POSDAO

Proof of Stake Decentralized Autonomous Organization

Authors: Igor Barinov, Vadim Arasev, Andreas Fackler, Vladimir Komendantskiy, Andrew Gross, Alexander Kolotov, Daria Isakova

Abstract

In this paper we introduce POSDAO, a Proof of Stake (POS) algorithm implemented as a decentralized autonomous organization (DAO). It is designed to provide a decentralized, fair, and energy efficient consensus for public chains. The algorithm works as a set of smart contracts written in Solidity. POSDAO is implemented with a general purpose BFT consensus protocol such as Authority Round (AuRa) with a proposer node and probabilistic finality, or Honey Badger BFT (HBBFT), leaderless and with instant finality. Validators are incentivized to behave in the best interests of a network through a configurable reward structure. The algorithm provides a Sybil control mechanism for managing a set of validators, distributing rewards, and reporting and penalizing malicious validators. The authors provide a reference POSDAO implementation, xDai POSDAO, which uses xDai as a stable transactional coin and a representative ERC677 token (STAKE) as a staking token. The reference implementation functions on an Ethereum 1.0 sidechain and utilizes the AuRa consensus protocol. Assets are bridged between the Ethereum mainnet and the xDai POSDAO network using several instances of the POA TokenBridge.

Table of Contents

Note on the paper layout: This paper consists of two distinct parts. The first part (sections 1-8) offers an overview of existing models, explains the algorithm's reward structure, validator set formation, rationale, and consensus fault management strategies. The second part (section 9) describes the reference implementation in detail, with many overlapping concepts. Readers interested in specific implementation details may want to skip to section 9.

1. Introduction

While Ethereum 2.0 (Serenity) will bring Proof of Stake Sybil control and many other innovations to the Ethereum ecosystem, the completion date is currently unknown. Following completion, Ethereum 1.0 will continue to provide a viable and usable network, requiring long term support and optimization. The POSDAO (Proof of Stake Decentralized Autonomous Organization) protocol provides an immediately available scalability solution for Ethereum 1.0, creating the opportunity for staking and delegated staking, very fast transactions, and very low transactional costs. POSDAO runs on a fully compatible EVM-based sidechain, allowing for mainnet interoperability while providing greater efficiency, lower fees, configurability, and other benefits relative to current EVM consensus implementations.

In addition to slow and costly transactions, there is increasing evidence that Nakamoto consensus (also known as Proof of Work) models are not ecologically viable in the long term. Bitcoin currently uses “at least 2.55 gigawatts of electricity”, with the potential to consume “7.67 gigawatts in the future”, comparable to the total usage of countries like “Austria (8.2 gigawatts)[1].” As a result, more blockchain networks are adopting Proof of Stake (POS) and Delegated Proof of Stake (DPOS) protocol variants as consensus alternatives[2]. These protocols are responsible for designating the network nodes that process transactions and update the ledger in a distributed system. They have been shown to provide the requisite security and consensus for a blockchain, and offer improved efficiency over current Nakamoto implementations[3]. In POSDAO, the POS algorithm is implemented as a decentralized autonomous organization (DAO).

Participants in POS protocols stake assets, in the form of tokens or coins, to protect the network and achieve agreement regarding blockchain transactions. There are many projects within the Ethereum ecosystem where project specific tokens are held by project supporters, however, these assets have limited utility. By converting project specific tokens to DPOS staking tokens, token holders can participate as validators or delegators in the consensus process on an Ethereum-based sidechain. They earn rewards (either block rewards or transactional rewards) based on their participation. This provides an opportunity for token holders to convert any amount of current holdings into staking tokens, which in turn earn reward-based dividends.

While many POS and DPOS implementations are currently available, they often have set criteria which determine their base functionality and limit their potential usage. The parameters in the POSDAO chain are highly configurable. This includes the underlying consensus protocol, block reward functionality, transaction rate, staking specifications, and other implementation details. The reference implementation provides settings and parameters which can be changed depending on the purpose of the chain and the needs of its users.

1.1 Proof of Stake model

Consensus algorithms provide an economic incentive for honest protocol execution and decentralization. In a Proof of Stake (POS) model[4], individuals stake an amount of tokens in an effort to be selected as a block producer (validator). Validators are chosen based on numerous criteria; typically the amount of stake and a randomness beacon are used in the selection process. In exchange for successful block creation, validators are rewarded with additional tokens.

A main advantage of a POS Sybil resistant system is the reduced energy expenditure in comparison to a Proof of Work (POW) model. Additionally, POS incentivizes validators to run high-bandwidth nodes and backup nodes for redundancy, which improves network throughput. POW, on the other hand, incentivizes the purchase of more mining hardware, which does not improve the maximum throughput.

There are two major POS models: *Chain-based proof of stake,* which "[..] features a chain of blocks and simulates mining by pseudorandomly assigning the right to create new blocks to stakeholders, and *Byzantine Fault Tolerant (BFT) proof of stake,* where an existing BFT consensus is repurposed for Sybil control and economic incentive models"[5]. POSDAO utilizes smart contracts for the validator selection process. These validator sets then run the underlying consensus protocol. It can be configured to use OpenEthereum's AuRa consensus engine[6] or Honey Badger BFT[7].

1.2 Delegated Proof of Stake

Delegated Proof of Stake (DPOS)[8] extends the POS model to allow additional individuals to stake their tokens on potential validators (candidates), without participating in block production themselves. Candidates who collect a higher percentage of tokens have greater odds of becoming validators on the network. Rewards are then divided amongst the validators and the staking entities (delegators).

DPOS provides the opportunity for delegators to "vote" on potential validators by staking tokens on them. Candidates are incentivized to maintain a good reputation in order to attract more delegators and increase their chances of becoming validators. POSDAO is designed to support a DPOS model.

1.3 Decentralized Autonomous Organization (DAO)

A DAO is a self-sustaining, virtual entity defined by "smart contracts that contain the assets and encode the bylaws of an entire organization"[9]. All financial transactions, rules, and decisions are enacted and stored on the blockchain, creating a transparent and verifiable record. Rules are initially set forth in smart contracts, and members (participating token holders) interact according to these regulations to further the goals of the organization. Organizational rules can be modified through mechanisms contained in the on-chain contracts or through an off-chain governance process, such as subjective resolution and/or software updates.

1.4 POSDAO consensus model

POSDAO consensus implements a layered POS model connected by smart contracts on a public blockchain (see Figure 1 below). Sybil control and incentives exist in smart contracts working within the EVM and the execution state is stored on a public chain implementing the POSDAO consensus. The underlying BFT consensus exists on the network protocol level. This model requires modifications to the BFT consensus algorithm implementations in the Ethereum client to facilitate information exchange between smart contracts and the consensus layer. This includes communication relays regarding consensus faults and validator set management. The AuRa implementation is implemented and operational on the OpenEthereum client. It is also operational on the Nethermind Client v1.10.71+, with plans to migrate to a Nethermind powered network with the upcoming deprecation of OpenEthereum support. HBBFT may also be implemented in a future release.

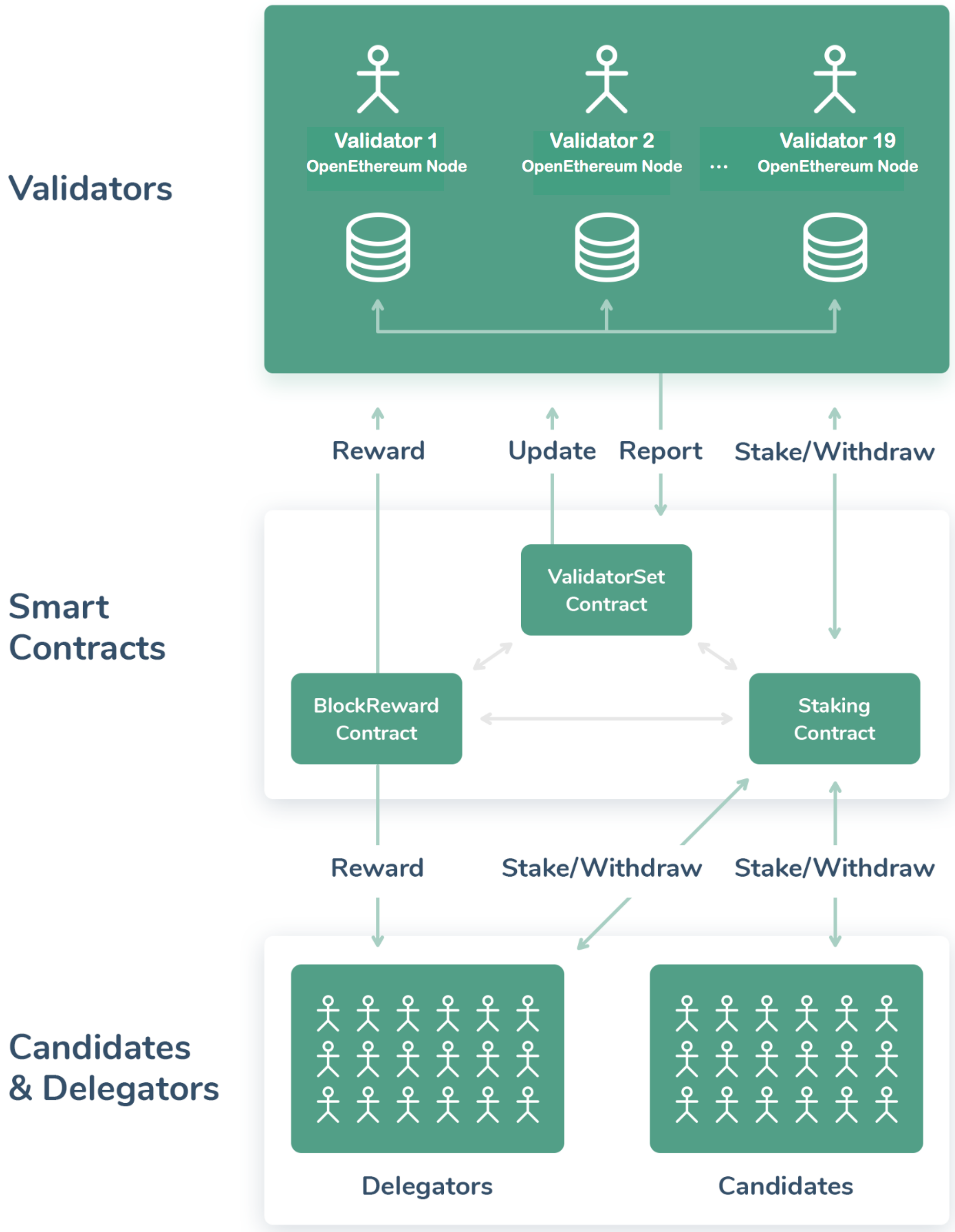


Figure 1: Interactions between participants and primary consensus contracts in POSDAO.

2. Terminology

Term	Math notation	Definition
Staking unit	$1\mu \in M$ M = {ETH, DAI, POA...}	A token denomination used by actors of the algorithm for staking. Actors include candidates, validators, and delegators.
Minimum candidate stake	$st^{C}_{min} = a\mu$ a − number of staking units	The minimum amount of staking units required to participate as a candidate for a validator slot.
Candidate stake	$st^{C} = a\mu$ $st^{C} \geq st^{C}_{min}$ a − number of staking units	The amount of staking units a given candidate has staked on itself. If the candidate is selected as a validator, the candidate stake is referred to as *validator stake* without changing the notation.
Minimum delegator stake	$st^{D}_{min} = a\mu$ a −number of staking units	The minimum amount of staking units required to participate in a pool as a delegator.
Delegator stake	$st^{D}_{i} = a\mu$ $st^{D}_{i} \geq st^{D}_{min}$ a − number of staking units	The amount of staking units the i-th delegator contributed to a given pool.
Candidate		An Ethereum address which deposited at least the minimum candidate stake to form a new pool.
Delegator		An Ethereum address that allocates staking units to a candidate's/validator's pool. Delegators and the candidate/validator together form liquidity of the pool.
Pool	P_{st} $P_{st} = \sum_{i=1}^{n} st^{D}_{i} + st^{C}$	The total sum of staking units allocated to a candidate/validator. This includes all delegators' staking units as well as units staked by the

		candidate/validator. The proportion of stake is used to determine pool reward distribution.
Validator		A candidate selected by the algorithm to participate in a validator set for a staking epoch.
Validator set		The set of validators selected for a staking epoch to keep consensus of a network. Each validator represents the validator's pool.
Initial validators		The set of validators defined during network initialization.
Staking epoch	t	The time duration (in blocks for AuRa, time in HBBFT) for which the validator set is selected. For example this is set to 120992 for a one-week timeframe with a 5 second block time (for AuRa).
Pool reward	P_{rw} $P_{rw} = BR * \frac{1}{j}$ $= \sum_{i=1}^{n} r_i^D + r^V$ j −number of validators n – number of delegators r^V −validator's reward r^D −delegator's reward	The block reward in reward tokens for a validator and delegators within a pool. The distribution between them is defined in 3. Reward distribution.
Block reward	BR	The total number of reward tokens appropriated per block (or per amount of time for consensus algorithms like Honey Badger BFT). The BR is distributed amongst all participating validator pools.
Reward token	T	An ERC677 or Native token used to reward validators and delegators.

3. Reward distribution

Reward distribution is the primary incentivization mechanism of the algorithm. In this section, we explain the reward distribution rules between validators and delegators within pools based on their contribution amounts.

POSDAO provides the option to implement a dual token environment. This is not a requirement of the algorithm, but allows the opportunity to explore various economic incentives. In the reference implementation, the POSDAO ERC677 token is used for staking. It is also used to provide token rewards from the ERC677-to-ERC677 bridge (entrance/exit fees), and block rewards. A second token (a native stable coin) is used for transaction fees (paid to validators or external contract only, not to delegators) and rewards from the ERC20-to-Native bridge (entrance/exit fees). The POSDAO implementation can also be configured using a single token where the network's native coin is used both for staking and rewards.

Block rewards can be configured according to the requirements of the implementation. In our single token reference implementation, the block reward is paid via a 2.5% annual inflationary measure applied to the native staking coin.

In our dual token implementation, an annual inflationary measure is applied only to the POSDAO staking token according to the following formula:

```
newTokensPerStakingEpoch = totalStakeAmount * ratio / 4800
```

where:

- `newTokensPerStakingEpoch` is the number of tokens minted and distributed among the pools during the current staking epoch;
- `totalStakeAmount` is the total sum of tokens staked on the current active validators (doesn't include the total amount of staked tokens during the current staking epoch);
- `ratio` is an emission ratio which is set to 15 representing 15% Annual Percentage Rate for staking token with a dual token implementation or 2.5 representing 2.5% APR for native coins with a single token implementation.

Additional rewards may be implemented with transaction fees and entrance/exit fees (for validators and their delegators) on the bridges (see 5.1).

3.1 Configurable reward structure

The following reward types may be implemented with the POSDAO algorithm.

3.1.1 Transaction fees

Each on-chain transaction is assessed a transaction (gas) fee. This fee is configurable, and the default is very low. The validator that seals the block (in the AuRa implementation) collects any transaction fees associated with that block. In HBBFT, fees will be split equally among the validators. This reward is not distributed to the delegators in the pool, it is only awarded to validators. While the initial implementation directs transaction fees to validators, transaction fees may also be designated for other purposes. With the implementation of EIP-1559, the fee structure will be updated and fees may be burned or re-allocated in another way.

3.1.2 Bridge fees

If a bridge is used in the implementation, bridge fees may be assessed when assets are transferred between chains (see 5.1 for details). An entrance fee is charged when assets are moved from the Ethereum mainnet to the sidechain, and an exit fee assessed when the asset is moved back to the mainnet. The fees are distributed to validators and delegators based on their staking ratios (see 3.2).

3.1.3 Fixed block rewards

The reward distribution function mints reward tokens (or native coins depending on the network settings and bridge mode) for all active validators and their delegators. If a validator is removed due to misbehavior, its pool is not included in the reward distribution.

To calculate block rewards, the total stake amount (calculated from the beginning of the staking epoch) is multiplied by a constant (inflation rate) and distributed among validators and their delegators. See 9.2.7 for more details based on our reference implementation.

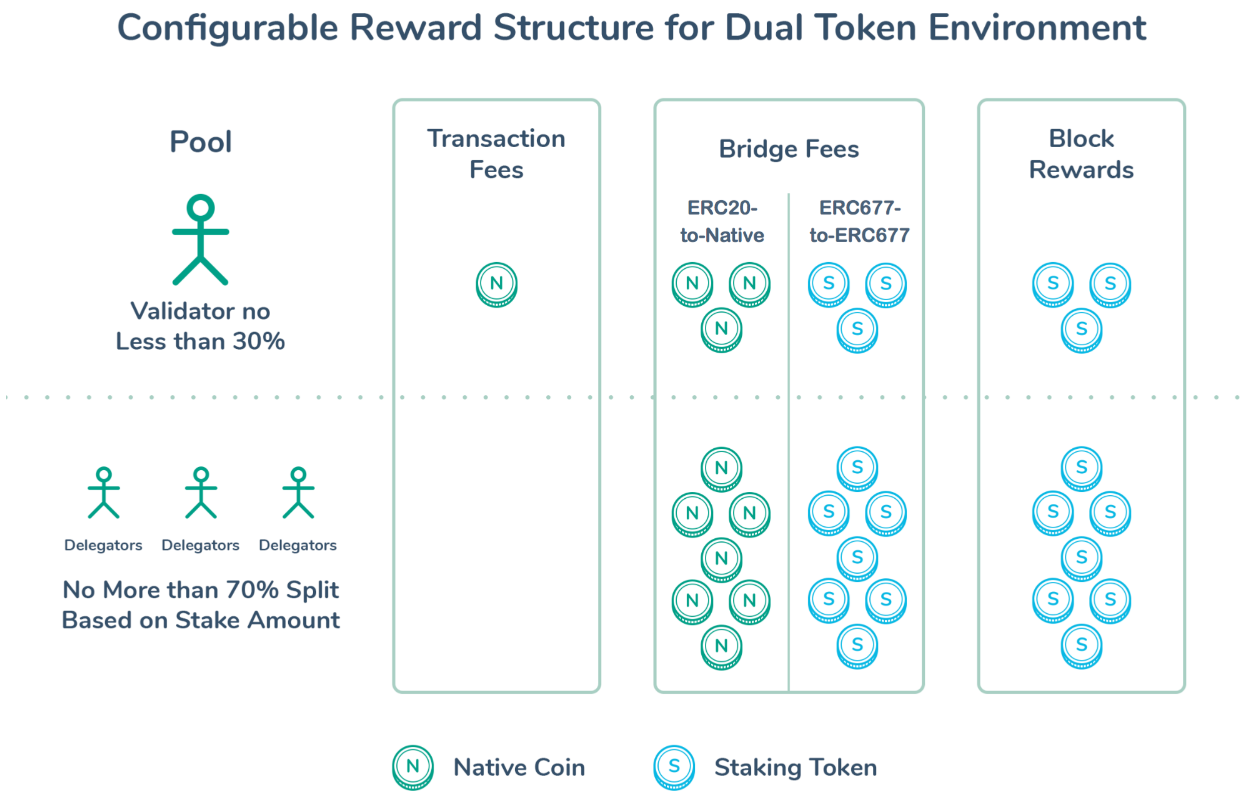


Figure 2: In this example, transaction fees are awarded to the validator only, and bridge fees and block rewards are distributed between the node validator and the associated delegators.

3.2 Three rules of reward distribution

In order to maintain fairness and incentivize elected validators, reward distribution is calculated according to the following rules:

1. Each pool within the validator set receives an equal share of the reward (if all validators always produce blocks and don't skip them) at the end of staking epoch.
2. Pool rewards are proportionally distributed between a validator and the staking delegators, as long as the total delegators' percentage of stake is below 70%*.
3. The validator is guaranteed to receive at least 30% of the pool reward. If the total delegators' stake exceeds 70%, the delegators' rewards are adjusted accordingly and the validator receives 30%*.
   **Note: This parameter is configurable, and we will conduct statistical analysis to determine its efficacy in our reference implementation. It may also be adjusted to a flat percentage based on proportional staking amounts irregardless of validator/delegator status.*

See Appendix A for a detailed example which uses the 70/30 rule.

3.3 Distribution examples

3.3.1 Pools receive an equal reward per block

Let's introduce a pool reward per block. We assume it is defined on network startup.

Block reward	1 reward token

If there is one validator in a validator set per a given staking epoch, that validator's pool will receive 100% of the block reward.

ID	Pool reward
Validator A	100% of block reward

Let's assume the minimum candidate stake is one staking token and the minimum delegation stake is 0.01 staking tokens.

Minimum candidate stake	1 staking token

Minimum delegation stake	0.01 staking tokens

If there are two validators in a validator set per a given staking epoch, even if their pools have a different amount of staking tokens ($P_{st\,A} \neq P_{st\,B}$), the pools will each receive 50% of the block reward.

ID	Number of staking tokens	Pool reward
Validator A	1	50% of block reward
Validator B	2	50% of block reward

Note: shares will only be equal if every validator produces blocks continuously. If a validator skips their blocks, their pool will receive proportionally less reward than other (continuously working) validators. For example, if there are two validators in the validator set and one of them produced 10 blocks, but another only 5 blocks, the first validator's pool will receive 10/(10+5)=66% of the total reward, and the second pool will receive the remaining 34%.

3.3.2 Validator shares the reward with delegators proportionally within a pool.

When a share of the total amount staked by delegators is less than or equal to 70% of a pool, the reward is distributed proportionally among participants.

$$\frac{\sum_{i=1}^{n} st_i^D}{P_{st}} \leq 0.7$$

$$r_i^D = P_{rw} * \frac{st_i^D}{P_{st}} \qquad r^V = P_{rw} * \frac{st^C}{P_{st}}$$

3.3.2.1 Case with no delegators

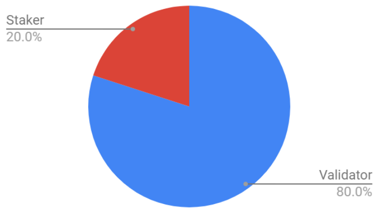
A validator owns 100% of the pool reward.

	Staking tokens	Pool reward, %

Dist.1: Reward distribution for a single validator without delegators	1	100
Blue validator	1	100

3.3.2.2 Case with one delegator and one validator

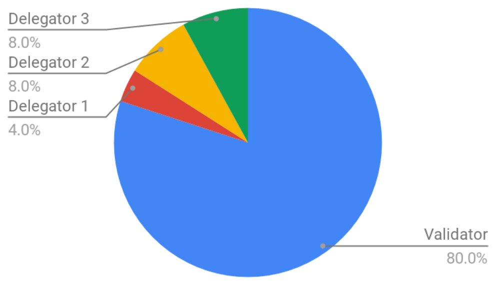
The delegator stakes an amount less than 70% of the total amount staked within a pool. The delegator shares the pool reward proportionally with the validator.

	Staking tokens	Pool reward, %
 **Dist.2**: Reward distribution for a single validator with a single delegator when the stake of the delegator is less than 70% of a pool	1.25	100
Blue validator	1	80
Red delegator	0.25	20

3.3.2.3 Case with multiple delegators

In this case, the sum of all delegators' stakes is less than 70% of the total amount staked within a pool. The group shares the reward proportionally with the validator.

	Staking tokens	Pool reward, %

 **Dist.3**: Pool reward distribution for a single validator with multiple delegators when the sum of all delegators' stake is less than 70% of a pool.	1.25	100
Blue validator	1	80
Red delegator 1	0.05	4
Green delegator 2	0.1	8
Yellow delegator 3	0.1	8

3.3.3 Validator shares a reward with delegators with proportion 30% to 70% within a pool*.

**this proportion is configurable and may be changed to 20/80 or a flat rate which corresponds directly to the amount staked.*

When the total amount staked by delegators exceeds 70% of a pool, the delegators' reward share is capped at 70%.

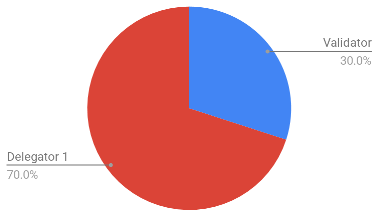
$$\frac{\sum_{i=1}^{n} st_i^D}{P_{st}} > 0.7$$

The validator's reward never drops below 30% within a pool. This ensures nodes are incentivized to be validators themselves, rather than merely delegate.

$$r_i^D = 70\% * P_{rw} * \frac{st_i^D}{\sum_{i=1}^{n} st_i^D} \qquad r^V = 30\% * P_{rw}$$

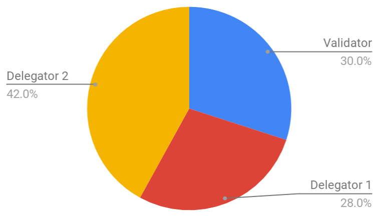
3.3.3.1 Case with one delegator

One delegator staked more than 70% of a pool. The pool reward of the validator remains at 30%. The delegator receives 70% of the pool reward.

	Staking tokens	Pool reward, %
 **Dist.4**: Reward distribution for a single validator with a single delegator when the stake of the delegator is more than 70% of a pool	4	100
Blue validator	1	30
Red delegator	3	70

3.3.3.2 Case with multiple delegators

Multiple delegators staked amounts totaling more than 70% of a pool. The validator's pool reward will not drop below 30%. The delegators proportionally share the 70% pool reward.

	Staking tokens	Pool reward, %
 **Dist.5**: Pool reward distribution for a single validator with multiple	6	100

delegators when the sum of stakes of all delegators is more than 70% of a pool		
Blue validator	1	30
Red delegator 1	2	28
Yellow delegator 2	3	42

4. Validator set formation

Any address with the minimum required candidate stake can become a validator. When an address calls the `addPool` contract function and meets the minimum required candidate stake, it becomes a candidate and forms a new pool.

4.1 Network participants

The number of network participants is configurable and cannot exceed the values assigned to the MAX_CANDIDATES and MAX_VALIDATORS parameters.

Any arbitrary address with at least DELEGATOR_MIN_STAKE tokens (ERC677) or native coins (depending on network's mode) can stake their tokens/coins and become a delegator.

4.2 Staking epochs

The network's operation is divided into staking epochs (STAKING_EPOCH_PERIOD - this value is configurable, the default is a one week cycle). A new staking epoch begins immediately following the termination of the previous epoch.

At the beginning of each staking epoch, the algorithm selects a new validator set from the current list of candidates and creates a snapshot of the current state of the validators' pools. If there are fewer than MAX_VALIDATORS+1 candidates, every candidate becomes a validator. The snapshot is used to calculate the reward amount for validators and delegators when they claim the reward.

4.3 Becoming a candidate

An arbitrary address A in the network launches its node and puts at least the minimum stake in ERC677 staking tokens or native coins (CANDIDATE_MIN_STAKE) on its own address. Address A then specifies address B as the *mining address* using the `addPool` function. A new active pool is created for address A, and this account becomes a candidate account.

- Address A is the staking address used to collect rewards and place stakes into their own pool.
- Address B (the *mining address*) is used by the validator's node to sign blocks, participate in the randomness beacon (see 4.7), and report on malicious validators. This address is defined in the `engine_signer` config option of validator's OpenEthereum node.

4.4 Candidate pools

At the beginning of each staking epoch, the algorithm selects active candidate pools to participate as validators in the next validator set. Inactive pools are ignored.

If a candidate withdraws all of their tokens/coins from their pool, the pool becomes inactive and does not take part in the next validator selection process. The candidate can either fully withdraw their tokens/coins and remove themselves as a pool, or partially remove their tokens/coins (provided that they leave CANDIDATE_MIN_STAKE) and participate in later staking epochs.

4.5 Staking and withdrawal to/from a pool

Participants can stake or withdraw their tokens/coins to or from pools during the majority of a staking epoch. The exception is a defined period at the end of each epoch (STAKE_WITHDRAW_DISALLOW_PERIOD). This measure prevents stake manipulation based on the random seed value generated at the very end of the epoch (see 4.7).

- The total stake amount of a candidate or validator on their own pool cannot be less than CANDIDATE_MIN_STAKE.
- The total stake amount of a delegator on any pool cannot be less than DELEGATOR_MIN_STAKE.

The minimum candidate stake is relatively large, encouraging institutional investors to become candidates and validators. This large stake creates additional incentives for candidates to protect their nodes and prevent DoS attacks. However, DoS attacks on individual validator nodes are still possible. Part of the validator's job is to defend against such attacks by using ISPs that provide DoS protection.

Additional token staking or withdrawal to/from a pool during the current staking epoch (and thereby changing the size of the pool) does not impact the current pool reward. The reward is determined based on the pool's state at the moment the staking epoch begins (see 4.2). However, these changes will impact validator selection probability for the following staking epoch.

A participant cannot withdraw their tokens/coins from an active validator's pool unless the amount was staked during the current staking epoch (this amount hasn't been allotted as a stake yet, so it can be withdrawn). Tokens/coins can be withdrawn from a candidate's pool at any time (because the candidate is not a validator).

If a participant (delegator or validator) wants to leave an active validator's pool or reduce their staked amount, they can schedule a withdrawal from the pool. The selected amount can be claimed after the current staking epoch is complete.

If a validator wants to terminate their validator status on the next staking epoch, they can schedule a withdrawal of their staked amount or call the contract's `removeMyPool` function (in this case the pool becomes inactive and won't be selected by the algorithm at the beginning of the next staking epoch).

4.6 Moving stakes

A participant (delegator or candidate) can move their full or partial stake amount from one pool to another without withdrawing the amount from the contract. Such a move is subject to the same withdrawal rules described above.

4.7 Randomness when selecting a validator

The protocol implements a random number generator similar to RANDAO, which is used to randomly select a set of validators from the group of candidates at the start of each staking epoch. Candidates with a larger pool have a higher probability of selection to a validator set for each staking epoch (candidates with higher stakes are probabilistically selected as validators for more staking epochs).

Note: random selection is not applicable if there are 19 or fewer candidates/validators.

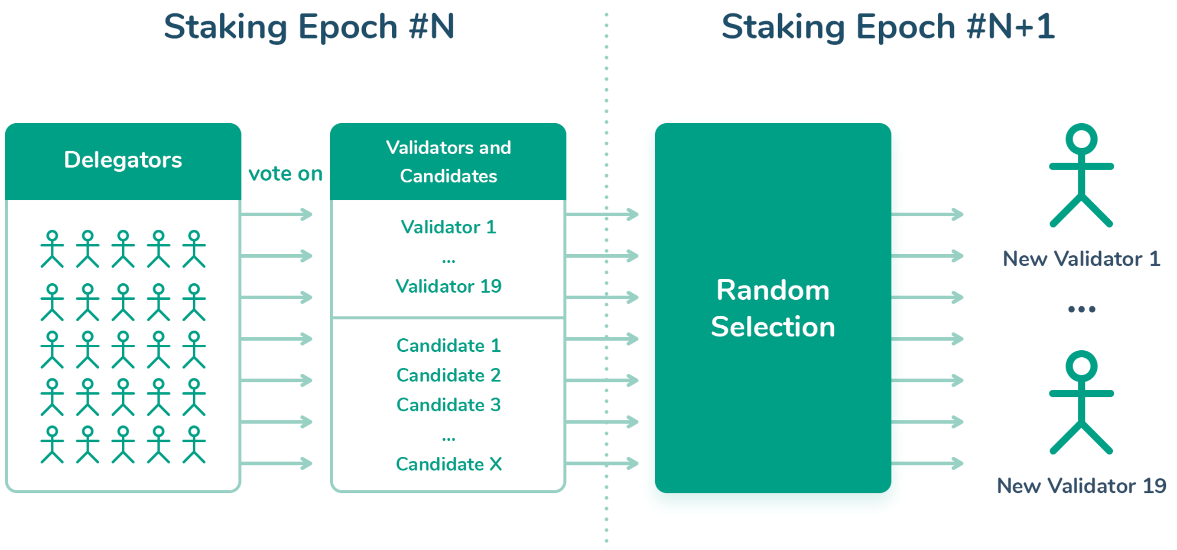


Figure 3: Validator set selection is determined by the validator/candidate pool size and a random value.

5. POSDAO network initialization

POSDAO smart contracts can be initialized either in the genesis block or in an arbitrary block on an already existing network. In the case of genesis initialization, the contracts' pre-configured parameters are included in the chain specification bytecode, and the set of initial validators is also defined in these parameters.

If the network is initiated from the genesis block, all of the addresses in the network (including initial validators) have zero balances. There are no pre-initialized stakes for initial validators, so their pools are also empty.

POSDAO may be configured to run as a stand alone blockchain. It can also run using a bridge or bridges which connect to one or more other networks. This bridged scenario is used in the reference implementation and described below.

5.1 Bridged network scenario

The POA TokenBridge is used to connect Ethereum chain instances, allowing users to transfer assets between chains. In the reference implementation, two POA TokenBridge instances connect the POSDAO sidechain network to the Ethereum mainnet.

Both bridges have their own validator sets which are not bound with the consensus validator set in the POSDAO network. Bridge validators are responsible for secure token transfer between chains, and they do not receive any reward for this service.

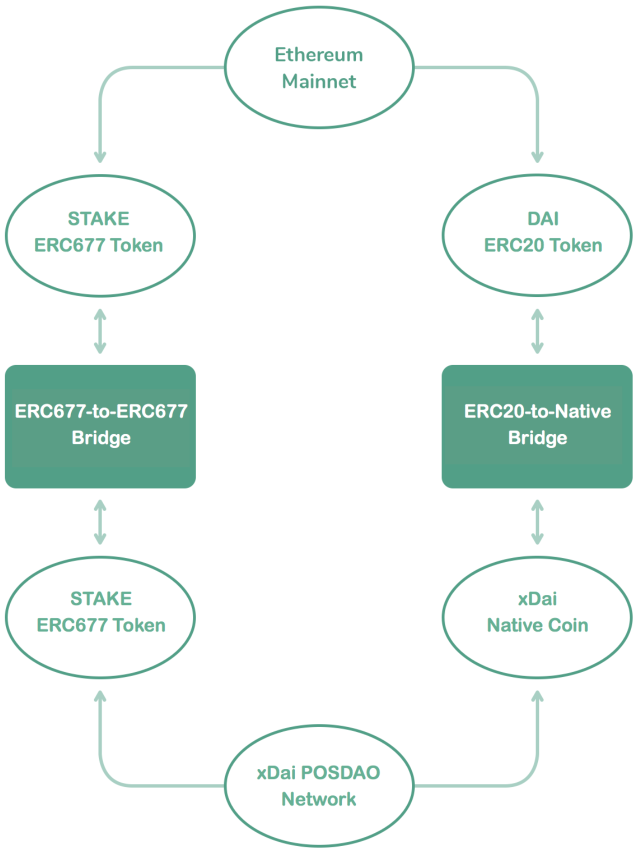


Figure 4: Reference implementation bridges between the Ethereum mainnet and the xDai POSDAO network.

5.1.1 Bridge 1: ERC677-to-ERC677 bridge

The first bridge instance operates in the ERC677-to-ERC677 mode, connecting the POSDAO network with the Ethereum mainnet. This allows participants to bridge their staking tokens (POSDAO ERC677 tokens in the reference implementation) from the mainnet to the POSDAO network (and move them back from POSDAO to mainnet when needed). See Appendix C for an example scenario.

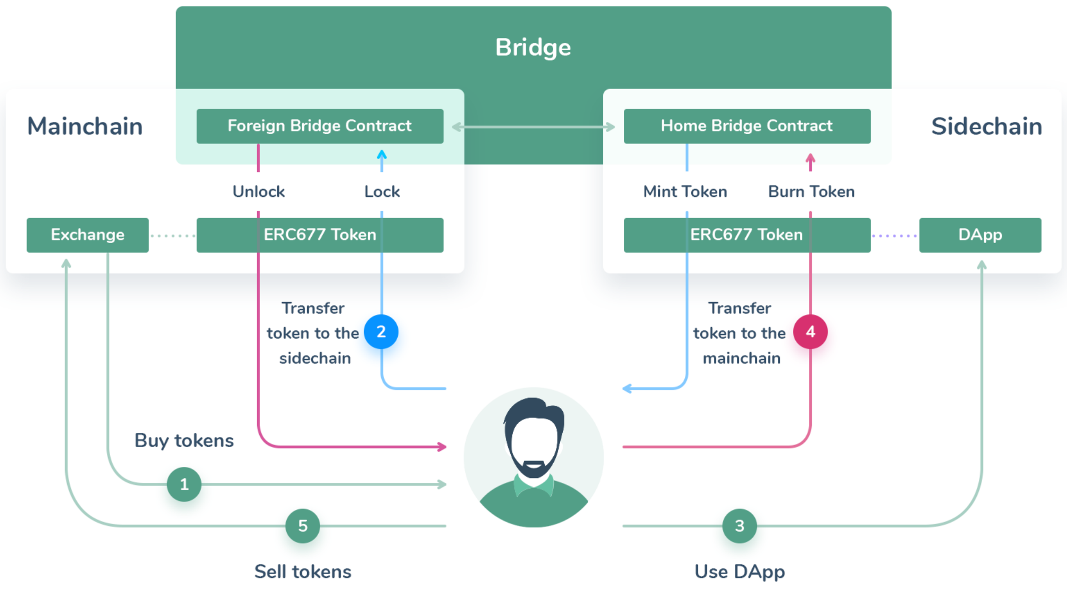
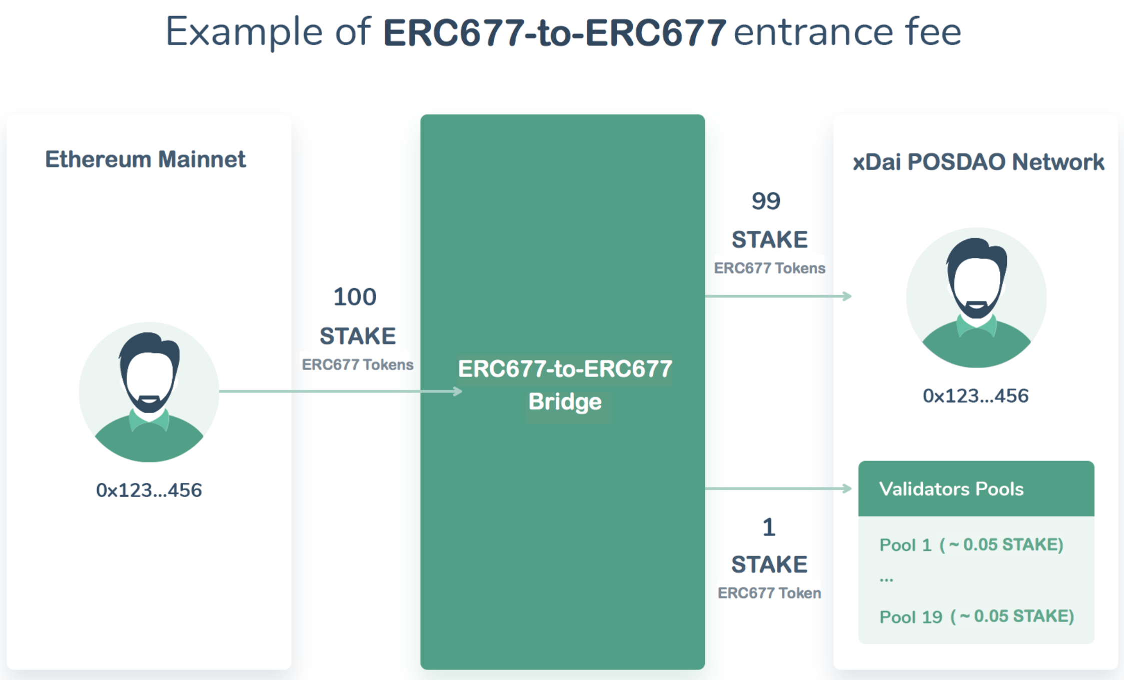


Figure 5: Token transfer between chains using the ERC677-to-ERC677 bridge. Note the ERC677 is an ERC20 extension which allows for easy transfer operations.

5.1.1.1 ERC677-to-ERC677 bridge entrance and exit fees

Entrance and exit fees are assessed when tokens are moved between the two networks. These funds are distributed to validators' pools to provide staking incentives. The values are configurable (see the BRIDGE_ENTRANCE_FEE and BRIDGE_EXIT_FEE constants in 9.4), in this example it is set to 1% of the total bridge transaction token value.



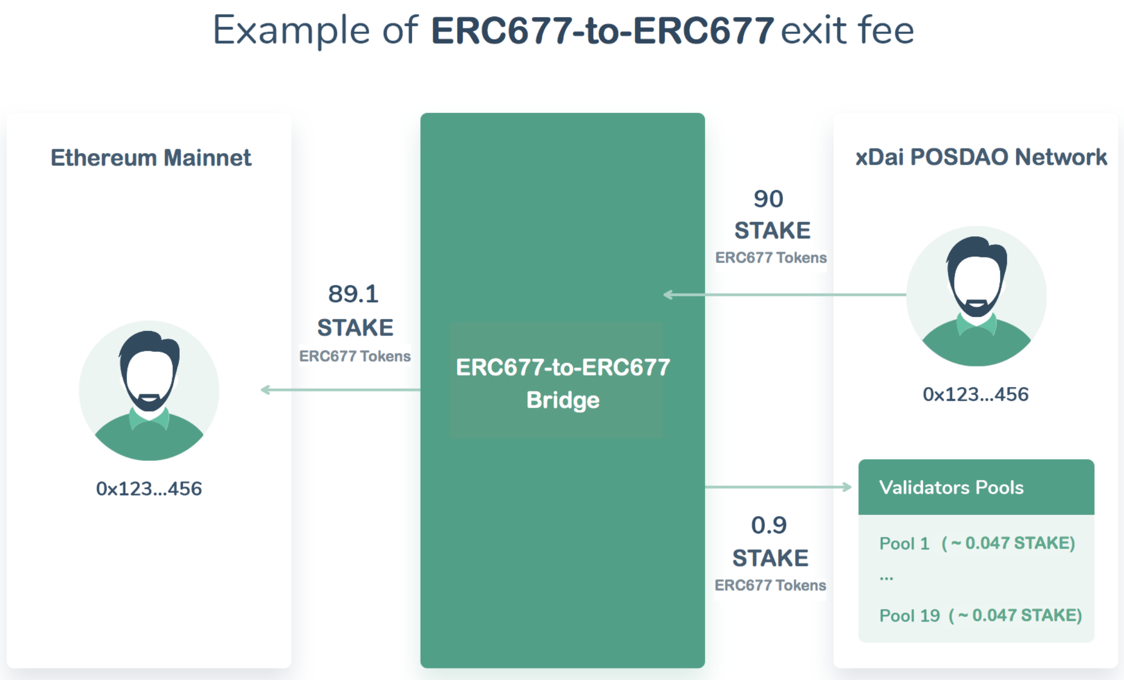


Figure 6-7: Bridge entrance and exit fees are distributed to active validator pools.

5.1.2 Bridge 2: ERC20-to-Native bridge

The second bridge instance operates using an ERC20-to-Native mode. In this scenario, participants bridge their ERC20 tokens from the Ethereum mainnet to the POSDAO enabled sidechain (and move the coins back to mainnet as needed). In the reference implementation we use DAI as the ERC20 token and xDai as the Native token.

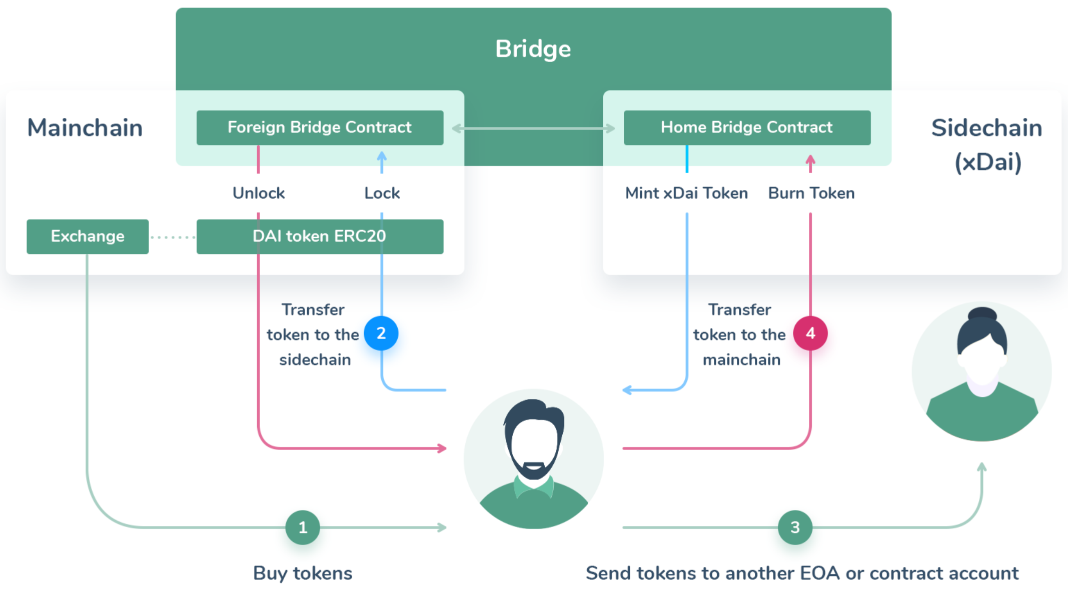
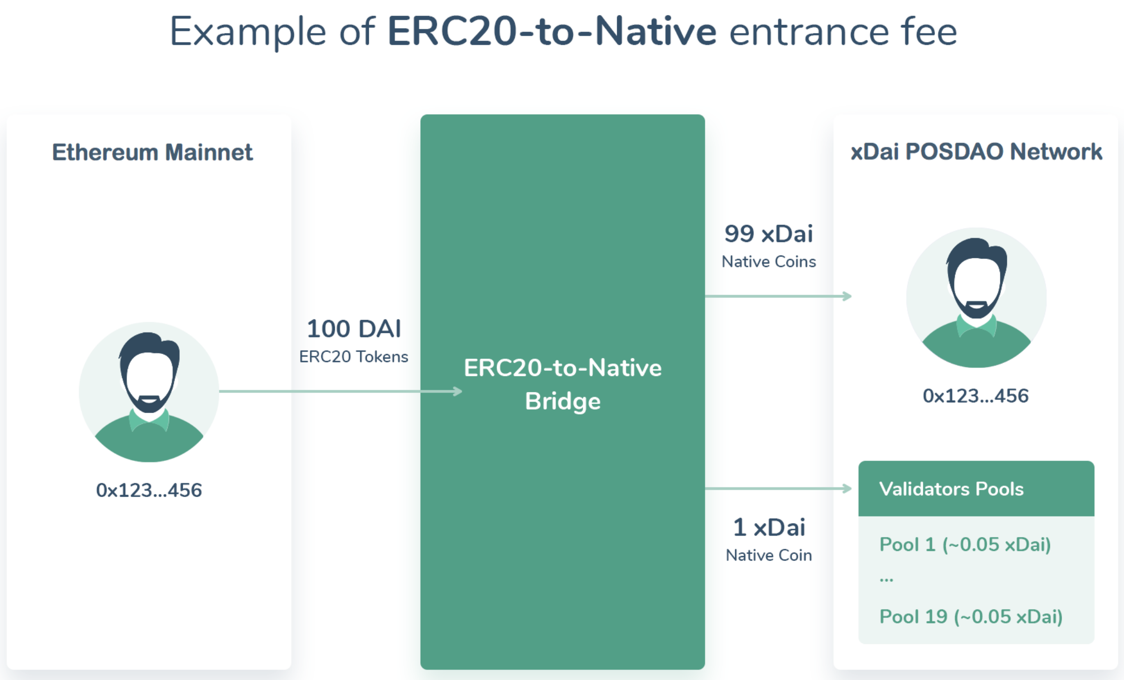


Figure 8: Token transfer between chains using the ERC20-to-Native bridge.

5.1.2.1 ERC20-to-Native bridge entrance and exit fees

Entrance and exit fees are also assessed when tokens are locked/unlocked and minted/burned between the two networks. These funds are also distributed to validators' pools to provide staking incentives. The value is configurable, in this example it is set to 1% of the token-coin transfer amount. See Appendix B for an example scenario.



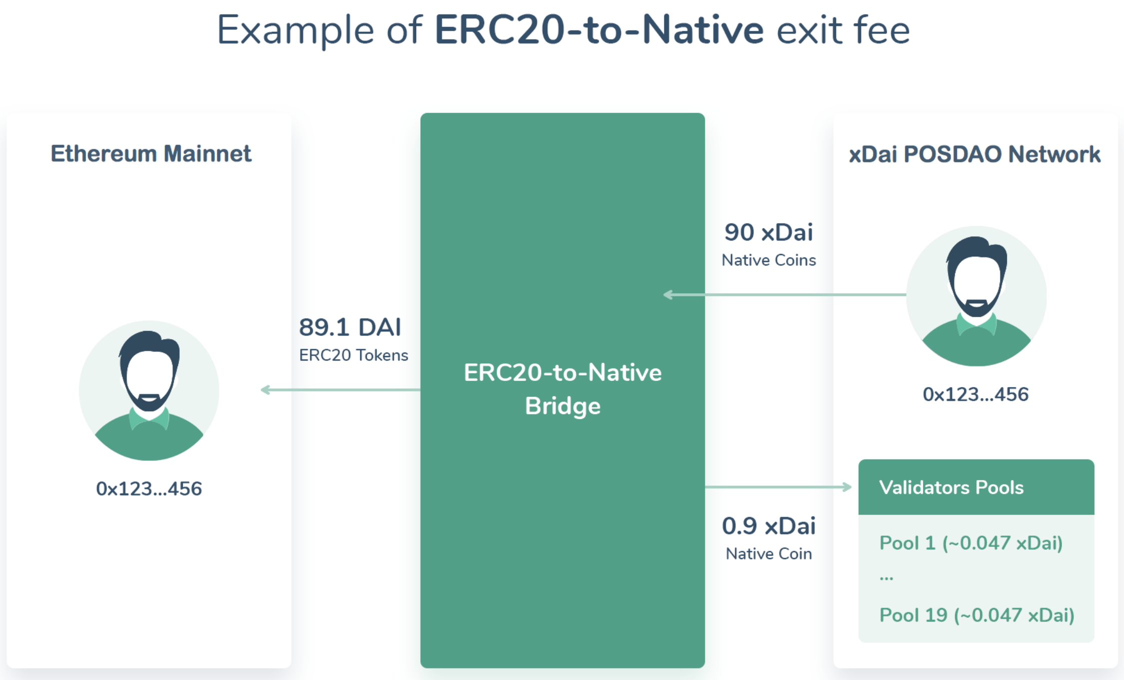


Figure 9-10: Bridge entrance and exit fees are distributed to active validator pools.

5.1.3 Bridge setup and initial validators

After the POSDAO network is started, the ERC677-to-ERC677 bridge is connected and initialized. Once the bridge is connected, the initial set of consensus validators (defined in the `ValidatorSetAuRa` smart contract) can bridge their POSDAO ERC677 tokens from the Ethereum mainnet to the POSDAO network and place stakes into their own pools.

Because the validators do not have native coins (xDai in our example implementation) when the network starts from genesis, they can make service transactions to the POSDAO contracts using zero gas. The validators can make unlimited service transactions but only within the scope of the consensus contracts. The `TxPermission` smart contract protects against possible spam sent from a validator. Figure 11 shows an example network initialization.

If an initial staking epoch ends and there are no candidates (none of the initial validators made a stake into their pool), the initial validator set is retained for the following staking epoch. However, if at least one candidate appears (the address which added a stake to its pool), any initial validators with empty pools are removed from the set, and the candidate becomes a validator on the new staking epoch. Thus, if an initial validator wants to keep their seat (and still has no staked tokens) after the initial staking epoch, they must place a stake into their own pool.

After the bridge is connected, individuals can bridge their POSDAO tokens and become candidates in the POSDAO network. Figure 12 illustrates the various possible interactions from a candidate's address.

If the number of candidates is greater than MAX_VALIDATORS at the beginning of new staking epoch, the validators are chosen randomly (using the randomness beacon) from the set of candidates: the larger the candidate's pool, the higher probability the candidate has of becoming a validator.

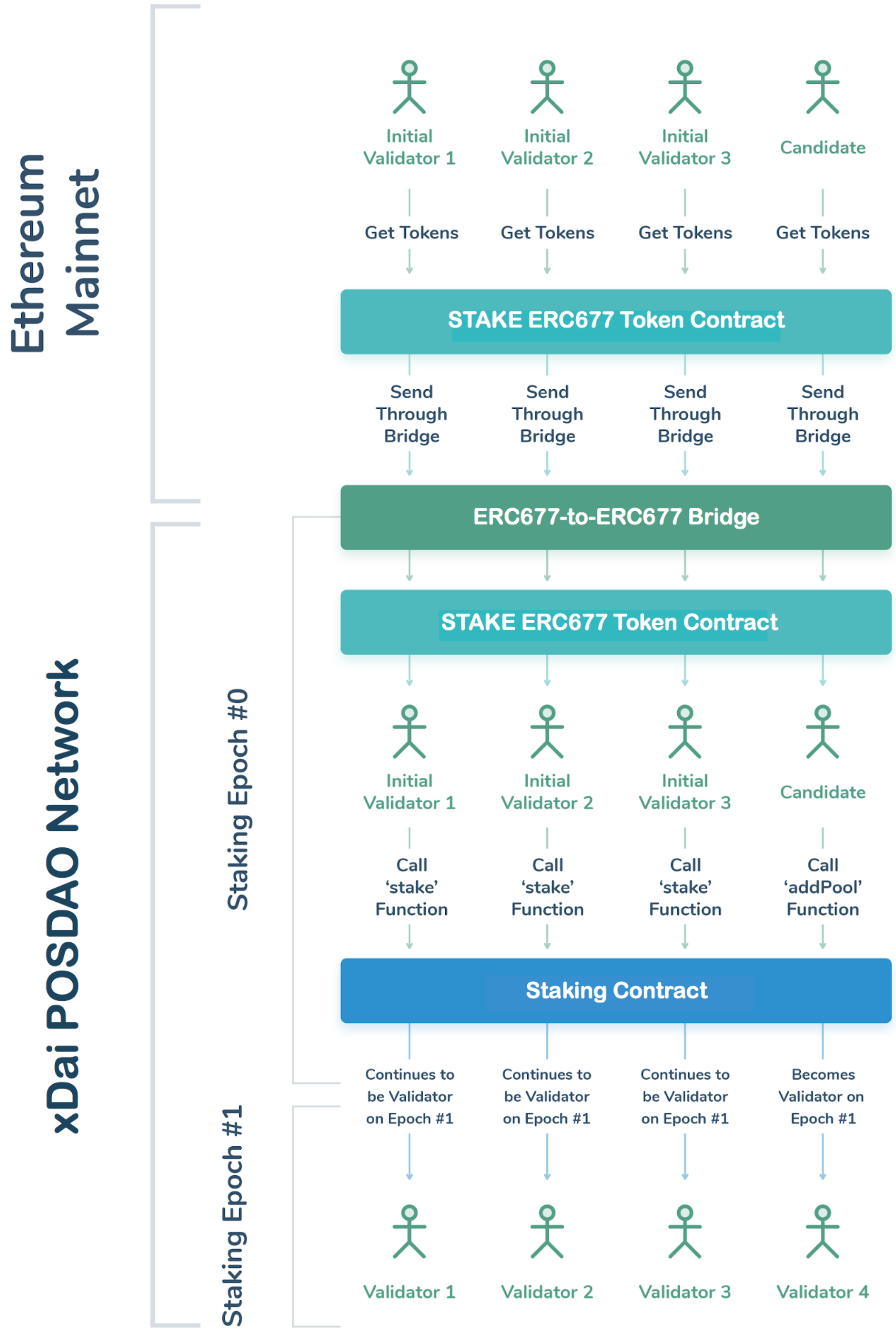


Figure 11: Initialization of three initial validators and one candidate when starting from genesis (random is not used here because the number of initial validators + candidates ≤ 19).

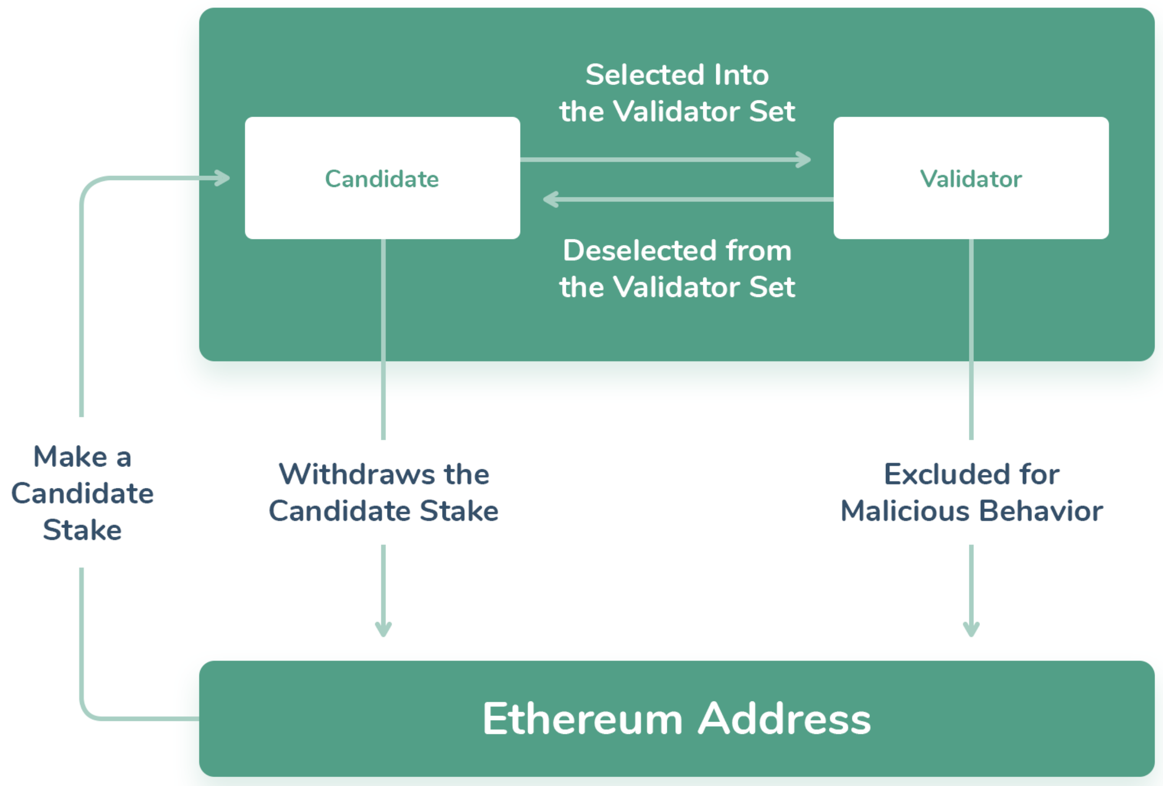

Selected Into the Validator Set
Candidate
Validator
Deselected from the Validator Set
Make a Candidate Stake
Withdraws the Candidate Stake
Excluded for Malicious Behavior
Ethereum Address

Figure 12: Candidate interactions from a single Ethereum address.

5.2 Initialization parameters

When a new network starts the initial validator set is populated.

Example initial parameters at network start:

Number of candidates	30
Number of validators	4
Number of delegators	0
Initial validator 1 (mining address)	0x...
Initial validator 1 (staking address)	0x...
Initial validator 2 (mining address)	0x...

Initial validator 2 (staking address)	0x...
Initial validator 3 (mining address)	0x...
Initial validator 3 (staking address)	0x...
Initial validator 4 (mining address)	0x...
Initial validator 4 (staking address)	0x...
Initial owner	0x...

In addition to the initial validators, an owner deploys the bridge contracts and has the ability to upgrade the bridge and consensus contracts when required (for example if bugs are found or the code needs to be modified). The owner should be a MultiSig smart contract which requires a trusted setup. The details of this setup can vary depending on the nature of the network. Future implementations may include a voting mechanism to allow validators the ability to vote on upgrades. This would alleviate the need for the owner contract.

5.2.1 Recommendations

When configuring the number of validators, the underlying consensus algorithm should be taken into account. For example, both our AuRa variant and HBBFT can tolerate up to and excluding one third of the validators being faulty. This means that 19 validators are more secure than 20, because in both cases, up to 6 faulty ones can be tolerated, but 7 out of 20 being faulty is more likely than 7 out of 19. In general, for these consensus algorithms, the number should be of the form 3 f + 1, e.g. 10, 13, 16 or 19.

We also recommend setting the number of candidates *much* higher than the number of validators. This way, the number of possible validator sets is extremely high, and an attacker won’t be able to simply wait for a specific set to occur, where their nodes have a majority.

6. Rationale

6.1 Consensus Mechanisms

6.1.1 Consensus layer options

POSDAO is compatible with a number of consensus algorithms. Chains may choose the underlying algorithm that best suits their use cases and users. For example, AuRa provides a consistent block rate whereas Honey Badger BFT produces varying block rates based on network performance. One of these variants may be advantageous based on the purpose of the network. *Note that the AuRa implementation will be completed first, with HBBFT planned for a future release.*

6.1.2 Delegated Proof of Stake (DPOS)

DPOS is known to provide a high level of scalability at the cost of limiting the number of validators on the network. While this creates a measure of centralization, researchers have argued that "a Byzantine quorum system of size 20 could achieve better decentralization than proof-of-work mining at a much lower resource cost.[19]" Block production is distributed equally among the preset number of elected validators, rather than concentrated among a few mining pools. Transactions can be processed quickly and efficiently, creating a highly scalable solution.

6.2 Reward Structure

6.2.1 Minimum candidate stake

The minimum candidate stake discourages the potential centralization of candidate seats, where individuals may attempt to register many candidate nodes and thus control a large percentage of validator sets. A high minimum candidate stake also deters a malicious set of validators from attempting a coordinated validator set attack. This value is configurable based on the network purpose and size (see 9.4 for reference implementation parameters).

6.2.2 Equal share of the block reward

Each validator pool within a validator set receives an equal share of the block reward*. While a higher stake impacts the odds of a candidate pool becoming a validator pool, each validator pool receives the same reward. This creates parity among the validators participating in each staking epoch.

**Note: shares will only be equal if every validator produces blocks continuously. If a validator skips blocks, their pool will receive proportionally less reward than other (continuously working) validators. For example, if there are two validators in the validator set and one of them produced 10 blocks, but another only 5 blocks, the first validator's pool will receive 10/(10+5)=66% of the total reward, and the second pool will receive the remaining 34%.*

6.2.3 Proportional reward distribution of 70/30%*

The 70/30 distribution ratio is a common revenue sharing heuristic. It is a configurable option that can be changed during or following POSDAO deployment. When set at the initial value, delegators receive block rewards within their validator pool(s) up to 70% of the total pool value, incentivizing delegators to quickly fund candidate pools they believe should be validators.

Once the 70% mark is reached, additional stake returns a proportionally smaller amount. At this point, delegators may choose to fund additional candidate pools, increasing the number and diversity of potential candidates, or stake additional tokens into the current pool, increasing the probability of a candidate pool becoming a validator pool in the next staking epoch.

When the ratio is set to 70/30, a validator never receives less than 30% of the pool reward. Validators are responsible for running a node and a reward baseline prevents a situation where delegators can claim an overwhelming percentage of the reward. Once a pool reaches the 70/30 threshold, a validator may choose to increase their stake to attract additional delegator funds or to increase their position on the leaderboard. Since reputation is a valuable commodity, successful validator sets (those with a high stake, high transactional throughput, and consistent node uptime) will continue to attract stake from delegators.

**After analysis of current reward collection, the reference implementation was modified 06/15/2021 to a straight proportional allocation of rewards to encourage additional delegators to join the protocol.*

6.2.4 Dual Token Environment

The POSDAO implementation provides an opportunity to experiment with different incentivization constructs. Organizations may choose to use a single token/coin for staking, transactions, and rewards, or separate tokens for transactions and staking. In the reference implementation, we describe usage for a single token as well as a dual token environment.

In the dual token implementation we use a stable native coin (xDai, which is pegged to DAI on the mainnet) for transactional purposes. A stable currency allows users to predict fees and conduct transactions where prices will not fluctuate by large margins.

For staking purposes, we use a token whose price is determined by the market. While there is more volatility with this token, it is not a transaction-based asset. Like other tokens, the price is determined by supply and demand in the Ethereum ecosystem. To provide block rewards, the POSDAO staking token STAKE is subject to a configurable annual inflation rate on the sidechain (see 3. Reward distribution). This block reward provides additional incentives for validators and delegators to continue staking.

7. Misbehavior and consensus fault management

In this section we review existing attacks on consensus algorithms and provide our solutions to avoid or mitigate them.

Although POSDAO is a robust algorithm designed to withstand attacks and misbehavior, there are circumstances where the protocol (as well as many other protocols) is susceptible to influence by an alliance of bad actors. The amount of exposure is affected by the underlying consensus algorithm.

Network Stall: Colluding validators can temporarily or indefinitely delay network messaging, resulting in a stalled network. This may also happen accidentally if many validator nodes crash or disconnect in parallel. With the modified AuRa consensus, the network may stall if 33% or more validator nodes are impacted (because the modified AuRa requires at least 67% of validators signatures). With Honey Badger BFT, 33% or more faulty nodes may cause network stalls.

Majority Collusion: The protocol cannot protect against a majority of colluding validators. In both Honey Badger BFT and AuRa, 67% or more of colluding validators can alter the entire protocol by forking the chain, sealing erroneous blocks, double-spending, or re-electing themselves as validators.

7.1 Long Range attack

Problem

An adversary creates an alternate chain (branch) starting from the genesis block. The branch contains a different historical record of transactions and can be maintained locally. Using various techniques, block production can be accelerated on the local branch to create a longer chain than the main (honest) chain. At this point, the branch can be made public, overtaking the honest chain as the ledger of record.

Longer branches may be created in several ways: forging time stamps, gaining access to former validator's keys, or collusion among former validators to quickly populate the branch (costless simulation[4]), or stake bleeding[10], where a malicious validator slows the honest chain by failing to validate blocks while simultaneously increasing stake on the alternate branch. When this attack is successful, an adversary can post a transaction to the honest chain, wait for confirmation, then present the longer branch to invalidate the previously confirmed transaction.

Solution

This issue is addressed on two levels.

First, the underlying consensus mechanism must prevent forks. Both choices we recommend do this:

- AuRa considers blocks to be final as soon as half the validators have signed them. In our modified AuRa implementation no more than 2*MAX_VALIDATORS/3 blocks can be reverted at all. In our reference implementation that means all but the last 13 blocks will never be changed.
- Honey Badger BFT has instant finality, i.e. no block can ever be reverted.

Second, mechanisms are needed to detect if an attacker gained control over a supermajority of some past set of validators and created a fork starting from that point. This is best solved through communication and social consensus outside the protocol itself. If multiple trusted sources occasionally publish the latest block's hash, it is easy to discern the correct chain from a forged one, just like the common genesis block allows nodes to reject otherwise well-formed blockchains that start with a different one. A possible solution is "weak subjectivity" - OpenEthereum provides the forkBlock and forkCanonHash fields in the spec json file to allow for checkpointing.

7.2 Nothing at Stake attack

Problem

In the event of a fork in the chain, either due to simultaneous block production or malicious intervention, validators may choose to continue generating blocks on both chains, thus collecting block rewards on both chains without penalty. In fact, "the optimal strategy for any miner (validator) is to mine on every chain, so that the miner gets their reward no matter which fork wins[11]." This attack reduces efficiency by slowing down consensus time. Moreover, it results in blockchain forks which weaken the ability of the blockchain to resolve double spending attacks and other threats[12].

Solution

This, too, depends on the underlying consensus algorithm's ability to prevent forks:

- OpenEthereum's AuRa implementation includes a mechanism to report validators that propose more than one block with the same number. In other words: mining on a fork *does* result in a penalty on the main chain, so there *is* something at stake in that scenario.
- With Honey Badger BFT, it is impossible to create a fork in the first place, unless more than two-thirds of the validators collude. In addition, the authors' implementation detects and reports misbehaving nodes.

7.3 Fake Stake attack

Problem

In POSv3 (Bitcoin-based) networks, headers and blocks are sent as separate data structures and committed to storage before all validation checks are complete. These checks are sufficient in Proof of Work implementations, however in Proof of Stake implementations several vulnerabilities are created[13]. An attacker can flood a node with either fake block headers or fake blocks filled with arbitrary values. The RAM (headers) or disk space (blocks) is filled before validation checks are run, and the node is rendered unusable. A variation of this attack, called a "spent stake attack[14]," impacts pre-POSv3 networks that utilize Unspent Transaction Outputs (UTXOs) to record transactions. In this case, an attacker can amplify their apparent stake because validation checks do not differentiate between spent and unspent UTXOs. With this apparent stake, the attacker can mine POS blocks at a past time and then use these invalid blocks to overwhelm a node's disk space.

Solution

In POSDAO, stake is not used directly to produce a block. Instead, the stake makes it more likely to be selected as a validator, and validators take part in an underlying consensus protocol that produces blocks independent of the stakes. Depending on that protocol, block headers can actually be validated very quickly: In AuRa the header is the step number and the signature by the block's proposer, and in Honey Badger BFT it is a threshold signature.

7.4 Cloning attack

Problem

The default OpenEthereum AuRa protocol is vulnerable to an attack[15] where a malicious validator may clone their node (create 2 or more instances of the same validator with the same public/private key pair), create a network partition (for example by delaying messages[16]), and then use these cloned nodes on both sides of the partition. Because AuRa requires 1/2 of the validators to be honest, using the cloned nodes gives the illusion to the block validators on each partition that their group contains an honest majority.

Once the blockchain is divided, the attacker issues conflicting transactions to each group, creating a double-spend scenario. The attacker waits for the transactions to be committed, then signs an additional block on one of the partitions (whichever one the attacker wants to preserve). This ensures the partition with this additional block becomes the longest chain (it has the greatest height) and it is accepted by the remaining nodes as the canonical chain. The other partition disappears, along with the transactions that were already accepted there.

Solution

Increasing the required number of signatures to 2/3 of validators rather than 1/2 of validators in the AuRa implementation is sufficient to protect against this type of attack. As long as the number of malicious validators is less than 1/3 of the total, the modified algorithm can guarantee safety, and liveness is still preserved (although block times may be impacted slightly in the event of a network partition). We have modified the AuRa protocol to utilize a 2/3 majority in order to protect against a cloning attack.

7.5 POSDAO implementation specific attacks

7.5.1 RANDAO attack

Problem

The POSDAO implementation requires a reliable source of randomness to manage validator sets. For AuRa we use the same algorithm as RANDAO[17], a widely used decentralized random number generator. In a “look-ahead” attack scenario, an attacker exploits the RANDAO reveal functionality by skipping block production. In order to be viable, this type of attack typically requires collusion or control of multiple nodes and a willingness to forfeit the block reward. When an attacker skips block validation, they can use the revealed information to pre-compute the possible outcomes of “(possibly many) reveal strategies…and thus may anticipate the effects of their contribution to the process and bias the generated random number to their advantage[18].”

Solution

POSDAO contracts know whether a validator revealed their secret or not. If the validator didn't reveal more than 55 times per staking epoch (this value is calculated based on the STAKE_WITHDRAW_DISALLOW_PERIOD and COLLECTION_ROUND_LENGTH constants), they are treated as malicious at the last block of that epoch and are removed by the algorithm from the list of active pools. In this case, the validator and all their delegators are marked as banned for the next 90 days (BAN_PERIOD), and they cannot withdraw their stakes from the banned pool during that period. The banned pool cannot participate in the following staking epochs for 90 days.

This limit of 55 allowed skips per staking epoch is to avoid punishing validators for temporary outages. However, it also allows malicious validators to influence the outcome by faking such an outage. This tradeoff avoids sometimes banning honest validators with connection problems at the cost of making the random numbers less secure. When the numbers are used for high-value applications (e.g. a casino game with high stakes), the limit should likely be 0, and the validators should be expected to run multiple (backup) nodes or take other measures to prevent any outages.

If a validator decides not to reveal their secret on the last reveal round of a staking epoch (to influence the outcome), they are treated as malicious. It is the validator’s responsibility to ensure their node is functional and connected during the final blocks of a staking epoch. They will be removed even if they have merely lost their connection during this phase.

We recommend validators run two separate nodes simultaneously with different internet connections to ensure reliability: a primary node with the `engine_signer` option and a secondary node without that option but with a guard script which sets that option for the secondary node dynamically when the first node goes offline. Since running nodes in the unsecured cloud is not recommended (for security reasons), each validator should have nodes in multiple secured data centers.

During the last 6 hours (STAKE_WITHDRAW_DISALLOW_PERIOD) of a staking epoch (this duration is configurable), no one can stake or withdraw, preventing any possible influence to the outcome.

The only possible attack is if a validator chooses not to reveal the secret at the end of the staking epoch. This results in a freeze of the validator's stakes (and all delegators that delegated stake to them) and a ban from participation for 90 days (BAN_PERIOD).

7.5.2 Exit from Bridge attack

Problem

To achieve interoperability with Ethereum, multiple bridge instances can connect the POSDAO chain to the Ethereum mainnet (see 5.1 for details). Validators on the bridge are responsible for locking assets on one side of the bridge and minting assets on another. These operations require multiple signatures from a number of validators before they are performed. The bridge validators are distinct from validators on the POSDAO chain, and each bridge instance requires a predetermined set of validators (they may be the same or separate for each bridge).

A colluding majority of bridge validators (either malicious or compromised nodes) could commandeer the bridge contracts and mint assets without the corresponding locked assets. This would create tokens or coins which are not backed by real assets. A colluding majority could also unlock assets on the other side of the bridge and drain the locked funds.

Solution

Separate from the POSDAO consensus validators, the bridge validators are known individuals who stake their reputations to secure the bridge contracts. These individuals (typically organizations) are community participants who agree to take on the responsibilities of managing the bridge. The bridge validation set does not rotate on a regular basis, although validators may be added or removed through a governance process. In addition, there are daily limits which safeguard against unchecked asset creation, destruction, or transfer. If a limit is exceeded, any transaction which requests to relay assets will fail.

7.5.3 Coordinated Validator Set attack

Problem

A colluding set of malicious validators (greater than 67% of the set) could subvert the network by deploying new POSDAO contracts, changing the *spec.json* on their nodes, and then restarting their nodes. The new *spec.json* file would contain a new block number in `engine.authorityRound.params.validators.multi` (see https://github.com/poanetwork/poa-chain-spec/blob/dai/spec.json). This would create a hard fork, and the validators would control the code of the contracts.

Solution

As mentioned in the introduction of section 7, the protocol itself cannot protect against a majority of colluding validators. However, increasing the CANDIDATE_MIN_STAKE parameter greatly reduces the feasibility of this attack, making it extremely expensive for a conspiring group to attempt. By increasing the value of this constant, the summary costs for a group are vastly increased, discouraging any efforts to coordinate a malicious takeover.

8. Future directions

8.1 Honey Badger BFT full implementation

While the POSDAO algorithm is configurable to allow for different consensus mechanisms, AuRa consensus is currently the only method fully supported by the protocol. We are working to integrate HoneyBadger BFT as an alternative pluggable consensus. See Appendix E for additional details.

8.2 Bridge governance development

In a bridged scenario, bridge validators are granted multisig owner status. They are trusted during launch, and in the reference implementation are known, credible organizations. We will explore bridge governance strategies to promote further decentralization for future releases.

Update 06/16/2021: Bridge Governance Board now in place.

8.3 Reward model analysis

Highly configurable incentivization structures allow us to model different reward scenarios (see Appendix D). We will continue to test additional designs to fine-tune and explore the impact of different reward models.

8.4 Hypothecation

It is possible to allocate tokens "locked" in the bridge as collateral for loans through cryptocurrency lending services such as Compound. Guarantees must be in place so that any locked funds are returned on request, whenever a bridge transfer is initiated. We will explore this methodology to ensure it is sound, and analyze the costs/benefits of this novel approach for providing additional staking incentives.

9. Reference implementation notes

9.1 POSDAO smart contracts

POSDAO is configured using a set of Solidity smart contracts to implement consensus, rewards, and staking logic.

- **`BlockRewardAuRa`:** generates and distributes rewards according to the logic and formulas described in section 3. Main features include:
  - distributes the entrance/exit fees from the ERC677-to-ERC677, Native-to-ERC20, and/or ERC20-to-Native bridges among validators and their delegators;
  - distributes staking tokens/coins according to defined inflation rate among validators and their delegators;
  - mints native coins needed for the ERC20-to-Native bridge;
  - makes a snapshot of the validators at the beginning of each staking epoch. The snapshot is used by `StakingAuRa.claimReward` function.

- **`Certifier:`** allows validators to use a zero gas price for their service transactions (see the OpenEthereum Wiki for more info). The following functions are considered service transactions:
  - `ValidatorSetAuRa.emitInitiateChange`
  - `ValidatorSetAuRa.reportMalicious`
  - `RandomAura.commitHash`
  - `RandomAura.revealNumber`

- **`InitializerAuRa`:** used once on network startup and then destroyed. This contract is needed for initializing upgradeable contracts when starting from genesis. The bytecode of this contract is written by the scripts/make_spec.js script into *spec.json* along with other contracts.

- **`RandomAuRa`:** generates and stores random numbers in a RANDAO manner (and controls when they are revealed by AuRa validators). Random numbers are used to form a new validator set at the beginning of each staking epoch by the `ValidatorSetAuRa` contract. Key functions include:
  - `commitHash` and `revealNumber`. These can only be called by the validator's node when generating and revealing their secret number (see RANDAO to understand the principle). Each validator node must call these functions once per "collection round." This creates a random seed which is used by the `ValidatorSetAuRa` contract;
  - `onFinishCollectRound`. This function is automatically called by the `BlockRewardAuRa` contract at the end of each "collection round." It controls the reveal phase for validator nodes and punishes validators when they don't reveal (see 7.5.1 for more details on the banning protocol).

- **`Registry`:** stores human-readable keys associated with addresses, like DNS information (see the OpenEthereum Wiki). This contract is primarily required to store the address of the `Certifier` contract (see the OpenEthereum Wiki for details).

- **`StakingAuRa`:** contains staking logic including:
  - creating, storing, and removing pools by candidates and validators;
  - staking tokens or native coins by participants (delegators, candidates, or validators) into the pools;
  - storing participants' stakes;
  - withdrawing tokens/coins by participants from the pools;
  - withdrawing participant's reward;
  - moving tokens/coins between pools by participants.

- **`TxPermission`:** controls the use of a zero gas price by validators in service transactions, protecting the network against "transaction spamming" by malicious validators. The protection logic is declared in the `allowedTxTypes` function.

- **TxPriority:** manages and stores the transactions priority list used by the Ethereum client. See https://github.com/NethermindEth/nethermind/issues/2300 for description.

- **`ValidatorSetAuRa`:** stores the current validator set and contains the logic for choosing new validators at the beginning of each staking epoch. The logic uses a random seed generated and stored by the `RandomAuRa` contract. Also, `ValidatorSetAuRa` along with the modified OpenEthereum client is responsible for discovering and removing malicious validators. This contract is based on `ReportingValidatorSet` described in the OpenEthereum Wiki.

Note that HBBFT contract implementations are not fully finished, and they are not listed nor described here.

For a detailed description of each function of the contracts, see the source code.

9.2 Implementation details

The following detailed outline is provided for the xDai POSDAO AuRa implementation. They describe starting a network from the genesis block. Note that these details may change as further optimizations are introduced. Any protocol changes will be documented in updated versions.

9.2.1 Network startup

Before the network starts from the genesis block, we compile all consensus contracts and write their bytecodes into the *spec.json* file with the *make_spec.js* script. The compiled bytecode contains constructor parameters for those contracts which have a constructor. The contracts can also be deployed on an existing AuRa network, not only on the genesis block.

The network should have several initial validators which are defined in the constructor of the `InitializerAuRa` contract. We pass their addresses to the *make_spec.js* (example) along with the other necessary parameters to configure the network on the genesis block.

After the *spec.json* file is ready, the initial validators must configure and start their OpenEthereum nodes. Each node uses the *spec.json* as a chain specification.

Once the network is started, the `ValidatorSetAuRa.finalizeChange` function is called by the system at the beginning of the finality block (see the OpenEthereum Wiki for details). This function sets the `initiateChangeAllowed` boolean flag to *true* (this flag is used by the `ValidatorSetAuRa.emitInitiateChange` function which is called by a validator's node when there is a change in the pending validator list). The `ValidatorSetAuRa.finalizeChange` function is described in more detail below.

Beginning from the genesis block, the initial validators do not have stakes because an ERC token contract is not yet deployed. They are able to place stakes after the ERC token contract is deployed and the ERC677-to-ERC677 and ERC20-to-Native bridges are installed (when using a dual token environment).

In addition to the initial validators, we also define the contracts' *owner*. The *owner* is an address which is used for the bridge deployments and future contracts' upgradability. Initially, the *owner* can make zero gas price service transactions because there are no native coins on its balance.

When the ERC contract and the bridges (ERC677-to-ERC677 and ERC20-to-Native) are deployed by the *owner*, the following functions are called (when using a dual token environment):

- `BlockRewardAuRa.setErcToErcBridgesAllowed`
- `BlockRewardAuRa.setErcToNativeBridgesAllowed`
- `StakingAuRa.setErc677TokenContract`
- `ERC677BridgeTokenRewardable.setBlockRewardContract`
- `ERC677BridgeTokenRewardable.setStakingContract`

Then the *owner:*

1. Deploys the MultiSigWallet contract with the trusted addresses defined in its constructor.
2. Calls the `transferOwnership` function (it may have another name depending on the contract) for each of the contracts it deployed and for each consensus contract.
3. Passes its address to the `Certifier.revoke` and `TxPermission.removeAllowedSender` functions to disallow itself from using zero gas price service transactions. At this point, the *owner* becomes a regular address without special system rights.

As each block is closed, the `BlockRewardAuRa.reward` function is called by the validator's node which created that block. This is an OpenEthereum feature and it occurs for each block (with no gaps - see https://openethereum.github.io/Block-Reward-Contract.html). This function primarily exists for dynamic block rewards, but we use it for POSDAO because `BlockRewardAuRa.reward` has no block gas limit, allowing us to make a lot of calculations. This function is described in more detail below.

9.2.2 Placing stakes in ERC677 tokens

During the first staking epoch the initial validators place stakes into their own pools in order to retain their seats after the first staking epoch finishes.

Each validator has two addresses: a *mining address* and a *staking address*. The *mining address* is used by the validator's node (this address must be specified as *engine_signer* and *unlock* in the OpenEthereum node configuration toml file) to make zero gas price service transactions and to sign blocks. The *staking address* is used for other purposes (placing stakes, storing tokens, etc.)

To retrieve the *mining address* by its *staking address*, and vice versa, the `ValidatorSetAuRa.miningByStakingAddress` and `stakingByMiningAddress` getters can be used.

Staking ERC tokens are acquired from the Ethereum mainnet (the foreign network) and bridged to the xDai POSDAO chain (the home network).

9.2.2.1 POSDAO ERC677 staking tokens

POSDAO tokens exist on the Ethereum mainnet as ERC677 tokens. Once acquired, users can transfer their tokens to the xDai POSDAO chain using the ERC677-to-ERC677 bridge. There is an entrance fee associated with this transfer. These tokens are then placed as stakes, as detailed below. Block rewards are paid in staking tokens, based on an annual inflation rate.

9.2.2.2 Initial validator stakes

To place a stake, a validator must have at least CANDIDATE_MIN_STAKE (see the `StakingAuRa.candidateMinStake` getter) tokens on the ERC balance of its *staking address.* This address must also have a balance of native coins to pay for gas costs. To get staking tokens from Ethereum mainnet (*foreign network*), a validator uses the ERC677-to-ERC677 bridge. To get native coins, the validator uses the ERC20-to-Native bridge to transfer ERC20 DAI tokens from the foreign network to the home network (xDai POSDAO chain).

After the tokens and native coins are received from the foreign network using the bridge, the initial validator calls the `StakingAuRa.stake` function from the *staking address* with a non-zero gas price. The validator passes their staking address and staking amount (greater than or equal to the CANDIDATE_MIN_STAKE) as parameters.

9.2.2.3 Candidate stakes

A new candidate must:

1. Acquire tokens and native coins from the foreign network.
2. Launch their OpenEthereum node.
3. Call `StakingAuRa.addPool` from their *staking address* and pass the staking amount (>= CANDIDATE_MIN_STAKE) and their *mining address as parameters*. The `StakingAuRa.addPool` function is similar to `StakingAuRa.staking` but is only for candidates.

9.2.2.4 Delegator stakes

A delegator must:

1. Acquire tokens and native coins from the foreign network.
2. Call the `StakingAuRa.stake` function and pass the staking address of the validator/candidate they would like to place their stake on, and the staking amount which is greater than or equal to DELEGATOR_MIN_STAKE (see the `StakingAuRa.delegatorMinStake` getter).

9.2.2.5 Staking window

The `StakingAuRa.stake` and `StakingAuRa.addPool` functions can only be called if the `StakingAuRa.areStakeAndWithdrawAllowed` getter returns *true*. This getter defines the window inside the staking epoch, during which staking and withdrawing can be performed by the participants. For AuRa, this window begins from the first block of a staking epoch and ends during the final blocks of the staking epoch when a disallow period begins (the disallow period duration is defined by `StakingAuRa.stakeWithdrawDisallowPeriod`).

9.2.2.6 Helpful StakingAuRa getters

- `StakingAuRa.stakeAmount`: the total staked amount for a specified pool made by the specified address. Doesn't include the amount ordered for withdrawal.
- `StakingAuRa.orderedWithdrawAmount:` the current amount of staking tokens/coins ordered for withdrawal from the specified pool by the specified staker.
- `StakingAuRa.stakeAmountTotal`: the total staked amount for a specified pool made by all delegators and the specified staking address. Doesn't include the amount ordered for withdrawal.
- `StakingAuRa.orderedWithdrawAmountTotal:` the current total amount of staking tokens/coins ordered for withdrawal from the specified pool by all of its stakers.
- `StakingAuRa.getPools`: the list of active pools (list of staking addresses of candidates and validators).
- `StakingAuRa.poolDelegators`: the list of current delegators of a specified pool.

9.2.3 New staking epoch, validator selection, and finalizing changes

During each staking epoch, participants can call these functions:

- `StakingAuRa.stake:` to place stakes;
- `StakingAuRa.addPool:` to create a new pool (for candidates only);
- `StakingAuRa.removeMyPool:` to remove an existing pool, make it inactive (for candidates and validators);
- `StakingAuRa.withdraw, StakingAuRa.orderWithdraw, StakingAuRa.claimOrderedWithdraw:` to withdraw stakes;
- `StakingAuRa.moveStake:` to move stakes.
- `StakingAuRa.claimReward:` to withdraw a reward from the specified pool for the specified staking epochs.

On the last block of the staking epoch, the `BlockRewardAuRa.reward` function calls the `ValidatorSetAuRa.newValidatorSet` function. This function:

- selects new validators;
- writes the list of new validators (validator set) into the *pending list*;
- increments the staking epoch number which is returned by `StakingAuRa.stakingEpoch` getter;
- resets the number of the validator set apply block which is returned by the `ValidatorSetAuRa.validatorSetApplyBlock` getter.

If the total number of candidates and validators is less than or equal to MAX_VALIDATORS, they will all become validators on the next staking epoch. Otherwise, the validators are selected based on a random value and weighted by their pool sizes (the larger the pool, the more likely the candidate becomes a validator). The random seed is taken from the `RandomAuRa` contract described below.

The pending list of new validators (their mining addresses) can be read by the `ValidatorSetAuRa.getPendingValidators` getter.

The validator set cannot be changed immediately because the `InitiateChange` event must be emitted (see the OpenEthereum Wiki) prior to this change, so the pending validator set is queued by the `ValidatorSetAuRa.newValidatorSet` function to be handled later by the `ValidatorSetAuRa.emitInitiateChange` and `ValidatorSetAuRa.finalizeChange` functions.

The `ValidatorSetAuRa.finalizeChange` function is only called for one `InitiateChange` event. If there are several such events, `finalizeChange` is called only once for the first emitted event (and the additional `InitiateChange` events are ignored). So, we can't emit the next `InitiateChange` unless the previous event is not finalized.

For that reason, there is a pending validator set used by the `ValidatorSetAuRa.emitInitiateChange` function. When the `ValidatorSetAuRa.emitInitiateChangeCallable` getter returns *true*, a validator's node calls `ValidatorSetAuRa.emitInitiateChange` which reads the pending validator set and emits the `InitiateChange` event to let the validators' nodes know that the set of validators should be changed (see the OpenEthereum Wiki).

When the `InitiateChange` event is emitted, the `ValidatorSetAuRa.emitInitiateChange` function sets the *`initiateChangeAllowed`* boolean flag to *false* so that the `ValidatorSetAuRa.emitInitiateChangeCallable` getter returns *false* until `ValidatorSetAuRa.finalizeChange` is called by the engine.

When applying the new validator set, the engine calls the `ValidatorSetAuRa.finalizeChange` function to notify the `ValidatorSetAuRa` contract that the pending validator set can be written to the current validator set, which is returned by the `ValidatorSetAuRa.getValidators` getter.

At this point, the `ValidatorSetAuRa.finalizeChange` function:

- writes the pending validator set to the current set;
- sets `validatorSetApplyBlock` to the current block number;
- sets the `initiateChangeAllowed` boolean flag to *true*.

The `ValidatorSetAuRa.validatorSetApplyBlock` getter is used to determine the block number where the current validator set was applied by the engine. If this getter returns 0, it means the new staking epoch is started (`ValidatorSetAuRa.newValidatorSet` is called) but the new validator set is not yet applied by the engine.

9.2.4 Validator set changes and pending validator set

The validator set is always evaluated at the very end of a staking epoch, however, it can also be changed during a staking epoch if a validator needs to be removed due to misbehavior.

In such cases the malicious validator is removed from the pending validator set (see the internal `ValidatorSetAuRa._removeMaliciousValidator` function) and the new pending validator set is marked to be handled later by the `ValidatorSetAuRa.emitInitiateChange` and `ValidatorSetAuRa.finalizeChange` functions.

As with staking epoch validator set changes, the `ValidatorSetAuRa.finalizeChange` function is used to finalize the validator set change when the malicious validator is removed.

The pending validator set checkmark (see `ValidatorSetAuRa._pendingValidatorsChanged` internal boolean flag) is used to finalize different validator sets one after another. This may be required when a malicious validator is removed and the new staking epoch begins immediately after the removal, or when one malicious validator is removed at block number *N*, but another malicious validator is removed on the next block *N+1*. Without the boolean checkmark, such cases would be handled incorrectly because some of the corresponding `InitiateChange` events would be ignored by the OpenEthereum engine (as mentioned above).

9.2.5 Random seed accumulation

When the number of pools (validators + candidates) is more than MAX_VALIDATORS, a random seed is used to select MAX_VALIDATORS validators for a new staking epoch (see the `ValidatorSetAuRa.newValidatorSet` function).

The random seed is stored in the `RandomAuRa` contract (see the `RandomAuRa.currentSeed` getter) and generated in the RANDAO manner. There are several collection rounds per staking epoch, each of which is split into two equal phases - a commit phase and reveal phase:

1. Commit phase: Each validator node generates its secret number on each phase and calls the `RandomAuRa.commitHash` function to commit the hash of the secret (one time per each commit phase).
2. Reveal phase: Each node passes its secret to the `RandomAuRa.revealNumber` function (once per each reveal phase). The `revealNumber` function XORs the revealed secret with the current seed stored in the contract, increasing entropy in every collection round.

Committing/revealing the secret is mandatory for each validator. These functions are called automatically by each validator’s node from the validator's *mining address* (see 9.3). An

explanation is also available in
https://github.com/openethereum/openethereum/pull/10946#issuecomment-547857765.

The length of a collection round is defined at network startup in the `InitializerAuRa` contract constructor when starting from genesis (or through the `RandomAuRa.initialize` function when starting on an existing network).

The `RandomAuRa.onFinishCollectRound` function is called on each block by the `BlockRewardAuRa.reward` function. At the end of each collection round `RandomAuRa.onFinishCollectRound` checks whether the validator skipped revealing the secret during the collection round and increments a skip counter if they did.

On the final collection round of each staking epoch the `RandomAuRa.onFinishCollectRound` function checks each validator to see

1. if they skipped revealing too often during the current staking epoch or,
2. if they skipped revealing on the last collection round.

If either of these are true, the validator is treated as malicious and removed with the `ValidatorSetAuRa.removeMaliciousValidators` function.

The maximum number of reveal skips is defined in the `RandomAuRa.onFinishCollectRound` function and depends on the disallow period duration (see the `StakingAuRa.stakeWithdrawDisallowPeriod` getter) and on the length of collection round (see the `RandomAuRa.collectRoundLength` getter).

The final collection round is especially important to check. During the final round, the last validator in the validator set can decide not to reveal their secret to try to influence the outcome in the `ValidatorSetAuRa.newValidatorSet` function. For this reason, any validator that doesn't reveal their secret during the last collection round is treated as malicious and removed from the validator set.

To prevent this from happening accidentally due to disconnection, it is recommended that each validator run two separate nodes simultaneously with different internet connections. The first node must have the `engine_signer` option in a configuration toml file, the second node should not have that option but should have the *watchguard* script which detects if the first node goes offline and sets the `engine_signer` option for the second node (see https://github.com/poanetwork/posdao-test-setup/issues/39).

For debugging purposes, it is possible to temporarily turn off the punishment for skipping random number reveals in the `RandomAuRa` contract (see its `setPunishForUnreveal` function). This boolean flag may be used for testing when a new version of OpenEthereum is unstable and block skipping is evident due to possible bugs in the engine.

Since a new validator set can be applied by the engine at any time (e.g. in the middle of some commits/reveals phase), there may be a case when new validators can't fully commit/reveal their secrets at the very beginning of the staking epoch. Because of this, there is a short grace period after the new validator set is applied during which the skip counter is not incremented if a reveal is skipped (see the code of `RandomAuRa.onFinishCollectRound` function).

`RandomAuRa.commitHash` and `RandomAuRa.revealNumber` are called with a zero gas price. This is enabled because the validators' *mining addresses* can have zero balances.

To protect the network against possible spamming by a malicious validator calling the `RandomAuRa` functions with zero gas price too often, there is a protection mechanism implemented in the `TxPermission.allowedTxTypes` function: it doesn't allow creating the `RandomAuRa` transactions unless they are permitted on the current block (see the `RandomAuRa.commitHashCallable` and `RandomAuRa.revealNumberCallable` getters).

The `TxPermission.allowedTxTypes` getter is called by the OpenEthereum node every time a new transaction is about to be added to a block. This getter checks if the transaction can be included into the block or not.

9.2.6 Removing malicious validators

There are two cases when a validator can be removed due to misbehavior:

1. by the `RandomAuRa` contract due to not revealing random numbers (see above);
2. by the `ValidatorSetAuRa.reportMalicious` function (see the OpenEthereum Wiki).

The `ValidatorSetAuRa.reportMalicious` function can be called by validators to report a specified validator's misbehavior on a specified block number. If more than 2/3 of validators report about the same validator for the same block, the validator is removed from the validator set (see the code of the `ValidatorSetAuRa.reportMalicious` function).

The validators' nodes call `ValidatorSetAuRa.reportMalicious` with a zero gas price. This is enabled because the validators' *mining addresses* can have zero balances.

To protect the network against possible spamming by a malicious validator calling the `ValidatorSetAuRa.reportMalicious` function with a zero gas price too often, there is a protection mechanism implemented in the `TxPermission.allowedTxTypes` function: it doesn't allow creation of the `ValidatorSetAuRa.reportMalicious` transaction unless it is permitted on the current block (see the `ValidatorSetAuRa.reportMaliciousCallable` getter).

Moreover, if some validator calls `ValidatorSetAuRa.reportMalicious` too often (much more often than others), such a validator is also treated as malicious (see the code of `ValidatorSetAuRa.reportMaliciousCallable` and `ValidatorSetAuRa.reportMalicious`).

The validator removal process is implemented in the `ValidatorSetAuRa._removeMaliciousValidator` internal function. It is called by `ValidatorSetAuRa._removeMaliciousValidators`. When the validator is removed, their *mining address* is banned for the BAN_PERIOD. This means that the banned validator and their delegators cannot withdraw their stakes until the BAN_PERIOD expires (see the `StakingAuRaBase._isWithdrawAllowed` internal function). Also, the banned *mining address* cannot be added into the validator set during that period.

When the validator is removed from the current validator set, the new pending validator set waits to be passed to the `InitiateChange` event (see 9.2.4).

Helpful public getters:
- `ValidatorSetAuRa.isValidatorBanned:` check if a specified mining address is banned;
- `ValidatorSetAuRa.banCounter:` read the ban counter, which is incremented when a validator is banned;
- `ValidatorSetAuRa.bannedUntil:` view the block number when a mining address is released from the ban.

In a test network, it is possible to set one validator that is not removable (see 9.2.11). Such a validator cannot be removed from the validator set even due to misbehavior (see `ValidatorSetAuRa._removeMaliciousValidator` and the section about the unremovable validator below). This feature prevents test network shutdown if there is a problem with the validator set (for example if all validators leave the network simultaneously).

9.2.7 Block reward distribution

The block reward is distributed among validators and their delegators by the `BlockRewardAuRa.reward` function. This function is called by the OpenEthereum engine when closing each block.

The pool reward distribution logic is implemented in the `BlockRewardAuRaBase._distributeRewards` internal function. Changes in stake amounts during the staking epoch do not affect the pool reward during that epoch, but they do affect the future rewards collected on the next staking epoch.

There are several types of block rewards available for distribution:
- ERC20-to-Native bridge fee;

- ERC20-to-Native bridge reward from Dai Savings Rate;
- ERC677-to-ERC677 or Native-to-ERC20 bridge fee;
- inflation distribution.

Block rewards are distributed at the last block of each staking epoch (except the very first staking epoch) - see the `BlockRewardAuRaBase._distributeRewards` internal function for details. Rewards are not distributed automatically, they must be pulled by participants using the `StakingAuRa.claimReward` function.

9.2.7.1 ERC20-to-Native bridge fee distribution

This distribution occurs after the bridge passes the fee amount to the `BlockRewardAuRa.addBridgeNativeRewardReceivers` function. The fee is distributed in native coins (xDai).

When the bridge passes the fee amount, its value is stored in the `BlockRewardAuRa` contract and then read by the `BlockRewardAuRaBase._distributeNativeRewards` internal function at the end of staking epoch. The function saves distribution shares to the contract's state and returns the amount which needs to be minted by the OpenEthereum engine. The distribution shares are then used by the `StakingAuRa.claimReward` function which transfers the participant's portion of the reward from the BlockRewardAuRa balance to the participant's balance.

9.2.7.2 ERC20-to-Native bridge reward from Dai Savings Rate

This type of reward works similarly to the ERC20-to-Native bridge fee. The reward in the form of native coins is accrued to the `BlockRewardAuRa` contract with the `BlockRewardAuRa.addBridgeNativeRewardReceivers` function, and the reward amount is used as part of the total reward distributed to participants at the end of staking epoch.

The Dai Savings Rate reward is accumulated on the Mainnet side in the form of DAI tokens (while the bridge has a positive DAI balance) to be transferred to xDai later.

9.2.7.3 ERC677-to-ERC677 or Native-to-ERC20 bridge fee distribution

This distribution occurs after the bridge passes the fee amount to the `BlockRewardAuRa.addBridgeTokenRewardReceivers` function. The fee is distributed in ERC staking tokens.

When the bridge passes the fee amount, its value is stored in the `BlockRewardAuRa` contract and then read by the `BlockRewardAuRaTokens._distributeTokenRewards` internal function. The function saves distribution shares to the contract's state and then calls the `mintReward` function of the ERC contract to mint the total reward amount The distribution shares are then used by the `StakingAuRa.claimReward` function which transfers the participant's part of the reward from the BlockRewardAuRa balance to the participant's balance.

9.2.7.4 Inflation distribution (when using ERC staking tokens)

Inflation distribution uses the total stake amount calculated as snapshots at the beginning of the staking epoch. Only the amounts staked on active validators are included in the snapshots.

The `BlockRewardAuRaBase._distributeRewards` internal function is called at the end of staking epoch. It reads the total stake amount snapshot calculated at the beginning of the staking epoch (see the `BlockRewardAuRa.reward`). The given amount is multiplied by a constant so that the inflation is defined by the formula (see 3. Reward distribution). Once calculated, the amount is distributed among the validators and their delegators. The distributed reward can be claimed using the `StakingAuRa.claimReward` function.

9.2.7.5 Inflation distribution (when using native staking coins)

The `BlockRewardAuRaBase._distributeRewards` internal function is called at the end of a staking epoch. It reads the total stake amount snapshot calculated at the beginning of the staking epoch (see the `BlockRewardAuRa.reward`). The given amount is multiplied by a constant so that the inflation is defined by the formula (see 3. Reward distribution) and then distributed by the `BlockRewardAuRaBase._distributeNativeRewards` among the validators and their delegators. The distributed reward can be claimed using the `StakingAuRa.claimReward` function.

9.2.8 Withdrawing stakes

Any participant can withdraw their stake (or part of it) using the `StakingAuRa.withdraw` function. The user passes the *staking address* of the pool and the amount to withdraw from that pool.

Withdrawals are allowed during the same period of time when stakes are allowed (see 9.2.2.5).

The user can withdraw their full stake or a portion of it. In the latter case the user must leave at least DELEGATOR_MIN_STAKE or CANDIDATE_MIN_STAKE (depending on their role) in the pool. This is controlled by a contract.

If the user withdraws from a candidate's pool which is not yet a validator, they can withdraw their entire stake amount minus the already ordered withdrawal amount.

When withdrawing from a validator's pool, the user can withdraw only the amount which they have staked during the current staking epoch (see `StakingAuRa.maxWithdrawAllowed` getter) or they can order a withdrawal. That amount can be claimed during the following staking epochs (see the `StakingAuRa.claimOrderedWithdraw` function).

The `StakingAuRa.maxWithdrawAllowed` getter checks the maximum **current** allowable withdrawal. If the pool is an active validator, the maximum possible withdrawal amount is calculated based on existing withdrawal orders as well as the amount staked during the current staking epoch.

The `StakingAuRa.maxWithdrawOrderAllowed` getter checks the maximum possible withdrawal amount available to order. If the pool is an active validator, the maximum possible withdrawal amount is calculated based on existing withdrawal orders.

The `StakingAuRa.orderWithdraw` function orders a withdrawal. The *staking address* of the pool and the amount to be withdrawn are passed into the function. The ordered withdrawal amounts can also be reduced or cancelled by passing a negative value into the `StakingAuRa.orderWithdraw` function. The ordered amount can be claimed during the following staking epochs with the `StakingAuRa.claimOrderedWithdraw` function.

The `StakingAuRa.orderedWithdrawAmount` public getter is used to check the current ordered withdrawal amount.

9.2.9 Moving stakes

In addition to withdrawing stakes, a participant can also move their stake (or a portion of their stake) from one pool to another using the `StakingAuRa.moveStake` function. This prevents the removal of staking tokens/coins from the balance of the `StakingAuRa` contract when moving from one pool to another. The stake moving functionality is subject to the same rules and limitations implemented for the withdrawal functions above.

9.2.10 Voluntary exit from a validator set

If a validator wants to exit from the validator set, they can use one of the following methods:

1. The validator can call the `StakingAuRa.removeMyPool` function from their *staking address*. The pool of the validator is removed from the list of active pools by this function, thus this pool won't take part in forming a new validator set at the beginning of the next staking epoch (see the `ValidatorSetAuRa.newValidatorSet` function).
2. The validator can order a withdrawal of their entire stake amount using the `StakingAuRa.orderWithdraw` function (see 9.2.8). In this case, the validator will have zero real balance in their own pool at the beginning of the next staking epoch and the pool will not be selected by the `ValidatorSetAuRa.newValidatorSet` function.

9.2.11 Unremovable validator

Optionally, the POSDAO network can have one unremovable validator (recommended for test networks only). This is defined once on network startup through the `InitializerAuRa` contract (see the `_firstValidatorIsUnremovable` boolean parameter of the constructor). If the parameter is *true*, the first initial validator defined in the constructor is unremovable.

Such a validator cannot be removed from the validator set either by the `ValidatorSetAuRa.newValidatorSet` (even if the validator didn't place any stake for themselves), nor by the `ValidatorSetAuRa.reportMalicious` function. However, if they want to change their unremovable status, they can call the `ValidatorSetAuRa.clearUnremovableValidator` function by their staking address. This function can also be called by the contract's owner. The `ValidatorSetAuRa.clearUnremovableValidator` function can only be called once. After it is called, it is not possible to have another unremovable validator.

The `ValidatorSetAuRa.unremovableValidator` public getter retrieves the *staking address* of the unremovable validator. If this getter returns a zero address, there is no unremovable validator in the validator set.

9.2.12 Zero gas price and service transactions

The *owner* and the current validators are able to make service transactions (regular transactions with zero gas price) from their *mining addresses*. This is done through the `Certifier` contract; its address is written into the *Registry* contract (see the OpenEthereum Wiki). The *owner* has already been mentioned in 9.2.1 above.

Initially, the validators don't have any native coins to pay gas fees when executing transactions (when calling the `RandomAuRa.commitHash`, `RandomAuRa.revealNumber`, `ValidatorSetAuRa.emitInitiateChange`, and `ValidatorSetAuRa.reportMalicious` functions; see 9.3). For that reason, their *mining addresses* can call those functions with a zero gas price.

All transactions are controlled by the `TxPermission` contract: it doesn't allow creating zero gas price transactions unless they are permitted on the current block (see the `TxPermission.allowedTxTypes` function). This protects against abusive use of a zero gas price: the service functions mentioned above can only be called when it is permitted, so they cannot be invoked too often to spam the network.

9.2.13 Claiming reward

After the reward is accrued and distributed, a participant can take it from the balance of the `BlockRewardAuRa` contract. To do that, the participant must call the `StakingAuRa.claimReward` function. This must be called by the participant’s *staking address* (for validators/candidates) or the delegator’s address (for delegators). The function can be called at any time without restrictions. The caller must pass the function a *staking address* of the pool from which they want to take their rewards, and an optional array of past staking epochs (an array of their numbers in ascending order). The array of staking epochs can also be left empty - allowing a participant to claim rewards for all past staking epochs. The function transfers the total reward amount (staking tokens and native coins) from the `BlockRewardAuRa` contract's balance to the caller's address according to their reward shares.

If the function requires too much gas when the empty staking epoch list is specified as input, a participant can specify explicit staking epochs and receive a partial reward by conducting several transactions. This can happen if a participant staked into a pool and didn't withdraw their reward for a long period of time (e.g. a couple of years or even longer). To get the list of staking epochs for which a specified staker can claim their rewards from a specified pool (staking address), the staker can use the `BlockRewardAuRa.epochsToClaimRewardFrom` getter, passing it the pool's staking address along with their own (staker's) address.

9.3 OpenEthereum client for AuRa

In order to launch the POSDAO network with AuRa, the stable OpenEthereum version has been modified to work with the POSDAO contracts. AuRa implementation in the latest OpenEthereum client contains the following features implemented since v2.7.2:

Note prior issues reference below have been added to this backports issue: https://github.com/openethereum/openethereum/issues/118

- Validator nodes automatically take part in random number generation by making calls to the randomness contract. In the commit phase, they generate a random number and publish its hash on the contract. In the reveal phase, they publish the number itself. The final random number is generated from all of the validators' contributions (see https://github.com/openethereum/openethereum/pull/10946).
- Validators call the `reportMalicious` function (see the OpenEthereum Wiki) if they observe other validators not following the protocol. We extended reporting to report when a validator tries to produce siblings blocks (see https://github.com/openethereum/openethereum/pull/11160).
- The validator node that creates a block in which the validator set changes, either because a new staking epoch begins or because a malicious validator is about to be removed, automatically emits the `InitiateChange` event (see the OpenEthereum Wiki and https://github.com/openethereum/openethereum/pull/11245).
- The transaction permission contract and its interface were extended, allowing validators to make most calls to governance contracts with a zero gas price, so that the signing address doesn't need a non-zero balance. That includes all of the above calls. The extended interface now allows filtering not only by sender, contract address and value, but also by gas price and transaction data (see https://github.com/openethereum/openethereum/pull/11170).
- We implemented a mechanism for validators to directly push their governance-related transactions onto blocks they are preparing, so they don't have to go through the transaction queue first. Some transactions, like malice reports, use both mechanisms, so that even if a node has been removed from the validator set at the end of a staking

epoch, it can still send its reports to others (see https://github.com/openethereum/openethereum/pull/11245).

- We added the `posdaoTransition` spec option to allow activation of all POSDAO features at a specified block (see https://github.com/openethereum/openethereum/pull/11245).
- We increased signature requirements to ⅔ of validators rather than ½ to protect against the cloning attack scenario (added `twoThirdsMajorityTransition` spec option, see https://github.com/openethereum/openethereum/pull/10909).
- We added the ability to change the BlockReward contract address multiple times (the `blockRewardContractTransitions` spec option). This enables an existing network to migrate to POSDAO at a specified block number. See https://github.com/openethereum/openethereum/pull/10875 for more info.
- The `stepDuration` spec parameter has been modified to allow for multiple step transitions on an AuRa chain. If step duration (block time) needs to change on an existing network, it is now possible to use the extended `stepDuration` parameter. See https://github.com/poanetwork/open-ethereum/issues/122#issuecomment-481556441 for instructions on use. Implemented in https://github.com/openethereum/openethereum/pull/10902 and https://github.com/openethereum/openethereum/pull/11379.
- We added the ability to define the block gas limit in a contract through the `blockGasLimitContractTransitions` spec option (see https://github.com/poanetwork/open-ethereum/issues/119#issuecomment-511379844). This option overwrites the default block gas limit calculations based on the `gas_floor_target` and `gasLimitBoundDivisor` parameters and allows setting the gas limit for each block dynamically. Implemented in https://github.com/openethereum/openethereum/pull/10928.
- `parity_clearEngineSigner` RPC method has been added in https://github.com/openethereum/openethereum/pull/10920 to provide the ability to deactivate validator's second (reserve) node when a primary node goes back to work again (see https://github.com/poanetwork/posdao-test-setup/issues/39).

Update 06/16/2021: Nethermind client v1.10.71+ now supports all POSDAO features and is the recommended client for implementation purposes.

9.4 xDai POSDAO network parameters

Constants used in the xDai POSDAO reference implementation

Constant	Value	Unit
MAX_CANDIDATES	3000	candidates (including validators)

MAX_VALIDATORS	19	validators
CANDIDATE_MIN_STAKE	20000	STAKE_UNITs
DELEGATOR_MIN_STAKE	200 <*updated from 1000 12/9/2020*>	STAKE_UNITs
STAKE_UNIT	10**18	native coin or ERC20 token with 18 decimals
SYSTEM_ADDRESS	2**160 - 2	-
BAN_PERIOD	90	days
STAKING_EPOCH_PERIOD	7 (120992)	days (blocks)
COLLECTION_ROUND_LENGTH	76	blocks
STAKE_WITHDRAW_DISALLOW_PERIOD	6 (4332)	hours (blocks)
BRIDGE_ENTRANCE_FEE	1	%
BRIDGE_EXIT_FEE	1	%

MAX_* constant values were obtained as a result of stress testing with AuRa where the step duration is equal to 5 seconds and block gas limit is 10 million.

The STAKING_EPOCH_PERIOD and COLLECTION_ROUND_LENGTH were set according to the formulas

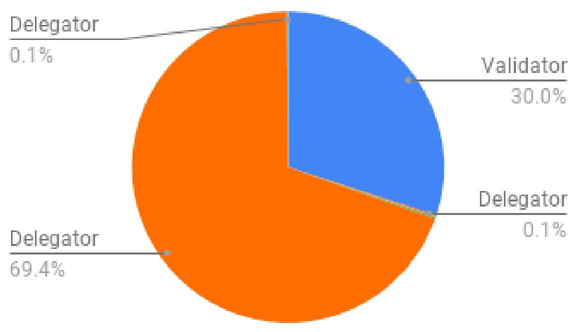
```
COLLECTION_ROUND_LENGTH % MAX_VALIDATORS === 0
STAKING_EPOCH_PERIOD % COLLECTION_ROUND_LENGTH === 0
```

to eliminate validator cartels and make sure every validator can participate in random number generation during the last collection round of an epoch.

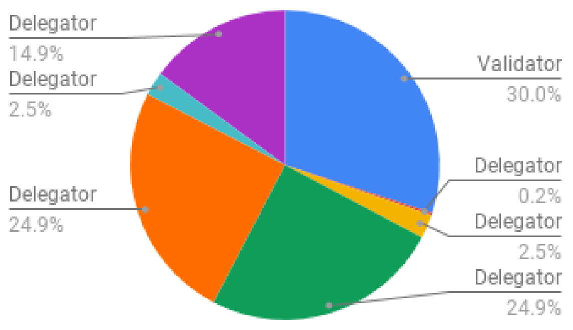
Appendix A: POSDAO Reward Distribution

A working version where values can be manually configured is available here (copy the spreadsheet to edit): Total sum staked: 554,200 Common reward for all pools: 100

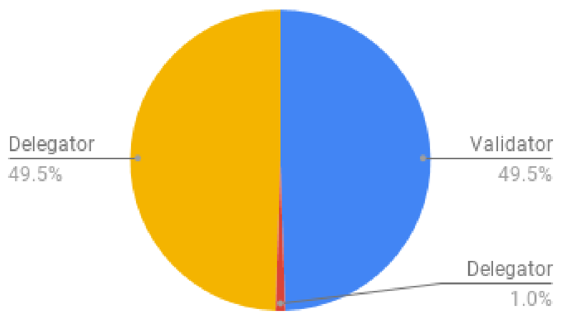
Pool1	Staked sum	Reward %	Reward tokens
Validator	2000	30	10
Delegator	1000	0.1388888889	0.0462962963
Delegator	1000	0.1388888889	0.0462962963
Delegator	1000	0.1388888889	0.0462962963
Delegator	500000	69.44444444	23.14814815
Delegator	1000	0.1388888889	0.0462962963
Total:	506000		33.33333333
Delegators:	504000		



Pool2	Staked sum	Reward %	Reward tokens
Validator	10000	30	10
Delegator	100	0.2491103203	0.08303677343
Delegator	1000	2.491103203	0.8303677343
Delegator	10000	24.91103203	8.303677343
Delegator	10000	24.91103203	8.303677343
Delegator	1000	2.491103203	0.8303677343
Delegator	6000	14.94661922	4.982206406
Total:	38100		33.33333333
Delegators:	28100		



Pool3	Staked sum	Reward, %	Reward, tokens
Validator	5000	49.5049505	16.50165017
Delegator	100	0.9900990099	0.3300330033
Delegator	5000	49.5049505	16.50165017
Delegator	0	0	0
Delegator	0	0	0
Delegator	0	0	0
Total:	10100		33.33333333
Delegators:	5100		



Appendix B: DAI-to-xDai / ERC20-to-Native bridge example scenario

Assumptions:

- Validators/delegators are elected and have bridged POSDAO (STAKE) tokens
- Conversion rate is 1 DAI = 1 xDai
- Bridge entrance fee is 1%, and exit fee is 1%
- Same consensus validators are active for entrance and exit. If different staking epochs, different validators and delegators may receive entrance/exit fee

Validator Pool Stakes:

- Validator Pool 1: Validator 1 has 100 STAKE and 2 delegators have 100 STAKE each (300 total)
- Validator Pool 2: Validator 2 has 100 STAKE and 1 delegator has 300 (400 total)
- Validators Pools 3-10 are single entities validators with 200 STAKE each

A user wants to use the xDai POSDAO chain for scalability purposes. They transfer 100 DAI into the xDai POSDAO chain.

1. 100 DAI are sent to the mainnet bridge contract.
2. 100 DAI are locked in the bridge smart contract.
3. Bridge entrance fee of 1 xDai native coin is minted on the xDai side and distributed to active validator pools (1 DAI = 1 xDai = 0.1 xDai for each of 10 pools)
4. The remaining 99 xDai are minted via a smart contract on the xDai side and sent to the user's address.
5. User conducts a lot of transactions, ending up with .3 xDai in transaction fees.
    1. Transaction fees are accrued as they occur. They are sent to the validator (only the validator, not the delegators) responsible for sealing the block where the transaction occurred.
    2. When finished, the user wants to return to the mainnet and has 90 xDAI remaining:
        1. 8.7 xDai remains on the xDai POSDAO chain, paid to others
        2. .3 xDai went to transaction fees
        3. 1 xDai went to the bridge entrance fee
6. User exits the xDai POSDAO chain. The bridge smart contract on the xDai side extracts exit fee = .9 xDai. Once the fee is extracted, 89.1 xDai are burned.
7. Bridge exit fee is distributed to active validator pools (.9 xDai= .09 xDai for each of 10 validators' pools)
8. 89.1 DAI are unlocked in the contract on the mainnet.

Total fees extracted:

- 1 xDai on bridge entrance
- .9 xDai on bridge exit
- .3 xDai in transaction fees (for simplicity lets say 4 separate validators participated in these transactions in an equal manner and the gasPrice/gasLimit were equal)

Validator rewards collected:

- Validator Pool 1: 0.19 xDai
    - Validator: .0634 xDai (34%) + .075 xDai in transaction fees
    - Delegator 1: .0633 xDai (33%)
    - Delegator 2: .0633 xDai (33%)

- Validator Pool 2: 0.19 xDai
    - Validator = .057 xDai (30%) + .075 xDai in transaction fees
    - Delegator: .133 xDai (70%)

- Validator Pool 3 - 4: 0.19 xDai
    - Validator = 0.19 xDai + .075 xDai in transaction fees

- Validator Pools 5 - 10: 0.19 xDai
    - Validator = 0.19 xDai

Appendix C: POSDAO / ERC677-to-ERC677 bridge example scenario

Assumptions:

- Current validators/delegators are elected and have bridged POSDAO (STAKE) tokens
- Weekly inflation rate is 15%/48 for STAKE tokens on the xDai POSDAO chain only
- Bridge entrance fee is 1%, and exit fee is 1%
- The same consensus validators are active for all staking epochs. This is unrealistic, but allows for simplified calculations.
- The total amount of STAKE ERC677 staked over the month (4 staking epochs) is 18,000, resulting in an additional 56.25 STAKE ERC677 (15 percent of 18000 tokens divided by 48 weeks) distributed over the course of 1 staking month.

Validator Pool Stakes:

- Validator Pool 1: Validator 1 has 100 STAKE and 2 delegators have 100 STAKE each (300 total)
- Validator Pool 2: Validator 2 has 100 STAKE and 1 delegator has 300 (400 total)
- Validators Pool 3: Validator 3 has 200 STAKE and 0 delegators (new delegator chooses this pool)
- Validator Pools 4 -10 start as single entities validators with 500 STAKE each.

A new delegator user wants to join the xDai POSDAO chain and stake some tokens! They transfer 100 STAKE to the xDai POSDAO chain.

1. 100 STAKE ERC677 are sent to the mainnet bridge contract.
2. 100 STAKE ERC677 are locked in the bridge smart contract.
3. Bridge entrance fee of 1 STAKE ERC677 staking token (1%) is minted on the xDai POSDAO side and distributed to active validator pools (1 STAKE ERC677 = 0.1 STAKE ERC677 for each of 10 pools)
4. The remaining 99 STAKE ERC677 are minted via a smart contract on the xDai side and sent to the delegator's address.
5. The delegator stakes all 99 STAKE on Validator 3's pool.
    1. The delegator will not receive any rewards for the current staking epoch. However, in the next staking epoch, the validator 3 pool is elected again and the delegator starts accruing rewards.
    2. The delegator remains with the active validator pool 3 for 4 staking epochs (1 staking month). At the end of the month, the delegator signals for a stake withdraw.

    1. Over the course of the month, 10,000 additional DPOS tokens were subject to entrance/exit fees. (There were probably additional delegators and validators added, and validator set changes, but for simplicity in example calculations let's say the validator/delegator sets remained static). This means 100 STAKE were collected in fees (1%). There are 10 pools, so 10 STAKE were distributed to each validator pool.
    2. Over the course of the month, the inflation rate resulted in 56.25 additional STAKE distributed as block rewards. Assuming validator sets did not change and all validators participated equally, 5.625 STAKE were distributed to each pool.
    3. For Validator pool 3, the delegator had ~33% stake, and a total of 0.1+10+5.625=15.725 tokens were added to the pool. This means the delegator received ~ 5.19 STAKE ERC677 tokens over the course of the month.

6. The delegator exits the xDai POSDAO chain with ~ 104.19 STAKE ERC677 tokens (original 99 + 5.19 in rewards). The bridge smart contract on the xDai side extracts a 1% exit fee = ~ 1.04 STAKE ERC677. 104.19 STAKE tokens are burnt and 1.04 STAKE tokens are minted.
7. The bridge exit fee is distributed to the active validator pools (1.04 STAKE ERC677 = .104 STAKE for each of 10 validators' pools)
8. 103.15 (104.19 - 1.04) STAKE ERC677 are unlocked in the contract on the mainnet. This is the delegator's new balance in STAKE ERC677 tokens.

Total fees extracted from the delegator:

- 1 STAKE ERC677 on bridge entrance
- 1.04 STAKE ERC677 on bridge exit

Validator pool rewards collected (through 1 month, all are approximate and based on simple assumptions): In this scenario, over the course of 4 staking epochs, each pool has received:

- 10.1 STAKE ERC677 as bridge exit/entrance fees
- 5.625 STAKE ERC677 as token inflation
- .104 STAKE ERC677 for bridge exit of example delegator

This results in the following approximate distributions:

- Validator Pool 1: 15.829 STAKE ERC677
    - Validator: 5.276 STAKE ERC677 (33.3%)
    - Delegator 1: 5.276 STAKE ERC677(33.3%)

    - Delegator 2: 5.276 STAKE ERC677 (33.3%)

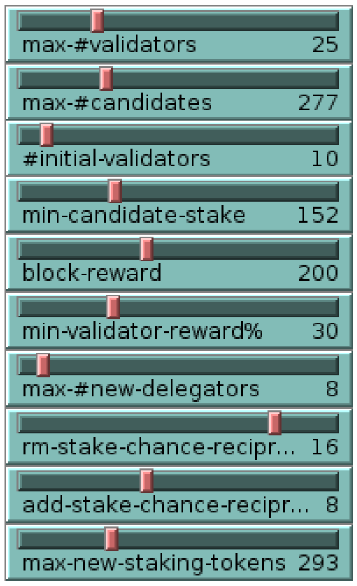
- Validator Pool 2: 15.829 STAKE ERC677
    - Validator = 4.75 STAKE ERC677 (30%)
    - Delegator: 11.079 STAKE ERC677 (70%)

- Validator Pool 3: 15.829 STAKE ERC677
    - Validator =   10.57 STAKE ERC677 (66.8%)
    - Delegator: 5.259 STAKE ERC677 (33.2%) - does not include .104 exit fee

- Validator Pools 4 - 10: 15.829 STAKE ERC677
    - Validator = 15.829 STAKE ERC677

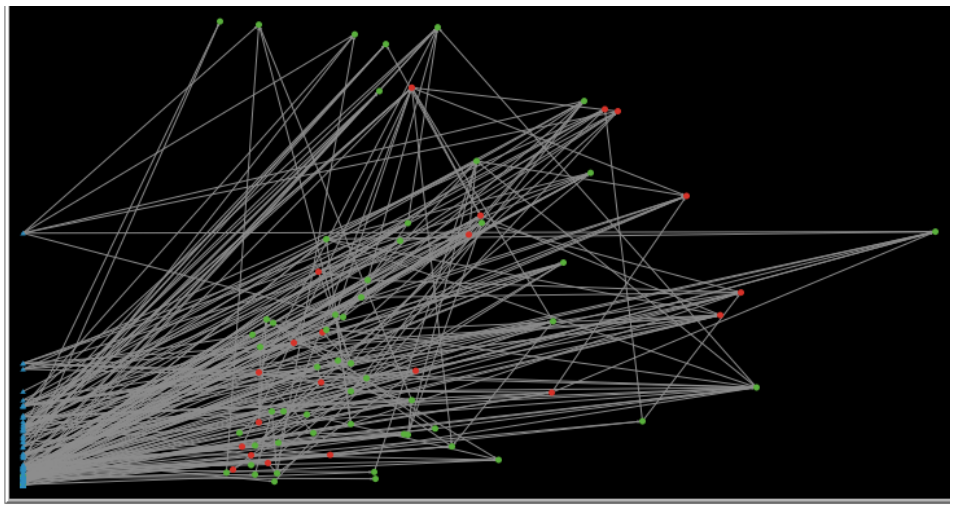
Appendix D: DPOS modeling

We modeled behavior of DPOS consensus participants using NetLogo[20]. The model takes several input parameters which constrain randomized steps of the modeling algorithm:

 max-#validators 25; max-#candidates 277; #initial-validators 10; min-candidate-stake 152; block-reward 200; min-validator-reward% 30; max-#new-delegators 8; rm-stake-chance-recipr... 16; add-stake-chance-recipr... 8; max-new-staking-tokens 293	Basic DPOS parameters: ● the maximum size of the validator set ● the maximum allowed amount of candidates ● the size of the initial validator set ● the minimum amount of candidate stake (expressed in staking tokens) ● the block reward (*BR*, expressed in reward tokens) ● the minimum validator reward share, in percents ● other control variables for stepping the model.

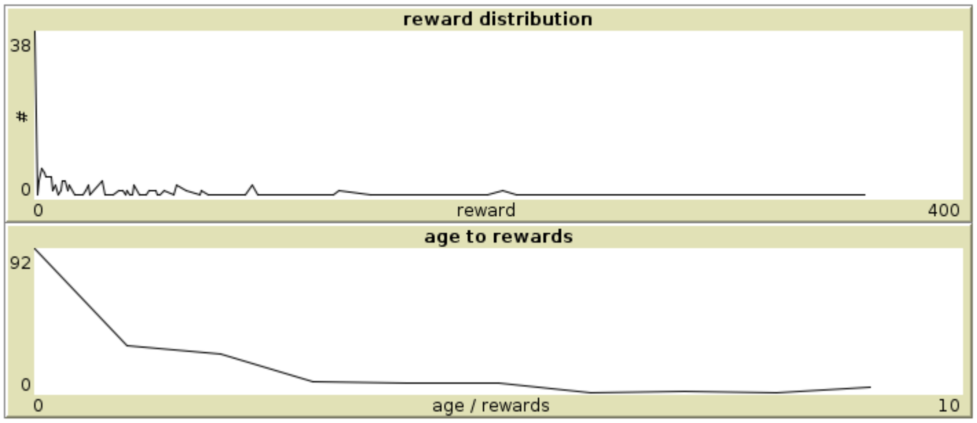
Each step of the model represents one staking epoch. The model abstracts away from the actual duration of a staking epoch. After 56 steps (modeling about a year in real terms assuming week-long staking epochs) we arrived at the following spread of network participants:

- horizontally with respect to sizes of the staking pools (from 0 for delegators to +infinity)
- vertically with respect to rewards the participants accrued (0 to +infinity).

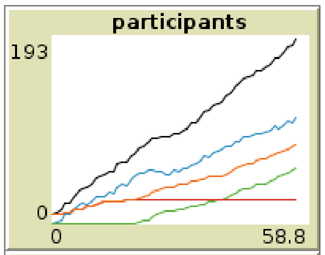
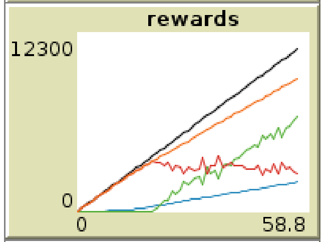
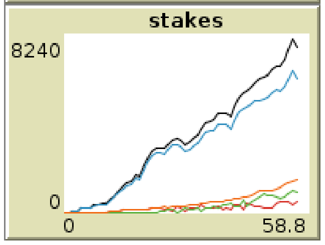


It is apparent from the network graph that rewards are assigned to candidates (green and red) in greater amounts compared to proper delegators (blue). This is mostly due to the fact that there were more proper delegators (participants that did not become candidates) than candidates.

Analysis also revealed that participants were rewarded proportionally to the time they spent on the network:



Additional network statistics

	The “participants” chart displays changes through all staking epochs (from 0 to 56) in numbers of all participants (black), delegators (blue), all candidates including current validators (orange), proper candidates excluding current validators (green), and current validators (red).
	The “rewards” chart displays the rewards accrued by participants in each of these categories.
	The “stakes” chart displays the amounts staked by participants in these categories.

A working model where values can be manually configured is available here.

Appendix E: Honey Badger BFT (HBBFT) integration

The Honey Badger BFT (HBBFT) consensus algorithm allows nodes in a distributed, potentially asynchronous environment to achieve agreement on transactions. The agreement process does not require a leader node, tolerates corrupted nodes, and makes progress in adverse network conditions. See the HBBFT github repository for more information.

We are currently integrating HBBFT with the OpenEthereum client to provide an additional consensus option for POSDAO. Modifications to OpenEthereum include:

- **Block agreement process:** HBBFT consensus is reached on a proposed list of transactions *before* a block is produced. This list is comprised of the union of all (or most) transactions proposed by the acting validators and each validator creates an identical block. The block is signed collaboratively using threshold signatures.
- **RNG generation**: HBBFT can create secure randomness from numbers that are only revealed after agreement has been reached on their contents. This randomness can be used to select new validator sets at the beginning of a staking epoch, and as a secure source of randomness for smart contracts.
- **Block validation:** Blocks sealed with a threshold signature need to be accepted as valid. OpenEthereum must keep track of the current validator set in order to match the threshold signature with the validator set's master key.
- **Block creation:** To increase throughput, new blocks can be created as soon as the transactions for the previous block have been determined, even if the previous block has not yet been sealed.
- **Governance contract interaction:** OpenEthereum needs to request the validator set to determine validator set changes, then use HBBFT to enact these changes. OpenEthereum should also report malicious validator behavior (fault reporting) to contracts.
- **Validator set changes/key generation:** Validator sets are transitioned over several blocks, and this process requires a set of threshold keys for the new validators, which must be generated "on-chain" (using a separate smart contract).
- **Networking:** To communicate efficiently, nodes must exchange low-latency, high-bandwidth targeted messages. When the validator set changes, new validators must establish direct connections to one another.
- **Network startup:** Items must be added to the chain specification so they are included in the genesis block. This includes compiled governance contracts and the public master key. In addition, corresponding key shares must be distributed to initial network validators (this may be accomplished outside of the network) prior to network start.

References

[1] de Vries, A (2018). Bitcoin’s growing energy problem. Joule, 2(5), 801-805. https://doi.org/10.1016/j.joule.2018.04.016
[2] J. Siim, "Proof-of-stake", *Research Seminar in Cryptography*, 2017. https://courses.cs.ut.ee/MTAT.07.022/2017_fall/uploads/Main/janno-report-f17.pdf
[3] Saleh, F. (2018). Blockchain Without Waste: Proof-of-Stake. SSRN Electronic Journal. 10.2139/ssrn.3183935.
[4] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi (2014). Cryptocurrencies without proof of work. CoRR, abs/1406.5694. https://arxiv.org/pdf/1406.5694.pdf
[5] Buterin, V., and V. Griffith (2017). Casper the Friendly Finality Gadget. CoRR, https://arxiv.org/abs/1710.09437.
[6] AuRa (Authority Round) Consensus Engine https://openethereum.github.io/Aura
[7] Miller, A., Xia, Y., Croman, K., Shi, E., Song, D. (2016). The Honey Badger of BFT Protocols. In: Cryptology ePrint Archive 2016/199 https://eprint.iacr.org/2016/199.pdf
[8] D. Larimer (2017). DPOS Consensus Algorithm – The Missing White Paper. https://steemit.com/dpos/@dantheman/dpos-consensus-algorithm-this-missing-white-paper
[9] V. Buterin (2014). A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper
[10] Gazi P., Kiayias A., Russell A. (2018). Stake-Bleeding Attacks on Proof-of-Stake Blockchains. 2018 Crypto Valley Conference on Blockchain Technology (CVCBT), 85-92. https://allquantor.at/blockchainbib/pdf/gazi2018stake.pdf
[11] “Nothing at stake attack ethereum.” [Online]. Available: https://github.com/ethereum/wiki/wiki/Problems
[12] Li, W., Andreina, S., Bohli, J., Karame, G. (2017). Securing Proof-of-Stake Blockchain Protocols. Data Privacy Management, Cryptocurrencies and Blockchain Technology. Springer, Cham, 297-315. https://www.researchgate.net/publication/319647471_Securing_Proof-of-Stake_Blockchain_Protocols
[13] Kanjalkar, S., Kuo, J., Yunqi Li, Y. & Miller, A. (2019). Short Paper: I Can’t Believe It’s Not Stake! Resource Exhaustion Attacks on POS. Financial Cryptography 2019. http://fc19.ifca.ai/preproceedings/180-preproceedings.pdf
[14] “Fake Stake” attacks on chain-based Proof-of-Stake cryptocurrencies. [Online]. Available: https://medium.com/@dsl_uiuc/fake-stake-attacks-on-chain-based-proof-of-stake-cryptocurrencies-b8b05723f806
[15] Parinya Ekparinya, P., Gramoli, V., Jourjon, G. (2019). The Attack of the Clones against Proof-of-Authority. https://arxiv.org/abs/1902.10244
[16] Parinya Ekparinya, P., Gramoli, V., Jourjon, G. (2018). Impact of Man-In-The-Middle Attacks on Ethereum https://doi.org/10.1109/SRDS.2018.00012
[17] Randao: Verifiable Random Number Generation (2017) [Online]. Available:

https://randao.org/whitepaper/Randao_v0.85_en.pdf
[18] Alturki, M. & Rosu, G. (2018). Statistical Model Checking of RANDAO's Resilience Against Pre-computed Reveal Strategies.
https://www.ideals.illinois.edu/bitstream/handle/2142/102076/rdao-analysis.pdf
[19] Gencer, A.E., Basu, S., Eyal, I., van Renesse, R., Sirer, E.G (2018). Decentralization in Bitcoin and Ethereum Networks. https://arxiv.org/pdf/1801.03998.pdf
[20] NetLogo. https://ccl.northwestern.edu/netlogo/

Version Log

Version	Date	Changes
v1.0	3.29.2019	● initial specification
v1.1	4.09.2019	● automatic withdrawal of token orders replaced by manual token claims [4.5, 9.2.8] ● long range attack solution revised to include reasoning and link to weak subjectivity [7.1] ● reward distribution code rewritten and optimized [9.2.7] ● threshold for the `reportMalicious` function increased from ½ to ⅔ [9.2.6] ● xDai DPOS network parameter constants adjusted to: 1500 MAX_CANDIDATES & 19 MAX_VALIDATORS [9.4]
v1.2	4.29.2019	● note on paper layout to address different audiences [Intro] ● added MAX_DELEGATORS_PER_POOL parameter in response to stress testing; it is necessary to limit the amount of delegators to maintain a 5 second block time in AuRa [4.1, [9.4] ● clarified snapshotting process which occurs at the end of each staking epoch [4.2, 9.2.2.5] ● provided additional information around checkpointing to mitigate a long range attack [7.1]
v1.3	5.20.2019	● Reference implementation network parameters updated based on stress testing results in production mode. MAX_CANDIDATES increased from 1500 to 3000 and MAX_DELEGATORS_PER_POOL increased from 200 to 3000 [9.4]
v1.4	6.11.2019	● Explanation and parameters related to native coin staking (single token) in addition to the dual token environment. ○ `_erc20Restricted` parameter in `InitializedAuRa` contract [9.2.1] ○ Native inflation distribution added [9.2.7.5] ● DPOS_INFLATION_RATE parameter was removed due to an updated the inflation formula [3]. [Appendix C] updated with example inflation rate. ● Functionality extended to allow contracts deployment on an existing AuRa network rather than invocation only on the genesis block. [9.2.1] ● Various Figures updated to reflect MAX_VALIDATORS = 19.

		• Additional `blockRewardContractTransitions` spec option added. [9.3]
v1.5	6.24.2019	• Additional `posdaoTransition` spec option added. [9.3] • AuRa `stepDuration` spec parameter extended. [9.3]
v1.6	7.22.2019	• Additional `blockGasLimitContract` spec option added. [9.3] • Value of the COLLECTION_ROUND_LENGTH parameter decreased from 200 to 114. [9.4]
v1.7	3.30.2020	• Updated terminology to include OpenEthereum (Rather than Parity) and STAKE token (rather than generic DPOS token). • DPOS ERC20 tokens renamed to POSDAO ERC677 tokens (and ERC20-TO-ERC20 bridge mode changed to ERC677-TO-ERC677). • Update image terminology. • Added qualifier to Equal Block Reward mechanism. If validators skip blocks their rewards are reduced accordingly [6.2.2] • `StakingAuRa.claimReward` function added (reward strategy updated from PUSH to PULL, where participants pull rewards) [9.2.13] • Pending validator set functionality now uses boolean flag rather than a queue. (`ValidatorSetAuRa._pendingValidatorsChanged`) [9.2.4] • Debugging functionality added via `RandomAura.setPunishForReveal` function where punishments may be temporarily removed. [9.2.5] • Added ERC20-to-Native bridge reward from Dai Savings Rate [9.2.7.2] • `parity_clearEngineSigner` RPC method added [9.3] • xDai POSDAO network parameters updated [9.4]
v1.7.1	5.19.2020	• Update previous Parity wiki links to direct to OpenEthereum wiki
v1.8	9.25.2020	• TxPriority contract added [9.1]
v1.9	6.16.2021	• Added Nethermind Client implementation [1.4] • Updated 70/30 reward distribution in reference implementation to straight proportional allocation [3.1.1] • Added potential implications for tx fee distribution related

		to EIP1559 [3.1.1]