RADICLE REGISTRY
SPECIFICATION
VERSION 0.1

MONADIC[†]

1. Transactions

All transactions on the registry take the form $\mathsf{transaction}(arg_1, \ldots, arg_n)_\sigma$, where $arg_1, \ldots arg_n$ are the *inputs* and σ is the EdDSA signature of the author of the transaction. Transactions always have an *author* and an *origin* (formally α), which is the author's account.

Transactions can be uniquely identified by their *hash*.

2. Accounts

An account A is a tuple:

$$A = \langle A_{\mathsf{id}}, A_{\mathsf{nonce}}, A_{\mathsf{bal}} \rangle$$

DEFINITION

– A_{id} is the unique account identifier obtained by hashing the account owner's public key,
– A_{nonce} is a number which starts at 0 and is incremented every time a transaction originates from this account.
– A_{bal} is the account's balance in the smallest denomination, and $A_{\mathsf{bal}} \in \mathbb{N}_{\geq 0}$.

The set of all accounts is $\mathcal{A}$. Accounts are never created or destroyed, rather, if they have never been used to transact, they have an initial state of:

$$A = \langle A_{\mathsf{id}}, 0, 0 \rangle$$

Hence, for all valid account ids, there exists an account with that id. In other words, $\forall a \in A_{\mathsf{id}}(A \in \mathcal{A})$.

Note that accounts can *never be removed*, since that would violate the invariant that nonces are only ever incremented, and removing an account is equivalent to setting A_{nonce} and A_{bal} to 0.

2.1. **Root accounts.** Some accounts are considered *privileged*. These 'root' accounts, formally $\mathcal{A}_R \subset \mathcal{A}$ are authorized to conduct certain transactions that are only valid when originating from these accounts.

The set of accounts in $\mathcal{A}_R$ is defined at *genesis*, and may not be further modified in the initial protocol.

2.2. **Transferring value.** The act of transferring coins between two accounts:

$$\mathsf{transfer}(A_{\mathsf{id}}, v)_\sigma$$

which will transfer value from the transaction origin α to account A.

INPUTS

– A_{id} is the account id of the *receiver* of the transfer,
– v is the value or 'balance' to transfer from the origin to the receiver, in the smallest denomination.

VALIDATION

– The transfer balance is positive, or $v \geq 1$,
– The origin's balance minus any transaction fee is $\geq v$.

OUTPUTS

– v is debited from the origin and credited to A.

3. Projects

A project P is a tuple:

$$P = \langle P_{\mathsf{id}}, P_{\mathsf{k}}, P_{\mathsf{account}}, P_{\mathsf{contract}}, P_{\mathsf{proof}}, P_{\mathsf{meta}} \rangle$$

DEFINITION

– P_{id} is the unique project identifier, defined as $\langle P_{\mathsf{name}}, P_{\mathsf{domain}} \rangle$,
– P_{k} is the current project *checkpoint* (See §3.3),
– P_{account} is the project account or *fund*,
– P_{contract} is the project contract, which governs permissions around the project, as well as its fund. It can be invoked as a function which takes a transaction t as input, and returns an output in $\{\top, \bot\}$, or:

$$P_{\mathsf{contract}}(t) \rightarrow \{\top, \bot, \mathbb{N}\}.$$

An output of $\top$ signifies *authorized*, while $\bot$ is for *unauthorized.*
– P_{proof} is the proof that was supplied during registration, verifying the owner's authority over the project,
– P_{meta} is a dictionary of additional metadata to associate with the project, for example the RADICLE *project id.* P_{meta} is *immutable* once defined.

Projects are created with the register-project transaction. The set of all projects is $\mathcal{P}$.

[†]Monadic, `team@monadic.xyz`.

3.1. **Project registration.** The act of registering a project under a unique name and domain:

$$\mathsf{register\text{-}project}(P_{\mathsf{id}}, P_{\mathsf{k}}, P_{\mathsf{proof}}, P_{\mathsf{meta}})_\sigma$$

INPUTS

- $P_{\mathsf{id}} = \langle P_{\mathsf{name}}, P_{\mathsf{domain}} \rangle$
- P_{name} is the unique name being requested, where
- P_{domain} is the domain under which P_{name} is being registered, which together form the unique identifier P_{id},
- P_{k} is the id of the initial *checkpoint* associated with this project, formally k_0. This checkpoint must always remain in the project ancestry.

VALIDATION

- P_{name} must be unique, i.e. not currently registered under P_{domain}, between 1 and 32 characters long, and valid UTF-8,
- P_{domain} must be an existing domain,
- P_{k} must represent an existing checkpoint.

OUTPUTS

- $P \in \rho$, where ρ is the set of registrations that are in a "pending" state, waiting to be accepted or rejected (See §**??**).

3.2. **Accepting and rejecting a project.** The act of accepting or rejecting a project being registered:

$$\mathsf{accept\text{-}project}(t_{\mathsf{hash}})_\sigma$$

or

$$\mathsf{reject\text{-}project}(t_{\mathsf{hash}})_\sigma$$

INPUTS

- t_{hash} is the *transaction hash* of the register-project transaction t of a project P being accepted or rejected.

VALIDATION

- The transaction *origin* is a member of $\mathcal{A}_R$,
- t_{hash} must be the hash of an existing transaction of type register-project. In other words, $P \in \rho$, where P is the project being registered.

For accept-project,

OUTPUTS

- $P \in \mathcal{P}$
- $P \notin \rho$
- $P_{\mathsf{account}} = \langle A_{\mathsf{id}}, 0, 0 \rangle$

For reject-project,

OUTPUTS

- $P \notin \mathcal{P}$
- $P \notin \rho$

3.3. **Checkpointing.** The act of notarizing a project's state and updating the network graph:

$$\mathsf{checkpoint}(K_{\mathsf{parent}}, K_{\mathsf{hash}}, K_{\mathsf{version}}, K_{\mathsf{contribs}}, K_{\mathsf{deps}})_\sigma$$

Checkpoints form a chain going from the latest checkpoint to the first.

INPUTS

- K_{parent} is the *id* of the previous or 'parent' checkpoint,
- K_{hash} is the new *hash* of the project state,
- K_{version} is the new *version* of the project,
- K_{contribs} is the list of contributions since K_{parent},
- K_{deps} is the list of dependency updates since the K_{parent}.

VALIDATION

- K_{parent} refers to an existing checkpoint in the registry, or is $\varnothing$.
- K_{hash} is a valid hash that hasn't been used in a parent checkpoint.
- K_{version} is a string between 1 and 32 bytes long that hasn't been used in a previous project checkpoint.
- K_{contribs} is a valid contribution list (See §3.3.1).
- K_{deps} is a valid dependency update list (See §3.3.2).

3.3.1. *Contributions.* The list K_{contribs} supplied to the checkpoint transaction is of the form:

$$K_{\mathsf{contribs}} = [\langle C_{\mathsf{parent}}, C_{\mathsf{hash}}, C_{\mathsf{author}}, C_{\mathsf{sig}} \rangle],$$

DEFINITION

- C_{parent} is the hash of the parent contribution, or $\varnothing$ if this is the first contribution of the first checkpoint of the project.
- C_{hash} is the hash of the corresponding commit,
- C_{author} is the public signing key of the commit referred to by C,
- C_{sig} is the author's GPG signature.

VALIDATION

- C_{parent} is a valid SHA-1 hash or $\varnothing$ if this is the first contribution. Note that if C is K_{contribs}'s first item, and C' is the *last* item of the *parent* checkpoint's contributions list, then C'_{hash} and C_{parent} must be equal, such that no gaps between contributions exist.
- C_{hash} is a valid SHA-1 hash,
- C_{author} is the creator of C_{sig},
- C_{sig} is a valid signature of C_{hash}.

Because all changes to a project's source code are described in checkpoints, it is possible to reconstruct a full hash-linked list of contributions for the entire project. When cross-referenced with the project's repository, this constitutes a complete historical record of who authored what code. This ensures the project history is auditable and tamper-proof, while providing fundamental information to for the network graph $\mathcal{N}$. Note that only contribution *metadata* is stored on-chain.

3.3.2. *Dependency updates.* Conceptually, a project P depends on another project P' if it is an "input" to P in some way: P references P' or parts of P' in its source code, or P' is a build/test dependency.

The dependency update list P_{deps} is a list of *dependency updates*, one of:

$$\begin{cases} \mathsf{depend}(P'_{\mathsf{id}}, P'_{\mathsf{version}}) \\ \mathsf{undepend}(P'_{\mathsf{id}}, P'_{\mathsf{version}}) \end{cases}$$

which refer to the project P' at a specific version P'_{version}. The depend update adds a new dependency while the undepend update removes a dependency. The updates are processed in order with depend only being valid if it adds a dependency that the project does not already have and undepend only being valid for current dependencies. The checkpoint is invalid if the update list contains duplicates.

VALIDATION

- P'_{id} must be a valid project id, but *does not* have to refer to an existing id in the registry. This allows dependent projects to checkpoint dependencies that have not yet been registered.
- P'_{version} must be a valid version string, but *does not* have to refer to an existing version of P'. This allows dependent projects to checkpoint before their dependencies.

As a project maintainer, adding a dependency signals a variety of things depending of the nature of the project:

- They have verified that P indeed depends on this specific version of P'.
- That P' is suitable as a dependency for P, e.g. if P has very high security requirements, that P' fulfills these.

Since contributions to a project carry additional weight—potentially increasing a project's rank—there is an incentive for maintainers to checkpoint their projects regularly. Similarly, adding dependencies may increase connectivity in the network graph, which may in turn indirectly improve a project's rank.

3.4. **Setting the project checkpoint.** The act of updating the project to point to a new checkpoint:

$$\mathsf{set\text{-}checkpoint}(P_{\mathsf{id}}, k')_\sigma$$

which updates $P_{\mathsf{checkpoint}}$ from k to k'.

INPUTS

- P_{id} is the id of the project being updated,
- k' is the id of the checkpoint the project should be associated to.

VALIDATION

- P_{id} refers to a project that has been accepted in the registry,
- k' is a checkpoint which has the original project checkpoint k_0 in its ancestry.
- $P_{\mathsf{contract}}(\mathsf{set\text{-}checkpoint}(P_{\mathsf{id}}, k')_\sigma) \equiv \top$

OUTPUTS

- $P_{\mathsf{k}} = k'$

3.5. **Updating the project contract.** The act of updating a project's contract:

$$\mathsf{set\text{-}contract}(P_{\mathsf{id}}, c)_\sigma$$

INPUTS

- P_{id} is the id of the project,
- c is the new contract.

VALIDATION

- $P \in \mathcal{P}$,
- $P_{\mathsf{contract}}(\mathsf{set\text{-}contract}(P_{\mathsf{id}}, c)_\sigma) \equiv \top$

OUTPUTS

- $P_{\mathsf{contract}} = c$

4. Names

4.1. **Registering a domain.** The act of registering a top-level domain:

$$\mathsf{register\text{-}domain}(domain)_\sigma$$

INPUTS

- *domain* is the unique domain being registered.

VALIDATION

- The transaction origin α is a member of $\mathcal{A}_R$,
- *domain* must be available for registration, between 1 and 32 characters long, and valid UTF-8.

For example,

$$\mathsf{register\text{-}domain}(\texttt{crates})_\sigma$$

5. Identity

5.1. **Identifying as a contributor.** The act of identifying yourself as a contributor, by linking a public key used to sign project contributions, to an account in the registry:

$$\mathsf{identify}(I_{\mathsf{pk}}, I_{\mathsf{proof}})_\sigma$$

INPUTS

- I_{pk} is the public key that is to be associated with the *origin* account α if this transaction succeeds.
- I_{proof} is a proof verifying that the transaction author owns I_{pk}.

VALIDATION

- I_{pk} is not already associated with an account,
- I_{proof} is α_{id} signed by the secret key sk that I_{pk} was derived from. In other words,

$$I_{\mathsf{proof}} = \mathsf{encrypt}(\mathsf{hash}(\alpha_{\mathsf{id}}), sk)$$

which is valid if

$$\mathsf{decrypt}(I_{\mathsf{proof}}, I_{\mathsf{pk}}) \equiv \mathsf{hash}(\alpha_{\mathsf{id}})$$

5.2. **Forgetting identities.** When a public key associated with the identify transaction is lost or no longer used, the following transaction will 'forget' the association:

$$\mathsf{forget}(I_{\mathsf{pk}})_\sigma$$

Appendix

Alternative contribution model. It's possible to drastically reduce the size of checkpoints by merkleizing the commit history and making contribution claiming an explicit action on the part of the contributor.

First, we change the checkpoint transaction's K_{contribs} argument from a *list* of commits to a single *hash*. This hash is the Merkle root of the commit list that was K_{contribs}, with $\langle C_{\mathsf{hash}}, C_{\mathsf{author}}, C_{\mathsf{sig}}\rangle$ at each leaf:

$$K_{\mathsf{contribs}} = \mathsf{hash}([\langle C_{\mathsf{hash}}, C_{\mathsf{author}}, C_{\mathsf{sig}}\rangle]),$$

where hash is a *Merkle* hash function. K_{contribs} is now constant-sized. Note that in this alternative model, C_{parent} is no longer used, since it can't be verified.

Then, we ask contributors to claim contributions if they wish to receive rewards, with a new claim-contribution transaction:

$$\mathsf{claim\text{-}contribution}(C_1 \ldots C_n, C_{\mathsf{proof}})_\sigma$$

INPUTS

- $C_1 \ldots C_n$ is the list of commits to claim, where $C = \langle C_{\mathsf{hash}}, C_{\mathsf{sig}}\rangle$, and $C_{\mathsf{author}} = \alpha$, the author of the transaction. Note that C_{author} is not actually included in the transaction, since it is assumed to always be α.
- C_{proof} is the Merkle proof asserting that $C_1 \ldots C_n$ are included in the K_{contribs} set. It is expected that C_{proof} can use structural sharing to compress the Merkle paths such that the transaction does not grow linearly in n. Formally, each C_{proof} is an ordered set $\{\pi_1 \ldots \pi_n\}$ of Merkle paths associated with $C_1 \ldots C_n$.

VALIDATION

- C_{proof} is valid if each path π hashes to the same root hash r, and there is a checkpoint K such that $K_{\mathsf{contribs}} = r$.
- For each $C \in C_1 \ldots C_n$, C_{sig} is a signature by the author α.

OUTPUTS

- $\alpha_{bal'} = \alpha_{bal} + v$, where $\alpha_{bal'}$ is the author's new account balance, and $v = P_{\mathsf{contract}}(t)$ where t is the claim-contribution transaction.