vim-markdown-preview.html

Elections

Introduction

The elections component enables functionality for accounts to become representatives, for representatives to become delegate candidates and for representatives to vote for delegate candidates.

During each epoch, there is a fixed set of delegate candidates. Throughout the epoch, each representative will issue one vote request. Each vote request can contain votes for up to 8 different candidates, and is processed through Axios consensus. Each vote is weighted by the casting representative's stake. At the conclusion of the epoch, voting ceases, and the 8 candidates that received the most weighted votes are written into the next epoch block, replacing the 8 oldest delegates who were elected in a previous epoch.

An account that wishes to vote must first become a representative, and can do so by issuing a `StartRepresenting` request. A representative can become a delegate candidate by issuing an `AnnounceCandidacy` request.

Overview

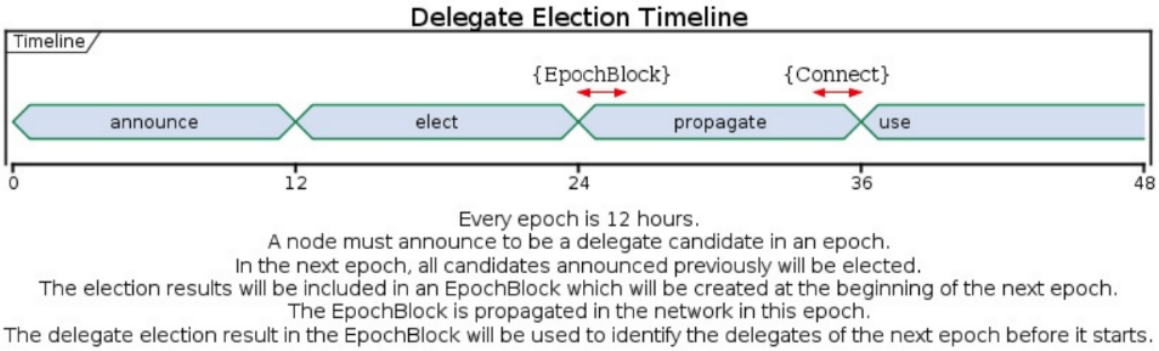
The elections component is not a single component but a collection of additions to existing components.

- Structs that inherit from `Request`
- Logic within `PersistenceManager<R>` to validate and apply these new `Request` types
- Representatives database and candidates database
- Logic within `EpochVotingManager` to select the next epoch's delegates
- Logic within `PersistenceManager<ECT>` to transition elections state to next epoch

Timeline

Candidates `AnnounceCandidacy` in epoch `i`, are voted on in epoch `i+1`, are written into the epoch block at the start of epoch `i+2` (assuming they won the election), serve as delegates from epoch `i+3` to `i+6`, and are eligible for reelection starting in epoch `i+5`. If a candidate is not elected, the candidate remains a candidate in subsequent epochs until they are elected or issue a valid `RenounceCandidacy` or `StopRepresenting`.

The epoch block post-committed in epoch `j` lists the delegates that will be active for epoch `j+1`.



Execution Concept

The processing of Requests related to elections follows the same execution model as processing any other type of Request (Send for instance).

The software keeps track of the current election results by maintaining a database, `candidacy_db`, consisting of the current candidates with their current total weighted votes received so far, which is updated each time an `ElectionVote` is post-committed. The software also maintains another database, `leading_candidates_db`, consisting of the current top 8 candidates with the most weighted votes, which is also updated each time an `ElectionVote` is post-committed.

The software keeps maintains a database, `remove_candidates_db`, consisting of any candidates that issued `RenounceCandidacy` during the epoch, as well as a database, `remove_reps_db`, consisting of any reps that issued `StopRepresenting` during the epoch.

When the epoch ends, voting ceases until the epoch block is post-committed. When the epoch block is proposed, the winners of the election are read from `leading_candidates_db`. When the epoch block is applied, multiple state changes occur:

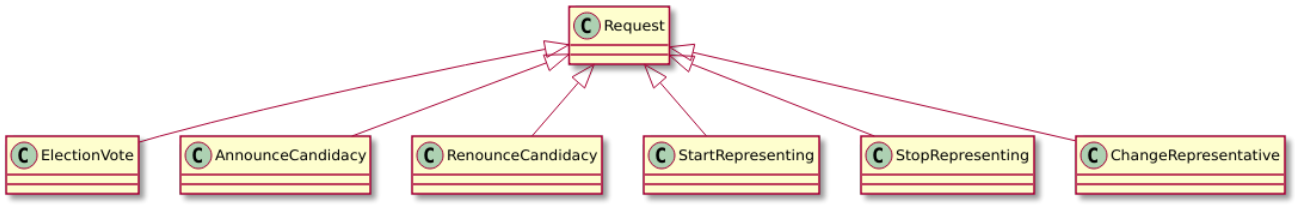
- Any candidates recorded in `remove_candidates_db`, as well as the election winners, are removed from `candidacy_db`
- Any representatives recorded in `remove_reps_db` are removed from the `representatives_db`.
- `Leading_candidates_db` is cleared. Note, the current total weighted votes for each candidate in `candidacy_db` are not reset on epoch transition, but instead are reset on the first vote that candidate receives in the next epoch.
- `remove_candidates_db` and `remove_reps_db` are cleared.

The time between epoch start and the epoch block being post-committed and written to the database (typically 10 minutes) is known as the elections dead period. No votes are accepted during the dead period; this is because prior to the epoch block being post-committed, the network has not come to consensus on the previous epoch's election results. If a candidate won the election in the previous epoch, that candidate can not receive votes in the next epoch. Delegates cannot be sure that their view of the election results is consistent with the other delegates until after the last microblock.

The elections dead period is also necessary for proper vote weighting, as a delegate may have missed some requests that affect the voting weight of a representative. All delegates must have the exact same view of the voting weight of a representative at the time votes are cast; otherwise, different delegates could end up with different election results. The elections dead period ensures that by the time votes are cast, all delegates are up to date on all requests that happened in the previous epoch (since the last microblock is proposed immediately prior to the epoch block, and delegates will know if they have missed any requests based on the last microblock). If a delegate realizes it needs to bootstrap after receiving the last microblock, that delegate cannot post-commit `ElectionVote`s until bootstrapping has finished (since that delegate will not know how to correctly weight the votes).

Interfaces

The interface to the election system is through issuing the proper `Request` which is processed through consensus.



- `StartRepresenting` Become a representative. Includes amount to stake.
- `AnnounceCandidacy` Become a candidate. If not already a representative, the account will become a representative as well. Includes amount to stake, which can be ommitted if account is already staking as a representative.
- `ElectionVote` Cast vote/s for a given epoch. Includes 1-8 current candidates
- `RenounceCandidacy` Remove from candidate list for upcoming epochs.
- `StopRepresenting` Remove from representative list for upcoming epochs. If sending account is also a candidate, this command also removes the account from the candidate list for upcoming epochs.

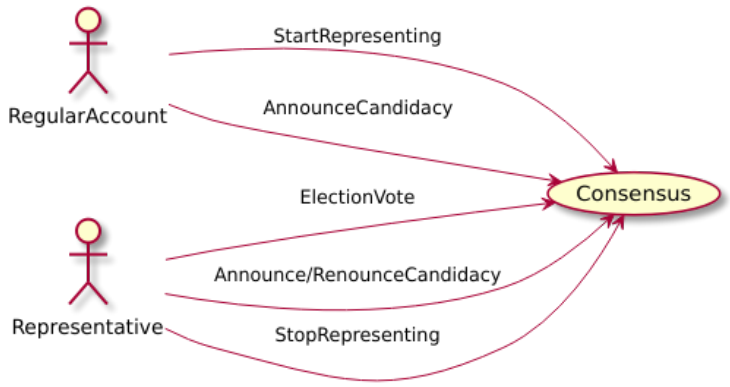
Each request, except `ElectionVote`, takes effect the epoch after that request is post-committed. For example, if an account issued `AnnounceCandidacy` in epoch `i`, then the account is a candidate no earlier than epoch `i+1`. If an account issued `StartRepresenting` in epoch `i`, the account can vote starting in epoch `i+1`. If an account issues `StopRepresenting` in epoch `i`, the account can still vote in epoch `i`, but not in any epoch after `i`. If an account issues `RenounceCandidacy` in epoch `i`, the account is still a candidate in epoch `i` (unless the account is a current delegate not yet eligible for reelection), but will not be a candidate in any epoch after `i`. Note, if a candidate issues a `RenounceCandidacy` in epoch `i` but also wins the election in epoch `i`, then the candidate still must serve as a delegate. However, the candidate will not be eligible for reelection, unless the account issues a subsequent `AnnounceCandidacy` request.

Note, a candidate can issue `StopRepresenting` while a candidate, which will remove that candidate from the candidate list and representative list for subsequent epochs. However, if that candidate wins the current epoch's election, that candidate must still serve as a delegate. This leads to the only situation where a delegate is not also a representative. The delegate can subsequently issue `StartRepresenting` or `AnnounceCandidacy` to become a representative and/or candidate again.

An account can only issue one of the above requests per epoch, except `ElectionVote`. For example, an account cannot issue `AnnounceCandidacy` and `RenounceCandidacy` in the same epoch. If an account wishes to become a representative and a candidate in one command, that account should issue `AnnounceCandidacy`. If an account wishes to stop being a representative and stop being a candidate in one command, that account should issue `StopRepresenting`.

See the IDD for further descriptions of the Requests.

Use Case Diagrams



Classes

`PersistenceManager<R>` (persistence manager for requests) has a `ValidateRequest(...)` method and `ApplyRequest(...)` method for each request type in the elections subsystem. The validation methods must take in the epoch number of the request, to ensure that the request is valid for that given epoch (for example, is the request during the dead period, did the representative already vote this epoch, etc). `PersistenceManager<ECT>` (persistence manager for epoch blocks) has a method `TransitionNextEpoch(...)` which updates the databases pertaining to representatives and candidates for the next epoch. This method calls various helper methods in the same class. `EpochVotingManager` has a method `GetNextEpochDelegates(...)` which determines the list of delegates to be written into the epoch block by swapping the retiring delegates with the election winners.

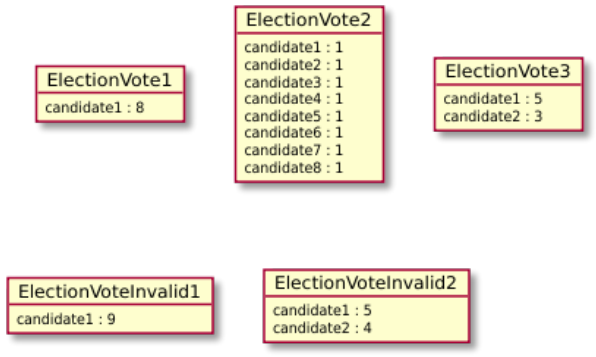
Resources

Elections will introduce five new databases. For detailed database format layout, please reference the database format documentation.

- `representative_db` : this database will store all of the representatives and any additional information about those representatives (voting weight, hash of their most recent vote, representative status). The hash of their most recent vote is stored, to ensure representatives only vote once. The hash of their most recent representative action (`StartRepresenting` , `StopRepresenting` or `AnnounceCandidacy`) is stored, as well as their most recent candidacy action (`AnnounceCandidacy` , `RenounceCandidacy` or `StartRepresenting`), to keep track of their status as a representative and candidate and validate future requests.
- `candidacy_db` : this database will store all of the current candidates, along with the amount of weighted votes that they have received so far.
- `remove_reps_db` : this database stores representatives that have issued `StopRepresenting` this epoch and will be removed from `representatives_db` when the epoch block is post-committed (see execution concept)
- `remove_candidates_db` : stores candidates that have issued `RenounceCandidacy` this epoch and will be removed from `candidacy_db` when the epoch block is post-committed (see execution concept)
- `leading_candidates_db` : stores top 8 candidates based on current election votes

Votes and Vote Weighting

Each epoch we are electing `N/L` delegates, where `N` is the total number of delegates and `L` is the term length. `ElectionVote` can contain votes for up to `N/L` different candidates. The votes are represented like a map from candidates to number of votes. In the diagram below, assuming `N/L == 8` , `ElectionVote1` , `ElectionVote2` and `ElectionVote3` are valid, while the others are not. Any `ElectionVote` that lists an account that is not an active candidate is also invalid.



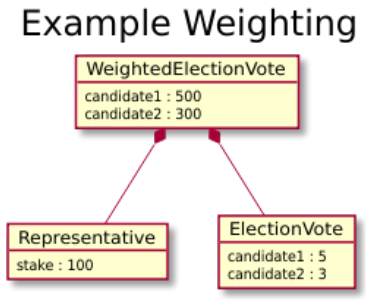
When a representative issues an `ElectionVote` , the vote is weighted by that representative's voting power. So if a representative `R` has voting power `s` , and issues an election vote with `x` votes for a candidate `C` , `C` receives `s * x` weighted votes from `R` .

Below is pseudocode for how a single `ElectionVote` is weighted and processed.

```
weightVoteAndAddToDb(ElectionVote v)
    validate v
```

```
    power = sender's voting power
    for(Candidate c in v)
        weighted_vote = (num_votes for c in v) * power
        add weighted_vote to current total votes for c in candidate_db
```

Below is an example representative, a vote that they cast and the resulting weighted vote. Note, WeightedElectionVote is not an actual class, but simply a concept used here to illustrate weighting.



Delegate Voting Power

During consensus, each delegate has a specific voting power, fixed for the entire epoch, that is proportional to the number of votes they received during elections. The function is first stated in english, and then specific pseudocode. We are capping voting power at 1/8 of total voting power, and in the rare case a delegate received 0 votes but was elected still, we give them 1 vote for free

English:

```
if a delegate has 0 votes, give it 1 vote

pick an unprocessed delegate
if delegate has greater than cap votes
    redistribute excess votes to unprocessed delegates, proportional to number of votes those delegates hav
mark delegate as processed
repeat until all delegates are processed
```

Pseudocode:

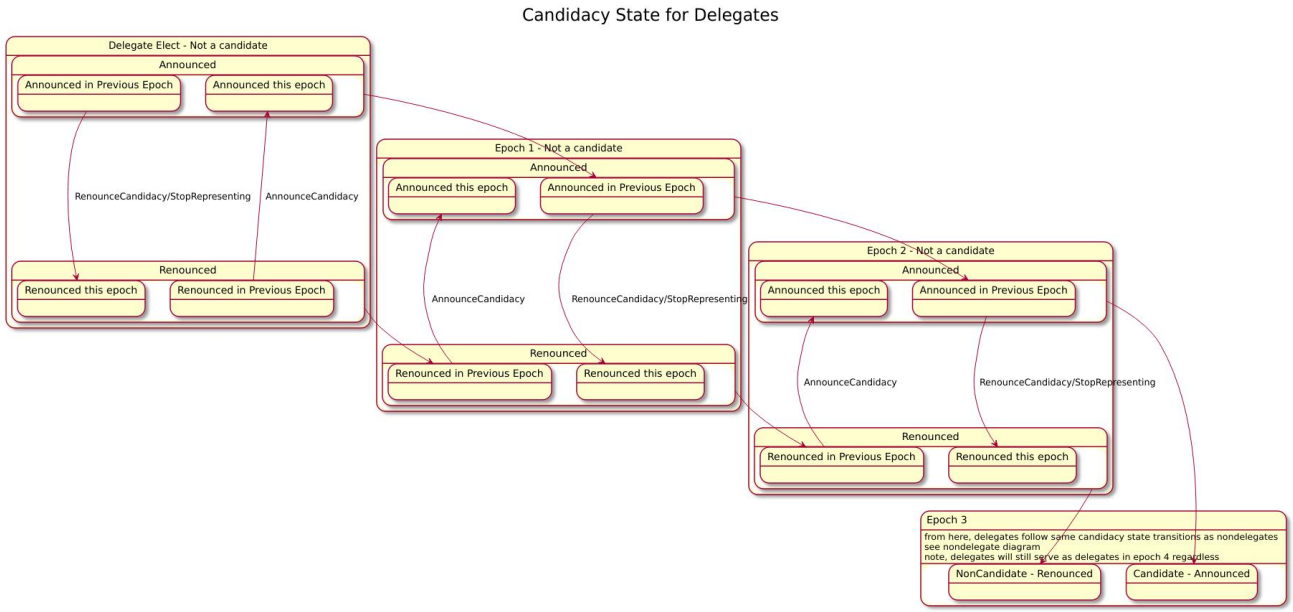
```
votes_i = votes delegate i received when elected, or 1 if delegate received 0 votes
total_votes = sum of all votes_i
votes_remaining = total_votes
cap = total_votes * 1/8
for each votes_i in sorted order:
    if votes_i > cap:
        excess = votes_i - cap
        votes_i = cap
        votes_remaining -= votes_i
        for each votes_j not yet processed:
            votes_j_ratio = votes_j / votes_remaining
            votes_j += excess * votes_j_ratio
        votes_remaining += excess
    votes_i.processed = true
```

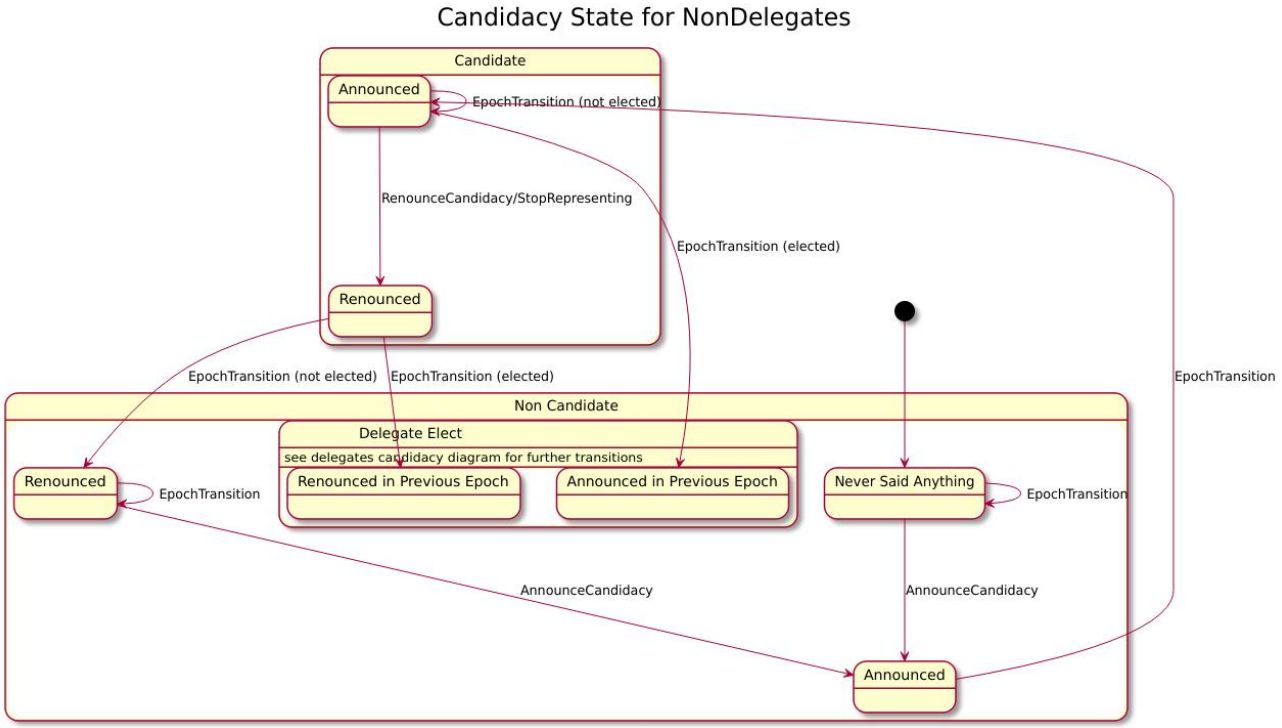
This function ensures no delegate receives more than 1/8 voting power, every delegate has greater than 0 voting power, and, if two delegates have less than 1/8 voting power after redistribution, their voting power relative to each other is the same before and after redistribution. When giving 1 vote as a freebie, that delegate is receiving at most 1/32 of voting power, since every other delegate will also have at least 1 vote as well. To change the cap, simply replace 1/8 with a different cap ratio.

Candidacy State

Below is a state diagram detailing the candidacy state transitions. Initially, an account starts in the `Never Said Anything` state for non-delegates. `AnnounceCandidacy` transitions the account to `Announced`, but the account is not a candidate until after epoch transition. After epoch transition, the candidate remains a candidate through subsequent epochs unless either: a) the candidate is elected or b) the candidate issues `RenounceCandidacy` or `StopRepresenting`, in which case they will be removed from the candidate list the epoch after they issue `RenounceCandidacy` or `StopRepresenting`. If a candidate is elected, they will transition to the `Delegate Elect` state in the Candidacy State for Delegates diagram, retaining their `Announced` or `Renounced` state. As a delegate or delegate elect, an account may change their candidacy state, but an account is not a candidate that can receive votes until their 3rd epoch as an active delegate. Note, a candidate is a delegate elect in the epoch after they are elected, and doesn't begin serving as a delegate until 2 epochs after they were elected (Epoch 1 in the diagram). Beginning in epoch 3, delegates become active candidates that can receive votes and follow the same candidacy state transitions as non-delegates. If a delegate is in the `Announced` state immediately prior to transitioning to epoch 3, then the delegate is an active candidate in epoch 3.

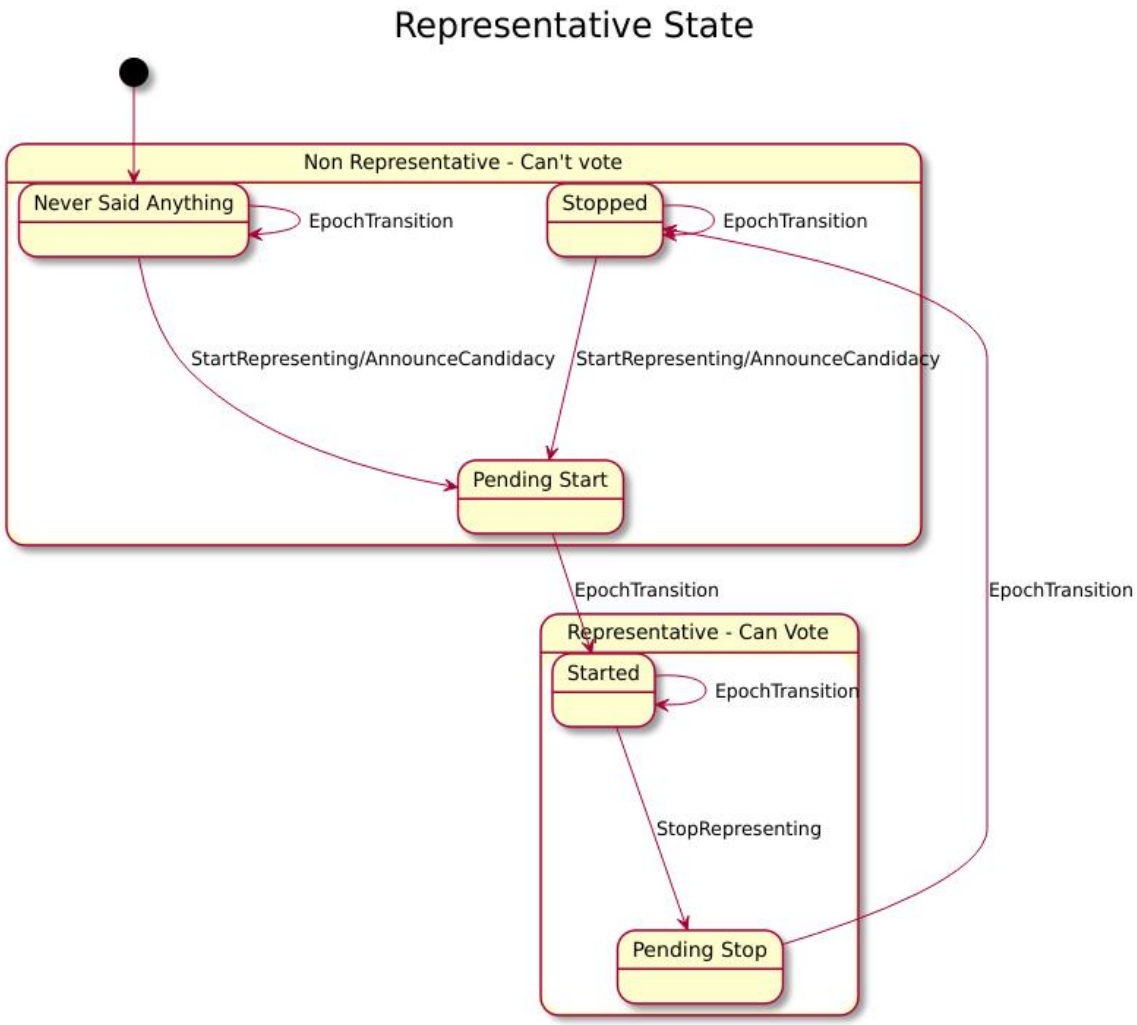
Note, an account can only issue one of `AnnounceCandidacy`, `RenounceCandidacy` or `StopRepresenting` per epoch, which means candidacy state can only change once per epoch.





Representative State

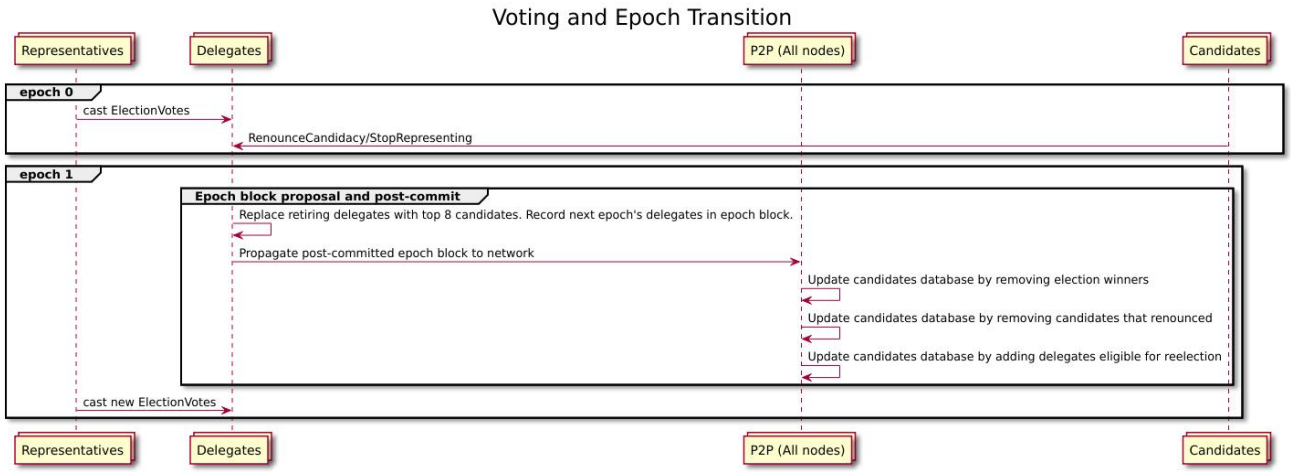
Below is a state diagram detailing the representative state transitions. The representative state determines whether an account can vote in elections. Accounts start in the `Never Said Anything` state of Non Representatives. To become a representative, accounts issue either `StartRepresenting` or `AnnounceCandidacy`, whereas the latter makes the account a rep and a candidate simultaneously. Each of these commands includes how much to stake. Once an account issues either of these commands, the account enters the `Pending Start` state, but is not yet a representative and cannot vote until after epoch transition. Note, an account can only issue one of `StartRepresenting`, `AnnounceCandidacy` or `StopRepresenting` per epoch, which means representative state can only change once per epoch.

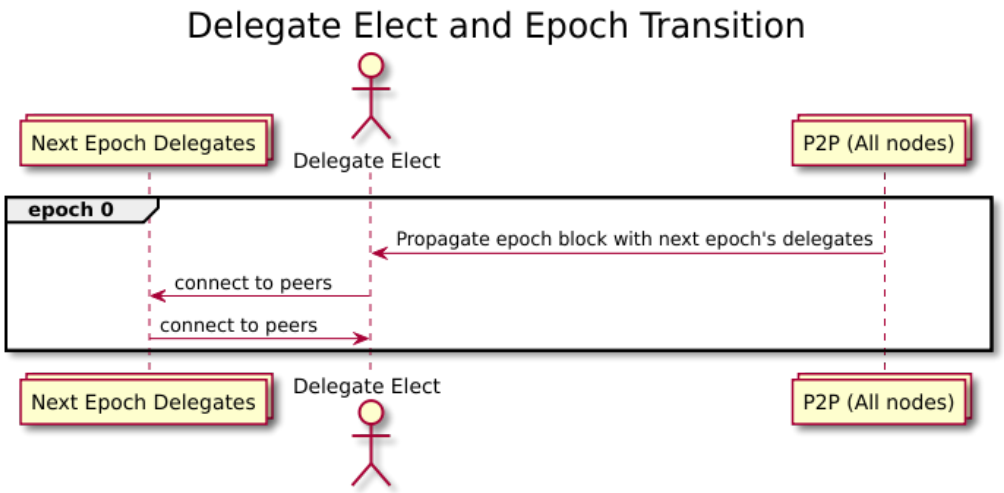


Scenarios and Sequence Diagrams

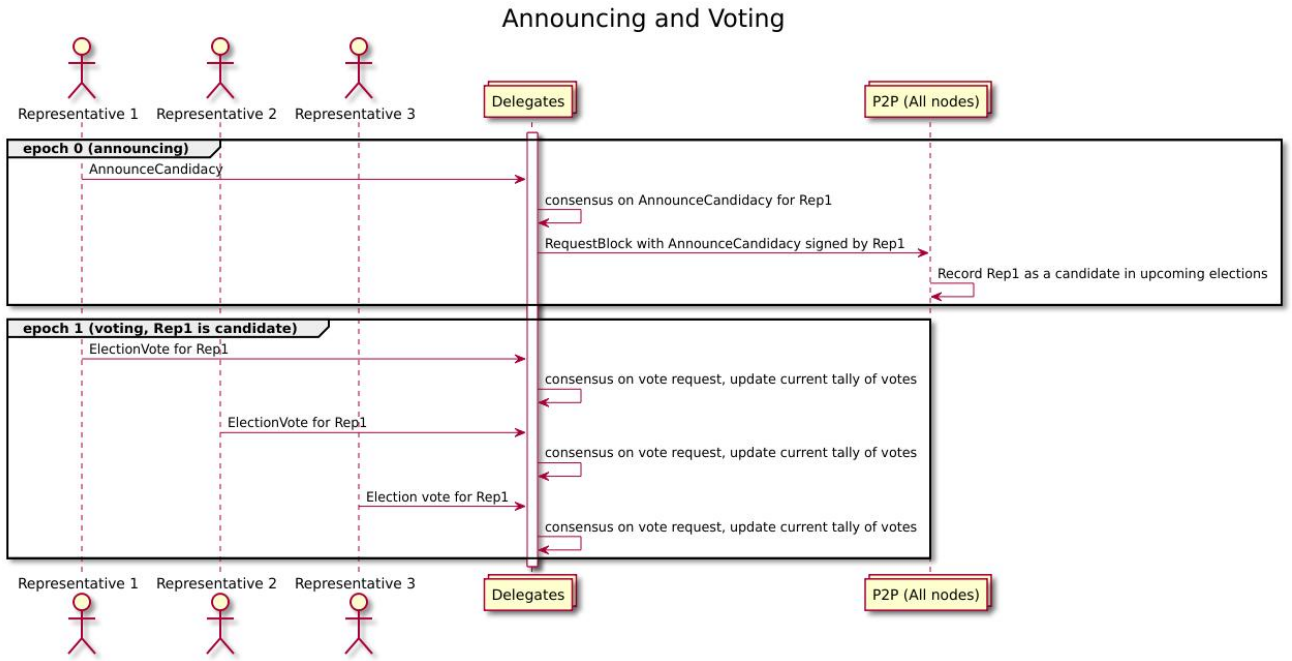
The below 2 diagrams detail election events that occur surrounding epoch block proposal and post-commit, and serve to transition the elections subsystem to the next epoch. During an epoch, the amount of votes received by each candidate is stored in `candidacy_db`, and the top 8 candidates are stored in `leading_candidates_db`. When the epoch ends, the elections dead period begins and voting ceases. When an epoch block is being built or validated, delegates read from `leading_candidates_db` to determine the candidates that won the election. The 8 retiring delegates are replaced with the 8 election winners. The list of delegates for the next epoch, which includes the election winners, is recorded in the epoch block. The epoch block is post-committed and distributed to the network over P2P. All nodes update their candidate databases upon receiving the epoch block, the dead period ends and elections begin for the new epoch. The nodes update the candidate database by removing delegate elects, removing candidates that issued `RenounceCandidacy` or `StopRepresenting`, and adding any delegates eligible for reelection. Delegates are eligible for reelection if they are in their second to last epoch and have not most recently renounced their candidacy, or if they are in their last epoch and are not delegate elects. See the timeline section and candidacy state section for more details on when delegates are eligible for reelection.

Candidates that won the election, who are now delegate elects, learn that they won the election when the epoch block is propagated to them, and will connect to their peers (the next epoch's delegates) at the end of the epoch.





The below diagram shows a representative announcing candidacy. Representative 1 submits `AnnounceCandidacy` in epoch 0, which all nodes hear about through P2P. The request makes Representative 1 a candidate for any subsequent epochs. In epoch 1, multiple representatives vote for Representative 1. A current tally of the amount of weighted votes that Representative 1 has received so far, recorded in `candidacy_db` and possibly `leading_candidates_db`, is updated after each `ElectionVote` is post-committed. At the beginning of Epoch 2, the election results are calculated. See the above sequence diagrams pertaining to epoch block persistence and transitioning the elections subsystem to the next epoch.



Staking

Overview

The staking system provides a way for accounts to stake funds. To stake funds means to make those funds unspendable in exchange for rewards. Staked funds have a concept of a target, which is an account that those funds are staked to. Accounts may reduce the amount of funds staked, however those funds must first enter a thawing state for 1 thawing period (defaults to 3 weeks) before the funds are spendable again. Thawing funds also have a concept of target, which is the account those funds were staked to prior to entering the thawing state.

There are two types of staking (and two types of staked funds). Staked funds or funds staked refers to both lockied proxied and self stake. Any request that changes the amount of staked funds is considered a staking request.

- Self Stake - this is when an account stakes funds to itself. This type of staking is only allowed for representatives (and delegates).
- Locked Proxy - this is when an account stakes funds to a representative. This type of staking is only allowed by non-representative accounts.

Staking primarily impacts voting power and the distribution of rewards. A representative's voting power depends on self stake and all stake lock proxied to them. A representative's voting power also depends on the available balance (excludes staked or thawing funds) of each account that is proxied to them. This amount is called the unlocked proxied amount, and contributes to the voting power of a representative, albeit a dilution factor. See Voting Power section for more details.

- Unlocked Proxy - the available balance of a non-representative account, multiplied by dilution factor

Accounts may increase or decrease the amount of funds staked. Accounts may also change the target of their currently staked funds. When accounts change the target of staked funds, a Liability is created, where accounts are Liable for the actions of the old target for 1 liability period (equal to thawing period). See Liabilities section for more details.

The software may use ThawingFunds to satisfy a staking request; when increasing amount staked, the software tries to draw the funds from existing thawing funds first before using an account's available balance (funds that are not staked or thawing). If an account is attempting to change their representative, the software will attempt to use existing staked funds (with a different target) to satisfy the request, followed by existing thawing funds and then an account's available balance. See Liabilities and StakingManager section for more details.

Databases

Staking introduces 7 new databases:

- `voting_power_db` Information about a representatives voting power. Maps rep address -> voting_power_snapshot
- `voting_power_fallback_db` Information about a representatives voting power. Used only in a certain race condition. See Implementation section. Maps rep address -> voting_power_fallback
- `master_liabilities_db` All liabilities in the system. Maps liability hash -> liability
- `rep_liabilities_db` Maps rep -> all hashes of liabilities for which rep is a target
- `secondary_liabilities_db` maps account -> all hashes of secondary liabilities for which account is a origin
- `staking_db` Information about currently staked to self or locked proxied funds. Maps account -> staked funds
- `thawing_db` Information about currently thawing funds. Maps account -> thawing funds. Uses duplicate keys. Records for an account are ordered in reverse order of expiration epoch (Furthest from expiring are first)
- `epoch_rewards_db` Information about voting rewards on a per epoch basis. Maps rep + epoch number -> epoch rewards info
- `global_epoch_rewards_db` Aggregate information about voting rewards on a per epoch basis. Maps epoch number -> global epoch rewards info

Information about an accounts staked funds, thawing funds and liabilities are stored separate from the account_db. This way, deserializing an account does not require deserialization of any info about staking.

Note, thawing_db supports duplicate keys; an account can have up to 32 different thawing funds. However, staking_db does not support duplicate keys; an account can only have one set of staked funds, though accounts can increase or decrease the amount staked.

Note, liabilities are stored only in master_liability_db. rep_liability_db and secondary_liability_db only store hashes of liabilities, which can be used to get the liability from master_liability_db. See liabilities section for more details.

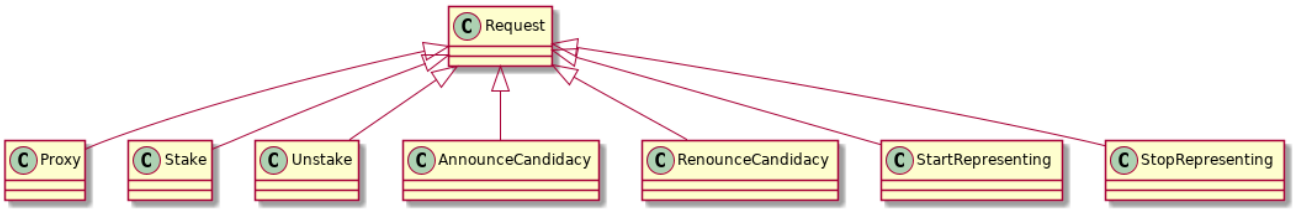
Staking adds fields available_balance and staking_subchain_head to the account_db.

See database format for more details.

Interface

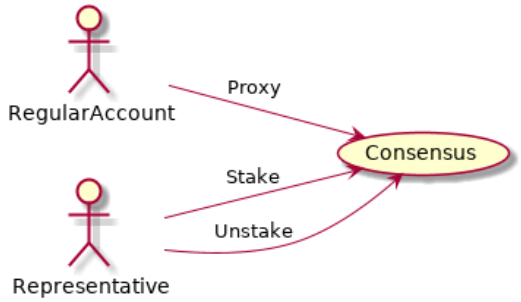
The interface to the staking system is through issuing the proper Request.

Stake is used to set the amount of funds staked to yourself. Proxy is used to choose a representative, and set the amount of funds locked proxied to that rep. StartRepresenting, StopRepresenting, AnnounceCandidacy and RenounceCandidacy each have fields to set the amount of funds staked to yourself. Unstake is used to move all currently staked to self funds to the thawing state.

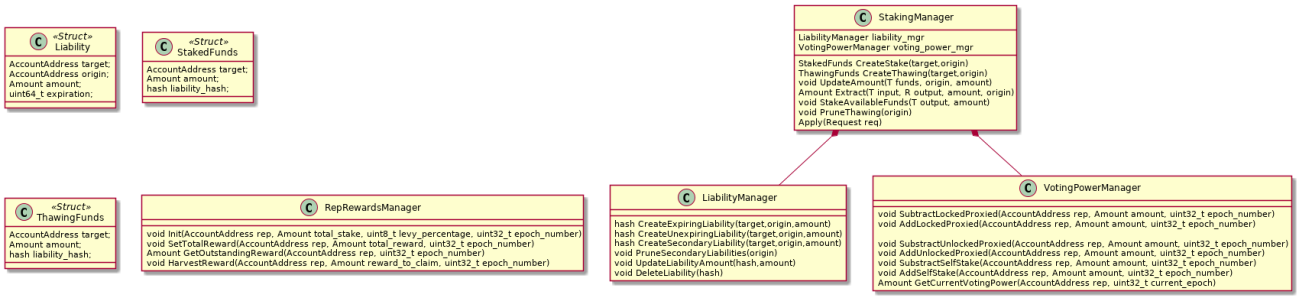


UseCase

Stake and Unstake are only used by representatives. Proxy is only used by non-representatives.



Classes



StakingManager is an interface to staking_db and thawing_db. StakingManager also manages the available_balance field of account_info, and calls LiabilityManager and VotingPowerManager as needed.

LiabilityManager is an interface to the various liability db's, and updates any `Liability` attached to `StakedFunds` or `ThawingFunds`.

VotingPowerManager is an interface to voting_power_db, and is used to get a representatives voting power, as well as update a reps voting power based on post-committed requests. See Voting Power section for more details.

RepRewardsManager is an interface to epoch_rewards_db. See Rewards section for more details.

StakedFunds is a struct that represents any staked to self funds or locked proxied funds.

StakedFunds and ThawingFunds each contain a hash of the associated Liability.

Thawing Period and Liability Period

The thawing period is equal to the liability period, and defaults to 3 weeks. The software represents 3 weeks as 42 epochs. ThawingFunds and Liability each have an expiration_epoch field. As soon as the current epoch is greater than or equal to the expiration epoch, the funds are spendable, and the ThawingFunds and associated Liability 's can be removed from their respective databases. Prior to the expiration, thawing funds are not spendable.

An expiration of 0 signifies a Liability or ThawingFunds that never expire. This is for currently staked funds, or when a delegate creates ThawingFunds while in office. During epoch transition, the software checks if current or incoming delegates created any ThawingFunds in the previous epoch, and marks those Thawing Funds as frozen.

Voting Power

The available balance of an account is the account balance minus the amount of funds staked to self, locked proxied or thawing; it is the spendable balance of an account.

```
Self_Stake = Amount R has staked to self
Locked_Proxy = Sum of all locked proxied funds with R as target
Dilution_Factor = 0.25
```

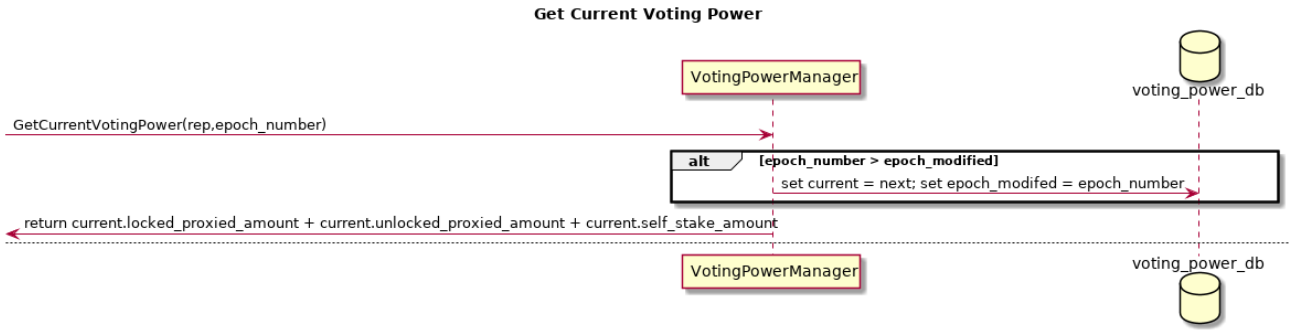
```
Unlocked_Proxy = Sum of Available Balance of all accounts proxying to R
Voting_Power = Self_Stake + Locked_Proxy + Unlocked_Proxy * Dilution_Factor
```

Voting power is stored in voting_power_db and is managed by VotingPowerManager.

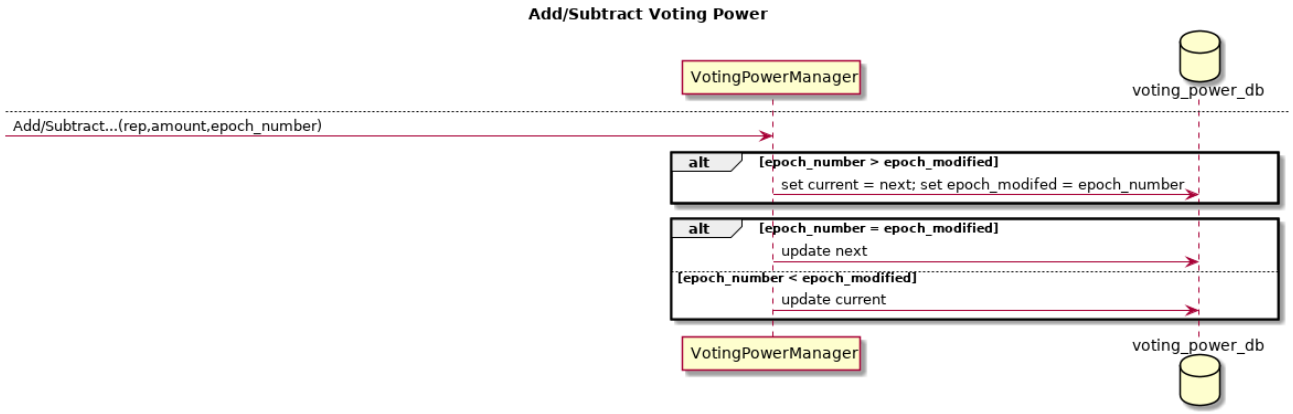
A representative's voting power is fixed for the entire epoch, and only reflects requests post-committed in a prior epoch. Voting_power_db stores voting power for the current epoch (`current`), voting power for the next epoch (`next`) and the epoch in which the data was most recently modified (`epoch_modified`) Throughout the course of the epoch, when any request is post-committed that affects voting power, the change is applied to `next` , and `epoch_modified` is set to the current epoch. Whenever a reader or writer notices that `epoch_modified` is older than the current epoch, set `current` equal to next, and increment `epoch_modified` . In this manner, `current` is only set once per epoch.

Sometimes requests in a request block from epoch i are not applied to the database until epoch i + 1. This can happen when requests are submitted very close to the epoch boundary, or while bootstrapping. In this instance, if the epoch number of the request is less than `epoch_modified` , the changes are applied to `current` instead of `next` . This should only ever happen during the elections dead period or while bootstrapping; once the last microblock is received, nodes will bootstrap any requests that may have been missed. A node should not process `ElectionVote` requests until they have processed the last microblock and bootstrapped if necessary; otherwise, that node may have an incorrect view of voting power. See Execution Concept for more information about elections dead period.

Note, when a representative issues StopRepresenting, the accounts proxied to that rep remain proxied to that rep, until the proxying accounts issue another Proxy request. When an account A issues StartRepresenting, to calculate voting power of A, the software must know the accounts that are already proxied to A. To achieve this, records are pruned from voting_power_db only when voting power goes to 0 (no accounts proxying to this rep anymore). When an account issues StartRepresenting, their voting power is read from voting_power_db.



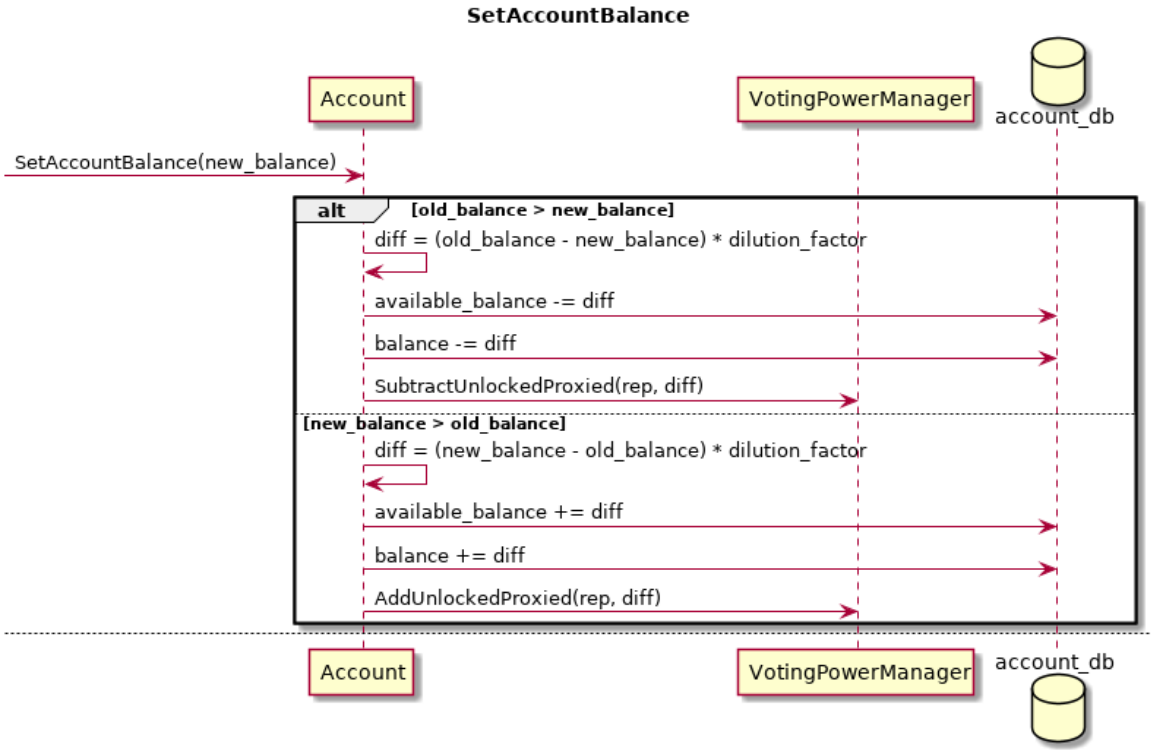
Below is the sequence diagram for any of the Add or Subtract methods of VotingPowerManager.

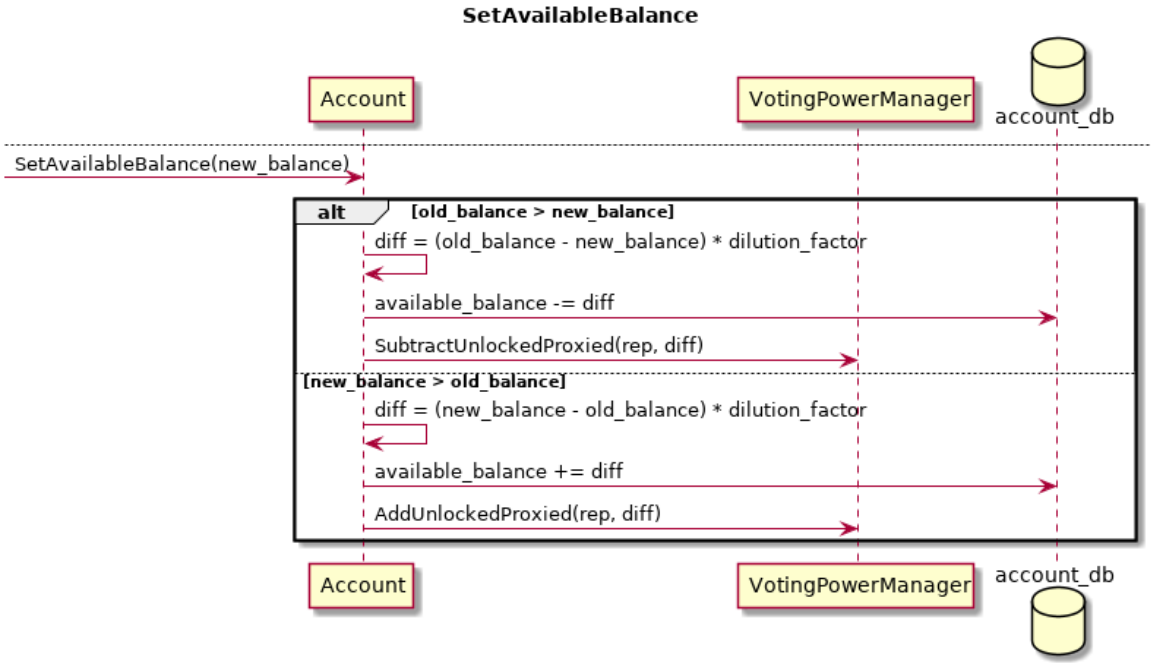


Account Balance

Any change to an accounts balance also changes the voting power of that account's representative. To enforce this invariant, the balance field of Account, and the available_balance of account_info, will both be set to private. Access to these members will be through getter and setter methods, where the setter methods calls the appropriate VotingPowerManager method everytime the balance is changed, while also ensuring each change is applied to balance as well as available_balance. The getter method for available_balance will check for and prune any expired ThawingFunds before returning an up to date available balance. ThawingFunds are ordered in the db by expiration time (earliest expiring first), so checking for ThawingFunds does not require looping over all ThawingFunds for an account; if no ThawingFunds are expired, the getter will simply check the first element and return.

The getter and setter will not call VotingPowerManager for token accounts.





Liabilities

A liability is a historical record of staked funds (both staked to self and locked proxied). Liabilities are needed for slashing; if a rep commits a slashable offense, every account lock proxied to that rep is slashed, as well as the rep itself. Liabilities are how the software determines which accounts to slash in the event of a slashable offense. Liabilities also keep track of the instant reproxy of locked proxied funds. These liabilities are known as secondary liabilities. Non-representative accounts are able to lock proxy an amount to a given rep, and then re-proxy those same funds to a different rep. Liabilities are used to ensure that when an account instantly reproxys, the account is still slashable for actions of the old rep for some time (1 liability period, which is 42 epochs).

Instant reproxy satisfies certain safety requirements. Locked proxied funds only increase the voting power of a single rep at a time; never do two reps derive voting power from the same locked proxy. While the software allows instant reproxy, the reproxy is recorded (via secondary liabilities) such that the proxying account is still slashable for the previous rep's actions for some time (1 liability period).

All secondary liabilities associated with an account must have the same target. This stops an account from lock proxying to A, then lock proxying to B, then lock proxying to C, and so on. Accounts can move their locked proxied funds from A to B, but then can't move them from B to C without waiting 1 liability period. This restriction allows for an instant reproxy of funds once every 42 epochs. If an account desires to change their proxy again prior to 42 epochs, that account must use available funds instead of reproxying existing lock proxied funds. This restriction ensures that a single set of lock proxied funds are only liable for the actions of at most two reps at a time.

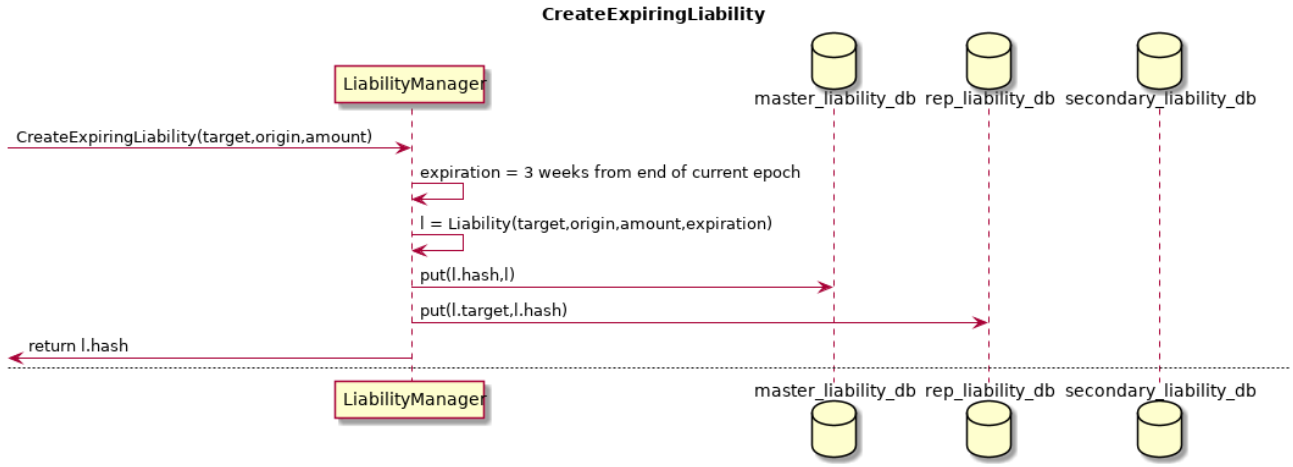
All StakedFunds S, owned by account A, have an associated liability L, where L.target = S.target, L.amount = S.amount, L.origin = A, and L.expiration = 0. An expiration of 0 represents an unset expiration. All ThawingFunds T, owned by account A, also have an associated liability L, where L.target = T.target, L.amount = T.amount, L.origin = A, and L.expiration = T.expiration.

To satisfy Proxy or Stake requests, the software may use existing StakedFunds (if Proxying to a new rep) or existing ThawingFunds to satisfy the request. If the software uses X funds from existing StakedFunds or ThawingFunds (call these funds E) to satisfy Proxy request R from account A, and E.target != R.representative, the software creates a liability L, known as a secondary liability, where L.target = E.target, L.amount = X, and L.origin = A. If E is ThawingFunds, L.expiration = E.expiration, and if E is StakedFunds, L.expiration = 1 liability period from present. Secondary liablities are a record of when an account reproxies their locked proxied funds to a new representative.

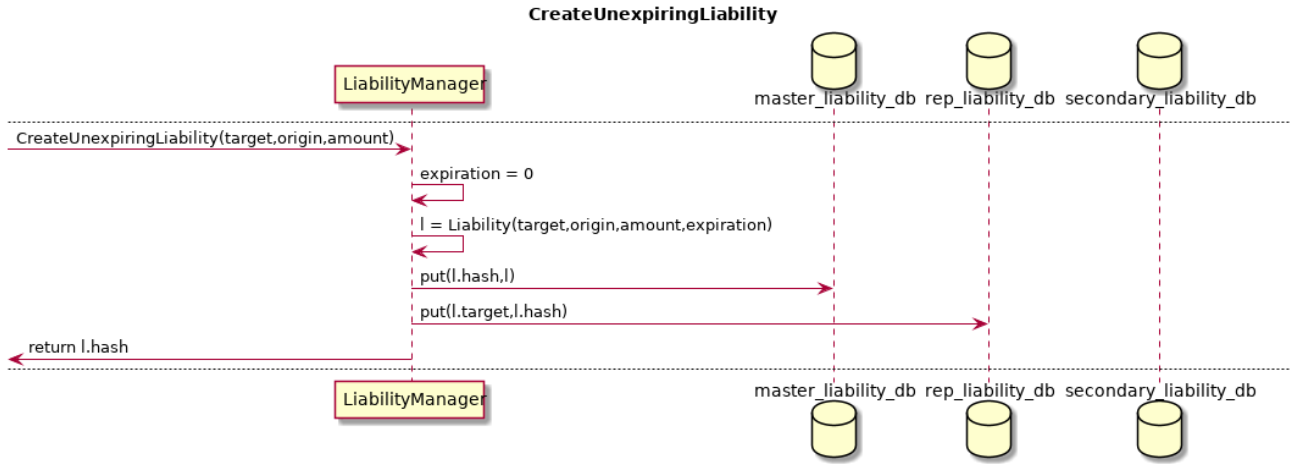
Every Liability is stored in master_liability_db, where the key is a 32 byte hash of origin, target and expiration. This allows the software to adjust the amount field of a Liability without changing the key. For every Liability with target R, the hash is also stored in rep_liability_db with key R. This db allows the software to access all liabilities for which a given rep is a target. For every Liability with origin A, if the Liability is a secondary liability, the hash is stored in secondary_liability_db with key A; otherwise, the hash is stored with the associated StakedFunds or ThawingFunds.

Whenever StakedFunds or ThawingFunds are deleted, the associated Liability is deleted as well, and the hash is removed from rep_liability_db. Expired secondary liabilities are pruned when applying a Proxy request.

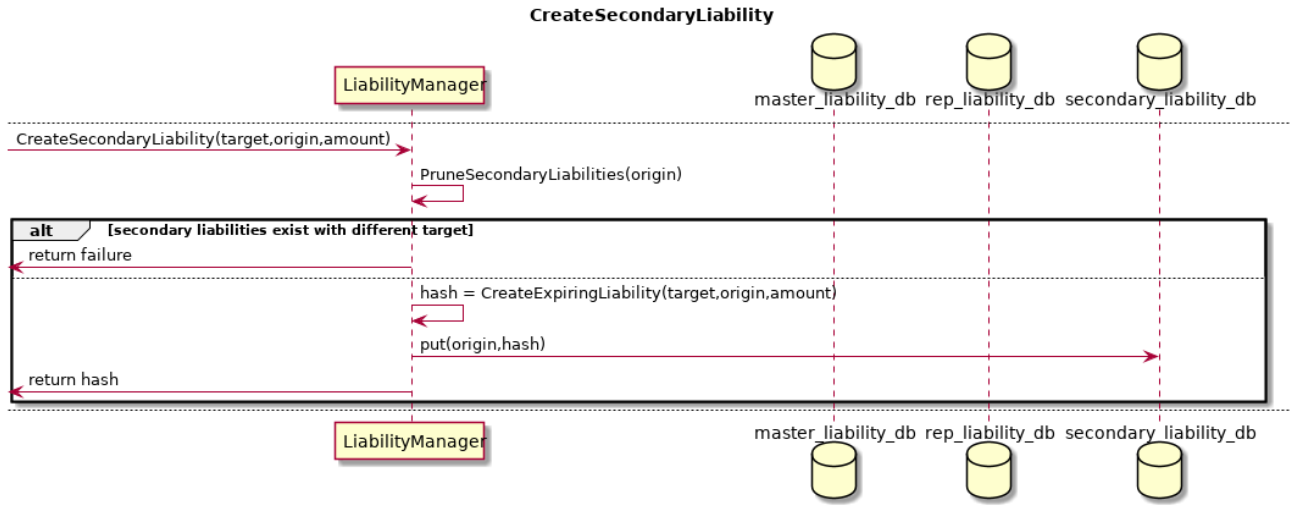
CreateExpiringLiability creates a liability in master_liability_db, adds a hash to rep_liability_db, and returns the hash. The liability has an expiration of 3 weeks from the end of the epoch. This is called when a secondary liability is created, or when thawing funds are created.



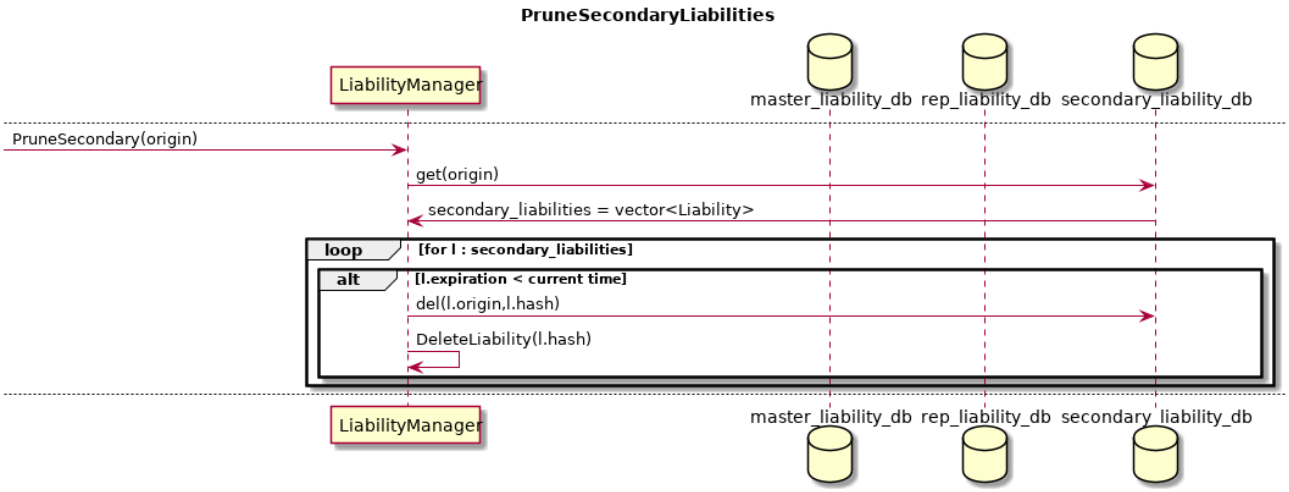
CreateUnexpiringLiability does the same as above, except sets the expiration to 0. This is called when staked funds are created (self stake or lock proxy)



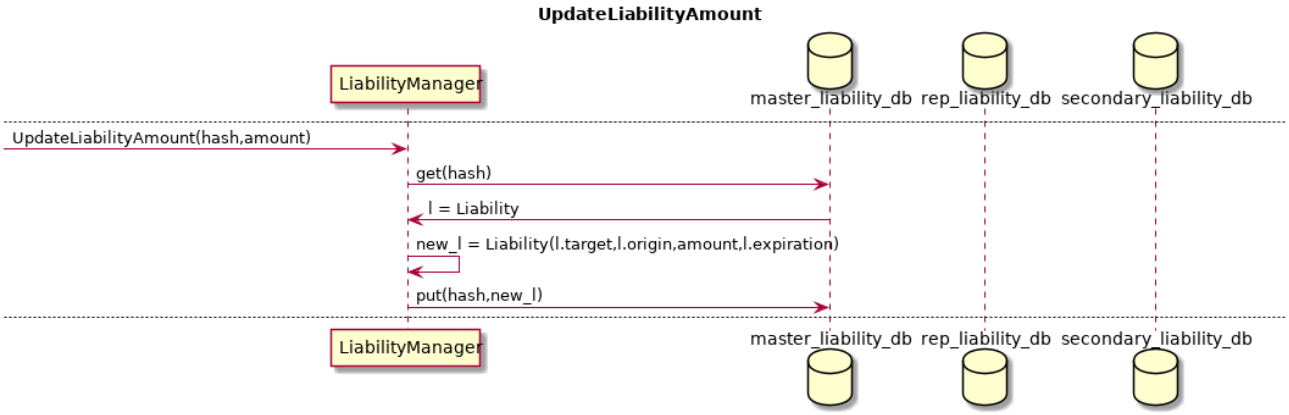
CreateSecondaryLiability calls CreateExpiringLiability, and also adds the hash to secondary_liability_db. The method will fail if an account already has unexpired secondary liabilities for a different target. This is called during the instant reproxy of locked proxied funds.



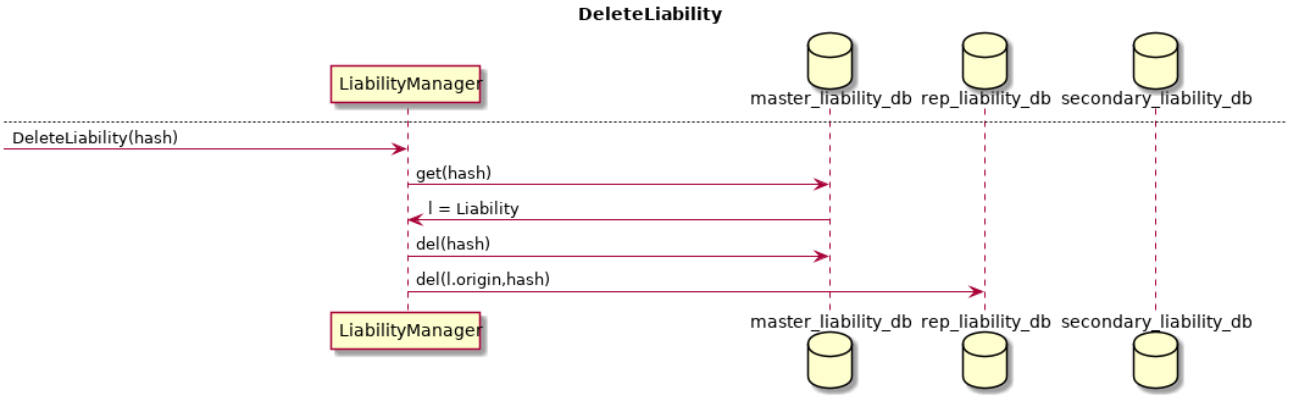
PruneSecondaryLiabilities deletes any expired secondary liabilities from all liability db's. This is called whenever the software creates new liabilities for a given account.



UpdateLiabilityAmount changes the amount field of the liability in master_liability_db. The other liability db's are not changed. This is called when an account changes the amount of funds staked (self staked or locked proxied).



DeleteLiability deletes the liability from master_liability_db and also deletes the hash from rep_liability_db. Note, the hash is not deleted from secondary_liability_db. Hashes are only deleted from secondary_liability_db via the PruneSecondaryLiabilities method. This is called when a liability expires.



Staking Manager

The StakingManager handles any requests that pertain to staking, and updates the staking_db, thawing_db and account_db, as well as makes calls to LiabilityManager and VotingPowerManager. StakingManager is the main entry point to the staking system.

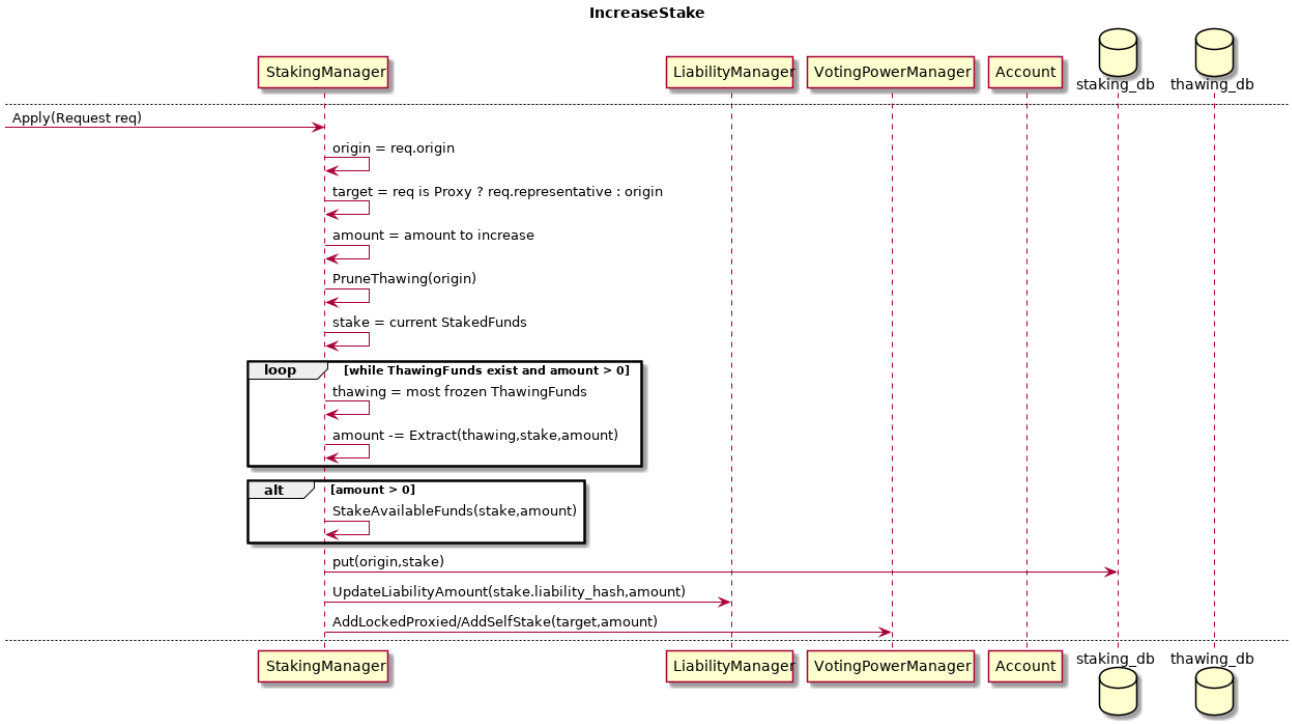
Note, an account can only have one StakedFunds, but can have up to 32 different ThawingFunds at a time.

When an account submits a Proxy request with an amount to Lock_Proxy, or a Stake request, to create the new `StakedFunds` the software first attempts to use existing `StakedFunds` (if those funds have a different target), then `ThawingFunds` in decreasing order of expiration time, and lastly falls back to using an accounts available balance. StakingManager does this by calling the Extract(T input,R output,amount,origin) method, which extracts the specified amount of funds from input and puts those funds into output. Note, input and output may be StakedFunds or ThawingFunds. See the sequence diagrams for more details.

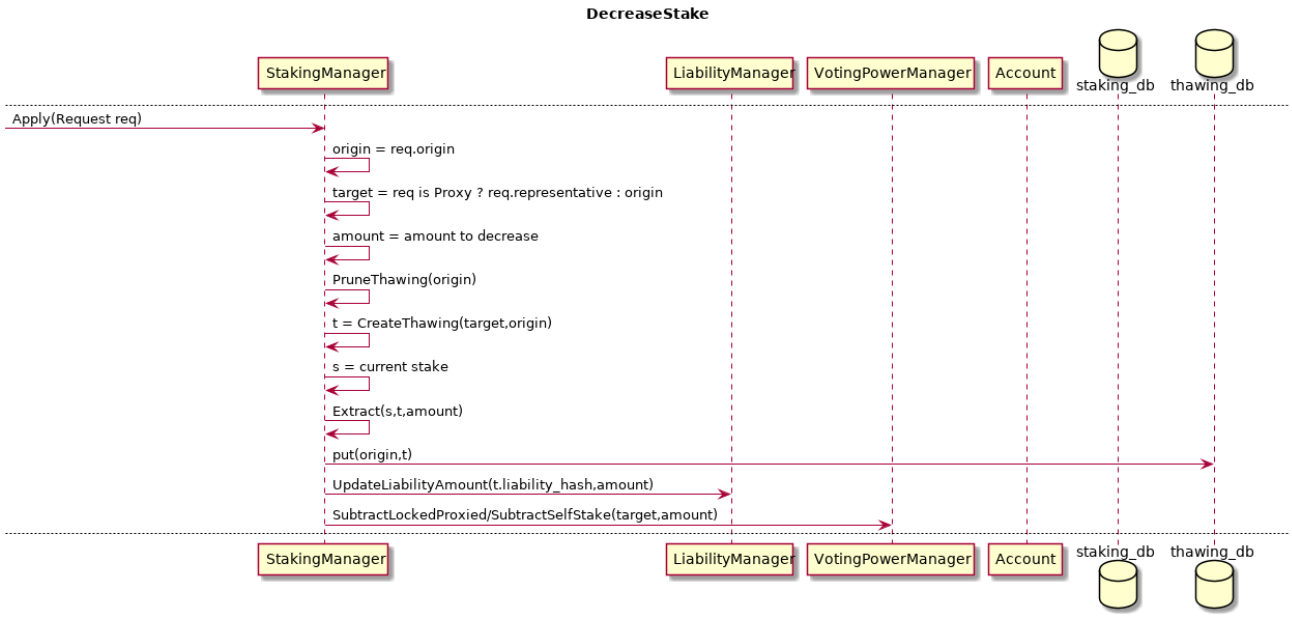
Any Request that involves staking falls under one of 3 cases:

- Increase amount of funds staked. Target of staked funds doesn't change
- Decrease amount of funds staked. Target of staked funds doesn't change
- Change the target of funds staked. This also handles the case where a sender currently has no funds staked, but the request specifies funds to stake.

The below diagrams detail each of these 3 cases. In these diagrams, helper methods of StakingManager are called. These helper methods have corresponding sequence diagrams further down in this section. In each case, StakingManager calls VotingPowerManager to update the voting power of the affected representative. To increase funds staked, StakingManager calls extract on any thawing funds, in decreasing order of expiration time, and then calls StakeAvailableFunds if neccessary.

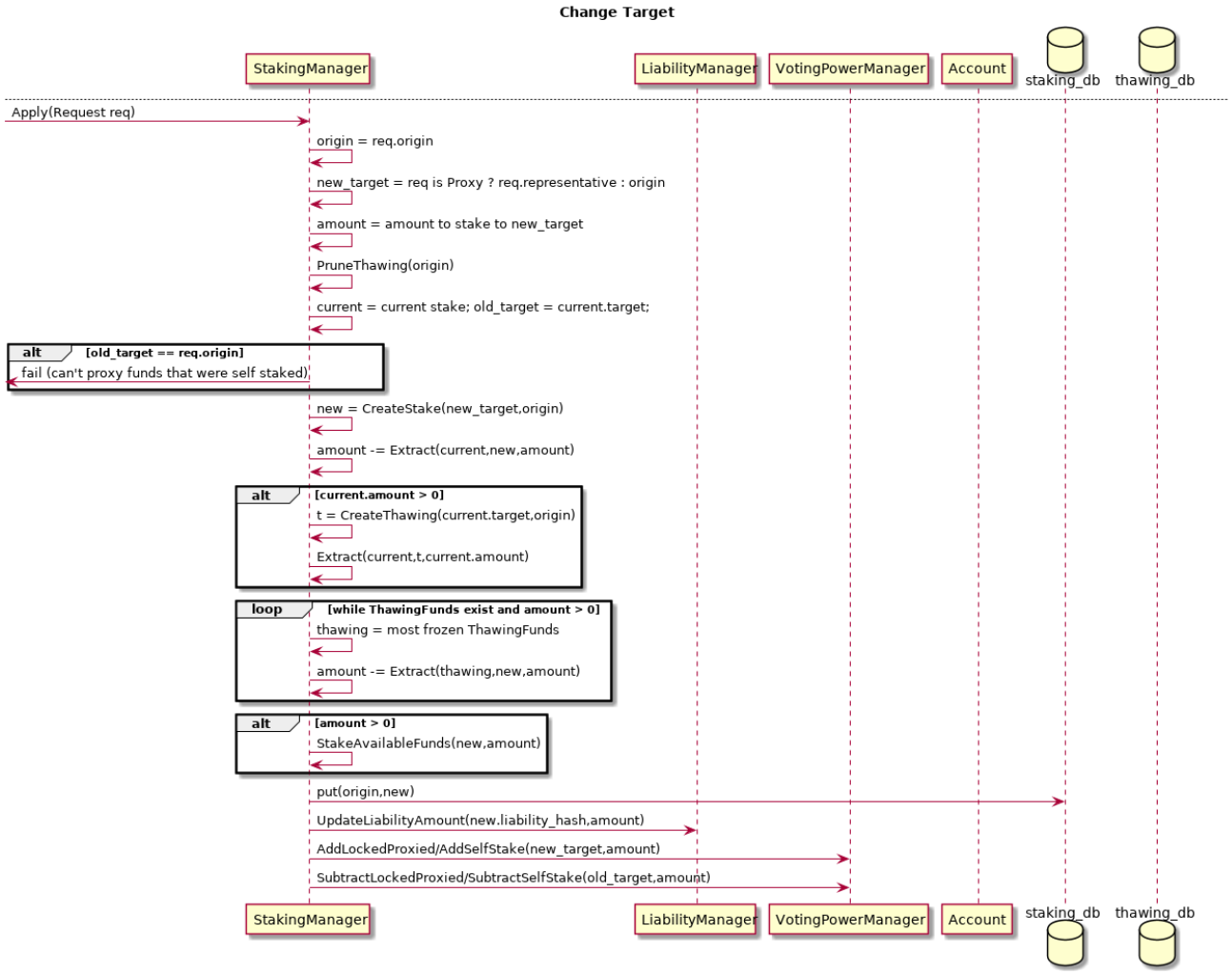


To decrease funds staked, StakingManager creates new thawing funds, then extracts the specified amount of funds from currently staked funds into the thawing funds.



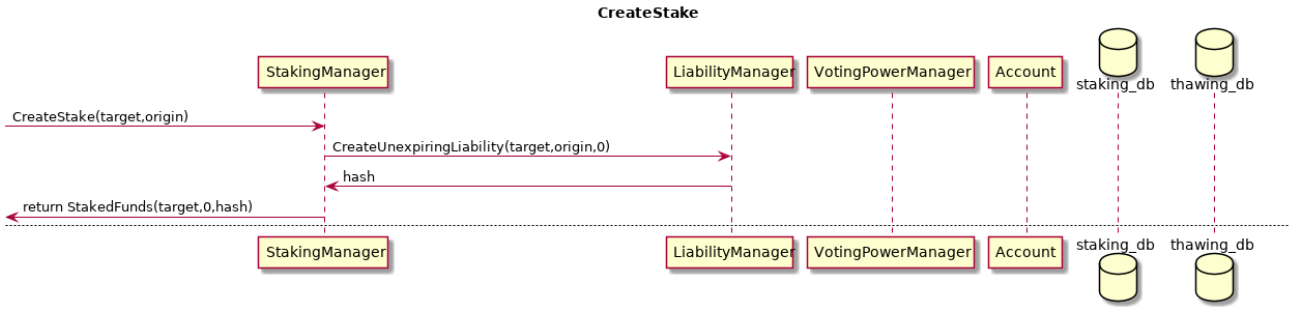
To change the target of staked funds, StakingManager creates new staked funds with the requested target, and then extracts the existing staked funds funds into the new staked funds. If there are funds remaining in the old staked funds, the old funds begin thawing. If the software needs more funds after extracting from the old staked funds, thawing funds and then available funds are used in the same manner as in the Increase Stake case. Note, in this case, the voting power of the old rep and the

new rep must be updated.

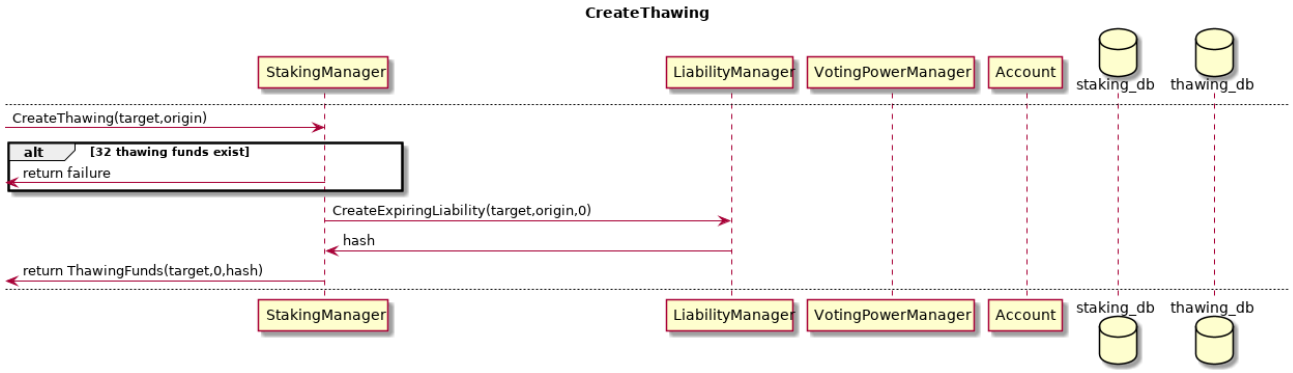


The below figures show the sequence diagrams for the various methods of StakingManager.

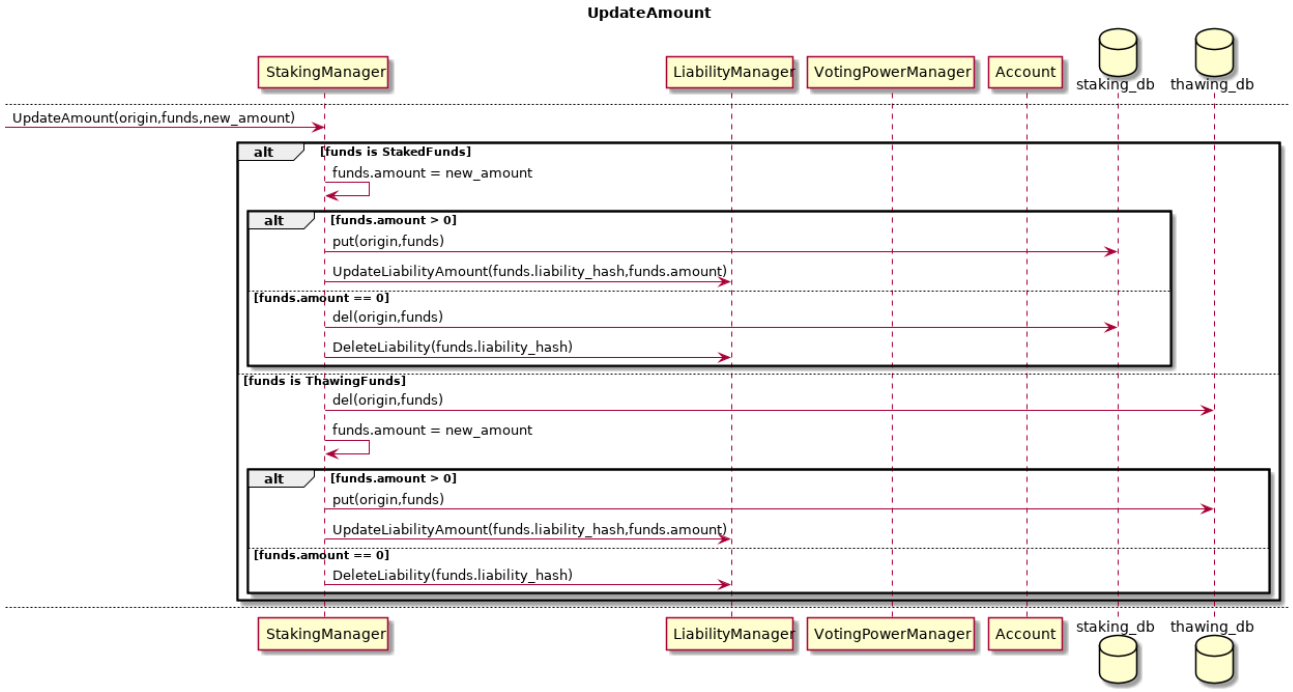
Create a staked funds object, with an amount of 0. Note, this method does not store the object in the db.



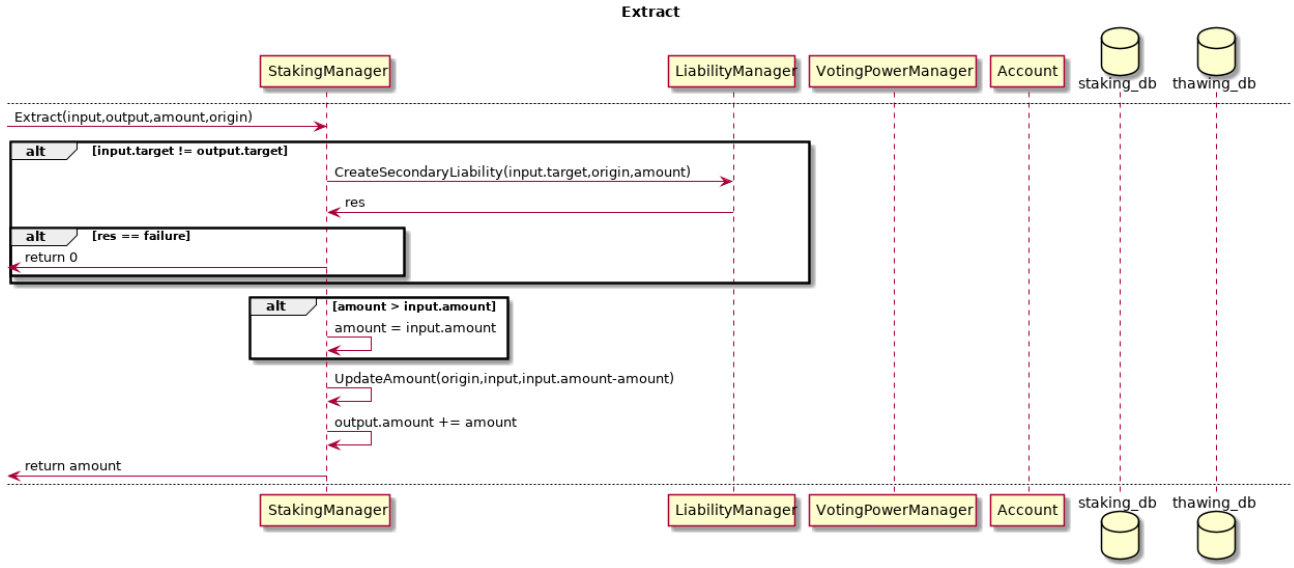
Create thawing funds object, with an amount of 0. Note, this method does not store the object in the db.



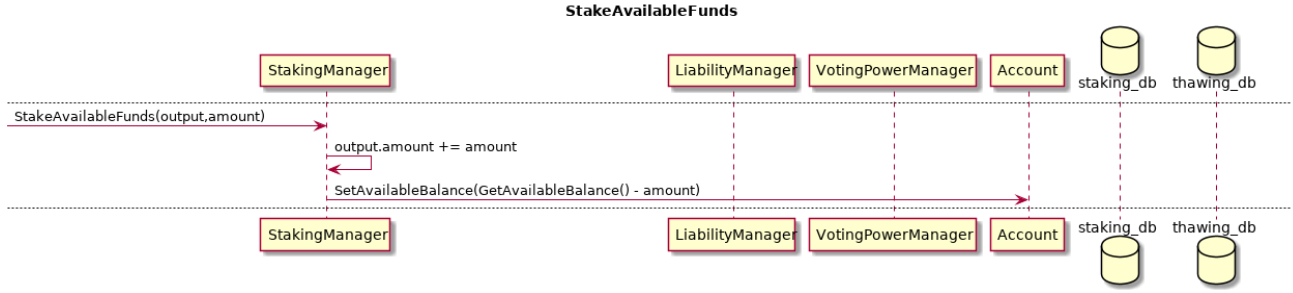
UpdateAmount takes in thawing funds or staked funds as a parameter, and updates the amount of funds staked or thawing. This method also updates any associated liabilities.



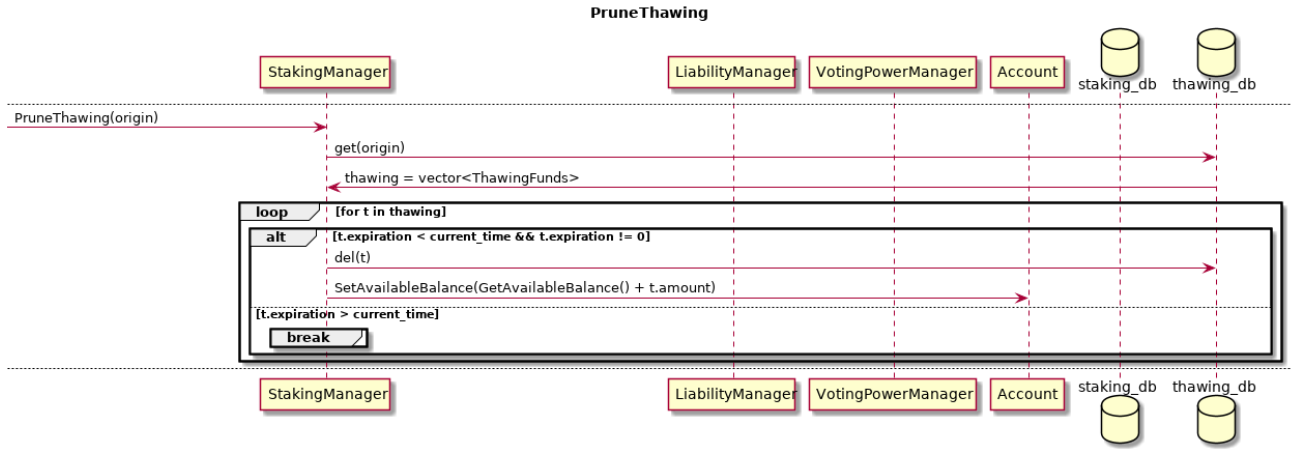
Extract moves the specified amount of funds from input to output, updates the associated liabilities and returns the amount of funds extracted. A secondary liability is created if necessary. If input is left with no funds after extraction, the input (and associated liabilities) are deleted. Note, the input funds are updated in the db, but the output funds are not. This is because to satisfy a staking request, the software may need to use many extraction inputs. Extract is called in many places within StakingManager. Extract is called whenever an account changes the amount of funds staked (staked to self or lock proxied), as well as anytime an account performs an instant reproxy.



StakeAvailableFunds uses available funds, rather than staked or thawing funds, to create or increase the amount of staked funds. The available_balance of an account is updated, which also updates the voting power of the affected rep.



PruneThawing deletes any thawing funds that have expired. Note, thawing funds are ordered by expiration date, so once an unexpired ThawingFund is found, this method can return.



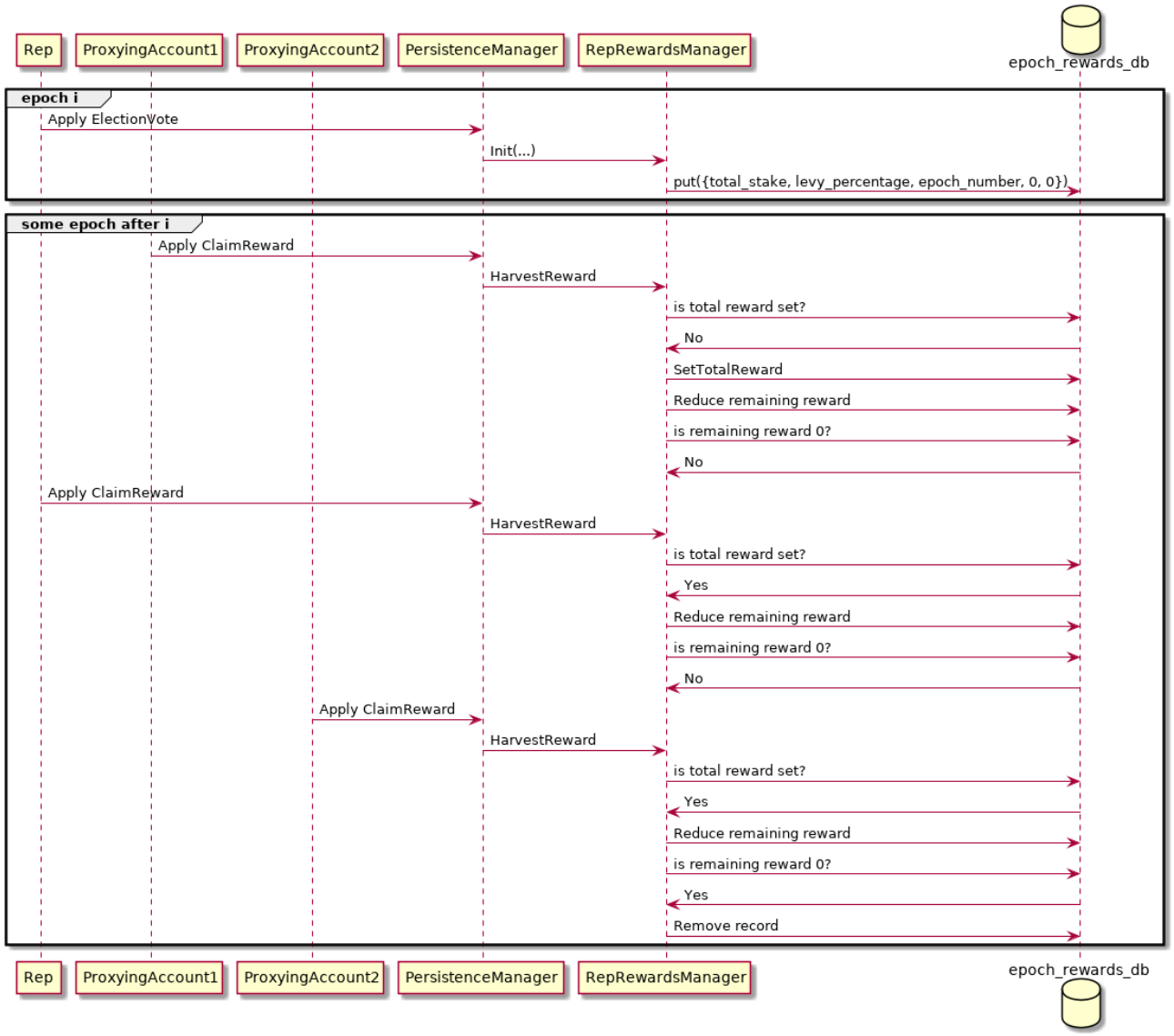
Representative Rewards

To process reward claiming for a proxying account A, the software must know that account's rep R, R's levy_percentage, whether R voted, the amount locked proxied to R by A, and the total amount locked proxied to R by all accounts. This info is on a per epoch basis, including R; an account may have a different rep each epoch. The identity of R and the amount locked proxied to R by A per epoch can be determined from A's staking subchain (see staking subchain section). The rest of the information is recorded in epoch_rewards_db. Epoch_rewards_db is keyed by concatenating R's address and the epoch number. The values of the db contain R's levy_percentage, the total amount locked proxied to R by all acccounts, the initial total reward pool for R (per epoch) and the remaining reward.

When a rep R votes in epoch i, the software calls Init(...) and records the rep's total_stake (self stake and locked proxied), levy_percentage for epoch i. Initially, the total reward pool is 0. The first time R, or an account that proxied to R during epoch i, claims a reward, the total reward pool is set. Once the remaining_reward goes to 0 (all accounts claimed their reward from R for epoch i), the record can be deleted from epoch_rewards_db.

If a record does not exist for a given rep and given epoch, that means either the rep did not vote during this epoch, or that all rewards associated with the rep voting have been claimed.

The below sequence diagram shows a rep voting. The rep has two proxying accounts. Once the rep and both proxying accounts claim rewards for epoch i, the record for rep for epoch i is deleted.



Delegates and Candidates

Within consensus, delegates use their stake to accept or reject Requests. However, a delegate's stake only consists of self stake, and ignores any form of proxied stake. A delegates stake is set to their self stake during the epoch the delegate was elected. If a delegate changes the amount staked to self, that change impacts voting power in the next epoch, but does not impact delegate stake until that delegate starts a new term (terms are 4 epochs).

When a delegate reduces their stake, the thawing period for those funds doesn't start until that delegate finishes their term. Whenever an epoch ends, the software checks the thawing funds of any retiring delegates, and sets the expiration accordingly. The same rules also apply to any liabilities associated with those thawing funds.

A delegate's stake within consensus is redistributed according to the same formula that redistributes delegate voting power. See delegate voting power for more details. Note, this operation does not actually move any funds, or alter any of the databases related to staking, but simply simulates the redistribution of stake, for the purposes of accepting/rejecting requests. The redistribution affects the stake field of each Delegate entry of the epoch block.

Staking subchain

All requests that could affect staked funds (staked to self or locked proxied), form a chain per account. Each request contains a field Staking_Subchain_Previous which is the hash of the previous request issued by this account that affects staking. The head of the staking subchain is stored in the account_db. Traversing this chain allows the software to determine the representative of this account and the amount locked proxied at any given time in the past, as well as the amount staked to self. The below requests are all part of the staking subchain:

- Proxy
- Stake
- Unstake
- StartRepresenting
- StopRepresenting
- AnnounceCandidacy
- RenounceCandidacy

See the IDD for more details.

Implementation

There is a certain race condition regarding voting near the epoch boundary.

Problem

Consider an ElectionVote submitted at the very end of epoch i, and a Send submitted at the very beginning of epoch i+1. Assume the origin account of the Send has a rep. Assume this rep is the origin account of the ElectionVote. When the ElectionVote is applied, the software will look up the reps voting power for epoch i in voting_power_db When the Send is applied, the software will transition the rep's voting power to epoch i+1. During this transition, the voting power for this rep for epoch i is overwritten. If the Send is applied before the ElectionVote, the voting power for this rep for epoch i is no longer stored anywhere.

Solution

To mitigate this race condition,there is a special database called voting_power_fallback_db Whenever the software is transitioning voting power of a rep to epoch i+1, the software checks that the rep actually voted in epoch i. If the rep did not vote in epoch i, the reps voting power for epoch i is first stored in voting_power_fallback_db, and then voting power for the rep (stored in voting_power_db) is transitioned to the next epoch. If the rep did vote in epoch i, no data is stored in voting_power_fallback_db. When applying an ElectionVote, the software checks if the epoch number of the ElectionVote is less than the epoch modified field of VotingPowerInfo. If so, the software reads voting power from voting_power_fallback_db. Otherwise, the software reads voting power from voting_power_db