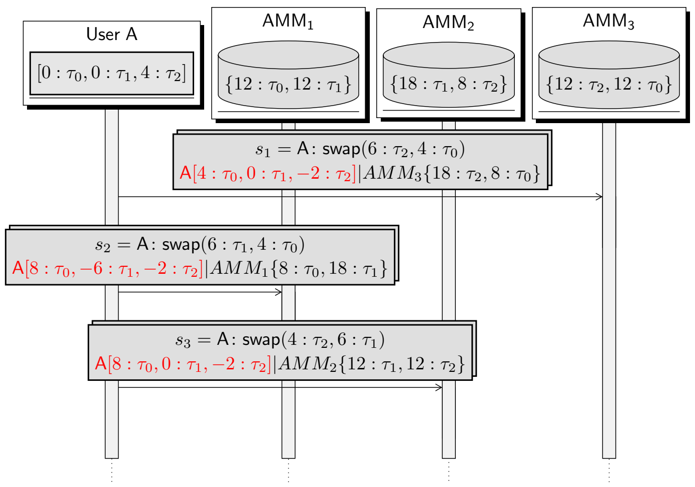

User A
$[0 : \tau_0, 0 : \tau_1, 4 : \tau_2]$
AMM$_1$
$\{12 : \tau_0, 12 : \tau_1\}$
AMM$_2$
$\{18 : \tau_1, 8 : \tau_2\}$
AMM$_3$
$\{12 : \tau_2, 12 : \tau_0\}$
$s_1 = \mathsf{A}\colon \mathsf{swap}(6 : \tau_2, 4 : \tau_0)$
A$[4 : \tau_0, 0 : \tau_1, -2 : \tau_2]|AMM_3\{18 : \tau_2, 8 : \tau_0\}$
$s_2 = \mathsf{A}\colon \mathsf{swap}(6 : \tau_1, 4 : \tau_0)$
A$[8 : \tau_0, -6 : \tau_1, -2 : \tau_2]|AMM_1\{8 : \tau_0, 18 : \tau_1\}$
$s_3 = \mathsf{A}\colon \mathsf{swap}(4 : \tau_2, 6 : \tau_1)$
A$[8 : \tau_0, 0 : \tau_1, -2 : \tau_2]|AMM_2\{12 : \tau_1, 12 : \tau_2\}$

Figure 4: Flow diagram of the Example 3.3.3 among User A and three AMMs.

Upon executing the three actions illustrated in Figure 4, the system achieves a **red** state where the user's wallet fails to meet liquidity constraints. Consequently, our mechanism triggers the NETTING rule and runs the *Algorithm 1*. During this process, s_1 is identified as the first action leading to an overdraft in A's wallet (see line 3 of *Algorithm 1*), and is thus removed from the queue. After this adjustment, the simulation is run again, resulting in a green state, achieved through the execution of s_2 followed by s_3. It is worth noting that without s_1, s_3 covers the overdraft caused by s_2, thus, the reached state is green. This example illustrates how our netting mechanism manages transactions while considering liquidity constraints: our mechanism enables the execution of two actions that subsequently cover the balance overdraft, which would typically not be feasible using a standard semantics to execute the transactions.

4. Related work

The widely spread interest in distributed ledger technology has fostered the development of optimized trading protocols. This section illustrates the most relevant proposals concerning *netting mechanism*, *optimal routing problems*, and *intent-centric protocols*, and compares them with our work.

4.1. Netting mechanism

Several papers have developed decentralized inter-bank payment systems where the role of the payment instructions operator is implemented through a public ledger-based protocol, and the netting mechanism through smart contracts. As a first example of this line of work, Jasper-Ubin Project [12] investigates the possibility of achieving near-instant cross-border payments using blockchain. The project removes the single point of failure and obtains an immediate settlement without transaction reconciliation but does not give a decentralized multilateral netting. Naganuma *et al.* [13] provide a secure netting protocol using the Hyperledger Fabric channel and its access control mechanism. Their secure settlement protocol does not require a specific central server and keeps part of the payment transaction information secret. Similarly, Wang *et al.* [14] introduce a blockchain-based netting solution that relies on a central party and preserves the total amount of liquidity, revealing only the net amount of each bank. Cao *et al.* [6] propose a non-interactive zero-knowledge proof mechanism to post payment instructions on a public ledger. More precisely, each bank locally computes the netting results and submits its proposal to a coordinating smart contract. However, this approach is not robust against cheating users, who can post-invalid partial netting proposals. The proposals above consider blockchain-related technologies to develop standard financial services, whereas our work introduces a standard financial mechanism into DeFi protocols. To the best of our knowledge, ours is the first attempt to apply LSM used by the works above [13, 14, 6] to DeFi services.

4.2. Optimal routing problems

Another relevant research field investigates how to execute several trades of different crypto-assets on networks of multiple CFMMs. These approaches are known as *routing problems* on Decentralized Exchanges. Angeris *et al.* [15] and Diamandis *et al.* [16] pointed out that the optimal routing problem can be formulated as an efficiently solvable optimization problem. Solving such a problem means determining the most efficient path for a trade, i.e., a sequence of crypto-assets exchanges in a network of CFMMs that realizes a given trade, to maximize the user's utility and minimize its costs.

Danos *et al.* [17] introduce arbitrage scenarios within exchange networks and develop an effective global routing system. In detail, they explore how optimal routing strategies are employed to capitalize on price differentials across multiple exchanges, enhancing opportunities for profitable arbitrage. Similar to those approaches, our mechanism aims at finding a solution to maximizing a specific parameter. However, the main difference between ours and the above-mentioned papers [15, 16, 17] is the objective function to maximize. As illustrated in Section 3, our objective function consists of maximizing the number of actions in the queue, i.e., the number of transactions to settle. On the other hand, those proposals formulate the optimization problem in terms of the largest possible user's utility, and their routing problem tries to detect the most convenient and profitable route for executing trades across AMMs of the same and different token types belonging to the network.

4.3. Intent-centric protocols

A significant area of research in Ethereum ecosystem focuses on addressing the aggregated token swap problem, i.e., determining a sequence of swaps on AMMs that realizes a given trade by improving the liquidity of users. Various DeFi applications, such as UniswapX [18], Uniswap V4 [19], and CoW Protocol [20], tackle this challenge by adopting *intent-centric protocols*. These protocols shift the focus from transaction execution to defining users' desired outcomes, creating competitive routing marketplaces, introducing gas-free cross-chain swaps, and incorporating batch auctions to discover profitable prices.

UniswapX [18] implements *intent-centric swaps*, combining on-chain and off-chain liquidity for a competitive trading marketplace. Uniswap V4 [19] enhances liquidity with *hooks*, enabling gas-free cross-chain swaps and offering flexibility through customizable features. CoW Protocol [20] utilizes batch auctions for efficient price discovery, and CoW Swap, its decentralized exchange interface, introduces CoW Hooks, allowing custom actions before and after trades.

While these protocols aim to optimize trading across multiple AMMs, our approach stands out by introducing a Liquidity-Saving Mechanism designed to maximize users' swap actions without a formalized analysis of users' intent.

5. Conclusion

We have proposed liquidity-optimized AMMs, providing an LSM that settles AMM swap actions through a netting algorithm at the application level. We precisely characterized our mechanism by extending the operational semantics proposed by Bartoletti *et al.* [7]. We removed the liquidity constraints on participants' balances when performing swap actions and introduced a queue of actions. When the queue size reaches a fixed bound, and the liquidity constraints are unsatisfied, a greedy algorithm is triggered to compute a sequence of actions with no liquidity violation. Our mechanism enables novel scenarios, allowing users to perform exchanges and arbitrages without the required upfront funds.

In future work, we plan to extend our mechanism to deal with the deposit and redeem actions and take into account price trends, enabling us to consider realistic DeFi protocols. Moreover, we want to introduce the formalization of fixed transaction costs (such as gas costs) in the LTS. Another line of investigation is to understand if knowing the internals of the netting mechanism encourages users to misbehave and take advantage of it. Additionally, we want to extend our mechanism to take into account also other properties, such as, for example, fairness when selecting the actions to remove. Furthermore, we would like to study the impact of the queue length (a key parameter in our mechanism) on the efficiency of our LSM. We also plan to investigate the impact of validators at the consensus layer on our approach and whether they could be the ones promoting transaction orders that result in near-to-optimal queue orderings, hence taking care of the optimal netting problem.

Another line of research involves considering our mechanism work in different AMM models, such as Concentrated Liquidity which has been recently added to Uniswap v3 [21]. Furthermore, our formalization is designed to be implemented as a smart contract on the blockchain. It encompasses the management of various aspects such as user subscriptions, netting procedures with corresponding incentive mechanisms, queue management, and striking a balance between the mechanism's cost and the incentives it offers. An additional research direction involves exploring off-chain management of the actions, either through a trusted third party or in a decentralized manner. This approach aims to ensure the proper behavior of the queue, mitigating potential control issues by validators and simplifying the analysis of incentivized actions for each validator. Finally, we plan to re-model our mechanism, considering users' intents and trying to maximize their utility related to their desired outcome. In this way, we could better analyze user objectives and strategies.

Acknowledgments

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

[1] Aave Documentation, Flash loans, https://docs.aave.com/faq/flash-loans, 2021. Accessed: 2023-11-23.

[2] K. Qin, L. Zhou, B. Livshits, A. Gervais, Attacking the defi ecosystem with flash loans for fun and profit, in: International conference on financial cryptography and data security, Springer, 2021, pp. 3–32.

[3] Uniswap V2 Guide, Flash swaps, https://docs.uniswap.org/contracts/v2/guides/smart-contract-integration/using-flash-swaps, 2023. Accessed: 2023-11-23.

[4] M. M. Güntzer, D. Jungnickel, M. Leclerc, Efficient algorithms for the clearing of interbank payments, European Journal of Operational Research 106 (1998) 212–219.

[5] M. Bech, K. Soramaki, Gridlock resolution in payment systems, Danmarks Nationalbank Monetary Review (2001) 67–79. doi:10.2139/ssrn.274290.

[6] S. Cao, Y. Yuan, A. De Caro, K. Nandakumar, K. Elkhiyaoui, Y. Hu, Decentralized privacy-preserving netting protocol on blockchain for payment systems, in: Financial Cryptography and Data Security: 24th International Conference, FC 2020, Springer, 2020, pp. 137–155.

[7] M. Bartoletti, J. H.-y. Chiang, A. Lluch-Lafuente, A theory of automated market makers in defi, Logical Methods in Computer Science 18 (2022) 12.

[8] H. Adams, Uniswap v3 core, https://uniswap.org/whitepaper-v3.pdf, 2021. Accessed: 2023-07-09.

[9] A. Person, Sushi protocol documentation, https://docs.sushi.com/pdf/whitepaper.pdf, 2020. Accessed: 2024-03-12.

[10] M. Egorov, Stableswap-efficient mechanism for stablecoin liquidity, Retrieved Feb 24 (2019) 2021.

[11] Y. M. Shafransky, A. A. Doudkin, An optimization algorithm for the clearing of interbank payments, European Journal of Operational Research 171 (2006) 743–749.

[12] B. of Canada, M. A. of Singapore, Enabling cross-border high value transfer using distributed ledger technologies, https://www.mas.gov.sg/-/media/Jasper-Ubin-Design-Paper.pdf?la=en&hash=EF5857437C4857373A9287CD86F56D0E7C46E7FF, 2018. Accessed: 2023-07-11.

[13] K. Naganuma, M. Yoshino, H. Sato, N. Yamada, T. Suzuki, N. Kunihiro, Decentralized netting protocol over consortium blockchain, in: International Symposium on Information Theory and Its Applications,ISITA 2018, IEEE, 2018, pp. 174–177. doi:10.23919/ISITA.2018.8664259.

[14] X. Wang, X. Xu, L. Feagan, S. Huang, L. Jiao, W. Zhao, Inter-bank payment system on enterprise blockchain platform, in: 11th IEEE International Conference on Cloud Computing, CLOUD 2018, IEEE Computer Society, 2018, pp. 614–621. doi:10.1109/CLOUD.2018.00085.

[15] G. Angeris, A. Evans, T. Chitra, S. Boyd, Optimal routing for constant function market makers, in: Proceedings of the 23rd ACM Conference on Economics and Computation, 2022, pp. 115–128.

[16] T. Diamandis, M. Resnick, T. Chitra, G. Angeris, An efficient algorithm for optimal routing through constant function market makers, arXiv preprint arXiv:2302.04938 (2023).

[17] V. Danos, H. E. Khalloufi, J. Prat, Global order routing on exchange networks, in: Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Springer, 2021, pp. 207–226.

[18] UniswapX whitepaper, Uniswapx, https://uniswap.org/whitepaper-uniswapx.pdf, 2023. Accessed: 2024-01-21.

[19] Uniswap V4 core whitepaper, Uniswap v4, https://github.com/Uniswap/v4-core/blob/main/docs/whitepaper-v4.pdf, 2023. Accessed: 2024-01-21.

[20] C. developers, Cow protocol, https://docs.cow.fi/, 2021. Accessed: 2023-10-10.

[21] A. Hayden, Z. Noah, S. Moody, K. River, R. Dan, Uniswap v3, https://uniswap.org/whitepaper-v3.pdf, 2021. Accessed: 2024-04-23.