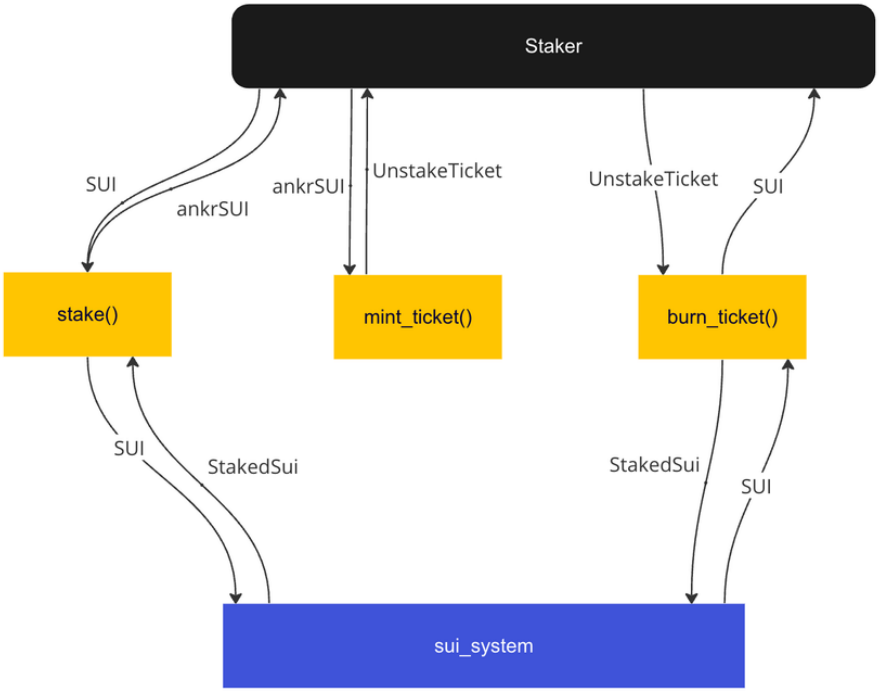
SUI native pool



Decentralization (SIP6)

The new **Native Pool** module for SUI Liquid Staking aims to increase the decentralization of staking and replace intermediary approaches. It uses `request_add_stake_non_entry` and `request_withdraw_stake_non_entry` methods of `sui_system` to stake the user's SUI tokens and receive StakedSui that we store inside the contract.

We diversify StakedSui using different validators for staking and unstaking. Our operator assigns different priority levels to the validators in the set, and the contract stakes only with the validator that holds the highest priority and unstakes starting with the validator that holds the lowest priority.

Problem

The SUI Staking mechanics doesn't allow to return the user their funds if they requested an unstake the same epoch they staked while the funds are still in the pending state.

Our goal is to fulfill unstake requests as soon as possible while avoiding big plunges in the APR that happen because such requests can only be served by stakes (StakedSUI) actively accruing rewards.

Solution

To solve this problem, we have implemented the following mechanics:

- Delayed unstake managed via *UnstakeTicket* for requests that take up the whole active SUI balance.
- Fee that lowers the financial appeal of unstake requests that are fulfilled within the current epoch.

Technical Details

Smart Contracts and Objects

- Liquid Staking (PACKAGE) — entry point for interaction with Liquid Staking modules.
- CERT_METADATA — object that stores the total supply of ankrSUI.
- NATIVE_POOL — main pool object that stores *ValidatorSet* and main data for the pool operations (balance, stakes, etc).
- COIN_METADATA — object that stores `Metadata(<CERT>)` (name, ticker, icon) of ankrSUI.
- OwnerCap — capability to call functions with the owner role access level.
- OperatorCap — capability to call functions with the operator role acces level.
- Vault — object that stores the staking queue for a specific validator; the queue is freed in the FIFO order in the unstake process.
- ValidatorSet — object that stores validators sorted by their priority level and their *Vaults*.

SUI Testnet

- PACKAGE (0x9ada5e2775ece271d80c8731c3f388a5228d3d2c8bc312fbad9ab8b6d4c852fb)
- CERT_METADATA (0x82b87af16cfad07fa28a244158bf1a18c78db2f3f7181f0f545ce2a19199005d)
- NATIVE_POOL (0xe36b9d308fb64623a8a31e482c81d46dda506a597cda6f9c55fc5c79bd051b6b)
- COIN_METADATA (0xd9775f8ab2ccadb1d0695a4b75cbfc94fd49454a7b4af0b803e167ba9e0b6207)

Dependencies

Some methods depend on Sui system objects.

- `SuiSystemState` — 0x5 (addresses)
- `Clock` — 0x6 (addresses)

Staking process

`stake(NativePool, Metadata<CERT>, SuiSystemState, Coin<SUI>)`

The main entry point for the user. Exchanges SUI to CERT (ankrSUI), adds SUI to the pool in the pending state and if pending balance is greater than 1SUI tries to stake all of it to a validator that holds the highest priority, otherwise the tokens remain in the pool. The stake amount must be greater or equal to *native_pool::min_stake*. The user begins receiving their rewards instantly, however the stake itself will start earning since next epoch after it has been staked.

Workflow

1. The user calls `stake(NativePool, Metadata<CERT>, SuiSystemState, Coin<SUI>)`. Inside, the following happens:
   a. `NATIVE_POOL::` checks that the amount of SUI in the transaction is > than the min limit.
   b. `NATIVE_POOL::` calculates the amount the user will get: *ankrSUI = stake amount * ratio*.
   c. `NATIVE_POOL::` adds the SUI to the pool pending balance.
   d. If the pending balance >= 1 SUI:
      i. `NATIVE_POOL::` finds a validator with the highest priority from the list committed by our backend.
      ii. `NATIVE_POOL::` calls the SUI system contract `sui_system::request_add_stake_non_entry()` which returns a StakedSUI object (stake).
      iii. `NATIVE_POOL::` puts the StakedSUI object in an array associated with the validator from Step i.
   e. `NATIVE_POOL::` mints and transfers to the user ankrSUI in the amount calculated at Step b.
2. A StakedEvent event is emitted.

Example

SuiScan - Sui Explorer

Rewards

After an epoch change, our backend queries for the total amount of accrued rewards for all StakedSUI, and stores the resulting sum to the contract.

As a result of this action, ratio gets updated by the formula: *shares_supply / (total_staked + (total_rewards - collected_rewards) - tickets_supply)* where *shares_supply* is `CERT_METADATA::total_supply`, *total_staked* is the value from the staking event and *total_staked = total_rewards - collected_rewards*, *total_rewards* is `NATIVE_POOL:total_rewards`, *collected_rewards* is `NATIVE_POOL::collected_rewards`, *tickes_supply* is the number of all unburned UnstakeTickets.

Unstaking process

Unstaking process involves 2 stages, that can be performed individually or in a single function call.

First stage

`mint_ticket(NativePool, Metadata<CERT>, Coin<CERT>)`

Burns ankrSUI tokens (which stops rewards generation for the staker) and mints UnstakeTicket object, that allows it's owner to receive SUI initial funds and rewards.

In most cases user can unstake in a single function call `unstake(NativePool, Metadata<CERT>, SuiSystemState, Coin<CERT>)` that includes both ticket minting and burning, if total value of all unstake tickets > total_active_stake.

UnstakeTicket

An UnstakeTicket is an non-transferrable object that gives the right to it's owner to unstake specified amount of SUI since a certain epoch.

An UnstakeTicket contains the following fields:

- value (uint64) — the amount of SUI to unstake. When burning an UnstakeTicket, the user receives `value - unstake_fee` where `value` is calculated by the ankrSUI-to-SUI exchange rate from the day the UnstakeTicket was created.
- unstake_fee (uint64) — a fee the protocol deducts during the second unstake stage if the total value of all minted UnstakeTickets for the last 2 SUI Chain epochs is higher than `unstake_fee_threshold` % from the total_staked (sum of all user stakes). unstake_fee is a fixed percentage N% (0.05% by default) and is calculated by the following formula: `unstake_fee = value * base_unstake_fee`.
- epoch (uint64) — number of SUI Chain epoch since a ticket can be burned (User is free to burn their ticket any epoch later). It is the current epoch by default, however, if *total value of all unstake tickets > total_active_stake*, then `epoch` = current_epoch + 1, where *total_active_stake = sum of all stakes up to the current epoch - sum of all burned unstake tickets, incl. current epoch*. In other words if all StakedSUI on validators is in the "pending" state, then user has to wait until the next epoch when it becomes "active" and available for unstake.

Example

…….

Second stage

The second stage is initiated by calling the `burn_ticket(NativePool, SuiSystemState, UnstakeTicket)` method, which sends the SUI `value` defined in the UnstakeTicket to the user.

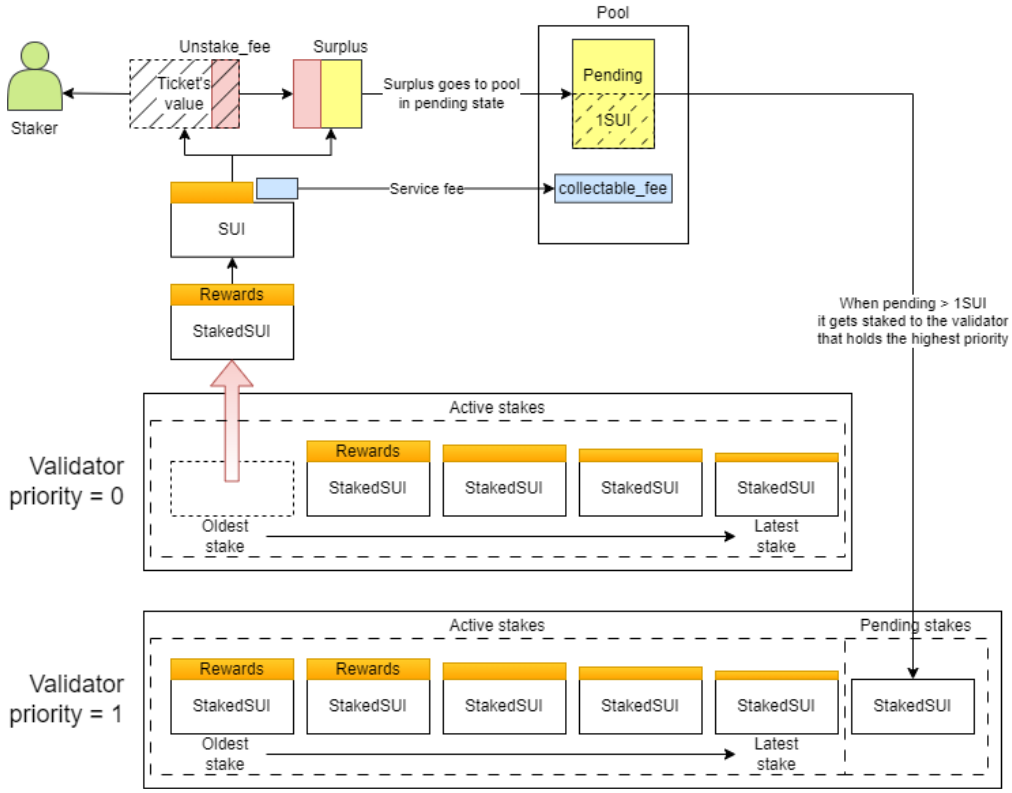
To obtain the `value`, the protocol unstakes funds the following way:

1. Starting from the validator that holds the lowest priority.
2. Within the validator, in a queue starting from the oldest stake (StakedSUI) through the newest.
3. For each StakedSUI that accrued rewards, the contract deducts a `base_reward_fee` % from the rewards and adds it to the `NativePool::collectable_fee` variable for the owner to collect later.

When the `value` of the UnstakeTicket is reached, the contract stops unstaking funds and sends the user their SUI (`value - unstake_fee`).

A surplus can remain — the contract can unstake an amount of SUI greater than the `value`. It can happen, as StakedSUI is an object containing >= 1 SUI, which is not only the body of the stake, but also the accrued staking rewards, and the final StakedSUI in the queue can have more funds that needed to cover the `value`.

The surplus is added to the pending contract balance, waiting to be staked when >=1 SUI.



burn_ticket() flow

Example

…

Workflow

1. The user calls `mint_ticket(NativePool, Metadata<CERT>, Coin<CERT>)`. Inside, the following happens:
   a. `NATIVE_POOL::` calculates the amount of SUI to return to the user: *SUI = unstake amount / ratio.*
   b. `NATIVE_POOL::` checks the amount from Step a is > 1 SUI.
   c. `NATIVE_POOL::` burns the ankrSUI the user sent at Step 1 (`Coin<CERT>`).
   d. `NATIVE_POOL::` calculates the `unstake_fee`:
      i. `NATIVE_POOL::` checks how much has been unstaked for the last 2 epochs and adds to it the amount the user unstakes.
      ii. If the sum from Step i > n% of total stake of the pool, `NATIVE_POOL::` calculates the `unstake_fee = unstake_amount * n%`.
   e. `NATIVE_POOL::` checks if the unstake request can be fulfilled in the current epoch: *sum of all unfulfilled unstake request + unstake amount from the current request.*
      i. If the result is > total active stake (all stakes before the current epoch), the request cannot be fulfilled this epoch and will be fulfilled the next epoch.
   f. `NATIVE_POOL::` creates and transfers the user an UnstakeTicket with the `unstake_amount` from Step a, `unstake_fee` from Step d, `epoch` from Step e.i.
   g. An UnstakedEvent event is emitted.
2. The user calls `burn_ticket(NativePool, SuiSystemState, UnstakeTicket)`.

a. `NATIVE_POOL::` checks that the `epoch` in the ticket is >= current epoch. If no, the transaction is reverted.

b. If yes, `NATIVE_POOL::` gets the list with validators ranged by their priority committed by our backend and unstakes starting from the validator with the lowest priority:

   i. `NATIVE_POOL::` gets the validator's array of StakedSUI objects and calls `sui_system::request_withdraw_stake_non_entry()` for each object until the `unstake_amount` in the ticket is reached.

   ii. `NATIVE_POOL::` subtracts `base_reward_fee` from the rewards each StakedSUI object accrued, and adds it to to the Ankr treasury.

c. `NATIVE_POOL::` subtracts `unstake_fee` from the `unstake_amount`, and adds it to Ankr treasury.

d. If there is any surplus, `NATIVE_POOL::` adds it to the pending balance.

   i. If the pending balance > 1 SUI, `NATIVE_POOL::` restakes it.

e. `NATIVE_POOL::` transfers the user the `unstake_amount`.

f. A TicketBurnedEvent event is emitted.

Unstaking corner cases

1. If the user tries to unstake in the current epoch an amount bigger than the one that was staked before the beginning of it, such unstake is impossible to pay out instantly. The user will have to wait till the end of the current epoch to get their tokens. A telling example can be: previous epoch stakes are 100K (TVL 100K), current epoch stakesare 200K, the user unstakes 150K in the current epoch (150% of the previous epoch TVL). From our practice, however, such cases arise during QA, and the users must never see them in the production environment.
2. If the user makes multiple stake/unstake transactions, it can significantly decrease the APR, as each transaction decreases the active TVL that gets rewards and increases the passive TVL that will get rewards only in 2 epochs. Although this malicious user will not gain any profit and, in fact, will pay a fee for each transaction, this behavior still affects all users of the protocol. To discourage attacks of this kind, we take a small fee that is nearly equal to the APR decrease if unstakes in the current epoch have used more than 10% of the TVL in the previous epoch.

To sum up:

1. Unstakes lower than the `unstake_fee_threshold` are instant and bear no fee (see UnstakeTicket above).
2. Unstakes higher than the `unstake_fee_threshold` are instant and bear the `unstake_fee`, which is included in the body of unstaked amount, i.e., the user gets `value - unstake_fee`.
3. Unstakes as big as or higher than the active part of the pool TVL make the `epoch` value in the UnstakeTicket equal to `current_epoch + 1`.

Updating validators

`update_validators(NativePool, vector<address>, vector<u64>, OperatorCap)`

Our backend observes the network state and prioritizes validators based on their APY, reputation and already staked amount; the number of validators may vary starting from minimum 5 validators, each validator receiving stakes until it reaches 20% TVL, then its priority is lowered not to receive any more stakes. The contract makes decision based on the priority: stake with the validator that holds the highest priority, unstake starting with the validator that holds the lowest priority.

IMPORTANT: to apply priority, see **Sorting validators**.

Example

If we want to send all new stakes to the validator `0xba4d20899c7fd438d50b2de2486d08e03f34beb78a679142629a6baacb88b013`

and unstake from the validator `0x3d618b03660f4e8b4ec99c52af08a814f5248154937782d22b5a8f2e44ba15fc` and `0x9c4155f9e901324198fc9c737e15e6b14da5b9d2f38243213f115a7d45f3d048`,

then the address who owns the Operator Cap should make a call with this 3 validators addresses and prioritize them in this way: [3,1,2]. As a result, `0xba4d` will take the first place, and contract will stake with it. The last one at the bottom of queue, `0x3d61`, will be the first candidate for unstakes.

SuiScan - Sui Explorer

Zero priority

If our backend sets a validator 0 priority, it means we want to prune its state from the module. Upon the last user unstake form this validator, we remove it from the *validator_set* module.

Sorting validators

`sort_validators(NativePool)`

Sorts the validators in the list by priority. For priority details, see the Update Validators method description above.

Example

SuiScan - Sui Explorer

ANKR's impact

Operator