Morphswap: A cross-chain automated market-making protocol

Anonymous

The cryptocurrency ecosystem has, ever since mid-2011 with the launches of Namecoin and IXcoin, had a desire for the ability to trustlessly switch assets running on separate blockchains. The ecosystem born from a trustless asset exchange protocol, with a focus on the importance of decentralization, ironically always depended on a trusted, centralized party to exchange assets. Ethereum, when launched, solved the problem of being able to exchange two assets without a middle-man, with early attempts such as EtherDelta and IDEX being the first to market, before Uniswap revolutionized asset swapping. Yet it is in this era, ushered in by Ethereum, in which the desire for trustless, cross-chain asset exchange, would be one of the largest unsolved problems facing the cryptocurrency ecosystem.

Due to advances in the blockchain industry, oracle technology, and automated market making, the solution for this problem is now possible, in the form of a specifically engineered multi-chain protocol.

The concept behind the protocol is simple - a Uniswapv2-style AMM spanning multiple chains. The execution of this concept is far more complex.

While one might think that the natural approach would be to create a seperate blockchain with the ability to interact with supported chains, the nodes of the resulting blockchain would be too empowered to corrupt the protocol with malicious actions resulting in irreversible consequences on supported chains. On a traditional blockchain, maliciously inserted data can later be undone. When such a blockchain is interacting with other blockchains, when a block causing an action on another chain gets added, it doesn’t matter if that block later gets rewritten by a good actor - the other chain can’t be rewritten.

The only way to effectively implement this concept - to split an AMM between multiple chains - is to utilize the current standard method for trustlessly putting off-chain data on-chain. Using an oracle protocol, specifically a network of Chainlink Operator nodes, any smart contract can trustlessly acquire off-chain data. So, by utilizing such an oracle network, if the state of other blockchains can be trustlessly curated, it can be treated as simple off-chain data and put on-chain.

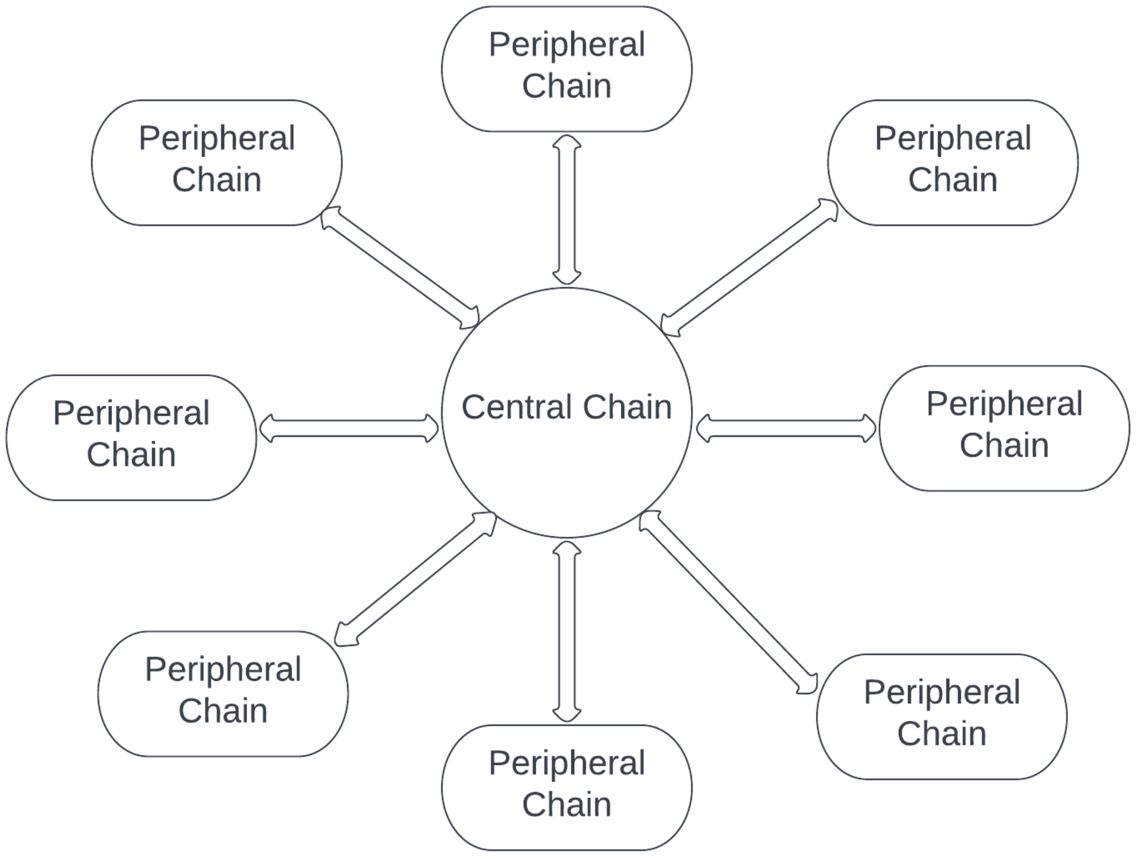
For this method to be used, the data the oracle network is querying needs to be curated and supplied trustlessly, in a way preserving of decentralization. Using a single simple RPC node is not sufficient, as it is still centralized, having a single point-of-failure. However, using a myriad of JSON-RPC APIs, cycling through chain-specific arrays of RPCs in a 'tortoise-and-hare' algorithm and comparing outputs from each of the pair of queried nodes, accepting only data from query sets passing an equality check, foreign-chain data can be curated and served to a protocol in a way that is fully trustless, maintaining full decentralization.

With a sufficient one-way data-transfer standard, the protocol's architecture needs to meet the criterion of allowing multiple contract deployments on multiple otherwise-independent chains to fully reliably use this standard to communicate bi-directionally, while achieving low time-to-finality. The 'naive approach' would be to have singular, equal deployments on each chain, each requesting data from other chains via the JSON-RPC mesh. However, this approach results in a protocol with unimpressive liquidity efficiency and a high barrier to entry for decoding data paths - the significance of which will be explained later on in this paper.

The most capital-efficient node structure for this protocol is to have a single central node and many peripheral nodes. Specifically, one chain's Morphswap node will be specifically engineered to operate as a central node, and all other chains that the protocol supports will have a typical Morphswap node deployed on it, which will all act as peripheral nodes. The central node can communicate with every peripheral node, while a peripheral node can only communicate with the central node.

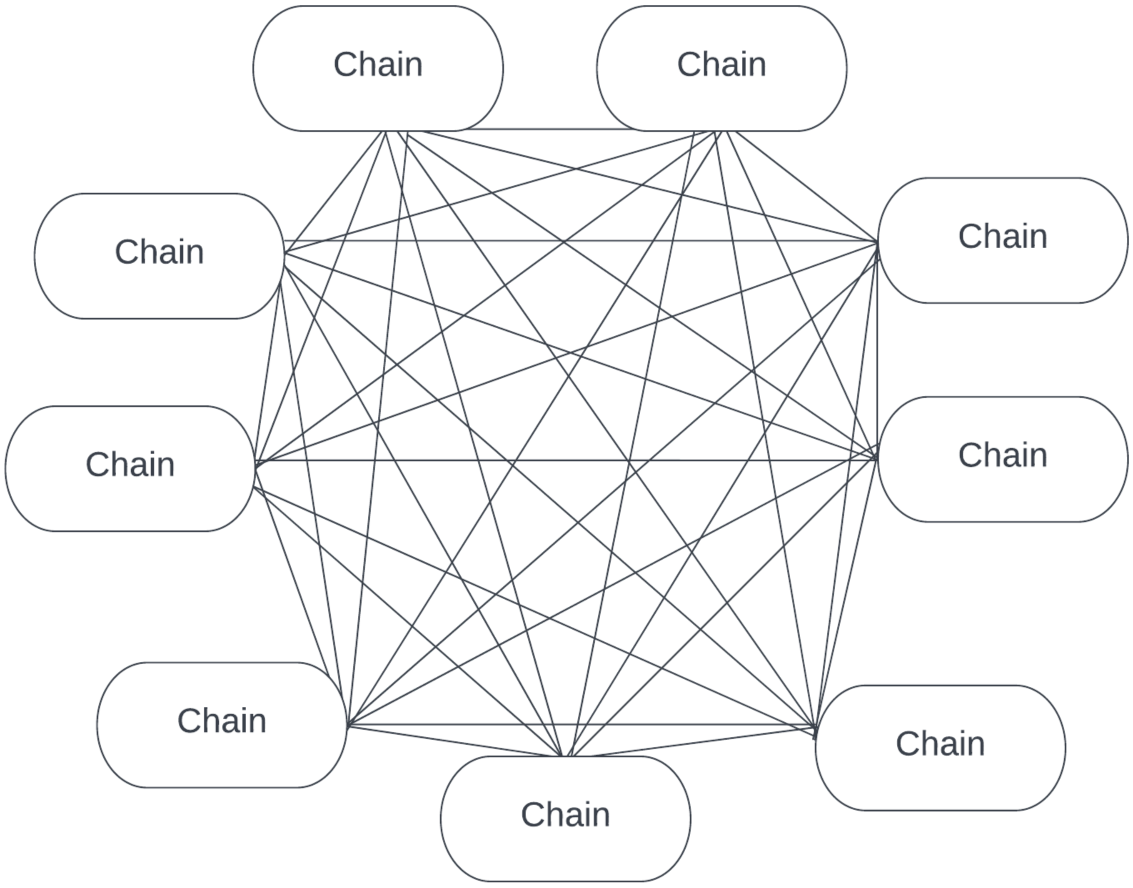
This structure has a drastically reduced amount of node communication channels, and increased capital efficiency. The naive approach would yield a complexity formula of $n \times (n-1)$, with the latter approach yielding a complexity of[1] $n - 1$. The capital efficiency follows the same pattern, with the number of required pools for the latter approach being $\frac{1}{n}$ as much as the number of pools required for the naive approach.

This structure yields the following architectural layout:



As opposed to the naive approach containing an exponentially increasing number of connections and decreasing liquidity efficiency:

[1] 2(n-1) connections



For the internal design of the individual nodes, both peripheral and central, they need certain constituent segments. Namely, an individual node is composed of a smart contract deployed on the node’s respective chain. Additionally, the mainnet 26KB size limit on deployed contracts call for additional complexity, as the contract architecture needs to be designed with these constraints in mind. The entire contract, compiled with the optimization flag set to just a single run, still yields a contract with over 100KB of size. The system design philosophy that yields us with a workable result is to split up the contract into many contracts that all interact with each other. Specifically, one master contract and many delegate contracts - the master contract being the contract where all the storage is allocated and state-modification is done, and the delegate contracts simply acting as code repositories hosting code for the master contract to run. When a user interacts with the master contract by calling a function, the master contract calls the delegate contract hosting the code specific to the called function, namely making a `delegatecall` encoded with the identical data (including the function signature) to the appropriate contract. The master contract additionally, in the same manner as Uniswap, deploys a new "AssetPool" contract for each asset pair involving assets on its chain. Each asset pool has its own ERC-20 compliant token, representing liquidity provided to that side of the pool. The additional functionality outside of the standard ERC-20 functions can only be accessed by the contract that deployed it (the master contract). Finally, as there is the capacity for a myriad of pools, each node requires a module responsible for routing actions to the correct pools.

The automated market making on Morphswap applies the same formula for determination of swap outputs as a traditional AMM. The only difference being that the calculation is done on the domestic chain, with the destination chain protocol node simply being a servant of the decisions made. The equation for determining pool balance and token output from a swap are the standard for traditional AMMs. Pool balance is perceived by the protocol nodes through this equation:

$$\frac{x_\Delta}{x_\Delta + x_0} = \frac{y_\Delta}{y_0}$$

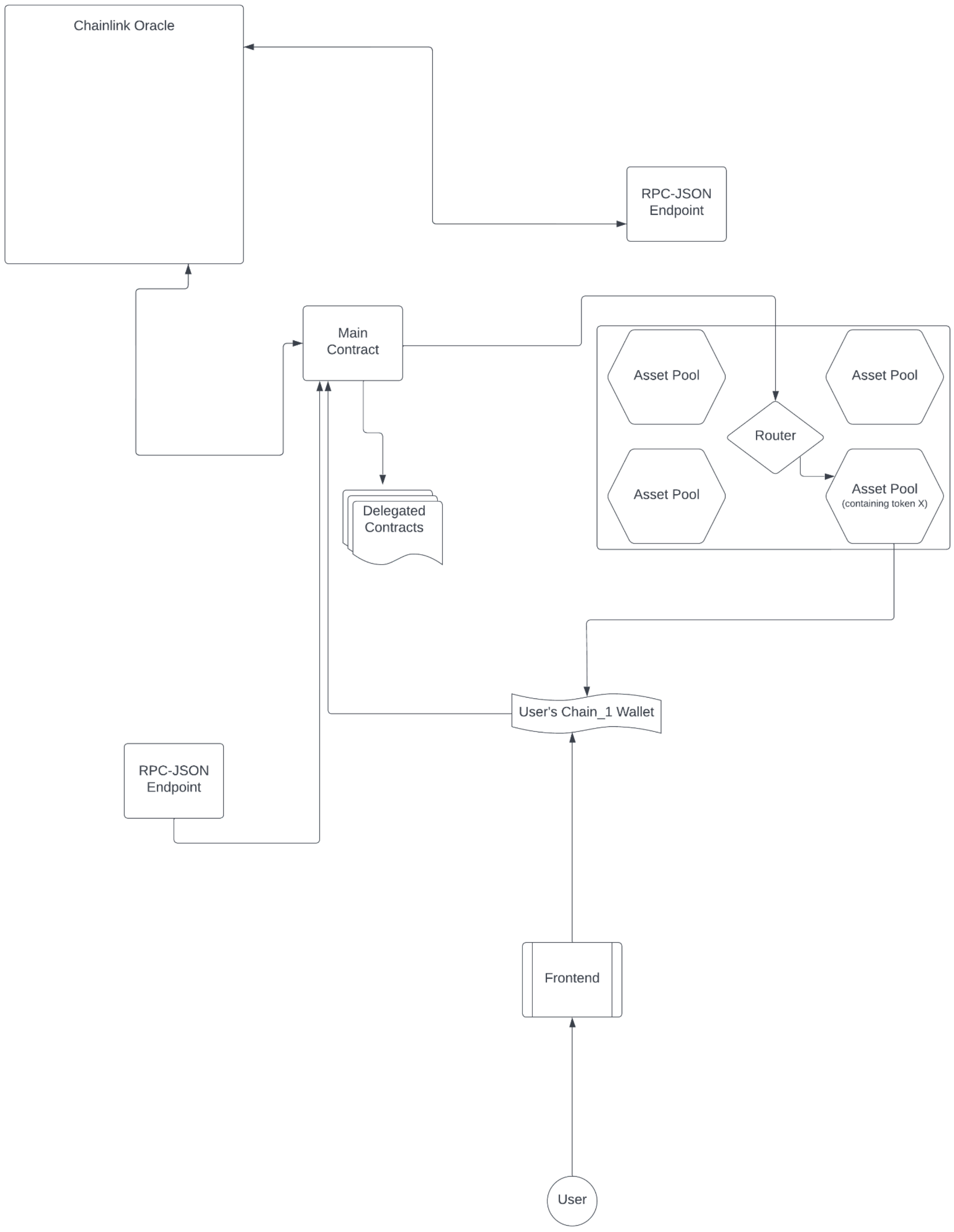
x_Δ = the nominal sent amount of token or coin x
x_0 = the total amount of token or coin x in the pool as liquidity at the start of the block
y_Δ = the nominal amount of token or coin y to be sent to user from the pool
y_0 = the total amount of token or coin x in the pool as liquidity at the start of the block

So, in terms of the output, we get:

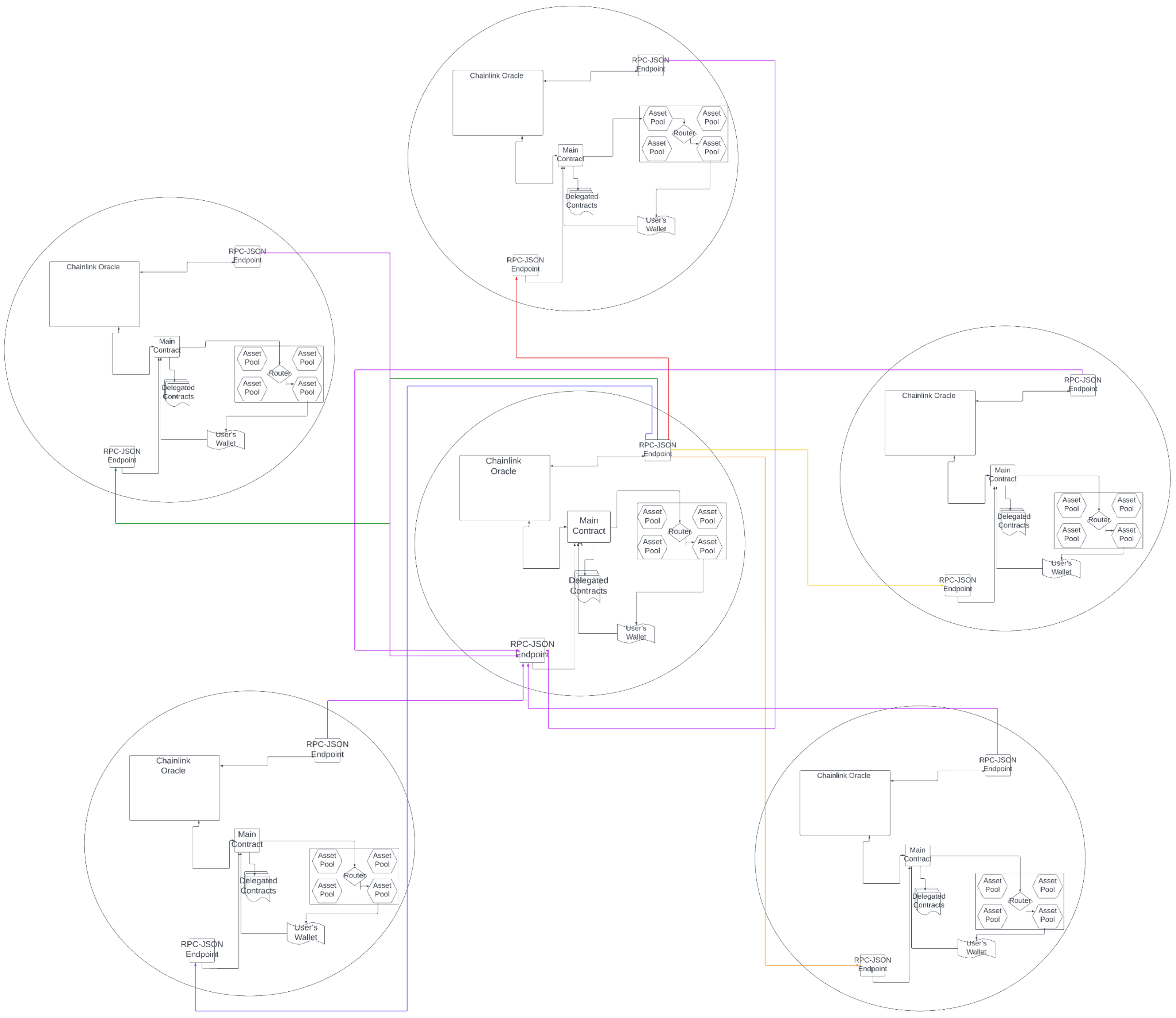
$$y_0 \cdot \left(\frac{x_\Delta}{x_\Delta + x_0}\right) = y_\Delta$$

With a node’s standalone architecture designed, the earlier discussed method of data transmission needs to be integrated to each node. This is where the node’s domain extends beyond solely on-chain deployed contracts. The JSON-RPC API mesh needs to be set up, consisting of many independently running nodes from a myriad of different sources, from custom hosted spun-up nodes to official node solutions, and this needs to be done for every supported chain. This mesh of JSON-RPC APIs need to be integrated with the Chainlink Operator Nodes the protocol is using at its oracle solution. Once each Chainlink Operator Node has ‘POST bridges’ calibrated to the IPs of the constituents of the chain-relevant segment of the JSON-RPC API mesh, custom job specifications utilizing the bridges can be set up for the Chainlink Operator Nodes to be run on request.

The preceding systems result in three components the protocol uses to support a chain: the Chainlink Nodes, the mesh of RPC nodes, and the deployed contracts. This yields us an intra-node architecture as follows:



When integrated with the overall inter-node architecture, we get:



With a protocol that is theoretically capable of facilitating cross chain automated market making, there still is the need to manage the scheduling of the data transfer between chains. As with any program running on the blockchain, there is only so much automation possible. Unlike traditional software, there have to be criteria that actions fire upon, rather than processing running constantly. So the data transfer needs to only occur when needed, and needs to occur reliably, or else the protocol becomes economically infeasible. The naive approach would be to utilize a scheduling service like Chainlink Keepers, but that is only applicable for taking chain state into account, and not useful for firing when certain off-chain criteria are met. As we are considering other chain data to be off-chain data, this rules out utilizing a scheduling service.

One may think the solution is simple, have users request the data themselves when they want it. Meaning, when a user initiates a transaction that is supposed to start on one chain and end on another, they simply ping the contract on the destination chain themselves, paying the costs themselves, for their transaction's data to be received. What this would look like is, they start a transaction on Chain A, to swap Asset A1 on Chain A to Asset B1 on Chain B. Then after initiating the transaction, they hop to chain B and request that the data gets transferred, paying the costs associated with this themselves. However, this is not easy to use, is not seamless, is an unimpressive solution, and yields a conundrum for users new to Chain B. How are they supposed to transfer assets to Chain B when they need assets on Chain B in the first place in order to receive them? In order for a protocol to gain wide adoption and to be of real value, utilization of the protocol needs to be a seamless experience.

The solution that allows for a seamless, economically feasible, self-contained model is incentivizing users to indirectly initiate these data transfers. On a traditional Proof-of-Work blockchain, the public is responsible for writing data to the chain every x amount of time - which is metered by the odds of getting a hash prefixed with the required amount of zeroes. Similarly but not identically, an incentive structure can be created where the metering for state-change is dynamic, based upon criteria where, if there is valid data needing to be transferred, the user initiating the state-change gets a reward, similar to a block reward on a traditional blockchain to the person writing the block to the chain. Two problems arise from this:

1) How does the protocol ensure that the requirements are met for payment if the reason for this subsystem is

that the protocol doesn’t know when certain off-chain requirements are met?

2) Where do these incentives come from?

Engineering a solution to the former is much simpler than for the latter. The solution to the first stated problem is that the criteria can be checked after finality - meaning after any data is written. Therefore, there is a delay in when the incentive gets paid; specifically, the delay in compensation can be thought of as the time-to-finality. The criteria can be checked with the state change, as the data fulfilling directly passed into the state-changing function can be made to include the data required to check if the criteria for data transfer are met. If not, then nothing happens: no state change, and no payment. The issue stemming from this is that for the eligibility to be evaluated, a data transfer already happened on some level, regardless of whether or not requirements are met and a state change happens. As long as there is a barrier to entry imposed on the user for initiating the data transfer process, and that barrier is sufficient to deter invalid requests, the protocol remains economically self-sufficient. Of course, this barrier to entry imposed on the user is in the form of value, similar to a deposit, and its return is dependent on the eligibility of the resulting state change request. What this looks like when taken from theory into practice, is:

- a user calls a function on the protocol’s main contract on that chain
- the contract performs a `delegatecall` on the deployed data-transfer delegate contract
- executing the code in the `delegatecall`, the contract selects relevant job specification IDs using a tortoise-and-hare algorithm
- The contract then makes a request to one of the Chainlink Operator Nodes integrated with the protocol, passing the selected job specification IDs
- The chainlink node then runs the relevant custom jobs
- The Chainlink Operator node makes a POST request over the previously integrated bridge specified in the job’s TOML specification

- This POST request, with the relevant request body, reaches the RPC pointed to by the bridge
- The RPC then queries the relevant data from the chain it is monitoring
- The RPC fulfills the POST request by responding to the chainlink node with a JSON object
- The chainlink node decodes the JSON object
- The chainlink node fulfills the original request made on the original chain
- The contract, upon request fulfillment, runs a series of evaluations to check data validity
- If the returned data is not valid, execution stops, reverting execution of certain delegate contract code
- If the returned data is valid, the request-initiating user gets sent their deposit back and their reward
- If the returned data was valid, the contract then proceeds to execute and take any appropriate state-changing actions

The way this can be used to integrate an AMM across multiple independent chains will be explained later on in this paper.

The second problem arisen by the solution taken to manage data transfer scheduling can be solved using the functionality of the protocol itself. Transferring assets across chains is something the protocol, at this stage, is conceptually capable of. As the problem involves someone on chain A wanting to move value to chain B, and someone on chain B wanting to be paid (for facilitating the on-time transfer of data), the protocol can facilitate payments from the user on chain A, who is making the swap, to the swapminer on chain B. This can be orchestrated to take place every time there is an action spanning multiple chains. Thus, each action will effectively always be made up of two sub-actions: the action the user wants facilitated by the protocol, and an additional swap between that user and the swapminer who indirectly facilitated the transfer of the data the first sub-action was dependent on. These pairs of sub-actions can be called meta-actions.

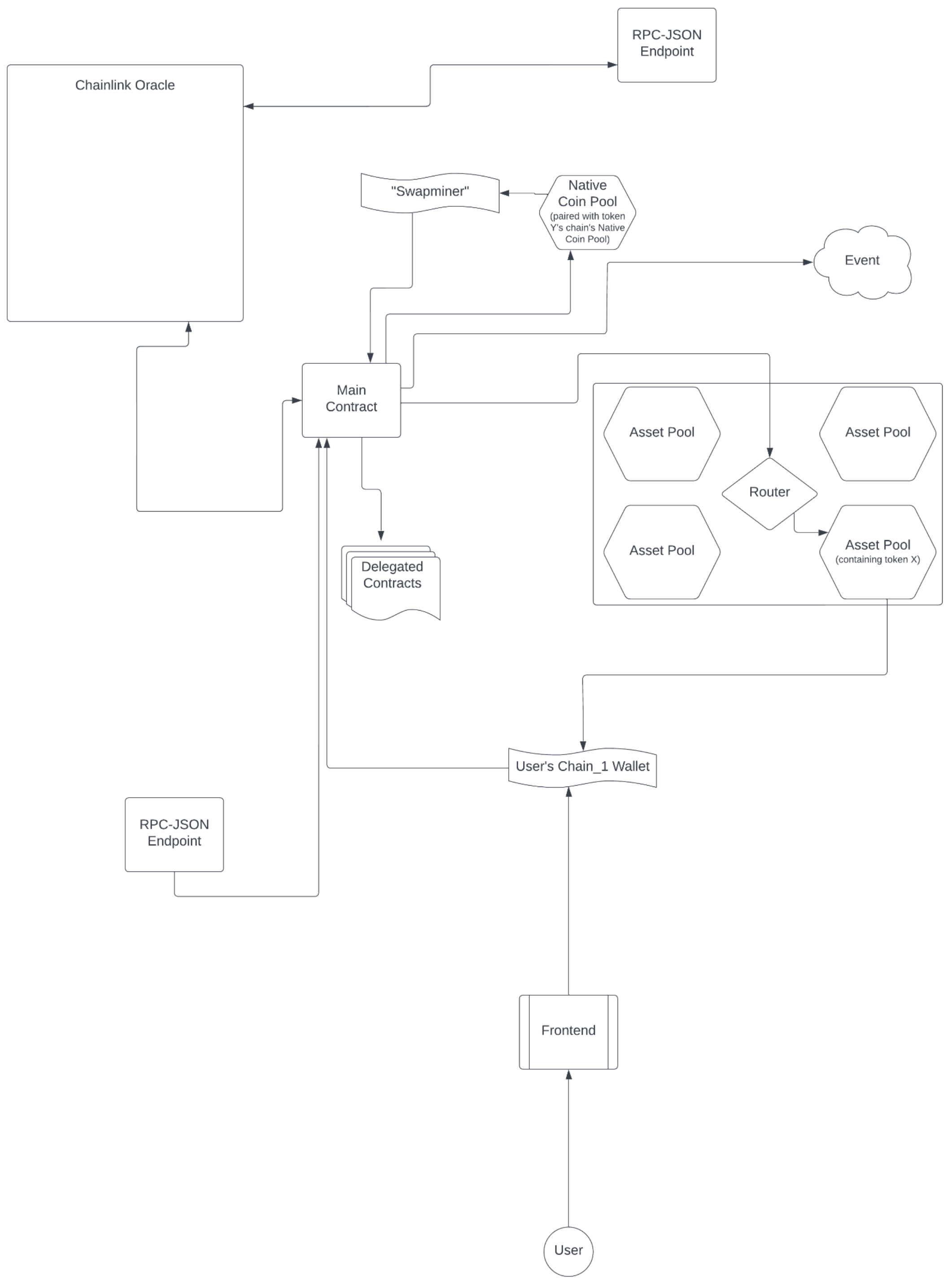
A minor issue that requires solving before this can be implemented accurately is ensuring that the user is paying the appropriate amount to properly fund the swapminer tip. The solution to this is simple, as this is in a category of issues that can easily be solved by utilizing Chainlink’s price feeds, and getting the price of the chain’s native token in terms of LINK, which is what the swapminer’s barrier-to-entry must be paid in. Specifically, as the swapminer’s barrier-to-entry goes directly to a Chainlink Oracle in the final protocol design for the sake of complete protocol self-sufficiency, the swapminer’s fee is paid in ERC-677 compliant LINK. Since ERC-677 LINK can be bridged at a 1:1 rate with ERC-20 LINK, the simple price feed is a sufficient solution, with further calculations being done in-contract. The price feed can be called, with the subsequent calculations being performed, by any user simply by invoking the updateTipDefault() state-changing function.

Retrieving data is only half of the system of cross-chain data transferring. The other half is supplying the data. Each node needs the ability to provide the relevant data at the time of request. The determination of relevance must be settled to, and for the sake of preventing any state-changes from being required when supplying data, the protocol will have to specify the data it wants at retrieval function invocation. Thus, the deployed contract serving the data must earmark data constituting a meta-action at the time such data is created, and do so to identical standards across all chain deployments. This can be done by creating a data structure containing all possible data resulting from cross-chain user-initiated actions, and accessing them via mapping a transaction number to said data structure. This method allows the peripheral contract and the central contract to stay in sync with each other, and to know what data to query next and how to serve it. By passing as an argument the number of node-B originating transactions that node-A has processed to the view function on node B’s deployed main contract, node A and node B are both serving chronologically sequential data - respective to node A’s perspective. This results in having a view function, meaning a non-state-changing function, that simply acts as an intermediary between the caller and the ‘data structure - transaction number’ mapping. The reason this cannot be simply a public variable is the requirement to revert on non-existant transactions rather than returning blank data.

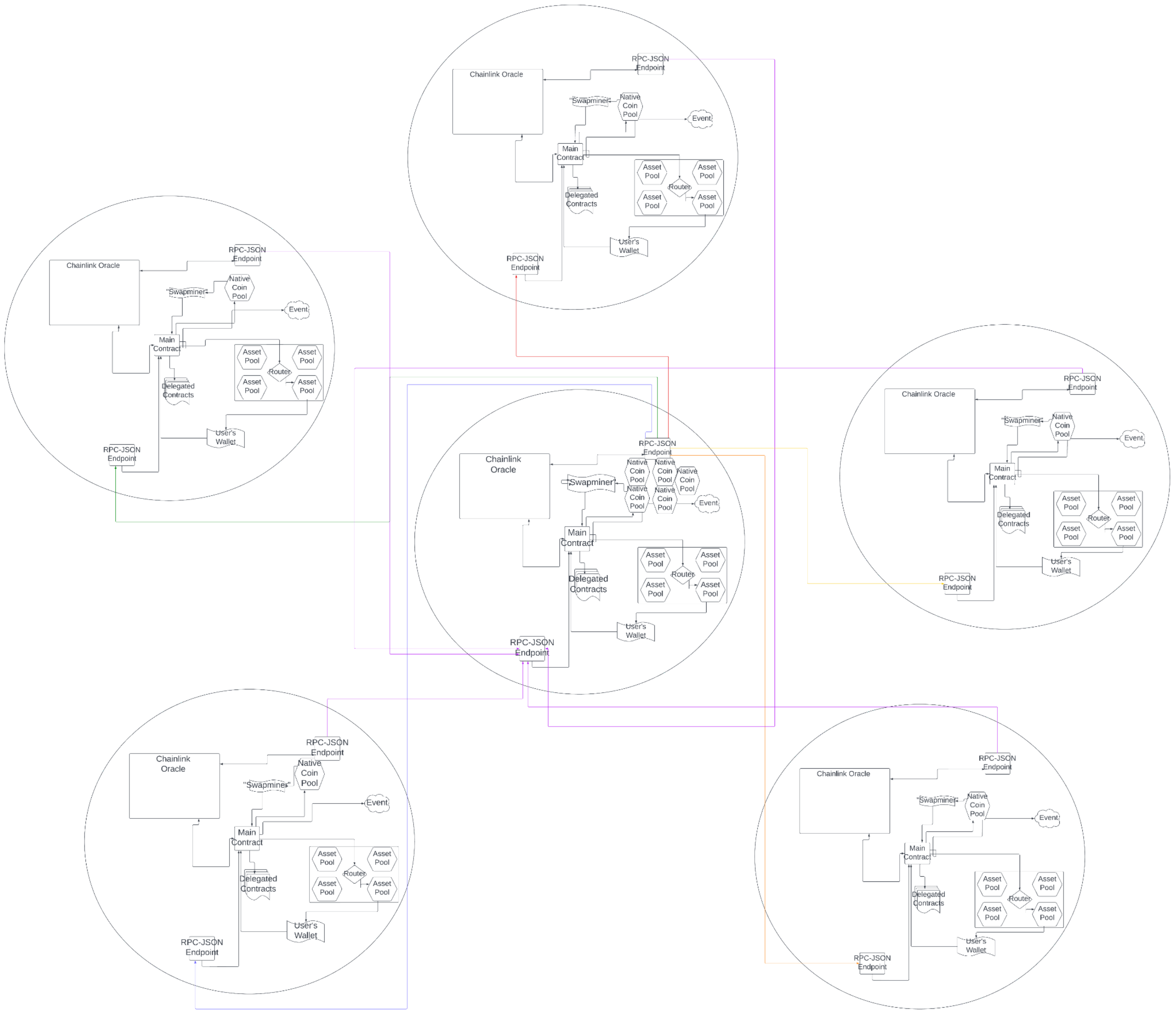
So now, our data-transfer system is conceptually mature enough to transfer assets between two chains in a seamless and economically self-sufficient manner. However, further engineering lies in adapting this concept for the central-peripheral layout ideal for a multi-chain AMM. Adapting the peer-to-peer data transfer arrangement to a central-peripheral arrangement calls for a simplification of this system for peripheral nodes. Peripheral nodes oraclePing function will be limited to having no parameter representing the internal chain ID for JSON-RPC API mesh segment being queried, and instead be restricted to querying the node on the chain with the internal chain ID of 0 - the central node’s chain. Central nodes will still take an argument, specified by the swapminer as an argument passed in at the time of function invocation, representing the chain possessing the data desired to be transferred.

The primary security concern that needed to be addressed and engineered for results from the fact that blockchains, despite being non-centralized, dependable and resilient, are only truly immutable introspectively or with a sufficiently large viewing delay. There are many times that a block written to the chain can subsequently be rewritten. The more resilient, slower, and difficult-to-write-to the chain is, the more rare this is, which results in a real world scenario of cheaper, faster chains being more vulnerable to experience reorgs, more likely to have longer sequences of blocks involved in a reorg. The rational countermeasure to take is to implement a temporal maturity threshold on data being served. Meaning, if the protocol can refuse to act on data that has been on chain for an insufficient amount of time. The most economical way this is accomplished is for a node to simply not serve data that has been on the chain for less than X amount of blocks. So, the view function for the data that acts as an intermediary between the caller and the ‘transaction number - data structure’ mapping simply reverts if the transaction took place less than X blocks ago. X can have a different value for each protocol node. By default it is being set to 5 at launch, meaning at the time of invocation, the view function will only avoid reverting if the relevant meta-action was initiated 6 blocks ago or more.

With the system fully designed, we have the following intra-node architecture:



And the following diagram represents the protocol's complete architecture and its conceptual layout:



Protocol Module Analysis

The following in-depth analysis of the systems and modules of the protocol will be broken up into a few sections.

- Asset Pools
- Meta-transactions
- Governance

Asset Pools

Asset pools are the heart of any automated market making technology. For Morphswap, these Asset pools are deployed by a protocol node for, as the name suggests, each Asset pair involving assets respective to the node’s chain. An asset pair involves two Asset Pools - one deployed on each asset’s domestic chain. These deployed contracts are technically ERC-20 compliant, and can be thought of as having their own ERC-20 tokens. These tokens are issued by the protocol to keep track of ownership of capital provisioned as liquidity - they can be thought of as similar to Uniswap v2’s Liquidity Provider (LP) tokens.

As far as liquidity management, there were some design decisions stemming from limitations and complications imposed by the protocol spanning multiple chains. Foremost, providing two assets with one transaction cannot be done in the same manner across multiple chains as on a single chain. There are a few ways that this was handled:

-single-sided liquidity
-automatic double-sided liquidity
-manual double-sided liquidity

For single sided liquidity, allowing a user to take this option is a fundamentally simple engineering task to accomplish. The reason some automated market makers do not allow this is because of massive risk exposure taken on by the unaware invoking user and the ability for value to be extracted from a protocol allowing single sided liquidity. Because of limitations imposed by the inherent nature of cross-chain decentralized commerce, single sided liquidity is something that will be present. With the countermeasures taken against liquidity value extraction that have been implemented, and edification of the users, the protocol allows single-sided liquidity to be provided. In this protocol, where LP tokens refer only to a single side of an asset pair, the specific exposure effects of providing single-sided liquidity can differ drastically from double-sided liquidity on a traditional automated market maker where LP tokens signify ownership of the entire pool.

Firstly, let's assume that the value for both sides of a token A - token B pair are at 1000. When a user provides 100 token A in a single-sided liquidity provision, they will own ~9.09% of the token A side of the asset pair, which currently has 1100 token A provided as liquidity. Or, as token A and token B were staked in equal amounts at the time of liquidity provision, 1 token A = 1 token B at t0, where t0 is the time of the user’s liquidity provision. So we can also say, in terms of t0 values, the user owns 4.76% of the pool, and in terms t1 (where t1 is the moment after liquidity provision), the user owns ~4.45% of the pool (9.09% of one side of the pair, so 9.09%/2 = 4.45%). From there, if users tend to provide liquidity to the token A side of the pool, leading to users swapping token B for token A due to price caused by the imbalance liquidity provision of the pool, the user’s share of the pool can fall dramatically. To elucidate, let’s look at some examples. Immediately after t1, another user might swap 100 token B for token A, reducing the amount of token A in the pool to 1000. We’ll call this point in time t2. Then a third user provides single-sided liquidity again in the same quantity as our first user, 100 token A. We’ll call this point in time t3. The pool is now back to equilibrium, 1100 token A, and 1100 token B. However, the original user’s share of the pool is at its lowest point in terms of the nominal amount of token A the user is entitled to. At t2, the amount of token A the user is entitled to is 9.09% of 1000, so 90.9 token A. Then at t3, the pool gets brought back to equilibrium at 1100 token A, however, the user is no longer entitled to 9.09% of the pool, as all ownership got diluted by 10% when the pool size increased by 10% due to single-sided liquidity provision. The user owns 8.26% of the pool, which is 90.9 token A. So in only 3 transactions, the user lost an eleventh of their value.

Now, let’s look at the other possibility, and what the liquidity providers of token B experienced in the previously stated series of events. A user provided 100 token X in a token X - token Y asset pair where both sides of the pool have 1000 of their respective tokens as provision liquidity. At the point in time immediately after these 100 are provided, which we will call t1, the user again owns ~9.09% of the token X side of the asset pair, which currently has 1100 token A provided as liquidity. Or, as token X and token Y were staked in equal amounts at the time of liquidity provision, 1 token X = 1 token Y at t0, where t0 is the time immediately preceding the user’s liquidity provision. So we can also say, in terms of t0 values, the user owns 4.76% of the pool and, in terms t1, the user owns ~4.45% of the pool (9.09% of one side of the pair, so 9.09%/2 = 4.45%). A second user then sells 47.61 token Y to receive 52.38 token X in exchange, thereby bringing the pool back to equilibrium where both sides of the pool have ~1047.61 of their respective tokens. We will call this moment to be t2. The user owns 9.09% of 1047.61 token X, which is only 95.2 tokens. Then, a user provides 200 token Y as liquidity. Subsequently, due to price difference, a user then chooses to swap 100 token X for 108 token Y. At this point, t3, there are 1147 token X and 1139 token Y. After this, another user provides another 200 token Y as liquidity, followed by another user attracted by the arbitrage opportunity swapping 100 token X for token 107 Y. This leaves this asset pair with 1247 token X, and 1232 token Y, which for the sake of simplicity, we will consider effectively at equilibrium. Our original user, who provided 100 token X, now owns 113.35 token X. Over time, this can compound greatly, effectively giving users exposure to the size of the liquidity pool itself, assuming they are on the side of the pool where single-sided liquidity is less common; ideally this rewards users for providing liquidity to the more neglected side of a pool.

In reality, it is far more complex than this, and most users should protect against volatility caused by this “asymmetric loss/gain” by using single-sided liquidity provision simply as a tool to easily and cheaply provide liquidity to both sides of a pool. That way, any “asymmetric loss” on one side is counteracted by “asymmetric gain” on the other side. It is recommended to not provide single-sided liquidity in increments greater than 50% of the respective side’s liquidity, and to maximize the transience of imbalance by providing both sides of liquidity in the minimum time frame feasible. This is due to the impermanent loss caused by arbitrage taken when two sides of a pool become too unbalanced. If there is a small pool that a user wants to provide 200% more liquidity than the pool currently has, it is highly recommended they do it in increments no greater than 50% of the respective side’s current liquidity (ideally no greater than 25%) at the time each increment is provided.

For automatic double-sided liquidity, this involves the protocol instantly selling half of the provided asset and supplying the proceeds as liquidity, the ownership of which is attributed to the initial provider. An important characteristic to note about this meta-transaction, is that since it does automate activity on a chain other than the domestic chain, it does require a swapminer tip to be paid. The automated process executed by the protocol begins on the domestic chain of this double-sided liquidity meta-transaction. For similar reasons as the recommended relative threshold of pool size for increments of provided single-sided liquidity, it is recommended to provide automatic double-sided liquidity in increments of the initial token no greater than 25% of the initial side’s provisioned liquidity.

The appropriate proportion of the second side of the liquidity pair will be attributed to the address passed in as an argument by the user invoking the double-sided liquidity meta-transaction. The appropriate amount of newly minted liquidity provider tokens is arrived at by applying the following formula:

$$\frac{\left(\frac{S_i \cdot x}{x^{-1} - 1}\right)}{x} = S_n$$

X = proportion of pool being allocated to user
Si = initial supply of destination chain liquidity provider tokens
Sn = amount of liquidity provider tokens for the asset pool to mint and send to stated address

Simplified, the equation becomes:

$$\frac{S}{\frac{1}{x} - 1} = S_n$$

Thereby, the formula for deriving the external chain liquidity provider token output based on the amount of the pair’s domestic chain asset input is:

$$\frac{S_i}{\left(t_n \cdot \frac{1}{2\cdot(t_n+t_i)}\right)^{-1} - 1} = S_n$$

Si = initial supply of destination chain liquidity provider tokens
Tn = amount of initial token provided by user
Ti = amount of initial token initially in the pool before invocation of this meta-transaction
Sn = amount of liquidity provider tokens for the asset pool to mint and send to stated address

The precise sequence of transactions that constitute a double-sided liquidity meta-transaction are as follows:

- the user invokes the autoDoubleLiquidity() function
- the protocol initiates a native coin swap for the purposes of tipping the swapminer who mines this meta-transaction
- The protocol transfers the fully specified amount of tokens from the user to the asset pool
- The asset pool mints new liquidity provider tokens, in proportion to 50% of the user’s tokens relative to the total amount of provided tokens.
- The asset pool transfers these newly minted liquidity provider tokens to the user
- A swapminer invokes the oraclePing() on the meta-transaction’s destination chain
- The segments of the JSON-RPC mesh selected by the tortoise-hare algorithm are mapped to their respective job specification IDs, and these IDs are sent to the Chainlink Oracle in the form of bytecode.
- the meta-transaction, which instantly became eligible for being queried by the JSON-RPC mesh segment and bypassed the reorg block protection as it is classified as a liquidity meta-transaction, gets queried by the previously selected RPCs in the mesh segment
- On the destination chain, the Operator request from the oraclePing() invocation is fulfilled
- On the destination chain, the protocol contract contacts the native coin asset pair pool
- On the destination chain, the native coin asset pair pool sends the appropriate proportion of its liquidity to the swapminer
- On the destination chain, liquidity provider tokens are transferred to the address passed in as an argument by user at invocation of autoDoubleLiquidity(), in the amount arrived at by the previous formula

This is a process that automates the process of providing both sides of liquidity to an asset pool, and comes at the cost of providing a swapminer tip. As such, it will primarily be of use for larger pools and on cheaper chains.

The final method of liquidity provision is requiring both sides to be provided before adding any more liquidity to the pool, and having no block re-org delay when transmitting confirmation of provided external-chain asset liquidity, and allowing this data to be transmitted and requested out of chronological order respective to other meta-actions. This requires more involvement and fees from the user than the other two methods of liquidity provision, but as it safely guards against arbitrage and pool imbalance, it is the recommended option for most users, especially when providing large amounts of liquidity.

Providing manual double-sided liquidity is a complex action requiring multiple meta-transactions. Specifically, the multiple components are:

1. User invoking manualLiquidity()
2. User then invoking finishManualLiquidity() on second chain
3. acknowledgeFinishLiquidity() then being invoked on initial chain automatically once the finishManualLiquidity() meta-transaction gets swapmined

Importantly, both user initiated meta-transactions require tips paid to swapminers.

Ultimately, the protocol allows the user to choose any of these methods when providing liquidity. As discussed above, because of the pricing equation

$$\frac{x_\Delta}{x_\Delta + x_0} = \frac{y_\Delta}{y_0}$$

, there are optimal scenarios for the first two options, with the third option being more involved but closest to standard for automated market making protocols, while the first two options provide ease of use to the liquidity provider, and the first one allows for unique asset exposure and hedging opportunities.

A way that a profit-seeking actor might extract value from this system would be to remove a single side of their provided liquidity every time a large sale was about to come in. This would greatly negatively impact the slippage experienced by the user initiating the swap while being a lucrative behavior taken by the actor withdrawing liquidity. The countermeasure implemented to prevent this from occurring and rendering the protocol useless was to impose a queue for withdrawing liquidity, where a user has to submit a transaction signifying intent to withdraw, and after 20 blocks have passed, they can proceed to confirm their withdrawal, which subsequently redeems the previously stated amount of liquidity tokens - reverting if amount exceeds the user's balance - and sends the corresponding amount of the asset to the user.

One of the new characteristics that Morphswap brings to the cryptocurrency cross-chain bridge ecosystem is its full extensibility. Creating the ability for any user to make any pair of any tokens on any chains to be eligible to be swapped is novel, at the core of the protocol, and one of the reasons for its emergence. Any user is able to create a new pair, as long as they have access to two wallets on two different chains, each with their respective assets that the user wants to provide. The user needs to initiate the pool creation on the peripheral chain involved in the chain. As elaborated on previously, peripheral-to-peripheral pairs are not valid for the protocol. Each pair must involve the central protocol node. When creating a new pool, the user must initiate this creation process on the pair's peripheral chain, while finishing the process on the central chain. The transactions involving the protocol for a new pair genesis is are as follows:

- User invokes the createPair() function on the peripheral node's contract
- The contract then makes a delegatecall to the createPair delegate contract, invoking the namesake function

- Protocol transfers native coin to native coin asset pair pool as a swap for the swapminer tip
- Protocol transfers the specified amount of tokens from user’s address to the escrow module
- The resulting meta-transaction from the createPair() invocation is seen by a swapminer, and the swapminer makes a transaction to oraclePing() on the central node contract

Then, on the central chain:

- The contract sends the job specification IDs to the Chainlink Oracle node
- Once the RPCs make non-reverting calls to the peripheral node contract and return the values to the Chainlink Oracle node, the request made by the swapminer is fulfilled on the central chain.
- The swapminer is sent the appropriate proportion of the native coin relative to the liquidity in the respective native coin asset pool being used to fulfill swapminer tips
- The protocol then marks this asset pair as eligible to be finished by the address specified by the user who invoked the createPair() function
- User invokes the finishPair() function
- The protocol contract then makes a delegatecall to the finishPair delegate contract, invoking the namesake function
- The protocol contract transfers native coin to native coin asset pair pool as a swap for the swapminer tip
- The protocol contract deploys a new asset pool contract, corresponding the new asset pair
- The protocol contract transfers the specified amount of the asset to the newly deployed asset pool

Then, back on the peripheral chain:

- The resulting meta-transaction from the finishPair() invocation is seen by a swapminer, and the swapminer

  makes a transaction to oraclePing() on the peripheral node contract
- The contract sends the job specification IDs to the Chainlink Oracle node
- Once the RPCs make non-reverting calls to the peripheral node contract and return the values to the Chainlink Oracle node, the request made by the swapminer is fulfilled on the peripheral chain.
- The swapminer is sent the appropriate proportion the of native coin relative to the liquidity in the respective native coin asset pool being used to fulfill swapminer tips
- The protocol then makes a delegetcall to the acknowledgeFinishedPair delegate contract, invoking the namesake function
- Once the protocol automatically marks the pair as "completed", it then deploys a new asset pool contract corresponding to the newly completed asset pair.
- The contract then transfers the appropriate assets from escrow to the newly deployed asset pool contract

One should note that this process involves two separate swapminer tips needing to be paid by the user.

Users should additionally be aware that only one pair can exist for each Asset-Asset pairing. This means there can not be more than one pool for a pairing of Token A - Token D. Pool pairs are exclusive. There are a few ramifications of this requirement that each resulted in necessary countermeasures. The first ramification is that a user could start creating a pool and never finish it, leaving it unable to be used. The naive approach to fix this would be to allow multiple people to create a pool, and only the first one finished would be the valid pool. While that initially seems like a valid solution, it requires a few extra components to make it work. Specifically, to prevent a malicious user from starting to create a pool after someone else while putting in the same address to expect the asset from, we had to add exclusivity for addresses in respect to asset pairings. So if an address was declared to be providing the second asset, that address could not be declared again for the same asset pairing. Second, in order to keep the nodes in mutually compatible states, the limitation was implemented to require all pair genesis completion to take place on the central chain. Otherwise, two Token X - Token Y pairs could possibly both be finished “first” according to their respective domestic chains’ perspectives. Therefore, createPair() can only be invoked on peripheral chains.

Meta-transactions

Meta-transactions are user-initiated actions taken on the Morphswap protocol that involve sub-actions across multiple entangled chains. When a user swaps from Token A1 on Chain A to Token C1 on Chain C, that is a meta-transaction. At the most, a meta-transaction can involve over a dozen transactions across three chains. The meta-transaction constituent transactions can be conceptually grouped into sub-actions and meta-actions as discussed previously. The resulting data that needs to be transmitted from one chain to another is stored and kept in a specific data structure, with this data structure being used by all deployed contracts and all protocol nodes at every stage of data transmission, both off-chain (eg. JSON Objects from the RPC API) and on-chain (eg. the mapping used for meta-action indexing). The data structure is as follows:

```
struct txObject {
        uint8 method_id;
        uint8 internal_start_chain_id;
        uint8 internal_end_chain_id;
        uint64 pair_id;
        address finalchain_wallet;
        uint64 secondpair_id;
        address firstchain_asset;
        address finalchain_asset;
        uint64 quadrillion_ratio;
        uint64 quadrillion_tip_ratio;
        uint128 rtx_num;
        bool alt_fee;
    }
```

This format allows for all transactions to utilize the same structure, maintaining a wide range of options for protocol governance. There are a few non-intuitive things to go over with the construction of this object. Primarily, the data structure does not store the nominal amount of tokens or currency sent in a user-initiated action. Instead, it stores to the output of the formula:

$$\frac{(S_\Delta * 1{,}000{,}000{,}000{,}000{,}000)}{S_0}$$

S_Δ = the nominal sent amount of the token or coin
S_0 = the total amount of liquidity in the pool on the side of the sent token or coin

The reason that the token amount is encoded this way is for two reasons. Secondarily, this reduces, by 50%, the size of data required to store the amount of tokens sent. Primarily, encoding the ratio rather than the amount still gives us precision to within one quadrillionth of the real amount, and protects against overflows. Encoding the raw number would require usage of a 256-bit uint, for the 0.001% of the time that someone was interacting with a token in amounts large enough to cause an integer overflow in a 128-bit uint. So, this reduces the data for recording and transferring the amount of sent tokens, which slightly improves the economics of the protocol. The calldata reduction is a minor benefit, since calldata is not very expensive, but on average, working with a uint128 without risk of integer overflow is both slightly cheaper and more salubrious for the protocol than either having to use 256-bit unsigned integers, or to tell people they can't use tokens with a supply exceeding (2^128)[2].

The method that Morphswap uses to swap between any two supported chains, despite only the central node having a connection to every supported chain, is to split up the meta-transaction into two meta-transactions, initiating the second upon completion of the first. An basic example of this is:

[2] In terms of the smallest possible units of commerce for that token or coin- for example if token C's contract has a decimal value of 8, "1 of token C" would be considered in the EVM as 100,000,000 units of token C

- The central chain is chain 0
- A user initiates a token swap from token E on chain 1 to token K on chain 2
- The protocol swaps from token E on chain 1 to Token V on chain 0
- The protocol then swaps token V on chain 0 with token K on chain 2

These peripheral-to-peripheral transactions are the same as normal transactions, except the method id is 10, and the data structure has a non-zero secondPairID. The central chain node, when decoding an incoming data structure, checks the method id unsigned integer, and if its value is 10, then creates a swap for the secondPairID. The pair ID is all the external data protocol needs to make a new swap, as it completes the first swap with itself as the recipient, and creates a new swap for the second pair using the proceeds from the first swap, with the recipient of the second swap to be the user-stated recipientWallet argument. The swapminer tips of these peripheral-to-peripheral transactions are handled in the same manner, with the original user needing to supply

$$(1 \cdot m_c \cdot d) + (1 \cdot m_2 \cdot d)$$

m_c = the required tip multiplier for the central node
m_2 = the required tip multiplier for the final chain
d = the default tip amount

for tips at the time of initiation. $(m_c \times d)$ is given to the swapminer for the peripheral-to-central swap, with $(m_2 \times d)$ being given to the protocol node to be used to fund the sub-action of tipping for the second swap.

Governance

As the protocol is governed by the token holders, there are myriad options that token holders can set and modules that token holders can activate or deactivate by a proposal being accepted with over 33% of the total tokens affirming the proposal. Additionally, the token holders have the power to change anything about the protocol by running arbitrary code, if the code proposal gets over half of all tokens affirming the proposal. This means if a malicious actor gets over 50% of the governance token supply, they can completely destroy the protocol. Therefore, all users of the protocol are stakeholders in the token's value, rather than just the protocol. The proposal types that need 33% of the total supply to be in favor (with fewer votes being against) in order for the proposal to take effect can be seen in this table:

<u>Governance Proposals Table</u>

Proposal number	Activation threshold	Description	Proposal creation data
1	33%	Changes the value of the volume fee given to liquidity providers	The new rate, in terms of basis points
2	33%	Changes the proportion of the referral bonus	The new rate, in terms of basis points
3	33%	Changes the base Oracle fee constant	The new base fee value, in terms of juels[3]
4	33%	Toggles the ability for users to pay swapminer tip with Morphswap tokens [4] [5]	Even number for off, odd number for on
5	33%	Toggles the ability for users to pay swapminer tip with Morphswap tokens[6]	Even number for off, odd number for on
6	33%	Changes the value of the default tip multiplier constant	The new value

[3] 0.000000000000000001 LINK (1 LINK ÷ 10^{18})

[4] Additionally toggles the usage of a dedicated Chainlink price data-feed for the Morphswap Governance token

[5] upon activation of the ability to pay with the Morphswap Governance Token, users can elect to pay with the Morphswap Governance Token for a fee discount

[6] Only toggles the alternate tip ability. Does not rely on a Chainlink price data-feed

7	33%	Changes the value of the default tip multiplier constant for the alternate tip	200% of the value of the new desired constant
8	50%	Changes the governance contract to the contract at the address set by the author of the proposal. Can be used to perform any state-changing operations to the protocol	The address of the new governance delegate contract[7]

The way to make a proposal is to use the *addProposal(uint proptype, uint newrate, uint startingweight)* function, with the first argument being the type (as seen in previous table), the second argument being the new value, and the third value being the amount of tokens the user creating the proposal is initially staking in favor of the proposal. In order for a proposal to be made, to discourage proposal spamming, 2% of the total supply must be staked at the proposal's genesis in order for the proposal to be created.

To vote on a proposal, you must stake your tokens in affirmation or denial of the proposal, with each staked token counting as a vote. You can unstake them at any time.

Proposal lifespans are by default 10,000 blocks. This means if the activation threshold for the proposal type has not been reached by 10,000 blocks, the proposal is no longer valid.

Documentation

V1 Internal Chain Identification numbers

Chain Name	Internal Chain ID
Polygon[8]	0

[7] encoded as a 256-bit unsigned integer
[8] The protocol's central chain

Binance Smart Chain	1
Ethereum	2
Fantom	3

```
function OraclePing(uint8 _icid) public returns (uint128, bool)
```

Accepts:

Parameter	Data type	Description
_icid	8-bit unsigned integer	The internal chain ID of the chain to be queried[9]

Returns:

Parameter	Data type	Description
rTXNumber	128-bit unsigned integer	The transaction number being queried for[10]
success	boolean	Status of the function invocation[11]

```
function buy(uint64 pairID, uint saleAmount, address
chain2Wallet, uint128 tipAmount, bool multichainHop, uint64
secondPairID, address referrer,  bool altFee, uint chain2)
public payable returns (bool)
```

Accepts:

[9] If the function is being called on a peripheral chain, this number should generally always be 0
[10] Alternatively, the one-indexed current number of transactions from queried chain proccessed by this chain
[11] Should always evaluate to true if transaction does not revert

Parameter	Data type	Description
pairID	64-bit unsigned integer	The ID of the asset pair involved in the swap[12]
saleAmount	256-bit unsigned integer	The amount of the asset being swapping from[13]
chain2Wallet	address	The wallet on the destination chain that will be receiving the tokens being swapped for
tipAmount	128-bit unsigned integer	The amount of the domestic chain's native currency to be sent as the swapminer tip[14]
multichainHop	boolean	True if the meta-transaction is peripheral-to-peripheral. Otherwise, false.
secondPairID	64-bit unsigned integer	If a peripheral-to-peripheral swap, this refers to the ID of the pair involved in the central-to-peripheral swap. Otherwise, this should be 0.
referrer	address	Refers to the address of the person who the msg.sender is being referred by.[15] Otherwise, this should be the zero-address.[16]

[12] In a peripheral-to-peripheral swap, this refers to the origin pair - the peripheral-to-central swap pair
[13] In terms of the smallest possible units of commerce for that token or coin- for example if token C's contract has a decimal value of 8, "1 of token C" would be considered in the EVM as 100,000,000 units of token C
[14] For example, Ether in terms of Wei
[15] Only applicable for the sender's first interaction with the contract.
[16] 0x0000000000000000000000000000000000000000

altFee	boolean	Refers to whether or not the msg.sender wants to pay the swapminer tip in the Morphswap Governance Token.[17] If true, the tipAmount then refers to the amount of the Morphswap Governance Token to be sent as swapminer tip.
chain2	256-bit unsigned integer	The external chain ID of the final chain of a swap

Returns:

Parameter	Data type	Description
success	boolean	Status of the delegate-call of the buy function on the buy delegate contract

```
function buyWithNativeCoin(uint64 pairID, address c2w, uint128
tipamarg, bool multichainhop, uint32 secondpairID, bool refbool,
address referrer, bool altfee, uint c2) public payable returns
(bool)
```

Accepts:

Parameter	Data type	Description
pairID	64-bit unsigned integer	The ID of the asset pair involved in the swap[18]
chain2Wallet	address	The wallet on the destination

[17] Only applicable if the token holders previously voted to enable the feature of allowing tips to be paid in the Morphswap Governance Token

[18] If a peripheral-to-peripheral swap, this refers to the origin pair - the peripheral-to-central swap pair

		chain that will be recieving the tokens being swapped for
tipAmount	128-bit unsigned integer	he amount of the domestic chain's native currency to be sent as the swapminer tip[19]
multichainHop	boolean	True if the meta-transaction is peripheral-to-peripheral. Otherwise, false.
secondPairID	64-bit unsigned integer	If a peripheral-to-peripheral swap, this refers to the ID of the pair involved in the central-to-peripheral swap. Otherwise, this should be 0.
referrer	address	Refers to the address of the person who the msg.sender is being referred by.[20]
altFee	boolean	Refers to whether or not the msg.sender wants to pay the swapminer fee in the Morphswap Governance Token.[21]
chain2	256-bit unsigned integer	The external chain ID of the final chain of a swap.

Returns:

Parameter	Data type	Description
success	boolean	Status of the delegate-call of the buyWithNativeCoin function on the

[19] For example, Ether in terms of Wei

[20] This is only applicable for an address's first interaction with the protocol.

[21] Only applicable if the token holders previously voted to enable the feature of allowing tips to be paid in the Morphswap Governance Token

		buyWithNativeCoin delegate contract

V1 Limitations

Version 1 (V1) will have a certain degree of limitations respective to the vision outlined in this paper. Mainly, due to a desire to minimize time-to-market and launch a fully functional proof-of-concept before the allocation of more resources and capital, there are two system modules that will be activated on the deployment of version 2 and are not in version 1. This is the selection method of the JSON mesh resources. In version 1, the JSON mesh will be optimized for simplicity and functionality rather than redundancy. In version 2, the RPC mesh will be fully adherent to the stated vision, and able to be contributed to by the stakeholder and tokenholders of the protocol via governance actions. The second shortcoming of version 1 will be the inability for double-sided liquidity to be added. This is because the ability to transfer meta-transaction constituent data structures in a temporally non-sequential order requires remaking the smart contract code skeleton from the ground up post-deployment, which is more than can be done from simply manipulating delegatecalls with proxy contracts.

Outlook

Supported chains are added to the central node after deployment, with internal chain IDs being allocated chronologically. This can be thought of as the internal chain ID system being zero-indexed, or as the internal chain IDs of the peripheral nodes being one-indexed, with the central node having the internal chain ID of 0. This design also allows for future flexibility, as all it takes for a blockchain to be supported is for smart contract compatibility, and a way to query the blockchain with a POST request. The protocol will be launched only with support for EVM-compatible chains initially, but with adequate development time, non-EVM chains such as the Bitcoin blockchain, Solana, Cardano, or possibly even Monero* *(although Monero will require an additional layer of software between the protocol and the Monero blockchain which may not be possible to do in a satisfactorily trustless way) will be supported by the Morphswap protocol. The impact that the ability to swap Bitcoin with any other asset will have on the industry cannot be overstated. The ability to directly swap any asset on any blockchain with any other asset in the cryptocurrency ecosystem without any centralized parties involved will revolutionize the industry and will end the dilemma of being dependent on centralized players to facilitate cross-chain commerce of decentralized finance.

Glossary

- **Node**: A deployed segment of the Morphswap protocol isolated to a single chain, dependent on the nodes deployed on other chains.
- **Central node**: The Morphswap node for Polygon. Has bi-directional asynchronous communication with peripheral nodes.
- **Peripheral node**: The Morphswap nodes for each supported chain (other than Polygon). Does not communicate directly with each other. Each communicate bi-directionally and asynchronously with the protocol's central node.
- **Meta-transaction**: any user-initiated action taken on the Morphswap protocol that involves data being communicated from one chain to another. This will always end at, begin at, or involve the central protocol node. Examples of actions on the Morphswap protocol that should be considered meta-transactions include: a token swap, creating a new pool, and providing automatic dual liquidity. Examples of actions taken on the Morphswap protocol that do **NOT** count as meta-transactions: single-sided liquidity, requesting liquidity removal, and governance actions.
- **Domestic chain**: the chain that a meta-transaction is initiated on

- **Swapminer:** any user (or bot) that pings a Morphswap node contract, incentivized by a reward fee paid every time the query initiated by the ping results in new data on meta-transactions that the queried node is entangled in
- **Oracle:** a node or service that functions as a way of trustlessly pulling off-chain data on-chain
- **Chainlink Operator node:** a type of Oracle, running on the Chainlink protocol, capable of fulfilling a larger variety of responses and returning a larger amount of data than a typical Chainlink node. Became available in v0.7.
- **JSON-RPC Mesh:** an interwoven mesh of JSON-RPC nodes that enable Morphswap to be fully decentralized, and avoid any single point-of-failure. Connected to each Chainlink node as bridges.
- **Asset Pool:** similar to a pair on a traditional, single-chain AMM, except each of the two sides of the pools that make up the pool are located on different chains. Each side of the pool is a contract that contains an asset constituting half of the pair (the asset running on the chain of its respective side).
- **Reorg:** this is when one or more of the most recent blocks of the blockchain get reorganized and replaced with different blocks. This is because nodes will always choose the longest chain, and it's possible that the longest chain is one other than the chain the node in question is referencing locally. This can cause problems if a specific transactions happens and is later removed.
- **Reorg protection delay:** a technique used by the Morphswap protocol to protect against block reorgs. It involves requiring 6 blocks to pass before considered a contract interaction 'complete', and reverting JSON-RPC queries of a meta-transactions unless 6 blocks have gone by.
- **Morphswap Governance Token:** an ERC-20 compliant token, the ownership of which signifies a proportional share of ownership of the Morphswap protocol, and allows the token holder to govern the protocol. Everything - from fee percentages, to how fees are paid and accrued, to which Oracle nodes are used, to the number of blocks in the reorg protection procedure - is decided upon by the holders. The protocol is also programmed to allow token holders to set

the ability to pay fees with the Morphswap governance token itself for a discount. The owners of the Morphswap governance token are the owners of the protocol.

- **Liquidity Provision Tokens:** upon providing an asset as liquidity to a pool, the providing user receives tokens that signify the proportion of the pool they are entitled to. These tokens are not tied to the account that provided the liquidity, and can be traded, staked, or otherwise treated like any ERC-20 compliant token. To get liquidity back from a pool, these tokens may be redeemed to their respective pool.
- **Impermanent Loss:** a side-effect of the concept and algorithm of automated market-making. In an AMM, the users who provide liquidity experience this effect. An example: *a user provides amounts of Token A and Token B to an AMM pair, at ratios proportional to current liquidity ratios of the pool (a provision of 10 Token A and 1 Token B, when the liquidity amounts of the Token A / Token B are at 10,000 to 1,000, respectively). Then, over time, the price of Token A increases to be equal in value to Token B. The user would have more Token B than they initially provided to the pool, and less Token A (around 3 of Token B and 3 of Token A). They still benefitted - nominally - assuming all other factors are equal, but not as much as they would have by simply holding their 10 Token A. Conversely, if it was Token B falling in value to become equal in price to a stable Token A, holding liquidity of theoretical pool, where Token A is a value-stable asset, and Token B was a speculative asset, the user hedged their exposure by limiting the leverage of the decline.* Effectively, providing liquidity is a way of gaining exposure to a speculative asset at 0.5x perpetual leverage in terms of the price of the paired token.
- **Asymmetrical Gain:** an effect specific to the Morphswap protocol; a necessary side-effect of the inablitiy for the blocks of two otherwise independent chains to arrive at finality at synced times with entangled (dependent) degrees of confidence. Put simply, it is not possible, in a decentralized way, to interact with two chains in the same way at the same time with full certainty that they will

either both succeed or both fail. The countermeasures from navigating this limitation have resulted in liquidity being applied to different sides of the same pool at different times. This can result in liquidity being applied completely unevenly, which can be very beneficial to one side of the pool. If one side of a pool keeps getting liquidity provided to it more (possibly to offset liquidity flowing towards the asset that makes up that side of the pool), the opposite side will increase in value without any decrease in the proportional ownership of pool liquidity for the owners of that side of the pair's liquidity. This can intuitively be thought of as "anti-dilution".

- **Asymmetrical Loss:** the counter-side of asymmetrical gain. If liquidity is being provided mostly to one side, that side of the asset pool will become increasingly diluted.
- **Internal Chain ID:** the unique number given - by the Morphswap protocol's 0-indexed 8-bit internal chain reference system - to each chain supported by Morphswap, in the chronological order they were added to the protocol. Originally to reduce the amount of write-to-chain data and cost of repeatedly encoding arguments on chain, as there are many chains with unnecessarily large ID numbers(allowing Morphswap to be compatible with any chain while also replacing the need for 256-bit unsigned integers with 8-bit integers).

Addendum:
Non-EVM Support

In this addendum to the whitepaper, we will go over the technology implemented post-launch that completes the architecture of the Morphswap protocol enabling the support of Non-EVM chains. Specifically, the term “Non-EVM chain support” can be thought of as merely a shorthand for chains that::

A) Do not have Chainlink Operator Node support
B) Do have hierarchical deterministic multi-signature wallet support

Infrastructure Architecture

On the simplest conceptual level, the infrastructure enabling the facilitation of trustless swaps involving non-EVM chains - the technology outlined in this addendum - can be thought of as simply an intermediate, interpretation layer between the core protocol and the non-EVM chain being supported.

On a slightly deeper level, this intermediate, interpretation layer facilitates the mapping of the subject chain to a format and location accessible by the core Morphswap protocol deployment, to be retrieved in the same fashion as Morphswap retrieves the data from any other chain. The components required for the efficient execution of this concept are:

- An EVM chain to hold and structure the mapping of this purposed-for-retrieval data
- An algorithm to discriminately filter the subject chain’s data
- A vehicle to write the data to the chosen EVM chain
- A system to hold the assets of the subject chain that are provided as liquidity

In theory, this likely doesn’t sound inhibitively difficult or complex when compared with existing technology in the blockchain industry. To do it in a centralized fashion may be considered clever, but not groundbreaking. The challenge - and the reason that it has previously been so difficult to bring non-EVM currencies into DeFi - is the requirement to do this in a trustless, decentralized fashion, in a way both generalized enough to support any chain with hierarchically deterministic multi-signature wallet support, while being engineered to communicate the necessary data in a way that is decentralized and trustless. The following several pages aim to be elucidatory on the conceptual overview of the architecture for each of these components.

The first option encountered - which EVM chain to choose for the mapping of the data - is a simple decision to make. As this data exists to be retrieved by the central deployment, it makes the most sense for this chain to be the same chain that our central deployment is on, otherwise known as the central chain. The next design requirement is the structuring and holding of this data. As the Morphswap protocol inherently has a system enabling support of peripheral chains, we can simply put a peripheral chain deployment - a standard contract deployment deployed as being for a peripheral chain - on the chain which is home to our central node. The central chain is the natural decision as not only is there no barrier to data-transmission, but also there is no need to protect against block re-organizations causing a de-syncing of data. Additionally, the initial reasons which make a chain an auspicious selection for a central chain are also applicable to house the peripheral chain contract mapping the data for the supported non-EVM chain.

In order for a standard Morphswap protocol contract deployment, there need to be a few alterations to an otherwise boilerplate version. The collection of significant changes consist of::

- Tracking of a gateway address to send hexadecimal-formatted data the Morphswap Nodes
- An altered liquidity pool contract
- A series of functions for which the parameters cover all relevant data when combined, which submit the arguments to Morphswap nodes upon invocation,
- A contract that can serve as a stand-in for the peripheral chain native-currency, a simulacrum of Bitcoin on the subject chain
- An accommodation to the utilizations of a stand-in contract

The peripheral mapping contract needs to track a destination address to be used to communicate data with the nodes. Otherwise, data-transmission would be unidirectional, and although non-EVM node integration is technically possible while being inhibited by unidirectional communication, it needlessly compounds the complexity. For our purposes, the contract pointed to by the tracked address can be thought of as the chain-side implementation of the intermediate interpretation layer. In reality, the execution of the chain-side implementation actually involves two separate contracts: a contract purposed for post-interaction and a contract purposed for pre-interaction. Regardless, the solution for multiple addresses pointing to contracts is the same solution as just one contract, only incurring the additional complexity of segregating the calls to their respective contracts before data-transfer or external function invocation.

Subsequently, the liquidity pool contracts have to be changed to track the non-EVM chain’s native coin, an asset not natively on the subject chain used for the mapping. Since we will have a contract that facilitates the function of the peripheral deployment through tracking the balances attributable to different addresses, the changes required to be made to the liquidity pool contracts revolve around both enabling functionality with this facilitatory contract, as well as communicating relevant data to the chain-side of the interpretation layer, primarily the post-Interaction contract. The data required to be submitted to the post-interaction contract will be covered in proper detail later in this addendum.

To give complete coverage for data-transmission to Morphswap nodes, the altered peripheral contract needs to submit data to both the contract constituting the pre-interaction segment, as well as the contract constituting the post-interaction segment. Data for whitelisting pools to make state-changing external function calls to the post-interaction contract must be sent to the post-interaction contract by a previously trusted source, which is the core contract of the peripheral deployment. Primarily, the external functionality that requires whitelisting is the submission of outbound BTC transactions to be sent out to an end-user’s non-EVM chain receiving wallet. Lastly, there needs to be a function that dereferences map keys for RIPEMD160 hashes of SHA256 hashes of external recipient addresses. The necessity for this will be explained later.

The contract that needs to exist to ensure continued functionality despite these changes, and the absence of an asset, of this chain-mapping format is effectively only a contract that can track balances of certain addresses, and allow the subtraction and addition of the balances of each address entry. This may sound like a simple, rudimentary version of token, and it can be thought of as a private, simplified version of the most simple token contract. However, as there are many functionalities this facilitatory contract lacks compared to a token contract, specifically an ERC compliant one. There are no emitted events, allowances, and so on, as it is really simply a permissioned “address to unsigned integer” hashtable.

The accommodations for the pseudo-coin contract manifest notably in the form of having the tip pool (pair ID: 100x) be a non-native-coin pool, and therefore this causes cascading need for changes in the “buy” delegate contract, as there will be no need in a delegate contract for buying with a native coin. Firstly, some of the custom alterations involved in changing the primary tip pool involve changes to the peripheral-side process of initializing the chain in the protocol. These changes are relatively minor, and in the form of having the relevant delegate contract deploy a pool utilizing the otherwise independent contract facilitating Morphswap contract operation by tracking the coin holdings of the protocol on-chain.

Data filtering

It is at this point where the low-level strategies for accommodating UTXO-based BIP32 blockchains, and CryptoNote-powered blockchains bifurcate. UTXO-based blockchains, such as Bitcoin, serve a massive amount of data (relative to the amount of an end-user’s relevant data) to a node, which then must be indexed and sorted through to harvest the meaningful data. A CryptoNote blockchain however, as it can be considered more “account-based”, is fairly simple for the data filtering. For the sake of brevity, this paper will cover the strategies for acquiring this data in terms sufficiently broad that it adequately covers both UTXO-based blockchain as well as blockchains powered by CryptoNote. Although the differences for data-filtering will be overlooked in this paper, we will later delve into the strategic accommodation differences with regards to inter-node consensus and keeping a canon record of accounts, in their respective sections.

The data we need to communicate with the pseudo-peripheral chain deployment is composed of multiple parts. We need to know everything that a typical meta-transaction needs as parameters to sufficiently populate a txObject data structure:

```
struct txObject {

uint8 method_id;

uint8 internal_start_chainid;

uint8 internal_end_chainid;

uint64 pair_id;

address finalchain_wallet;

uint64 secondpair_id;

address firstchain_asset;

address finalchain_asset;
uint64 quadrillionratio;

uint64 quadrilliontipratio;

uint128 rtxnum;

bool alt_fee;

}
```

This data is sent to the pseudo-peripheral chain deployment indirectly, via Morphswap nodes. A node will receive this data from a user prior to sending their Bitcoin. This data which is given to the Morphswap nodes, is effectively mapped to a future transaction. There are naturally two ways to handle this. One of which would be mapping the data to the transaction input's previous transaction output. This method can be thought of as mapping a transaction to the sender. The other method is to map these future meta-transaction parameters to a transaction output address. There is also an option of doing niether, and having the user include additional data in the block that can be deciphered into the parameters. The last option is not doable, as it would be relatively simple on the protocol's end, it is very difficult for most users to know how to write arbitrary data to non-smart chains, and in some cases not feasible. So that leaves two reasonable options; mapping to transaction input, and mapping to transaction output.

Mapping to transaction input is relatively simple when compared to mapping to transaction output. However, deciphering the sender from a list of previous transaction outputs poses potential issues. Foremost, although there are techniques that generally produce an accurate answer, there are exceptions. For example, going off of "prevTx.outs[0]", the first output of the previous transaction, although generally yielding the sender, occasionally produces unexpected results for non-standard Bitcoin transactions (e.g if a different address is listed as the transaction’s first output). Second, it greatly increases the chance of user error, as the average user doesn’t really understand how UTXO-based blockchains work, and may be unaware that their Bitcoin may be sent from a different address than they originally sent it to. Most importantly, CryptoNote-powered chains, although account-based, are generally private, making it impossible to know where the transaction was sent from. So, in the interest of broad non-EVM chain support, this is not an option.

This brings us to our most complex option: mapping to transaction outputs. Transaction outputs are the simplest thing for any end-user to dictate, even more so than the amount to send, due to transaction fees. The natural issue with encoding data in transaction outputs, is that any value sent to them can not be retrieved by any entity who does not know that address’s private key. So, if by the process of elimination, the protocol is going to use the information encoded into transaction outputs to work out the previously mapped data, there are two methods. One option is to have the user send a negligible amount of Bitcoin to an address that, in adherence to some previously decided upon “unsigned integer to BEC32 address” system before sending to a wallet the protocol has decentralized custody of. The other option is significantly more complex for the protocol to achieve: mapping the parameters to a P2SH address that the protocol has control of.

With a dynamically increasing list of P2SH addresses, the nodes can make sure they always have a sufficient amount of P2SH addresses to be available at any time. It is important to note that, for the sake of efficiency, each address is set to be used again, following one of these two conditions being satisfied:

1. The given P2SH address has a UTXO
2. There has been a sufficient amount since mapping (tracked by blocks passed on the central chain)

Writing data to the central chain

Once each node sees a new UTXO for one of P2SH addresses in the dynamically-sized list of maintained addresses, it communicates that with the other nodes. It is important to note the precautions taken to prevent exploits. The list of currently serving nodes is immutable and trustless, with the central-chain interface contract serving as the canonical truth. Additionally, each node communicates its RSA public keys, and encrypts all data using the node’s respective public key.

Each node attempts to write data to the interface contracts as soon as any UTXO for a maintained P2SH address has 1 confirmation. The interface contract then coordinates the path from that point.

The nodes specifically communicate an address string concatenated with the transaction hash string as one argument, the address string as its own argument, and the amount of sats in the UTXO. This allows each received transaction to be unique, preventing inadvertent duplicate write attempts from ever registering.

If >50% of the nodes pass in a certain sats amount for a given transaction hash, the interface contract then acts as a proxy buyer, using the facsimile-ERC20 contract that tracks BTC being mapped over the EVM chain.