`SocialHelix`: Decentralized Fungible Token Market on the Blockchain

David Grossman

Abstract

This paper proposes a decentralized, self-contained, and scalable fungible token market that allows social media users to invest in content creators. In the `SocialHelix` platform, we implement this market with two associated smart contracts that leverage the Ethereum Virtual Machine to execute Solidity code and record a public ledger of all transactions. The `SocialHelixMarket` contract manages an unbounded set of verified creator tokens as well as a fiat-backed stablecoin pegged to the U.S. dollar. The `SocialHelixOrderBook` contract allows users to place orders to buy or sell creator tokens in exchange for stablecoin. Once an order is verified, its data is emitted on the blockchain to be retrieved by a network of off-chain order matching engines. Through the Helix Matching Protocol, this allows for a decentralized and robust market in which a network of off-chain order matching engines can parallelize the CPU-intensive process of matching orders.

1 Introduction

This paper provides an overview of the `SocialHelix` platform, which is our implementation of a decentralized, self-contained, and scalable fungible token market that allows social media users to invest in content creators. We will focus on five main components of this platform: the `SocialHelixMarket` smart contract, the `SocialHelixOrderBook` smart contract, the network of off-chain order matching engines, the `SocialHelix` website (https://socialhelix.sh), and the Helix Matching Protocol.

1.1 Terminology

Before covering the details of the `SocialHelix` platform's architecture, it is essential to define several relevant terms.

U.S. Dollar (USD)
: The official currency of the United States.

Helix Stablecoin (`$HSC`)
: Native stablecoin in the `SocialHelixMarket` contract, equivalent in value to 1 USD.

Milli-Helix Stablecoin (`$mHSC`)
: Smallest denomination of `$HSC`, equal to 10^{-6} `$HSC`.

`<name>` Token (`$T-<symb>`)
: One unit of the approved creator token with name `<name>` and symbol `<symb>`.

Milli-`<name>` Token (`$mT-<symb>`)
: Smallest denomination of `$T-<symb>`, equal to 10^{-6} `$T-<symb>`.

Order Matching Engine (OME)
: Algorithm that listens to events in the `SocialHelixOrderBook` contract and matches orders for `$HSC`/`$T-<symb>` exchange executions.

Helix Matching Protocol (HMP)
: Protocol that integrates `SocialHelix`'s smart contracts to distribute the CPU-intensive process of matching orders across a network of off-chain OMEs.

Helix Lightning Network (HLN)
: Set of functions in the `SocialHelixMarket` contract that allow users to withdraw, deposit, and transfer money in the form of `$HSC`.

Token Investor (TI)
: User in the `SocialHelixMarket` contract who has provided valid bank account information and authenticated with an external social media account, allowing them to invest in creator tokens on the `SocialHelix` website.

Token Creator (TC)
: TI who has proposed a creator token in the `SocialHelixMarket` contract that was approved by the contract owner.

1.2 `SocialHelix` Principles

Decentralized: `SocialHelix` leverages the Ethereum blockchain to execute Solidity code on a decentralized network of nodes that record a public ledger of all transactions. HMP allows users to exchange creator tokens for `$HSC` through an off-chain OME of their choosing. The `SocialHelixOrderBook` contract must then confirm that both parties consent to the exchange before any assets are transferred on the blockchain. This allows for a decentralized market in which an OME cannot conduct a malicious exchange.

Self-contained: The `SocialHelixMarket` contract is solely responsible for the creation and movement of all assets in the `SocialHelix` market. By condensing all market functionality into a single contract, the gas fee for a decentralized exchange of assets is minimized.

Scalable: The `SocialHelixMarket` contract maps each Ethereum address to a distinct creator token. This allows the market to handle an effectively unbounded set of tokens, which is more than sufficient to support the creator economy of 50+ million creators [1]. As more creator tokens experience a high volume of trade, HMP maintains exchange efficiency by distributing the order matching process across a network of OMEs.

Robust: The `SocialHelixMarket` contract produces a public ledger of all transactions, but malicious actors cannot manipulate a transaction once it has been confirmed by the blockchain. HMP also ensures that an exchange cannot occur unless both parties have sufficient funds to fulfull the agreed-upon terms of their orders.

Lawful: The owner of the `SocialHelixMarket` contract has the exclusive power to blacklist a malicious user, preventing them from transferring or exchanging any assets in the market. This will only be used when necessary to abide by U.S. law against money laundering and other criminal activity.

1.3 Harmony One

We published the `SocialHelixMarket` and `SocialHelixOrderBook` contracts on the Harmony One blockchain, which is a Layer 2 scaling solution for Ethereum. This allows all transactions in the `SocialHelix` market to execute quickly (finality $\approx$ 2 seconds) and cost-efficiently (average gas fee $\approx$ \$0.000001) [2].

2 SocialHelixMarket Contract

Contract Address: 0xaAC740D386395eE2D1f969dfEc48638290258a57

The SocialHelixMarket contract manages a userbase that can participate in a fungible token market of crypto assets. This is accomplished through a combination of six integral components: user addresses, Helix Stablecoin ($HSC), creator tokens, order book compatibility, owner oversight, and transfer fees.

2.1 User Addresses

The SocialHelixMarket contract stores the set of user addresses and the set of blacklisted addresses as public address to bool mappings. These sets are mutually exclusive, which means that an address loses its status as a user when it is blacklisted.

Any non-blacklisted address can become a user by calling the becomeUser function in the SocialHelixMarket contract, which emits the UserCreation event. This allows the SocialHelix platform to maintain a database of user addresses and other relevant information, making the platform compatible with Know Your Customer (KYC) guidelines.

Many of the functions in the SocialHelixMarket contract are inaccessible to blacklisted addresses due to utilization of the *requireUser* modifier. For example, blacklisted addresses cannot transfer or exchange their assets. While they technically can withdraw their stablecoin balance, the SocialHelix platform reserves the right to deny a USD payout to a blacklisted address's associated bank account.

2.2 Helix Stablecoin ($HSC)

The Helix Stablecoin ($HSC) serves as the SocialHelix market's universal currency through which all creator token exchanges are conducted. Each user's balance of $HSC is stored in a public mapping from user address (address) to $mHSC balance (uint256). The total supply of $mHSC is also stored as a separate uint256.

$HSC, a fiat-backed stablecoin, is pegged to the U.S. dollar. We accomplish this by backing each $HSC in the contract's total supply with a real U.S. dollar. This allows a user to withdraw their balance at any time and receive full USD compensation after transfer fees.

Milli-$HSC ($mHSC), equal to 10^{-6} $HSC, is the smallest denomination of $HSC. The SocialHelixMarket contract represents all stablecoin balances in terms of $mHSC. For example, a balance of 1 $HSC would be stored as 10^6 $mHSC. Floating point numbers are avoided in Solidity to prevent rounding errors.

2.2.1 Helix Lightning Network (HLN)

The HLN is a set of functions in the SocialHelixMarket contract that allow users to withdraw, deposit, and transfer money in the form of $HSC. Due to the contract's utilization of the Harmony One blockchain, as described in Section 1.3, each transfer of $HSC executes quickly and cost-efficiently.

The SocialHelix website requires that each token investor (TI) provides valid bank account information. This allows the platform to convert between $HSC and USD for withdrawals and deposits. SocialHelix can only guarantee proper monetary compensation for users who register on the website and provide valid bank account information. As a result, we do not recommend directly calling HLN functions outside of the SocialHelix website.

A TI can purchase $HSC by depositing the equivalent amount in USD (minimum $1) on the SocialHelix website. Once the funds are officially received by the SocialHelix master wallet, the contract owner calls the createStableCoins function in the SocialHelixMarket contract to add the $HSC funds to the TI's balance. This is the only contract function that increases the total $HSC supply. To eliminate concerns of double spending, the SocialHelix platform does not offer refunds for USD to $HSC conversions.

A user can send any amount of their $HSC balance to any other user in the SocialHelixMarket contract with the sendStableCoins function. For each stablecoin transfer, the contract owner receives a fee that is calculated using a publicly available formula. The details of this fee calculation are explained in Section 2.6. Stablecoin transfers do not alter the total $HSC supply in the contract.

A TI can withdraw any amount of their $HSC balance (minimum 1 $HSC) to be compensated with the equivalent amount in USD. To do so, a TI must first call the withdrawStableCoins function in the SocialHelixMarket contract. This immediately removes the withdrawal amount from the TI's balance and emits a StableCoinWithdrawal event with the TI's address as well as the withdrawal amount. The SocialHelix platform is constantly listening for emitted StableCoinWithdrawal events to process each withdrawal, which involves sending the appropriate USD amount to the TI's bank account. This is the only contract function that decreases the total $HSC supply.

2.3 Creator Tokens

A creator token must be proposed by a TI and approved by the contract owner before becoming ownable, transferable, and exchangeable. Thus, the `SocialHelixMarket` contract stores creator tokens in two distinct data structures: the creator token proposal queue and the creator token mapping.

2.3.1 Creator Token Proposal Queue

A TI without an active token proposal or an approved token can propose a token by calling the `makeTokenProposal` function in the `SocialHelixMarket` contract. For each proposal, the contract owner receives a fee that is calculated using a publicly available formula. The details of this fee calculation are explained in Section 2.6.

Each proposal is stored as a `CreatorTokenProposal`, which is a struct that contains the creator (`address`), name (`bytes32`), and symbol (`bytes32`) of the proposal. When a TI makes a token proposal, a `CreatorTokenProposal` instance is initialized and enqueued to the creator token proposal queue. The fields of this `CreatorTokenProposal` are then emitted publicly on the blockchain through the `NewCreatorTokenProposal` event. The taken names and symbols of existing tokens are stored in mappings from `bytes32` to `bool` to prevent duplicates. Because Solidity does not offer a built-in queue data structure, a mapping from index (`uint256`) to proposal (`CreatorTokenProposal`) is used as a substitute, in conjunction with `uint256` variables for the head and tail indices.

The contract owner has the exclusive power to process a proposal, which means either approving or denying it. Thus, the contract owner must be constantly listening for emitted `NewCreatorTokenProposal` events to maintain a log of all proposal information. As a result of the first-in, first-out structure of a queue, only the oldest active token proposal can be processed at any given time.

When the contract owner is ready to process a token in the queue, they first call the `getNextTokenProposalCreator` function to get the address of the proposal creator at the head of the queue. They can then refer to their event log to find the name and symbol of the next proposal that can be processed.

Considering the name and symbol of the proposed token as well as other associated information stored by the `SocialHelix` platform, the contract owner must decide whether to approve or deny the token proposal. For the token to be approved, the name must match `[A-Za-z0-9]+` and the symbol must match `[A-Z0-9]+`. To process the proposal, the contract owner calls the

`processNextTokenProposal` function, which takes in a `bool` argument that represents their approval decision (`True` = approve, `False` = deny).

Regardless of the approval decision, the `CreatorTokenProposal` is dequeued from the creator token proposal queue, allowing the next proposal to be processed. If the token is approved, the name and symbol from the `CreatorTokenProposal` are copied into a `CreatorToken` struct, which is explained in greater depth in Section 2.3.2. If the token is denied, the name and symbol of the proposal are deleted from the respective mappings for taken names and symbols.

2.3.2 Creator Token Mapping

A TI officially becomes a token creator (TC) when their creator token proposal is approved by the contract owner. This change is reflected on both the `SocialHelixMarket` contract and the `SocialHelix` website, but we will focus on the `SocialHelixMarket` contract in this section.

Approved creator tokens are stored in a public mapping from the TC's address (`address`) to their approved creator token (`CreatorToken`), which prevents a TC from creating multiple tokens. The `CreatorToken` struct contains the name and symbol of the `CreatorTokenProposal` as well as a mapping from user address (`address`) to `$mT-<symb>` balance (`uint256`). Because the creator address is used as the key of the creator token mapping, including it in the `CreatorToken` struct would be redundant.

All approved creator tokens are fixed-cap assets with a constant total supply of 10^6. The `SocialHelixMarket` contract represents token balances as integer multiples of 10^{-6}, which is the smallest possible denomination of a creator token. Thus, the contract stores this total supply as 10^{12}.

To conceptualize the creator token mapping, consider the fictitious example of an aspiring token creator named Alice (address: 0x71...) who made the first token proposal (name: Aliceum, symbol: ALI) in the contract. As soon as the contract owner approves this token proposal, the token balance mapping of her `CreatorToken` is initialized such that Alice's address maps to the total supply of Aliceum Token (`$T-ALI`). We denote the smallest denomination of Aliceum Token as `$mT-ALI`, which is equivalent to 10^{-6} `$T-ALI`. The structure of the creator token mapping immediately after the approval of Aliceum Token is visualized in Figure 1.

The `SocialHelixMarket` contract provides functionality similar to the HLN for transferring creator tokens. A user may send any portion of their balance of a creator token to any other user with the `sendCreatorTokens` function. The contract owner receives a publicly-known `$HSC` fee for each

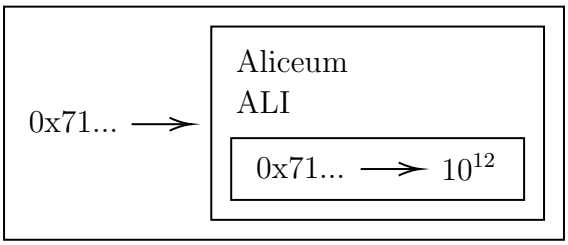

Figure 1: Creator token mapping after approval of first token.

transfer of creator tokens, the details of which are described in Section 2.6. Since each creator token is a fixed-cap asset, a transfer can never change the total supply of a certain token.

Let's now add a TC named Bob (address: 0xc2..., token: BobCoin) and a TI named Charlie (address: 0x49...) to our fictitious example from above. As long as they have sufficient `$HSC` balances to cover the transfer fees, Alice, Bob, and Charlie can call the `sendCreatorTokens` function to send tokens amongst each other. It is important to note the distinction between a token owner and creator—even if a TC transfers the entirety of their token's supply to other addresses, the TC never loses their status as the token's creator. Figure 2 shows a possible configuration of the creator token mapping after several token transfers.

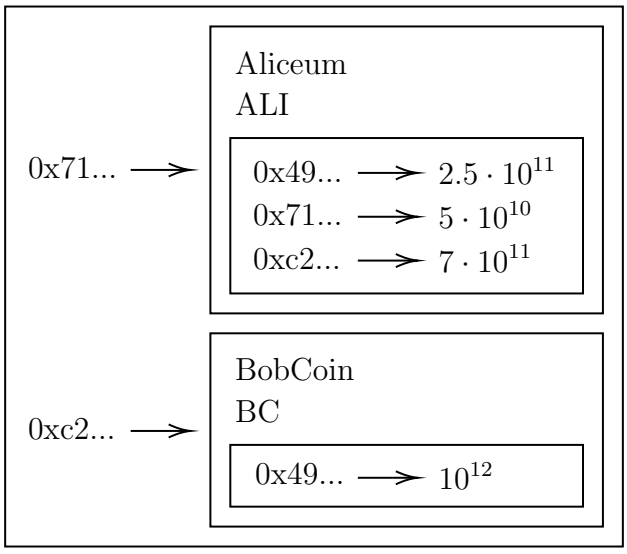

Figure 2: Creator token mapping with multiple tokens after several transfers.

Note: Figure 1 and Figure 2 use the ASCII representations of all `bytes32` variables for ease of understanding.

2.4 Order Book Compatibility

The `SocialHelixMarket` and `SocialHelixOrderBook` contracts store each other's contract addresses. The `SocialHelixMarket` contract also implements the `OrderBookCompatible` interface, which allows the `SocialHelixOrderBook` contract to call the `SocialHelixMarket` contract's order-related functions: `verifyOrder` and `executeOrderExchange`. To prevent malicious exchanges without sacrificing decentralization, the `SocialHelixMarket` contract utilizes the *requireOrderBook* modifier on these two functions. As specified in Section 6, this is the basis for all `$HSC/$T-<symb>` exchanges executed via HMP.

2.4.1 Order Verification

The `SocialHelixMarket` contract's `verifyOrder` function ensures that the attributes of a given order are valid. This is accomplished by a series of Solidity require statements for the following mandatory conditions:

1. The party (i.e., the address that submitted the order) must be a user.
2. The creator token must exist in the creator token mapping.
3. The amount of requested tokens must be positive and no greater than the total supply of each creator token (10^{12} `$mT-<symb>`).
4. The exchange address must either be a user or the null address (0x00...).
5. If the order is a limit buy, then the party must have enough `$HSC` to purchase the tokens at the specified limit price.
6. If the order is a sell, then the party must have at least as much of the creator token as they proposed to sell.

Refer to Section 3.1.1 for precise specifications of the attributes of an order.

If any of these conditions are not satisfied, the entire transaction that called the `verifyOrder` function is reverted (i.e., all state changes are undone). Thus, since the `submitOrder` function in the `SocialHelixOrderBook` contract calls the `verifyOrder` function, it only saves valid order submissions.

2.4.2 Order Exchange Execution

When the SocialHelixOrderBook deems that a pair of orders is compatible for a $HSC/$T-<symb> exchange, it must call the executeOrderExchange function in the SocialHelixMarket contract to execute the actual asset tranfer. This function accepts the buyer (address), seller (address), creator (address), exchange (address), token amount ($mT-<symb>), and token price ($mHSC per $T-<symb>) for the exchange. Based on this information, it calls the internal transferCreatorTokens and transferStableCoins functions to transfer the appropriate amount of assets between the buyer and seller. Each of these internal functions will revert if one of the parties lacks sufficient funds to fulfill the agreed-upon terms of their order, which will then cause the overall executeOrderExchange function to revert. Reversion of a Solidity function undoes all state changes, so this ensures that all exchanges are executed on an all-or-nothing basis—either both orders are fulfilled or the exchange is canceled.

For each exchange, the buyer is required to pay $HSC fees to the TC and the OME to reward them for their work. Thus, a buyer must have enough $HSC to cover both of these fees in addition to the price of the tokens. Standard fees for transferring $HSC and $T-<symb> also apply. Refer to Section 2.6 for details about the calculations of all transfer and exchange fees.

2.5 Owner Oversight

The SocialHelix market strives to be as decentralized as possible without infringing upon U.S. law. The contract owner cannot seize assets or manipulate exchanges in the SocialHelixMarket contract—the most they can do is blacklist an address from engaging in future transfers. This blacklisting power will only be used when there exists substantial evidence that an address is involved in criminal activity. Nonetheless, if a user believes that they have been unfairly blacklisted, they may submit an appeal through the SocialHelix website.

To become a TI, a user must enter valid bank account information on the SocialHelix website. This provides the SocialHelix platform with sufficient user information to follow Know Your Customer (KYC) guidelines.

The contract owner is solely responsible for approving or denying each creator token proposal. While the SocialHelix platform aims to be as inclusive of different TCs as possible, this oversight is necessary to prohibit tokens that promote dangerous ideologies.

2.6 Transfer Fees

The `SocialHelixMarket` contract implements several `$HSC` fees to reward OMEs, TCs, and the contract owner for each of their contributions to the `SocialHelix` platform. Because fees are always transferred between addresses in the stablecoin balance mapping, they never change the total `$HSC` balance.

2.6.1 Fee Constants and Variables

The following table lists all of the constants and variables that are directly used in fee calculations:

Name	Value/Range
_sendStableCoinFixedFee	$[0, 10^6]$
_sendStableCoinPortionFee	$[0, 5{\cdot}10^4]$
_sendTokenFixedFee	$[0, 10^6]$
_tokenProposalFee	$[0, 10^9]$
BUY_TOKEN_CREATOR_PORTION_FEE	10^4
_buyTokenExchangePortionFee	$[5{\cdot}10^3, 3{\cdot}10^4]$

These variables can only be mutated within a reasonable, predefined range, enabling a proper balance between scalability and decentralization.

Each constant/variable represents either a fixed fee or a portion fee, signified by the final two words of the variable name. A fixed fee is a set amount of `$mHSC` that does not depend on the numeric parameters of the associated function. On the other hand, a portion fee is directly proportional to the amount of `$mHSC` that is transferred. To be exact, for a function with portion fee value p that transfers x `$mHSC`, the fee is $\lfloor 10^{-6}px \rfloor + 1$ `$mHSC`. It should also be noted that variables are in camel case with a leading underscore, whereas constants are in capitalized snake case.

2.6.2 Fee Calculations

The following is a comprehensive list of actions in the `SocialHelix` market and their associated fee calculations:

Send `$HSC`

When a user sends x `$mHSC` to another user on the HLN, they must additionally pay the following `$mHSC` fee to the contract owner:

$$\lfloor 10^{-6} \cdot \text{\_sendStableCoinPortionFee} \cdot x \rfloor + 1 + \text{\_sendStableCoinFixedFee}$$

Send `$T-<symb>`

When a user sends `$T-<symb>` to another user, they must pay a fixed fee of _sendTokenFixedFee `$mHSC` to the contract owner, regardless of the `$T-<symb>` amount.

Make a Token Proposal

When a user makes a token proposal, they must pay a fixed fee of _tokenProposalFee `$mHSC` to the contract owner.

Buy `$T-<symb>` with `$HSC`

If a user places a valid order to buy `$T-<symb>` with x `$HSC`, they must pay several fees when the exchange is executed. In addition to the standard fee to send x `$HSC` on the HLN, they must pay the following `$mHSC` fee to the OME:

$$\lfloor 10^{-6} \cdot \text{\_buyTokenExchangePortionFee} \cdot x \rfloor + 1$$

They must also pay the following `$mHSC` fee to the TC of the token being exchanged:

$$\lfloor 10^{-6} \cdot \text{BUY\_TOKEN\_CREATOR\_PORTION\_FEE} \cdot x \rfloor + 1$$

Sell `$T-<symb>` for `$HSC`

If a user places a valid order to sell `$T-<symb>` for `$HSC`, they must pay the standard fixed fee to send `$T-<symb>`.

3 `SocialHelixOrderBook` Contract

Contract Address: 0x9c9B294A770bc1f027D5b57b62bAd6B93fe570CF

The `SocialHelixOrderBook` receives, stores, and processes all orders to exchange assets in the `SocialHelix` market. This is achieved through the integration of two major features: order data and order match submission.

3.1 Order Data

An order is a request to buy or sell a certain amount of `$T-<symb>` in exchange for `$HSC`. The `SocialHelixOrderBook` contract maintains a database of all active orders in the `SocialHelix` market by implementing three main functionalities: order storage, order submission, and order removal.

3.1.1 Order Storage

In the `SocialHelixOrderBook` contract, the relevant data of each order is represented as an instance of the `Order` struct, which has the following fields:

party (`address`)
: The address of the user who submitted the order.

creator (`address`)
: The address of the TC of the token that the party wants to buy or sell.

exchange (`address`)
: The address of the OME that the party trusts to execute their order exchange. If the null address, then the party trusts any OME.

buying (`bool`)
: True if the party wants to buy `$T-<symb>` with `$HSC`. False if the party wants to sell `$T-<symb>` for `$HSC`.

limit (`bool`)
: True for a limit order. False for a market order.

limitPrice (`uint256`)
: The limit price (in `$mHSC` per `$T-<symb>`) of the order, which represents the maximum price to be paid (for a limit buy order) or the minimum price to be received (for a limit sell order) [3]. This value is ignored for a market order.

tokenAmount (`uint256`)
: The amount of tokens (in `$mT-<symb>`) that the party wants to buy or sell.

Using this struct, the `SocialHelixOrderBook` contract stores all active orders in a public mapping from order ID (`uint256`) to order data (`Order`). The order ID is an arbitrary value that serves as the unique identifier of an order. In our implementation of the `SocialHelixOrderBook` contract, we start with an order ID of 0 and increment this value by 1 for each successive order. Since the order ID of each order submission is emitted publicly on the blockchain, it would be futile make order IDs cryptographically secure. If a limit order specifies a trusted OME, it is impossible for a malicious user to manipulate the order in any way. However, market orders can be extremely dangerous if submitted to a malicious OME, as there is no constraint on the exchange price that the OME can choose.

3.1.2 Order Submission

A user in the `SocialHelixMarket` contract can submit an order by calling the `submitOrder` function in the `SocialHelixOrderBook` contract. This function accepts all of the fields of the `Order` struct, except for party, as paremeters. The address that submits an order is always the order's party, so party is set to the Solidity global variable for the sender address of the function (msg.sender). To verify the order and revert the entire transaction if the order is invalid, the `submitOrder` function calls the `verifyOrder` function in the `SocialHelixMarket` contract. See Section 2.4.1 for a comprehensive list of the conditions that must be satisfied for a valid order.

If the order is valid, the order data is saved in the `SocialHelixOrderBook`'s mapping from order ID (`uint256`) to order data (`Order`), with the order ID set to 1 more than that of the most recently submitted order. To notify the network of off-chain OMEs about the new valid order, the `verifyOrder` function emits an `OrderSubmission` event with the order ID and exchange address of the order. For each `OrderSubmission` event, an off-chain OME can use the order ID to fetch the rest of the order data from the public mapping. The exchange address is nonetheless included in the event so that each OME can easily filter out orders meant for other OMEs, but it would be far too costly to emit all of the order data in each event. Refer to Section 4 for details on how each off-chain OMEs handles the stream of `OrderSubmission` events.

3.1.3 Order Removal

An active order in the `SocialHelixOrderBook` contract can be deleted by calling the `removeOrder` function with the corresponding order ID. This emits the `OrderRemoval` event with the order ID of the removed order, signaling to the network of OMEs that the order should no longer be considered. Although all order data is public, only the party or exchange address of an order has the power to call the `removeOrder` function on the order.

Due to rapid price fluctuations and asset transfers, an active order that was originally valid may become unfulfillable before an OME has time to process it. If the `executeOrderExchange` function receives two seemingly valid orders and catches a reversion when exchanging their assets, the order with the party who could not fulfill the order is immediately deleted. This type of order removal also emits the `OrderRemoval` event, allowing the `SocialHelix` website to alert the party that their order is no longer being considered.

3.2 Order Match Submission

When an OME matches a pair of orders, it must submit the match by calling the `submitOrderMatch` function in the `SocialHelixOrderBook` contract. Specifically, the OME must submit the buy order ID (`uint256`), sell order ID (`uint256`), token amount (`$mT-<symb>`), and token price (`$mHSC` per `$T-<symb>`) for the exchange. If the orders corresponding to the buy and sell order IDs are valid, the `submitOrderMatch` function requires that the following conditions are satisfied:

1. The buy and sell order IDs correspond to buy and sell orders, respectively.
2. The buy and sell orders each have a token amount at least as great as the token amount submitted by the OME.
3. The buy and sell orders trust the OME that called the `submitOrderMatch` function.
4. The buy and sell orders are exchanging the same creator token.
5. If the buy order is a limit order, then its limit price is no greater than the token price submitted by the OME.
6. If the sell order is a limit order, then its limit price is no less than the token price submitted by the OME.

The `submitOrderMatch` function reverts if any of these conditions fail. This should never happen to order matches submitted by an honest OME, but the requirements are necessary to prevent a malicious OME from exploiting an order. If the conditions all succeed, the `submitOrderMatch` function calls the `executeOrderExchange` function in the `SocialHelixMarket` contract, which is detailed in Section 2.4.2

Even if an honest OME is functioning perfectly, insufficient party balances and/or non-instant blockchain finality may cause the exchange of assets for a pair of matched orders to fail. To help OMEs handle this uncertainty, the `submitOrderMatch` function returns a status code (`uint256`) describing the result of the attempted asset exchange. Status code 0 is the only code that signifies a successful asset exchange. For all other status codes, no state changes are persisted in the `SocialHelixMarket` contract. The possible status codes are as follows:

0 Success
: The order exchange execution succeeded. Assets were transferred properly between the parties, the `ExchangeExecution` was emitted with the exchange data, and completely fulfilled orders were removed.

1 Creator Token Transfer Reversion
: The order exchange execution failed because the creator token transfer reverted. This usually occurs when the seller lacks the necessary amount of creator tokens to fulfill their order.

2 Helix Stablecoin Transfer Reversion
: The order exchange execution failed because the `$HSC` transfer reverted. This usually occurs when the buyer lacks the necessary amount of `$HSC` to fulfill their order.

3 Internal Error
: The order exchange execution failed due to an uncaught internal error in the integration of the `SocialHelixMarket` and `SocialHelixOrderBook` contracts.

4 Inactive Buy Order
: The inputted buy order ID does not correspond to an active buy order. This either means that the order never existed or that it was removed.

5 Inactive Sell Order
: The inputted sell order ID does not correspond to an active sell order. This either means that the order never existed or that it was removed.

6 Inactive Orders
: Neither of the inputted order ids correspond to active orders. For each order, this either means that the order never existed or that it was removed.

For status codes 1 and 2, the order with the party who lacked the necessary funds is automatically removed. This prevents an unfulfillable order from repeatedly failing and slowing down the matching algorithm.

4 Network of Off-Chain OMEs

The `SocialHelix` platform's network of off-chain OMEs parallelizes the CPU-intensive process of matching orders, which is essential to the scalability and

decentralization of HMP. Rather than requiring a centralized OME, HMP allows any user-trusted OME to match orders. Different OMEs may focus on distinct subsets of creator tokens, allowing them to function extremely efficiently. The buy-token-exchange portion fee, as described in Section 2.6, motivates OMEs to submit as many correct order matches as possible. Furthermore, HMP is flexible to a variety of OME implementations, as long as each implementation contains three integral components: Web3 integration, a matching algorithm, and error handling.

4.1 Web3 Integration

To successfully match orders, an off-chain OME must call functions and listen to events in the `SocialHelixOrderBook` contract. The `submitOrder` and `executeOrderExchange` functions are used to submit order data, whereas the `OrderSubmission`, `OrderRemoval`, and `ExchangeExecution` events are used to receive order data. All of this interaction requires integration with a Web3 provider.

For our implementation, we employ a custom JavaScript API that leverages Web3.js and the Truffle HDWalletProvider. Users, creators, OMEs, and the owner are each initialized with a distinct BIP39 mnemonic passphrase, which converts to a unique Ethereum address. In addition to contract functionality, the API also allows users to view and transfer their Harmony One balance.

4.2 Matching Algorithm

To integrate with the `SocialHelixOrderBook` contract via HMP, the matching algorithm for each OME must accept a stream of orders and return pairs of matched orders. Our implementation of this algorithm utilizes the skip list data structure, which allows for $\log(n)$ search, insertion, and deletion. Buy and sell orders are sorted by price level in two distinct skip lists (buy skip list and sell skip list) with queues at each price level. When an order is received, the algorithm adds it to the appropriate queue in the appropriate skip list, as determined by the price and type of the order.

The algorithm repeatedly matches a buy order from the lowest price level with a sell order from the highest price level, allowing for optimal matches for both parties. Because not all orders request the same amount of tokens, completely fulfilling a given order may require matches with multiple other orders. As long as there are at least two orders with compatible limit prices, order matching will continue indefinitely.

4.3 Error Handling

By default, our OME implementation assumes that all of its matches are successful, and it removes an order from the skip list data structure if it thinks that the order has been fulfilled. However, unexpected asset exchange failures will cause a portion of orders to be left unfullfilled.

As described in Section 3.2, the `submitOrderMatch` function returns a status code (`uint256`) for each order match submitted by an OME. For each of its order match submissions, our OME asynchronously handles this status code while running the matching algorithm. Upon receiving a status code of 1 or 5, our OME fetches updated order data for the buy order and adds it back into the buy skip list. Likewise, upon receiving a status code of 2 or 4, our OME fetches updated order data for the sell order and adds it back into the sell skip list. This allows our OME to fulfill 100% of its valid orders.

5 `SocialHelix` Website

The `SocialHelix` website (https://socialhelix.sh) provides a user-friendly GUI to interact with the `SocialHelix` platform, which bridges Web2 and Web3 functionalities in an easily accessible format. To prioritize the security and safety of the `SocialHelix` userbase, we focus on optimizing three major functionalities of the website: user authenification, data storage, and user protection.

5.1 User Authentication

To register an account on the `SocialHelix` website, a user is required to provide their full name along with an email and password. This email must be verified, allowing for optional two-factor authentication with email.

The `SocialHelix` website requires two additional verification steps for a user to become a TI:

1. A user must prove ownership of at least one social media account on a supported, external social media platform. One of these accounts must serve as the user's primary account, which determines the public display name, username, and profile picture of that user. Proof of ownership of this primary account can also be used for two-factor authentication. All external social media authentication on the `SocialHelix` website is accomplished with a custom API that leverages cross-site OAuth technologies (e.g., "Login with Facebook").

2. A user must provide valid bank account information, which is needed for all USD payments and payouts. This additional user information also helps us follow Know Your Customer (KYC) guidelines.

To ensure that the `SocialHelix` website abides by U.S. law, we reserve the right to revoke the privileges of a malicious TI.

All users on the `SocialHelix` website are identified and verified by their social media identities, so it is impossible to impersonate a creator without controlling one of their social media accounts. In the event that a user's primary account gets hacked, the user can disconnect the compromised account and switch to a primary account on a different platform.

To improve the experience for frequent users of the `SocialHelix` website, we utilize an expiring JSON Web Token (JWT) to automatically authenticate a user who has recently logged in. A JWT provides a compact method for securely representing claims between two parties a JSON object [4]. In our case, the two parties are the client and the server.

5.2 Data Storage

Web3 storage sacrifices some of the efficiency and scalability of traditional Web2 storage for improved verifiability and decentralization. Rather than compromising, the `SocialHelix` backend leverages both of these options for their optimal applications.

To store and access the vast majority of user and token data, we utilize a standard MongoDB database. Sensitive data, such as passwords and BIP39 mnemonic passphrases, are kept secure with SHA256 encryption. Our custom Web3.js API, as described in Section 4.1, is only employed when decentralization and correctness of market data are absolutely necessary.

5.3 User Protection

Many of the functions in the two smart contracts can be extremely dangerous if used incorrectly. We intend to make `SocialHelix` accessible to users without any understanding of the blockchain, so it is essential that the website provides an extra layer of protection between the smart contracts and the `SocialHelix` userbase. The following are some of the most notable examples of this user protection:

- A malicious OME could exploit the conditions of a market order to steal the entire `$HSC` balance of the order's party. To prevent this, all orders

placed on the SocialHelix website are limit orders by default. If a user insists on placing a market order, they must read and accept a warning about the possible dangers of their decision.

- All asset transfers in the SocialHelixMarket contract are irreversible. To prevent users from permanently losing their assets, the SocialHelix website displays a confirmation dialogue before calling any contract functions that could transfer assets.

- A user must have a certain balance of Harmony One to call a state-modifying contract function. Additionally, a user must be able to cover the send-token fixed fee, as descibed in Section 2.6, to exchange their creator tokens for $HSC. Therefore, it is dangerous to have a near-zero balance of Harmony One or $HSC. To mitigate this danger, the SocialHelix website prevents users from calling contract functions that would cause them to fall below a minimum balance threshold.

6 Helix Matching Protocol

The Helix Matching Protocol (HMP) integrates SocialHelix's smart contracts to distribute order matching across a network of off-chain OMEs. As mentioned in Section 4.2, order matching is a CPU-intensive process, so it would be inefficient to perform on-chain. However, the smart contracts cannot directly impose any restrictions on the logic that each off-chain OME uses to match orders, making it difficult to guarantee that all order exchange executions are valid. HMP provides a unique solution to this problem, prioritizing both robustness and decentralization.

6.1 Architecture

HMP depends on constant interaction between the SocialHelixMarket contract, the SocialHelixOrderBook contract, the SocialHelix userbase, and the network of off-chain OMEs. Figure 3 provides an overview of how these components communicate about order data to execute order exchanges.

The process begins with the SocialHelix userbase, which submits both buy and sell orders for a variety of creator tokens. When each of these orders is received and verified by the SocialHelixOrderBook contract, the corresponding order data is emitted on the blockchain. Each OME in the network of off-chain OMEs continuously listens for these order events to find all orders

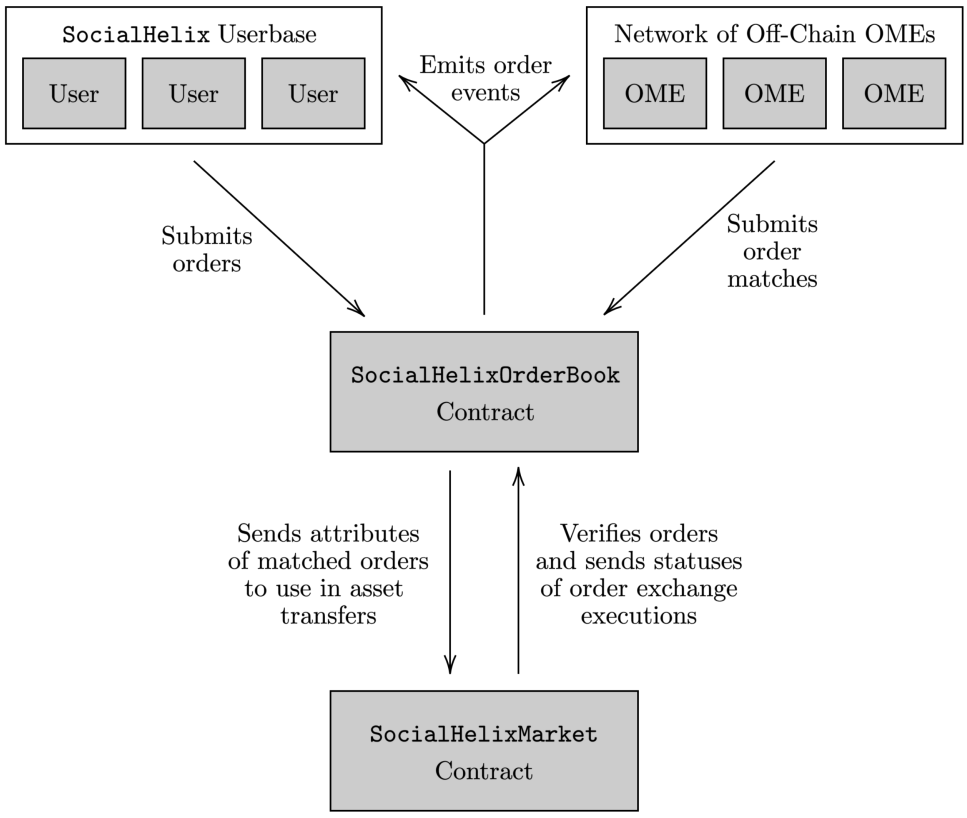

Figure 3: Architecture diagram of the Helix Matching Protocol.

that trust the OME. With this order data, each OME implements an algorithm to match compatible orders. Section 4.2 describes an example of such an algorithm, but HMP does not restrict OMEs to a specific implementation.

To execute the exchange of an order match, an OME must submit the order match to the `SocialHelixOrderBook` contract with a token price and amount. Since the `SocialHelixOrderBook` contract saves all of the original order data, it can perform extensive checks to ensure that both parties agree to the terms of the exchange. Thus, even though the order matching process is not conducted on-chain, a malicious OME cannot exploit an order with proper constraints.

For each received order match, the `SocialHelixOrderBook` contract relays the attributes of the matched orders to the `SocialHelixMarket` contract. This gives the `SocialHelixMarket` contract the necessary information to transfer assets for the exchange execution. The `SocialHelixMarket` con-

tract then informs the `SocialHelixOrderBook` contract of the status of this execution. Based on this status, the `SocialHelixOrderBook` may emit the `ExchangeExecution` event or remove an order as appropriate. By emitting events for all significant state changes, the `SocialHelixOrderBook` contract is able to quickly notify users and OMEs of any updates to their orders.

6.2 Versatility

HMP allows for a decentralized, self-contained, scalable, robust, and lawful exchange, perfectly coinciding with the `SocialHelix` principles. While we present a proof of concept specifically for the `SocialHelix` market, HMP is readily adaptable to any self-contained system of exchangeable assets on the Ethereum blockchain.

References

[1] Yuan, SignalFire's Creator Economy Market Map
`https://signalfire.com/blog/creator-economy`

[2] Harmony ONE, Transactions
`https://docs.harmony.one/home/general/technology/transactions`

[3] Charles Schwab, 3 Order Types: Market, Limit and Stop Orders
`https://www.schwab.com/resource-center/insights/content/3-order-types-market-limit-and-stop-orders`

[4] JWT, Introduction to JSON Web Tokens
`https://jwt.io/introduction`