Engineering Design and Incentive Specification for Aion-Unity

Ali Sharif
ali@aion.network

October 17, 2019

"Most magic numbers in blockchain protocols are determined through careful analysis."
— @VitalikButerin, *Twitter*

Abstract

This document is a complete specification for the Aion-Unity upgrade, documenting the design of the staking, delegation and incentive schemes introduced in the release.

Acknowledgements

This specification was produced in collaboration with the entire R&D staff at the Aion Foundation, with major contributions from Yulong Wu, Yao Sun, Dipesh Pradhan, Jeff Disher, Aayush Rajasekaran, Shidokht Hejazi, Sam Pajot-Phipps, Yunfei Zha and Ge Zhong.

Contents

1 Introduction

This document aims to specify all subsystems required for the implementation of the Unity consensus algorithm [WZS19] as an upgrade to the Aion blockchain.

The key words *MUST*, *MUST NOT*, *REQUIRED*, *SHALL*, *SHALL NOT*, *SHOULD*, *SHOULD NOT*, *RECOMMENDED*, *MAY*, and *OPTIONAL* in this document are to be interpreted as described in RFC 2119 [Bra97].

2 System Overview

The Unity consensus algorithm combines both proof of work (PoW) and proof of stake (PoS) [WZS19]. This section of the specification outlines the implementation of the PoS subsystem in isolation, to simplify exposition.

2.1 Fixing Terminology

In the PoS subsystem of Unity [WZS19], any coin holder of a system can run a full node in the blockchain network in order to produce PoS blocks; we refer to these coin holders as *stakers*.

In order to support a staker's participation in PoS consensus, coins must be rendered immobile (i.e. they cannot be freely transferred to other users) via their submission into the staking system; these coins are referred to as ***stake***. As long as these coins are held (on behalf of the owner) in the staking system, they contribute to the total influence of the staker in the PoS consensus (as discussed in the Unity paper [WZS19]).

When a holder casts their own coins as stake, we call this action a ***bond***. Notice that we use the word "stake" strictly as a noun throughout this document and the implementation; this is done to avoid confusion with the action of casting coins as stake (for which we use the verb *to bond*). In adding to bonding coins to themselves (increasing their power as a staker in PoS), coin holders can bond their coins to another staker; this action is referred to as ***delegation***. We further discuss the motivation for delegation in §4.1.

Any coin holder that bonds to another staker (in order to earn a share of PoS block rewards in exchange for increasing the power of said staker in PoS) is referred to as a ***delegator***. Note that a delegator is not a staker, since they don't operate a node and contribute to PoS security.

At any point, a delegator or a staker can choose to withdraw their bonded coins (reducing the PoS power of the staker delegated or bonded to). When a delegator withdraws their delegated coins, they are said to ***undelegate***. When a staker withdraws their bonded coins, they are said to ***unbond***. In either scenario, when the coins in question get unbonded in the staking system, they are subject to a post-lock period ([WZS19] §4.1.1). After the post-lock period has elapsed, their coins exit the staking system and become ***liquid*** (free to transfer to other users).

A delegator is allowed to ***transfer*** their delegation between stakers (subject to stipulations discussed in §3.3).

2.2 Solution Synopsis

The PoS subsystem in Aion-Unity consists of two major components. The first component is the core staking mechanism (implemented as a privileged contract called "Staker Registry"); it manages

the state of all coins that have been staked in the system to secure the PoS subsystem. The second component is the public stake delegation protocol (implemented as a non-privileged contract called "Pool Registry"); it allows coin-holders in the system to delegate their coins to pool operators, who produce blocks on the delegators' behalf and share earned block rewards with the delegators. A tertiary, but critical component of the PoS subsystem is the user-interface that is used by delegators in order to interact with pools.

3 Core Staking Mechanism (Staker Registry)

The security of the PoS subsystem depends on system participants bonding their coins as stake. All logic related to the management of stake life-cycle is implemented in an AVM smart contract called the *Staker Registry* contract. This contract tracks all stake and stakers in the system and is core to the Aion-Unity protocol (i.e. this is a privileged contract, whose state is required by the protocol to resolve PoS consensus). Any user interested in running a staking node to participate in PoS consensus (within Unity) must register their stake in the Staker Registry.

3.1 Stake Sovereignty

The term non-custodial, in the context of PoS, refers to systems where coins never leave the user's account while the user is participating in staking. The design of Unity consensus [WZS19] poses restrictions on a staker's coins (such as immobility and post-active period) to satisfy consensus security requirements, which makes it impossible to implement a truly non-custodial staking scheme. Instead, we come up with a compromise, which retains the sovereignty-over-stake aspect of non-custodial systems (one could call this system "pseudo-non-custodial").

In Unity PoS, the user must send their coins to the Staker Registry in order to bond them as stake. The Staker Registry must guarantee that the state of the coins, while committed to the Staker Registry, cannot be modified by anyone other than the coins' owner (i.e. the coins cannot be unbonded or transferred by anyone other than their owner).

3.2 Protocol (Kernel) Responsibilities

Since the function of the protocol depends on the Staker Registry contract's state, special interactions that must to coded into it's implementation (i.e. "the kernel"). Every time a PoS staker (node operator) produces a block, the kernel must look-up the staker by his signing key (§4.2) to validate the following:

- The claimed time-to-next-block is valid, based on their committed stake in the Staker Registry
- The claimed coinbase (from the block header) is the registered coinbase in the Staker Registry
- The staker is eligible to produce a block at the proposed block height

The kernel needs to call the method:

$$getEffectiveStake(signingAddress, coinbaseAddress),$$

which returns the effective stake, after taking the staker's self-bonded coins into consideration (e.g. if the staker has violated the self-bond requirement, then this function should return 0).

3.3 Coin Life-cycle in Staker Registry

Figure 1 outlines the *states* that a coin be in, as it makes it's way through the bonding life-cycle in the Staker Registry:

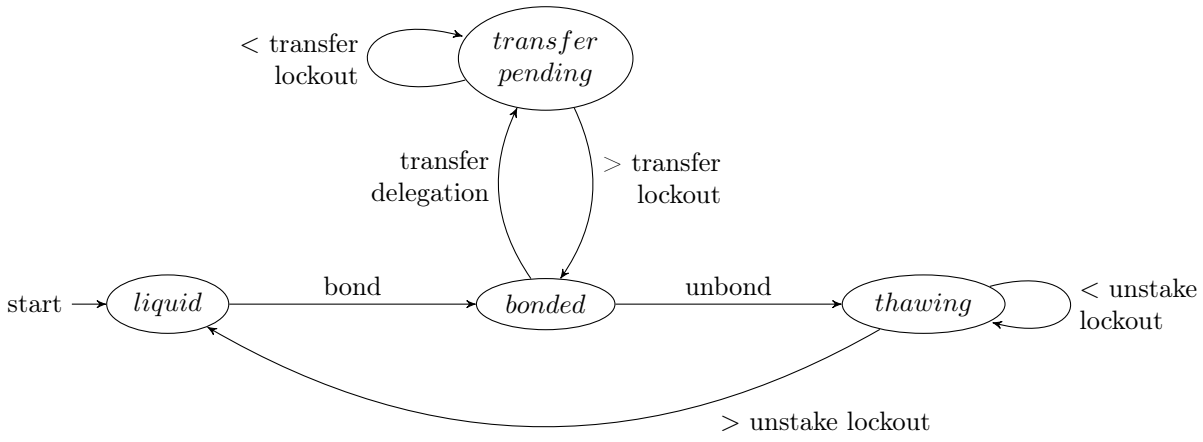

Figure 1: States that a coin can be in, within the Staker Registry

- ***Liquid***: A coin is liquid when it is owned by the user's account. In this state, the user is free to transact in the system (e.g. transfer to another user, stake, pay transaction fee for smart contract interactions, etc.).
- ***Bonded***: A coin is bonded if it has been cast as stake by its owner (§4.2). Bonded coins can reclaimed by the user via unbonding, or can be transferred to another staker.
- ***Transfer-pending***: Bonded stake can be transferred from one staker to another in the Staker Registry. A transfer is equivalent to a staker unbonding, transferring the liquid coins to another staker, and that staker bonding those coins for herself (i.e. a transfer of stake signs over the sovereignty of those coins to another staker). Before a transfer is complete, the coin must be locked for some short period (transfer-pending period) in order to resist stake grinding attacks (as described in [WZS19]).
- ***Thawing***: Any time a coin has been unbonded, a thawing period must be applied to preserve Unity consensus security invariants [WZS19]. After the thawing period has elapsed (defined in some number of blocks elapsed since the unbond action), the coins go into the liquid state (i.e. get returned to the user's account).

3.4 Asynchronous Tasks

There are several instances of "asynchronous" tasks in the Aion-Unity staking system. These features are asynchronous in the sense that these interactions cannot be completed within one transaction initiated by the user; they require some action by the protocol itself (not initiated by a user), after some condition on contract or chain state has been met.

The Staker Registry contract involves the following asynchronous actions:

- ***Unbonding of Stake***: Upon unbonding of stake, the coins must be ineffective in PoS and immobile (not liquid) for at-least the thawing period §3.3.
- ***Transfer of Stake***: Upon transfer of stake from one staker to another, the coins must be ineffective in PoS and immobile (not liquid) for at-least the transfer-pending period §3.3.

No mechanism exists in the protocol for transactions to be scheduled in the future or be automatically triggered if a (chain or contract) state change occurs, since it violates security constraints of the AVM computation metering system. Therefore, the aforementioned features must be implemented vis à vis two disparate transactions: an *initiating transaction* and a *finalizing transaction*.

For users of the Staker Registry, two-step transactions represent a source of severe friction. In order to keep the user experience as "familiar" (w.r.t traditional web services) as possible, incentive mechanisms are designed to shift the responsibility of calling the finalization transaction from the staker initiating the asynchronous transactions, to anonymous users of the system. The insight enabling this mechanism stems from the idea that, once an "asynchronous" transaction has been initiated by an authorized user, finalization can technically be performed by anyone in the system (since it is not an authorized action).

In the case of ***unbonding and transfer of stake***, the delegator must specify a fixed-fee when initiating these transactions. This fee is the bounty transferred to the caller of the finalize transaction, and is subtracted from the coins being transferred or unbonded. The implementation must perform the following range check on the fee:

$$0 <= fee <= amount\ unbonded\ or\ transferred.$$

Furthermore, to keep implementation compatible for all contract callers of finalization transactions, even if fee is set to zero (0) by the user, transfer of a 0 value should be called upon finalization to invoke any contract logic that handles payments, to handle the case of contract-caller needs to manage some state based on this the coin-payment interaction.

3.5 Self-Bond Requirements on Stakers

We propose imposing a self-bond requirement on all stakers registered in the Staker Registry. If the self-bond goes below the prescribed minimum, then all stake bonded to the staker becomes ineffective (i.e. loss of influence in PoS). There are two major reasons to implement such a scheme.

First, this creates a minimum cost (outside of the cost to run computer hardware) for someone to become a staker. This adds a barrier for anyone looking to launch a sybil attack (a single stakeholder creating a large number of staking pools (§4) in the hopes of dominating PoS voting power by means of random delegators choosing their pool).

Second, this facilitates the (potential) implementation of a slashing mechanism[1] to punish any misbehaviour (protocol deviation) on part of a staker who's purely a pool operator (§4). If there

[1] slashing mechanisms will be considered in subsequent protocol updates

were no self bond requirement, the pool operator has no "skin in the game", and could behave maliciously (w.r.t the protocol) with impunity.

3.6 Key Management

The Staker Registry should disperse system-responsibilities over several asymmetric key-pairs, to carefully refine the security requirements on each key, which is standard practice in industry. A staker is required to manage the following four (4) keys, each of which correspond to a distinct function:

- ***Identity key***: The address corresponding to this key becomes the identity of the staker in the Staker Registry system. This key cannot be used to perform any management actions in the context of the Staker Registry. Each identity key can only be used once to register a staker (since identities are unique in the Staker Registry). There should be no mechanism to replace this key with another one.
- ***Management key (cold key)***: The address corresponding to this key is recorded upon staker registration. There should be no mechanism to replace this key with another one. This key should be kept in cold storage (HSM) at all times, and only be retrieved when management tasks need to be performed. This key has the following management rights and responsibilities:
  - Change registered coinbase address
  - Change registered block signing address
  - Set the active status of the pool
  - Bond, unbond and transfer stake.
- ***Block-signing key (hot key)***: This key entitles its holder the rights to produce blocks on behalf of the stake owned by the staker. This key needs to be kept online while connected to the Aion-Unity network, since this key is required to sign any blocks won by the staker in the course of the PoS "lottery". At the moment, the guidance is to load this private key in the memory of an appropriately permissioned process, or in a commercial HSM (e.g. YubikeyHSM 2) that supports EdDSA (ED25519) signatures. If this key is compromised, the attacker can censor (deny inclusion of) transactions within the blocks supported by the delegated stake. Furthermore, block signing key can't be changed too fast (subject to a cooling period) to avoid a variant of the stake grinding attack (as described in [WZS19]), where the variable that an attacker can "grind" is the block-signing key to give the attacker an advantageous time-to-next-block.
- ***Coinbase key***: PoS block rewards get paid out to the address corresponding to this key. If the address corresponding to this key is a smart contract, then it must be secured from known attacks on contract balances. If this address is a user account, the owner of the account must secure it appropriately.

3.7 Staking System Parameters

Key system parameters pertaining to PoS, as defined in the Unity paper ([WZS19]), are enforced in the Staker Registry contract:

- ***Post-active (thawing) period***: This refers to the time (in blocks) that stake is immobile and ineffective (w.r.t. PoS staking) after a staker has initiated a stake withdrawal. See §3.3 and [WZS19] §4.1.1 and §5 for further details and rationale.
- ***Pre-active (thawing) period***: The pre-active period as defined in [WZS19] §4.1.1 will be zero (i.e. no pre-active stake locking).
- ***Transfer lockout period***: This refers to the time (in blocks) that stake is immobile and ineffective (w.r.t. PoS staking) after a staker has initiated a stake transfer to another staker. See §3.3 for details.
- ***Self-bond minimum***: This refers to the minimum amount of bond required to be "put up" by a staker in the Staker Registry. See §3.5 for details.
- ***Signing address cooling period***: This refers to the minimum amount of time a staker must wait before changing the block signing address in the Staker Registry. See §3.6 for details.

4 Public Delegation Protocol (Pool Registry)

The delegation system is completely decoupled from the core protocol. The Staker Registry has no notion of delegation; it exposes a very simple interface to bond, unbond and transfer stake (while maintaining security-related system invariants). This core feature-set in the Staker Registry enables the development of arbitrary delegation protocols, through the use of AVM smart contracts we will refer to as *delegation contracts*. These contracts are primarily responsible for distribution of rewards to delegators and managing the life-cycle of the stake delegated to a pool. Figure 2 offers a component-level sketch of how the purported delegation contracts relate to the core Staker Registry contract.

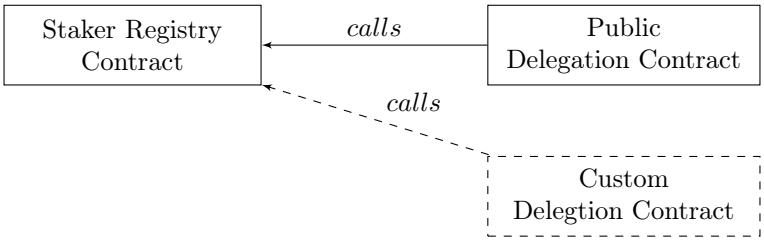

Figure 2: Contracts implementing staking and delegation in Aion-Unity

Public Delegation Protocol A key goal in Unity system-design is the engagement of all Aion coin-holders in staking. To serve that goal, we further specify an opinionated *public delegation protocol*, we refer to as the ***Aion Delegation Standard (ADS)***. In the Aion-Unity upgrade, the

ADS is implemented as a delegation contract called the ***Pool Registry***. All Foundation-supported staking user interfaces will support delegation as specified by this standard.

Furthermore, a clear specification and reference implementation for a public delegation standard allows third party tools, like wallets, to rapidly implement Aion stake-delegation in their products; this further amplifies the accessibility of participating in staking as an Aion coin holder.

Custom Delegation Protocols Since the public delegation protocol is a (non-privileged) contract leveraging the API exposed by the Staker Registry, one can theoretically construct other on-chain stake delegation protocols.

Private Staking Henceforth, we refer to the act of staking directly through the Staker Registry as ***private staking***, in order to discern this action from ***delegation of stake*** through a delegation contract. To contract this to the PoW paradigm, technically inclined users interested in running the equivalent of a PoW *solo-miner* may stake directly through the Staker Registry, whereas users wishing to participate in the equivalent of PoW mining pools can commit their stake through a delegation contract.

4.1 The Need for Delegation

In the PoW paradigm, mining pools allow participants with modest hash power to *pool* their resources with strangers to achieve a more reliable outlay of rewards. Similarly, in PoS networks, *delegation* allows users to transfer their rights to participate in the proof of stake (PoS) protocol to *stake pools*.

The rationale for stake-delegation is very much in-line with the rationale for hash-power-pooling: since PoW miners controlling small amounts of hash power are not expected to run a full node in order to write blocks on rare occasions, we should not expect stakers with a small amount of stake to do so either. Furthermore, while miners are generally technically proficient and commit their time to maintaining on-premises infrastructure, owners of stake in the network might lack the expertise or time to do so. Even if one has the willingness to operate a staking node, one might have too little stake to cover operational costs.

Delegation allows all coin holders of Aion to contribute to PoS security, regardless of the amount of coins they own, or their technical capabilities.

4.2 Smart Contract Architecture

Here, we outline the design of the smart contract implementation of the staking and delegation system in Aion-Unity. The system consists of three distinct contracts (with their relationship depicted in Figure 3):

- ***Staker Registry Contract***: This contract tracks all stake and stakers in the system and is core to the Aion-Unity protocol (defined in §3). Anyone wishing to cast their coins to stake (including other contracts, e.g. delegation contracts), need to register here.
- ***Pool Registry Contract***: This contract is non-core to the system (i.e. has no special privileges in the system). This is the implementation of the Aion public delegation protocol, and is responsible for:

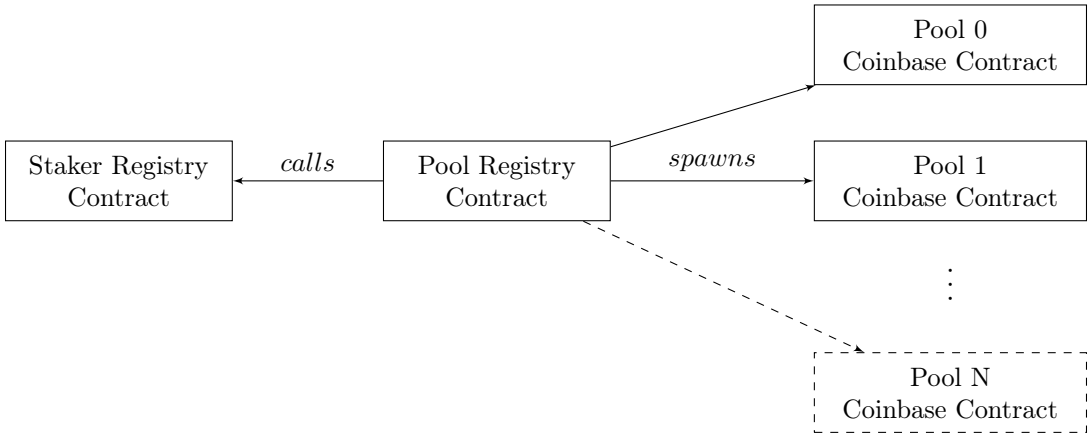

Figure 3: Smart contract architecture for stake delegation in Aion-Unity

  - keeping a registry of all staking pools and associated meta-data,
  - receiving coins (as delegation to a pool) and casting them to stake,
  - managing internal state of each pool (including delegation and rewards),
  - and splitting the rewards among the pool operator and delegators.

  A key feature in this contract is that it guarantees to delegators that they have complete ***sovereignty over their stake***; in delegating coins, no-one except the delegator can affect the state of the stake or accumulated rewards in any way (e.g. unbond, transfer delegation, withdraw rewards, etc.)

- ***Pool Coinbase Contract***: An instance of the Pool Coinbase contract is spawned by the Pool Registry every time a pool is instantiated. The pool's coinbase contract receives block rewards on behalf of the pool. For a particular pool, when the operator or any delegator requests a withdrawal (in the Pool Registry), the coins are withdrawn from this contract by the Pool Registry. Note that this is an implementation artifact; the end user should never need to understand the function of this contract.

  Recall that the Pool Registry has no special privileges. Block rewards are disbursed by crediting the coinbase account cited in the block header. Since all pools and their states are managed in the Pool Registry, if the block rewards for all pools were paid out to this contract, one would not be able to discern which pool the disbursement was intended for (since no contract code can get executed by the protocol upon block reward outlay, as-per AVM design). Therefore, the Pool Registry must spawn and manage a unique contract (per pool) to receive block rewards in the standard way, and then retrieve the rewarded coins from this contract when withdrawals are requested by any delegator or operator.

4.3 System Actors & Actions

The public delegation protocol has two primary actors interacting with the system: the delegator and the pool operator. For each system actor, we now enumerate the actions that each of the system actors can perform with respect to the Pool Registry contract.

4.3.1 Delegator Actions

The following are the actions that a coin holder seeking to delegate staking rights can perform with respect to the Pool Registry contract:

- **Stake Management**
  - ***Delegate stake***: A user can delegate to a pool by sending their coins to the Pool Registry contract, citing the identity address of the pool as a transaction parameter.
  - ***Undelegate stake***: A user can withdraw any fraction of their funds from the staking service by performing an undelegate action in the Pool Registry. For a period of time measured in number of blocks since the unbond action, the coin will be in the thawing state; it will be held in the staking contract and will neither contribute to stake securing the system, nor will it be liquid until the unbonding period has elapsed (see [WZS19] for details).
  - ***Transfer-delegation***: A user can transfer any proportion of their bonded stake to another staking pool (subject to a transfer-pending period).
- **Auto rewards delegation**:
  - ***Opt-in auto rewards delegation***: A user can opt auto-rewards delegation scheme at any time (regardless of the fact that they have stake delegated in the system, rewards accumulated, etc.). A user can call this function if they have already opted-in, to change the fee offered for auto-rewards delegation (see §4.7 for details).
  - ***Opt-out auto rewards delegation***: A user can choose to opt-out of the auto-rewards delegation scheme at any time.
- **Rewards management**:
  - ***View rewards***: Given a delegator address, users should be able to publicly view accumulated rewards (including all rewards distribution events).
  - ***Withdraw rewards***: A user can choose to withdraw accumulated rewards to their account. No unbonding period is applied to this amount; as-soon-as the withdraw transfer is committed on-chain, the user should be able to access the liquid balance in their account.
  - ***Re-delegate rewards***: A user can manually re-delegate their rewards, to increase the total amount of bonded stake they have committed in the system.

4.3.2 Pool Operator Actions

The following actions can be performed by the staking pool operator with respect to the Pool Registry:

- ***Register***: To facilitate discovery of stake pools by delegators, as part of pool initialization, all pools must register with the Pool Registry contract. This registry will contain a list of all active pools. In the transaction that registers a pool, the operator must send at-least the minimum number of coins (to be self-bonded) to fulfill the minimum self-bond requirement in the Staker Registry (§3.5). In addition, the registering pool operator must provide the following data:

  - *Commission rate*: A number between 0%-100% (up to two (4) decimal place precision), which indicates fees the pool operator charges. See §4.6 for details on how this number is applied to block rewards and fees are distributed to pool operators.
  - *Metadata URL*: The pool operator must host a JSON file at this URL (HTTPS over TLS), containing the metadata that is displayed in ADS user interfaces (§5). See the section on the metadata protcol (§4.11) for requirements on what this JSON object should contain and other ancillary concerns.
  - *Metadata content-hash*: This is the Blake2b hash of the JSON object hosted at the metadata URL. This is used as an on-chain commitment of the data hosted at the metadata URL. See §4.11 for details on the function of this content-hash within the metadata protocol.
  - *(Block) Signing address*: This is the address corresponding to the secret key that the pool operator will use to sign the blocks produced (see §4.9 on the key-management scheme in the public delegation protocol).

- ***Tear-down***: No graceful pool tear-down functionality has been built into the Pool Registry, to simplify the state-space of the contract. Instead, if a pool operator no longer wishes to a operate a pool, they should withdraw their self-bonded stake (violating the pool invariants and putting the pool into a *broken* state) §4.4. The operator can optionally send an out-of-band message to their delegators (by means of updating their pool meta-data), informing them that this pool will no longer be maintained.

- **Management functions**: The following functions related to the management of a pool can be performed by the pool operator:

  - ***Update commission rate***: The pool operator can change the commission charged for operating the pool, due to market conditions, etc. A delay should be enforced between the time the commission change requested and is applied; this delay should exist to allow delegators ample time to respond to fee-changes (e.g. stay delegated to the pool, transfer their delegation to another pool, etc).
  - ***Update meta-data***: The pool operator can change the metadata shown in the wallet, by updating the content hash and/or the metadata URL. Since the metadata only contains display information about pool, this feature can be used by pools to communicate updates and announcements to the delegators.
  - ***Update bonded-stake***: The pool operator can bond new coins or withdraw bonded coins, in order to satisfy the pool's delegation capacity ratio (see §3.5 & §4.5 for details).
  - ***Update block-signing address***: If the pool operator suspects their block-signing key is compromised, they can update the block-signing address (the "hot key"). See §4.9 for details.

4.4 Staking Pool Life-cycle

The staking pool life-cycle is very simple, as depicted in Figure 4. A staking pool can either be in the *active* state or the *broken* state.

- ***Active State***: If the staking pool in the active state, it is eligible to receive delegated stake and fulfill all responsibilities of a pool-operator within the protocol. The pool starts in the *active* state; the pool is required to fulfill the core minimum self-bond requirements as outlined in §3.5 as part of the initialization call.
- ***Broken State***: The pool transitions into this state if either one of the system self-bond requirements is violated (the self-bond requirement in the Staker Registry (§3.5), or the delegation capacity ratio requirement in the Pool Registry (§4.5).

  The following penalties are applied to a pool in the broken state:

  – The pool is banned from producing new blocks (i.e. any coins delegated to a broken pool will be ineffective as stake).
  – No new delegations can be made to the pool.

  Due to the stake-sovereignty property of the system (§3.1), the coins committed to a broken pool remain in the delegated state (any coins delegated to a broken pool will be ineffective as stake); the user must manually undelegate or transfer the coins out of the pool (applicable lockouts apply). Furthermore, delegators are indefinitely eligible to withdraw earned rewards from a pool that has transitioned into a "broken" state.

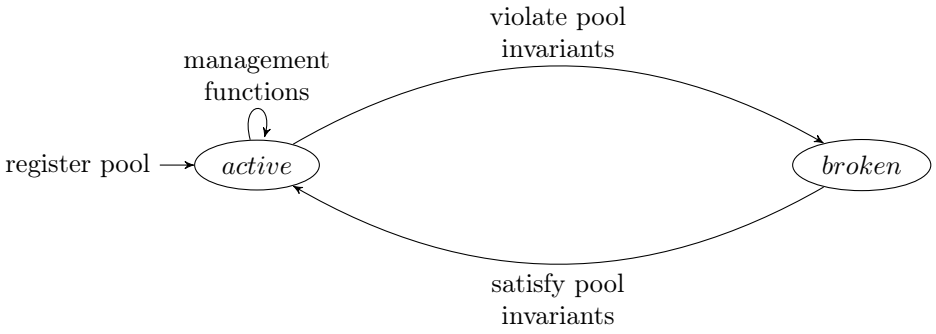

Figure 4: Staking pool life-cycle

4.5 Restricting Pool Size (Delegation Capacity Ratio)

Designing incentive mechanisms that promote decentralization in PoS delegation protocols is an open problem. PoW has a tendency to centralize via the creation of mining pools; in the Bitcoin network, mining pools have been observed to control over 50% of network hash-rate on occasion [RJZH19]. The design of a public stake-delegation protocol for Aion-Unity requires mechanisms to incentivize decentralized behaviour, to avoid the pitfalls encountered by PoW mining pools.

The challenge inherent to such a design is the trade-off between efficiency and decentralization:

– *Efficiency* is optimized if only one pool operator exists (that all stakeholders delegate to). This would minimize the operational costs of the network (since all the stake is delegated to one node, which would be the only PoS node operating on the network); this would in-turn maximize profits for all stakeholders.
– *Decentralization* is maximized if every single stakeholder runs their own node to contribute to PoS network security. Note that the PoS subsystem in Unity was designed to boost this kind of maximally-decentralized network configuration.

Furthermore, stake pool systems are susceptible to *Sybil attacks*; consider an attack where the adversary can take over the network by registering a large number of stake pools, hoping to accumulate enough stake to mount an attack just by people randomly delegating to them. Operation of the PoS network in any of the aforementioned regimes is undesirable. Therefore, an incentive-based mechanism needs to designed to find a balanced solution, such that some large number of pools with uniformly distributed stake-delegations can be encouraged.

Brunjes et al. propose a rewards-distribution function such that a targeted number of stake pools can be incentivized to exist (a proof for a Nash equilibrium arising from rational play for such a condition is provided) [BKKS18]. Although promising, further analysis is required to adapt this rewards mechanism to Unity consensus, and may be considered in an upcoming update.

Instead, the following simple scheme (inspired by Tezos [Goo14]) is proposed. It involves imposing a limit on the amount of stake that can be delegated to a pool based on a ratio of the self-bonded stake to the total capacity of a pool. Consider an example: assume that the delegation capacity ratio (in the Pool Registry) is 2% and the minimum self-bond requirement (in the Staker Registry - §3.5) is 100 coins. If the pool-operator has self bonded 1000 coins (satisfying the self-bond requirement in the Staker Registry), then the maximum stake that can be delegated to this pool is $50,000$ coins (since $2\% * 50,000 = 1,000$). Any operation that attempts to "over delegate" the pool should fail.

Although this scheme does not provably deter centralization or Sybil attacks, it produces some barriers to them, by requiring stake pool operators to allocate a finite resource to each individual pool they register, while enabling a large number of honest pools to operate in the PoS ecosystem.

Pool Operators Pool operators should earn staking rewards for their self-bonded stake, just like any other delegator in the pool. Therefore, the pool operator's self bonded stake must be delegated through the Pool Registry contract.

4.6 Reward-Splitting Scheme

Rewards sharing must be an on-chain, automatic process that does not require any action on part of the pool operator. For each block that a pool "wins", the block rewards are split between the operator and the delegators using the following scheme: first the operator fee is deducted from the block rewards, then the remaining balance is split between the delegators, weighted by their respective stakes pledged to the pool. See §4.6.1 for details regarding how this rewards splitting is implemented.

For example, if the block reward is 5 coins, the pool fees are 20% and there are three delegators, whose pledges account for 50%, 37.5% and 12.5% of the pools total stake respectively. Then the rewards distribution (for that block) will look like the following:
– Operator rewarded 1 coin (20% of 5 coins)
– Delegator #1 gets 2 coins (50% of the remaining 4 coins, after deducting operator fees)

– Delegator #2 gets 1.5 coins (37.5% of the remaining 4 coins, after deducting operator fees)
– Delegator #3 gets 0.5 coins (12.5% of the remaining 4 coins, after deducting operator fees)

4.6.1 Rewards Distribution Algorithm

The problem of on-chain rewards distribution appears trivial at first glance, but deserves significant attention in its design.

First, lets consider the naive solution to the problem of block rewards distribution. Assume some hypothetical "onBlockProduced" function that executes as the last transaction in a PoS block. In this function, the block rewards are disbursed to the pool's operator and delegators. The time-complexity of this function is linear in the number of delegators in the pool. Since the disbursement to each delegator requires at-least one state update (database write), state updates linear in the number of delegators in the pool need to be performed. Furthermore, since state updates are a fairly costly operation in the Aion-AVM regime, the operation of updating the state for every delegator in a pool could easily surpass the total gas limit of a block for large pool (recall that the block gas limit represents the maximum amount of computation that network participants can perform in the validation of blocks, while keeping the block-time stable). Since the combined gas cost for PoS rewards disbursements and the transactions included in a block cannot exceed the total block gas limit, this naive solution is clearly impractical: consider a "full" PoS block; there is no gas budget left (in the block) for running the rewards disbursement!

F1 Rewards Distribution A clever solution this problem was proposed by Ojha [Ojh19]. A detailed description of the rewards distribution mechanism is out of the scope of this report (please refer to [Ojh19] or a toy-implementation by the author [Sha19]).

This algorithm moves all rewards distribution computation from some hypothetical "onBlock-Produced" function, to any event where the rewards-per-unit-stake changes (e.g. delegate, undelegate, withdraw, etc.). The rewards distribution is still linear in the number of delegators, but the computation is amortized over the interactions users have with the system (i.e. all fee distribution computations are "fuelled by" user-provided gas).

Furthermore, this algorithm is approximation-free; in the rewards distribution calculations, the only source of approximation is finite decimal precision, which cannot be avoided (confirmed using simulation in [Sha19]).

4.7 Asynchronous Tasks

Asynchronous tasks, as introduced in the context of the Staker Registry (§3.4), also exist in the delegation system. These features are asynchronous in the sense that these interactions cannot be completed within one transaction initiated by the user; they require some action by the protocol itself (not initiated by a user), after some condition on contract or chain state has been met.

The paradigm that asynchronous transactions should be implemented through two disparate (initiating and finalizing) transactions, and the fact that finalization can be performed by anonymous users applies to the Pool Registry contract by the same rationale as argued for the Staker Registry (§3.4).

In order to keep the user experience as "familiar" (w.r.t traditional web services) as possible, incentive mechanisms are designed to shift the responsibility of calling the finalization transaction from the staker (initiating the asynchronous transactions), to anonymous users of the system.

The following is a list of asynchronous actions that should be implemented in the Pool Registry, along with the incentive-design of finalization calls:

- ***Undelegation of Stake***: This is a managed-wrapper around the unbond functionality exposed in the Staker Registry.
  - **Incentive**: A fixed fee incentive mechanism should be implemented (as defined for the unbond function in Staker Registry §3.4).
- ***Transfer Delegated Stake***: This is a managed-wrapper around the transfer functionality exposed in the Staker Registry.
  - **Incentive**: A fixed fee incentive mechanism should be implemented (as defined for the unbond function in Staker Registry §3.4).
- ***Auto Rewards Delegation***: Rewards earned by delegators in the system can be automatically cast to stake on behalf of the delegator.
  - **Incentive**: The delegator must specify a fee as a percentage of the coin transfer amount (with four (4) decimal precision) that callers of the finalization function could collect. Bounty-seekers could then scrape accounts registered for this finalization function; they would wait for enough coins to be accumulated such that the fee collected upon the function call exceeds the caller's transaction cost by some profit threshold.
- ***Update Commission Rate***: Update of the commission rate (by an operator) is subject to a lock-out period to give delegators time to react to fee changes, before rate-change is imposed.
  - **Incentive**: This is the only asynchronous transaction in the Pool Registry that exclusively affects the pool operators. It is assumed that pool operators have sufficient motivation to promptly come back online after the lockout period has elapsed in order to make the finalization transaction to make the new commission rate effective. Therefore, no incentive mechanisms shall be built-in for this call.

4.8 Delegation System Parameters

Several of the core staking system parameters (as defined in §3.7) are transitively enforced in the Pool Registry (since features of the delegation system are "back-ended" by the Staker Registry). Upon unbond and undelegation, the *Post-active (thawing) period* is enforced (i.e. stake is immobile and ineffective w.r.t. PoS for the thawing period, before finalization can be called). Similarly, upon the transfer call, the *Transfer lockout period* is enforced.

The following is the list of parameters exclusive to the Delegation system:

- ***Delegation capacity ratio***: This refers to the ratio of self-bonded stake an operator must maintain, in proportion to the total amount stake delegated to a pool, at all times during the operation of the pool (see §4.5 for details).
- ***Commission-rate change lockout period***: This refers to the time (in blocks) that a pool operator must wait (starting from initiating a commission-rate change), until the commission-rate change is applied. (see §4.3.2 for details).

4.9 Key Management

The Pool Registry disperses system-responsibilities over several asymmetric key-pairs, to carefully refine the security requirements on each key. The pool operator is required to manage two private keys, each of which corresponds to a distinct function in the operation of the pool:

- ***Block-signing key (hot key)***: This key is functionally equivalent to the corresponding key described in the key management guidelines for the Staker Registry (§3.6). Same recommendations and guidelines apply. The address corresponding to this key must be provided upon instantiation of the pool.
- ***Identity key (cold key)***: This is the key used by the operator to register the pool. This key entitles it's holder the rights to all pool management tasks (§4.3.2). This key should be kept in cold storage (HSM) at all times, and only be retrieved when management tasks need to be performed. There is no mechanism to replace this key with another one. The address corresponding to this key is the identity of this pool; this address is used to address this pool for all on-chain interactions (e.g. delegators use this address to identify the pool, etc.).

Note that the pool operator is shielded (by the Pool Registry contract) from much of the complexity of managing the extra keys associated with the operation of a private staker through the Staker Registry (§3.6). Also, note that when registering for a pool (§4.3.2), do not use exchange addresses for any of the keys. The operator needs to control the private keys for these accounts in order to perform pool actions such as retrieve the rewarded coins, sign the blocks, perform management actions, etc.

4.10 Operational Requirements

The pool operator is the entity that gets delegated staking rights from users of the system. The pool operator's primary obligation with respect to the public delegation protocol is to run a Aion-Unity full node that is well connected to the blockchain network, in order to participate in the staking protocol as a block producer. In order to satisfactorily fulfill this obligation, the operator must run computer hardware with comparable or better specifications than:

- Intel i7 (Skylake, 6th generation) processor with 4 cores, 8 threads.
- 16 GB of DDR4 RAM
- 512 GB SATA SSD
- 50Mbps dedicated internet connection

The operator is required to keep at-least 99.9% ("three nines") availability, which corresponds to at-most 8.77 hours of down-time per year. Pool operators are encouraged to host a web page, advertising self-reported up-times and hardware specifications, among other pertinent information about pool operations, to instill confidence in the pool operator's operational capabilities.

4.11 Metadata Protocol

When a new staking pool is registered, it gets added to an on-chain registry of all active staking pools. During registration, the pool operator is required to provide metadata for the pool (e.g. logo, web-page, human-readable name, etc.). This metadata could then be consumed by user interfaces enabling stake delegation in Aion-Unity.

We define an explicit protocol for metadata management, which involves both on-chain and off-chain actions. This protocol standardizes the way user interfaces enabling delegation on Aion-Unity can retrieve rich data about staking pools. This empowers both pool operators and delegators:

- The pool operators have an easy process to maintain rich contextual descriptors about their pools. Without intervention on part of the operators, these descriptors automatically get pulled into, and updated across all user interfaces implementing the ADS. Without such standardization, pools would have to manage relationships with all relevant public-delegation front-end providers in order to get listed and manually manage delivery of contextual-information requirements (e.g. logo, pool descriptions, etc.)

- The delegators can rely on a rich set of descriptors provided by the pool operators to provide meaningful data points to inform their delegation decisions. Furthermore, the same rich set of descriptors shall be widely available across all user interfaces implementing the ADS.

4.11.1 Metadata schema

When a staking pool is registered in the Pool Registry, the pool operator must provide a *metadata URL* and *metadata content-hash*. The pool operator must host a JSON file at the metadata URL (HTTPS over TLS), with the following schema:

- ***Schema version***: A version number, to identify the schema. This is here to enable upgradability.
- ***Logo***: A thumbnail containing the logo of the pool. The image must be base64 encoded PNG, with the dimensions of 256 pixels-square.
- ***Description***: A "tell me about yourself"-style, short description for users to consume when making stake delegation decisions. This field shall not exceed 256 characters.
- ***Name***: A human-readable name for the pool. This field shall not exceed 64 characters.
- ***Tags***: These serve as keywords for any search functionality to be exposed by any public delegation protocol user interface. This is a JSON array. The size of this array shall not exceed 10 elements, with each element not exceeding 35 (valid) characters.
- ***Pool URL***: This is a URL, pointing interested delegators to the homepage of the pool, for additional information to peruse, in order to help make their delegation management decisions.

The JSON must be valid according to the RFC 7159 JSON specification [Bra14]. The hash of the JSON object must match the content hash provided. The document must be less than 1024×1024 bytes (1 mb). All characters must be UTF8 encoded. The document hosting service must guarantee three nines availability (99.9% uptime). The Blake2b hash of the JSON object must match the content hash provided on-chain.

4.11.2 Proxy servers

We further propose the use of *proxy servers* to cache the list of staking pools and their associated metadata. This enables rapid implementation of ADS user interfaces, since one could query a web-

service to retrieve this list, as opposed to querying each metadata URL in the staking Pool Registry. The implementation of this server is very simple:

- *Pool metadata caching*: When an operator updates the metadata hosted at the metadata URL, they must also update the metadata content hash on-chain. The proxy server polls the Pool Registry, listening for changes in content hash. When the content hash changes, the cache is invalidated and the new metadata is loaded into the cache.
- *Security screening*: The proxy server shall implement rules to filter out malicious content in the metadata. As attackers evolve mechanisms to attack the user interfaces, this simple server can be adaptively updated in response.

To address concerns around centralization and censorship of pool lists by the proxy server, we will open-source the proxy server implementation and encourage community participants to run these proxy servers. Furthermore, we require any ADS user interface to be configurable to query a number of these proxy servers.

4.11.3 Alternative approaches

An alternative approach considered in the design of the metadata protocol was to implement the Pool Registry off-chain. There were centralization risks associated with this approach, namely the censorship by registry-maintainers of pools on this list (either by omission or manipulation of rankings).

Another approach considered was storing the metadata directly on-chain (and doing away with the metadata-URL and proxy server scheme). This strategy was not selected due to flexibility concerns. The metadata is a field primarily used by pool operators as an advertisement avenue; this schema can conceivably be required to adopt additional fields to improve richness of pool metadata. In addition, this scheme allows for the metadata to become much larger than currently specified, without concerns of block size restrictions or on-chain data bloat.

Although this operator-hosted metadata scheme increases the implementation complexity slightly on part of the pool operators, it opens-up the opportunity to improve the quality of service of the ADS user interfaces in the future, without updates to the contract.

5 User Interface Concerns

We outline requirements for what a user interface (UI) for the ADS Pool Registry must achieve. No guidance is provided pertaining to how to achieve said requirements to afford implementation flexibility based on the medium (desktop, web, mobile), target audience, etc.

5.1 Pool Registry Presentation

Pool Listing The user interface shall produce a list of all active and broken (§4.4) pools in the Pool Registry. For each pool, any pertinent information required by the user to make delegation decisions should be presented (e.g. fee, capacity remaining, rewards estimates, etc.).

In order to help users make rational decisions with respect to their stake delegations, we propose that the Pool Registry listing should be default-sorted using some weighted function of:

– the fees charged by the pool,

– the apparent performance of the pool (see §5.2), and
– the remaining contribution capacity.

The goal of this proposed "attractiveness score" is to promote pools that are reliable, have not yet reached saturation, and have a low cost. This function is left unspecified (up to the implementation).

User Accounts The user should be able to delegate to multiple pools. The user should be able to view all their outstanding delegations, and the rewards earned from each delegation (at the block-level resolution).

Pool Metadata The user interface shall retrieve the Pool Registry and associated metadata from the metadata-proxy server (as defined in §4.11). The UI should be designed to handle scenarios where metadata is malformed (i.e. any of the metadata rules defined in §4.11 are violated). Furthermore, the UI may be configurable to query a number of metadata-proxy servers to promote diversity in metadata-proxy server providers.

5.2 Apparent Performance of Pools ("Onlineness")

The wallets should report some notion of up-time for a pool. This measure is critical to gauging the reliability of a pool, and directly impacts the rewards a delegator can expect to receive by delegating to this pool. Delegators should rationally choose pools with the highest possible historical up-times; even if a pool offers low fees, a spotty up-time track-record will manifest itself in diminished rewards.

Since there is no explicit way to capture an up-time metric in the design of Aion-Unity PoS (due to it's stochastic nature), we propose a simple measure called *apparent performance*, that serves as a proxy for the "onlineness" of the pool operator. For any established pool with sufficient size (in terms of delegated stake), we can infer its up-time by looking at the ratio of blocks produced by the pool over some historical interval, to the number of blocks one would expect the pool to produce over the same interval (by comparing average delegation to the pool and to the system as a whole). Concrete parameters and algorithm specification is left up to the implementation.

When a new pool is created, there is no data to determine it's apparent performance. New pools should be distinguished from the established pool (e.g. displayed in a separate section of the UI), since no reasonable measure for future performance can be inferred.

5.3 Pool Life-cycle Notifications

The UI is responsible to produce notifications for all key life-cycle events for the pools a user has delegated staking rights to, to enable a user to make appropriate delegation decisions.

Management Actions The UI must notify a user when a pool goes into the *broken* stake and changes it's fees (§4.3.2). Any *broken* pools must be clearly identified. The user should be able to transfer any delegations from a *broken* pool to an active pool at any time.

Inactive and Underperforming Pools The user should monitor the attractiveness score (§5.1) of all pools one has delegated stake to, in order to notify the user of any significant changes in this metric; particularly, large drops in this metric implies one or more of the following things:
– A large amount of stake has left the pool.
– The pool's average up-time has dropped significantly (operator has stopped producing blocks).

– The pool operator has changed the fee.
– The rewards earned by the user have diminished significantly.

This way, if the pool ceases to operate, it's delegators will be incentivized to transfer their delegation to another pool.

5.4 Reference Implementation

We offer a concrete instantiations of the parameters and mechanisms outlined above in §5.1 & §5.2, as implemented in the reference client.

5.4.1 Up-time Inference Algorithm

We offer an algorithm which determines the apparent performance (or "onlineness") of a pool as defined in §5.2. A pool is considered *established* if it has been active for at-least one (1) week, which translated to $\sim 60,480$ blocks, based on a mean block time of 10 seconds. Over a moving window of $60,480$ blocks (1 week), we define the following metrics:

– ***Stake Delegated to*** n^{th} ***Pool*** (S_n): Since the stake amount delegated to a pool can fluctuate over time, take a rolling average of the active stake delegated to the n^{th} pool over the moving window.
– ***Stake Delegated in System*** (S_t): Perform a similar calculation as above, except over the stake delegated to all the pools, to get an average for the total stake delegated to the system over the moving window.
– ***PoS Blocks Produced by*** n^{th} ***Pool*** (B_n): This measures the number of PoS blocks produced by the n^{th} pool in the system.
– ***PoS Blocks Produced in System*** (B_t): Since in Unity consensus, a chain consists of both PoS and PoW blocks, we need to measure the total number of PoS blocks produced in the system over the moving window.

We can then determine the ratio ($P_{expected}$) of blocks this pool was expected to produce (w.r.t all the PoS blocks produced in the sliding window), by computing:

$$P_{expected} = S_n / S_t.$$

We can also determine the ratio ($P_{expected}$) of blocks this pool actually produced (w.r.t all the PoS blocks produced in the sliding window), by computing:

$$P_{actual} = B_n / B_t.$$

We can then compute the performance ($Perf$) of the n^{th} pool, by calculating:

$$Perf = \frac{P_{actual}}{P_{expected}} * 100.$$

Pool Initialization The apparent performance of the pool will only be visible in the UI until at-least $60,480$ blocks have elapsed from the time the pool registers on the network. This is done because no reliable performance measure can be inferred until at-least a week's worth of data has been collected on the behaviour of the pool.

Bucketing by Performance Based on the $Perf$ metric defined above, the user interface categorizes the apparent performance of the pool into three (3) categories, to make it easier for users to differentiate between different general classes of up-times:

- ***Excellent***: If $Perf \geq 99$
- ***Good***: If $95 \geq Perf > 99$
- ***Poor***: If $Perf \leq 95$

Discussion There are several factors that could skew this calculation. First of all, the notional 1 week as the sliding window might not be a long enough time over which to compute these averages. Furthermore, large swings in stake contributions could skew the computation of the arithmetic mean; this may potentially be fixed by sizing the window as a function of the standard deviation of the stake contribution over time.

5.4.2 Pool Registry Presentation

The following parameters get displayed when a pool gets listed in the Registry view:

- ***Image***: The logo / thumbnail and image of a pool.
- ***Self Bonded Amount***: The coin amount that has been bonded by the pool operator.
- ***Pool Size***: The total stake managed by the pool, (from stakers + self bond).
- ***Stake Weight***: What percentage % of total stake in the network has been committed to this pool.
- ***Capacity Remaining***: The total capacity (in terms of stake) that the pool can accept without becoming *saturated*. This is derived from the Delegation Capacity Ratio defined in §4.5. Let S_{self} be the number of coins bonded by the pool operator, let $S_{delegated}$ be the amount of coins bonded by delegators, and let C_r be the capacity ratio as a percentage (between 0 - 100); then the capacity remaining is computed as follows:

$$CapacityRemaining = (S_{self} * (100 - C_r)) - S_{delegated}.$$

- ***Fee***: The fee (as a percentage) charged by the pool.
- ***Status***: The state of the pool (§4.4: *active* or *broken*.
- ***Performance***: Indicate the apparent performance of a pool, by using a colored indicator that corresponds to the "performance classes" defined in §5.4.1:
  - Green = Excellent
  - Yellow = Good
  - Red = Poor

5.4.3 Ranking in Pool List

This section proposes a strategy to rank the pools (impose an ordering) when they are displayed in a list format. The assumption underlying this scheme is that when a collection of ites is displayed in a list format, the order in which the items are presented implies some value-judgement to the items in the list (items at the top of the list being somehow "better" than the ones lower down). As it applies to this system, the default order in which the pools are presented in a list may introduce the following unintended behavior on part of delegators and pool operators:

- Delegators might be biased in favour of pools sorted on top of the list, leading to concentration of stake to a few pools.
- Pool operators could be incentivized to gamify any variables which could lead them to be ranked at top of the list.

Ranking Factors In order to mitigate the biases outlined above, any mechanism designed to order a pool list must take into account the following factors:

- ***Gamification of Ranking*** The ranking should be difficult to gamify by any participant in the system.
- ***Randomization*** Any pools which map to the same "bucket" should be randomized.
- ***Delegation Diversity*** The ordering of pools in the list should nudge users to select pools in such that it doesn't lead to concentration of stake within a few pools.

Ordering Algorithm We propose a simple (first-order) algorithm (which leverages intermediate results from the up-time inference algorithm defined in §5.4.1) to achieve the aforementioned requirements.

1. First, define three collections (or "buckets") for pools to be sorted into: G1, G2 & G3.
2. For all pools whose delegated stake as a percentage of total stake in the system (the $P_{expected}$ value as defined for the up-time inference algorithm in §5.4.1) is $> 1/K$, move them into G3 collection (where K is some "minimum number of pools required to achieve an acceptable level of decentralization").
3. Then, for all **remaining pools**, allocate them to a collection based on their achieved performance class (informed by running the up-time inference algorithm (§5.4.1)) like so:
   - *G1*: pools in performance class "Excellent"
   - *G2*: pools in performance class "Good"
   - *G3*: pools in performance class "Poor"
4. Then, within each collection, randomize the order in which the pools are listed (to be refreshed at-least once an hour)
5. Then concatenate the three collections, with G1 at the top, followed by G2 and then followed by G3, to generate a list view over all pools in the Pool Registry.

Note that the collection G1 contain all well performing pools, *except* for the ones which have more than 1/K of the total stake in the system delegated to them, to promote decentralization.

References

[BKKS18] Lars Brünjes, Aggelos Kiayias, Elias Koutsoupias, and Aikaterini-Panagiota Stouka. Reward sharing schemes for stake pools. *arXiv preprint arXiv:1807.11218*, 2018.

[Bra97] Scott Bradner. Rfc 2119: Key words for use in rfcs to indicate requirement levels, 1997.

[Bra14] T Bray. Rfc 7159: The javascript object notation (json) data interchange format. *Internet Engineering Task Force (IETF)*, 2014.

[Goo14] LM Goodman. Tezos—a self-amending crypto-ledger white paper. *URL: https://www. tezos. com/static/papers/white_ paper. pdf*, 2014.

[Ojh19] Dev Ojha. F1 fee distribution. Cosmos SDK Documentation, 2019. URL:https://github.com/ali-sharif/f1-fee-distribution/blob/master/Ojha19.pdf (version: 2019-01-03).

[RJZH19] Matteo Romiti, Aljosha Judmayer, Alexei Zamyatin, and Bernhard Haslhofer. A deep dive into bitcoin mining pools: An empirical analysis of mining shares. *arXiv preprint arXiv:1905.05999*, 2019.

[Sha19] Ali Sharif. F1 toy implementation. Github, 2019. url:https://github.com/ali-sharif/f1-fee-distribution/ (version: 2019-08-02).

[WZS19] Yulong Wu, Yunfei Zha, and Yao Sun. A unifying hybrid consensus protocol, 2019.

Appendices

A System Smart Contracts

Following are the AVM smart contracts (extracted as Java interfaces) for the staking and delegation contracts. Only the functions utilized in the major control flows (as described in appendix B) are provided. Also note that access modifiers (e.g. public, private, etc.) at the class and method level have also been stripped for brevity.

```
interface StakerRegistry {
    void registerStaker(Address identityAddress, Address signingAddress,
                        Address coinbaseAddress);
    void bond(Address staker);
    long unbond(Address staker, BigInteger amount, BigInteger fee);
    long unbondTo(Address staker, BigInteger amount, Address receiver,
                  BigInteger fee);
    long transferStake(Address fromStaker, Address toStaker,
                       BigInteger amount, BigInteger fee);
    void finalizeUnbond(long id);
    void finalizeTransfer(long id);
    void setState(Address staker, boolean newState)
    void setSigningAddress(Address newSigningAddress);
    void setCoinbaseAddress(Address newCoinbaseAddress);
}

interface PoolRegistry {
    Address registerPool(Address signingAddress, int commissionRate,
                         byte[] metaDataUrl, byte[] metaDataContentHash);
    void delegate(Address pool);
    long undelegate(Address pool, BigInteger amount, BigInteger fee);
    void finalizeUndelegate(long undelegateId);
    void redelegateRewards(Address pool);
    long transferDelegation(Address fromPool, Address toPool,
                            BigInteger amount, BigInteger fee);
    void finalizeTransfer(long transferId);
    BigInteger withdrawRewards(Address pool);
    void enableAutoRewardsDelegation(Address pool, int feePercentage);
    void disableAutoRewardsDelegation(Address pool);
    void autoDelegateRewards(Address pool, Address delegator);
    long requestCommissionRateChange(int newCommissionRate);
    void finalizeCommissionRateChange(long id);
    void updateMetaData(byte[] newMetaDataUrl, byte[] newMetaDataContentHash);
    void setSigningAddress(Address newAddress);
}

interface PoolCoinbase {
    void transfer(BigInteger amount);
}
```

B System Interactions

This appendix illustrates major user and cross-contract interactions, for didactic purposes. Note, that anything in curly brackets ({}) after function calls indicates the coin amount enclosed in the transaction (function call).

B.1 Private Staking (Bonding)

Figure 5 describes the scenario where a Staker chooses to privately bond their coins as stake (i.e. run their own node). The staker must first register with the StakerRegistry contract (sending at-least the minimum self bond amount). Then the staker may send a "bond" message (to their own address in the StakerRegistry), sending the amount to bond as the transaction amount.

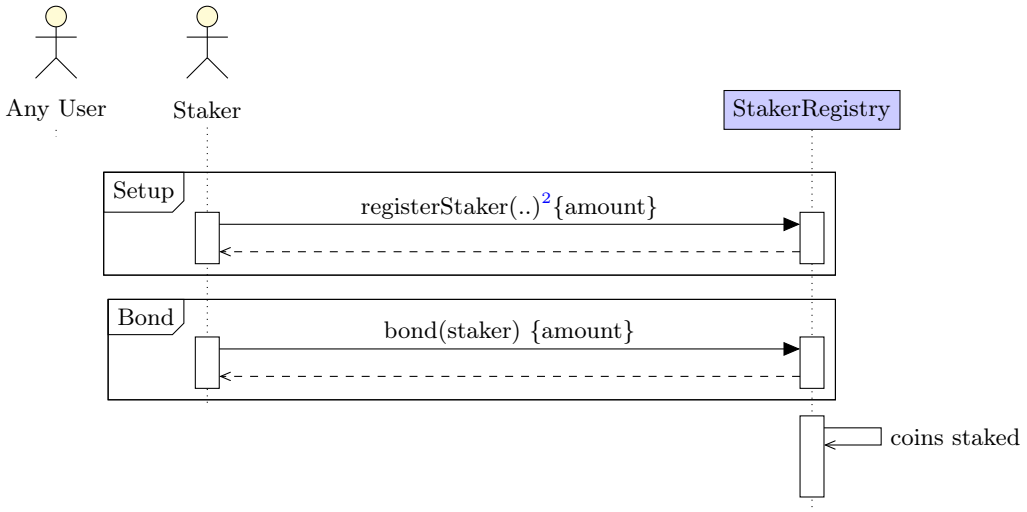

Figure 5: Private staking (bonding) in Aion-Unity

[2]registerStaker(identityAddress, signingAddress, coinbaseAddress)

B.2 Private Staking (Unbonding)

Figure 6 describes the scenario where a Staker chooses to privately bond their coins as stake (i.e. run their own node), and then after some time, unbond those coins. When the staker is ready to unbond the stake, the staker must do so across two transactions. First, the staker must send the unbond (with the amount to unbond) transaction, which commits the stake into a *thawing* state. This function returns an *unbondId*, which uniquely identifies this unbond operation. After the thawing period has elapsed (§3.7), at any time, a user in the system can call the finalize function, to release the thawed coins back to the staker.

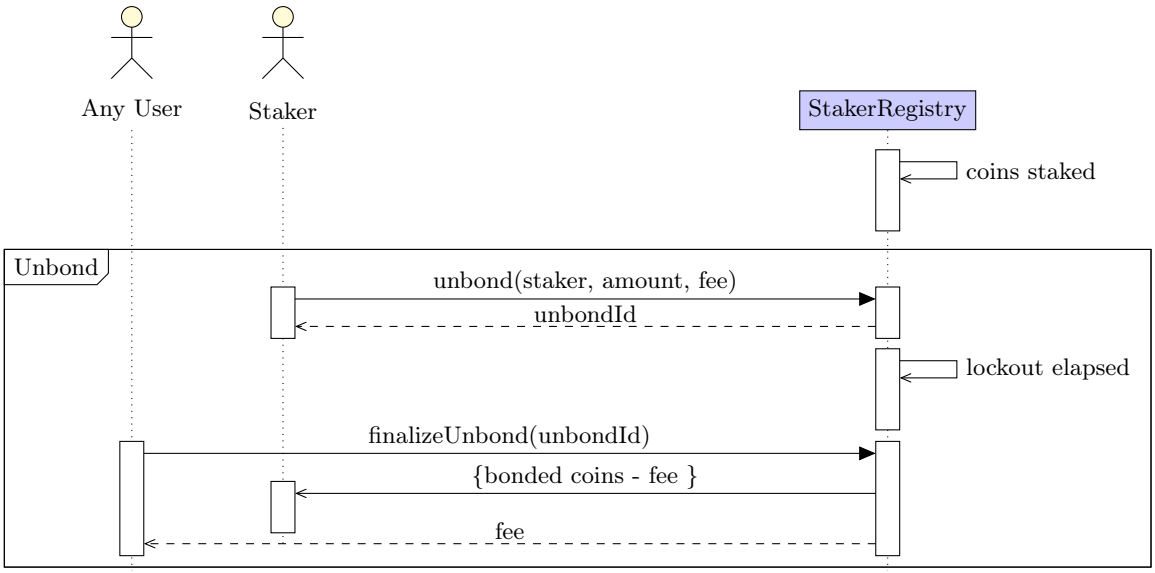

Figure 6: Private staking (unbonding) in Aion-Unity

B.3 ADS Pool Setup

Figure 7 describes the setup of a staking pool. The operator must sign up with the Pool Registry (using their hot and cold keys). The Pool Registry sets up the Pool Coinbase contract and registers the pool as a staker in the Staker Registry contract.

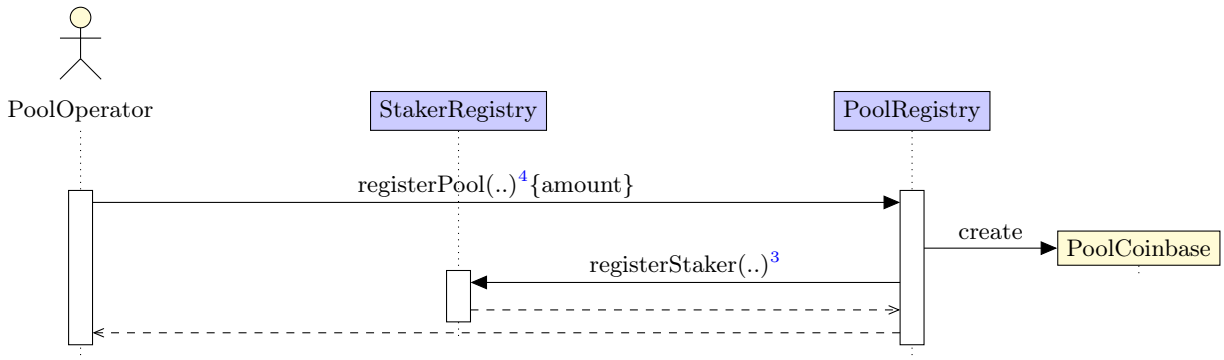

Figure 7: ADS pool setup in Aion-Unity

[3] registerStaker(identityAddress, signingAddress, coinbaseAddress)
[4] registerPool(signingAddress, commissionRate, metaDataUrl, metaDataContentHash)

B.4 Delegation to an ADS Pool

Figure 8 illustrates the mechanism of delegation and undelegation of stake to an ADS pool. In order to delegate stake, a user must send their coins to the PoolRegistry in a delegate transaction. When delegate is invoked, the PoolRegistry records the delegation and adds the stake to the pool's balance in the StakerRegistry; to the staking registry, the pool looks like one big staker. The undelegation of stake is a two-step process (since unbonding of stake is involved).

When a user calls the undelegate function in the PoolRegistry, an unbondTo is triggered in the StakerRegistry, which returns the corresponding unbondId that uniquely identifies the thawing of this parcel of stake. After the thawing period has elapsed, any user can call the finalizeUndelegate function through the PoolRegistry to release the liquid coins back to their account.

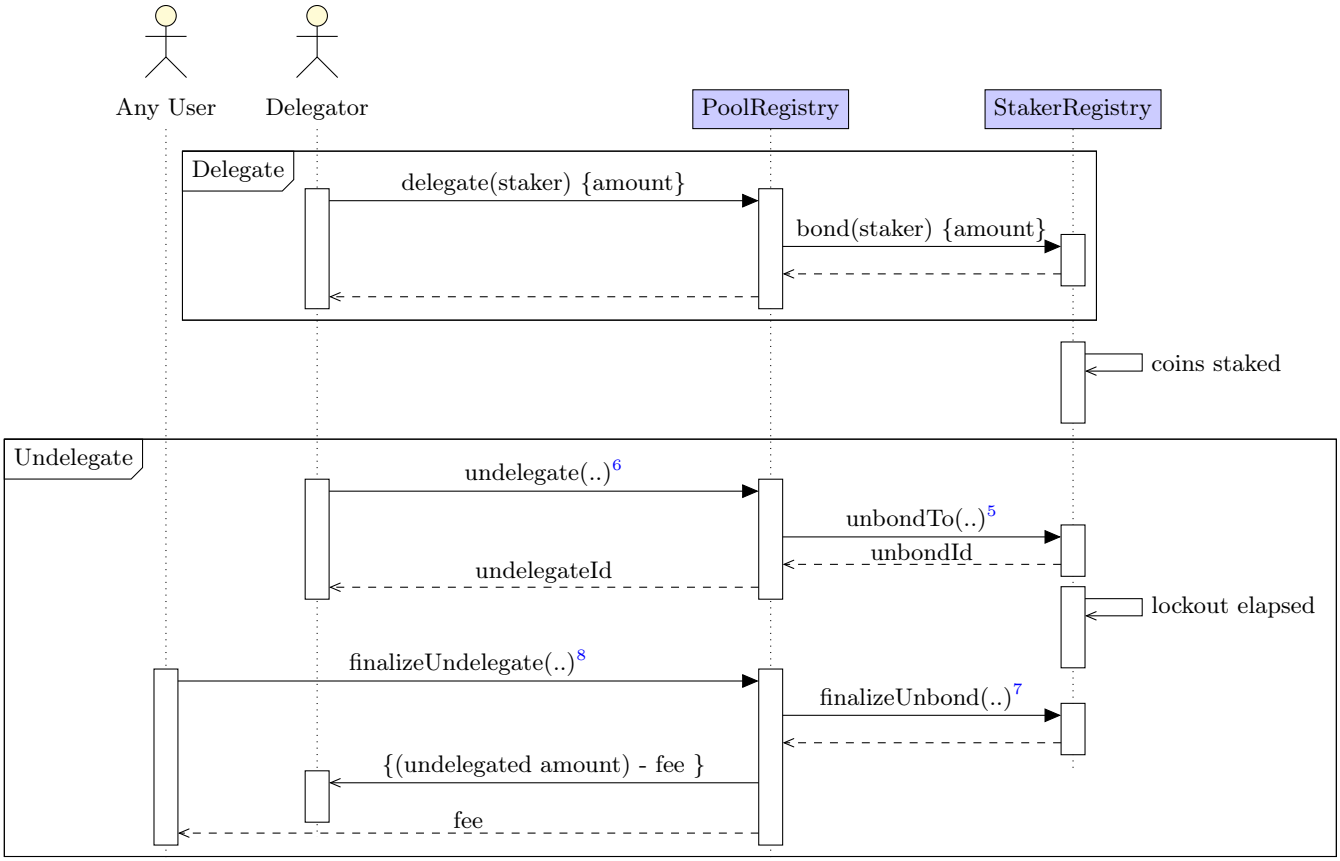

Figure 8: ADS delegation and undelegation in Aion-Unity

[5] unbondTo(staker, amount, receiver, fee)
[6] undelegate(pool, amount, fee)
[7] finalizeUnbond(unbondId)
[8] finalizeUndelegate(undelegateId)

B.5 Auto-Delegation of Rewards

Figure 9 demonstrates the auto rewards delegation feature in ADS. Once enabled, this feature enables any user to call the auto-redelegate function to commit a delegator's earned rewards as stake. At any point, the delegator can disable this feature.

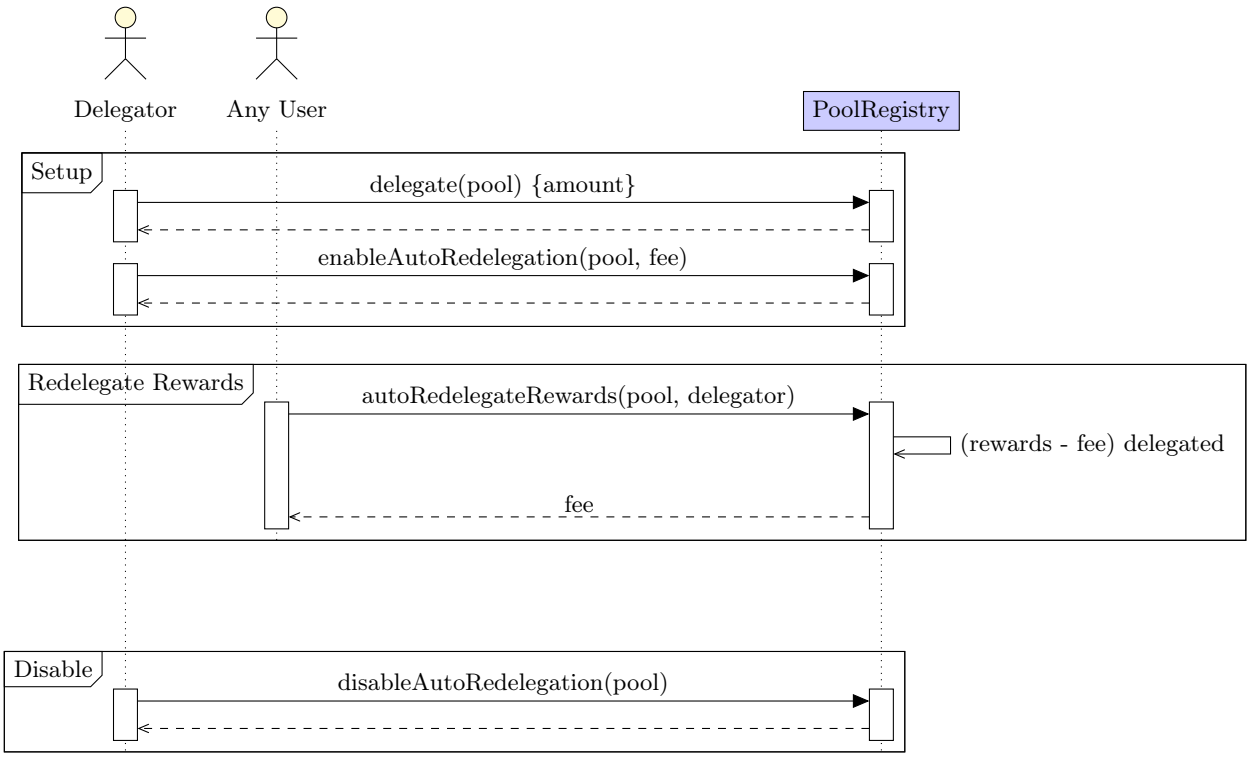

Figure 9: Auto redelegation of rewards in Aion Unity

B.6 Stake Transfer

Figure 10 demonstrates the stake transfer feature, at the level of the PoolRegistry (although this interaction looks almost identical for a solo-staker interacting directly with the StakerRegistry).

A delegator must initiate a transfer of stake between stakers (pools) at PoolRegistry, which in-turn reflects the transfer in the StakerRegistry. A transferId is returned, which uniquely identifies this transfer. After the transfer lockout period has elapsed, any user can call finalize to move the stake between the source and destination stakers.

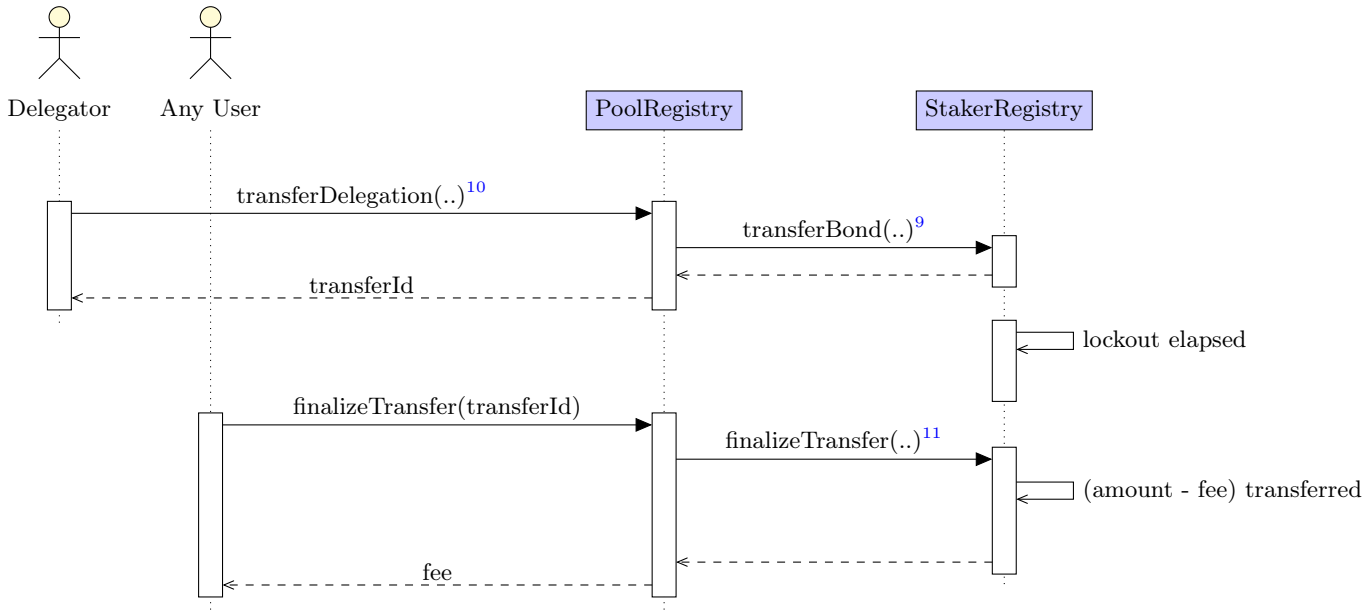

Figure 10: Stake transfer feature in Aion-Unity

[9] transferBond(fromStaker, toStaker, amount, fee)
[10] transferDelegation(fromPool, toPool, amount, fee)
[11] finalizeTransfer(transferId)

B.7 Rewards Withdrawal

Figure 11 demonstrates a withdrawal of rewards earned by a delegator. Rewards are continually withdrawn from a pool's coinbase contract any time the stake apportionment in the pool changes (via a delegation, undelegation, etc.) and managed by the PoolRegistry (see §4.6.1 for details). A withdraw is yet another trigger for the pool's coinbase to be emptied of accumulated rewards (if any exist). Then, the F1 rewards sharing algorithm is invoked to compute the rewards owed to the delegator, which are promptly disbursed before winding down the transaction.

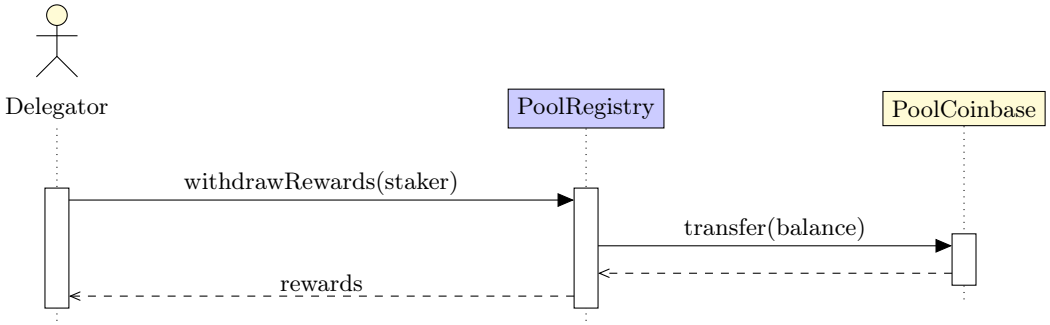

Figure 11: Delegation rewards withdrawal in Aion-Unity