Modular Version Staking v0.2 Technical Design

Table of Contents - Modular Version Staking v0.2 Technical Design

Motivation

This document outlines the technical implementation of Staking v0.2.

Staking v0.1

Staking v0.1 was released in December 2022 as a public beta. v0.1 aimed to introduce additional cryptoeconomic security to the Chainlink Network focused on helping secure the ETH/USD Data Feed on Ethereum. In the event that the node operators responsible for running the ETH/USD DON failed to maintain certain performance requirements (uptime), then those node operators would see a portion of their rewards slashed. For increasing the security of the network, stakers earn rewards in the form of the LINK token.

Staking v0.2

The primary goal of Staking v0.2 is to build upon learnings taken from the v0.1 release, introduce greater flexibility for stakers, improve the security guarantees for supported oracle services, and redesign the rewards mechanism. Furthermore, v0.2 was built to use a modular set of smart contracts, which allows for future improvements and features to be added in an iterative manner. Further context on the high-level overview of v0.2 can be found in this blog.

NOTE: This documentation and code contains parameter config values. These config values are for illustration purposes only in order to explain how the code executes through examples. Such configs will be set at different values upon launch.

For Auditors

System Invariants

RewardVault

- sum total of staker rewards should never exceed emitted rewards
- total non-emitted rewards should never exceed the LINK token balance of the reward vault
- operator base reward bucket with zero emission rate should have zero reward end time
- community base reward bucket with zero emission rate should have zero reward end time
- operator delegated reward bucket with zero emission rate should have zero reward end time

Community Staking Pool

- total sum of staker staked LINK amounts must always equal the total staked LINK amount
- operator should not be staking in the community pool
- total staked amount should never exceed max pool size
- contract's LINK token balance should be greater than or equal to the totalPrincipal

Operator Staking Pool

- total sum of staker staked LINK amounts must always equal the total staked LINK amount
- community staker should not be staking in the operator pool
- total staked amount should never exceed max pool size
- contract's LINK token balance should be greater than or equal to the sum of totalPrincipal and s_alerterRewardFunds.

PriceFeedAlertsController

- An alert can only be raised for a Data Feed if the feed is stale
- An alert can only be raised for a Data Feed if the alerter is staking in one of the staking pools
- Only one alert can be raised for a Data Feed per round

Design Summary

Stakers

- Community Stakers earn a base reward that is proportional to their stake in the pool. Additionally, Community Stakers pay a portion of their rewards as a delegation fee to Node Operator Stakers.
- Node Operator Stakers earn a base reward **and** a delegation reward, both proportional to their stake in the pool. The delegation reward is derived from the Community Stakers' rewards.

Migration

- Existing stakers in v0.1 can manually migrate to v0.2 or choose to withdraw their staked LINK and rewards.
- v0.1 stakers will have priority access to v0.2.

Rewards

- Rewards in v0.2 are made available in fixed quantities and earned by stakers in the pool at a rate that is commensurate with how much each staker has staked.
- The amount of token amounts made available is consistent regardless of the amount of total stake in the pool, whereas in v0.1 the amount of tokens made available is dynamic to meet a target reward rate.
- Claimable rewards are claimable at any time, but total rewards are subject to a ramp up period.
- Users can initiate the withdrawal process at any point in time, and undergo an unbonding period and a claim period, with a ramp-up (multiplier growth) period. This multiplier will start at 0 when the staker stakes and caps at 1 after a given time period. This means that withdrawing before the time to reach the 1x multiplier will result in the staker forfeiting a portion of the rewards that they would have otherwise earned. The forfeited rewards will be made available amongst the stakers in the pool.

Withdrawals

- Staked LINK
    - Withdrawal of staked LINK can be initiated at any time but is subject to an unbonding period.
    - After the unbonding period, the staked LINK can then be withdrawn within a claim period. If no LINK is unstaked within the claim period, all LINK is automatically restaked. Staked LINK will still continue to accrue rewards during the unbonding and claims periods.
    - Staked LINK **does not** have to be withdrawn in full and can be partially unstaked.
    - Withdrawing staked LINK will fully reset the ramp-up multiplier regardless of the amount unstaked.

- Rewards
    - Rewards can be withdrawn at any time and **are not** subject to the unbonding period.
    - Rewards are always withdrawn in full.

Alerting

- At the launch of v0.2, the alerting conditions will be the same as v0.1. Stakers can raise an alert if they find that the ETH/USD Data Feed on Ethereum is down for more than 3 hours.
- Node operators will have priority to raise an alert and can do so during the priority round, which is the first 20 minutes after the ETH/USD Data Feed has been down for 3 hours.
- Community Stakers will then be able to raise an alert after the priority round is over.
- Affected Node Operator Stakers will see a portion of their staked LINK slashed.

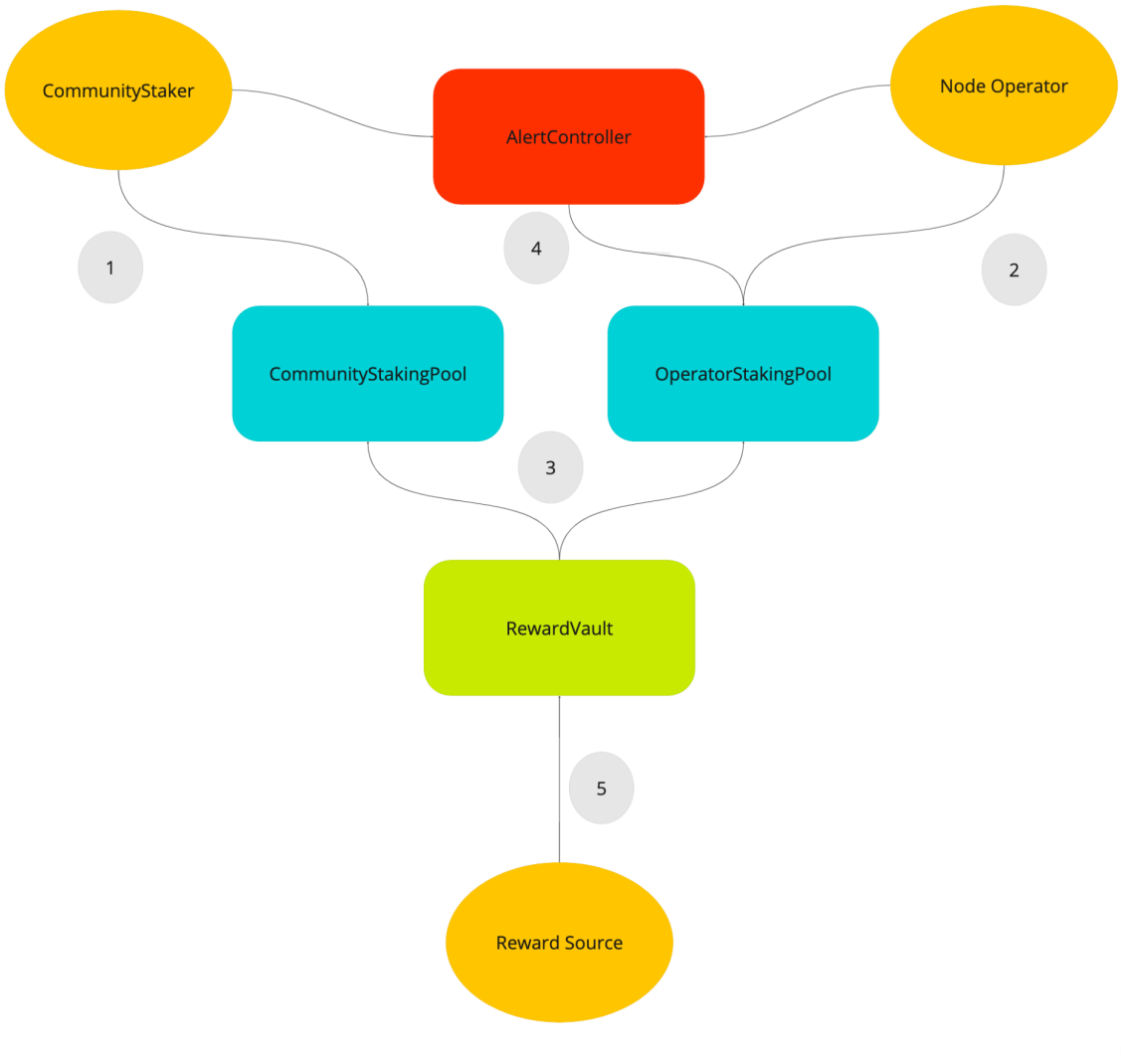
Access Controls

- Node operators can stake at any time.
- The availability for Community Stakers will involve a progressive rollout
    - **Priority Migration:** v0.1 stakers will have the opportunity to migrate their v0.1 stake and accrued LINK rewards to v0.2. v0.1 stakers can choose to migrate all of their stake, a portion of their stake, or withdraw all stake and rewards. If choosing to migrate a portion, then all non-migrated v0.1 stake or rewards during this process will be withdrawn. If a v0.1 staker takes no action, then their staked LINK and rewards will remain in v0.1 and no longer earn rewards. All v0.1 stakers have guaranteed access to v0.2 during this Priority Migration period. v0.1 stakers can attempt to migrate after the Priority Migration phase is over, but access to v0.2 will not be guaranteed, and will be dependent on availability within the pool.
    - **Early Access:** LINK token holders who meet at least one predefined criterion on an allowlist will have the opportunity to stake LINK in v0.2 up to the per-wallet maximum. Access to v0.2 is not guaranteed as the allowlist is larger than available pool space.
    - **General Access:** Anyone has the chance to stake up to the per-wallet maximum, provided that the v0.2 pool has not yet been filled

Glossary

- **Contract Manager:** The user with the role to modify certain staking parameters.
- **Juel:** The smallest unit of a LINK token. This is similar to wei being the smallest unit of ETH. (Note that 1,000,000,000,000,000,000 (1e18) Juels are equal to 1 LINK)
- **Staker**: A user (an Operator or Community Staker) who stakes in the staking pool.
    - **Node Operator Staker**: A service provider in the Chainlink ecosystem who operates the Chainlink node software that supports the operation of various oracle services and stakes LINK. Also referred to as operators.
    - **Community Staker**: A LINK token holder that is not a Node Operator and contributes to the network's cryptoeconomic security through the ability to raise alerts based on predefined performance requirements of oracle services.
- **Alerter**: A staker who raises an alert on-chain (e.g., downtime).
- **Staking Allowlist**: A list of addresses that are allowed to stake during Early Access.
- **Auto-Delegation Reward:** The portion of rewards earned by Community Stakers that is made available to Node Operator Stakers. The reward is made available proportionately amongst the Node Operator Stakers depending on how much a Node Operator Staker has staked relative to the other Node Operator Stakers.
- **Ramp-up multiplier:** A staker-specific value that determines the proportion of total rewards a staker can claim at the current time. Grows linearly from 0 to 1x.
- **Unbonding period**: A period of time stakers are required to wait before claiming unstaked LINK.
- **Claim period:** The period of time following the unbonding period during which a staker may unstake and withdraw previously staked LINK. If no LINK is unstaked within the claim period, the LINK is automatically restaked without penalty.
- **Reward duration**: The period of time an amount of rewards is made available for.

Staking v0.2 presents a modular role-based architecture that attempts to reduce the amount of logic in the staking contract so that parts of the staking program can be easily upgraded without requiring that a user migrates to a different staking contract. Whilst the accounting logic will still be implemented in the core staking contract, other logic will live in external contracts.



1. Community Stakers stake LINK into the `CommunityStakingPool`. (See: Mechanisms, Staking)
2. Node Operator Stakers stake LINK into the `OperatorStakingPool`. (See: Mechanisms, Staking)
3. The `OperatorStakingPool` and `CommunityStakingPool` contracts store the core accounting logic that tracks users' staked LINK balances but read the reward calculations from the `RewardVault` contract. The `RewardVault` contract has all the reward distribution and calculation logic.
4. `AlertsControllers` slash stakers on the staking pool contract. There can be many different `AlertsController` contracts that each check for a different alerting condition on the different Chainlink services that are secured by staking. (See: Slashing)
5. Rewards are made available to the `RewardVault` contract and are claimed by stakers through their respective staking pools. Support for other reward mechanisms can be

added in the future with newer `RewardVault` versions. Stakers can claim rewards by directly calling the `claimReward` function in each `RewardVault` contract which will be responsible to send the rewards to the staker. (See: Reward Mechanism)

StakingPool

Stakers can stake LINK into one of the `StakingPool` contracts depending on which staker type they are. The pool holds all the LINK staked by its members and stores each staker's balance along with each staker's average staked at time. In addition to storing each staker's state, a pool can also define its own set of rules that govern how its members can interact with the protocol. For instance, a pool can define:

- Minimum and maximum staking limits
- The maximum pool size
- Access controls

In v0.2, there will be a `CommunityStakingPool` and an `OperatorStakingPool` for Community Stakers and Node Operator Stakers to stake. Additional pools could be added in the future to accommodate other staker types.

IStakingPool

```
Unset
interface IStakingPool {
 function unstake(uint256 amount, bool shouldClaimReward) external;

 function getTotalPrincipal() external view returns (uint256);

 function getStakerPrincipal(address staker) external view returns
(uint256);

 function getStakerPrincipalAt(
  address staker,
  uint256 checkpointId
 ) external view returns (uint256);

 function getStakerStakedAtTime(address staker) external view
returns (uint256);

 function getStakerStakedAtTimeAt(
```

```
    address staker,
    uint256 checkpointId
  ) external view returns (uint256);

  function getRewardVault() external view returns (IRewardVault);

  function getChainlinkToken() external view returns (address);

  function getMigrationProxy() external view returns (address);

  function isActive() external view returns (bool);

  function getStakerLimits() external view returns (uint256,
uint256);

  function getMaxPoolSize() external view returns (uint256);
}
```

Relevant Sections

- Mechanisms, Staking
- Slashing
- Access Controls

RewardVault

A `RewardVault` contract receives incoming reward tokens and determines how to unlock its rewards as well as how those rewards should be made available amongst all the stakers. `RewardVault`s can be upgraded over time to change how rewards are made available without requiring stakers migrate their stake to a different pool. Each version of a `RewardVault` can define:

- Multiplier logic that can be applied to each staker's share of rewards
- For instance in v0.2, a portion of the total rewards go to the Node Operator Stakers, while the rest go to the Community Stakers.
- Certain actors that have been assigned the `REWARDER_ROLE` can call the `addReward` function to add additional rewards into the pool and have it emit over a certain period of time.

IRewardVault

```
Unset
interface IRewardVault {
 function claimReward() external returns (uint256);

 function updateReward(address staker) external;

 function finalizeReward(
  address staker,
  uint256 stakerPrincipal,
  uint256 unstakedAmount,
  bool shouldClaim
 ) external returns (uint256);

 function close() external;

 function isOpen() external view returns (bool);

 function getReward(address staker) external view returns (uint256);

 function isPaused() external view returns (bool);
}
```

Relevant Sections

- Fixed Quantity Rewards

AlertsController

The alerting logic is delegated to the `AlertsController` contracts such that when an alert is raised, the contracts check whether different alerting conditions have been met and determine which stakers to slash and how much. A `StakingPool` that implements the `ISlashable` interface grants a `SLASHER_ROLE` to an `AlertsController` that can `slash` the pool if an alerting condition is met. Initially, the `PriceFeedAlertsController` contract tracks the `ETH-USD` Data Feed on Ethereum. This `AlertsController` can support adding other alert controllers in the future. Other `AlertsController` contracts can be added in the future to support more alerting conditions for other Chainlink services.

ISlashable

A `StakingPool` that can slash its members should implement the `ISlashable` interface and give an `AlertsController` the `SLASHER_ROLE` so that it can call `slash`. Slashing is currently only performed on the operator's staked LINK and **does not slash** the operator's earned rewards.

```
Unset
 function slash(
  address[] calldata operators,
  address alerter,
  uint256 principalAmount,
  uint256 alerterRewardAmount
 ) external;
}
```

SLASHER_ROLE

- Given to addresses that call the `slash` function to slash node operators when an alerting condition is met.
- This will typically be given to an alerting smart contract (`AlertController`) that alerters will interact with.
- Example flow
    1. An alerter calls a `raiseAlert` function in the `AlertController`.
    2. The `AlertController` does the following
        1. Checks to see if an alerting condition has been met.
        2. Determines which stakers to slash and how much each staker should be slashed.
        3. Slashes the stakers by calling `slash` on the `StakingPool` contract.
    3. `StakingPool` contract slashes the affected Node Operators and transfers some LINK rewards to the alerter. **The staking contract should revert the transaction if the alerter has not staked.**

IAlertsController

A staker can call `canAlert` on an `AlertsController` to check if the alerting condition has been met and whether the staker is eligible for raising an alert at the moment. For v0.2, the bytes data will consist of an abi-encoded address of the ETH-USD feed. If a `canAlert` returns

true for a staker, they should be able to call `raiseAlert`, slash operators, and get the alerter rewards.

```
Unset
interface IAlertsController {
 function raiseAlert(bytes calldata data) external;

 function canAlert(address alerter, bytes calldata data) external
view returns (bool);

 function getStakingPools() external view returns (address[]
memory);
}
```

IAlertsControllerOwner

The contract manager can set the slashable operators for the AlertsController. These are the addresses that get slashed when an alert is successfully raised.

```
Unset
interface IAlertsControllerOwner {
 function setSlashableOperators(address[] calldata operators, bytes
calldata data) external;

 function getSlashableOperators(bytes calldata data) external view
returns (address[] memory);
}
```

Mechanisms

Staking

Motivation

Staking is the primary action that users will execute throughout the program in order to lock in their LINK tokens to earn rewards for increasing the cryptoeconomic security of services.

Staking Limits

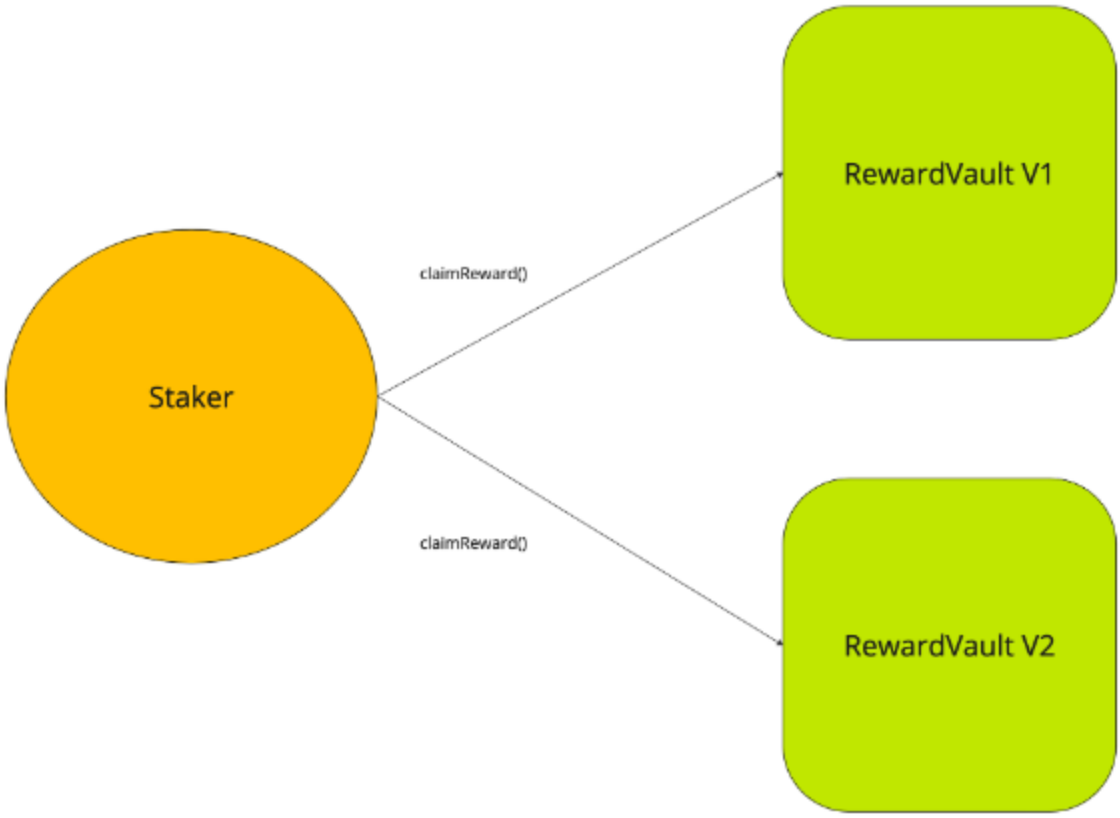
Similar to v0.1, the v0.2 staking pool will have a capped pool size. A portion of the pool will be reserved for Node Operators Stakers whilst the rest can be filled up by the Community Stakers. Additionally, there will be individual staker limits for both Node Operator Stakers and Community Stakers.

Variable	Can be increased?	Can be decreased?
Maximum Pool Size	Yes	No
Maximum Node Operator Stake	Yes	No
Maximum Community Staker Stake	Yes	No

In order to achieve the maximum limit across the entire protocol, each `StakingPool` will configure a maximum limit that adds up to the total. These limits will be stored in each `StakingPool`'s `PoolConfigs` struct.

Staking Token Flow

Stakers will call the ERC677 `transferAndCall` function on the LINK token contract in order to stake LINK tokens in the pool. `transferAndCall` was reused so that the user does not have to execute 2 separate transactions to `approve` and `stake` their LINK tokens. The staking flow works as follows:



1. Staker calls `transferAndCall` on the LINK contract and passes in the required parameters.
    1. `_to` The address of the `StakingPool` contract the staker is staking LINK to.
    2. `_value` The amount of LINK to stake.
    3. `_data` The ABI encoded data that will be passed along to the Staking v0.2 contract. This will be used by the staker to prove that they can stake LINK in the target pool.
2. LINK is transferred the staker's address to the `StakingPool` contract and a call is made to the `onTokenTransfer` function in the `StakingPool` contract.
3. `onTokenTransfer` validates that the staker is eligible to stake and that the amount of staked LINK is within the pool's minimum and maximum limits.

Staker Balances

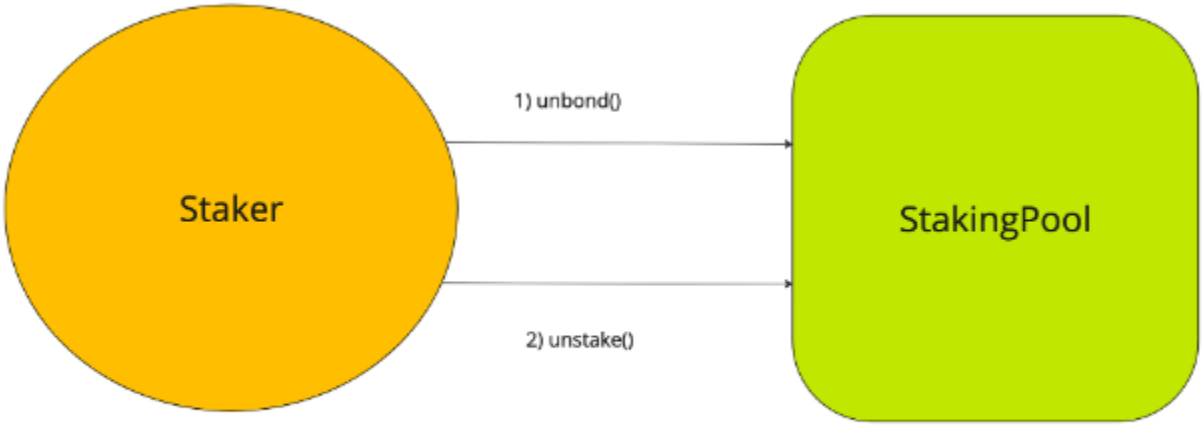
A `StakingPool` will keep track of each staker's state by mapping the staker's address to a `Staker` struct. The `Staker` struct has two fields `principalHistory` and `stakedAtTimeHistory`, which track the staker's staked LINK and staked at time history.

```
Unset
struct Staker {
 Checkpoints.Trace224 principalHistory;
 Checkpoints.Trace224 stakedAtTimeHistory;
}
```

Unstaking

Motivation

In order for a staker to unstake their staked LINK, the staker must first initiate an unbonding period. This is then followed by a claim period in which the staker is able to unstake their staked LINK. Any LINK that is not unstaked during the claim period remains staked. After the claim period, the staker must go through another unbonding period before withdrawing their remaining staked LINK.



Unbonding Period

Stakers can initiate the unbonding period by calling the `unbond` function on the `StakingPool` contract. This function writes the time the unbonding period ends at to the `unbondingEndsAt` in the staker's `Staker` struct and emits an event. In order to determine the time unbonding should end, the contract can read from the pool's `PoolState` struct. This function reverts if called whilst the staker is in the unbonding period and if the `msg.sender` is not a staker.

Pseudocode

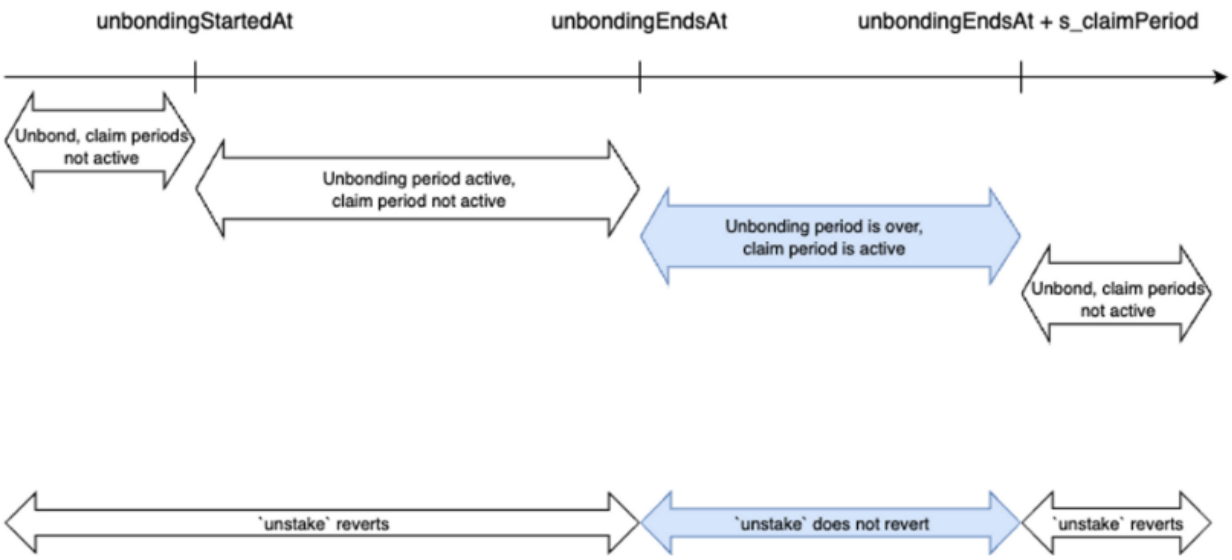
```
Unset
function unbond() external {
  IStakingPool.Staker storage staker = s_stakers[msg.sender];
  if (staker.principal == 0) revert StakeNotFound(msg.sender);

  uint256 stakerUnbondingEndsAt = s_unbondingEndsAt[msg.sender];
  if (stakerUnbondingEndsAt > 0 && block.timestamp <
stakerUnbondingEndsAt + i_claimPeriod) {
   revert UnbondingPeriodActive(stakerUnbondingEndsAt);
  }

 s_unbondingEndsAt[msg.sender] = block.timestamp +
s_pool.configs.unbondingPeriod;
 emit UnbondingPeriodStarted(msg.sender);
}
```

Staked LINK Withdrawals

Once the unbonding period has passed, a staker will have a claim period to call `unstake` to unstake their LINK. The staker is allowed to partially unstake their LINK one or more times up to their staked LINK amount. Additionally, the `unstake` function accepts a boolean flag that indicates whether the staker wants to also claim the rewards or not. If the flag is true, all of the earned rewards will be withdrawn along with the staked LINK.

In order to prevent the staker from withdrawing outside the claim period, the `unstake` function will revert if any of the following are true.

- Unbonding period is not active and the claim period is not active
  - `s_unbondingEndsAt[msg.sender] == 0`
- Unbonding period is active and the claim period is not active
  - `s_unbondingEndsAt[msg.sender] > block.timestamp`
- Unbonding period is not active and the claim period is over
  - `s_unbondingEndsAt[msg.sender] + i_claimPeriod < block.timestamp`
  - If this condition is true then the staker will need to call `unbond` again to restart the unstaking process.



In addition to unstaking the staker's staked LINK, the contract will also reset the multiplier by setting the staker's `averageStakedAtTime` to the current `block.timestamp`.

The current design also fully resets the unbonding period when a staker stakes an additional amount of LINK.

Example scenario

- Parameters
  - `unbondingPeriod = 30 days`
- T = 1
  - User unbonds their stake of 100 LINK.
  - User should be able to unstake after 30 days (Day 31).
- T = 21
  - User stakes 30 LINK 20 days into their unbonding period.
  - Their unbonding period is reset. The staker will need to unbond again to unstake.

Mutable Unbonding and Claim Periods

The current design allows the contract manager to call `setUnbondingPeriod` to change the unbonding time and `setClaimPeriod` to change the claim period. Changing the unbonding and/or claim time will only affect future calls to unbond and will not affect the unbonding and claim period times for stakers who are already in the unbonding process. This is currently done by immediately setting the staker's `unbondingEndsAt` timestamp to `block.timestamp + s_pool.configs.unbondingPeriod` inside the `unbond` function and the staker's `claimPeriodEndsAt` to `block.timestamp + s_pool.configs.unbondingPeriod + s_pools.configs.claimPeriod`. In order to prevent the contract manager from setting an

unbonding and claim period that is too low or too high, the contract is initialized with a minimum and maximum unbonding/claim period.

Example scenario

- Parameters
  - `unbondingPeriod = 30 days`
- T = 1
  - User unbonds their stake of 100 LINK.
  - User should be able to unstake after 30 days (Day 31)
- T = 10
  - Contract manager changes unbonding period parameter
  - `unbondingPeriod = 7 days`
- T = 17
  - 7 days after contract manager changed unbonding period
  - User is still unable to unstake
- T = 29
  - Staker unstakes 100 LINK

Staking During The Unbonding/Claim Period

A staker who stakes during the unbonding or claim period will reset their state and have to restart the unbonding process to unstake their staked LINK. This is done by deleting the staker's entry in the `s_unbondingEndsAt` and `s_claimPeriodEndsAt` mappings.

Sample code

```
Unset

function onTokenTransfer(address sender, uint256 amount, bytes
memory data)
      external
{
      if (s_unbondingEndsAt[staker] > 0) {
   delete s_unbondingEndsAt[staker];
   delete s_claimPeriodEndsAt[staker];
   emit UnbondingPeriodReset(staker);
  }
      ...Other staking logic
}
```

Forfeited Rewards

If a staker unstakes before their ramp-up multiplier reaches 1, the staker will forfeit some of their earned rewards depending on their current multiplier and the amount they unstake relative to their current staked LINK. This is calculated using the following formula:

$$stakerEarnedRewards = \frac{stakerPrincipal * emittedRewards}{totalPoolStakedLINKAmount}$$

$$forfeitedRewrds = \frac{unstakeAmount * (1 - multiplier) * stakerEarnedRewards}{stakerStakedLINKAmount}$$

The amount of forfeited rewards is then made available amongst the stakers in the pool by incrementing the reward bucket's `vestedRewardPerToken` variable

```
Unset
rewardBucket.vestedRewardPerToken += missedRewards / totalStakedInPool;
```

Claiming Rewards When Unstaking

In addition to unstaking their staked LINK, stakers can also call the `unstake` function with the `shouldClaimReward` flag set to `true` to claim the rewards earned from the currently configured rewards vault in the same transaction. The staker must still call `claimReward` from previous reward vaults if they want to claim any earned rewards in those vaults.

Fixed Quantity Rewards

Motivation

Rewards in v0.2 will be funded in intervals and made available amongst the stakers. The amount of rewards each staker is able to claim will be proportional to the amount they have staked relative to the amount staked in the pool.

Additionally, Staking v0.2 will also need to differentiate the rewards for both Community Stakers and Node Operator Stakers. Whilst all stakers are entitled to their proportion of rewards, Community Stakers delegate a part of their stake to Node Operator Stakers, thus giving up

some of the rewards they have earned. The amount that they give up is determined by the pool's delegation rate.

Adding Rewards

The contract manager makes available rewards through the `RewardVault` contract by calling the `addReward` function and specifying the total amount of rewards, its target reward rate, and the reward token. The rate is given by the following formula.

$$targetRewardRate = \frac{totalRewards}{totalPoolSize}$$

For example, if Staking v0.2 has a maximum pool size of 35 million LINK and a target reward rate of 5%, then the contract manager would need to add 1.75 million LINK of rewards made available over one year. Note that whilst the **example** target reward rate is 5%, stakers would earn more or less than the target reward rate depending on how full the staking pool is. For instance stakers would earn more in a pool that is not full as the 1.75 million LINK would be made available to a lower number of staked LINK.

Rewards will be split into 3 buckets, namely a `communityBaseRewardsBucket`, `operatorBaseRewardsBucket` and a `operatorDelegationRewardsBucket`. The contract will calculate the proportionate amount of rewards for each staker type based on their respective max pool sizes. The calculation for the reward amount for a staking pool is as follows:

$$operatorBaseRewardAmount = \frac{totalRewardsAdded * operatorMaxPoolSize}{operatorMaxPoolSize + communityStakerMaxPoolSize}$$

$$communityStakerBaseRewardAmount = \frac{totalRewardsAdded * communityStakerMaxPoolSize}{operatorMaxPoolSize + communityStakerMaxPoolSize}$$

In addition to the above formula, some of the base rewards earned by the community stakers will be made available to the Node Operators as delegation rewards. This is calculated using the reward vault's `s_delegationRateDenominator`.

$$operatorDelegationRewardAmount = \frac{totalRewardsAdded * communityStakerMaxPoolSize}{delegationRateDenominator * (operatorMaxPoolSize + communityStakerMaxPoolSize)}$$

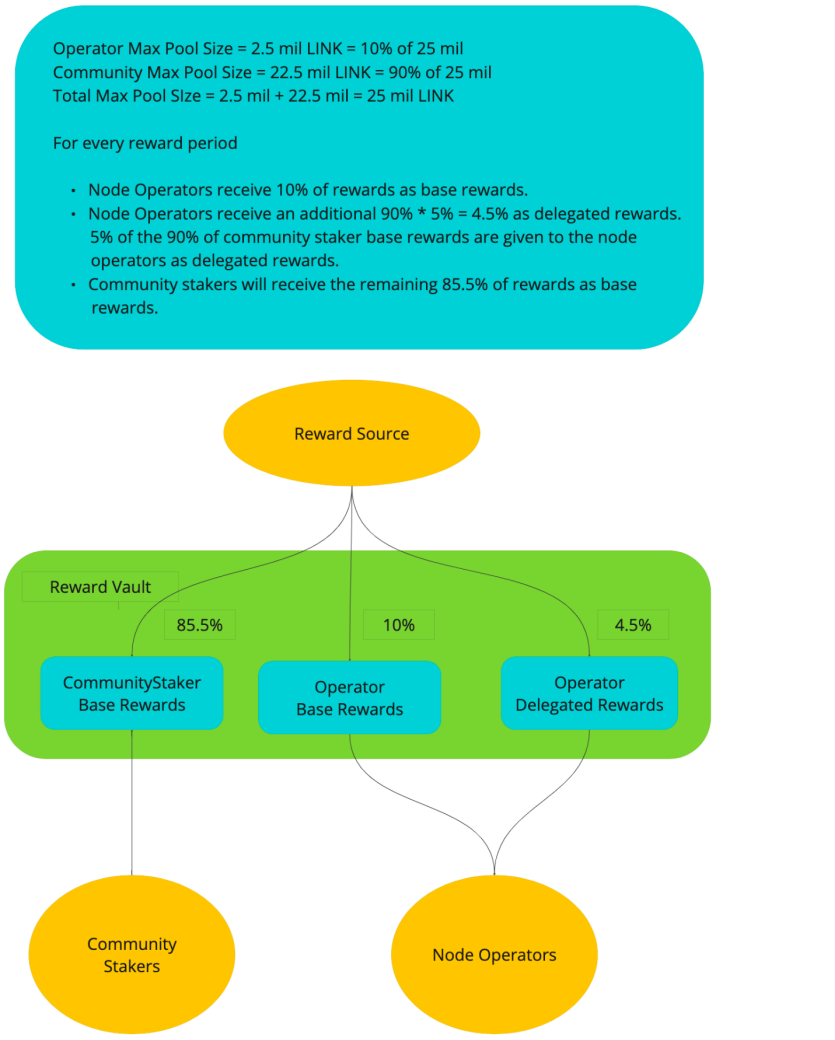
$$communityStakerBaseRewardsBucketAmount = communityBaseRewardsBucketAmount - operatorDelegationRewardAmount$$

The reward duration and the end times of each pool are also calculated at the time rewards are added. The calculations are as follows:

$$rewardDuration = \frac{poolRewardAmount + poolRemainingRewards}{poolEmissionRate}$$

$$rewardDurationEndsAt = block.timestamp + rewardDuration$$

Diagram with example parameter configs illustrating how rewards would be split between the buckets.



The rewards in the `RewardVault` will be split amongst 3 buckets. These are:

1. Community Staker Base Rewards that hold all the rewards that can be claimed by a Community Staker.

$$communityStakerReward = \frac{multiplier * communityStakerStakedLINKAmount * emittedCommunityStakerBaseRewards}{totalCommunityStakerStakedLINKAmount}$$

2. Operator Base Rewards that holds all the rewards that can be claimed by a Node Operator Staker. The rewards an operator can claim is proportional to their stake relative to other operators.

$$operatorReward = \frac{multiplier * operatorStakedLINKAmount * emittedOperatorBaseRewards}{totalOperatorStakedLINKAmount}$$

3. Operator Delegation Rewards that holds all the delegation rewards that can be claimable by a Node Operator Staker. **Note that delegation rewards are not affected by a staker's multiplier.**

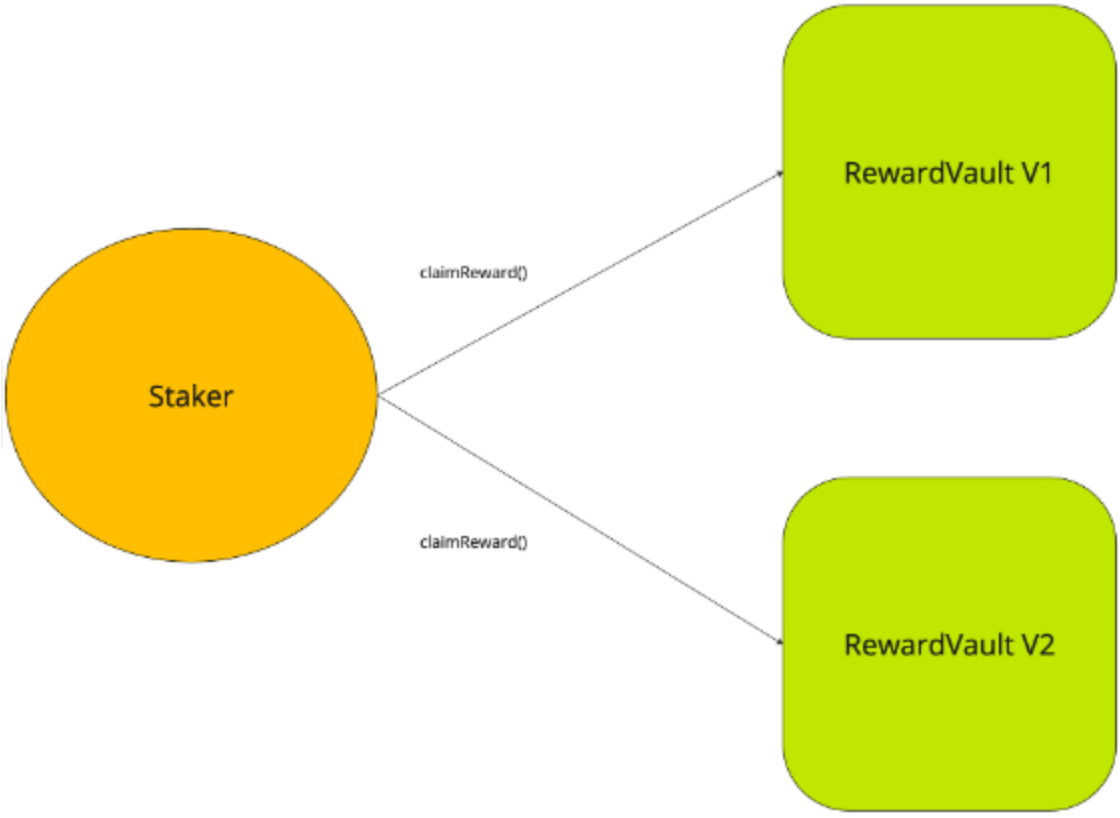
$$operatorReward = \frac{operatorStakedLINKAmount * emittedDelegatedRewards}{totalOperatorStakedLINKAmount}$$

Claiming Rewards

Stakers can claim their rewards by calling the `claimReward` function on the `RewardVault` contract they wish to claim rewards from. The vault calculates the amount of rewards the staker has earned and transfers it **from the `RewardVault`** to the staker. A later section outlines how the `RewardVault` calculates the rewards for each staker. In order to protect the protocol from a malicious actor passing in a malicious `token` address, future `RewardVaults` may store a list of eligible token addresses. This is however outside the scope of v0.2 as the `token` parameter is unused due to the v0.2 reward vault only providing LINK token rewards.

`claimReward`

A staker can claim rewards from the current of previous reward vaults by calling `claimReward` directly on each vault.



Ramp-up Multipliers

Motivation

The multiplier feature leverages incentives and penalties around staking rewards.

Multiplier calculation

Each staker's reward multiplier starts at 0 and ramps up to 1 over a set duration of time. We store the history of the times a staker stakes in the `Staker` struct's `stakedAtTimeHistory` field in the StakingPool contract, and the time to reach the maximum multiplier is stored as a storage variable, `multiplierDuration`, in the `RewardVault` contract.

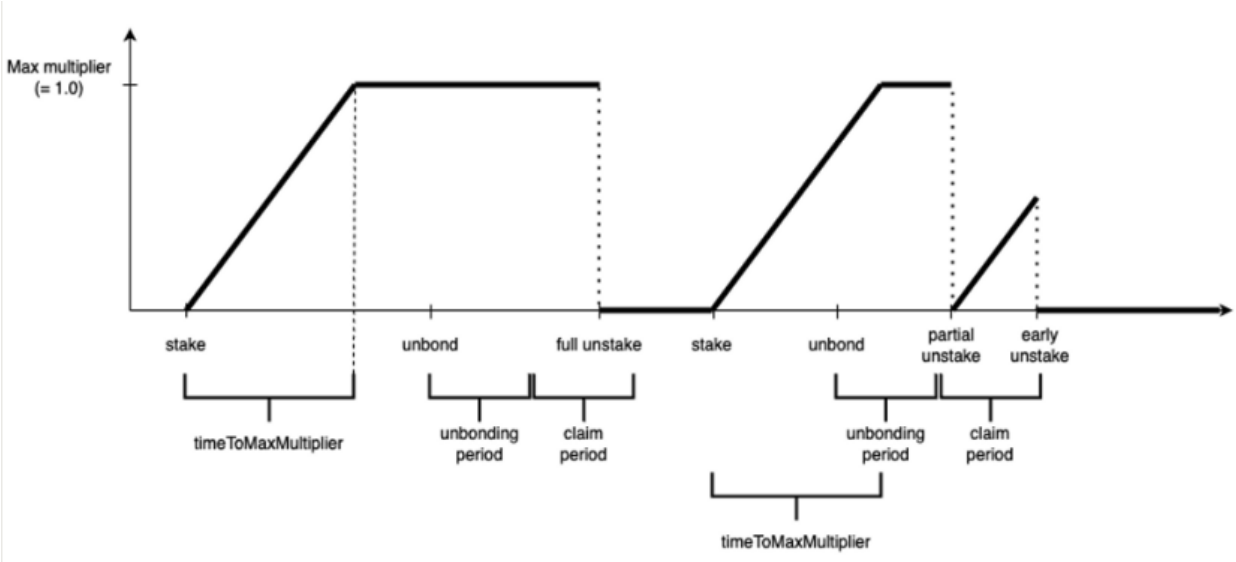
To preserve the precision, this function returns a value between 0 and 1e18 (a wad). We'll need to divide by 1e18 when we apply the multiplier to the rewards.

$$multiplier = min(\frac{(block.timestamp - latestStakerStakedAtTime) * 10e18}{multiplierDuration}, 10e18)$$

How unstaking affects multipliers

In v0.2, a staker can fully or partially unstake their staked LINK, and in both cases, the multiplier should be reset to 0 at the time of withdrawal. We achieve this by pushing a new entry with a value of `block.timestamp` into the staker's `stakedAtTimeHistory`.

- When a staker unstakes their staked LINK by calling `unstake` after the unbonding period (multiplier is **reset**):
  - `stakerReward.finalizedBaseReward += staker.storedBaseReward * multiplier`
  - `stakerReward.storedBaseReward = 0`
  - `staker.stakedAtTime = block.timestamp`
    - This makes `multiplier = (block.timestamp - block.timestamp) / multiplierDuration = 0`, which shows that it's been reset.
- When a staker claims rewards by calling `claimReward` (multiplier is **not** reset):
  - With regards to rewards specifically, only full reward withdrawals are supported, so we transfer the sum of `stakerReward.finalizedBaseReward` and `stakerReward.storedBaseReward` to the staker and reset those values to zero. In addition to the base rewards, the `RewardVault` will also transfer the staker's `stakerReward.finalizedDelegatedBaseReward` if the staker is an operator.
  - `staker.stakedAtTime` is not affected when the rewards are withdrawn.
-



Pausing multipliers

The multipliers will not be paused during a staker's unbonding/claim period, during a contract pause, or after it's closed.

Reward calculations with multipliers

- There are two types of rewards:
  - Base reward that is earned by both Community Stakers and Operators
  - Delegation reward that is only earned by Operators. **Multipliers do not apply to the delegation rewards.**
- Rewards are calculated per staked token and stored in the `vestedRewardPerToken` variable within the `RewardBucket` of each rewards bucket (Operators' base rewards, delegated rewards, Community Stakers' base rewards). These values need to be updated whenever a new reward event is added or staked LINK amounts change (users stake/unstake).

```
Unset
vestedRewardPerToken +=
  (block.timestamp - lastUpdateTime) * emissionRate * 1e18 /
totalPrincipal;
```

- In addition to the accumulated per-token reward of the bucket, an individual staker's per-token reward needs to be accumulated and stored as well. This value indicates the per-token reward at the moment the staker first joined or when the staker's reward was

last accrued. When the staker first joins the staking pool, the `baseRewardPerToken` and `operatorDelegatedRewardPerToken` represent the “missed” rewards of the staker. After the value gets updated whenever the staker stakes more/unstakes/claims rewards, the reward is accrued in the `storedReward`, and the `baseRewardPerToken` and `operatorDelegatedRewardPerToken` become the “missed + already accrued” rewards.

- Staker’s reward is updated when `onTokenTransfer`, `unstake`, `claimReward`, `slash`, or `removeOperators` is called. A staker’s accrued reward since the last update is calculated with the following formula and gets stored in `storedReward`.

```
Unset
storedBaseReward +=
      principal * (poolVestedRewardPerToken - stakerRewardPerToken)
```

- We accrue the unclaimed staker’s base rewards in `storedBaseReward`, without applying the multiplier. The multiplier is only applied at the time of withdrawal or unstake. Exceptions are 1) when a staker unstakes partially and 2) the delegation rewards. When a staker partially unstakes, the multiplier (staked at time) is reset, so the remaining rewards accrued up to that moment should be stored separately with the multiplier before the reset. This value is added to the `finalizedBaseReward`.

- When we calculate the accrued multiplier-applied rewards (when `claimReward` or `slash` is called), we can calculate the unused rewards `(1 - multiplier) * reward` and reclaim and share these.

Example scenario with placeholder values

- emissionRate = 1 (1 LINK/sec)
- s_timeToMaxMultiplier = 100
- t = 0
    - staker1 stakes 10
    - cumulativeRewardPerToken = 0
    - lastUpdateTime = 0
    - staker1
        - staker1.storedRewards = 0
        - staker1.rewardPerToken = 0
        - staker1.averageDepositTime = 0
        - multiplier = 0

        - staker1.principal = 10
    - totalPrincipal = 10
- t = 1
    - staker2 stakes 10
    - cumulativeRewardPerToken = 0 + 1 * 1 / 10 = 0.1
    - lastUpdateTime = 1
    - staker1 *// what staker1 would get if they were to claim rewards.* `storedRewards` *not actually updated at this point*
        - staker1.storedRewards = 0 + 10 * (0.1 - 0) = 1
        - multiplier = min((1 - 0)/100, 1) = 0.01
    - staker2
        - staker2.storedRewards = 0
        - staker2.rewardPerToken = 0.1
        - staker2.averageDepositTime = 1
        - multiplier = 0
        - staker2.principal = 10
    - totalPrincipal = 20
- t = 2
    - staker2 stakes another 20
    - cumulativeRewardPerToken = 0.1 + 1 * 1 / 20 = 0.1 + 0.05 = 0.15
    - lastUpdateTime = 2
    - staker1 *// what staker1 would get if they were to claim rewards. storage not actually updated at this point*
        - staker1.storedRewards = 0 + 10 * (0.15 - 0) = 1.5
        - multiplier = min((2 - 0)/100, 1) = 0.02
    - staker2
        - staker2.storedRewards = 0 + 10 * (0.15 - 0.1) = 0.5
        - staker2.rewardPerToken = 0.15
        - staker2.averageDepositTime = 1 + (20/30) * (2 - 1) = 1.667
        - multiplier = min((2 - 1.667)/100, 1) = 0.00333
        - staker2.principal = 30
    - totalPrincipal = 40
- t = 3
    - staker1 unstakes 10
    - cumulativeRewardPerToken = 0.15 + 1 * 1 / 40 = 0.175
    - lastUpdateTime = 3
    - staker1
        - multiplier = min((3 - 0)/100, 1) = 0.03
        - staker1.storedMultiplierAppliedRewards = 10 * (0.175 - 0) * 0.03 = 0.0525
        - staker1.rewardPerToken = 0.175
        - staker1.averageDepositTime = 3
        - new multiplier = min((3 - 3)/100, 1) = 0
        - staker1.principal = 0

    - staker2 *// what staker2 would get if they were to claim rewards. storage not actually updated at this point*
        - staker2.storedRewards = 0.5 + 30 * (0.175 - 0.15) = 1.25
        - multiplier = min((3 - 1.667)/100, 1) = 0.01333
    - totalPrincipal = 30
- t = 4
    - staker1 restakes 30
    - cumulativeRewardPerToken = 0.175 + 1 * 1 / 30 = 0.20833
    - lastUpdateTime = 4
    - staker1
        - staker1.storedMultiplierAppliedRewards = 0.0525 // no change
        - staker1.storedRewards = 0 + 0 * (0.20833 - 0.175) = 0
        - staker1.rewardPerToken = 0.20833
        - staker1.averageDepositTime = 3 + (30/30) * (4 - 3) = 4
        - multiplier = min((4 - 4)/100, 1) = 0
        - staker1.principal = 30
    - staker2 *// what staker2 would get if they were to claim rewards. storage not actually updated at this point*
        - staker2.storedRewards = 0.5 + 30 * (0.20833 - 0.15) = 2.25
        - multiplier = min((4 - 1.667)/100, 1) = 0.02333

Sample code

```
Unset
function _updateRewards() {
      // Calculate and update the all reward buckets' rewards per token accumulated
      // Update lastUpdateTime
}

function _updateStakerRewards(Staker storage stakerInfo) private {
      // Calculate and update user's accrued rewards & rewards per token
}

function onTokenTransfer(address sender, uint256 amount, bytes memory) external {
      _updateRewards();

      Staker storage staker = getStake(sender);
```

```
        _updateStakerRewards(staker);

        // Increase the amount-weighted average deposit time
        stakerInfo.averageDepositTime =
getAverageDepositTimePostDeposit(stakerInfo, amount);
        // Increase staked amounts
}

function unstake(uint256 amount) {
        // Check unbonding & claim periods

        _updateRewards();

        Staker storage staker = getStaker(msg.sender);
        _updateStakerRewards(staker);

        // Reset staker's average deposit time
        stakerInfo.averageDepositTime = block.timestamp;
        // Decrease staked amounts

        LINK.transfer(address(this), msg.sender, amount);
}

function claimRewards(uint256 amount) {
        _updateRewards();

        Staker storage staker = getStaker(msg.sender);
        _updateStakerRewards(staker);

        // Decrease staker rewards
        // Move the unused rewards (i.e. the (1 - multiplier) portion)
back to the reward pool

        LINK.safeTransferFrom(address(rewardVault), msg.sender,
amount);
}
```

Slashing

Motivation

v0.2 increases the security provided by slashing a Node Operator Staker's staked LINK instead of their rewards. Slashing staked LINK is a more severe penalty because, in addition to losing a fixed quantity of LINK, the Operator will also lose a portion of their future rewards due to them losing a share of the pool.

Fixed Slashing Amount

In the case of a slashing event, the amount slashed does not vary based on the staked LINK amounts. The following table depicts the consequence of slashing in relation to staked LINK and reward size using example config values for minimum, maximum, and slash amounts:

	Minimum Node Operator Stake	**Average Node Operator Stake**	**Maximum / Median Node Operator Stake**
Amount staked (LINK)	1,000 LINK	30,000	50,000
Fixed Slash Amont total (LINK)	200 LINK	200 LINK	200 LINK
Slashing quantity as a % of stake	10%	0.67%	0.4%

Rate Limited Slashing

A safeguard has been put in place to not give slashers the ability to completely wipe an operator's stake by calling the slash function in quick succession while a feed is down.

Each slasher will have a slash capacity that must be refilled over time. When attempting to slash over this capacity, the contract will revert until sufficient capacity has been refilled (controlled by the slasher's refill rate).

If a new slasher config is set, the current slash capacity is refilled to full capacity.

Pseudocode

Unset

```
 struct SlasherConfig {
  uint256 refillRate;
  uint256 slashCapacity;
 }

 struct SlasherState {
  uint256 lastSlashTimestamp;
  uint256 slashCapacityUsed;
 }

 /// @inheritdoc ISlashable
 function setSlasherConfig(
  address slasher,
  SlasherConfig calldata config
 ) external override onlyOwner {
   if (config.slashCapacity == 0 || config.refillRate == 0) {
    revert ISlashable.InvalidSlasherConfig();
   }

   s_slasherConfigs[slasher] = config;

            // refill capacity
   s_slasherState[slasher].slashCapacityUsed = 0;
   s_slasherState[slasher].lastSlashTimestamp = block.timestamp;
 }

 function getSlashAllowance(address slasher) public view returns
(uint256) {
  return s_slasherConfigs[slasher].slashCapacity -
_getSlashCapacityUsed(slasher);
 }

 function _getSlashCapacityUsed(address slasher) private view
returns (uint256) {
  uint256 refilledAmount = (block.timestamp -
s_slasherState[slasher].lastSlashTimestamp)
   * s_slasherConfigs[slasher].refillRate;
```

```
    return refilledAmount > s_slasherState[slasher].slashCapacityUsed
      ? 0
      : s_slasherState[slasher].slashCapacityUsed - refilledAmount;
  }
```

- **Example with placeholder configurations**

(max slash capacity = 100 LINK and rate = 1 LINK/sec)

- t = 0
  - slash allowance = 100 LINK
- t = 10, slash event for 10 LINK
  - slash allowance -= 10 LINK => 90 LINK
- t = 11, slash another 10 LINK
  - slash allowance before = 100 LINK - 10 LINK + 1 LINK = 91 LINK
  - slash allowance after -= 10 LINK => 81 LINK
- t = 20, slash 100 LINK
  - slash allowance before = 100 LINK - 20 LINK + 9 LINK = 89 LINK
  - can't slash (100 LINK > 89 LINK), revert
- t = 31, slash 100 LINK
  - slash allowance before = 100 LINK - 20 LINK + 20 LINK = 100 LINK
  - slash allowance after = 100 LINK - 100 LINK => 0 LINK

PriceFeedAlertsController

In staking v0.2 the condition for slashing works similarly as v0.1. When Data Feed is down for a specified amount of time, an alert can be raised which will trigger slashing.

The `PriceFeedAlertsController` checks a Data Feed being secured, and compares its `lastUpdatedAt` field to the current block timestamp. If the time difference is greater than or equal to a preconfigured staleness threshold, the feed is considered down and the slashing condition has been met. This section outlines how the alert controller will determine a feed being down.

Pseudocode

The slashing mechanism is triggered by the `PriceFeedAlertsController` contract:

```
Unset
function raiseAlert(bytes calldata data) external
override(IAlertsController) {
  if (!_canAlert(msg.sender, data)) revert AlertInvalid();
  address feed = abi.decode(data, (address));
  (uint256 roundId,,,,) =
AggregatorV3Interface(feed).latestRoundData();
  s_lastAlertedRoundIds[feed] = roundId;

  // slash the operators with the values found in the feed config
  FeedConfig memory feedConfig = s_feedConfigs[feed];
  i_operatorStakingPool.slash(
   feedConfig.slashableOperators,
   msg.sender,
   feedConfig.slashableAmount,
   feedConfig.alerterRewardAmount
  );

  emit AlertRaised(msg.sender, roundId,
feedConfig.alerterRewardAmount);
 }
```

The `StakingPool` contract will then slash the staked LINK amount:

```
Unset

/// @inheritdoc ISlashable
 function slash(
  address[] calldata operators,
  address alerter,
  uint256 principalAmount,
  uint256 alerterRewardAmount
 ) external override onlySlasher {
  if (s_slasherConfigs[msg.sender].maxSlashAmount == 0) revert
ISlashable.SlasherConfigNotSet();
```

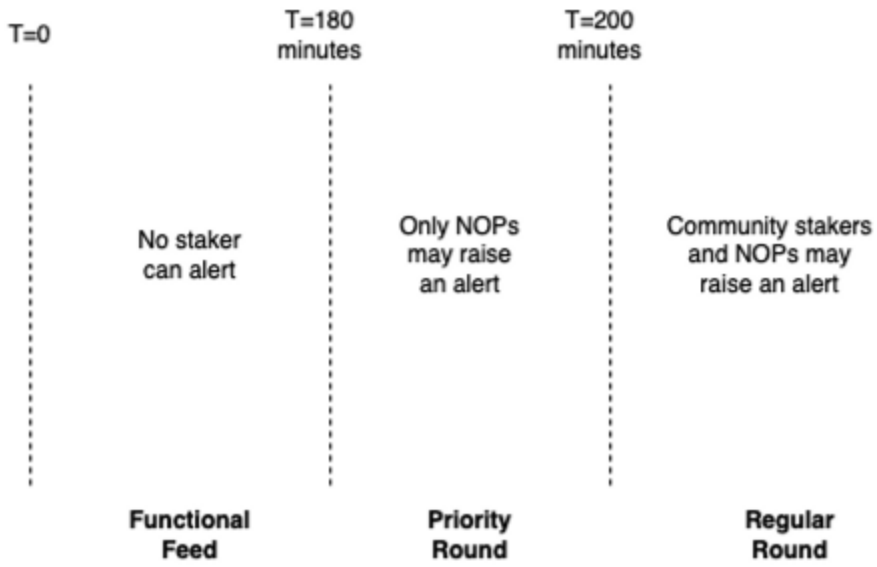
```
  uint256 totalSlashedAmount = _slashOperators(operators,
principalAmount);
  if (totalSlashedAmount > _getAllowedSlashAmount(msg.sender)) {
   revert ISlashable.SlashAmountExceedsAllowance();
  }
  s_lastSlashTimestamps[msg.sender] = block.timestamp;
  s_alerterRewardFunds += totalSlashedAmount;
  s_pool.state.totalPrincipal -= totalSlashedAmount;
  _payAlerter(alerter, totalSlashedAmount, alerterRewardAmount);
 }
```

Detecting Feed Downtime

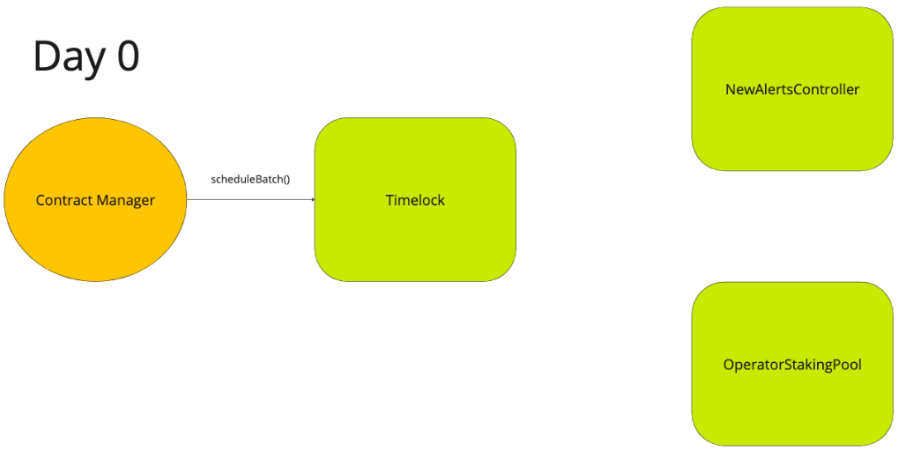
The contract manager sets a `priorityPeriodThreshold` and `regularPeriodThreshold` to specify when each staker type can raise alerts. If a feed's `lastUpdatedAt` timestamp was updated outside the `priorityPeriodThreshold` but before the `regularPeriodThreshold`, then a Node Operator Staker can raise an alert on the feed to claim a reward. If no alert is raised during the priority round then both Node Operators Staker and Community Stakers may raise an alert.

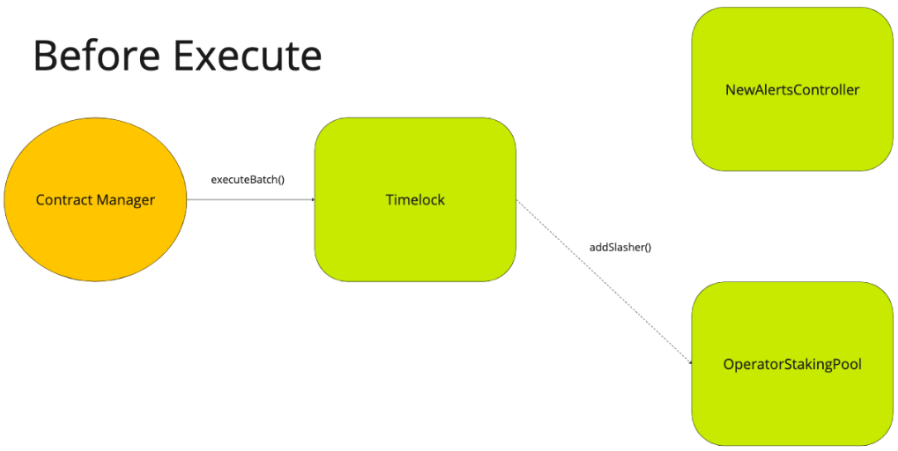
Example

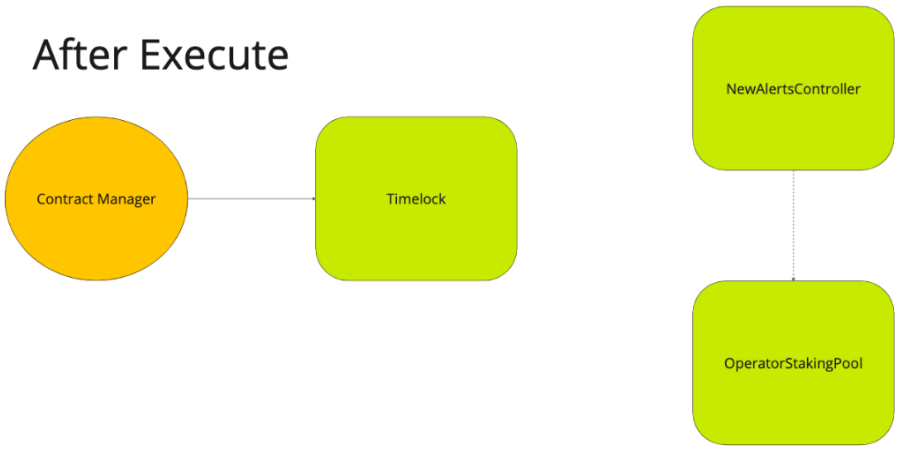
- Consider a feed that has a `priorityPeriodThreshold` of 180 minutes and a `regularPeriodThreshold` of 200 minutes.
- The feed's `lastUpdatedAt` timestamp was at T=0
- 180 minutes later, the feed's alerting system goes into the priority period and only Operators can raise an alert.
- 200 minutes after the feed's `lastUpdatedAt` timestamp was updated, the feed's alerting system enters the regular alerting period and both community stakers and Node Operators Stakers can raise an alert.



Adding a New AlertsController


Day 0
Contract Manager
scheduleBatch()
Timelock
NewAlertsController
OperatorStakingPool


Before Execute
Contract Manager
executeBatch()
Timelock
addSlasher()
NewAlertsController
OperatorStakingPool


After Execute
Contract Manager
Timelock
NewAlertsController
OperatorStakingPool

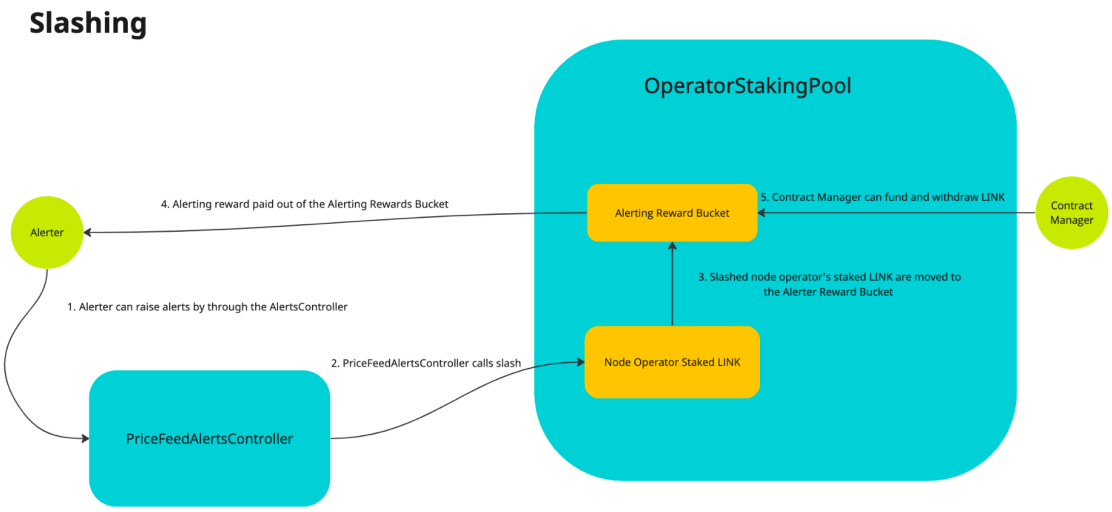
v0.2's modular design enables upgrades to the alerting mechanism to change the alerting condition, add a new feed to secure, or even support a new service to secure. This happens according to the following steps:

1. Implement and deploy a new `AlertsController` contract with the intended update to the alerting mechanism.
2. The `StakingPool` contract manager proposes the `SLASHER_ROLE` to be given to the `AlertController` contract by scheduling the `addSlasher` call to the StakingTimelock contract.
3. After a timelock longer than the unbonding period, the `StakingPool` contract manager confirms the `SLASHER_ROLE` to be given to the `AlertsController` contract by executing the scheduled `addSlasher` call.

The access control roles implementation can be inherited from the OpenZeppelin `AccessControl` contract. Each `StakingPool` contract will be able to define unique access control roles, and a base `StakingPool` contract will contain roles to be inherited by all staking pools.

Alerting Rewards

In v0.2, the `OperatorStakingPool` has an alerter reward fund that holds slashed staked KUBJ and alerter rewards. Whilst it is expected that the alerting reward is fully covered by the total slashed amount, the contract manager address can add LINK into the alerter reward funds in case there is insufficient LINK in the alerter reward funds to reward an alerter.



The movement of tokens is reflected in the updated `slash` function below.

Unset

```
function slash(
  ISlashable.SlashingParam[] calldata slashingParams,
  ISlashable.RewardParam calldata rewardParam
 ) external override onlySlasher {
  // Helper function to slash operators
  uint256 totalSlashedAmount = _slashOperators(slashingParams);

  s_alerterRewardFunds += totalSlashedAmount;
  s_pool.state.totalPrincipal -= totalSlashedAmount;

  // Helper function to pay alerter
  _payAlerter(totalSlashedAmount, rewardParam);
 }
```

Pool Entry Access Controls

Motivation

Because stakers now stake LINK into different staking pools with different rules based on type, each staking pool implements a mechanism to restrict the addresses that can stake LINK to each pool.

OperatorStakingPool

The `OperatorStakingPool` adds restrictions so that only node operators can stake in the pool. The pool can track this by storing a mapping between addresses to `Operator` structs to track an operator's accessibility. The `Operator` struct contains an `isOperator` field to track whether or not an address has been added as an operator. In addition to implementing the `IStakingPool` and `ISlashing` interface, the `OperatorStakingPool` contract should also implement the functions below to manage the list of operators.

```
Unset
// Returns the operator's mapping
function isOperator(address staker) external view returns (bool);

// Sets the operator's mapping entry to true
function addOperators(address[] memory operators) external view;

// Sets the operator's mapping entry to false
function removeOperators(address[] memory operators) external view;
```

CommunityStakingPool

The `CommunityStakingPool` starts as a access-limited pool, where only allowlisted users can stake. There are three stages for pool entry in v0.2. Stakers enter the pool in the following order:

- **Stage 1 (Priority Migration):** Stakers **only** migrating from v0.1. **v0.1 stakers will not be allowed to stake directly into community staking pool.** In this stage, the Merkle Tree will only contain the `MigrationProxy`'s address. The `MigrationProxy` is discussed in more detail in the Migration section below.
- **Stage 2 (Early Access):** Combined v0.1/v0.2 allowlisted stakers
- **Stage 3 (General Access):** Open to all stakers

Each new stage is a strict superset of the previous stage (Stage 2 also includes everyone in Stage 1, Stage 3 includes everyone in Stage 2.) At the final stage, the pool is public and open to everyone.

In addition to this the `CommunityStakingPool` must prevent Operators from staking LINK to it. This can be done by calling the `isOperator` function on the `OperatorStakingPool` and making sure that it returns `false` before allowing the staker to stake in the pool.

Merkle Trees

We use Merkle trees and proofs, similar to the Staking v0.1 Allowlist.

The root hash of this Merkle tree is stored on-chain for verification purposes.

`StakingPool`s can inherit a `MerkleAccessController` contract, which contains functions to set and read the merkle root of addresses that meet Early Access criteria.

```
Unset
interface IMerkleAccessController {
 event MerkleRootChanged(bytes32 newMerkleRoot);

 function hasAccess(address staker, bytes32[] calldata proof)
external view returns (bool);

 function setMerkleRoot(bytes32 newMerkleRoot) external;

 function getMerkleRoot() external view returns (bytes32);
}
```

- The active **merkle root** is updated on each stage change to gradually expand the allowlist with the `setMerkleRoot()` function.
- The staking contract can check if a staker is allowed to stake by calling the `hasAccess()` function.

Staking With Merkle Roots

In order to stake into an access-limited pool, the staker will need to supply a Merkle proof when they stake:

```
Unset
stake(..., bytes calldata proof)
```

This proof is verified against the active Merkle root. If it's valid, the staker can stake. If not, the transaction reverts.

Migration

This section outlines how stake migrations work.

Motivation

The migration feature enables upgrades to the Staking protocol. At the end of the Staking v0.1 program, stakers can choose to either withdraw or move their stake to a new Staking v0.2 program.

Enabling Migrations

Migrations from v0.1 to v0.2 starts as follows:

- The Staking v0.1 contract manager proposes a migration target that is the `MigrationProxy` address
- After a timelock, the Staking v0.1 contract manager commits the v0.2 migration target by calling `acceptMigrationTarget` on the Staking v0.1 contract.
- Then, the Staking v0.1 contract manager closes Staking v0.1 by calling `close`
- At this point, a staker can:
    - Withdraw their staked LINK and rewards by calling `unstake()` OR
    - Migrate to v0.2

Migration Control Flow

To migrate, stakers calls a `migrate(bytes data)` function on the v0.1 Staking contract, which sends an `ERC677.transferAndCall` to the `MigrationProxy` contract:

```
Unset
  i_LINK.transferAndCall(
   s_migrationTarget, // v0.2 address
   uint256(amount + baseReward + delegationReward),
   abi.encode(msg.sender, data)
```

```
  );
```

On the receiving end, the v0.2 `MigrationProxy` contract accepts the migration via an ERC677 `onTokenTransfer` callback:

```
Unset
 function onTokenTransfer(
  address source,
  uint256 amount,
  bytes memory data
 ) public {
  (address staker, bytes memory stakerData) = abi.decode(
   data,
   (address, bytes)
  );
  if (source == i_v01StakingAddress) {
      _migrateToPool(staker, amount, data);
  }
 }

 function _migrateToPool(address staker, uint256 amount, bytes
memory data) internal {
  address pool =
   i_operatorStakingPool.isOperator(staker) ?
address(i_operatorStakingPool) : address(i_communityStakingPool);
  i_LINK.transferAndCall(pool, amount, data);
 }
```

Handling Full and Partial Migrations

When a staker migrates, they can choose to do a full or partial migration. Their chosen migration path is specified in the `bytes data` parameter of the `migrate` function:

```
Unset
function migrate(bytes calldata data)
```

The following migration paths are available:

- An **empty** bytes data performs **a full migration**, staking both their staked LINK and rewards in v0.2.
- A **non-empty** bytes data allows the user to do a **partial migration**, where the data is**:**
    - Non migrated funds are automatically withdrawn to the staker's wallet.

```
Unset
bytes migrationData = abi.encode(**amountToStake**,
**amountToWithdraw**)
stakingv01.migrate(migrationData);
```

In the v0.2 `MigrationProxy`, the above migration data will be parsed accordingly:

```
Unset
 // Check migration data
 if (stakerData.length == 0) { // Full migration
  _migrateToPool(staker, amount, data);
 } else { // Partial migration
  (uint amountToStake, uint amountToWithdraw) = abi.decode(
   stakerData,
   (uint, uint)
  );
  require(amountToStake + amountToWithdraw == amount);
  _migrateToPool(staker, amountToStake, data); // stake a partial
amount
  i_LINK.transfer(staker, amountToWithdraw) // withdraw the rest
 }
```

Migrations-Only Phase

Initially, when the Staking v0.2 opens, only the migrating Community Stakers will be able to stake into the new staking pools. They will be able to migrate their staked LINK and rewards

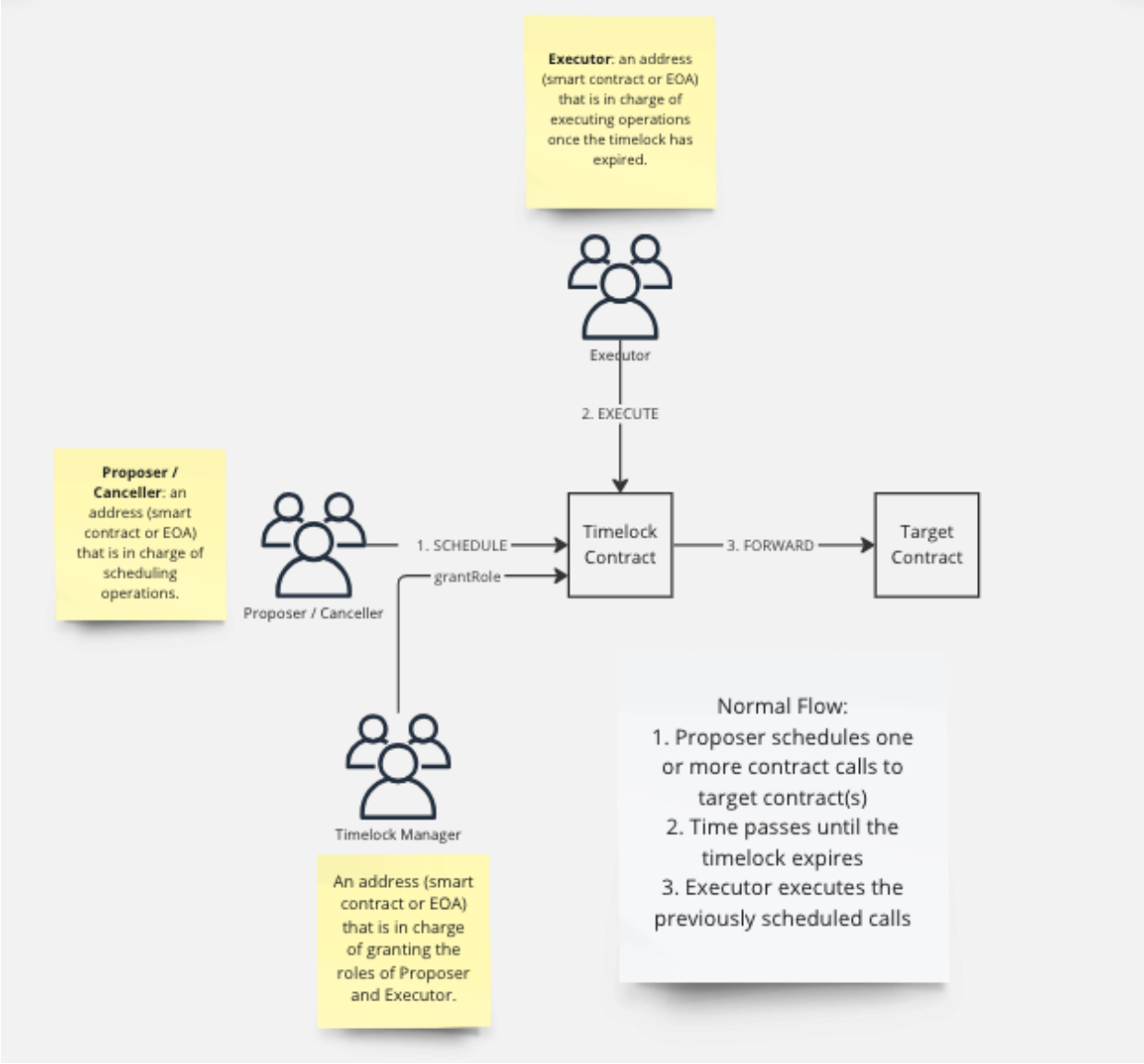
from v0.1 to v0.2 but not deposit any additional stake, even if the migrated value is less than the staker's max staked LINK and the pool hasn't reached its max staking capacity.

This phase is implemented by allowlisting only the `MigrationProxy` contract using a merkle tree. During the migrations-only phase, the allowlist (merkle tree's leaves that are used to compute the merkle root) will only contain the address of the `MigrationProxy` contract so that the stakers can only migrate from v0.1 to the v0.2 `CommunityStakingPool` through the `MigrationProxy`.

Note that Node Operator Stakers are not subject to this restriction.

Timelock

A timelock is a smart contract that delays function calls of another smart contract after a predetermined amount of time has passed.



Staking v0.2 contracts are owned by the Timelock contracts. All contract manager interactions with the Staking v0.2 contract must go through the Timelock contract and associated delay.

Operations

Motivation

The Staking v0.2 contract will have a set of operational functions that the contract manager can call to change the pool conditions.

Opening

The contract manager can call the `open` function to allow stakers to open staking into a `StakingPool`. Stakers from v0.1 will be unable to migrate their stake into a `StakingPool` until `open` has been called. Opening a pool is a one-way operation that cannot be undone after it has been called. Only allowing stakers to stake after a pool is opened gives the contract manager time to correctly configure the pool first after deploying the Staking contract. In addition to opening the pool by setting the `isOpen` flag on the `s_pool` struct to `true`, the function should revert if any of the following conditions are true:

`CommunityStakingPool`

1. The Merkle Root has not yet been set
2. Pool is already closed. See below.

`OperatorStakingPool`

1. Not enough node operators have been added to the pool.
2. Pool is already closed. See below.

```
Unset
function open() external onlyAdmin {
      // 1) Do validation checks
      // 2) Activate pool
      s_pool.isOpen = true;
      emit PoolOpened();
}
```

What functions must the contract manager call to open Staking v0.2?

1. Call `setMerkleRoot` to set the merkle root on the `CommunityStakingPool`.
2. Call `addOperators` to set the minimum number of Operators on the `OperatorStakingPool`.
3. Call `addReward` on the `RewardVault`
4. Call `open` on both the `CommunityStakingPool` and `OperatorStakingPool`.

Closing a `StakingPool`

The contract manager can close the pool by calling the `close` function. Closing a `StakingPool` allows stakers to migrate to a new `StakingPool` contract. Like `open`, calling `close` is a one-way operation that cannot be reversed.

What can stakers do after a pool is closed?

- Withdraw their staked LINK stake **without** an unbonding period.
- Continue growing their multiplier.
- Claim any claimable rewards.

What can stakers NOT do after a pool is closed?

- Cannot stake more LINK into the pool

What can the contract manager NOT do after the pool is closed?

- Cannot change any mutable variables
- Cannot call `open` to reopen the pool.

```
Unset
function close() external onlyAdmin {
      s_pool.isOpen = false;
      emit PoolClosed();
}
```

Closing a `RewardVault`

In addition to concluding a `StakingPool`, the contract manager can also call `close` on the `RewardVault`. Closing a vault is a one-way operation and can be used by the contract manager to decommission the vault by stopping rewards from emitting. In addition to this, the contract manager can also call `withdrawUnvestedReward` to withdraw any non-emitted LINK from the vault.

What can stakers do after a `RewardVault` is closed?

- Claim any claimable rewards

What can stakers NOT do after a `RewardVault` is closed?

- Stake more LINK into the `StakingPool` that are pointing to the `RewardVault`. This prevents stakers from staking more into a pool that is configured to point to a `RewardVault` that is no longer emitting any rewards.

What can the contract manager do after `RewardVault` is closed?

- Call `withdrawUnvestedReward` to withdraw any non-emitted rewards

What can the contract manager NOT do after the `RewardVault` is closed?

- Call `addReward` to add more rewards into the pool.

Pausing

In the event that there is an unforeseen issue with a contract in the Staking protocol, the contract manager should be able to pause the contract in order to limit functionality.

What can stakers do when a `StakingPool` is paused?

- Withdraw their staked LINK **without** an unbonding period.

What can stakers NOT do when a `StakingPool` is paused?

- Stake LINK into the pool
- Migrate LINK into the pool

What can the contract manager do when a `StakingPool` is paused?

- Change any mutable variables such as changing the pool limits using `setConfig`.
- Call `emergencyUnpause` in the case it's determined that there is no issue with the pool.

What can stakers NOT do when the `RewardVault` is paused?

- Cannot call `claimReward` to claim rewards from the vault.

What can the contract manager do when the `RewardVault` is paused?

- Call `close` to close the vault to allow the **contract manager** to withdraw LINK rewards.

What can the contract manager NOT do when the `RewardVault` is paused?

- Cannot make available more rewards to a paused reward vault using the `addReward` function.

In order to implement the pausing functionality, all the contracts will inherit the Pausable contract from OpenZeppelin.

```
Unset

function emergencyPause() external override(IStakingOwner)
onlyOwner {
 _pause();
}

/// @inheritdoc IStakingOwner
function emergencyUnpause() external override(IStakingOwner)
onlyOwner {
 _unpause();
}
```

Adding rewards

- Contract manager calls `addReward(pool, amount, targetEmissionRate)` on the `RewardVault` contract X days before the end of the current period.
    1. `targetEmissionRate` is set instead of the `rewardDuration` so that the emission rates of the pool and the expected reward rates of stakers can be changed.
    2. Here `targetEmissionRate` is the sum of all reward buckets' `emissionRates` (i.e., without taking into account the splits between reward buckets).
    3. Note that an emission rate is the number of LINK tokens emitted per second to the reward pools (Juels/s), whereas the reward rate is the % of rewards earned from the staked LINK.
    4. To change the pool's emission rate without adding new rewards, the `newRewards` can be set as 0 when calling `addReward()`.
    5. The pool to add rewards to can be specified. For example, setting the `stakingPool` to the zero address means rewards would be added to the entire pool, for all reward buckets. However, if `stakingPool` is set to the address of the `OperatorStakingPool`, it will only add the newRewards to the Operator pool's distribution.
- The `addReward()` calculates the exact remaining rewards at the time of execution and derives the new reward duration from the remaining rewards, new rewards, and target emission rate.

1. `remainingRewards = currentEmissionRate * (currentRewardDurationEnd - block.timestamp)`
2. `newRewardDuration = (remainingRewards + newRewards) / targetEmissionRate`
3. To illustrate how this would look with placeholder values, if we add 1.75M LINK tokens to the entire pool of size 35M, targeting the reward rate at 5%,
    1. Number of Community stakers = 22500000/7000 ≈ 3215
    2. If a Community Staker stakes 7000 and earns rewards at 5%, they earn 350 LINK per year.
    3. Total Community rewards per year = 3215*350 = 1,125,250 LINK
    4. Community base emission rate should be 1,125,250/ (365 * 24 * 60 * 60) = 0.03568144343
    5. Total emission rate should be 0.03568144343 / 85.5% = 0.04173268237 LINK/s
    6. Contract manager will call `addReward(1750000*10^18, 0.04173268237*10^18, linkTokenAddress, 0x0)`
    7. Assuming it's the beginning of v0.2 (no remaining rewards)
    8. Node Operator Stakers' base reward emission rate = 0.04173268237 LINK/s * 10% = 0.004173268237 LINK/s
    9. Community Stakers' base reward emission rate = 0.04173268237 LINK/s * 90% * 95% = 0.03568144343 LINK/s
    10. Delegated reward emission rate = 0.04173268237 LINK/s * 90% * 5% = 0.001877970707 LINK/s
    11. Reward Duration = 1750000 / 0.04173268237 = 41,933,561.434766696 seconds ≈ 485 days
- LINK tokens move from the contract manager and into the `RewardVault`.

Example scenario with placeholder values

- T = 0. Start
    1. deploy
        1. Operator max pool size = 1000
        2. Community Staker max pool size = 9000
        3. total max pool size = 10000
    2. addReward: amount = 1000 LINK, emission rate = 10 LINK/s, to all pools
        1. Operator pool emission rate = 10 * (1000/10000) = 1 LINK/s
        2. Community Staker pool emission rate = 10 * (9000/10000) = 9 LINK/s
        3. reward duration = 1000 / 10 = 100 s
        4. reward duration end time = 0 + 100 = **100 s**
- T = 10. Add additional rewards before the reward duration has ended, without changing emission rates
    1. remaining rewards
        1. Operator: (100 - 10) * 10 * (1000/10000) = 90 LINK

        2. Community: (100 - 10) * 10 * (9000/10000) = 810 LINK
    2. addReward: amount = 1000 LINK, emission rate = 10 LINK/s, to all pools
        1. Operator pool emission rate = 10 * (1000/10000) = **1 LINK/s**
        2. Community Staker pool emission rate = 10 * (9000/10000) = **9 LINK/s**
        3. reward duration = (900 + 1000) / 10 = 190 s
        4. reward duration end time = 10 + 190 = **200 s**
- T = 20. Change emission rate without adding rewards
    1. remaining rewards
        1. Operator: (200 - 20) * 10 * (1000/10000) = 180 LINK
        2. Community Staker: (200 - 20) * 10 * (9000/10000) = 1620 LINK
    2. addReward: **amount = 0 LINK, emission rate = 5 LINK/s**, to all pools
        1. Operator pool emission rate = 5 * (1000/10000) = **0.5 LINK/s**
        2. Community Staker pool emission rate = 5 * (9000/10000) = **4.5 LINK/s**
        3. Community Staker reward duration = (1620 + 0) / 4.5 = 360 s
        4. Community Staker reward duration end time = 20 + 360 = **380 s**
        5. Operator reward duration = (180 + 0) / 0.5 = 360 s
        6. Operator reward duration end time = 20 + 360 = **380 s**
- T = 30. Add rewards to only one pool, without changing the emission rate
    1. remaining rewards
        1. Operator:: (380 - 30) * 5 * (1000/10000) = 175 LINK
        2. Community Staker: (380 - 30) * 5 * (9000/10000) = 1575 LINK
    2. addReward: amount = 100 LINK, emission rate = 4.5 LINK/s, **only to the community staker pool**
        1. Community Staker pool emission rate = 4.5 LINK/s
        2. Community Staker reward duration = (1575 + 100) / 4.5 = 372 s
        3. Community Staker reward duration end time = 30 + 372 = **402 s**
- T = 360. Add rewards for the next reward emission schedule
    1. remaining rewards
        1. Community Staker = (402 - 360) * 4.5 = 189
        2. Operator = (380 - 360) * 0.5 = 10
    2. addReward: amount = 1000 LINK, emission rate = 5 LINK/s, to all pools
        1. Community Staker pool emission rate = 4.5 LINK/s
        2. Community Staker reward duration = (189 + 1000 * (9000/10000)) / 4.5 = 242 s
        3. Community Staker reward duration end time = 360 + 242 = **602 s**
        4. Operator pool emission rate = 0.5 LINK/s
        5. Operator reward duration = (10 + 1000 * (1000/10000)) / 0.5 = 220 s
        6. Operator reward duration end time = 360 + 220 = **580 s**

Setting the pool configs

Changing the `maxPoolSize`

- Changing the pool size of one pool can affect the reward rates of **all pools**.

- Especially for the pools that didn't get their `maxPoolSize` increased, when the reward proportions are rebalanced, their emission & reward rates will go down.
- To add rewards while increasing one of the pools' `maxPoolSize`, the contract manager could call the `setConfig()` function on the staking pool and then `addReward()` on the reward vault, batched in a single transaction so that the two effects are atomic and immediately followed by one another.
- Example scenario with placeholder values
    - T = 0. Start
        1. deploy
            1. Operator max pool size = 1000
            2. Community Staker max pool size = 9000
            3. Total max pool size = 10000
        2. addReward: amount = 1000 LINK, emission rate = 10 LINK/s, to all pools
            1. Operator pool emission rate = 10 * (1000/10000) = 1 LINK/s
            2. Community Staker pool emission rate = 10 * (9000/10000) = 9 LINK/s
            3. emitting period = 1000 / 10 = 100 s
            4. emitting end time = 0 + 100 = 100 s
        3. Community Staker pool is full and every staker has staked 100 LINK
            1. each staker is earning 9 LINK/s * (100/9000) = 0.1 LINK/s
    - T = 10. Increase CS pool's `maxPoolSize`, **without** adding more rewards to the cs pool
        1. Remaining rewards
            1. Operator: (100 - 10) * 10 * (1000/10000) = 90 LINK
            2. Community Staker: (100 - 10) * 10 * (9000/10000) = 810 LINK
        2. csStakingPool.setConfig: maxPoolSize = **14000**
        3. All pools' emission rates, emitting end times will be the same as before
        4. Community Staker pool is now less than full
            1. Each staker is still earning at the same rate 9 LINK/s * (100/9000) = **0.1 LINK/s**
            2. If the pool fills up again, previous stakers will earn at a lower rate 9 LINK/s * (100/14000) = **0.064 LINK/s**
    - T = 90. Add more rewards to the pool
        1. Remaining rewards
            1. Operator: (100 - 90) * 10 * (1000/10000) = 10 LINK
            2. Community Staker: (100 - 90) * 10 * (9000/10000) = 90 LINK
        2. rewardVault.addReward: amount = 1000 LINK, emission rate = 10 LINK/s, to all pools
            1. Community Staking pool emission rate = 10 * (14000/15000) = 9.33 LINK/s
            2. Community Staking pool emitting period = (90 + 1000 * (14000/15000)) / 9.33 = 101 s
            3. Community Staking pool emitting end time = 90 + 101 = 191 s
            4. Operator pool emission rate = 10 * (1000/15000) = 0.67 LINK/s

5. Operator emitting period = (10 + 1000 * (1000/15000)) / 0.67 = 114 s
6. Operator emitting end time = 90 + 114 = 204 s

Changing the `maxPrincipalPerStaker`

- Changing the max staking limit of one pool can affect the reward rates of the particular pool.
- To keep the reward rate of the pool, the `maxPoolSize` of the pool would need to change.
- Whether just changing the `maxPrincipalPerStaker` or `maxPoolSize` as well, the contract manager can call the `setConfig()` function on the staking pool once.
- Similar to the `maxPoolSize` case, to add rewards while changing the configs to adjust the reward rates, the `setConfig()` and `addReward()` can be batched transactions.
- Example scenario with placeholder values
    - T = 0. Start
        1. deploy
            1. Operator max pool size = 1000
            2. Community Staker max pool size = 9000
            3. Total max pool size = 10000
        2. addReward: amount = 1000 LINK, emission rate = 10 LINK/s, to all pools
            1. Operator pool emission rate = 10 * (1000/10000) = 1 LINK/s
            2. Community Staker pool emission rate = 10 * (9000/10000) = 9 LINK/s
            3. emitting period = 1000 / 10 = 100 s
            4. emitting end time = 0 + 100 = 100 s
        3. Community Staker pool is full and every staker has staked 100 LINK
            1. Each staker is earning 9 LINK/s * (100/9000) = 0.1 LINK/s
    - T = 10. Increase the cs pool's `maxPrincipalPerStaker`, increasing the pool's `maxPoolSize`, **without** adding more rewards to the pool
        1. remaining rewards
            1. Operator: (100 - 10) * 10 * (1000/10000) = 90 LINK
            2. Community Staker: (100 - 10) * 10 * (9000/10000) = 810 LINK
        2. communityStakingPool.setConfig: maxPoolSize = 18000, maxPrincipalPerStaker = 200
        3. All pools' emission rates, emitting end times will be the same as before
        4. Community Staker pool is now half full
            1. Each staker is still earning with the same rate 9 LINK/s * (100/9000) = **0.1 LINK/s**
            2. If the pool fills up again, previous stakers will earn 9 LINK/s * (100/18000) = **0.05 LINK/s**
            3. But before the pool fills up, if the previous stakers stake an additional 100, they will earn 9 LINK/s * (200/18000) = 0.1 LINK

Adding/Removing Node Operator Stakers

Node Operator Stakers can be added or removed by the contract manager by calling the `addOperators` and `removeOperators` functions on the `OperatorStakingPool`.

`**addOperators(address[] memory operators)**`

Addresses can be added as Node Operator Stakers by the contract manager calling the `addOperators` function. This function can only be called at any time by the contract manager.

What can an operator do?

- Earn delegated rewards by being able to withdraw from the delegated rewards pool
- Earn Operator base rewards proportional to their stake relative to the total amount Operators have staked.

`addOperators` will revert when any of the conditions below are true.

- When `numOperators * maxOperatorStakeAmount > maxOperatorPoolSize`
- An operator in the `operators` array has already been added into the pool.
- An operator in the `operators` array already has a stake. This is to prevent the contract manager from promoting a Community Staker to an Operator.
- An operator in the `operators` has been removed in the past.

`removeOperators`

Node operators can be removed from the Staking program by the contract manager by calling the `removeOperators` function. This function can only be called when the pool is active and is a one-way operation in the sense that the contract manager cannot add back an operator that has already been removed.

An operator is affected in the following way

- Rewards are frozen
- The Operator's staked LINK can still be slashed.

What can a removed operator do?:

- Claim frozen rewards
- Unstake their staked LINK after an unbonding period
- Stake as a Community Staker but with their multiplier reset

What can a removed operator NOT do?

- Cannot migrate to a new `OperatorStakingPool`

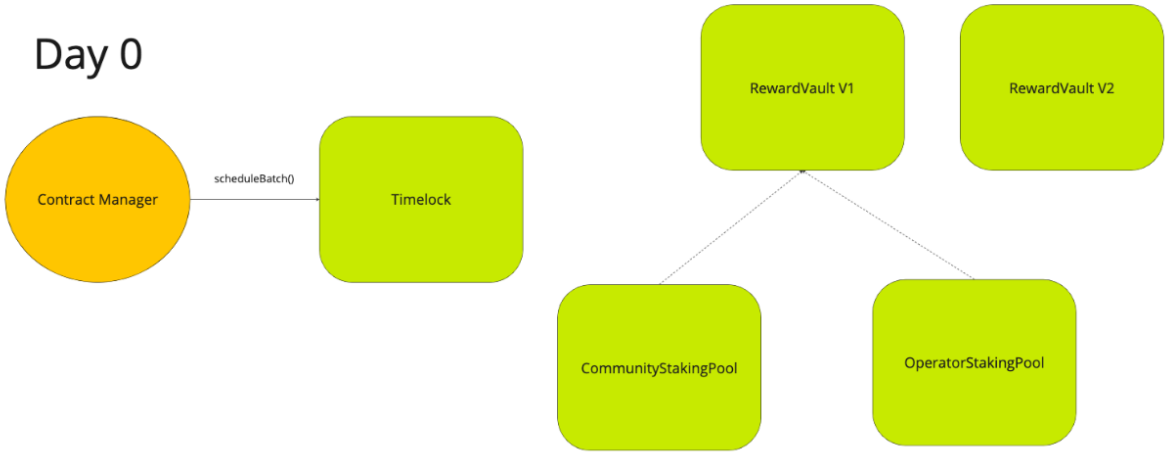
The `OperatorStakingPool` will freeze future rewards by setting the Operator's `principal` to 0 so that they cannot claim any future rewards. In order to preserve the Operator's staked LINK and earned rewards, the contract will write the values to the StakerReward struct's `finalizedBaseReward` and `finalizedDelegatedReward` and `Operator` struct's `removedPrincipal` fields. A removed Operator can call `claimReward` to withdraw their `finalizedBaseReward`, `finalizedDelegatedReward`, or first call `unbond` and then `unstakeRemovedPrincipal` to withdraw their `removedPrincipal`.

Upgrading a `RewardVault`

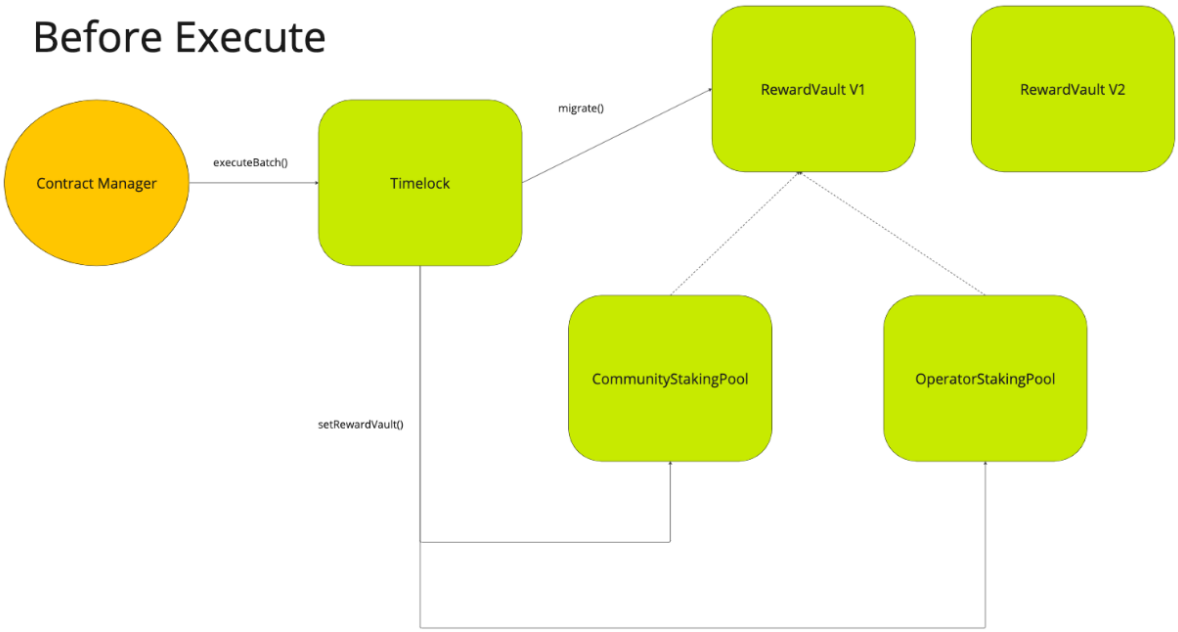
Motivation

Staking v0.2 allows for the introduction of new reward and/or multiplier calculations without requiring stakers to migrate their stake. This can be achieved by implementing and introducing an updated version of the `RewardVault`.

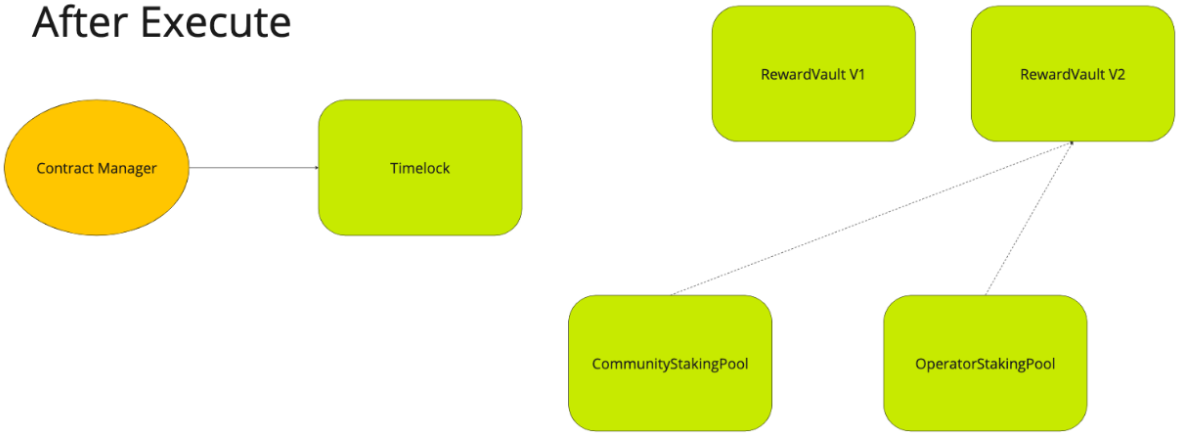
Day 0



Before Execute



After Execute



Upgrade Procedure

A new version of the `RewardVault` can be introduced by the contract manager in the following way.

1. **Day 0**
    - Deploy and Configure `RewardVaultV2`
        - A new reward vault is deployed denoted by `RewardVaultV2`.
        - `RewardVaultV2` is configured to have `RewardVaultV1` as it's migration source by calling `setMigrationSource`
        - Ownership of `RewardVaultV2` is transferred to the contract manager
    - Scheduling the upgrade
        - The contract manager calls `scheduleBatch` on the `Timelock` contract to call the functions below.
            - `setRewardVault(addressRewardVaultV2)` on the `CommunityStakingPool`
            - `setRewardVault(addressRewardVaultV2)` on the `OperatorStakingPool`
            - `setMigrationTarget(addressRewardVaultV2)` on `RewardVaultV1`
            - `migrate()` on `RewardVaultV1`
2. **Final Day**
    - The contract manager calls `executeBatch` on the `Timelock` contract to batch execute the functions above.
    - This will
        - Migrate non-emitted rewards from `RewardVaultV1` to `RewardVaultV2`.
        - Close the `RewardVaultV1`
        - Point the reward vault address on the `CommunityStakingPool` and `OperatorStakingPool` contracts to `RewardVaultV2`.

Preserving Unclaimed Rewards In The Old Vault

In order to correctly calculate the user's rewards in an older version of a `RewardVault`, the old vault needs to lock in the user's staked LINK amount and the average staked at time when `migrate` or `close` is called. Capturing a snapshot of the staker's staked LINK and the total pool stake is required in case a user stakes or unstakes after the `RewardVault` has been migrated/closed as this would impact the user's share of already emitted rewards in the older vault.

Formula to calculate rewards

$$rewards = \frac{stakerStakedLINKAmount * emittedRewardAmount}{totalStakedLINKAmount}$$

The staking pool tracks the amount that the staker has staked as well as their staked at time history in the `Staker` struct's `principalHistory` and `stakedAtTimeHistory` fields. These fields are updated whenever the staking pool calls the `StakingPoolBase._updateStakerHistory` function. These history fields are implemented using the OpenZeppelin's Checkpoints library, so that we can look up the historical values at a given checkpoint Id. Also, when the vault is migrated/closed, the `RewardVault._checkpointStakingPools` function is called to store the staking pools' total staked amount and checkpoint Ids at the time of migration/closing. These values are tracked by the `VestingCheckpointData` struct.

Once the vault is closed, the `RewardVault.getReward` and `RewardVault.getMultiplier` functions will use these fields instead of reading the staker's latest staked LINK or staked at time from the staking pools. In addition, the reward vault will use the `operatorPoolTotalPrincipal` and `communityPoolTotalPrincipal` fields in the `VestingCheckpointData` struct to know the total amount staked at the time the reward vault is migrated/closed.

Example

- Assuming an example emission rate of `10 LINK/day`
- Day 0
  - Staker 1 stakes 100 LINK in a 1000 LINK pool (Owns 10%)
- Day 100
  - 1000 LINK has emitted with 5000 LINK more to emit
  - Contract manager calls `migrate` to migrate. This locks 1000 LINK of claimable rewards in the older vault. The remaining 4000 LINK is moved to the newer vault.
- Day 200
  - User stakes 400 LINK and now owns 29% of the pool.
  - 1000 more LINK has emitted over the last 100 days in the new `RewardVault`
  - At this point the user's reward should be:

$$stakerTotalRewards = stakerRewardsVaultOne + stakerRewardsVaultTwo$$

$$stakerRewardsVaultOne = \frac{stakerStakedLINKAmountAtMigrate * totalRewardsVaultOne}{totalStakedLINKAmountAtMigrate}$$

$$stakerRewardsVaultTwo = \frac{stakerCurrentStakedLINKAmount * totalRewardsVaultTwo}{totalStakedLINKAmount}$$

$$stakerRewardsVaultOne = \frac{100 * 1000}{1000} = 100$$

$$stakerRewardsVaultOne = \frac{400 * 1000}{1400} = 285$$

$$stakerTotalRewards = 100 + 285 = 385$$

migrate

The `migrate` function would be called by the contract manager to migrate rewards from the previous `RewardVault` to the latest `RewardVault`. At a high level migrate will

1. Update the rewards earned per token staked in the vault.

```
Unset
updateReward(address(0), 0, true);
```

2. Calculate non-emitted rewards

```
Unset
unvestedRewards = (rewardDurationEndsAt - block.timestamp) *
emissionRate;
```

3. End reward emission

```
Unset
rewardDurationEndsAt = block.timestamp;
emissionRate = 0;
```

4. Close the current vault.

```
Unset
delete s_vaultConfig.isOpen;
```

5. Transfer non-emitted rewards to the target reward vault.

```
Unset
i_LINK.transferAndCall(newVaultAddr, unvestedRewards, bytes(0))
```

The new `RewardVault` will handle the received LINK rewards through the `onTokenTransfer` function.

Claiming Rewards From Older Vaults

Stakers can continue to claim rewards from older `RewardVault`s by calling the `claimReward` function directly on that vault.

Upgrading a `PriceFeedAlertsController`

Motivation

To introduce slashing community staked LINK or slashing operators' rewards in the future, the `PriceFeedAlertsController` will need to be updated, as well as the staking pools (requiring a migration).

Upgrade Procedure

A new version of the `PriceFeedAlertsController` can be introduced by the contract manager in the following way.

1. A new `PriceFeedAlertsController` is implemented, deployed, and configured.

2. Ownership of the `PriceFeedAlertsController` is transferred to the timelock contract by calling `beginDefaultAdminTransfer`.
3. The contract manager schedules a `acceptDefaultAdminTransfer` call to the timelock to make the timelock contract the default contract manager of the new `PriceFeedAlertsController`.
4. The contract manager schedules a `setMigrationTarget` call to the timelock to set the new `PriceFeedAlertsController` as the migration target on the old `PriceFeedAlertsController`.
5. The contract manager schedules `sendMigrationData` calls to the timelock to send migration data (feeds' last alerted round IDs) from the old to new `PriceFeedAlertsController`.
6. The contract manager schedules a `migrate` call to the timelock on the old `PriceFeedAlertsController` to decommission it.
7. The contract manager schedules a `grantRole` call to the timelock on the `OperatorStakingPool` to grant the slasher role to the new `PriceFeedAlertsController`.
8. After the timelock period is over, the contract manager batch-executes the registered operations (`acceptDefaultAdminTransfer`, `setMigrationTarget`, `sendMigrationData`, `migrate`, and `grantRole`) on the timelock, which will call the actual functions on the `PriceFeedAlertsController` and `OperatorStakingPool`.

Preventing duplicate alerts on the same round Id

In order to prevent stakers from raising duplicate alerts on the same round Id on both old and new `PriceFeedAlertsController`, we migrate the latest round Ids that had an alert raised on from the old `PriceFeedAlertsController` to the new `PriceFeedAlertsController`. When the new contract receives the migration data (feed address and its last alerted round Id), it will set its last alerted round Id of the feed to the data so that alerters can only raise alerts on round Ids greater than the last alerted round Id. Also, the migrated feed's config is removed from the old `PriceFeedAlertsController` so that no alerts can be raised anymore.

Contract Definitions

StakingPoolBase

Common functionality between both `OperatorStakingPool` and `CommunityStakingPool`.

Structs

- **`Staker`**

- Used to track a staker's current state in the pool.

```
Unset
struct Staker {
      Checkpoints.Trace224 principalHistory;
      Checkpoints.Trage224 stakedAtTimeHistory;
}
```

Variable Name	Description
`principalHistory`	A series of checkpoint data of the staker's historical staked LINK. Current staked LINK can be accessed by calling `s_stakers[staker].principalHistory.latest()`.
`stakedAtTimeHistory`	A series of checkpoint data of the staker's historical staked at times. Current staked at time can be accessed by calling `s_stakers[staker].stakedAtTimeHistory.latest()`.

- **`PoolState`**
  - Used to track the global state of the pool.

```
Unset
struct PoolState {

      uint256 totalPrincipal;

      bool isOpen;

      uint256 closedAt;

}
```

Variable Name	Description
`totalPrincipal`	Tracks the total amount of LINK staked by all stakers.

`isOpen`	Flag to track whether or not the pool is open for staking
`closedAt`	The time that contract has been closed

- **`PoolConfigs`**

```
Unset
struct PoolConfigs {
      uint256 maxPoolSize;
      uin256 maxPrincipalPerStaker;
      uint256 unbondingPeriod;
      uint256 claimPeriod;
}
```

Variable Name	Description
`maxPoolSize`	Tracks the maximum amount that can be staked in the pool
`maxPrincipalPerStaker`	Tracks the maximum amount a staker can stake
`unbondingPeriod`	The amount of time a staker must wait before they can unstake their staked LINK. This can be increased or decreased within the maximum and minimum bounds.
`claimPeriod`	The amount of time after a staker has to unstake their LINK following the claim period. This can be increased or decreased within the maximum and minimum bounds.

- **`Pool`**

```
Unset
struct Pool {
      PoolState state;
```

```
        PoolConfigs configs;
    }
```

Storage Variables

Variable Name	Type	Description
`s_rewardVault`	`IRewardVault`	Stores the address of the latest `RewardVault`.
`s_unbondingEndsAt`	`mapping(address => uint256)`	Mapping of a staker to the time their unbonding period ends
`s_claimPeriodEndsAt`	`mapping(address => uint256)`	Mapping of a staker to the time their claim period ends
`s_migrationProxy`	`address`	Migration proxy address
`s_rewardVault`	`s_rewardVault`	The latest reward vault address
`i_minPrincipalPerStaker`	`i_minPrincipalPerStaker`	The min amount of LINK that a staker can stake
`i_minClaimPeriod`	`uint256`	The minimum claim period
`i_maxClaimPeriod`	`uint256`	The maximum claim period
`i_claimPeriod`	`uint256`	The length of the claim period
`i_maxUnbondingPeriod`	`uint256`	The maximum unbonding period
`MIN_UNBONDING_PERIOD`	`uint256 constant`	The minimum unbonding period
`s_checkpointId`	`s_checkpointId`	The current checkpoint ID. Incremented whenever any staker's state changes.
`i_LINK`	`LinkTokenInterface`	Stores the address of the LINK token
`s_pool`	`Pool`	Stores the global state of the pool.
`s_stakers`	`s_stakers`	Mapping of a staker's address to their staker state

Write Functions From ERC677ReceiverInterface

Function Name	Parameters	Returns	Description	Reverts
`setPoolConfig`	`maxPoolSize: uint256` `maxPrincipalPerStaker: uint256`	N/A	Sets max values for pool size and staked LINK per staker.	- Reverts when not called by owner - Reverts when max pool size is zero - Reverts when max pool size is less than the previously set max pool size - Reverts when max staked LINK per staker is 0 - Reverts when max staked LINK per staker is greater than max pool size - Reverts when max staked LINK per staker is less than or equal to previously set max staked LINK per staker
`open`	N/A	N/A	N/A	- Reverts when not called by owner
`close`	N/A	N/A	N/A	- Reverts when not called by owner
`setMigrationProxy`	`migrationProxy: address`	N/A	Sets the migration proxy for migrations from staking v0.1	- Reverts when not called by owner - Reverts when the migration proxy address is zero

Write Functions From IStakingOwner

Function Name	Parameters	Returns	Description	Reverts
`unstake`	`amount: uint` `shouldClaimReward: bool`	N/A	Unstakes `amount` LINK tokens from the staker's staked LINK. Also claims the rewards earned from the currently configured rewards vault if `shouldClaimReward` is true.	- Reverts when unbonding period is not over yet or if the unbonding period is not started. - Reverts if the claim period is over

Write Functions From IMigratable

Function Name	Parameters	Returns	Description	Reverts
`migrate`	`data: bytes`	N/A	Migrates to next version	- Reverts when no migration target is set - Reverts when v0.2 is not closed
`setMigrationTarget`	`newMigrationTarget: address`	N/A	Sets a new migration target	- Reverts when target is address 0 - Reverts when target is the same as previously set target

				- Reverts when target is not a contract address

Write Functions From IPausable

Function Name	Parameters	Returns	Description	Reverts
`emergencyPause`	N/A	Pauses the pool during emergency.	N/A	- Reverts when no migration target is set - Reverts when v0.2 is not closed
`emergencyUnpause`	N/A	Unpauses the pool.	- If the contract is not paused	- Reverts when target is address 0 - Reverts when target is the same as previously set target - Reverts when target is not a contract address

Additional Write Functions

Function Name	Parameters	Returns	Description	Reverts
`unbond`	N/A	N/A	Starts the unbonding period to unstake the staked LINK staked	- Reverts when called whilst the staker is still unbonding

Operational Functions

These functions can be called to change the pool's parameters. Only the contract manager may call these functions.

Function Name	Parameters	Description	Revert Conditions
`setConfig`	`maxPoolSize: uint256` `maxPrincipal: uint256`	Sets the new pool configs	- The amount staked in the pool is greater than the new `maxPoolSize`
`setRewardVault`	`IRewardVault: address`	Sets the new reward vault	- If the caller is not the contract manager

`setUnbondingPeriod`	`newUnbondingPeriod: uint256`	Sets the new unbonding period	- If the caller is not the contract manager - If the new unbonding period is below the minimum or above the maximum unbonding period
`setClaimPeriod`	`newClaimPeriod: uint256`	Sets the new claim period	- If the caller is not the contract manager - If the new claim period is below or above the maximum claim period

View Functions

Function Name	Parameters	Returns	Description
`getTotalPrincipal`	N/A	`uint256`	Returns the total amount staked in the pool.
`getUnbondingEndsAt`	`staker: address`	`uint256`	Returns the time a staker's unbonding period ends
`getClaimPeriodEndsAt`	`staker: address`	`uint256`	Returns when the staker's current claim period ends
`isPaused`	N/A	`bool`	Returns a boolean that is true if the pool is paused and false otherwise.
`isActive`	N/A	`bool`	Returns a boolean that is true if the `s_pool.isOpen` flag is true **AND** there are remaining rewards to emit in the pool.
`getStakerLimits`	N/A	`uint256`, `uint256`	Returns the minimum and maximum amounts a staker can stake in the pool
`getStakerPrincipal`	`staker: address`	`uint256`	Returns the staker's current staked LINK
`getStakerPrincipalAt`	`staker: address` `checkpointId: uint256`	`uint256`	Returns the staker's staked LINK at checkpoint
`getStakerAverageStakedAtTime`	`staker: address`	`uint256`	Returns the staker's current average staked at time
`getStakerAverageStakedAtTimeAt`	`staker: address` `checkpointId: uint256`	`uint256`	Returns the staker's average staked at time at checkpoint
`getRewardVault`	N/A	`IRewardVault`	Returns the reward vault address
`getChainlinkToken`	N/A	`address`	Returns the chainlink token address

`getMigrationProxy`	N/A	`address`	Returns the migration proxy address
`getMaxPoolSize`	N/A	`uint256`	Returns the max pool size
`getUnbondingParams`	N/A	`uint256`, `uint256`	Returns the unbonding period and the claim period
`getCurrentCheckpointId`	N/A	`staker: address`	Returns the current checkpoint Id
`getClaimPeriodLimits`	N/A	`uint256, uint256`	Returns the minimum and maximum claim periods
`getUnbondingPeriodLimits`	N/A	`uint256, uint256`	Returns the minimum and maximum unbonding period limits

OperatorStakingPool

In addition to implementing the functions in the `StakingPool`, the `OperatorStakingPool` must also implement the following functions.

Structs

```
Unset
struct Operator {
     uint256 removedPrincipal;
     bool isOperator;
     bool isRemoved;
}
```

Storage Variables

Variable Name	Type	Description
`s_operators`	`mapping(address => Operator)`	Mapping of addresses to the Operator information.

`s_alerterRewardFunds`	`uint256`	Tracks the balance of the rewards available to be paid out to an alerter.
`s_slasherConfigs`	`mapping(address ⇒ ISlashable.Config)`	Mapping of addresses of slashers to their configs
`s_slasherState`	`mapping(address ⇒ ISlashable.SlasherState)`	Mapping of addresses of slashers to their current slasher state
`s_numOperators`	`uint256`	Current number of operators in the pool
`i_minInitialOperatorCount`	`uint256`	The minimum initial amount of operators needed in the pool
`s_alerterRewardFunds`	`uint256`	A balance of funds that can source rewards for alerters
`SLASHER_ROLE`	`bytes32`	The keccak256 hash of `SLASHER_ROLE`

Write Functions From ISlashable

Unset

```
struct SlasherConfig {
      uint256 refillRate;
      uint256 slashCapacity;
}

struct SlasherState {
      uint256 lastSlashTimestamp;
      uint256 slashCapacityUsed;
}

function setSlasherConfig(
      address slasher,
      SlasherConfig calldata config
) external

function slash(
      address[] calldata operators,
```

```
        address alerter,
        uint256 principalAmount,
        uint256 alerterRewardAmount
    ) external
```

Function Name	Parameters	Returns	Description	Reverts When
`addSlasher`	`slasher: address` `config: SlasherConfig`	N/A	Configures a new slasher address on the slashable pool	- Called by the non admin - Slasher capacity is 0
`setSlasherConfig`	`slasher: address` `config: SlasherConfig`	N/A	Sets a new configuration for a slasher	- Called by the non admin - `slasher` does not have the `SLASHER_ROLE` - Slasher capacity is 0

`slash`	`operators: address[]`
	`alerter: address`
	`principalAmount: uint256`
	`alerterRewardAmount: uint256`
		N/A	Called by an address with the `SLASHER_ROLE` to slash the targeted stakers	- Reverts if the caller does not have the `SLASHER_ROLE`

`withdrawAlerterReward`	`amount: uint256`	N/A	Called by the contract manager to withdraw rewards from the alerter reward funds.	The amount passed in is greater than the alerter reward funds.

View Functions From ISlashable

Function Name	Parameters	Returns	Description
`getSlasherConfig`	`slasher: address`	`SlasherConfig`	Returns the configuration for the `slasher`
`getSlashCapacity`	`slasher: address`	`uint256`	Returns the current capacity the `slasher` can slash

Write Functions

Function Name	Parameters	Returns	Description	Reverts When
`addOperators`	`operators: address[]`	N/A	Adds new operators to the pool. This has the side effect of allowing these new operators to stake into the pool.	- The product of the number of operators and the maximum staked LINK is greater than the maximum pool size.
`removeOperators`	`operators: address[]`	N/A	Removes operators from the pool. This has the side effect of no longer allowing the removed operators from staking into the pool.	- Any of the operators is not added
`unstakeRemovedPrincipal`	N/A	N/A	Called by removed operators to withdraw their removed staked LINK.	- Reverts when the caller doesn't have any removed staked LINK
`depositAlerterReward`	`amount: uint256`	N/A	Called by the contract manager to deposit rewards to the alerter reward funds.	Pool is paused or closed
`withdrawAlerterReward`	`amount: uint256`	N/A	Called by the contract manager to withdraw rewards from the alerter reward funds.	The amount passed in is greater than the alerter reward funds.

View Functions

Function Name	Parameters	Returns	Description
`isOperator`	`staker: address`	`bool`	Returns true if the `staker` is an operator.
`getAlerterRewardFunds`	N/A	`uint256`	Returns the balance of the pool's alerter reward funds
`getSlasherConfig`	`slasher: address`	`SlasherConfig`	Returns the slasher's config
`getSlashCapacity`	`slasher: address`	`uint256`	Returns the slasher's current slash capacity
`isRemoved`	`slasher: address`	`bool`	Checks if a staker has been removed
`getRemovedPrincipal`	`staker: address`	`uint256`	Returns a removed operator's staked LINK
`getNumOperators`	N/A	`uint256`	Returns the number of operators
`typeAndVersion`	N/A	`string`	Returns the type and version of the contract

CommunityStakingPool

In addition to implementing the functions in the `StakingPool`, the `CommunityStakingPool` must also implement the following functions.

Storage Variables

Variable Name	Type	Description
`s_operatorStakingPool`	`mapping(address => Operator)`	Mapping of addresses to the Operator information.
`s_merkleRoot`	`bytes32`	The merkle root of the merkle tree generated from the list of staker addresses with early access.

Write Functions From `IMerkleAccessController`

Function Name	Parameters	Returns	Description
`setMerkleRoot`	`merkleRoot: bytes`	N/A	Sets the new merkle root of addresses that are able to stake in the pool. This can be set to an empty bytes to make the pool public

Write Functions From TypeAndVersionInterface

Function Name	Parameters	Returns	Description	Reverts
`typeAndVersion`	N/A	`string`	Returns the type and version of the contract	N/A

View Functions

Function Name	Parameters	Returns	Description

`hasAccess`	`staker: address` `proof: bytes32[]`	`bool`	Returns true if a staker can stake and false otherwise
`getMerkleRoot`	N/A	`bytes32`	Returns the merkle root

RewardVault

Structs

- **`StakerReward`**
  - Used to track a staker's rewards and reward per token values.

```
Unset
struct StakerReward {
      uint256 storedBaseReward;
      uint256 finalizedBaseReward;
      uint256 finalizedDelegatedReward;
      uint256 baseRewardPerToken;
      uint256 operatorDelegatedRewardPerToken;
      uint256 claimedBaseRewardsInPeriod;
      uint256 totalEarnedBaseRewardsInPeriod;
}
```

Variable Name	Description
`storedBaseReward`	The staker's accrued base rewards after the last time they staked/unstaked. This is denominated in juels.
`finalizedBaseReward`	The amount of base rewards that a staker has earned up to the last time they staked/unstaked. This is denominated in juels.
`finalizedDelegatedReward`	The amount of delegated rewards that a staker has earned up to the last time they staked/unstaked. This is denominated in juels and is always 0 for community stakers.
`baseRewardPerToken`	The last updated per-token base reward of the staker.
`operatorDelegatedRewardPerToken`	The last updated per-token delegated reward of the staker. Only applicable to the

	operators.
`claimedBaseRewardsInPeriod`	The amount of rewards a staker has claimed since the last time they staked/unstaked. This is reset to 0 whenever they stake/unstake.
`totalEarnedBaseRewardsInPeriod`	The total amount of base rewards a staker has earned since the last time they staked/unstaked. This is slightly different from `storedBaseReward` as `storedBaseReward` gets reset whenever a staker calls `claimReward`. This value is the sum of `storedBaseReward` and `claimedBaseRewardsInPeriod``

- **`RewardBucket`**
  - Used to track a reward bucket's configs and statuses.

```
Unset

struct RewardBucket {
      uint256 emissionRate;
       uint256 vestedRewardPerToken;
      uint256 rewardDurationEndsAt;
}
```

Variable Name	Description
`emissionRate`	The reward emission rate of the reward bucket in Juels/second.
`vestedRewardPerToken`	The last updated emitted reward per token of the reward bucket.
`rewardDurationEndsAt`	The time at which the current reward event will end.

- `RewardBuckets`
  - Used to track the 3 reward buckets (Operator base rewards, delegated rewards, and Community Staker base rewards).

```
Unset
```

```
struct RewardBuckets {
            RewardBucket operatorBase;
            RewardBucket communityBase;
            RewardBucket operatorDelegated;
}
```

- `VaultConfig`
    - Used to track the reward vault's configs

```
Unset

struct VaultConfig {
      uint256 delegationRateDenominator;
      uint256 multiplierDuration;
      bool isOpen;
}
```

Variable Name	Type	Description
`multiplierDuration`	`uint256`	The time it takes for a multiplier to reach its max value in seconds.
`delegationRateDenominator`	`uint256`	Indicates the % of Community Staker rewards that should be delegated to the Node Operators, inversed. (example: 100 * 5% = 100 / 20 = 5)
`isOpen`	`bool`	N/A

- `VestingCheckpointData`
    - Used to track the total staked LINK amounts and checkpoint Ids of the connected staking pools at the time of migration or closing the vault.

```
Unset

struct VestingCheckpointData {
      uint256 operatorPoolTotalPrincipal;
      uint256 communityPoolTotalPrincipal;
      uint256 operatorPoolCheckpointId;
      uint256 communityPoolCheckpointId;
}
```

Variable Name	Type	Description
`operatorPoolTotalPrincipal`	`uint256`	The total staked LINK of the operator staking pool at the time the reward vault was migrated or closed
`communityPoolTotalPrincipal`	`uint256`	The total staked LINK of the community staking pool at the time the reward vault was migrated or closed
`operatorPoolCheckpointId`	`uint256`	The checkpoint ID of the operator staking pool at the time the reward vault was migrated or closed
`communityPoolCheckpointId`	`uint256`	The checkpoint ID of the community staking pool at the time the reward vault was migrated or closed

Storage Variables

Variable Name	Type	Description
`i_operatorStakingPool`	`IStakingPool`	The `OperatorStakingPool`.
`i_communityStakingPool`	`IStakingPool`	The `CommunityStakingPool`.
`s_rewardBuckets`	`RewardBuckets`	The Operator base, delegated, and Community Staker base reward buckets.
`s_vaultConfig`	`VaultConfig`	The reward vault's configs.
`i_LINK`	`LinkTokenInterface`	

`s_finalVestingCheckpointData`	`VestingCheckpointData`	The checkpoint information at the time the reward vault was closed or migrated.
`s_rewardPerTokenUpdatedAt`	`uint256`	The time the reward per token was last updated
`s_migrationSource`	`address`	The address of the vault that will be migrated to this vault.
`s_rewards`	`mapping(address ⇒ StakerReward)`	Stores reward information for each staker

Write Functions

Function Name	Parameters	Returns	Description	Reverts

`addReward`	`amount: uint256`
	`emissionRate: uint256`
	`stakingPool: address`
		N/A	Adds more rewards into the reward vault. Calculates the reward duration from the amount and emission rate.	- When called by an address that does not have the `REWARDER_ROLE` - When the `stakingPool` is not zero nor one of the registered staking pools.

`setMigrationTarget`	`targetRewardVault: address`	N/A	Sets a migration `RewardVault` target	- The `targetRewardVault` is invalid (is empty, equals to the current address, equals to the currently proposed address)
`migrate`	`data: bytes`	N/A	Migrates non-emitted rewards	- When called by an address that is not the vault's owner. - When timelock period following the time the contract manager called `proposeMigrationTarget` has not yet elapsed
`updateReward`	`staker: address` `updatePools: bool`	N/A	Updates the emitted rewards earned per token. This is typically called by a staking pool whenever a staker stakes/unstakes or is slashed and when the reward vault is upgraded to a new one.	- Reverts when not called by either the `OperatorStakingPool` or the `CommunityStakingPool`.
`withdrawUnvestedReward`	`tokens: address[]`	N/A	Withdraws any non-emitted rewards	- When the vault is still open.

`claimReward`	`staker: address`	`uint256`	Called by a staker or staking pool to claim a staker's rewards and returns the amount claimed.	- If the staker's staked LINK is 0
`close`			Called by the contract manager to close the reward vault. Disables adding more rewards and staking for pools pointing to a closed reward vault.	N/A
`setDelegationRateDenominator`	`newDelegationRateDenominator: uint256`	N/A	Updates the delegation rate denominator.	- When the new rate is the same as the current - When the new rate causes insufficient rewards
`setMultiplierDuration`	`newMultiplierDuration: uint256`	N/A	Sets the new multiplier ramp-up time. Existing finalized rewards will not be affected.	N/A
`setMigrationSource`	`newMigrationSource: address`	N/A	Sets the migration source for the vault	N/A
`onTokenTransfer`	`sender: address amount: uint256 data: bytes`	N/A	Handler function for when a staker migrates	- When the non-emitted reward amounts do not add up
`finalizeReward`	`staker: address`	`uint256`	This applies any final logic such as the multipliers to the staker's newly accrued and stored rewards and store the value.	N/A

View Functions

Unset

```
struct RewardBucket {
      uint256 emissionRate;
      uint256 rewardDurationEndsAt;
      uint256 vestedRewardPerToken;
}

struct RewardBuckets {
      RewardBucket operatorBase;
      RewardBucket communityBase;
      RewardBucket operatorDelegated;
}
```

Function Name	Parameters	Returns	Description
`getReward`	`staker: address`	Should return rewards that the staker would get if they withdraw now. Based on their current multiplier.	Returns the total rewards a user has accumulated.
`getMultiplier`	`staker: address`	Returns a staker's current multiplier	N/A
`isPaused`	N/A	Returns a boolean that is true if the reward vault is paused and false otherwise.	Returns whether or not the pool is currently paused.
`isOpen`	N/A	Returns a boolean that is true if the reward vault is open.	Returns whether or not the pool is active.
`getRewardBuckets`	N/A	Returns three RewardBuckets structs that indicate the emission rates, the timestamps when the reward duration ends, and the last updated event reward per token of the reward buckets.	Returns the reward information of all reward buckets.
`getRewardPerTokenUpdatedAt`	N/A	Returns the timestamp of the last reward per token update	N/A
`supportsInterface`	`interfaceId: bytes4`	This function allows the calling contract to check if the contract deployed at this address is a valid LINKTokenReceiver. A contract is a valid LINKTokenReceiver if it implements the onTokenTransfer function.	N/A
`getMultiplierDuration`	N/A	Returns the multiplier ramp-up time	N/A
`getMultiplier`	`staker: address`	Returns the ramp-up multiplier of the staker	N/A
`getStoredReward`	`staker: address`	Returns the stored reward info of the staker	N/A
`getVestingCheckpointData`	N/A	Returns the migration checkpoint data	N/A
`calculateLatestStakerReward`	`staker: address`	Calculates and returns the latest reward info of the staker	N/A

PriceFeedAlertsController

Structs

- **`FeedConfig`**
    - Used to track a feed's config values

```
Unset
struct FeedConfig {
      uint256 priorityPeriodThreshold;
      uint256 regularPeriodThreshold;
      uint256 slashableAmount;
      uint256 alerterRewardAmount;
      address[] slashableOperators;
}
```

Variable Name	Description
`priorityPeriodThreshold`	The number of seconds until the feed is considered stale and the priority period begins.
`regularPeriodThreshold`	The number of seconds until the priority period ends and the regular period begins.
`slashableAmount`	The amount each node operator gets slashed if a feed goes down. This is denominated in juels.
`slashableOperators`	The operators that get slashed when a feed goes down.
`alerterRewardAmount`	The amount of juels an alerter will receive for successfull raising an alert for a feed

Storage Variables

Variable Name	Type	Description
`i_operatorStakingPool`	`OperatorStakingPool`	The address of the `OperatorStakingPool` contract.
`i_communityStakingPool`	`CommunityStakingPool`	The address of the

		`CommunityStakingPool` contract.
`s_feedConfigs`	`mapping(address => FeedConfig)`	The round ID of the last feed round an alert was raised
`s_lastAlertedRoundIds`	`mapping(address => uint256)`	The round ID of the last feed round an alert was raised

Write Functions

```
Unset
struct SetFeedConfigParams {
      address feed;
      uint256 priorityPeriodThreshold;
      uint256 regularPeriodThreshold;
      uint256 slashableAmount;
      uint256 alerterRewardAmount;
}
```

Function Name	Parameters	Returns	Description	Reverts
`setFeedConfigs`	`configs: SetFeedConfigParams[]`	N/A	Sets the FeedConfig for a Data Feed, given the feed's addresses	- Reverts when the feed is invalid or inactive - Reverts when the config values are invalid
`removeFeedConfig`	`feed: address`	N/A	Removes the feed config for a Data Feed, given the feed's address	- Reverts when the feed is not set
`raiseAlert`	`data: bytes`	N/A	Creates an alert for a chainlink service that has met the slashing conditions.	- Reverts when the slashing condition hasn't been met. - Reverts when the alerter is not a staker - Reverts when the alerter is a Community Staker and it's still in the priority period
`setSlashableOperators`	`data: bytes slashableOperators: address[]`	N/A	Sets the operators that will be slashable if a Data Feed goes down.	- Revert when the operator is not an operator in the `OperatorStakingPool`.
`migrate`	`data: bytes`	N/A	Migrate to the next Data Feed alerts controller	
`sendMigrationData`	`feeds: address[]`	N/A	Function for migrating the data to the	- When a Data Feed does not exist

			migration target.	

View Functions

Function Name	Parameters	Returns	Description
`getFeedConfig`	`base: address`	`FeedConfig`	Returns the FeedConfig of a Data Feed, given the feed's base and quote addresses.
`canAlert`	`alerter: address data: bytes`	`bool`	Returns whether the alerter may raise an alert to claim rewards
`getSlashableOperators`	`data: bytes`	`address[]`	Returns the slashable operators.
`getStakingPools`	N/A	`address[]`	Returns the staking pools
`typeAndVersion`	N/A	`string`	Returns the type and version of the contract

In addition to the functions above, `PriceFeedAlertsController` inherits functions from `Migratable` and `PausableWithAccessControl`.

MigrationProxy

Storage Variables

Variable Name	Type	Description
`i_operatorPool`	`OperatorStakingPool`	The address of the `OperatorStakingPool` contract.
`i_communityPool`	`CommunityStakingPool`	The address of the `CommunityStakingPool` contract.
`i_LINK`	`LinkTokenInterface`	The address of the LINK token contract.
`i_v01StakingAddress`	`address`	The address of the Staking v0.1 contract.

Write Functions

Function Name	Parameters	Returns	Description	Reverts

`onTokenTransfer`	`source: address`
	`amount: uint256`
	`data: bytes`
		N/A	The migration from v0.1 will get routed to a `OperatorStakingPool` or a `CommunityStakingPool`, depending on the staker's type.	- When called by an address other than `i_v01StakingAddress`

View Functions

Function Name	Parameters	Returns	Description

`getConfig`	N/A	`address,`
		`address,`
		`address,`
		`address`
			Returns the Link token, Staking v0.1, Operator staking pool, and community staking pool.

`supportsInterface`	`interfaceId` `: bytes4`	`bool`	This function allows the calling contract to check if the contract deployed at this address is a valid LINKTokenReceiver. A contract is a valid LINKTokenReceiver if it implements the onTokenTransfer function.
`typeAndVersion`		`string`	Returns the type and version