Staking Smart Contracts

[WD] Staking Smart Contracts **1**
- VoteWeighting Contract 2
  - Overview 2
  - Functionality summary. 2
  - How the contract handles voting weights. 3
- Dispenser Contract 4
  - Overview 4
  - Functionality summary. 4
  - How the contract handles staking calculation and distribution 6
- TargetDispenserL2 Contract 7
  - Overview 7
  - Functionality summary. 7
- Modular approach for L1-L1 and L1-L2 message and token transfers 10
- L2-L1 message 12
- TokenomicsContract 12
  - Overview 12
  - Functionality summary. 13
- StakingFactory contract, Staking contract implementation stakingContractProxy, StakingActivityCheck, and StakingVerifier contracts. 13
  - Overview 13
  - StakingFactory 13
  - Staking contract implementation 15
    - ActivityCheck 16
      - How does the staking contract interact with the activity checker? 16
    - Configurations to select to deploy a staking instance. 16
    - StakingContractProxy 17
  - StakingActivityCheck contract 18
  - StakingVerifier contract 19
    - Verification on staking contract enabled by StakingVerifier 19
- Staking workflow: launch, vote, claim 22

VoteWeighting Contract

Overview

The VotingWeight contract enables Olas DAO members (via veOLAS) to vote on staking programs, assigning weights according to their preferences.

It adopts a model similar to the Curve Gauge Controller, maintains a list of gauges and their associated weights. Modifications from the original Curve Gauge Controller include granting anyone the ability to add staking contracts by removing ownership control on this functionality, and eliminating additional categorization by contract type. To ensure compatibility with veOLAS, changes in vote weights per staking contract are scheduled for the next week. Consequently, when a user applies a new weight, it gets applied at the start of the next week.

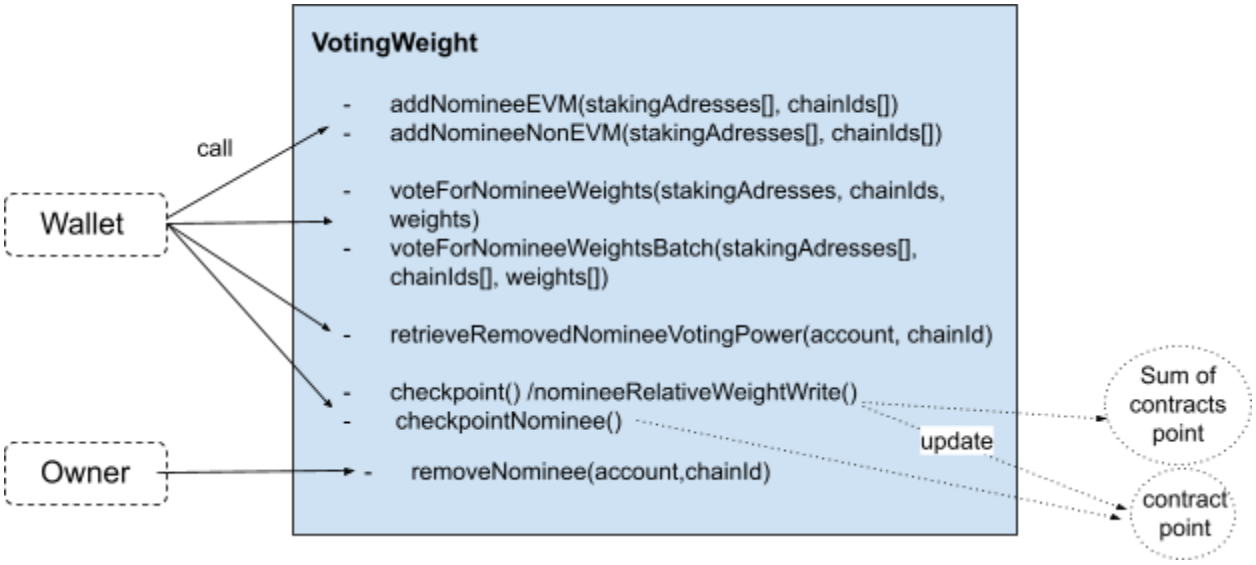
Functionality summary.

1. Use the **addNomineeEVM()** method for a staking contract on an EVM-compatible chain, or, use the **addNomineeNonEVM()** method for nominees on non-EVM chains: Any user can add staking contract addresses, along with the corresponding chainId.
2. Voting for Staking Contracts using **voteForNomineeWeights()** method for a single nominee and **voteForNomineeWeightsBatch()** method for multiple nominees:
    a. Users can vote for multiple staking contracts, with their vote weight determined by their veOLAS holdings.
    b. Votes are restricted by a cooldown period, preventing frequent voting.
    c. Users can allocate voting power for changing contract weights, with weights expressed as a percentage.
3. The contract owner[1] can update managers of the VotingWeight contracts and can remove a nominee (with **removeNominee()** method) from the contract owner and zeros its weight. Note that when a Nominee is removed, the dispenser contract (when set) is called to update its nominee records.
4. Voters can retrieve their weights from a removed nominee with **retrieveRemovedNomineeVotingPower**()
5. Checkpointing: There are functions (**checkpoint()** and **checkpointNominee()**) and for regular checkpoints to fill data common for all staking contracts and specific to each staking contract. Additionally, **nomineeRelativeWeightWrite()** method fills

[1] The contract owner will be the Timelock, so ownership methods can be executable via a DAO governance vote.

data for both nominee and the overall sum, and returns the relative nominee weight and the total sum of voting power used in veOLAS at a specified time.

6. Some methods to query information are implemented.



How the contract handles voting weights.

The contract implements weight voting parameters handling linear character of voting power per user. It records points (bias + slope) per contract in **Point**, and schedules changes in biases and slopes for those points in **changesWeight** and **timeWeight**. New changes are applied at the start of each week.

For each user, staking contract slopes are stored in **voteUserSlopes**. Additionally, the power that the user has used is recorded in **voteUserPower,** and the time their vote-lock ends for each staking contract is stored in **lastUserVote** .

The totals for slopes and biases for vote weight per staking contract are scheduled or recorded for the next week. Moreover, the points when voting power reaches 0 at lock expiration for some of the users.

When a user changes their weight vote (which can be done only once in 10 days), the change is scheduled for the next week, instead of taking immediate effect. This approach reduces the number of reads from storage required by each user, as it is proportional to the number of weeks since the last change rather than the number of interactions from other users.

The **removeNominee()** function is designed to remove a nominee from the system. This operation is restricted to the contract owner and the retainer nominee (cf. retainer in

Dispenser for more details) cannot be removed. Whether the nominee exists, the nominee's current weight is then set to zero for the next checkpoint time. The total weight sum is updated to reflect the removal of the nominee's weight. If a dispenser contract is configured, the function calls the removeNominee method on the dispenser to ensure this is aware of the removal. In the case a nominee is removed, but users have non-zero weight allocated to such a removed nominee, the voting power associated with that nominee becomes orphaned. The contract does not automatically retrieve or reallocate user voting power in the **removeNominee()** function itself. However, the contract provides a **retrieveRemovedNomineeVotingPower()** function that can be called by voters to retrieve their voting power from a removed nominee.

Additionally, note that, when a nominee is removed, the last nominee in the set will take the place of the removed nominee, changing its ID at the end of the **removeNominee()** call. It's important to note that the same ID can correspond to different nominees at different times, depending on removals.

Dispenser Contract

Overview

Originally designed to facilitate the claiming of rewards and top-ups for agent and component owners, the contract has now been updated to manage the distribution of incentives for service staking. *Since the old functionality remains unchanged, here, we focus on the functionalities related to staking incentives.*

Functionality summary.

Anyone can claim incentives for a staking contract using the **claimServiceStakingIncentives()** function, spanning from the last claimed epoch up to any subsequent epoch [2] (To avoid gas error, it is imposed a maximum number of Epochs for which it is possible to claim in one transaction. Tests will allow us to estimate gas consumed and the correct maximum value to select). For instance, if the last claimed epoch was epoch 5, users can claim from epoch 6 to epoch 11, but they cannot claim from epochs 9 to 11. Similarly, **claimServiceStakingIncentivesBatch()** allows anyone to claim staking incentives for sets staking targets. To use these functions, users need to provide an array of target staking contract addresses along with corresponding chain IDs (To avoid gas error, it is imposed a maximum number of targets and chainIDs for which it is possible to claim in one transaction. Tests will allow us to estimate gas consumed and the correct maximum values to select) . Moreover, if applicable they also need to provide an

[2] This is done to avoid unclaimed rewards deducted by inflation for a long time.

array of payloads corresponding to each staking target provided, containing additional data or parameters essential for cross-chain depositing of staking incentives.

Anyone can call the **retain()** method that allows to retain staking incentives according to the retainer address to return it back to the staking inflation.

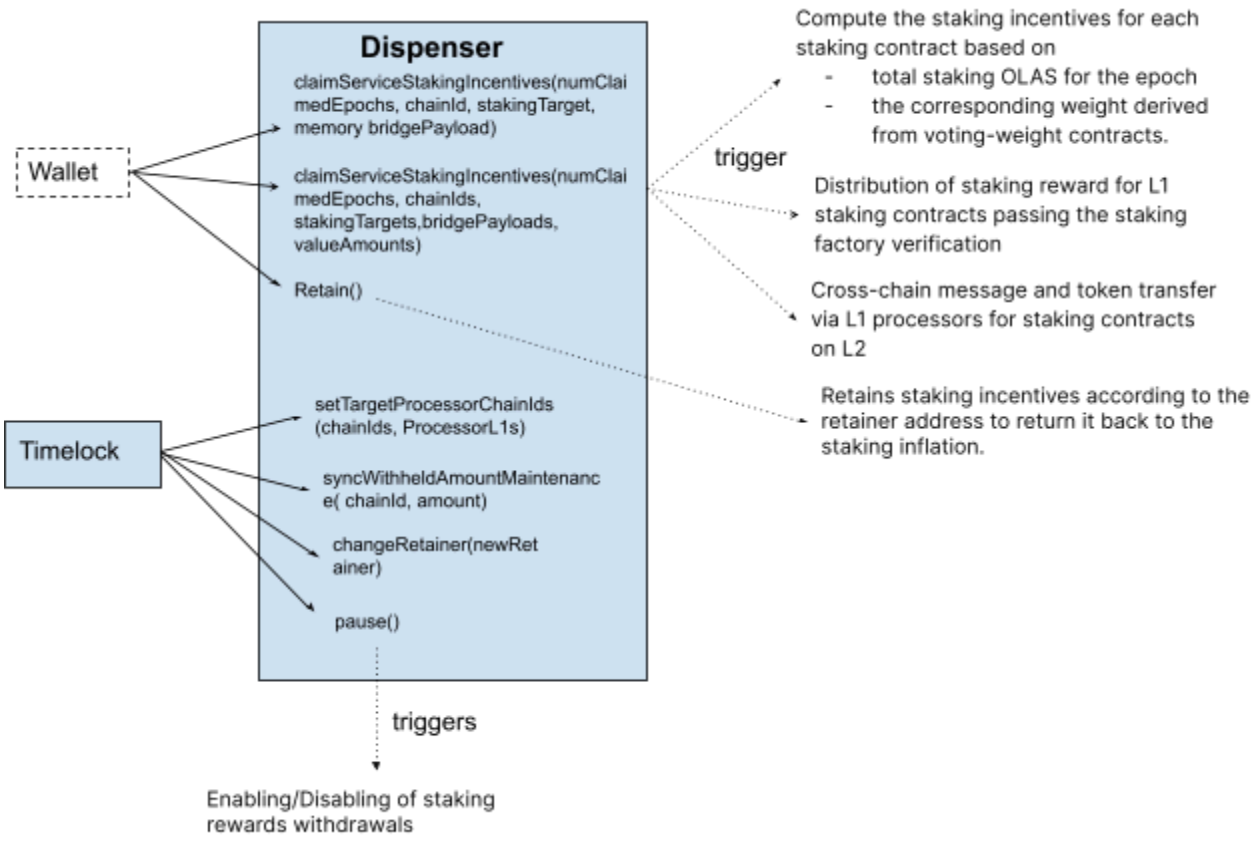
The DAO (via a governance vote) retains the authority to pause the sets the pause state of the contract, controlling various incentive claiming operations based on the pause status, thereby enabling the Dispenser to access OLAS tokens from the Treasury for incentives solely when the function is unpaused.

The DAO (via a governance vote) can set target processor contracts (responsible for cross-chain chain specific logic) and L2 chain Ids via **setTargetProcessorChainIds()** method. DAO is responsible for setting correct L1 target processor contracts.

The DAO (via a governance vote) can change staking parameters, i.e. maximum number of claiming epochs and staking targets via **changeStakingParams()** method.

Finally, the DAO retains the authority to synchronize information on the amount withheld[3] by L2 TargetStakingDispenser via **syncWithheldAmountMaintenance()** function. DAO is responsible for setting the correct amount for the specific ChainID and for using this method when there are failures (for bridge provider downtimes) or errors (incorrect source or calculation on withheld amount from the L2 side) in the automatic L2-L1 synchronization (cf. **syncWithheldTokens()** method in the TargetStakingDispenser).

[3] The TargetStakingDispenser withholds staking rewards for staking target addresses that fail factory verification.



How the contract handles staking calculation and distribution

Incentives calculation.
Incentives are distributed based on staking weights derived from the voting-weight calculated for each staking target. Note that only staking contracts fulfilling a minimum vote threshold[4] are entitled to rewards. Additionally, to prevent any single staking contract from acquiring an unfairly large share of OLAS within a single epoch, each contract is capped at receiving the maximum[5] amount of staking rewards per epoch**.**

The Dispenser sources **ServiceStakingPoint**-related information, encompassing OLAS allocation for staking across epochs, vote weighting thresholds, and maximum OLAS allocations per contract, from the Tokenomics contract. It iterates through the provided staking targets, computing the staking amount for each staking contract based on factors such as the total staking OLAS for the epoch and the corresponding weight derived from voting-weight contracts.

[4] This parameter is set by the DAO via a governance process
[5] This parameter is set by the DAO via a governance process

Upon calculating the staking amount for each target, the contract compares the staking weight against the service staking weighting threshold (resp. maximum amount of contract staking rewards per epoch).

Processing Staking Amounts.
When the **claimServiceStakingIncentives()** function is unpaused, the contract proceeds to process staking reward amounts. The contract initiates the withdrawal of OLAS tokens from the Treasury to cover the staking amount deposit, which are subsequently distributed to the corresponding staking targets (resp. stakingDispenserTargets for L2s). Specifically, distribution is managed using chain-specific processors that define the logic for L1-L1 or L1-L2 token-message transfers.

Retain.
The retaining logic is designed to manage staking incentives that DAO members have voted to retain. Specifically, this logic ensures that OLAS emissions, which have been decided to be retained by veOLAS holders through voting, are returned to the staking inflation pool for the next epoch.

TargetDispenserL2 Contract

Overview

This contract serves as an intermediary to bridge messages and OLAS staking rewards between L1 and L2 networks.

Functionality summary.

Message Processing.
Receive Messages: Initiated by the source processor contract on L1, the contract processes incoming messages via the receiveMessage() function. This function validates several parameters:

- The caller must be the L2 message relayer.
- The message must originate from the L1 chain.
- The source processor (L1 cross-chain OLAS and message passer) address must be correct.

Upon successful validation, the contract processes the received data by extracting information such as the target staking contract address, the amount of OLAS intended for the target staking contract, and update the stakingBatchNonce. The stakingBatchNonce is used to generate unique identifiers (hashes) for each staking transaction; this ensures that each transaction, even if targeting the same address with the same amount, has a unique identifier.

If thebalance of the contract is sufficient, the function is unpaused, and the staking factory verification passes[6],the transfer transaction sending OLAS staking rewards to the target staking contract is executed. See section Verification on staking contract enabled by StakingVerifier for details on the token amount sent to the staking contract and amount withheld by the targetStakingDispner (resp. transfer to Timelock by the Ethereum processor for L1 instance). Ifthe staking factory verification passes and either the balance of the contract is insufficient or the function is paused, the transaction is queued for later processing. In the computation of the queueHash.
Finally, if the factory verification fails, staking emissions are withheld (resp. transfer to Timelock by the Ethereum processor for L1 instance) by the TargetStakingDispenser.

Redeem Functionality.
The **redeem()** function, callable by anyone, facilitates OLAS token transfers to staking contracts by processing queued transactions identified by their queueHash. Upon verifying that the target and amount are queued and the contract's sufficient OLAS balance, the specified amount is transferred, and the corresponding entry in the stakingQueueingNonces mapping is marked as processed.

Cross-Chain Messaging.
The **syncWithheldTokens()** function allows sending a cross-chain message to the L1 Dispenser to inform about the OLAS emissions withheld in the contract, e.g. emissions which were reserved for staking target contracts and that failed the verification process. This ensures that withheld tokens are considered in future staking reward calculations on L1.

DAO Controls (via governance votes).
The DAO retains the authority to **pause()** the contract, thereby freezing fund and message transfers in case of attacks, bugs, or bridge provider's downtimes.

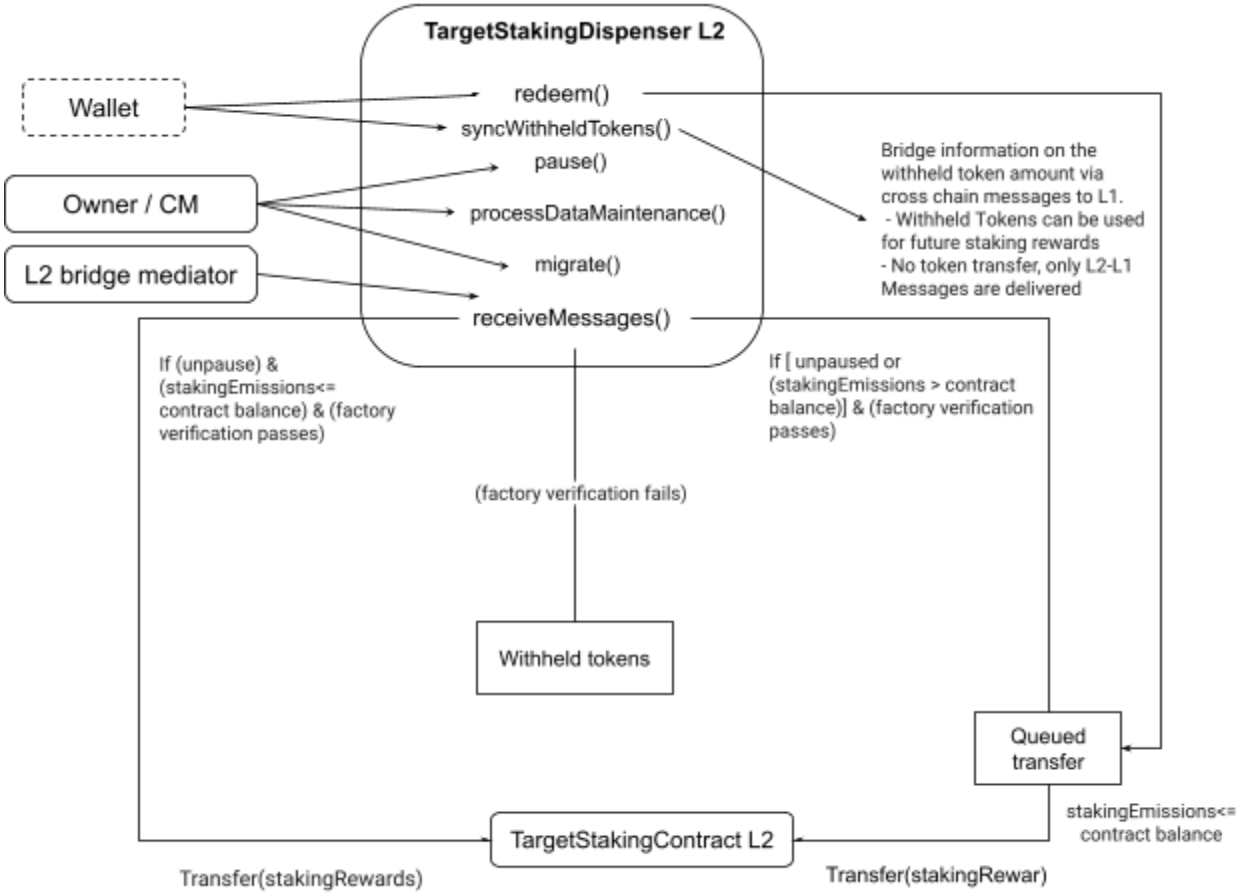
The DAO retains the authority to synchronize information on this contract. Specifically, the **processDataMaintenance()** function in the DefaultTargetDispenserL2 contract is designed to handle scenarios where the normal automated data processing for L1-L2 communication fails due to issues such as bridge failures, errors, or incorrect data from the source chain (L1). Specifically, this processes the data manually provided by the DAO

[6] For more details on the factory verification logic see section Verification on staking contract enabled by StakingVerifier

in order to restore the data that was not delivered from L1. DAO is responsible for setting the correct amount for the specific ChainID and the specific target address and for using this method only when there are failures (for bridge provider downtimes) or errors (incorrect source or calculation of staking reward amount from the L1 side) in the automatic L1-L2 synchronization (cf. Modular approach for L1-L2 messages and token transfers).

The DAO retains the authority to use the **migrate()** function designed to transfer funds from the current targetDispenser to a specified new target dispenser contract on the same L2. If there are withheld tokens, once the migration happens, the DAO directly **syncWithheldTokens()** the Dispenser on the L1 side. If there are outstanding queued requests, once the migration happens, they can be processed via **processDataMaintenance()** by DAO directly on the L2 side.

Finally, the DAO retains the ability to withdraw native funds sent to the contract via the **drain()** method.



Modular approach for L1-L1 and L1-L2 message and token transfers

Since each L2 network might have its own contract interaction implementation - e.g. for Gnosis chain we use native bridge for message and token transfer, while for Celo we use Wormhole - but at the same time utilize the same message passing algorithm from chain to chain, the modular processor approach is suggested. This implies the association of each L2 chain Id with the specifics of cross-chain interaction and dedicated constants / contracts corresponding to that chain Id.

Each processor module comes in the form of an external library (without its own storage) set up with required parameters. When the deposit takes place, the parsed calldata points to the processor contract. Using that information, a corresponding library is called to verify the rest of the logic for message and token transfer.

Here is a diagram depicting the L1-L1 and L1-L2 modular architecture.

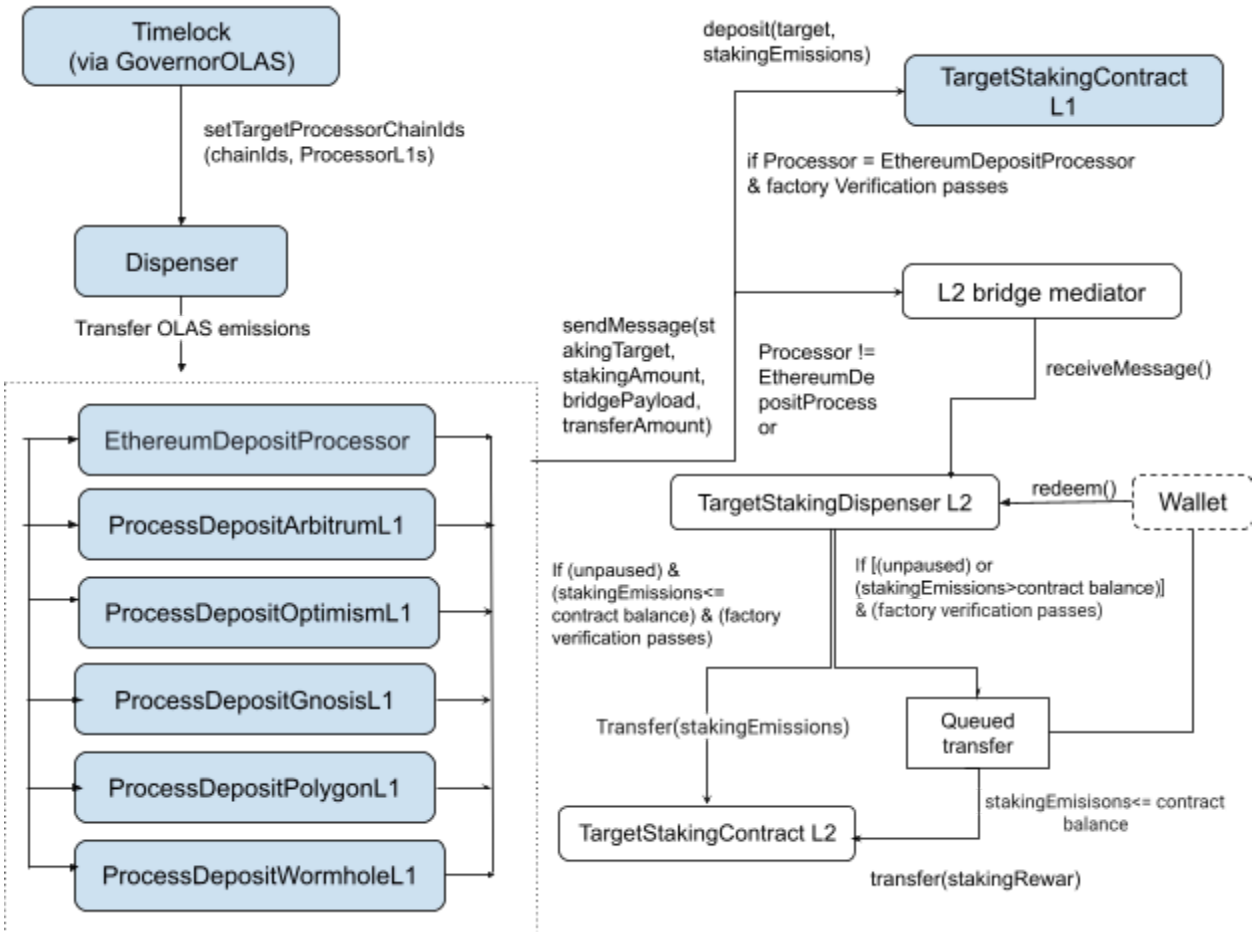


Fig 1. Modular approach: L1-L1 and L1-L2 staking deposit from Dispenser to targetStaking contracts via corresponding L1-L2 processors.

In Fig. 1, one can observe the setup of the processor by the means of a *DAO* vote. Note that, after the DAO vote is executed, the Timelock will update the relevant information in the Dispenser contract. Each target processor has a unique association with the L1 bridge mediator that relays data between L1 and L2. Thus, when a deposit is made for an L2 target Staking contract, this will trigger a bridge transaction using the target processor logic.

The bridge transaction triggers the L2 bridge mediator. The latter, engaging via L2 target StakingDispenser **receiveMessages()** function, allows for the validation of the necessary parameters and triggers a transfer transaction sending OLAS staking rewards to the target staking contract, or queues the transaction for later processing, or withheld to the the staking emissions reserved to staking target failing factory verification.

As already mentioned above, the TargetStakingDispenser implements a **redeem()** function callable by anyone to facilitate OLAS token transfers to staking contracts for queued transactions.

L2-L1 message

The target staking dispenser on L2 allows anyone to broadcast to L1 information on the emissions withheld via **syncWithheldAmount()** method. Here is a diagram depicting the L2-L1 message architecture.

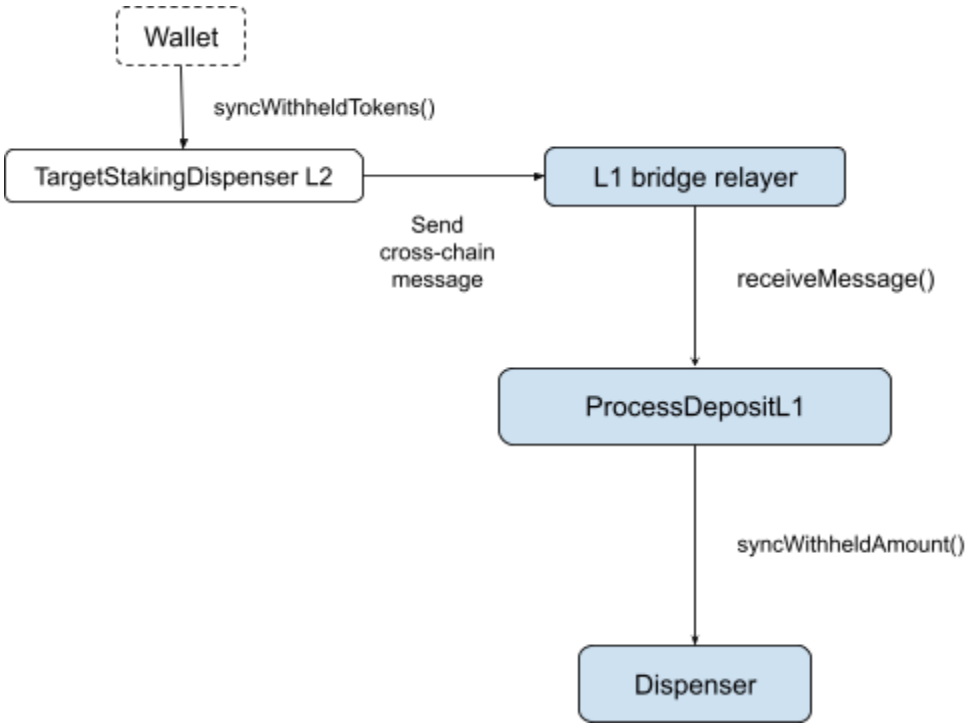


Fig 2. Inheritance approach: TargetStakingDispenser sending messages to L1 Dispenser via the L2-L1 processors.

TokenomicsContract

Overview

Originally designed to manage dev incentives and the bonding, the contract has now been expanded to include the management of the incentives for staking. Since the old

functionality remains unchecked, here, we focus on the functionalities related to staking incentives.

Functionality summary.

In the tokenomics contract we have **ServiceStakingPoint** structure encompassing information regarding the following parameters: OLAS allocation for staking per epoch, vote weighting threshold (minStakingWeight), maximum OLAS allocations per contract (maxStakingIncentive), and the staking fraction (i.e. the share of the inflation reserved to staking rewards per the epoch). The DAO has the authority to adjust the weighting threshold, maximum OLAS allocations per contract, and the staking fraction, subject to the constraint that the sum of staking fraction, OLAS top-up, and maximum bond fraction must not exceed 100.

In addition to **ServiceStakingPoint**, there's the effectiveStaking parameter, which represents the staking fraction for the epoch combined with any remaining amounts from previous epochs.

This function, **refundFromStaking()**, records the amount returned from emissions to the inflation pool. It checks if the caller is the authorized dispenser and updates the staking incentive for the current epoch.

The checkpoint function can be called by anyone and is pivotal in calculating the epoch's effective staking based on the parameters defined in ServiceStakingPoint. The effective staking value derived from this function can be subsequently utilized by the Dispenser contract for determining staking rewards in the upcoming epoch.

StakingFactory contract, Staking contract implementation stakingContractProxy, StakingActivityCheck, and StakingVerifier contracts.

Overview

The StakingFactory, staking contract implementation, stakingContractProxy, and StakingActivityCheck, and StakingVerifier contract collectively form a robust system for managing staking services within an Olas ecosystem. The StakingFactory serves as a gateway for deploying staking contracts, offering flexibility through customizable verification logic. Staking contracts, designed for compatibility, provide standardized functionalities while allowing variation in activity checks tailored to specific use cases. The StakingActivityCheck contract optimistically ensures that stakers meet predefined activity criteria, contributing to the fair distribution of rewards.

StakingFactory

Anyone can create instances[7] of a staking contract implementation through the factory via **createServiceStakingInstance()** method. Upon calling this method, it is checked that the implementation address is not zero, the implementation is a contract, and when the stakingFactoryVerifier[8] address is not zero, it verifies whether the implementation and configurations of the instance adhere to predefined conditions embedded in the stakingFactoryVerifier logic.
If the verification is successful, a contract instance is created using the **create2()** method to deterministically deploy it based on the chain ID and nonce.

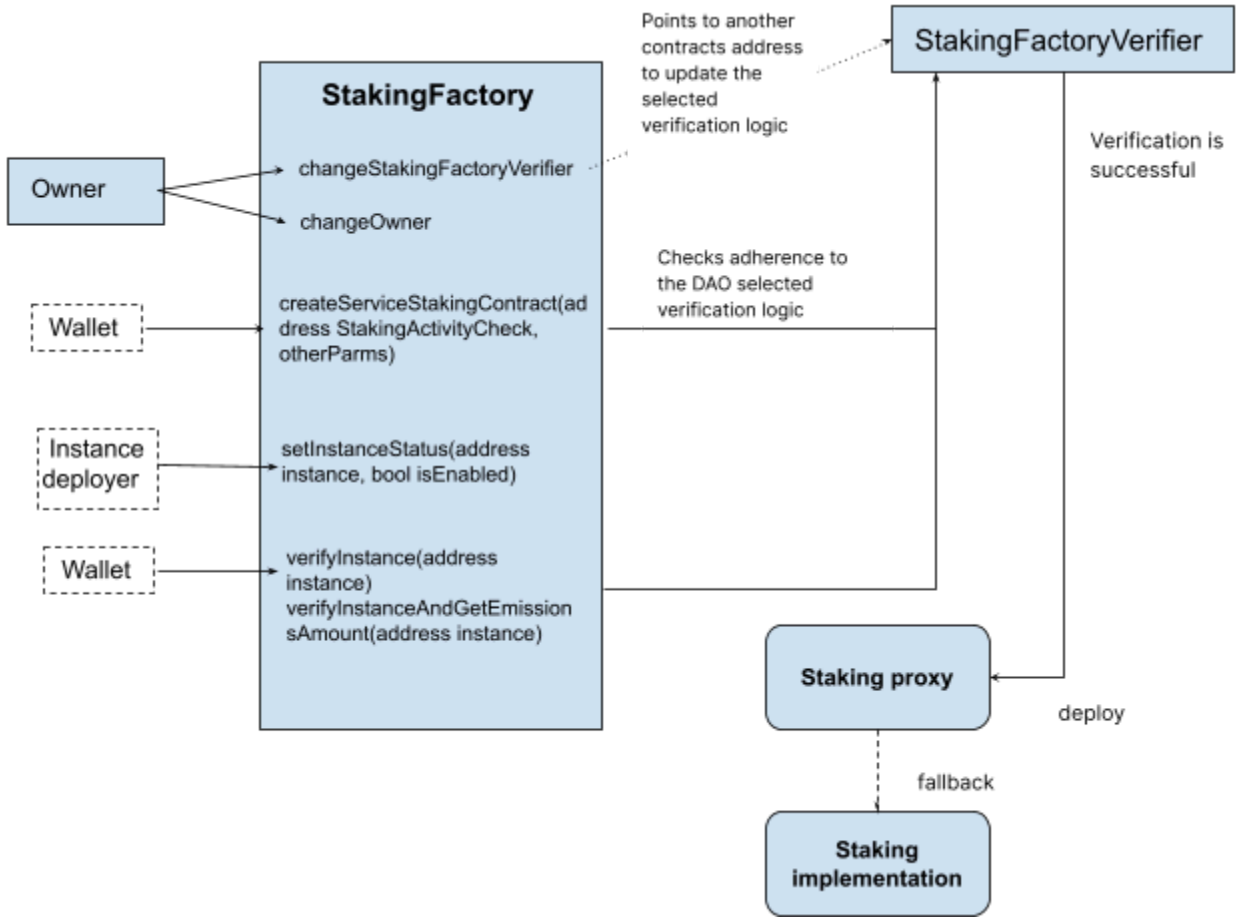
The deployer of a staking instance can update the status of an instance enabling or disabling it via the **setInstanceStatus()** method. This can be used when, for example, an instance was created, but the deployer wants to deprecate such an instance to receive emissions from the dispenser or targetDispenser contract.

Anyone can use the **verifyInstance()** method which inherits the logic of the stakingFactoryVerifier when not zero and returns false whether the deployer of a staking instance disabled such created instance. This is relevant for StakingTargetDispenser and EthereumProcessor to verify that configurations of the staking target receiving emissions are correctly configured according with DAO selected parameters.

Anyone can use the **verifyInstanceAndGetEmissionsAmount()** method. This function is designed to verify a proxy instance and then retrieve and possibly adjust an emissions amount based on certain conditions. Specifically, this retrieves the emissions amount from the verified instance. If a verifier is set, it checks the maximum allowed emissions for the instance and adjusts the amount if necessary to ensure it does not exceed the limit. The final emissions amount is then returned.

Additionally, the DAO can update the address of the contract owner and of the stakingFactoryVerifier via a governance vote. The latter allows the DAO to retain the ability to select and update the necessary checks on staking implementations and instance configurations by updating stakingFactoryVerifier without the need of the factory redeployment.

[7] See service staking proxy
[8] For more details on the factory verification logic see section Verification on staking contract enabled by StakingVerifier



Staking contract implementation

Compatible staking contracts are contracts selected by the DAO that adhere to specific configurations and share identical contract logic, with the exception of the activity check, which may vary.

The staking contracts compatibility ensures that the contracts can be registered and deployed using a standardized implementation via the factory contract (cf. section Staking Contracts Factory), streamlining the deployment process and allowing for diverse activity criteria tailored to specific use cases. Once the stakingVerifier address is set in the staking factory and the check for a prescribed standardized implementation is enabled, the core logic of compatible staking instances deployed with the factory remains consistent. This includes essential functionalities related to staking rewards, reward distribution and service states. The flexibility lies in the variability of their activity check, which can be customized to suit the specific requirements of different autonomous services.

ActivityCheck

At the deployment time, the staking instance deployer needs to provide the address of the StakingActivityChecker contract. The latter will just handle the logic to monitor whether a specific service activity has been performed.

How does the staking contract interact with the activity checker?

Once the stakingContract.checkpoint() is called, this triggers stakingContract._calculateStakingIncentives(). The latter, in turn, triggers a call() to the StakingActivityChecker.isRatioPass() method. Whether StakingActivityChecker. isRatioPass() returns true (resp. false), the staking contract assumes that the activity has (resp. hasn't) been performed and such a service (should not) receives staking rewards. In order to avoid unexpected behaviors, the stakingContract will always handle as false any possible reverts (or non-bool return) from the StakingActivityChecker. isRatioPass() method.

Configurations to select to deploy a staking instance.

Compatible staking contracts introduce have the following configuration:

- Token for Rewards: The token in which staking rewards are distributed to participants. This has to be the address of the canonical OLAS address on L1 or L2 depending on the fact that the staking instance needs to be deployed on L1 or L2.
- Number of Agent Instances per Autonomous Service: The predetermined quantity of agent instances associated with an autonomous service registered in the staking contract.
- Max Number of Stakers: The maximum amount of stakers the contract permits at any given time. This needs to be lower than the MaxNumberOfStakerLimit selected by the DAO on the stakingVerifier contract
- MinStakingDeposit: The established value of minimal non-slashable security deposit and minimal slashable operator bonds required for staking.
- Liveness Period: The designated time frame during which the staking contract assesses the activity of the service. This sets a regular interval for evaluating the service's activity, contributing to the determination of staking rewards.
- timeForEmissions: The designated time frame during which the staking contract is expected to run and hence to receive emissions.
- Max Allowed Inactivity: The maximum duration of inactivity permitted for the service before facing potential eviction. This sets a threshold for acceptable inactivity, preventing services from remaining inactive for extended periods.
- Min Staking Duration: The minimum required duration for which a service must remain staked to qualify for staking rewards. This ensures incentive compatibility, as staking without activity incurs an opportunity cost equivalent to foregone staking rewards.

- Reward Per Second: The designed yield that service can receive per second, contingent on the activity check confirming correct agent activity. This needs to be lower than the RewardPerSecondLimit selected by the DAO on the stakingVerifier contract. The annualized amount represents the active staker's APY
- numAgentInstances: Number of agent instances in the service
- agentIds: agent Ids in the service. This is an optional parameter
- Threshold: service multisig threshold requirement. This is an optional parameter
- configHash: service configuration hash requirement. This is an optional parameter
- proxyHash: Approved multisig proxy hash
- serviceRegistry: Address of the ServiceRegistry contract
- serviceRegistryTokenUtility: Address of ServiceRegistryTokenUtility contract.
- activityChecker: Service activity checker address
- metadataHash: Metadata staking information, i.e., name plus contract description that will be saved on IPFS

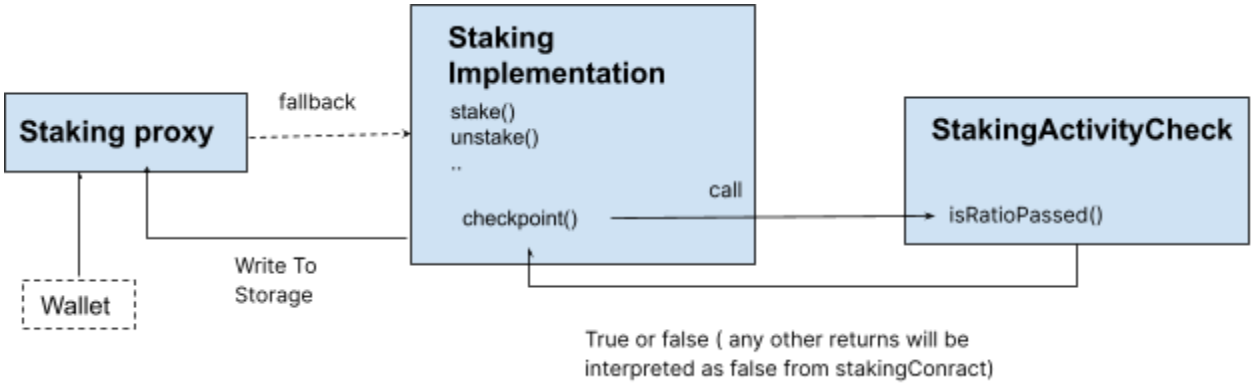
StakingContractProxy

The ServiceStakingProxy contract serves as a proxy for the implementation logic of a staking system. Proxy implementation is created based on the Universal Upgradeable Proxy Standard (UUPS) EIP-1822. Note that, from the EIP-1822 standard, we do not need the upgradability, but only used that to store the implementation in a pre-calculated slot.

Upon deployment, the constructor requires the address of the service staking implementation. It ensures that the provided implementation address is not zero. The implementation address is stored in a specific storage slot designated for the proxy contract.

The contract implements a **fallback()** function that delegates all incoming calls to the stored implementation address. It copies the calldata to memory, executes a delegatecall to the implementation contract, and handles the return data. If the delegatecall is unsuccessful (returns false), it reverts with the received error data.

The contract provides the **getImplementation()** function to retrieve the current implementation address. This function reads the implementation address from the designated storage slot and returns it.



StakingActivityCheck contract

This contract should contain customized logic to check whether stakers performed a predefined activity as contained in the contract logic. The contract should return true when the service passed the predefined encoded check and false otherwise. The basic StakingActivityChecker contract for a compatible staking implementation can extend the functionality by adding new methods and modifying existing ones. Specifically, whether prescribed on-chain actions are done within specific timeframes can be checked by

1. Extending the **getMultisigNonces()** method to consider more nonces, each representing a specific number of actions done on-chain (e.g. mech calls, market created).
2. Updating the **isRatioPass()** method to include the new checks.
3. Selecting the **livenessRatio** parameter indicates exactly how many on-chain actions are expected to be completed during the livenessPeriod.

The protocol optimistically assumes that the StakingActivityChecker contract used for deploying staking instances is implemented with correct logic. Therefore, unless unexpected behavior such as reverts or non-boolean returns occur, the contract's results will be considered accurate. However, this optimistic assumption can be exploited by malicious users. For instance, malicious users could deploy multiple contracts with flawed activity checks that always return true. They could then vote for these contracts, causing the OLAS amount to be distributed to all stakers, including those without activity. Conversely, malicious users could deploy contracts with incorrect liveness checks that always return false, leading to a situation where the OLAS amount is sent, but funds remain stuck in the staking contracts and cannot be recovered.
The following measures can be considered to mitigate eventual abuses:

1. Set a Sensible Threshold: The DAO needs to establish a sensible threshold to enable staking emissions.
2. On-Chain Blacklist: Implement an on-chain blacklist that can be updated through governance votes, allowing the community to monitor and exclude malicious contracts.

3. Off-Chain Reputation System: Consider using an off-chain reputation system, possibly leveraging oracles, to assess the trustworthiness of contracts.

StakingVerifier contract

The StakingVerifier contract is designed to verify the parameters and statuses of service staking contracts. It ensures compliance with specific staking limits set by the DAO and maintains a whitelist of implementation addresses.

Key Features

1. Enable/Disable Whitelisting Check: The DAO can toggle the implementation whitelisting check by setting implementationsCheck to true or false.
2. Update Whitelist Status: The DAO can update the whitelisting status for a list of implementation addresses.
3. Staking Rewards Limits
    a. The DAO can set or update the following limits via a governance vote:
        i. rewardsPerSecondLimit
        ii. timeForEmissionsLimit
        iii. maxNumServicesLimit

Verification on staking contract enabled by StakingVerifier

Instance Creation Verification

When a new instance is created via the staking factory, the following verifications are performed by StakingVerifier:

1. Implementation Whitelisting Check:
    a. If implementationsCheck is true, the selected implementation must be whitelisted.
    b. If implementationsCheck is false, the whitelisting status of the implementation is ignored.
4. Parameter Limits:
    a. The selected rewardsPerSecond must be less than or equal to rewardsPerSecondLimit.
    b. The number of stakers (maxNumServices) must be less than or equal to maxNumServicesLimit.

If any of these conditions are not satisfied, the createInstance process is reverted, preventing the instance creation.

Additional check to ensure avoid sending too much emissions to staking instances

The staking factory uses StakingVerifier to:

1. Verify Proxy Instance Parameters:

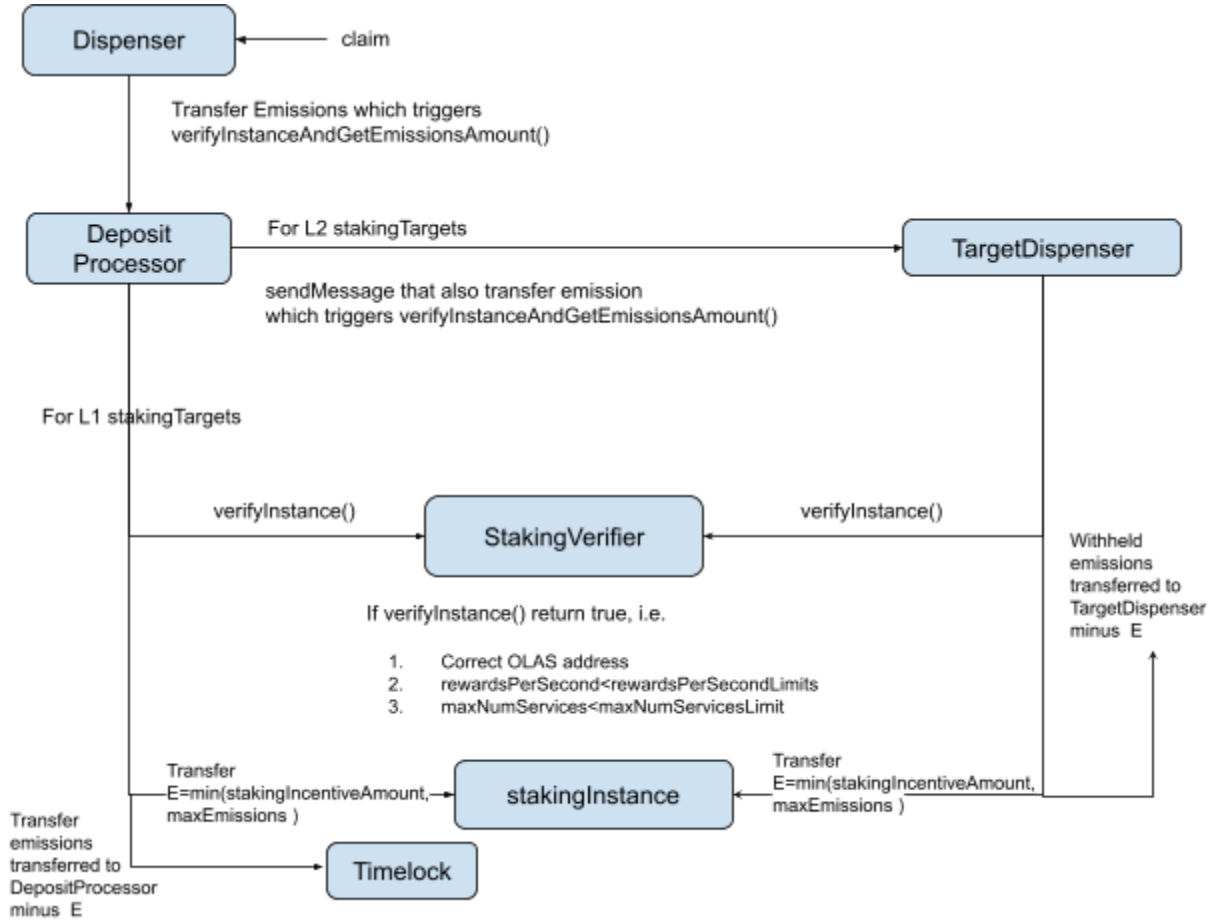
    - Ensure that the proxy instance parameters comply with the DAO-set limits. (rewardsPerSecond and maxNumServices staking instance parameters are less than or equal to rewardsPerSecondLimit and maxNumServicesLimit).
2. Retrieve and Adjust Emissions:
    - Calculate the emissions amount using the formula:
        - emissionsAmount = rewardsPerSecond * maxNumServices * timeForEmissions, where rewardsPerSecond, maxNumServices.
    - Compare the calculated emissions amount to the allowed maximum:
        - maxEmissionsAmount = rewardsPerSecond * maxNumServices * timeForEmissionsLimit
    - Adjust the emissions amount if necessary to ensure it does not exceed the allowed limit. Specifically, stakingIncentives to send to the stakingInstance are E= min(emissionsAmount,maxEmissionsAmount)

The TargetDispenser (resp. EthereumDepositProcessor) for an L2 (resp. L1) staking instance, provided that instance is verified, send

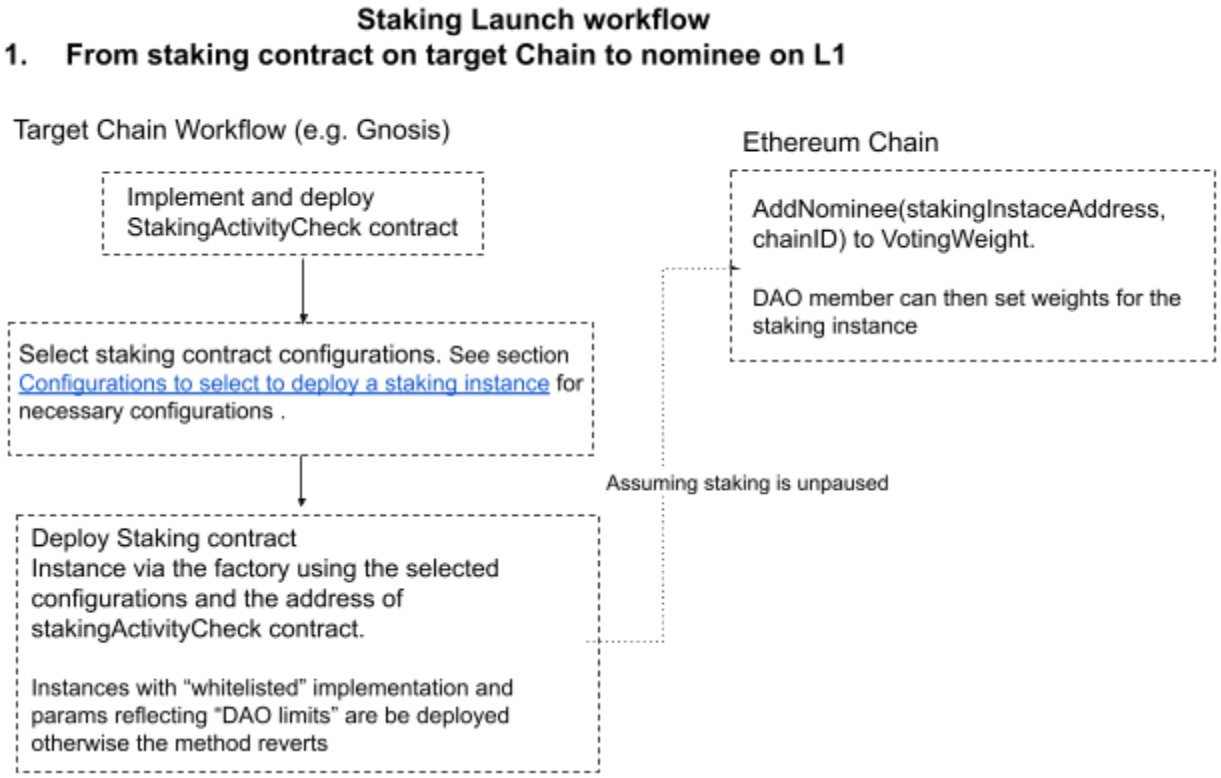
E= min(emissionsAmount,maxEmissionsAmount)

to the staking instance and withheld (resp. transfer to the Timelock)

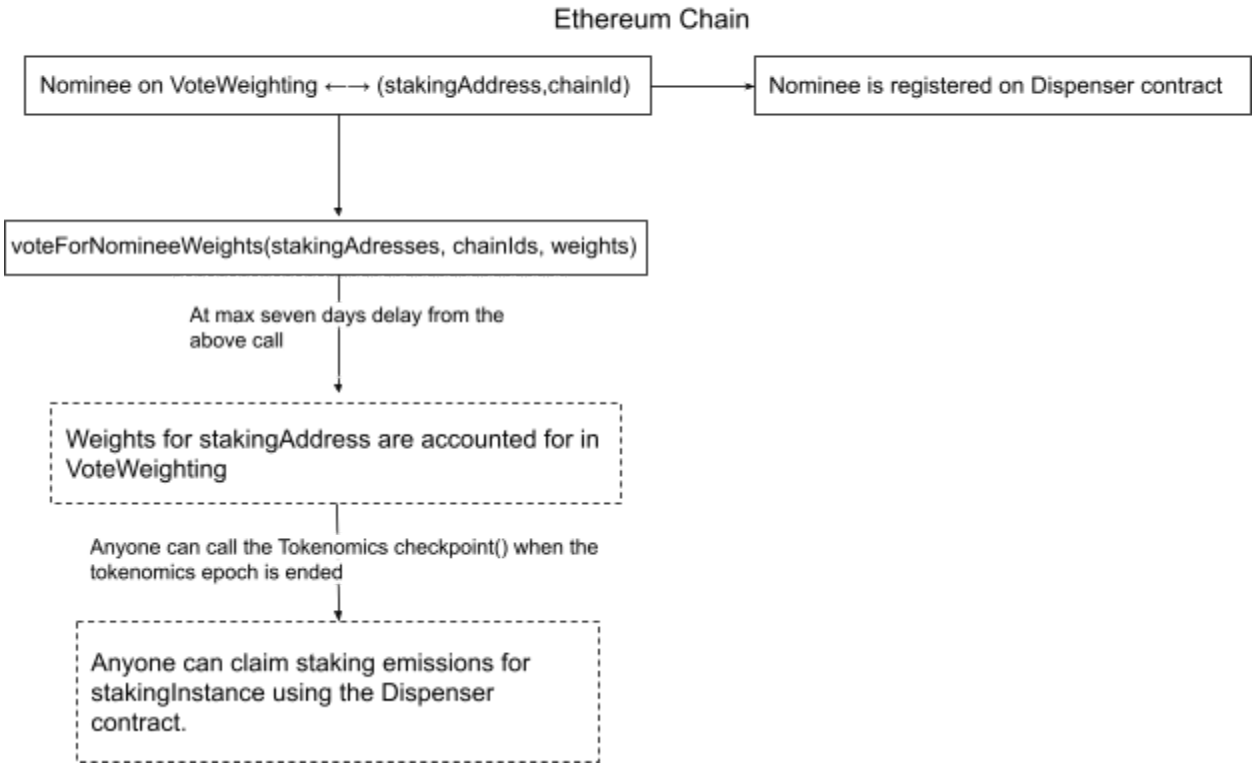
AmountOfEmissionsSendByDispenser-E


Dispenser
claim
Transfer Emissions which triggers
verifyInstanceAndGetEmissionsAmount()
Deposit Processor
For L2 stakingTargets
TargetDispenser
sendMessage that also transfer emission
which triggers verifyInstanceAndGetEmissionsAmount()
For L1 stakingTargets
verifyInstance()
StakingVerifier
verifyInstance()
Withheld emissions transferred to TargetDispenser minus E
If verifyInstance() return true, i.e.
1. Correct OLAS address
2. rewardsPerSecond<rewardsPerSecondLimits
3. maxNumServices<maxNumServicesLimit
Transfer E=min(stakingIncentiveAmount, maxEmissions)
stakingInstance
Transfer E=min(stakingIncentiveAmount, maxEmissions)
Transfer emissions transferred to DepositProcessor minus E
Timelock

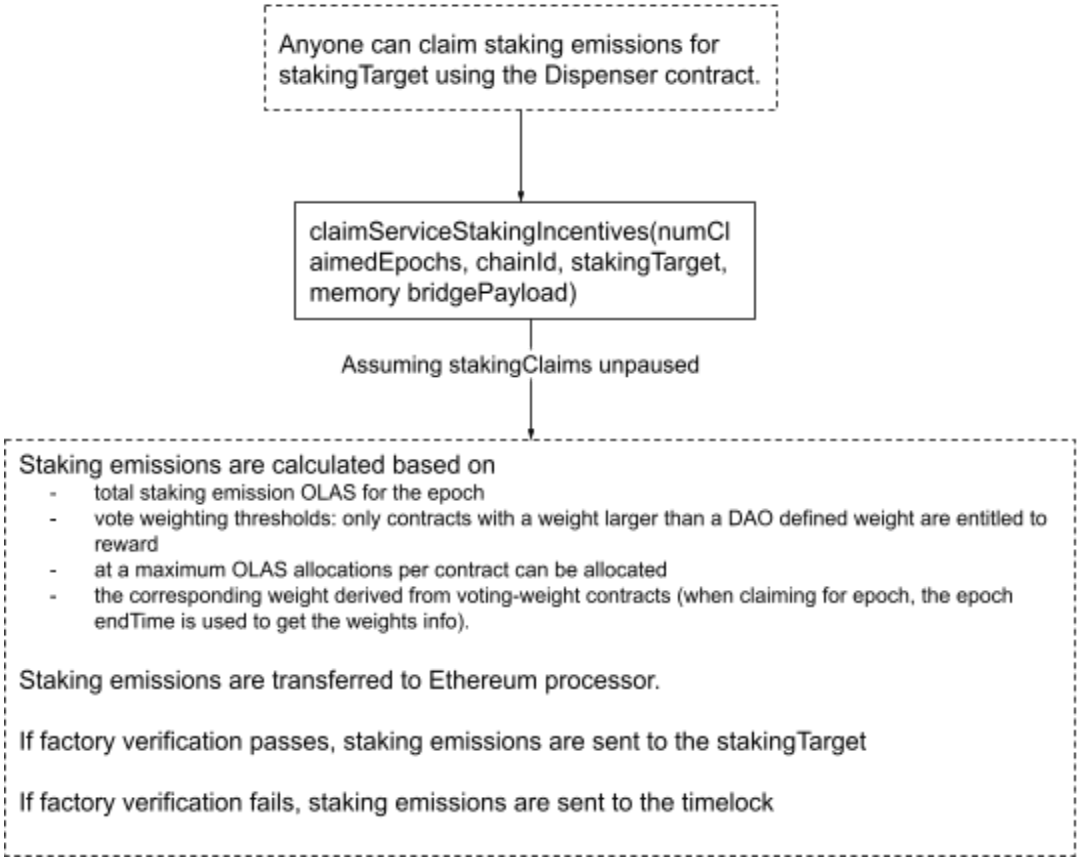
Staking workflow: launch, vote, claim



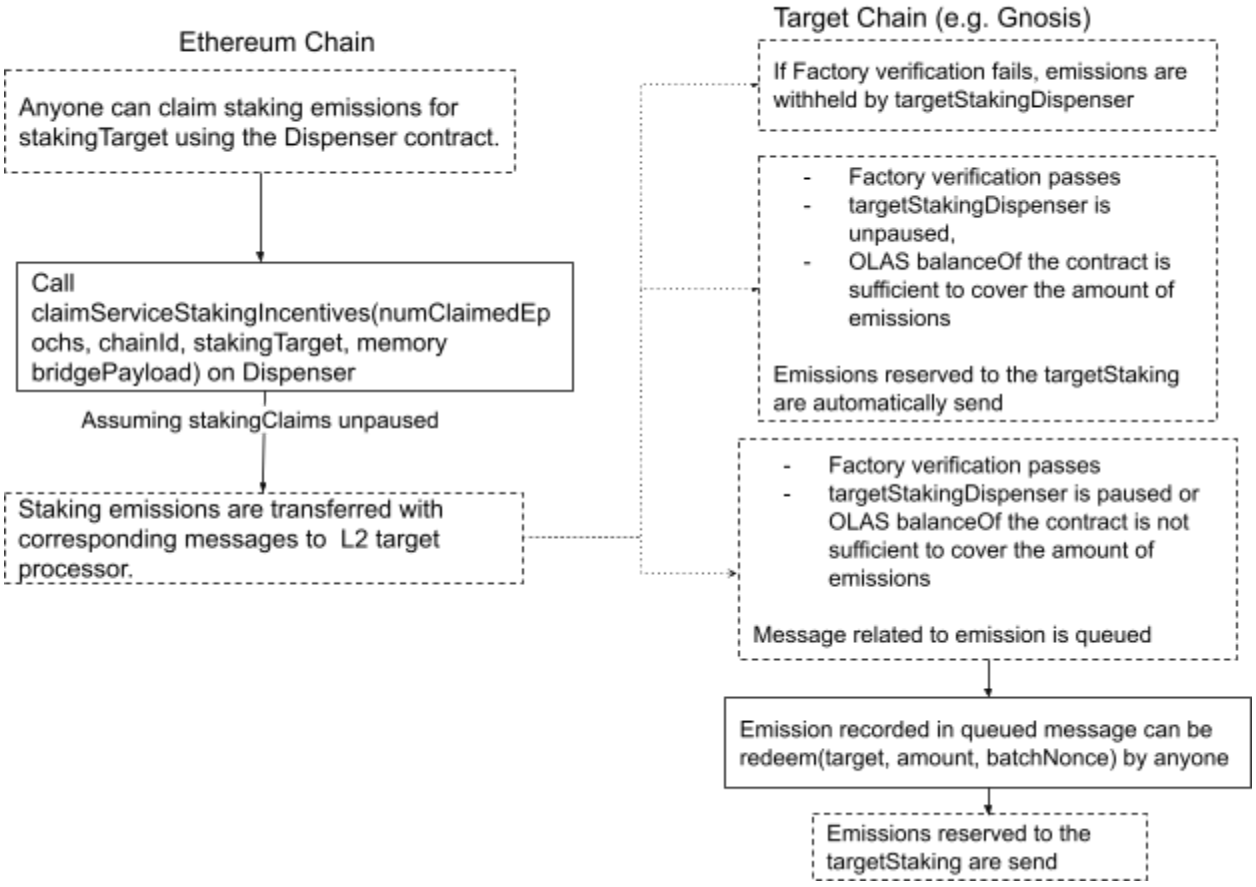
2. From nominee on L1 to claimable emissions



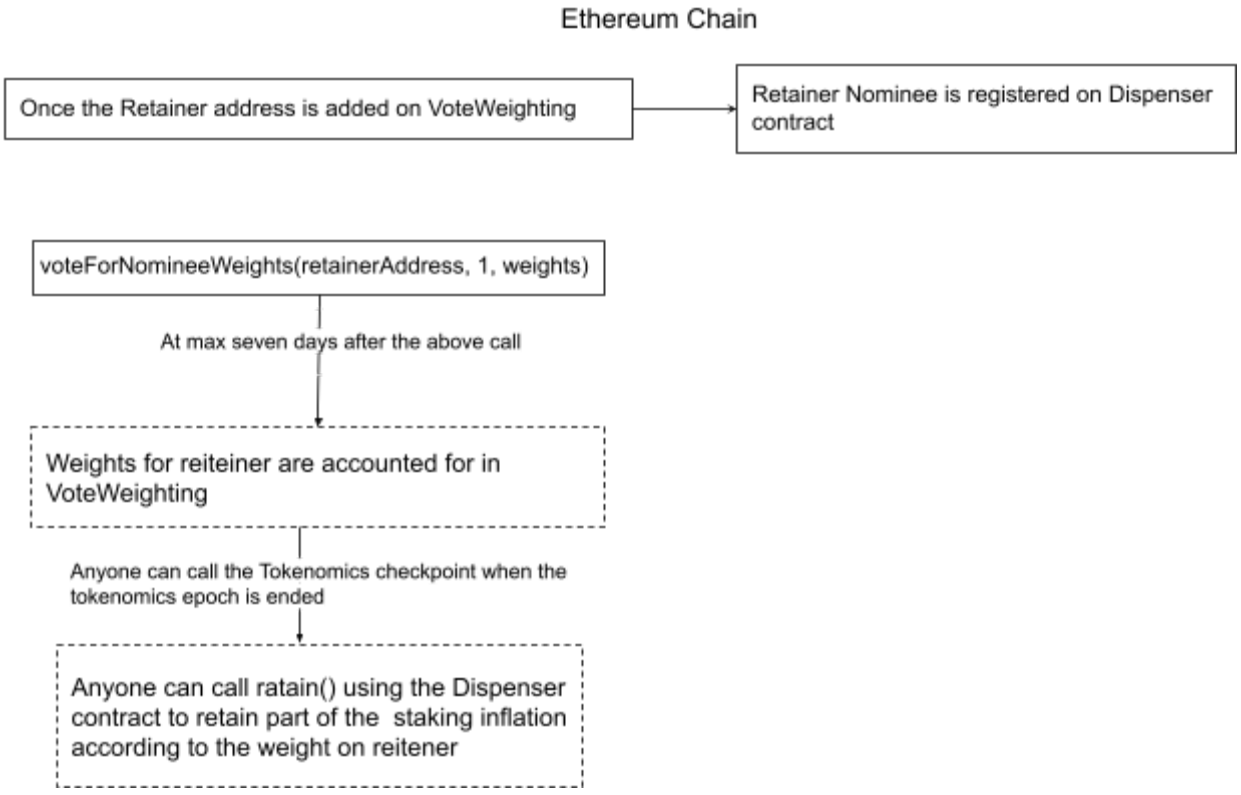
3.a From claimable emissions to emission on StakingTarget L1



3.b From claimable emissions to emission on L2 StakingTarget



3. Vote for Retain inflation



3. Disable the emissions for a stakingInstance

1. Launcher needs to acknowledge that they want their stakingInstance to receive no new emissions.

This should be done as soon as possible so that there are no new vote casted for such an instance and DAO members that previously voted for such an instance can cast a new vote with 0 weight

2. Launcher (i.e. deployer of the staking instance) needs to setInstanceStatus to false on staking factory.

Once this is done, dispenser or target dispenser will not send emissions to such an instance. For L2 instance, the eventual emissions sent by L1 will be withheld by the TargetStakingDispenser. For L1 instances, the eventual emissions will be accounted for the next epoch

3. A DAO vote is necessary to remove nominee corresponding to the stakingInstance from VotingWeight.

Before the execution, launcher should encourage DAO members that previously voted for such instance to cast a new vote with 0 weight

4. DAO members can retrieve their voting power from a removed nominee by calling retrieveRemovedNomineeVotingPower on votingWeight