Architecture

Mozaic System

Development use

mike@mozaic.finance

Table of Contents

Architectural Decisions

- Mozaic, the system, and Mozaic system refers to the software system that this project is going to develop, launch, operate, and maintain.
- This document describes architectural decisions that implement the system requirements.
- This document serves as an additional technical terminology of the project.
- This document is shared and maintained by team members.

1. Definitions and assumptions

- **System Asset Snapshot**, **system asset state**, **system asset/request state**, or simply **asset state**, is the state identified by the followings, at a given time:

- **Deposits**: The vector of all deposit requests. A deposit request can be modeled by:

```
struct Deposit {    // sends assets to the system and receives mLP in return
    address user;       // the user who requests the deposit
    address to;         // wallet to receive mLP
    address token;      // the denominator token of the assets
    uint    token_chain // the chain that hosts the denominator token
    uint    amount;     // the amount of the assets in the denominator token
    uint    amountLP;   // the amount of mLP
    uint    lp_chain;   // the chain that hosts the mLP token
    uint    usd_equ;   // the usd-equivalent of the asset amount
}
```

- **Withdrawals**: The vector of all withdrawal requests. A withdrawal request can be modeled by:

```
struct Withdraw {   // sends mLP to the system and receives assets in return
    address user;       // the user who requests the withdrawal
    address to;         // wallet to receive assets
    address token;      // the denominator token of the assets
    uint    token_chain // the chain that hosts the denominator token
    uint    amount;     // the amount of the assets in the denominator token
    uint    amountLP;   // the amount of mLP
    uint    lp_chain;   // the chain that hosts the mLP token
    uint    usd_equ;   // the usd-equivalent of the asset amount
}
```

- **Stakes**: The vector of all staking instances. A staking instance can be modeled by:

```
struct Stake {   // a stake of asset is staked on a pool
    address token;      // the denominator token of the assets
    uint    token_chain // the chain that hosts the denominator token
    uint    amount;     // the asset amount in the denominator token
    uint    pool_id;    // the local/global pool ID
    uint    usd_equ;   // the usd-equivalent of the asset amount
}
```

- **Rewards**: The vector of all reward instances. A reward instance can be modeled by:

```
struct Reward {   // a reward is pending collecting on a pool
    address token;      // the denominator token of the assets
    uint    token_chain // the chain that hosts the denominator token
    uint    amount;     // the asset amount in the denominator token
    uint    pool_id;    // the local/global pool ID
    uint    usd_equ;   // the usd-equivalent of the asset amount
}
```

- **Treasury**: Assets reserved for system operation, development, and management. Treasury consists of TreasuryItems. A treasury item can be modeled by:

```
struct TreasuryItem {
    address token;      // the denominator token of the assets
    uint    token_chain // the chain that hosts the denominator token
    uint    amount;     // the amount of the assets in the denominator token
    uint    source;     // the source of treasury item, like performance fee, etc.
    uint    usd_equ;   // the usd-equivalent of the asset amount
}
```

- **Pools state**
  $PoolState = (\ PS_i \mid i = 1..N\ )$,
  where PS_i is the i-th pool state:
  $PS_i = (RewardRate_i, RewardToken_i, TotalStake_i, StakingToken_i, MozaicStake_i, Price_i^R, Price_i^S)$
  , where
  - $RewardRate_i$ is the amount of reward emitted during a given time frame
  - $RewardToken_i$ is the *user-visible* token type of reward
  - $TotalStake_i$ is the total amount of staked asset in the pool
  - $StakingToken_i$ is the *user-visible* token type of staking asset
  - $MozaicStake_i$ is the amount of asset staked in the name of Mozaic
  - $Price_i^R$ is the *average* price of reward token in the time frame
  - $Price_i^S$ is the *average* price of staking token in the time frame

- **Staking tokens**

$$StakingTokens = (\ StakingToken_i \mid i = 1..N\ )$$

- **Regularity of staking pool**
  We assume reward on PS_i is calculated as:

$$MozaicReward_i = \frac{RewardRate_i}{TotakStake_i} \times MozaicStake_i$$

2. Omnichain mLP token

2.1. Requirements

Omnichain mLP requires that:

- The mLP token should exist on all listed chains.
- When the system **stake**s assets that a user **deposit**ed, the system returns mLP tokens to the user. The amount of the returned mLP token should represent the newly staked asset in the **Staking Stock** immediately after the asset is staked.
- A user can **withdraw** assets from the **Staking Stock**, in any listed token format on any listed chain, by first returning mLP tokens from their wallet to the system wallet. The amount of asset that is **withdraw**n is the portion of **Staking Stock** that is represented by the returned mLP tokens immediately before the asset is **withdraw**n.

Additional requirements:

- The mLP token cannot have initial supply

  This is to ensure that mLP has no features of security.

2.2. Design decisions

- mLP token contracts will be independent of vaults, except that local vaults should be able to mint and burn local mLP tokens
- mLP token contracts will be independent of administration
  - mLP tokens will be completely free from administrator or DAO.
  - mLP tokens will not be upgradeable and rebased.
- mLP tokens will be cross-chain swapped 1:1
  - A 3rd party cross-chain transportation services will be used
  - Mint-burn mechanism will be used
  - Lock-release will not be used
- mLP token swap will be either completely successful or completely reverted on both the source chain and the destination chain
  - It will employ the same technique as Stargate's swap, **if we find no alternatives**.

3. Omnichain vaults

We need a consolidated, omnichain module that implements the use case **Control asset move** identified in the requirements specification, solely and completely. We call the module the vault, because from users' perspective,

- the module keeps users' assets, control and logs moves of the assets and profits generated from the assets, and returns the assets with profits.
- no modules other than that module have privilege to carry out these tasks.

3.1. Omnichain vault as smart contracts coordinated across chains

According to the requirements, vaults are exclusively responsible to de-centrally

- make all changes to **system assets**
- log all changes to **system assets**

The only way is to have smart contracts on chains cooperate with each other to form the omnichain omnichain vault module. We call them local vaults or vault contracts individually.

3.2. Limited responsibility

According to the requirements, vaults do *not* have to

- find the *best possible* asset state to **optimize asset/request state** to
- precisely execute asset staking requests coming from off-chain side, like transitionPlan identified in requirements, because there will not be negative profit

3.3. Local transportation to/from wallets

Local vaults are responsible to:

- Withdraw pending rewards
- Stake and un-stake on local staking pools
- Send/receive assets to/from other local vaults

- Pull assets from the user if a deposit request involves a home token

- Export a deposit request if it involves an away mLP token
- Import an exported deposit request if it involves a home mLP token
- Calculate and push mLP tokens to the user if a deposit request involves the home mLP

- Pull mLP to the user if a withdrawal request involves the home mLP token
- Export a withdrawal request if it involves an away token
- Import an exported withdrawal request if it involves a home token
- Calculate and push asset to the user if a withdrawal request involves a home token

3.4. Reference source code adding requests

```
pragma solidity ^0.8.0;

// imports
import "../libraries/lzApp/NonblockingLzApp.sol";
import "../libraries/stargate/Router.sol";
import "../libraries/stargate/Pool.sol";
import "./MozaicLP.sol";

// libraries
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";

abstract contract SecondaryVault is NonblockingLzApp {


    // The caller submits _amount of _token, and wants MLP tokens on _chain.
    function addDepositRequest(address _token, uint _amount, uint _chain) external  {
        require(userTokens[_token] != 0 && _amount > 0, "Wrong token/amount");
        _safeTransferFrom(_token, msg.sender, address(this), _amount);

        if (_chain == thisChain) { // The request is local-token for local-mLP
            pending.ds.push( Deposit(msg.sender, _token, _amount, 0) );
            // 0 for mLP amount to send to the user.
        } else { // The request is local-token for away-mLP
            pending.dsToExport.push(DepositToExport(msg.sender, _usdt(_token, _amount), _chain));
            // Foreign chain _chain will store this like:
            // pending.dsImported.push(DepositImported(msg.sender, usdEq, 0));
            // 0 for the undefined mLP amount to send to the user
        }
    }

    // The caller submits _amount of mLP, and wants _token tokens on _chain chain.
    function addWithdrawalRequest(uint _amountLP, address _token, uint _chain) external  {
        require(userTokens[_token] != 0 && _amountLP > 0, "Wrong token/amount");
        _safeTransferFrom(mLP, msg.sender, address(this), _amountLP);

        if (_chain == thisChain) { // The request is local-LP for local-token
            pending.ws.push( Withdrawal(msg.sender, _token, 0, _amountLP) );
            // 0 for the undefined amount of token to send to the user.
        } else { // The request is local-LP for away-token
            pending.wsToExport.push(WithdrawalToExport(msg.sender, _token, _amountLP, _chain));
            // Foreign chain _chain will store this like:
            // pending.wsImported.push(WithdrawalImported(msg.sender, _token, 0, _amountLP));
            // 0 for the undefined amount of token to send to the user
        }
    }
}
```

3.5. Vaults identified

We identify vaults through their surrounding modules interacting with them.
The external actors in the following use case diagram, together with their interactions with vaults are already described. We

can now explore the use cases of vault.

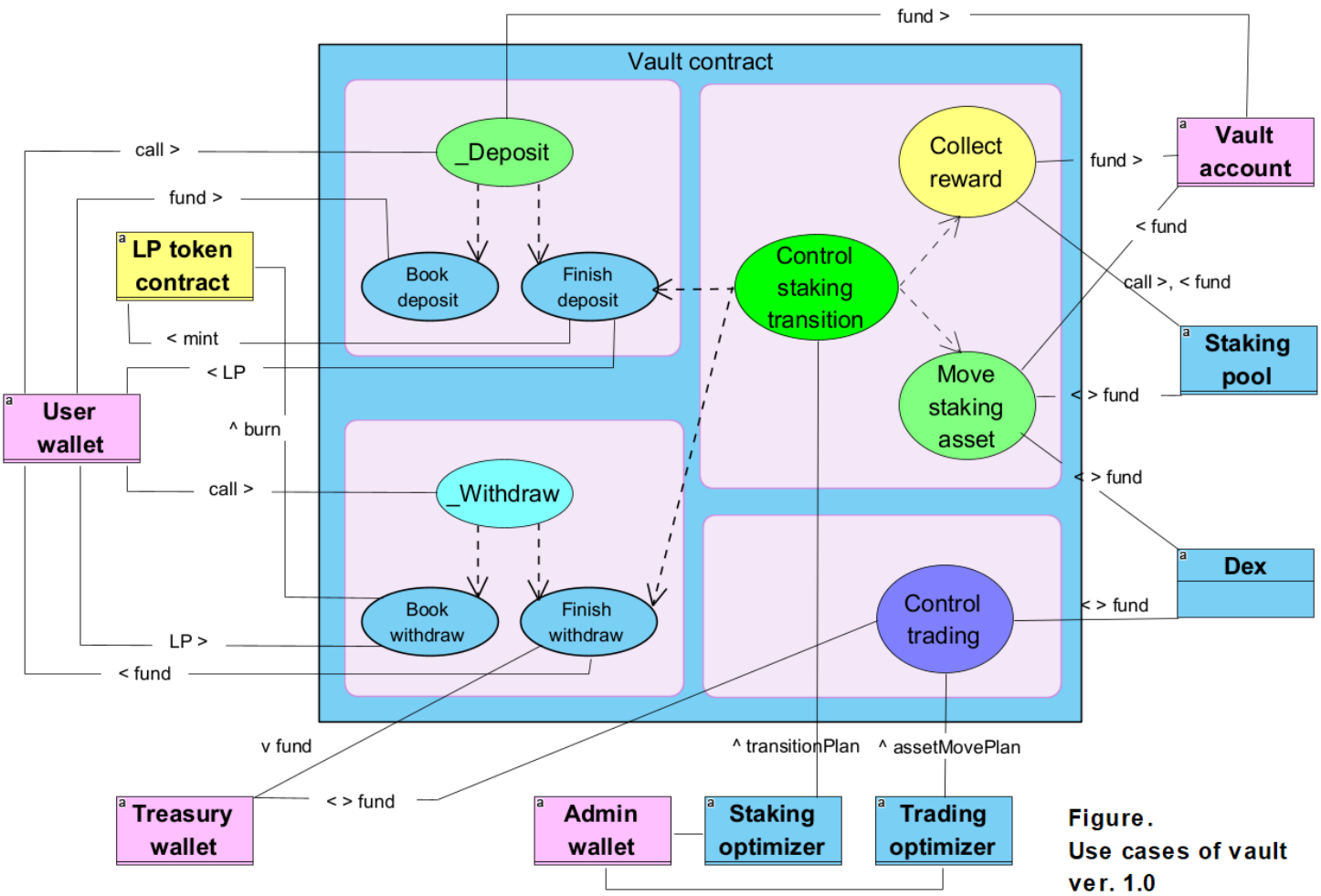

Figure.
Use cases of vault ver. 1.0

- **_Deposit**: This happens at vault contracts when the **Deposit** use case is invoked at the system. Invoked by the **User wallet** with assets ready to deposit, this use case coordinates the following two use cases.
- **Book deposit**: This use case
  - collects the assets from **User wallet**,
  - books the deposit request with the system,
  - and pauses the session of "_Deposit".
- **Finish deposit**: This use case
  - resume the session of "_Deposit",
  - retrieves the booked deposit request,
  - mints mLP tokens to cover the new assets,
  - returns the mLP tokens to **User wallet**,
  - and helps **Control staking transition** stake the assets.
- **_Withdraw**: This is what happens at the level of vault contracts when the **Withdraw** use case is invoked at the system level. Invoked by **User wallet** with mLP tokens returned, this used case coordinates the following two use cases.
- **Book withdraw**: This use case
  - collects the returned mLP tokens,
  - burns the collected mLP tokens,
  - books the withdrawal request with the system,

    - and pauses the session of "_Deposit".
- **Finish deposit**: This use case
    - resume the session of "_Deposit",
    - retrieves the books withdrawal request,
    - subtract assets, as much as covered by the returned mLP tokens, from the total system assets, and
    - returns the assets to **User wallet**
- **Control staking transition**: This use case transitions to a new asset state by executing **transitionPlan** provided by **Staking optimizer**. (This is the most challenging part of vault implementation.) It does *collectively*, by using **Move staking asset**,
    - **Collect reward**,
    - Collect staked assets, to cover the assets to **Finish withdraw**,
    - **Finish deposit**,
    - **Finish withdraw**,
    - execute remaining part of **transitionPlan**
- **Control trading**: This use case executes **assetMovePlan** provided by **Trading optimizer**.

3.6. Logical components

The overall architectural requirements for vault was/is to **minimize vault as much as possible** leaving most compute to off-chain modules.

The design decisions are as illustrated in the following figure:

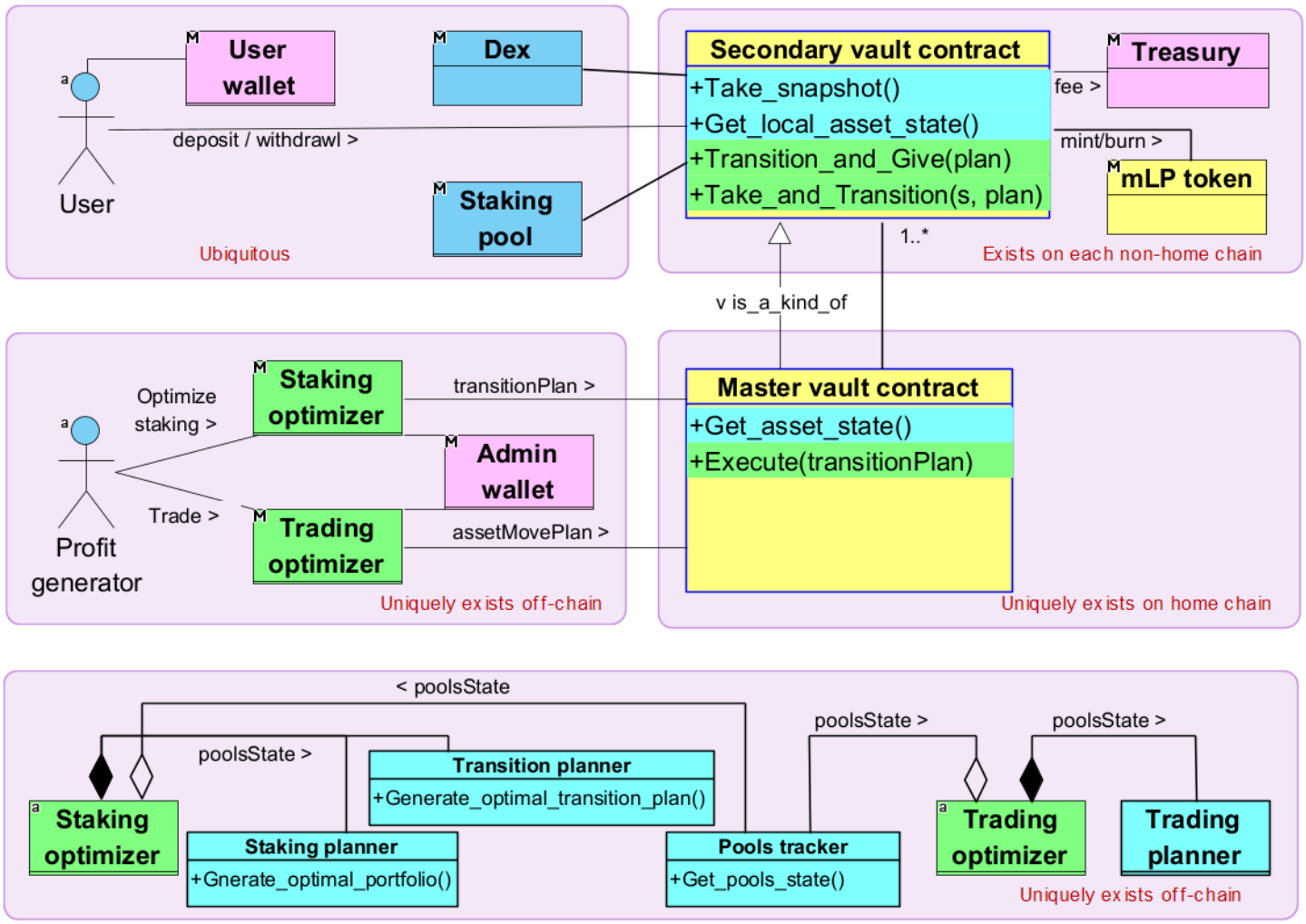

Figure. High-level functional modules of Mozaic DeFi ver. 1.0
Goal: Minimize the functionality of vaults and cross-chain communications.

Functional modules are described below:

- **Secondary vault contract**: This module is a local vault and deployed on each chain except the home chain.
- **Master vault contract**: This module is a special **Secondary vaults contract** and deployed on the home chain.
- **mLP token contract**: This module manages the mLP token balances of **User**s.
- **User wallet**: This is a blockchain wallet and identifies a **User**. **User**'s actions; like deposit and withdraw are authenticated/authorized with this wallet.
- **Treasury**: This is a blockchain wallet or a contract account, and a place to store and retrieve system treasury. It will be better if it is the account of a smart contract that only obeys vault contracts, for better decentralization.
- **Staking optimizer**: This is an off-chain module that can invoke **Master vault contract**s. This module is globally unique, calculates optimal **transitionPlan**s, and lets the master vault to execute the plans (in cooperation with secondary vaults).
  - Transparency debate: **User**s will not be able to track why the system chose particular **transitionPlan**s technically.
  - Security debate: If the calculation of **transitionPlan** is hacked or compromised, then the system will make a less-optimal staking maneuver.
  - Justification: Only the second of the following concerns becomes less transparent, leading to both un-assured best profitability and assured huge gas- and time- savings.
    - how much of what assets from which pool to which pool, is the move about
    - whether all the asset moves are securely and/or reasonably/optimally chosen

        - whether all the asset moves are securely executed and logged
        - whether the move logs are readily available to check later
        - whether the execution of transitionPlan is integrated
- **Trading optimizer**: This off-chain module is similar to **Staking optimizer**, except it relates to trading.
- **Adimin wallet**: This wallet is used to invoke **Master vault contract", in privilege, on behalf of the administrator.
- **Staking planner**: An integral component of **Staking optimizer**, this module predicts the next most profitable **staking_portfolio**, based on **poolsState** provided by **Pools tracker**. Running this module on-chain would enhance transparency, but would at the same time incur huge gas fees and effectively disable the system.
- **Transition planner**: An integral component of **Staking optimizer**, this module predicts the most efficient **transitionPlan**, which is the best procedure of asset move that implements the transitioning to a given **staking_portfolio**, based on the current **poolsState**.
- **Trading optimizer**: This is similar to **Staking optimizer**, except that it relates to trading.
- **Trading planner**: This is similar to **Staking planner**, except that it relates to trading.
- **Pools tracker**: A shared module between **Staking optimizer** and **Trading optimizer**, this module retrieves and tracks all relevant information from chains, like Reward Release Speed, and total Staked mLP of each pool. Running this module on-chain would enhance transparency, but would at the same time incur huge gas fees and effectively disable the system.

4. Omnichain staking, or Cross-chain optimization

4.1. Definition

Omnichain staking requires that:

- Assets can be deposited in any listed token format on any listed chain, *all of users' choice*.
- Deposited assets can be swapped/transferred, and staked in any staking pool on any listed chain, *guided by the system's optimization plan*.
- Staked assets and rewards can be withdrawn in any listed token format on any listed chian, *all of users' choice*.
- Rewards collected can be swapped/transferred, and staked in any staking pool on any listed chain, *guided by the system's optimization plan.*

4.2. Staking planner

Note. All errors, like numerical processing rounding and price slippage, are ignored at this stage of architectural design.

4.2.1. Task definition

- Goal: Calculate the best staking portfolio off-chain in order to
    - Save vault contracts long calculations of staking optimization, thus to save gas.

    - Keep vault contracts insulated from future algorithm upgrades of staking optimization.
- Consideration
    - Input may not be idealistically consistent, because an idealistic snapshot of multiple chain states is impossible logically.
    - Output staking portfolio may not be completely/perfectly implemented, because input may have errors and there are unpredictable price slippages and change of fees.

- Input
    - Current **system asset/request state**
    - Current poolsState, defined below

- Output
    - optimal staking portfolio

4.2.2. Notations

- **Asset vectors**
  For the **system asset/request state** snapshot taken just before the transition t, we can deduce the following asset and token vectors through *one-to-one* mapping in the same order.
    - **Deposit assets**, at transition index t
      $Deposits^t = (\ (D_i^t,\ ,T_i) \mid D_i^t :\ deposited\ amount.\ T_i :\ token.\ )$
    - **Withdrawals assets**, at transition index t
      $Withdrawals^t = (\ (\ -W_i^t,\ T_i) \mid W_i^t : withdrawal\ amount,\ T_i :\ token.\ )$
    - **Vector of collected rewards**, at transition index t
      $Rewards^t = (\ (R_i^t,\ T_i) \mid D_i^t : reward\ amount,\ T_i :\ token.\ )$
    - **Vector of staked assets**, at transition index t
      $Stakes^t = (\ (S_i^t,\ T_i) \mid D_i^t : stake\ amount,\ T_i :\ token.\ )$
- **Transformations**
    - **Transformation USD^{+1}**
      USD^{+1} transforms an asset vector to USD-denominated asset vector.
      Example: USD^{+} transforms (2 USDC, 3 ETH) to (2.02, 1300), assuming USD/USDC = 1.01 and USD/ETH = 1300.
    - **Transformation USD^{-1}**
      USD_{TK}^{-1} transforms a USD-denominated asset vector and a token vector TK to tokens-denominated vector.
      Example: $USD_{(USDC,ETH)}^{-1}$ transforms (2.02 USD, 1300 USD) to (2 USDC, 3 ETH), assuming USD/USDC = 1.01 and USD/ETH = 1300.
    - **Transformation FOP** - the core of the Archimedes algorithm
      FOP, standing for Find Optimal Portfolio, finds the *best* USD-denominated **vector of staked assets** for a given total USD amount. In other words, *it finds what amounts of (USD-denominated) value should be allocated to what staking pools, provided that the total (USD-denominated) value is given*. **best** is relative and subjective.
    - **Transformation Sum**
      Sum sums up all elements of a vector when they are denominated by the same token.

- **Asset snapshot**, at transition t

- Asset snapshot just before the transition t:
  $AS^t = (Deposits^t,\ Withdrawals^t, Rewards^t, Stakes^t)$
  Or, simply, $AS^t = (D^t,\ W^t, R^t, S^t)$
- Asset snapshot just after the transition t:
  $AS^{t+} = (0Deposits^t,\ 0Withdrawals^t, 0Rewards^t, optimal\ Stakes^t)$
  Or, simply, $AS^{t+} = (0D^t,\ 0W^t, 0R^t, optimal\ S^t)$
  , where 0D, 0D, and 0R are a vector of zero valued elements of their respective types.

4.2.3. Formula

If a state transition t is optimal, the following diagram holds:

$$
\begin{array}{ccc}
AS^t=(D^t, W^t, R^t, S^t) & \xrightarrow{(Resulting\ transition)} & AS^{t+}=(0D, 0W, 0R, optimal\ S^t) \\
{\scriptstyle USD^{+1}}\downarrow & & \uparrow{\scriptstyle zeros,\ USD^{-1}_{StakingTokens}} \\
AS^t_U=(USD^{+1}(D^t), USD^{+1}(W^t),...) & \xrightarrow[(Implicit)]{} & AS^{t+}_U=(0D, 0W, 0R, FOP(Total\ in\ USD)) \\
{\scriptstyle Sum}\downarrow & & \uparrow{\scriptstyle zeros,\ FOP} \\
Total\ in\ USD & \xrightarrow{Identity} & Total\ in\ USD
\end{array}
$$

The algorithm for Staking planner AS^t is a chain of transformations:

Optimal Staking Portfolio =

$$optimial\ S^t = USD^{-1} \circ FOP \circ Sum \circ USD^{+1}(T,\ D^t,\ -W^t,\ R^t,\ S^t)$$

- USD^{+1} and USD^{-1} are obvious, except that we may need systematic methods to find best Dexes and swap paths.
- An analytical version of FOP demonstrated 9% of competitive edge over the public. *A Machine Learning version should give higher edge.*
- Sum is trivial.
- D^t and W^t can be retrieved from the booked requests of deposits and withdrawals.
- S^t is found when we "Collect reward" pending rewards.

The formula essentially does:

- Sum up all asset amounts available for the new staking:

```
= the **Staking Stock** (i.e. staked assets plus pending rewards),
+ assets received from deposit users,
- assets to send to withdrawing users.
```

- Find the USD-equivalent of the sum assets: Total_in_USD,
- Allocate the Total_in_USDT **optimally** across all staking pools, by using the FOP algorithm, Note: **optimally** is relative and subjective.
- Transform the allocated USD amounts back to the native tokens on the staking pools.

4.3. Transition planner

4.3.1. Regular asset move plans

An asset move plan is a set of elementary asset move instructions. We need to eliminate redundant value flows from asset move plans to save the cost of executing the plan.
A regular asset move plan as a plan that has no redundant value flows. **For any asset move plan, there exists a regular equivalent of the original plan. It should be unique(?) and easy to find if we introduce an external asset place as the hub.**

Below comes two example of asset move plan: an irregular plan and its regular equivalent:

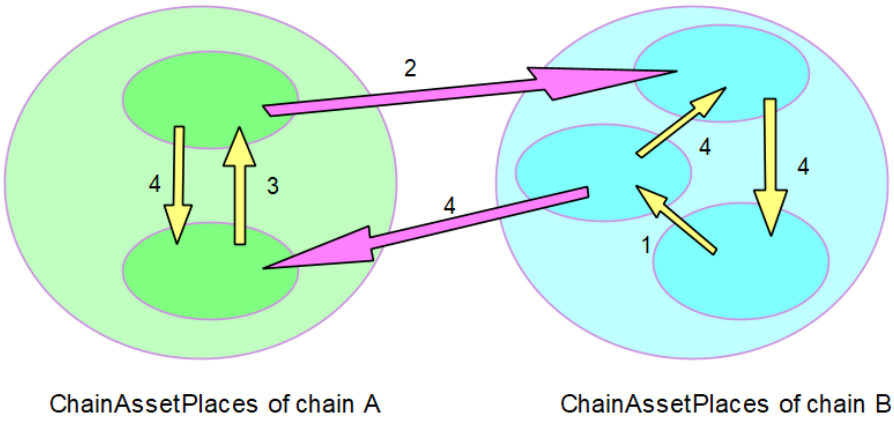

Figure. Irregular asset moves. (swaps are hidden)
Intra-Chain Moves and Inter-Chain Moves have cyclical value moves.
Some asset places both give and take, for example.

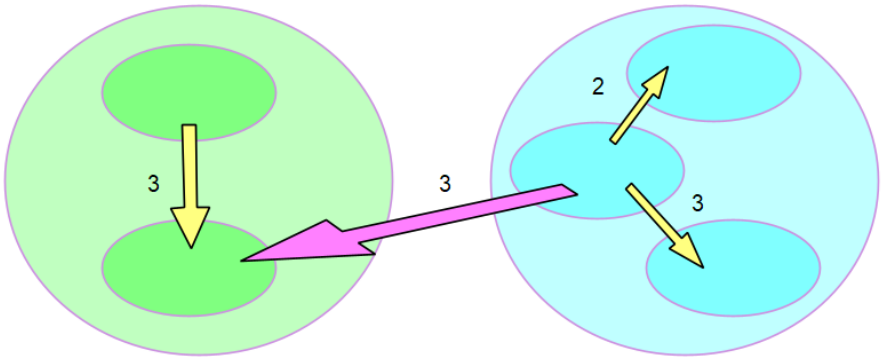

Figure. Regular asset moves. (swaps are hidden)
Intra-Chain Moves and Inter-Chain Moves have no cyclical value moves.
Any group of asset places cannot both take from and give to outside places.

4.3.2. Definitions and notations

- **Asset vectors**
  For the **system asset/request state** snapshot taken just before the transition t, we can deduce the following asset and token vectors through *one-to-one* mapping in the same order.
  - **Deposit assets on a given chain**, at transition index t
    $Deposits_c^t = (\ (D_i^t,\ ,T_i)\ |\ D_i^t:\ deposited\ amount.\ T_i:\ token\ on\ chain\ c.\ )$
  - **Withdrawals assets on a given chain**, at transition index t
    $Withdrawals_c^t = (\ (-W_i^t,\ T_i)\ |\ W_i^t : withdrawal\ amount,\ T_i:\ token\ on\ chain\ c.\ )$
  - **Vector of collected rewards on a given chain**, at transition index t
    $Rewards_c^t = (\ (R_i^t,\ T_i)\ |\ D_i^t : reward\ amount,\ T_i:\ token\ on\ chain\ c.\ )$
  - **Vector of staked assets on a given chain**, at transition index t
    $Stakes_c^t = (\ (S_i^t,\ T_i)\ |\ D_i^t : stake\ amount,\ T_i:\ token\ on\ chain\ c.\ )$
  - **Vector of assets on a given chain**, at transition index t

- Asset snapshot just before the transition t:
  $VA_c^t = (Deposits_c^t,\ Withdrawals_c^t, Rewards_c^t, Stakes_c^t)$
  Or, simply, $VA_c^t = (D_c^t,\ W_c^t, R_c^t, S_c^t)$
- Asset snapshot just after the transition t:
  $VS_c^{t+} = (0Deposits_c^t,\ 0Withdrawals_c^t, 0Rewards_c^t, optimal\ Stakes_c^t)$
  Or, simply, $VA_c^{t+} = (0D_c^{t+},\ 0W_c^{t+}, 0R_c^{t+}, optimal\ S^t)$
  , where 0D, 0D, and 0R are a vector of zero valued elements of their respective types.

4.3.3. Design decisions

We deduce the following design decisions:

- Asset moves will be **regularized**. We believe regularization will reduce the total cost of asset moves and the number of inter-chain swap/transfers.
- **ChainValutWallet** will be the hub for regular **Intra-Chain Moves**. This means an **Intra-Chain Move** will be between the **ChainValutWallet** and another of **ChainAssetPlaces**. We wil *not* allow direct asset moves between **ChainAssetPlaces** that are not **ChainVaultWallet**.
- We need one special vault that oversees the cooperation between vaults, including itself, de-centrally.
  - **Master vault**: the special vault
  - **Master chain**: the chain that hosts the master vault
- The **Master vault** will be the hub for regular **Inter-Chain Moves**. This means an **Inter-Chain Move** will between the **Master vault** and another of local vaults. We wil *not* allow direct asset moves between **ChainAssetPlaces** of different chains.
- This algorithm will be executed off-chain to produce a transition plan, because
  - it will save huge gas fees that the algorithm would spend if it ran on-chain
  - the requirements don't require decentralization-level of asset move planning

Note: The off-chain execution of this algorithm raises the concerns of decentralization.

4.3.4. Algorithm input and output

- Input:
  - Target staking portfolio, given in the extended form of

$$VA_c^{t+} = (0D_c^{t+},\ 0W_c^{t+}, 0R_c^{t+}, optimal\ S^t)$$

  - Current staking portfolio, given in the extended form of

$$VA_c^t = (D_c^t,\ W_c^t, R_c^t, S_c^t)$$

- Output:
  - A transition plan generated
  - The transition plan executed

4.3.5. Process

- Find

  $$Asset\ Surplus_c = VA_c^t - VA_c^{t+}$$

  or, in other words,

  $$Asset\ Surplus_c\ =\ Current\ amounts\ -\ Target\ amounts$$

  for all chian c.

  - Asset surplus element is, therefore, defined for each of deposit, withdrawal, staking, and reward places on all chian.
  - The elements are called an **asset surplus** or simply a **surplus**.
  - An asset place is a giving place if it has a positive surplus, and a taking place if it has a negative surplus.
  - **The goal of transitioning is to take surplus amounts from giving places, and divide them to taking places.**
  - A deposit, as an asset place, has always a positive surplus and is a giving place, because its current amount is positive and the target amount is zero (i.e. we have to empty the place). The source will be the local vault when the request is pending processing.
  - A withdrawal, as an asset place, has always a negative surplus and is a taking place, because its current amount is negative (debt) (when the request is pending processing and the debt amount has been calculated) and the target amount is zero (i.e. we have to settle the debt). It has the destination of assets. The destination is be the "to" wallet of the request.
  - A reward, as an asset place, has always a positive surplus and is a giving place, because its current amount is positive and the target amount is zero (i.e. we have to empty the place). The source will be the local vault when the request has been collected.
  - A stake, as an asset place, has either a positive, zero, or negative surplus, because the target amount may be less, equal, or greater than the current amount. The source is the staking pool combined with the staking/un-staking method.
- Classify all asset places on al chains into giving and taking places, again.
  - If the asset surplus is significantly greater than zero, it is a **giving surplus** and it is the giving amount.
  - If the asset surplus is significantly less than zero, it is a **taking asset surplus** and its absolute value is the taking amount.
  - Else, it is a neutral asset place.
  - An asset place has has either
    - a positive **giving amount** and a zero taking amount,
    - a positive **taking amount** and a zero giving amount,
    - or, a zero giving amount and a zero taking amount.
- Classify chains into giving chains and taking chains

    - If the sum of giving amounts on a given chain is significantly greater than the sum of taking amount, it is a **giving chain**, and the absolute difference is called the **giving amount**, or surplus, of the giving chain.
    - If the sum of taking amounts on a given chain is significantly greater than the sum of giving amount, it is a **taking chain** and the absolute difference is called the **taking amount**, or deficit, of the taking chain.
    - Else, it is a neutral chian
    - A chain has either
        - a positive giving amount and a zero taking amount,
        - a positive taking amount and a zero giving amount,
        - or a zero giving amount and a zero taking amount.
- Generate a regular **intra-chain asset move plan** for giving chains
    - Collect the local swap prices and fees
    - Collect giving amounts of all giving asset places to the vault.
    - Swap, divide, and send the collected amount to fill the taking amounts of taking asset places. *Put priority on instances in W_c^t and make sure to fill in them.*
    - Regularize the plan. (See previous sections)
- Execute the regular intra-chain asset move plans for giving chains.
    - The giving, surplus amount of giving chains should remain in their vault.
- Generate a regular **inter-chain asset move plan** that divides giving amount of assets of giving chains to taking chains.
    - Collect giving amounts of giving chains from their vaults to the master vault.
    - Swap, divide, and send the collected amount to fill in the taking amounts of taking chains at their vaults.
    - Regularize the plan.
- Execute the regular inter-chain asset move plan.
    - There should not remain assets in the master vault, except dusts.
- Generate a regular intra-chain asset move plan for taking chains
    - Collect the local swap prices and fees.
    - Note: the vault has already got some assets that came from giving chains.
    - Collect giving amounts of all giving asset places to the vault.
    - Swap, divide, and send the collected amount to fill the taking amounts of taking asset places. *Put priority on instances in W_c^t and make sure to fill in them.*
    - Regularize the plan. (See previous sections)
- Execute regular intra-chain asset move plans for taking chains.
    - There should not remain assets in the their vaults, except dusts.

Note: This algorithm should be tweaked to cope with changing price slippages and fees, and numerical dusts, in implementation phases.

The algorithm is illustrated below:

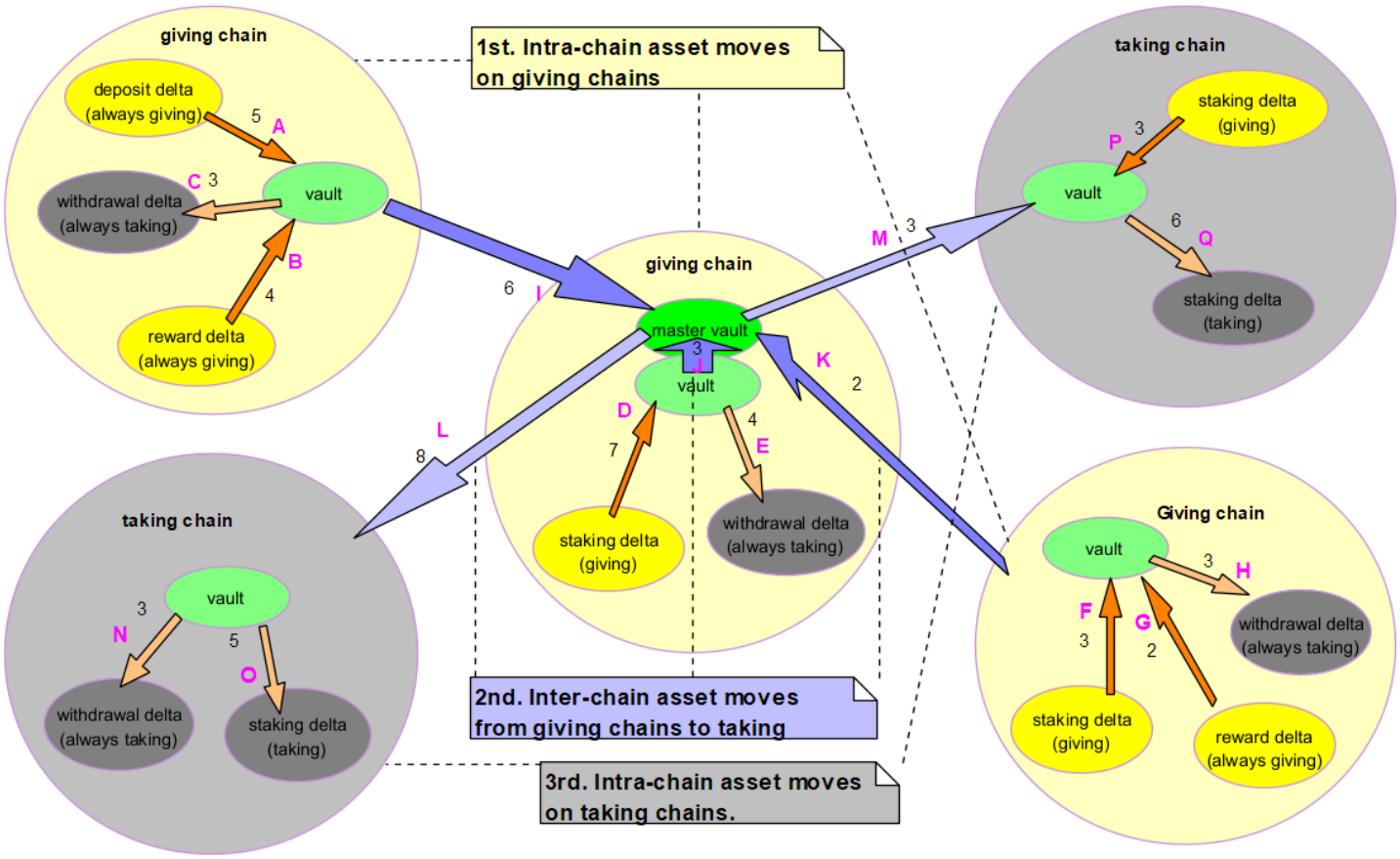

Figure. Transition planning algorithm. See the description of algorithm for details.
Core: Define [surplus = current amount - target amount] for each of deposit, withdrawal, staking, and reward places on all chain. A withdrawal is always a taking (negative) place, because its surplus is negative: target = 0 and current < 0 (debt). A reward is always a giving (positive) place, because its surplus is positive: target = 0 and current > 0. Goal of transition: Collect positive surpluses and divide them to negative surpluses. 1. Classify all surpluses on all chains into giving and taking places. 2. Classify classes into giving and taking chains. 3. (1rd) Move assets intra-chain to achieve the goal on giving chains. 4. (2nd) Move assets Inter-chain between vaults from giving chains to taking chains' vaults. 5. (3rd) Move assets intra-chain to achieve the goal on taking chains.

4.4. Optimization rounds

- On-chain: Take a snapshot of **system asset/request state**
- Off-chain: Calculate and send mLP tokens to users who requested deposit
- Off-chain: Calculate the amount of tokens to send to users who requested withdrawal
- Off-chain: Call Staking Planner to generate Optimal Staking Portfolio
- Off-chain: Call Transition Planner to generate Optimal Transition Plan
- On-chain: Execute the Optimal Transition Plan

4.5. Protocol drivers

4.5.1. Considerations

4.5.2. Design decisions

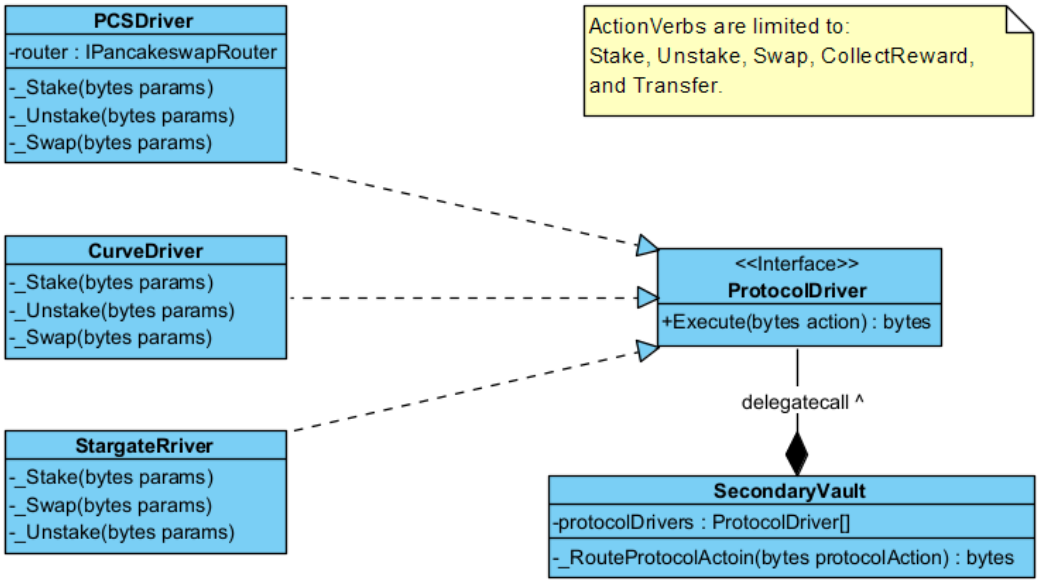

Figure. Driver scheme. The Execute function is "delegatecall"-ed.
protocolAction = abi.encode(protocol, action)
action = abi.encode(actionVerb, params)
params = abi.encode(decided by protocol and actionVerb)

4.5.3. Symmetry between on-chain and off-chain modules

TransitionPlanner will have the correspondent layer structure:

- Layer3: correspondent of vaults, that don't know about action type.
- Layer2: correspondent of drivers' Exexute(.) function, which doesn't know action parameter structure.
- Layer1: correspondent of drivers' internal action functions, like _Swap(.), which know action parameter structure.

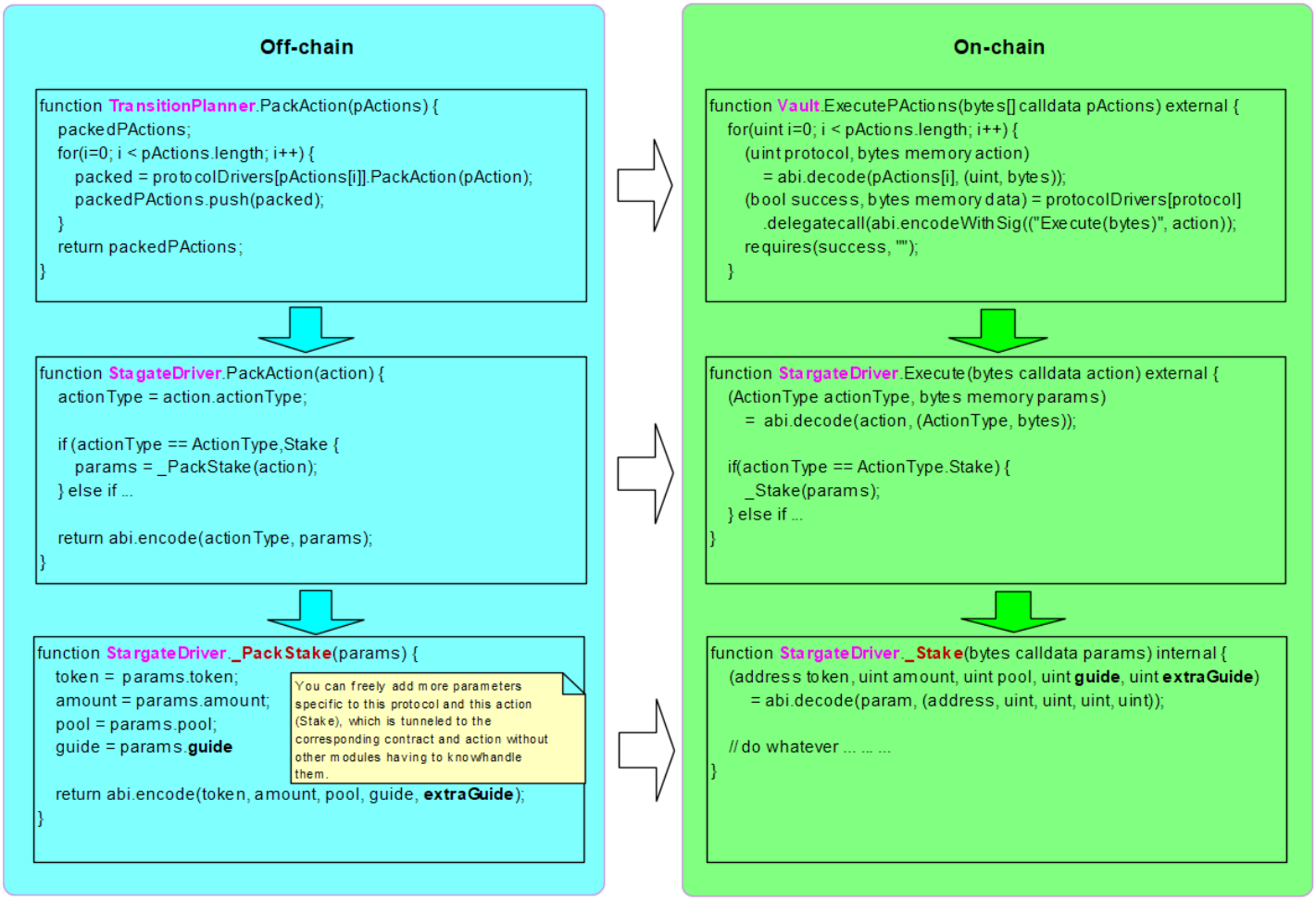

Figure. Protocol driver scheme illustrated. On-chain and off-chain protocol drivers are symmetric to each other.
Each layer has its own knowledge. Only the bottom layer handles specific actions and knows specific parameters of the particular protocol and action.

4.5.4. Reference source code

ProtocolDriver

```
abstract contract ProtocolDriver is Ownable {
    address public vault;

    function SetVault(address _vault) external virtual onlyOwner {
        require(vault != address(0), "Wrong vault address");
        vault = _vault;
    }

    modifier delegatedByVault() {
        require(address(this) == vault, "Wrong vault");  // this: Assuming delegatecall.
        _;
    }

    function Execute(bytes calldata action) external virtual delegatedByVault {
    }
}
```

Vaults

```
    ... ... ...
    mapping (uint => address) public protocolDrivers;

    function ChangeProtocolDriver(uint protocol, address driver) external onlyOwner {
        protocolDrivers[protocol] = driver;
    }

    function ExecuteActions(bytes[] calldata protocolActions) external onlyOwner {
        for (uint i = 0; i < protocolActions.length ; i++) {
            (uint protocol, bytes memory action) = abi.decode(protocolActions[i], (uint, bytes));
            (bool success, bytes memory data) = protocolDrivers[protocol]
            .delegatecall(abi.encodeWithSignature(("Execute(bytes)"), action));
            require(success, "");
        }
    }
    ... ... ...
```

Driver example

```
contract StargateDriver is ProtocolDriver {
    ... ... ...

    function Execute(bytes calldata action) external virtual override delegatedByVault {
        (ActionType actionType, bytes memory params) = abi.decode(action, (ActionType, bytes));

        if(actionType == ActionType.Stake) {
            _Stake(params);
        } else if(actionType == ActionType.Unstake) {
            _Unstake(params);
        }
    }

    function _Stake(bytes calldata params) internal virtual {
        (address token, uint pool, uint special_for_stagate_staking) = abi.decode(params, (address, uint, uint));

        // do whatever ...
        // You are free to introduce any (protocol x action)-specific param, like special_for_stagate_staking
        // because you have StargateDriver correspondent on the off-chain side (TransitionPlanner)
    }

    function _Unstake(bytes calldata params) internal virtual {
    }

    ... ... ...
}
```

5. Cross-chain transportation

5.1. Considerations

- Decentralized inter-chain transportation is required for omnichain-ness

- Decentralized inter-chain transportation may lead to bad User Experience, for its inherent long asynchronous operation
- As such, we need to reduce the use of inter-chain transportation as possible
- Layer Zero is the de facto industry standard of decentralized inter-chain transportation service
- There are total 6 cross-chain calls between the master vault and a (local) vault when the system carries out a round of optimization. (See below diagrams.)
    - If the system is deployed on 10 chains and optimizes 24 times a day, we will have **1,440 cross-chain calls a day**.
    - The 6 cascaded cross-chain calls may pose significant risks to integrity/consistency and User Experience, like runtime responsiveness and coding/maintenance complexity.
- If we compromise on the integrity/consistency of optimization (not on asset moves), by adopting off-chain version of executing transition plans and thus exposing the system to rarely feasible hacking/attacks, then cross-chain calls will be cut down 50%, in return. (See below diagrams.)
- **Not all inter-chain transportation need to be decentralized** (explained below)

5.2. Decentralized operations required

Some vault operations should be decentralized to meet the requirements. Tracking of asset/mLP amount should be executed decentrally, without intermediate off-chain agent or relayer, and with transparent loggs

- Collecting pending rewards from all staking pools to vaults
- Withdrawing from and staking to staking pools to/from vault
- Query for local amounts of assset and mLP token
- Cooperation between vaults to calculate and exchange the information of asset/mLP amounts and indexes derived therefrom
- Sending assets from users' wallets to vaults, on users' deposit requests
- Calculating mLP amount to send to users, in return for their deposited assets
- Sending mLP tokens from vaults to users' wallets, on users' deposit requests
- Sending mLP from users' wallets to vaults, on users' withdrawal requests
- Calculating asset amount to send to users, in return for their returned mLP tokens
- Sending assets from vaults to users' wallets, on users' withdrawal requests

5.3. Employ the LayerZero service

Transparent cross-chain transportation is the fundamental basis of omnichain operations. We choose LayerZero service for our cross-chain transportation.

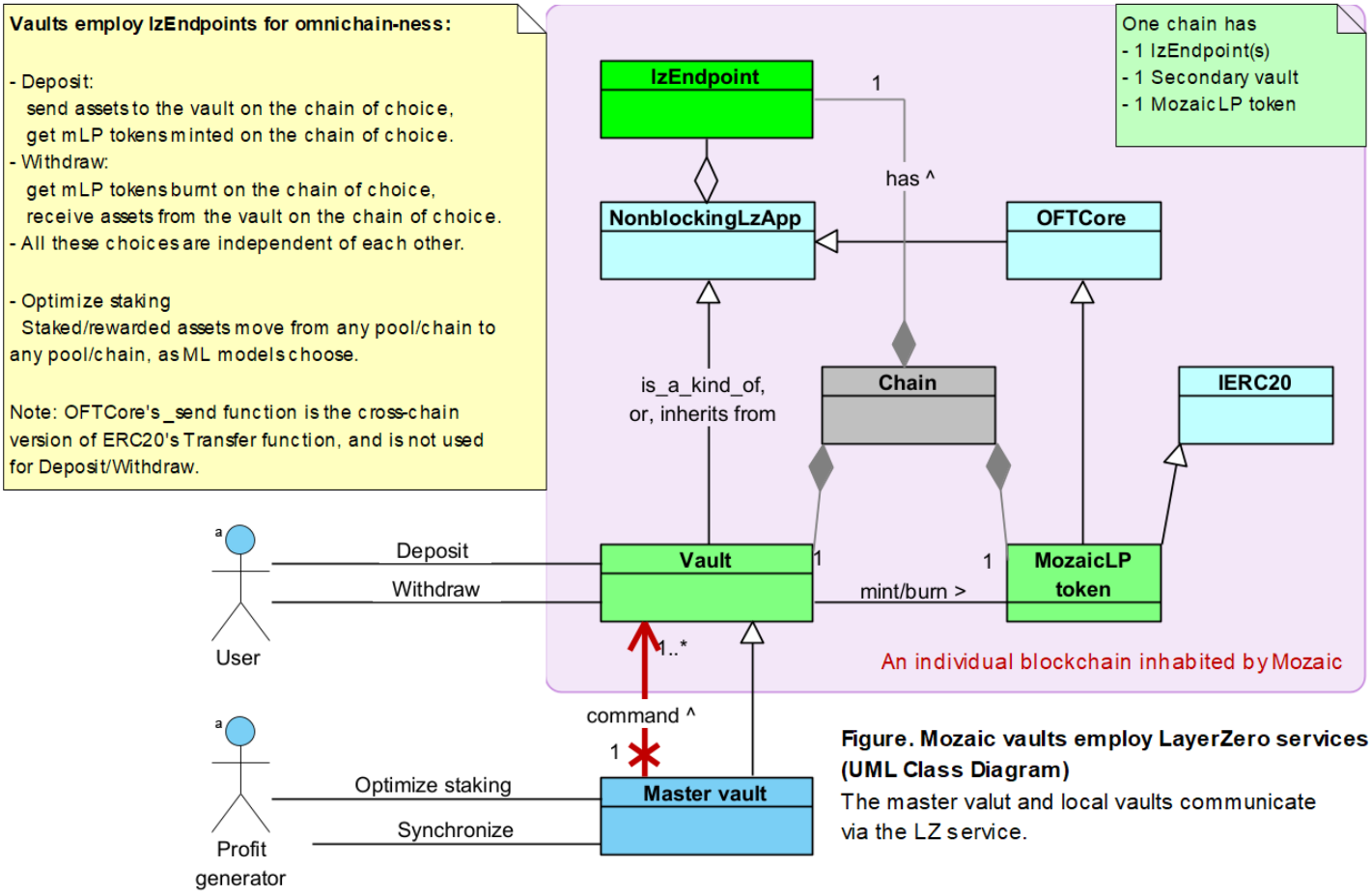

Figure. Mozaic vaults employ LayerZero services (UML Class Diagram)
The master valut and local vaults communicate via the LZ service.

5.4. Operations exemptible of decentralization

Vaults cooperation for staking optimization **does not have to be decentralized**, in the meaning that the optimization doesn't have to provide ideal maximum profit nor have to be successful

- Collecting pools information from chains to off-chain modules, could be done by off-chain modules
- Sending asset move plana to chains, could take detour via Mozaic off-chain modules with Admin wallets, at the risk of
  - the plan could be tempered by (inauditable/unautidited) Mozaic modules or hackers. (But the plan itself is calculated by off-chain modules.)
  - the plan may even fail to be conveyed. (But this type of off-chain failture can also happen when we don't employ off-line detours.)
- Relaying requests between local vaults, during transitioning to a new staking, could take detour via Mozaic off-chain modules with Admin wallets, with the same risks as above

5.5. Design recommendations

- We will choose *decentralized* inter-chain transportation between off-chain modules and vault contracts when finding new optimal staking portfolio and transitioning to the new staking
- Inter-chain messages, once identified as required, will carry as much information as possible.
- Read the section "Reference source code" for more.

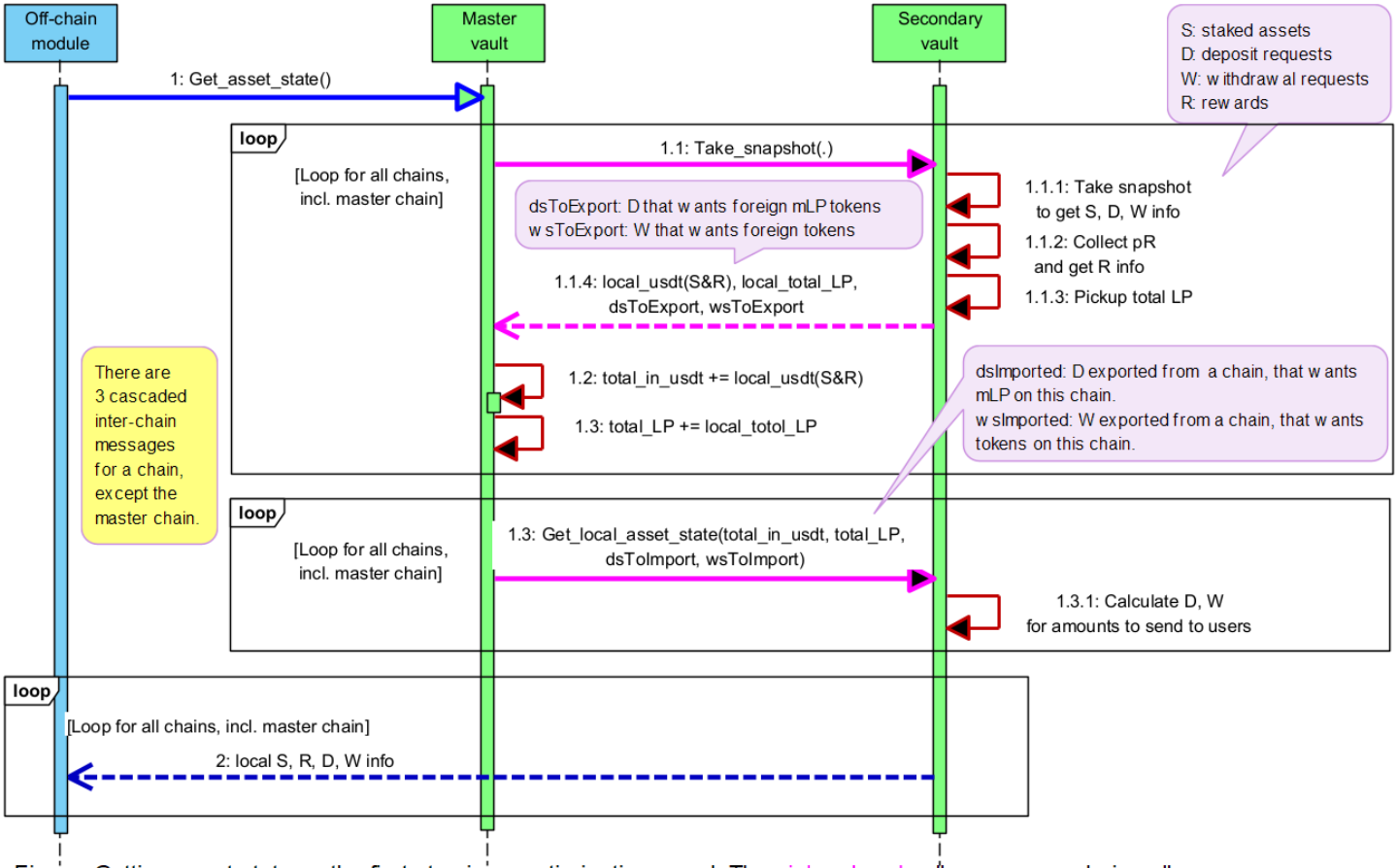

Figure. Getting asset state as the first step in an optimization round. The pink-colored calls are cross-chain calls. The information of S, R, D, and W is created without off-chain involvement. (A contract can call an off-chain module by emitting an event that the module is awaiting.)

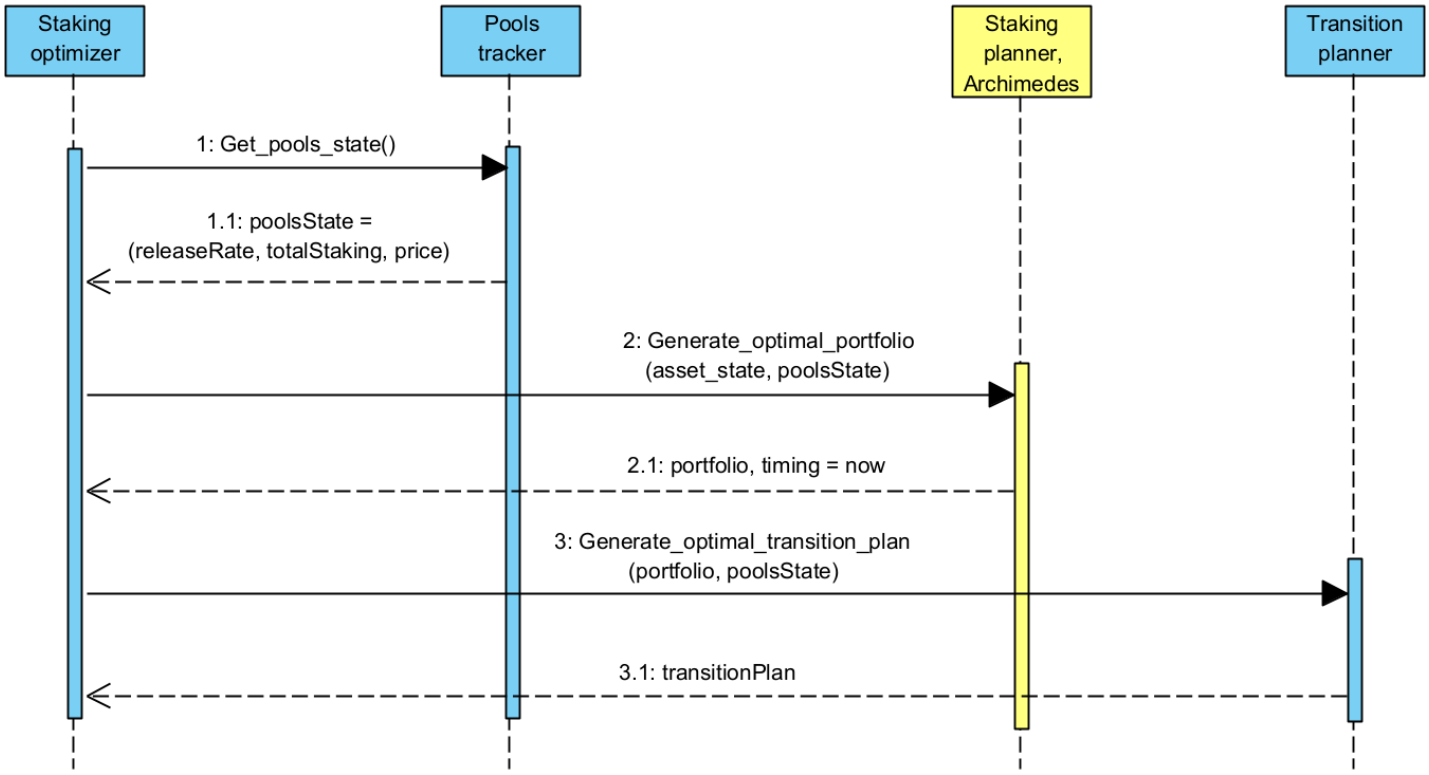

Figure. Generating optimal transition plan, as the second and middle step in an optimization round.
All modules run off-chain. Pools tracker constantly monitors staking pools. Staking planner, the heart of

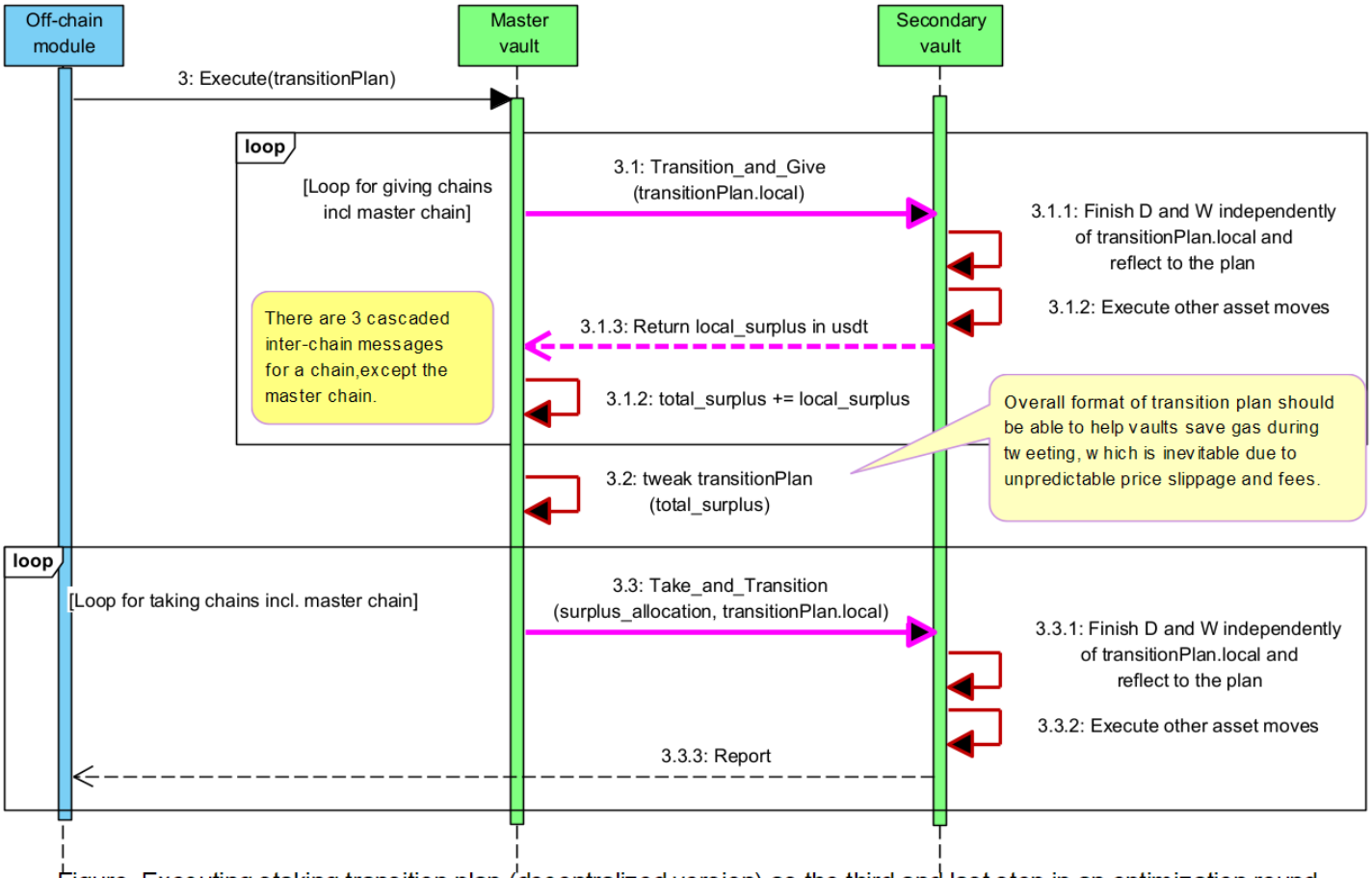

Figure. Executing staking transition plan (decentralized version) as the third and last step in an optimization round. Pink-colored calls are cross-chain calls.

5.6. An off-chain detour for inter-chain transportation

- An off-chain module monitors event logs of a smart contract for a target event happening
- Once detected, the event will be consumed by off-chain modules to produce response
- The produces response will be sent to a proper smart contract

6. Overall state transition

6.1. Design decisions

We choose the **Toggle-Between-Optimize-and_Stay** model for the overall system behavior.

- The system will not always be transitioning, but staying most of the time accepting deposit/withdrawal requests from users.
- If the system **accept**s a deposit request, it
  - collects the asset the user wants to deposit, on the asset's chain,

    - tells the user to wait until the next optimization round, when the system will send some mLP tokens to the user in return for the asset,
    - and book the request with the system for later processing.
- If the system **accept**s a withdrawal request, it
    - collects the mLP tokens the user wants to return, on the mLP's chain,
    - tells the user to wait until the next optimization round, when the system will send some assets of the requested token type in return for the mLP tokens,
    - and book the request with the system for later processing.
- The system will **optimize system asset/request state**, or **transition to new staking** at regular or irregular intervals. The frequency of optimization rounds will be optimized, as frequent moves of asset may incur more costs while infrequent optimization rounds will hinder from quick maneuver of staking.
- When an optimization round is requested, the system leaves the **Staying** state and enters the **Optimizing** state.
- When entering the **Optimizing** state, or an **Optimization round**, the system takes a **system asset/request snapshot**. Then the system transforms/changes the state to an optimal **system asset state** for more rewards, while continuing to **accept** deposit/withdrawal requests, which will be handled at the next round of optimization.
- When an optimization round is finished, the system leaves the **Optimizing** state and enters the **Staying** state.
- In the **Staying** state, the system does nothing but continues to **accept** deposit/withdrawal requests, which will be handled at the next round of optimization.

6.2. Visual description

The **Toggle-Between-Optimize-and_Stay** model of behavior can be expressed in a UML State Machine diagram shown below:

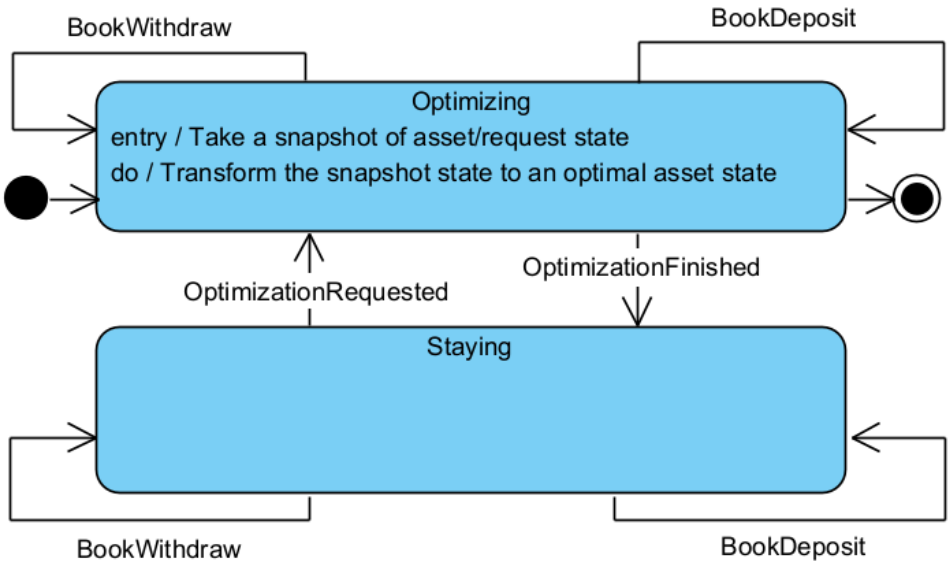

Figure. High-level state machine of Mozaic system

6.3. Reference source code

```
abstract contract SecondaryVault is NonblockingLzApp {
    ... ... ...

    struct Deposit {
        address user;
        address token;
        uint    amount;
        uint    amountLP;   // undefined initially
    }

    struct DepositImported {
        address user;
        uint    usdEq;
        uint    amountLP;   // undefined initially
    }

    struct DepositToExport {
        address user;
        uint    usdEq;
        uint    chainId;
    }

    struct Withdrawal {
        address user;
        address token;
        uint    amount;    // undefined initially
        uint    amountLP;
    }

    struct WithdrawalImported {
        address user;
        address token;
        uint    amount;    // undefined initially
        uint    amountLP;
    }
```

```
    struct WithdrawalToExport {
        address user;
        address token;
        uint    amountLP;
        uint    chainId;
    }

    struct Workspace {
        Deposit[] ds;
        DepositToExport[] dsToExport;
        DepositImported[] dsImported;
        Withdrawal[] ws;
        WithdrawalToExport[] wsToExport;
        WithdrawalImported[] wsImported;
    }

    Workspace private pending;
    Workspace private staged;

    uint public thisChain;
    address public mLP;

    function _safeTransferFrom(
        address _token,
        address _from,
        address _to,
        uint256 _value
    ) private {
        // bytes4(keccak256(bytes('transferFrom(address,address,uint256)')));
        (bool success, bytes memory data) = _token.call(abi.encodeWithSelector(0x23b872dd, _from, _to, _value));
        require(success && (data.length == 0 || abi.decode(data, (bool))), "transfer failed");
    }

    function _clean(Workspace storage ws) internal {

    }

    // This call begins transitioning.
    function takeSnapshot() external {
        Workspace storage temp = staged;
        staged = pending;
        pending = temp;
        _clean(pending);

        // do whatever with staged

    }
}
```

7. Miscellaneous

7.1. Compounding

7.1.1. Considerations

- Rewards should be compounded as frequently as possible, unless the gas fees grows larger than rewards

- Compounding can be executed either:
  - at stocking transition rounds
  - or at its own intervals

7.2. Design recommendations

- Leave it open

7.3. Gas supply

7.3.1. Considerations

- Optimization will spend significant amount of gas, although we minimize vaults
- Gas is spent chain-wise, although the profit generation is not necessarily chain-wise

7.3.2. Design recommendations

- The initial version will be sourcing local gas fees from the local Staking Stock. **If the local Staking Stock is not sufficient for gas fees, the chain will be set inactive**.
- Future versions will maintain a distributed treasure manager to provide local gas spending.

7.4. Auxiliary descriptions of the architecture

7.4.1. Deposit / Withdraw - deposits and rewards mixed 1:1

- We maintain a variable $deposits$ per user.
- Alice's $deposits$ is now 170 stable coins.
- Alice deposits 30 stable coins,
  - Her $deposits$ increases by 30 to become 200.
  - Her total mLP token increases by some amount of mLP token that is calculated to be the share of 30 in the system's resulting renewed Staking Stock.
- When she wants to withdraw with 20 mLP tokens
  - We first find she has a total 100 mLP tokens
  - Decrease her $deposits$ by 200 * 20 / 100 = 40 stable coins
  - Return the withdrawal assets to her, say 120 stable coins, which is a mix of deposits and rewards
  - We know she withdraws 40 principal deposit, together with 120 - 40 = 80 rewards
- **Now, the performance fees = 80 * 10% = 8 stable coins.**
- This means the withdrawal amount is forced to be a 1:1, which is not numerical but proportional, mix of original/principal deposits and rewards generated by staking/compounding them.
- The system will not serve other mix ratios but 1:1, because a ratio is meaning less as $deposits$ and rewards are all mixed and work together with constant compounding.

8. Reference source code

Below comes reference code that sketches and/or decides the code architecture.

```
pragma solidity ^0.8.0;

// imports
import "../libraries/lzApp/NonblockingLzApp.sol";
import "../libraries/stargate/Router.sol";
import "../libraries/stargate/Pool.sol";
import "./MozaicLP.sol";
import "./ProtocolDriver.sol";

// libraries
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";

abstract contract ProtocolDriver is Ownable {
    address public vault;

    function SetVault(address _vault) external virtual onlyOwner {
        require(vault != address(0), "Wrong vault address");
        vault = _vault;
    }

    modifier delegatedByVault() {
        require(this == vault, "Wrong vault");  // this: Assuming delegatecall.
        _;
    }

    function Execute(bytes calldata action) external virtual delegatedByVault {
    }
}

abstract contract SecondaryVault is NonblockingLzApp {
    function _usdt(address _token, uint _amount) internal returns (uint usdEq) {
        // use whatever source of price to get usdt-equivalent of
        // the _amount amount of _token token.
    }

    mapping(address => uint) public userTokens;

    function addUserToken(address _token) external {
        require( _token != address(0), "");
        userTokens[_token] = 1;
    }
    function removeUserToken(address _token) external {
        require( _token != address(0), "");
        userTokens[_token] = 0;
    }

    struct Deposit {
        address user;
        address token;
        uint    amount;
        uint    amountLP;   // undefined initially
    }

    struct DepositImported {
        address user;
        uint    usdEq;
        uint    amountLP;   // undefined initially
    }

    struct DepositToExport {
```

```
    address user;
    uint    usdEq;
    uint    chainId;
}

struct Withdrawal {
    address user;
    address token;
    uint    amount;    // undefined initially
    uint    amountLP;
}

struct WithdrawalImported {
    address user;
    address token;
    uint    amount;    // undefined initially
    uint    amountLP;
}

struct WithdrawalToExport {
    address user;
    address token;
    uint    amountLP;
    uint    chainId;
}

struct Workspace {
    Deposit[] ds;
    DepositToExport[] dsToExport;
    DepositImported[] dsImported;
    Withdrawal[] ws;
    WithdrawalToExport[] wsToExport;
    WithdrawalImported[] wsImported;
}

Workspace private pending;
Workspace private staged;

uint public thisChain;
address public mLP;

function _safeTransferFrom(
    address _token,
    address _from,
    address _to,
    uint256 _value
) private {
    // bytes4(keccak256(bytes('transferFrom(address,address,uint256)')));
    (bool success, bytes memory data) = _token.call(abi.encodeWithSelector(0x23b872dd, _from, _to, _value));
    require(success && (data.length == 0 || abi.decode(data, (bool))), "Stargate: TRANSFER_FROM_FAILED");
}

function _clean(Workspace storage ws) internal {

}

// This call begins transitioning.
function takeSnapshot() external {
    Workspace storage temp = staged;
    staged = pending;
    pending = temp;
    _clean(pending);
```

```
        // do whatever with staged

    }

    // The caller submits _amount of _token, and wants MLP tokens on _chain.
    function addDepositRequest(address _token, uint _amount, uint _chain) external  {
        require(userTokens[_token] != 0 && _amount > 0, "Wrong token/amount");
        _safeTransferFrom(_token, msg.sender, address(this), _amount);

        if (_chain == thisChain) { // The request is local-token for local-mLP
            pending.ds.push( Deposit(msg.sender, _token, _amount, 0) );
            // 0 for mLP amount to send to the user.
        } else { // The request is local-token for away-mLP
            pending.dsToExport.push(DepositToExport(msg.sender, _usdt(_token, _amount), _chain));
            // Foreign chain _chain will store this like:
            // pending.dsImported.push(DepositImported(msg.sender, usdEq, 0));
            // 0 for the undefined mLP amount to send to the user
        }
    }

    // The caller submits _amount of mLP, and wants _token tokens on _chain chain.
    function addWithdrawalRequest(uint _amountLP, address _token, uint _chain) external  {
        require(userTokens[_token] != 0 && _amountLP > 0, "Wrong token/amount");
        _safeTransferFrom(mLP, msg.sender, address(this), _amountLP);

        if (_chain == thisChain) { // The request is local-LP for local-token
            pending.ws.push( Withdrawal(msg.sender, _token, 0, _amountLP) );
            // 0 for the undefined amount of token to send to the user.
        } else { // The request is local-LP for away-token
            pending.wsToExport.push(WithdrawalToExport(msg.sender, _token, _amountLP, _chain));
            // Foreign chain _chain will store this like:
            // pending.wsImported.push(WithdrawalImported(msg.sender, _token, 0, _amountLP));
            // 0 for the undefined amount of token to send to the user
        }
    }



    mapping (uint => address) public protocolDrivers;

    function ChangeProtocolDriver(uint protocol, address driver) external onlyOwner {
        protocolDrivers[protocol] = driver;
    }

    function ExecuteActions(bytes[] calldata protocolActions) external onlyOwner {
        for (uint i = 0; i < protocolActions.length ; i++) {
            (uint protocol, bytes memory action) = abi.decode(protocolActions[i], (uint, bytes));
            (bool success, bytes memory data) = protocolDrivers[protocol]
            .delegatecall(abi.encodeWithSignature(("Execute(bytes)"), action));
            require(success, "");
        }
    }
}


contract StargateDriver is ProtocolDriver {
    function Execute(bytes calldata action) external virtual override delegatedByVault {
        (ActionType actionType, bytes memory params) = abi.decode(action, (ActionType, bytes));

        if(actionType == ActionType.Stake) {
            _Stake(params);
```

```
        } else if(actionType == ActionType.Unstake) {
            _Unstake(params);
        }
    }

    function _Stake(bytes calldata params) internal virtual {
        (address token, uint pool, uint special_for_stagate_staking) = abi.decode(params, (address, uint, uint));

        // do whatever ...
        // You are free to introduce any (protocol x action)-specific param, like special_for_stagate_staking
        // because you have StargateDriver correspondent on the off-chain side (TransitionPlanner)
    }

    function _Unstake(bytes calldata params) internal virtual {
    }
}
```