COLONY

Technical White Paper

2018-02-16 — commit cb41534

Alex Rea[*] Daniel Kronovet[†] Aron Fischer[‡] Jack du Rose[§]

Organizations, for the Internet

The Colony Network is an Ethereum-based protocol for creating and operating Internet Organizations. In a traditional organization, rules are defined in policy documents and enforced by a management hierarchy; in an Internet Organization, rules are defined in code and enforced by a blockchain mining process.

Automating business rule enforcement makes organizational models which previously involved high coordination costs, such as Holacracy, more feasible. More exciting still, this automation enables entirely new and previously impossible organizational forms to come into being. By reducing the trust needed for people to collaboratively manage shared funds and enforce standards of conduct, it becomes possible to coordinate the entirety of an organization's operations via market-style interactions, rather than through the internal processes of a firm.

An Internet Organization's operations are not *necessarily* analogous to traditional company processes, such as project management, budgeting, and decision-making, although they certainly *can* be, and indeed Colony's own application (`https://colony.io/connect`) is modeled that way. We anticipate the Colony protocol will be used to build economic incentives and decision-making mechanisms for platforms whose content or value proposition is created by users *without top down decision-making of any kind.* Examples may include a photo-sharing app which meritocratically divides revenue between users, a driver-run P2P ridesharing service, crowdsourced claims handling in an insurance dApp, or a merchants' guild in a virtual world.

This is primarily a technical document describing the current design of the Colony protocol, focusing on on-chain actions and affordances. In many cases off-chain functionality, such as messaging or other forms of signalling, will be essential for the proper functioning of the organization. However, as that communication is not consensus-relevant, it does not need to happen on-chain, so the specifics of how it might occur are not treated here.

Onward!

[*] `alex@colony.io`
[†] `krono@colony.io`
[‡] `aron@colony.io`
[§] `jack@colony.io`

Contents

1 Overview **3**
- 1.1 Preamble . . . 3
- 1.2 Theory of the firm . . . 3
- 1.3 Confidence and trust . . . 4

2 Structure of a Colony **6**
- 2.1 Domains and permissions . . . 6
- 2.2 Funding and expenditures . . . 8
- 2.3 Internal tokens . . . 10
- 2.4 Revenue and rewards . . . 11
- 2.5 The reputation system . . . 13
- 2.6 Managing stakes . . . 17
- 2.7 Upgradability and security . . . 19
- 2.8 Arbitrary transactions . . . 20

3 Extending Functionality **21**
- 3.1 Tasks . . . 21
- 3.2 Funding queues . . . 23
- 3.3 Budget box . . . 26
- 3.4 Motions and disputes . . . 27
- 3.5 Miscellaneous . . . 32

4 The Colony Network **35**
- 4.1 Revenue model . . . 35
- 4.2 The Metacolony and CLNY . . . 36

5 Reputation Mining **38**
- 5.1 Merkle-Patricia trees and proofs . . . 38
- 5.2 The Reputation Tree . . . 39
- 5.3 Calculating the new root hash . . . 39
- 5.4 Submission of a new root hash . . . 40
- 5.5 Dealing with false submissions . . . 41
- 5.6 Calculating reputation updates . . . 47
- 5.7 Denial of service attacks . . . 48
- 5.8 Costs and rewards of mining . . . 50
- 5.9 Emergency shutdown . . . 51

6 Conclusion **52**

References **52**

A Appendices **53**
- A.1 Gas-Efficient Reputation Penalty in Dispute Resolution . . . 53
- A.2 Reputation Decay Calculation Details . . . 56

1 Overview

1.1 Preamble

This paper is split into sections corresponding to logically separate components of the Colony Network. The Colony Network's features however, are in some cases quite interdependent, making it difficult to explain a feature without referencing another before it has been introduced. This overview therefore provides a high level overview of Colony's capabilities to contextualise the more technical expositions which follow.

Section 1, this introduction, lays out the organizational theory which inspires and guides the project. Section 2, **Structure of a Colony**, details the basic functionality of a colony, which can be thought of as an organization's operating system. Much like a computer operating system provides a secure interface for managing the system's underlying resources, a colony provides a secure interface for managing an organization's underlying resources. As with a computer operating system, a colony can be extended with specific applications to implement specific behaviors. We call these applications **Extensions**, and are the subject of Section 3. Section 4 is concerned with the **Colony Network** as a whole, covering the governance, operation, and token economics of the Colony Network and the special Metacolony. Section 5 is specifically concerned with the **Reputation Mining** process, which powers Colony's reputation system.

Throughout this document various numerical parameters are specified. These values will be subject to review as we gather evidence from the field, and any parameter values proposed in this document should be seen as good-faith suggestions, not final judgments. Similarly, nothing in this document should be interpreted as a guarantee that any described functionality will be either developed or deployed. The version of the Colony Network that is live on Ethereum mainnet should be considered complete as-is, and the reader should not assume, irrespective of the content of this document, that any existing functionality will be upgraded beyond its current state.

1.2 Theory of the firm

Companies exist to coordinate the production of goods and services. Transaction Cost Economics (TCE) theory, popularised by Ronald Coase's 'Theory of the Firm' [1], postulates that companies form, employ people, and invest in capital because there is a threshold at which it is more efficient to control the factors of production directly than to coordinate production via the market mechanism, once transaction costs are accounted for. These transaction costs come in three flavours:

- **Search & information**: Costs associated with finding information to inform decisions, and discovering and evaluating suppliers.
- **Bargaining**: These are costs associated with reaching an agreement with a supplier. Bargaining costs can be very low (e.g. buying a coffee), or very high (e.g. buying a company).
- **Monitoring & enforcement**: The costs of ensuring adherence to the terms of an agreement (e.g. that widgets are manufactured on time and to the agreed quality). People often deviate from the agreed terms due to chance, negligence, or malice, and potentially high enforcement costs (e.g. legal fees) are required to resolve disputes.

TCE theory states that firms are more efficient than the market mechanism at coordinating production due to *imperfect information* and *bounded rationality.* Given perfect information, companies would not be necessary, as market forces would provide the necessary mechanisms to incentivise and coordinate production — everyone would know the exact value of their and other's contributions. As this is not the case in traditional markets, these knowledge and trust barriers are overcome by due diligence and contracts, and require a legal system to provide recourse when things go wrong. These processes are expensive, and so traditional firms often find that replacing free-market bargaining with command-and-control hierarchy makes them more efficient and competitive.

As new technologies have improved the diversity and flow of information, new organizations are emerging which are able to merge the efficient decision-making of a market with the shared value-capture of a traditional firm. Gig economy platforms (e.g. Uber, Airbnb), market networks (e.g. eBay, Amazon Marketplace), and cryptocurrencies (e.g. Bitcoin, Ethereum) have demonstrated that if *the product is sufficiently well defined, and the supply sufficiently large, fungible, or diverse*, then it is possible to dramatically reduce the transaction costs of the market mechanism by making search and information discovery easy, bargaining straightforward, and having policing and enforcement provided essentially for free by the platform. This has enabled these new platforms to be orders of magnitude more efficient than had they attempted to coordinate equivalent supply within the hard boundaries of a firm.

1.3 Confidence and trust

The firm is able to coordinate complex production at scale by organising labour into a management hierarchy. Seniority within the hierarchy (ideally) represents the amount of confidence the company has in the employee, and in the Platonic ideal of a firm, confidence is a pure function of competence. The more confident the company is in their employee, the higher their competence, and thus the greater their responsibility, influence, and compensation.

Across the internet however, it's hard to have confidence in other people. Up to now we've relied on platform operators to mediate relationships between parties in online transactions (often via various rating and reputation systems), and in some cases (such as payment processing), to underwrite the risk of those transactions. On the blockchain it's even harder, as all you know about a counterparty is that they control a public key. It is difficult to imagine how a traditional organization or hierarchy could exist in this pseudonymous, adversarial environment. A blockchain has no geographical boundaries and cannot differentiate between who or *what* controls public keys. As Richard Gendal Brown put it in his twist on Peter Steiner's classic meme: *'On the blockchain, nobody knows you're a fridge.'*

Internet Organizations must thus assume the lowest common denominator: that every member is rationally self interested and focussed entirely on maximising personal utility and profit, and given incentives accordingly. This gets to the heart of Colony: a protocol seeking to facilitate the same kind of meritocratic division of labour and authority that the idealised model of the corporate hierarchy should, except from the bottom up, and less prone to error. Decentralised, self-organising companies, where decision-making power derives from a fairly-assessed contribution of value.

Work therefore, is where we start. A colony member is compensated for the value they create for the colony, in the form of ETH, any ERC20-compatible token, or *Reputation*, a non-fungible, time-decaying measure of aggregate past contributions.

Active colonies are likely to have a variety of types of work ongoing at any given time; in order to ease the management of work (and their budgets), colonies can be divided into *Domains*. Domains are how you structure your colony. You can think of them as teams, departments, circles, or whatever makes sense in your context. These make it easy to group related tasks together and separate them from other unrelated work in other domains, and make it possible to incorporate contextually-appropriate decision-making logic (in which one domain may be controlled by an administrator, while another is controlled by reputation-weighted voting).

When a colony member gets paid in the colony's internal token, they also receive Reputation for the *Skills* they used, and in the *Domain* in which the value was created. Reputation is used to quantify the historical contributions of members to a colony, and to make sure they are fairly rewarded. By earning Reputation in a *Skill* (e.g. Javascript) and a *Domain* (e.g. BigCo Client Project), the recipient earns proportional influence in decisions pertaining to those skills and domains. Reputation is not transferable between accounts, and slowly decays over time. This decay ensures that any reputation a member has is as a result of recent behaviour deemed beneficial to the colony (and thus a function of the judgment of the current membership). As the calculations involved are too complex to carry out on the Ethereum blockchain, updates to a member's reputation are calculated off-chain, with an on-chain reporting mechanism secured by economics and game theory (See Section 5).

Many decisions within a colony can be made by informal consensus. Members are expected to verify their colleagues conduct, but hopefully will only rarely need to intervene. Intervention in this context means 'making a motion' and is the subject of Section 3.4. Decision via vote is infrequent within Colony because it is slow and carries a high coordination cost; costs which are justified in the (hopefully rare) case of dispute resolution. The dispute resolution system allows for many kinds of decisions to be put to a contextually-relevant vote of some or all members of the colony. Ballots are weighted meritocratically, according to voters' contextually-relevant reputation.

Colonies may be voluntary, non-profit, or for profit. A revenue-generating colony may elect to pay out a portion of its revenue to its members. When the colony pays out rewards, the amount a member receives is a function of their combined token *and* reputation holdings; this ensures those who have contributed the most gain the greatest benefit. Members maximise rewards by contributing to a colony over its whole lifetime (thus maintaining high levels of reputation) rather than sitting on early accumulations of tokens. The details of the rewards payout process can be found in Section 2.4.

We want people to use Colony for as many different workflows as possible, even those that are not immediately apparent as being able to leverage the Colony protocol. Section 3.5 provides a brief outline of some more complex behaviours that we have envisaged as possible with the tools described here.

2 Structure of a Colony

Colonies exist to enable collaboration between their members, and direct collective efforts towards common goals. Facilitating effective division of labour, management of incentives, and allocation of resources are therefore some of the most important functions of the Colony protocol.

2.1 Domains and permissions

The essential structure of the colony revolves around domains and the permissions that accounts may have in them. These two concepts jointly define the structure and security of a colony and provide a flexible framework for creating colonies of many kinds.

2.1.1 Domains

Like any organization, without structure, a large colony would quickly become difficult to navigate due to the sheer number of participants and interactions taking place — domains solve this problem. A domain is like a folder in a shared filesystem, except instead of containing files and folders, it can contain subdomains, funding, and expenditures. This simple modularity enables great flexibility as to how an organisation may be structured. Domains could be used to represent teams, departments, projects, tribes, circles, and so forth. A toy example is shown in in Figure 1.

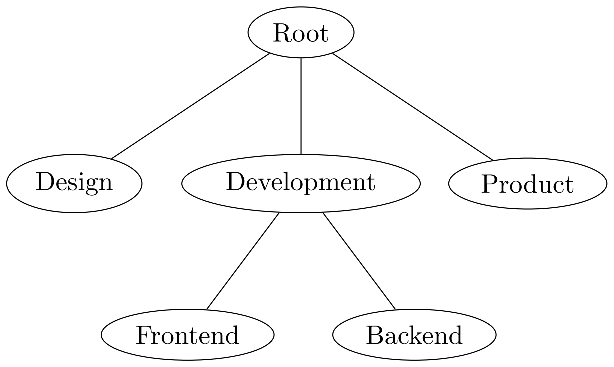

Figure 1: Parts of a domain hierarchy for a colony developing a web service.

It is ultimately up to individual colonies to decide how they wish to use domains—some might only use them for coarse categorisations, whereas others may use them to precisely group only the most similar expenditures together, or even multiple expenditures that other colonies would consider a single expenditure. Some might use domains to represent long-lived organizational departments, while others might use them more ephemerally, to represent projects with start and end dates. We aim to provide a general framework that colonies may use however they see fit, and to be prescriptive only where necessary.

Among other things, this compartmentalisation of activity provides an essential benefit to the colony as a whole by making reputation *contextual.* When arbitration occurs, it occurs at a specific level in the colony's domain hierarchy. This means that people with relevant contextual knowledge can be included for their opinion, and that when arbitration occurs, the whole colony is not required to participate in the process.

2.1.2 Permissions

Access control in a colony is organized around the concepts of **permissions**. There are six different permissions (roughly in order of influence): recovery, root, arbitration, architecture, funding, and administration, each unlocking a bundle of semantically-related functionality.

With the exception of the recovery and root permissions, all permissions are domain-specific (much like permissions in a Unix file system are directory-specific), with the rule that permissions held in a parent domain are inherited in all child domains. Put another way, have a permission in a domain gives you that permission in the entire *subtree* rooted in that domain. To implement this inheritance, permissioned functions require a *domain proof* of the following arguments:

- `permissionDomainId` - The (parent) domain in which the account holds the permission.
- `childSkillIndex` - The index of `domainId` in `permissionDomainId`'s `children` array.
- `domainId` - The (child) domain in which the action is being taken.

These arguments can be evaluated on-chain in constant time to determine whether the account is authorized to call the privileged function.

Permissions are held by Ethereum accounts. This means that permissions may be given to human administrators, or assigned to contracts which implement more complex behavior (such as voting mechanisms). These types of contracts are known as **extensions** and are discussed in-depth in Section 3. The use of extensions to flexibly 'plug-in' various decision-making mechanisms is a key concept in the Colony protocol.

It is worth noting that the list of accounts that have the permission in question have the full permission; no additional restrictions exist at the protocol level. In some cases, these are extremely powerful capabilities (such as emitting arbitrary reputation penalties) and require absolute confidence in whomever or whatever controls it. We anticipate therefore that in many cases, extension contracts will be used to offer varying degree of moderation to the underlying permissions.

Recovery

The recovery permission gives accounts access to the colony's emergency 'recovery' functionality, which allows for arbitrary state-changes to the colony's data. Recovery mode is described in more detail in Section 2.7.

Root

The root permission gives accounts access to high-level administrative functions in the colony, such as setting colony-wide parameters, upgrading the colony, and minting new internal tokens. This permission also gives accounts the ability to assign permissions throughout the colony (including in the root domain).

Arbitration

The arbitration permission gives accounts the ability to make domain-specific state changes, meant as a means of resolving motions. This permission also enables accounts to emit reputation penalties (but not reputation increases).

Architecture

The architecture permission gives accounts the ability to create new domains in a colony, as well as assign permissions in those new domains. Unlike root, accounts with this permission cannot edit permissions in the domain in which they hold the permission, only in subdomains.

Funding

The funding permission gives accounts the ability to move tokens between funding pots. In practice, this means that this permission is responsible for allocating money amongst domains and for funding expenditures. Financial management in a colony is described in more detail in Section 2.2.

Administration

The administration permission gives accounts the ability to create and manage (but not fund) expenditures, the basic incentive unit in a colony, described in Section 2.2.

Broadly, permissions are designed as a 'separation of powers': different permissions must work in concert to carry out the functioning of a colony. For example, administration can create an expenditure, but only funding can actually provide the resources, while arbitration resolves motions as they arise. Complex extensions may require multiple permissions in order to function properly (such as 'tasks', which requires both arbitration and administration).

The intention is that, since permissions are grouped into semantic bundles of functionality, it will be possible to develop *specialized* mechanisms for mediating access to the underlying functionality (i.e. specialized funding mechanisms and specialized dispute-resolution mechanisms, as opposed to a general-purpose 'voting' mechanism meant to handle all possible decisions).

Colony's long-term vision is of *trustless organizations*; organizations in which members can safely collaborate and manage shared resources, without needing to know or trust each other. Early colonies may find that a larger emphasis on human moderators to be useful, while more mature colonies may find reasons to devolve increasingly more decision-making to extensions implementing trustless functionality. We will refer to colonies which make substantial use of these extensions as *trustless colonies.*

2.2 Funding and expenditures

All tokens and currencies are administered by the colony contract; it is responsible for all the bookkeeping and allocations. The former are managed via funding pots, the latter via expenditures.

2.2.1 Funding pots

Each domain and each expenditure in a colony has an associated *funding pot.* A funding pot can be thought of as a wallet specific to a particular domain or expenditure, and are used to move funds around *within* a colony. To each funding pot, the colony contract may associate any number of Ether or ERC20-compatible tokens it holds. Depending on context, the funds in a funding pot may be referred to as the payout, bounty, budget, salary or working capital. In addition to the funding pots, there is a special *rewards pot* which accumulates tokens to be distributed to members as *rewards* (see Section 2.4).

Only accounts holding the **funding** permission may move tokens; the rule is that they may move tokens between any two pots in the subtree rooted in the domain in which they hold the permission. It is the expectation that this permission will in many cases be given to an *extension contract* implementing a specialized decision-making mechanism, such as the *funding queue* described in Section 3.2.

2.2.2 Expenditures

The basic payment primitive of a colony is the 'expenditure'. Expenditures are used to transfer funds out of a colony to any Ethereum account. An expenditure has several properties:

- An `owner` (the account address which created the expenditure).
- A `status` (active, cancelled, or finalized).
- One or more `recipients`.
- `payouts` for each recipient, denominated in one or more tokens.
- Optionally, a per-recipient `skill`.
- Optionally, a per-recipient `payoutModifier`.
- Optionally, a per-recipient `claimDelay`.

The owner is responsible for setting the properties of the expenditure. The recipients are simply Ethereum accounts. While it is anticipated that recipients will be individuals, there is nothing to prevent these accounts being contracts under the control of multiple people.[1]

Once the expenditure is finalized, all properties become locked (but subject to arbitration) and payouts can be claimed (and reputation awarded). Prior to finalization, the owner has the ability to cancel the expenditure entirely. Any funds that have already been assigned to the expenditure can be reassigned to the domain that the expenditure was created in.

Defining the payouts for each recipient, of course, does not provide the funds — this must be done through the funding mechanisms in Colony. Payouts do not have to all be in the same token, and an expenditure's payouts can be made up of an arbitrary number of tokens.

The expenditure is meant to be an abstract primitive which can support many types of workflows, and so contains optional attributes to support more complex behavior (see Section 3). For instance, the `payoutModifier` and `claimDelay` can be used to implement a rating and review system, where good or bad reviews lead to an across-the-board reputation increase (or payout decrease) for a recipient, while the `claimDelay` is set to allow for any relevant motions to be decided before funds can exit the colony.

Once the tokens have been received by an account, they are under the control of the recipient — there is no way to reclaim the funds. The funds have to cross the 'Cryptographic Rubicon' somewhere in the system (by the nature of the blockchain), and it makes sense to do so here.[2]

[1]With the protocol as described in this document, any reputation earned would be assigned to the contract in question and not able to be moved to the appropriate users. In these cases, it might be better to develop an extension contract which would determine the per-user allocation in advance and configure the expenditure accordingly.

[2]Currently, the only way this rule can be broken is by the Colony conspiring to abuse the 'arbitrary transaction' feature described in Section 2.8.

2.3 Internal tokens

Every colony has its own ERC20-compatible ‘internal token’. These are the tokens that, when earned as an expenditure payout, also generate reputation for the receiver (and thus distribute control within the colony). What these tokens represent apart from this is up to the colony to decide. For example, they may have financial value, or they may be purely symbolic; some possible scenarios are outlined in this section.

In addition, colonies may ‘bring their own token’ and designate an existing ERC20-compatible token as reputation-bearing. While this may be advantageous in some contexts, it’s worth noting that this weakens the incentive alignment underpinning the game theoretic security of trustless colonies, in that the value of the token is divorced from the performance of the colony. Note that the internal token cannot be changed once a colony has been created, so *choose wisely.*

In cases where a colony creates a new token, that colony is in control of the supply of the token. Specifically, **root** permission holders can mint tokens at-will. In some cases, this may look like a founder managing the token supply unilaterally, while in other cases colonies may manage the minting process via an extension contract (see Section 3.5.1 for an example).

A common question is why only internal tokens (as opposed to all tokens) are reputation-bearing. The reason for having a single token be reputation-bearing is that it avoids tricky exchange-rate problems, such as incentives to receive more of a less valuable token to earn more reputation.

2.3.1 Token use-cases

Ultimately, internal tokens are used to distribute reputation, and thus both ownership and decision-making power. Since users with more reputation can both exercise more influence over the activity of the colony, as well as claim a greater share of the rewards, reputation functions to align incentives among the members of colony. Here we give a few examples of different use-cases for internal tokens, demonstrating the variety of schemes colonies may adopt for distributing ownership and influence alongside cash compensation.

Tokens as early rewards

One of the chief benefits of a colony having its own token is that it can offer rewards for work before it has any revenue or external funding to draw on. A new colony may offer token payouts for expenditures with the hope that the reputation earned by these token payments (and the future revenue earned by the colony) will eventually lead to financial rewards. By allowing ‘spending’ before fund-raising, the financial burden during the start-up phase of a new colony is eased. Once a colony is profitable, payment in tokens may be the exception rather than the norm.

Tokens representing hours worked

We could imagine a colony in which all expenditures are paid in Ether, but include a number of the colony’s own tokens as well, equal to the expected number of hours worked. The members of the colony would be responsible for assigning ‘correct’ token and Ether payouts to expenditures. This extra responsibility would also ensure users doing the same amount of work received the same reputation gain, rather than the reputation gain being dependent on the rates they charged.

Tokens as performance-based bonuses

Alternatively, we could imagine a colony which seeks to balance predictable compensation (i.e. salaries) with performance-based incentives. Such a colony could pay out salaries in a token such as Ether or DAI, and reserve their internal token for performance-based bonuses (i.e. for hitting quarterly OKRs). Such an approach makes reputation (and decision-making power) a function of achievement, without making members of the colony feel as though their ability to pay rent depends on their ability to hit quarterly goals.

2.3.2 Colony's `Token` contract

Colony has developed a customized `Token` contract, with some additional functionality:

- `mint` — lets the token contract owner introduce new tokens into circulation.
- `burn` — lets anyone permanently remove tokens from circulation.

In addition, Colony's `Token` contract introduces the idea of 'locking' — tokens being non-transferrable until a one-way boolean flag is flipped. This is useful for colonies which want more control over how and when their tokens can be liquidated and exchanged.

This contract underlies the Colony Network Token (see Section 4.2) and is the default token contract for new colonies, although colonies are free to choose any ERC20-compatible token.

2.4 Revenue and rewards

A colony may sell goods and services in exchange for Ether or any ERC20-compatible tokens, and this revenue may be sent to the colony's address. Whenever a colony receives such payments, we say that the colony has earned *revenue*. Revenue is distinct from a colony's working capital: the latter is the sum of all tokens held by the colony in various domains (see Section 2.2), while the former is implicitly defined as the colony's token holdings not yet accounted for in any of the existing pots.

There is an expectation that some fraction of any Ether or other tokens received by the colony are paid out to their members. 'Members', in this context, means accounts holding both tokens and reputation in the colony. Whenever a colony distributes a portion of revenue to its members, we say that the colony is paying out *rewards*.

2.4.1 Processing revenue

Revenue accumulates as the colony receives transfers of tokens. In order to be processed, any user can make a special `claimColonyFunds` transaction, indicating for which token they wish to process accumulated revenue.

The transaction then calculates the amount of token-denominated revenue that has accumulated since the last such transaction, and transfers some proportion to the colony's rewards pot. The remainder is then made available to the colony as working capital. The percentage split is configurable by the **root** permission via the `setRewardInverse` function.

2.4.2 Claiming rewards from the rewards pot

Rewards accumulate in the rewards pot. To trigger a payout to users (i.e. to make rewards claimable) a **root** user makes a special `startNextRewardPayout` transaction (no more than once every 60 days), initiating a process by which all members may claim a payout based on the reward pot's holdings.

This reward payout transaction includes the specific currency that should be paid (reward payouts for each token are handled separately). Once the process begins, all users' tokens are locked until they claim their payout. Locking is necessary because the token balance of each account factors into the rewards formula of equation (1). Locking is done by incrementing the token's `totalLockCount`.

Our `TokenLocking` contract contains a locking mechanism ensuring that a user cannot move tokens while they have (token-weighted) votes to reveal; we use the same mechanism here to ensure that a user cannot move tokens after a payout is approved by the members of the colony but before the user has claimed their rewards. The colony has a counter for each user that is incremented whenever they claim a payout; they can also waive their claim to a payout that will increment this counter.

Rewards are only available to accounts that hold both tokens and reputation, and the amount claimable by each account depends on *both* token balance and reputation (see equation (1) below). Therefore we need to have a similar behaviour to 'lock' the reputation of the users for the payout. When a payout is activated, the current state of the reputation tree is recorded in the payout itself. Users are paid out according to their reputation in this state, rather than the most recent state, to ensure all users get an appropriate payout and to avoid exploiting the system (which might otherwise be possible via e.g. delaying reward collection until after completing an expenditure, increasing their reputation).

2.4.3 The rewards formula

The amount that each user (u_i) of a colony ($\mathcal{C}$) is entitled to claim (p_i) is a function of their colony token holdings (t_i) and their total reputation in the colony (r_i):

$$p_i = \left(\frac{t_i r_i}{T \times R}\right)^{\frac{1}{2}} \qquad \text{where} \quad T = \sum_{u_j \in \mathcal{C}} t_j \quad \text{and} \quad R = \sum_{u_j \in \mathcal{C}} r_j. \tag{1}$$

This is a (normalised) geometric average of the user's token holdings and reputation. We note that this is very unlikely to payout all the tokens set aside for a payout — the only way it would do so is if everyone had the same proportion of reputation in the colony as they did proportion of tokens in the colony. However, the geometric average is the natural way to fairly capture the influence of two variables with different ranges, and ensures that large token holders must earn large amounts of reputation to get the most from the payouts. The total reputation and user reputation in the colony are all provable on-chain at claim time via a Merkle proof that the `ReputationRootHash` (Section 5) contains some values claimed by the user; the user's balance of colony tokens and the total number of tokens issued is trivial to lookup.

After some sufficiently long period of time (60 days), all unclaimed tokens can be reclaimed on behalf of the colony by a user, and the payout closed. Any users that have not claimed their payout by that point will still have their tokens locked, and they will remain locked until they issue a transaction waiving their claim to the payout (indeed, they already passively did this by not claiming it in a timely fashion). Unclaimed tokens are returned to the rewards pot and become part of the next reward cycle.

2.5 The reputation system

Reputation is a number associated with each user which attempts to capture the value of that user's contributions to the colony over time. Reputation is used to weight a user's influence in decisions related to the expertise they have demonstrated, and to determine amounts owed to a colony's members when rewards are disbursed. Because reputation is awarded to users by either direct or indirect peer assessment of their actions, we argue that influence and rewards can be seen as being (roughly) distributed by merit. Colony's aim is that the reputation system will enable an emergent and dynamic decision-making hierarchy in which all of the right people are in the right places.

Colony aims to be broadly meritocratic. Consequently, the majority of decisions in a trustless colony are weighted by the relevant reputation. Unlike tokens, reputation cannot be transferred between accounts, as it represents an appraisal of the account's activities by their peers. Reputation must therefore be earned by direct action within the colony. Reputation that is earned will eventually be lost through inactivity, error, or misbehaviour; a description of how reputation is gained and lost is given in Section 2.5.2.

2.5.1 Types of reputation

Reputation by domain

The hierarchical domain structure of a colony was described in Section 2.1. Reputation is earned in this hierarchy, and a user has a reputation in all domains that exist — even if that reputation is zero. When a user earns or loses reputation in a domain, the reputation in all parent domains changes by the same amount. In the case of a user losing reputation, they also lose reputation in all child domains, but in this case the child domains lose the same *fraction* of reputation that was lost in the original domain. If a reputation update would result in a user's reputation being less than zero, their reputation is set to zero instead.

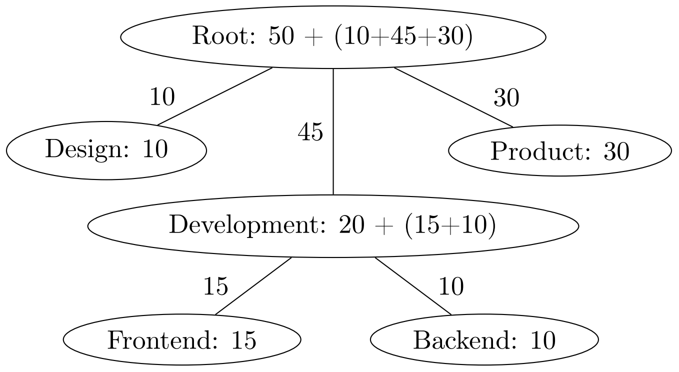

Figure 2: Reputation flowing up a domain hierarchy.

An example makes this clearer. Suppose a colony has a ‘development’ domain which contains a ‘backend’ domain and a ‘frontend’ domain, as in Figure 2. Any time a member of the colony earns reputation for work completed in the backend domain, it will increase their backend reputation, their development reputation and their reputation in the all-encompassing root domain of the colony. Reputation earned in the development domain will only increase the development and root domain reputation scores of the user.

Later, the user behaves badly in the ‘development’ domain, and they lose 100 reputation out of the 2000 they have in that domain. They also lose 100 reputation in the parent domains, and 5% $\left(\frac{100}{2000}\right)$ of their reputation in each of the child domains of the ‘development’ domain (which in this example, includes both frontend and backend domains).

Reputation by skill

We anticipate domains to mostly be used as an organisational hierarchy within a colony. However, this would not necessarily capture the *type* of work that a user completed to earn their reputation. If the domain were a project, with expenditures involving both design and development work, reputation earned by completing expenditures related to these skills would not be distinguishable. To have a more granular account of the work a user completes to earn their reputation, a skill cloud is also maintained.

This global cloud of skill tags is available to all colonies, and is curated and maintained by the Metacolony. When an expenditure is created, as well as being placed in a particular domain in the colony, may also be tagged with one or more skills from the skills cloud. When the recipient earns reputation by claiming the payout, they will earn reputation in all skills the expenditure was tagged with, with the reputation divided uniformly amongst the skills. This is in addition to the reputation earned in the relevant domains.

Even though the skills cloud is universal, specific skills reputation is unique to each colony. Earning reputation in a skill in one colony has no effect on the user’s reputation in that skill in any other colonies.

Reputation by colony

A user’s total reputation in a colony is their reputation in the root domain. This is the reputation they will be voting with in any decisions that require input from everyone in a trustless colony (i.e. modifying colony-wide parameters). Reputation in a colony has no effect outside the colony. In particular, reputations held in one colony have no bearing on reputations held by the same account in another colony.

2.5.2 Earning and losing reputation

There are three ways to receive reputation in a colony.[3] The first (and by far the most common) is through receiving a payout via an expenditure. The second is through the arbitration process. The third is upon the creation of a colony and the associated bootstrapping process.

Reputation losses broadly occur as the result of arbitration, and extension contracts (see Section 3) makes it possible to implement mechanisms which involve reputation penalties (such as tasks and disputes). In addition, all reputation earned by users is subject to a continual decay over time.

[3]The Metacolony is a special case where reputation may also be earned by reputation mining (see Section 5).

The rest of this section outlines each of these mechanisms, with references to the more detailed descriptions given elsewhere where appropriate.

Reputation change via expenditures

Whenever an expenditure recipient receives a payout denominated in the colony's internal token, the recipient also receives some amount of reputation, scaled by that recipient's `payoutScalar`. A value of 1 gives reputation equivalent to the token payout, but a multiple of up to 2x is possible. The reputation is earned in the domain (and all parent domains) of the expenditure, and divided equally among any skills associated with that recipient.

Reputation change as a result of arbitration

Arbitration permission holders have the ability to emit arbitrary reputation penalties (but not increases) in both domains and skills. While this might seem to be a significant power available to arbitration permission holders, recall that this permission will in many cases be assigned to extension contracts, which will mediate this ability via various mechanisms, such as the motions system (see Section 3.4).

Bootstrapping reputation

Since a trustless colony's decision making procedure rests on reputation weighted voting, we are presented with a bootstrapping problem for new colonies. When a trustless colony is new, no-one has yet completed any work in it and so nobody will have earned any reputation. Consequently, no motions can be made and no disputes can be resolved as no-one is able to vote. Then, once the first expenditure is paid out, that user has a dictatorship over decisions in the same domains or skills until another user earns similar types of reputation.

To prevent this, when a colony is created, the creator can choose accounts to have initial reputation assigned to them in the root domain to allow the colony to bootstrap itself. The reputation assigned to each user will be equal to the number of tokens received, i.e. if a member receives ten tokens, they also receive ten reputation in the root domain. Given that reputation decays over time, this initial bootstrapping will not have an impact on the long-term operation of the colony. This is the only time that reputation can be created without an associated expenditure being paid out. Users receiving reputation are presumably the colony founder and their colleagues, and this starting reputation should be seen as a representation of the existing trust which exists within the team.

We note that the same is not required when a new domain is created in a colony. We do not wish to allow the creation of new reputation here, as this would devalue reputation already earned elsewhere in the colony. This bootstrapping issue is resolved by instead using reputation within the parent domain, when a child domain contains less than 10% of the reputation of its parent domain. A domain below this threshold cannot have domains created under it.

Reputation decay

All reputation decays over time. Every 90 days,[4] a user's reputation in every domain or skill decays by a factor of 2. This decay occurs every 1 hour, rather than being a step change every 90 days

[4]It is likely that this parameter will be configurable on a per-colony basis in the future.

to ensure there are minimal incentives to earn reputation at any particular time. This frequent, network-wide update is the primary reason for the existence of the reputation mining protocol, which allows this near-continuous decay to be calculated off-chain without gas limits, and then realised on-chain.

The decay serves multiple purposes. It ensures that reputation scores represent *recent* contributions to a colony incentivising members to continually contribute to the colony. It further ensures that wild appreciations in token value (and the corresponding decrease in tokens paid per expenditure) do not permanently distort the distribution of reputation but instead serves to smooth out the effects of such fluctuations over time.

One might wonder why we have chosen to *decay* reputation, rather than pursue a strategy of reputation *dilution* via inflation. In one sense, they are equivalent: decaying reputation that is earned at a constant rate is the same as earning reputation at increasingly inflated valuations. Mathematically, however, decay is the cleaner approach, and so the use-case for inflation is that it is more feasibly calculated on-chain. In the case of Colony, reputation cannot be calculated on chain, since reputation updates effect an unbounded number of reputation nodes (due to the unbounded size of the domain tree). Since reputation cannot be calculated on chain, we choose to decay reputation in our off-chain reputation mining process.

2.5.3 On-chain representation of skills and domains

In the context of reputation, domains and skills are the same, differing only in that domains are colony-specific categorisation and skills are universal categorisation. In this subsection, each instance of 'skill' should be taken to mean 'skill or domain'.

Each skill that reputation can be earned in is assigned a `skillId` that is unique across the whole network. When a skill is created, additional properties are recorded and initialised.

$$\texttt{skillId} \rightarrow \begin{cases} \texttt{nParents} & \text{total number of parent skills.} \\ \texttt{nChildren} & \text{total number of child skills.} \\ \texttt{parents}\,[\cdots] & \text{array of } \texttt{skillIds} \text{ of a \textit{logarithmic subset} of parent skills,} \\ & \text{where } \texttt{parents[i]} \text{ gives the } 2^i\text{-th parent.} \\ \texttt{children}\,[\cdots] & \text{array of } \texttt{skillIds} \text{ of \textit{all} child skills.} \\ \texttt{globalSkill} & \text{whether the skill is a skill or domain.} \\ \texttt{deprecated} & \text{whether the skill has been deprecated.} \end{cases}$$

Upon creation, `nChildren` is 0 and `children[]` is empty. These two attributes in all parents are updated with the `skillId` of the new child skill on creation.[5]

Storing these pieces of data on-chain is required, as they are used by the reputation mining protocol (see Section 5) and the procedures for appealing motions (see Section 3.4). They are stored under the control of the `ColonyNetwork` contract.

[5]We acknowledge that this is fundamentally gas limited, but the only consequence of this will be the inability to create new skills once the maximum depth allowed by the block size is reached. Our calculations suggest this corresponds to a depth of around 80, which we believe is likely to be sufficient for the majority of use cases.

2.5.4 Reputation update log

Whenever an event that causes one or more users to have their reputation updated in a colony occurs, a corresponding entry is recorded in a log in the `ColonyNetwork` contract. Each entry in the log contains:

- The user experiencing the reputation loss or gain.
- The amount of reputation to be lost or gained.
- The `skillId` of the reputation to be lost or gained.
- The colony the update has occurred in.
- How many reputation entries will need to be updated (including parent, child and colony-wide total reputations). This is the motivation for storing `nParents` and `nChildren` for each skill and domain.
- How many total updates to reputations have occurred before this one in this cycle, including decays and updates to parents and children.

If the reputation update is the result of a dispute being resolved (as outlined in Section 2.5.2), then instead of these first three properties, there is a reference to the dispute-specific record of stakes in the relevant colony. For the structure of this log, and an explanation of the way that it allows individual updates to be extracted in constant gas, see Appendix A.1.

This log exists to define an ordering of all reputation updates in a reputation update cycle that is accessible on-chain. In the event of a dispute during the reputation mining protocol (described in Section 5), the `ColonyNetwork` contract can use this record to establish whether an update has been included correctly.

2.6 Managing stakes

Staking is a key concept in trustless systems, as a way to ensure that participants have 'skin in the game' and can be incentivized towards good behavior. As Colony wishes to enable an ecosystem of extensions implementing various cryptoeconomic mechanisms (see Section 3), a shared system for managing stakes improves usability and security by saving users from needing to send and retrieve tokens to and from many different contracts. In colonies, all stakes are denominated in that colony's internal token.

2.6.1 Storing Tokens

All stakes are stored in the network-wide `TokenLocking` contract. A singleton contract has the advantage that in scenarios where a user is a member of multiple colonies sharing the same internal token, a single deposit suffices for all colonies.

Any slashing of stakes occurs as a result of a function call coming from the *colony* and is a result of colony-specific arbitration logic.

2.6.2 Approvals and Obligations

Stakes are managed via a sequence of *approvals* and *obligations*. Users *approve* an account to then *obligate* them up to the maximum amount of their approval. If an obligation is made in excess of the deposits held in the `TokenLocking` contract, the transaction will fail. Once an obligation is made, a user cannot withdraw tokens if the withdrawal would result in a balance less than their obligations. At any time, the approved extension can *deobligate* the user, freeing the tokens for withdrawal (see Figure 3). In practice, we expect that the same underlying deposit will be obligated and deobligated repeatedly without the user needing to move any additional tokens.

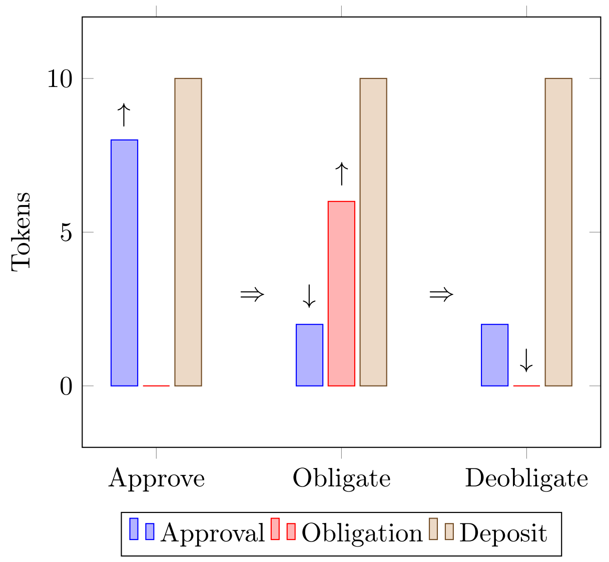

Figure 3: Example stake lifecycle with deobligation

While an obligation is active, any **arbitration** permission holder can *slash* the stake up to the amount of the obligation (see Figure 4). We reiterate that this is a powerful ability and in most cases should be mediated by an appropriate extension (such as motions, described in Section 3.4).

For reasons of security, approvals are keyed by *domain*, as well as by address of approvee (i.e. `approve(approvee, domain, amount)`). Otherwise, a malicious actor could use *any* arbitration permission holder in the colony to slash a stake, rather than arbitration permission holders in the intended domain inheritance path. However, because `TokenLocking` does not know about the domain structure of specific colonies, the obligations in `TokenLocking` are aggregates of all colony- and domain-specific obligations.

Overall, this design allows arbitration to be generalized and separated from the implementation of any particular extension: extensions obligate a stake (and define the period of obligation), while during that period separate arbitration processes can slash that stake.

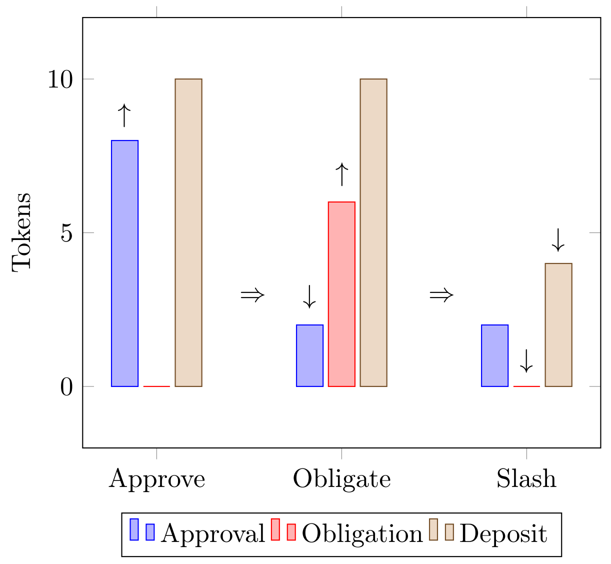

Figure 4: Example stake lifecycle with slashing

2.7 Upgradability and security

2.7.1 Upgradability

We foresee the Colony Network being continuously developed. Providing an upgrade path is important to allow people to use Colony without preventing themselves from using new features as they are added to the network.

We intend to allow colonies and tokens to be upgraded by using the pattern made available under the name EtherRouter [3]. This implementation uses two contracts in addition to the contract(s) providing the functionality implemented. The first additional contract is the `EtherRouter` contract, which passes on transactions — via `delegatecall` — to the contract that implements that function. The second additional contract is the `Resolver` contract, where the accounts of the contracts that implement the desired behaviour are defined. Whenever a transaction is received by the `EtherRouter` contract, it looks up the contract that implements that function (if any) in the `Resolver`, and then `delegatecalls` that contract.

In order to upgrade, new contracts are deployed with new functionality, and then contracts that the `Resolver` contract points to must be changed to point to these new contracts. In order to avoid a situation where the contract partially implements both old and new functionality, a new instance of `Resolver` will be deployed for each upgrade, and then a single transaction can point `EtherRouter` at the new `Resolver`. From the perspective of the colony, an upgrade is then simply swapping out one address (the `Resolver`) for another.

The choice of upgrading the underlying Colony contract will always fall to the colony, and never the Colony Network. While the network is in control of what upgrades are available, they are not able to force any colony to upgrade the underlying contracts. The colony itself must decide that it wants to upgrade to a new version.

2.7.2 Security

While we aspire to bug-free contracts, bugs are inevitable, and so the adoption of a 'defensive programming' mentality will limit the impact of any vulnerabilities that may be discovered in the deployed contracts.

The ultimate fallback is known as 'recovery mode'. In this state, whitelisted accounts (those with the **recovery** permission) are able to access special functions that allow the state of the contract to be directly edited — in practise, this will correspond to access to the functions to allow setting of variables, as well as being able to upgrade the contract. With the agreement of multiple whitelisted accounts, the contract will then be able to be taken out of recovery mode once the contract has been returned to a safe state. Removal from recovery mode requires the approval of multiple whitelisted accounts. This ensures that a single whitelisted account cannot, in a single transaction, enter recovery mode, make a malicious edit, and then exit recovery mode before the other parties on the whitelist have had a chance to react.

It is conceivable that colonies will be able to deactivate the recovery mode feature in the future, once the network and contracts have matured sufficiently.

In general, the contract may enter recovery mode due to:

- A transaction from a whitelisted account signalling that the contract should enter recovery mode.
- Something that should always be true of the colony not being true — for example, after an expenditure payout checking that the amount of funds promised to expenditures and not yet paid out is still less than the balance of the colony. If not, then abort the transaction and put the contract into recovery mode.
- A qualitative trigger suggesting something may be amiss — perhaps too many tokens have been paid out in a short amount of time.

Any approvals from whitelisted accounts to leave recovery mode must be reset whenever a variable is edited. A whitelisted account agreeing to leave recovery mode records the timestamp at which the agreement occurred, and any change of variables also update a timestamp indicating the last edit. When attempting to leave recovery mode, only agreements made after the last edit are counted towards meeting the threshold.

The first **recovery** permission holder is set at colony creation and is the creator of the colony. Additional **recovery** permission holders can be added by the **root** permission.

2.8 Arbitrary transactions

Of course, it is possible that a colony will want to engage in some behaviour that we haven't foreseen, that could be implemented in a contract outside the control of the Colony Network (such as changing a parameter in a contract when the colony as a whole is responsible for governing that contract). To that end, we wish to have a mechanism by which a colony can create an arbitrary transaction on the blockchain to interact with contracts and tokens without requiring the network to explicitly support them. As they are powerful, such transactions should be rare occurrences requiring **root** authorization.

3 Extending Functionality

The vision of Colony is the creation of decentralized, trustless organizations, in which decisions are driven by reputation, not a subset of moderators. Yet, at the level of the core Colony contracts, access is mediated by permissions, not reputation.

The decision to make 'permissions' Colony's core access-control logic is doubly motivated. First, it makes it possible to launch a Colony which is admin-controlled (appropriate for small teams with substantial existing trust) and to transition to a more decentralized, trustless style of operating as the organization matures. Second, the permissions-based approach makes it possible to experiment with a wide variety of mechanisms without needing to continually deploy new Colony contracts. Much like the distinction between **kernel space** and **user space** in the design of operating systems, permissions can be thought of as providing the **system calls** needed to give end-user applications (extensions) the ability to securely manipulate the underlying resources of the system. Just as this model has proven very successful in enabling a wide variety of software applications to safely share computing resources, so do we think that the colony-and-extensions model will be successful here.

This section will discuss in-depth how a variety of key trustless functionality can be implemented and made available for colonies to use, and to demonstrate the flexibility enabled by the permissions system.

3.1 Tasks

Unlike an expenditure, which represents an abstract transfer of resources, the 'task' represents a more concrete exchange of labor for value, and a unit of work requiring no further subdivision or delegation. A task has three roles associated with it:

- `Manager` — responsible for defining and coordinating the delivery of the task.
- `Worker` — responsible for executing the task.
- `Evaluator` — responsible for assessing whether the work has been completed satisfactorily.

The manager (initially the creator of the task) is responsible for selecting the evaluator and worker and setting additional metadata for the task:

- A `dueDate`.
- `payouts` for each of the manager, worker and evaluator.
- A `specificationHash`: the address of a specification on IPFS, used by the worker to guide the work, and the evaluator for assessing the satisfactory completion of the task.

In order to create a task, the manager must have the **administration** permission. Future variations of the 'tasks' extension may instead impose minimum reputation requirements and/or staking, making task creation trustless.

Defining what the payouts for each role should be, of course, does not provide the funds — this must be done through the funding mechanisms in Colony (see Section 2.2). Payouts do not have to all be in the same currency, and a task's payout can be made up of an arbitrary number of currencies. If a payout for the task is denominated in the colony's token, the recipient will also earn reputation when the task is completed as long as their work was well received.

If no worker has been assigned to a task, the manager has the ability to cancel the task entirely. Any funds that have already been assigned to the task via funding proposals may be reassigned to the domain of the task.

Assigning either the worker or the evaluator requires the mutual agreement of the manager and the assignee. Once assigned, changes which involve either the worker or evaluator (such as changing the task brief or due date, cancelling the task, or changing assignments or payouts) require mutual consent (i.e. multisig approval), or can be triggered via the motions process.

After the task has been assigned, the worker has until the due date to make a 'final submission', which includes some evidence that the work has been completed.

Once the due date has passed or the worker has made their submission, the evaluator may rate the work. Regardless of whether the rating is positive or not, the task enters a state in which **motions** to the change final state of the task can be raised and **disputes** can be initiated (see Section 3.4). Once the motions period has elapsed, payouts are eligible to be claimed.

As mentioned, the performance of the user who has completed the work is determined after the work is submitted. At this point, the evaluator grades[6] the work submitted by the worker, and the worker rates the manager's ability to coordinate delivery of the task, on a scale of one to three points. In the case of the evaluator, a rating of one point counts as them rejecting the work and a rating of two or three points counts as accepting the work. The rating received determines the reputation change the user will experience:

1 point:	User was unable to complete the task. Reputation penalty equal to 1x payout.
2 points:	User completed the task acceptably. Reputation gain equal to 1x payout.
3 points:	User completed the task superbly. Reputation gain equal to 1.5x payout.

The worker receives reputation both in the domain (and parent domains) and the skill(s) of the task, while the manager receives only in the domains, not in the skill(s) as they have not actually done the task. While it is likely some knowledge is required to coordinate delivery of the task, this is not always the case; we believe that skill reputation should exclusively demonstrate ability to perform tasks.

Upon completion of a task, the evaluator also earns domain reputation (with an implicit rating of "2"). There is no explicit rating of the evaluator, but as with all other payouts a motion can be made before a payout is claimable; the outcome of the motion may be a reduction of payout or an explicit reputation penalty.

Tasks are built on expenditures, and implementing tasks as an extension contract makes use of the **arbitration** and **administration** permissions – the latter to manipulate the expenditure and the former to implement consequences of the rate-and-reveal flow.

When the task is finalized, the underlying expenditure's `claimDelays` are set to allow for motions to be made. Based on the rating recipients receive, their `payoutModifiers` are set to give a reputation boost (for an excellent review) or reduce the payout they can claim (for an unsatisfactory review). Also in the case of an unsatisfactory review, a reputation penalty is emitted.

[6] These scores should be submitted using a pre-commit and reveal scheme to ensure secrecy during the rating process and avoid retaliatory grading in the event that the manager and evaluator are the same person, which we expect to be a reasonably common occurrence. In the event of a user not committing or revealing within a reasonable time, the rating of their counterpart is assumed to be satisfactory, and they receive a mild reputation penalty.

3.2 Funding queues

Section 2.1 describes the funding permission, which is used to transfer funds in between **funding pots**. This permission is powerful — for daily operations, it is better to mediate the allocation of funds via a more specialized mechanism. One such mechanism is the **funding queue**, which leverages time as a driving factor to enable *collaborative*, *asynchronous*, and *trustless* decision-making, *without* voting.

The essential idea behind the funding queue is that tokens are *continuously* allocated over time, rather than all at once in a pass/fail fashion, with user inputs controlling the *speed* and *direction* of the allocation. As an analogy, one can think of water running down a mountain, where the shape of the mountain determines the direction and speed of the flow. With funding queues, users determine this shape. Whereas with voting-based systems, unpopular proposals fail entirely, with a funding queue, unpopular proposals simply take a very long time to be fulfilled, while popular proposals are fulfilled quickly.

Any member of the colony may create a funding proposal. The proposer must have 0.1% of the reputation of the domain that is the most recent common ancestor of the source and target pots, and stake an equivalent amount of the colony's tokens. This stake is used to help discourage spamming of funding proposals and provide a mechanism whereby the creator can be punished for bad behaviour.

A funding queue contains of a series of **funding proposals**, each with the following attributes:

- `Creator` – The person that created the proposal.
- `From` – Funding pot funds are coming from.
- `To` – Funding pot funds are going to.
- `TokenType` – The token address (0x0 for Ether).
- `CurrentState` – The state of the proposal (i.e. `inactive, active, completed, cancelled`).
- `TotalPaid` – Amount transferred thus far.
- `TotalRequested` – The total amount requested.
- `LastUpdated` – The time when the proposal was last updated.
- `Rate` – Rate of funding, a function of the backing reputation.

We distinguish between two types of funding proposals: **Basic Funding Proposals (BFP)** intended for normal use, and **Priority Funding Proposals (PFP)** intended to be used when a basic proposal is inadequate, or for unusual circumstances. The basic funding proposal may start funding the target straight away, whereas a priority funding proposal must be explicitly voted on before it starts directing funds. Furthermore, for a basic funding proposal the target pot must be a direct descendant of the source in the hierarchy whereas a priority funding proposal has no such restrictions. Priority funding proposals should be used when funds need to be directed somewhere that is not a direct descendant of the source, when the funding rate needs to be very high (including immediate payment), or when multiple funding proposals must occur in parallel (e.g. in the case of paying of salaries).

3.2.1 Attributes in detail

`From`, `To` and `TokenType`

The purpose of a funding proposal is to move tokens of `TokenType` from pot `From` to pot `To`. The `TokenType` may be Ether or any ERC20-compatible token. The `From` field must be a funding pot associated with a domain or an expenditure in the colony, while the `To` field may be any funding pot (including the special rewards pot, see Section 2.4). If the funds are to move 'downstream' from a domain to one of its children, a basic funding proposal is often sufficient.

`CurrentState`

The state of a funding proposal is either `inactive`, `active`, `completed` or `cancelled`. Only an active funding proposal is in line to channel funds. A basic funding proposal begins in active state while a priority one begins inactive (i.e. it must be activated by a vote). A funding proposal is active until it is either completed (when its `TotalPaid` reaches `TotalRequested`) or cancelled.

The `creator` of a funding proposal may cancel it (setting `CurrentState` to `cancelled`) at any time. This is analogous to the creator of a task being able to cancel the task if it has not yet been assigned a worker (see Section 3.1). Note that if an expenditure is cancelled, funding proposals that have that expenditure's funding pot as their target (`To`) are automatically cancelled when they are next pinged, and no funds are reallocated. However, the funds that had already been transferred are not automatically returned; it may require a PFP to return the funds 'upstream'.[7]

`TotalPaid` and `TotalRequested`

The total number of funds that a funding proposal wishes to reallocate is called its `TotalRequested` amount. Due to the mechanism by which funding proposals accrue funds over time, it is common that a funding proposal will have received a part but not all of its `TotalRequested` amount. The total number of tokens accrued to date are stored in its `TotalPaid` amount.

The `creator` of a funding proposal may edit the `TotalRequested` property of a funding proposal at any time, but doing so resets the reputational support that the proposal has in the funding queue to zero. The intention here is for changes to funding to be potentially quick to achieve with the agreement of others in the colony if the requirements for the recipient pot change (e.g. the scope of a domain increases).

`Rate` and `LastUpdated`

When a funding proposal is eligible to accrue funds, it does so at a specific `Rate`, denominated in 'tokens per second'. Since nothing happens on the blockchain without user interaction, the funding system uses a form of lazy evaluation. To claim funds that the proposal is due, a user may 'ping' the proposal — i.e. the user manually requests a pro-rated distribution of funds.[8] When pinged, the time since `LastUpdated` is multiplied by the `Rate` to determine how many tokens the proposal would have accrued in the interim if funding flow were continuous. This amount is added to `TotalPaid`, the funds are transferred, and the current time is recorded as `LastUpdated`.

[7]It is conceivable that such return-funds-from-cancelled-tasks PFPs have lower hurdles of activation.

[8]This does not preclude a front-end interface displaying the 'current' level of funding, updated in real-time.

`TotalPaid` is only ever increased up to `TotalRequested` and when this happens as a result of a pinging transaction, the `LastUpdated` value is set to the earliest time at which this could have occurred.

3.2.2 Basic funding proposals

A basic funding proposal (**BFP**) is a funding proposal from some domain's funding pot to one of its children's. It starts out in the `active` state and is thus immediately eligible for funding. It may be cancelled at any time by the `Creator`.

Ordering of BFPs

When created, a basic funding proposal gets placed at the back of the queue. Users can give a proposal 'backing' weighted by their reputation in the source domain[9] at the time of backing[10]. The more reputation backs a proposal, the higher up the queue it is placed. Every transaction that adds backing to a proposal (or otherwise updates the backing level) inserts the proposal in the correct place in the queue. Only the proposal at the front of the queue accrues funds.

There are no costs to backing a proposal (other than gas costs) and the users obtain no direct benefits; it does not represent them putting their earned reputation at risk, nor any tokens — it merely helps the proposal achieve funding in a more timely fashion, and benefits them indirectly by helping the colony run better.

The rate of funding for BFPs

The more reputation backs a proposal, the faster it is funded. The rate scales linearly, and at the limit, if 100% of the reputation in the source domain backs a basic funding proposal, then that funding proposal will be funded at a rate of 50% of the domain's holdings (of `TokenType`) *per week.* The goal is a steady and predictable allocation of resources directed collectively by the domain's (reputation weighted) priorities.

When a user backs a proposal, both the user and their reputation at the time are recorded. Consequently, the user is able to update their backing at a later date. However, we note that such an update is not automatic and even if a user loses reputation due to bad behaviour, their backing level remains unchanged until explicitly updated. Anyone is allowed to make this update – imagine a scenario where a user lost a lot of reputation due to bad behaviour, and other users wanted to stop a funding proposal backed by that user from continuing to accrue funding.

We emphasised that a user could back a proposal with their reputation at the time of backing because the reputation backing a proposal will not change when that user's reputation does so. If by a quirk in this system, the reputation recorded as backing a funding proposal ends up higher than 100% of the total of that reputation in the colony, then the funding occurs no quicker than it would at 100%.

Completing a BFP

If an update finds that a proposal is fully funded (i.e. `TotalPaid` = `TotalRequested`), it is removed from this queue to allow the next-most-popular funding proposal to accrue funds. Explicitly, the

[9]The source domain of a BFP is the domain of the funding pot that the funding proposal is `From`.

[10]A user's reputation may change, but the backing weight is recorded at the time of backing and does not change without further user action.

following steps need to happen:

1. The time at which the funding proposal was fully funded is calculated.

2. `TotalPaid` is set to `TotalRequested`.

3. The BFP is removed from the queue.

4. The next BFP in the queue is promoted to the top of the queue, and its `LastUpdated` time is set as the time calculated in **1.**

Three days after the BFP has been fully funded (and thus `complete`) the creator's stake is released. Until that time, the stake can be slashed by the arbitration permission.

3.2.3 Priority funding proposals

A priority funding proposal (**PFP**) is a funding proposal that can request funds to be reallocated from any pot to any other at any rate. PFPs begin in the `inactive` state and can only become `active` via an explicit vote. The vote is based on reputation in the domain that is the most recent common ancestor of the two pots that money is being transferred between. We imagine PFPs will be used to:

- reclaim funds from child domains.
- reclaim funds from cancelled tasks.
- fund tasks across domains.
- set aside funds designated as a person's salary.
- make large, one-off payments.

Unlike Basic Funding Proposals, Priority Funding Proposals are not ordered in a queue — all active PFPs are eligible to receive funding at any time. Note that, since the amount of funds transferred is ultimately a function of the funds available, too many large PFPs can interfere with each other (and the leading BFP) by substantially reducing the amount of funds available in the funding pot.

3.3 Budget box

As an alternative to funding queues, colonies may instead choose to use a Budget Box[11] to allocate funding between subdomains (or to divide funds among multiple recipients of an expenditure). Compared to funding queues, in which items are funded *serially*, Budget Boxes allow items to be funded *in parallel*, proportionally out of some fixed budget. Budget Boxes are described in detail in [**?**].

Depending on the usage, the extension would need the **funding** and potentially **administration** permissions.

[11]See https://colony.io/budgetbox.pdf

3.4 Motions and disputes

The most successful organisations are those which are able to effectively and efficiently make decisions, divide labour, and deploy resources. Often, these many decisions are structured via management hierarchies. But trustless colonies are intended to be low trust, decentralised, and pseudonymous — a hierarchy is not suitable.

In most DAO frameworks, the mechanism of collective-decision making is usually voting, but Colony is designed for day-to-day operation of an organisation. In a colony, voting on every decision is wholly impractical. The emphasis should be on 'getting stuff done' and not about 'applying for permission'. Therefore, Colony is designed to be permissive. Task creation does not require explicit approval (Section 3.1), nor do basic funding proposals (Section 3.2) or any number of administrative actions throughout a colony.

The **Motions System** provides a self-regulating mechanism which, via a balanced set of incentives, lets users keep their colony running harmoniously. It is there to resolve disagreements and to punish bad behaviour and fraud. The motions system allows colony members to signal disapproval of and potentially *force a vote* against users who have acted inappropriately.

When a member of a colony feels that something is amiss, they may make a **motion**. By doing so, they are fundamentally proposing that either a) a variable, or more than one variable, in the colony should be changed to another value, or b) a user, or more than one user, should receive a reputation penalty. For this reason we call supporters of the motion the 'change' side and opponents the 'keep' side.

The user making the motion must also put up a stake of the colony's internal token (see Section 3.4.2). In essence, they are inviting the rest of the colony to disagree with them. In the spirit of avoiding unnecessary voting, the motion will pass automatically *unless* someone else stakes on the 'keep' side and thereby elevates the motion to a **dispute**.

We say that a dispute has been made whenever a motion has found support on both the 'change' side as well as the 'keep' side. Once raised, disputes must be resolved by voting.

3.4.1 Making motions

The user making a motion submits the following data:

- The data that should be changed, or users to receive penalties.
- The reputation(s) that should vote on this issue (a maximum of one from each of the domain and skill hierarchies).
- Proof that these reputations should be allowed to make the change in question.

The first item identifies the subject of the motion, and what the appellant believes the state should be.[12] The second and third points concern appeals. The basic rule in Colony is: *you cannot appeal a decision to higher management, you can only appeal to larger sets of reputation.*

For example, suppose that the motion concerns a task in the 'frontend' domain. The appellant could choose to have all 'development' reputation vote on it — we say the decision was 'appealed

[12]The exact structure of this is dependent on the method used to implement contract upgradability. The function that uses it is likely to require being coded with inline assembly in the contracts, and require significant effort in the client to make it intuitive to generate and verify.

to the development domain'. In this example, the third point would be a proof that the domain 'frontend' was indeed a subdomain of 'development'. The highest domain any decision can be appealed to is the root domain, where all domain reputation is entitled to vote.

Whenever an appeal occurs, we need to ensure that the reputation we are appealing to is a parent of the reputation associated with the variable being changed. This is possible to do efficiently because of metadata that is placed on the reputations (for domains) when they are created, which includes pointers to at least the direct parent (see Section 2.5.3). When a user makes a motion, instead of directly specifying the domain they are appealing to, they provide the steps needed to get there from the domain associated with the variable that is to be changed. This ensures that the domain they appeal to is a direct parent of that associated with the variable.

3.4.2 Costs and rewards

Cost of making a motion

To make a motion, a user must possess enough reputation and must also stake some number of the colony's tokens. The reputation they need to be able to make the motion depends on the domain they are appealing to; the 'higher up' the decision goes, the higher the reputation requirement (and potential loss). To be able to create a motion, the user must have 0.1% of the reputation in the domain and must stake 0.1% of the corresponding fraction of tokens. Thus, if a motion involves 13% of total colony reputation, then the motion requires 0.013% (0.1% of 13%) of reputation and the required stake is 0.013% of all colony tokens.

If the initial user does not have the required number of tokens or reputation, they can still create such a proposal by staking as little as 10% of the tokens required, which requires them to have a correspondingly smaller amount of reputation.[13] In this case the motion will not be 'live' until other users stake tokens, and take it over the 0.1% threshold. The amount of tokens required to be staked for a particular motion is recorded at the time when it is created. Users can only stake tokens in proportion to the reputation they have. For example, if they wanted to stake 40% of the tokens required, they must have at least 40% of the reputation that would be required to create the motion outright.

Cost of opposing a motion

Once enough tokens have been staked on a motion it becomes active and, barring any further actions for three days, the suggested change will take place (when the motion is 'pinged' by a user). However, if there are users who oppose the suggested 'change', they may stake tokens in support of the 'keep' side. If the keep side receives sufficient support, a dispute is created.

If the 'change' side does not garner enough support in three days, the motion fails and is rejected. If, three days after the 'change' side had enough tokens staked and the 'keep' side does not, then it is assumed that the change is acceptable.

Voting on disputes

If both sides stake the required number of tokens within the time limit, then the dispute goes to a vote. The weight of a user's vote is the sum of their reputations in the skills chosen by the user

[13]This basic minimum required to propose a change prevents users from spamming motions — even those that won't ever be voted on — to large numbers of people, which would impede the smooth running of the colony.

who originally made the motion.

The duration of the poll is determined by the amount of reputation eligible to vote as a fraction of reputation in the colony. If a larger fraction is eligible, the longer the poll is open for. The minimum duration is two days and the maximum is seven. This is a trade-off between allowing disagreements between small groups to be resolved quickly, but to also allow adequate debate to occur when more people are involved.

Voting takes place using a commit-and-reveal-scheme. This scheme is desirable because votes are kept secret during the voting period, preventing users from being influenced by what they perceive to be the majority opinion.[14] To make a vote, the user submits a hash that is `keccak256(secret, optionId)`, where `optionId` indicates the option that the user is voting for. Once voting has closed, the poll enters the reveal phase, where a user can submit `(secret, optionId)` and the contract calculates `keccak256(secret, optionId)` to verify it is what they originally submitted.

As the secret is revealed it cannot be sensitive. It must also change with each vote so that observers cannot establish what people are voting for after they have revealed their first vote. While there are many reasonable schemes for generating secure secrets, we suggest a (hash) of the consequence field of the poll signed with their private key, as it is easily reproducible by a client at a later date with no local storage required.

To combat voter apathy, 10% of the staked tokens are set aside to pay voters when they vote: if a voter has 1% of the reputation allowed to vote on a decision, they receive 1% of this pot that is set aside. They receive this payout when they reveal their vote, regardless of the direction they voted in or the eventual result of the decision. This 'payout regardless of opinion' is to avoid us falling victim to the Keynesian beauty contest [7], in which due to receiving a reward for being "correct", voters are incentivised to vote for what they believe most other people will vote for, rather than what they independently believe. Any tokens that would have been awarded to users who abstained from voting, or are not revealed in the reveal window, are sent to the root domain funding pot once the poll closes.

Once a vote has been in the reveal phase for 48 hours, a transaction may be made to finalise the vote. Any subsequent reveals of votes do not contribute to the decision being made, but serve only to unlock the user's tokens if it was a token-weighted or hybrid vote (see below).

Consequences of the vote

If the 'change' side wins the vote then the change in question is made, but only if the reputation that voted for this outcome is more than previous votes on the same variable. If the 'keep' side wins, then the variable is not changed. In either case, the fraction of total reputation in the colony that voted for the winning side is noted.

At the conclusion of the poll, losing stakers receive 0-90% of their staked tokens back and they lose the complementary percentage of the reputation that was required to stake. The exact amount of tokens they receive back (and therefore reputation they lose) is based on:

- The fraction of the reputation in the colony that voted.
- How close the vote ultimately was.

[14]Even better than commit-and-reveal voting would be receipt-free voting in which votes are *never* revealed, as this would disincentivise vote-buying. Unfortunately the technology to make this possible is still under development.

At the end of a vote, if the vote was very close, then the losing side receives nearly 90% of their stake back. If the vote is lopsided enough that the winning side's vote weight (w) reaches a landslide threshold (L) of the total vote weight, then they receive 0% of their staked tokens back. L varies based on the fraction of total reputation in the colony that voted (R):

$$L = 1 - \frac{R}{3}. \tag{2}$$

So for a small vote with little reputation in the colony being allowed to vote, the decision has to be close to unanimous for the losing side to be punished harshly. For a vote of the whole colony, the landslide threshold L reduces to 67% of the votes — i.e. the reputation of the colony overall was split 2-to-1 on the decision.

Between these extremes of a landslide loss and a very slim loss, the loss of tokens and reputation suffered by the losing side beyond the 0.1 minimum (Δ) varies linearly:

$$\Delta = 0.9 \times \min\left\{\frac{w - 0.5}{L - 0.5}, 1\right\} \tag{3}$$

and so the total loss ($0.1 + \Delta$) varies between 0.1 and 1.

What happens to the tokens lost?

Any tokens lost beyond the initial 10% are split between the colony and those who staked on the winning side, proportional to the amount they staked. Half of the reputation lost beyond the initial 10% is given to those who staked on the winning side, and half is destroyed (the colony as a whole having reputation has no meaning, unlike the idea of the colony as a whole owning tokens).

The motivation here is efficiency — it aims to discourage spurious motions and disputes. A close vote is a sign that the decision was not a simple one and forcing a vote may have been wise. Therefore, the opposition should not be harshly punished. On the other hand, if a vote ends in a landslide, it is a sign that the losing side was going up against a general consensus. We encourage communication within the colony. Members should be aware of the opinions of their peers whenever possible before motions are made.

Repeated motions

In order to reduce the number of repeated motions and disputes over the same variable, the fraction of total reputation in the colony that voted for the winning side is recorded after every vote. This is the threshold that must be exceeded in any future vote in order to change the variable again. We reiterate that this value is updated after every vote on the variable, even if the decision was to maintain the current value of the variable.

This requirement is the primary driver of the appeals process. If a decision was made in a domain, with low voter turnout, then it may be possible to reverse the decision by holding another vote in the same domain (along with a more vigorous off-chain get-out-the-vote campaign). However, if a large fraction of the domain's reputation participated in a vote, then the only way to gain the support of a larger body of reputation (necessary to reverse the decision) would be to appeal the motion to a higher domain, or to a larger body of skills.

There is one exception to this rule: to ensure that a variable can always be changed if necessary, this threshold for changing the variable is ignored if the motion was appealed to the root domain of the colony. Anytime a vote is held in the root domain, regardless of prior votes on the variable, the variable can be changed.

3.4.3 Types of vote

Depending on the context and potential consequences of the vote, Colony supports three types of voting. The type of vote a particular action merits is predetermined based on the action, and is not a choice of the appellant.

Reputation-weighted voting

Most votes in a colony will be due to motions related to tasks. In these cases, the weights of the users' votes is proportional to the reputation that each user has in the domain and skill that the vote is taking place in. When such a vote starts, the current reputation state is stored alongside the vote. This allows the current reputation state to be 'frozen' for the context of the vote, and prevents unwanted behaviours that might otherwise be encouraged (for example, delaying submission of a task until closer to voting so that the reputation earned has not decayed as much).

When revealing their vote, the user also supplies a Merkle proof of their relevant reputation contained within the reputation state that was saved at the start of the vote. The total vote for the option they demonstrated they voted for is then incremented appropriately.

Token-weighted voting

While Colony encourages the use of reputation as the primary sibyl-resistance mechanism, there are situations where tokens are more appropriate. Specifically, if reputation is a stand-in for 'labour', and tokens are a stand-in for 'capital', then token-weighted votings are appropriate whenever a decision must be made by capital, rather than by labour. Whenever a decision would be made by 'investors' or 'shareholders' in a conventional firm, a token-weighted vote may be appropriate.

Unlike with reputation, we do not have the ability to 'freeze' the token distribution when a vote starts. While this is effectively possible with something like the MiniMe token [8], we envision token-weighted (or hybrid) votes will still be regular enough within a Colony that we do not wish to burden users with the gas costs of deploying a new contract every time.

When conducting a token-weighted vote, steps must be taken to ensure that tokens cannot be used to vote multiple times. In the case of 'The DAO', once a user had voted their tokens were locked until the vote completed. This introduced peculiar incentives to delay voting until as late as possible to avoid locking tokens unnecessarily. Our locking scheme avoids such skewed incentives by locking tokens only during the reveal period.

Instead, once a vote enters the reveal phase, any user who has voted on that poll will find themselves unable to see tokens sent to them, or be able to send tokens themselves — their token balance has become locked. To unlock their token balance, users only have to reveal the vote they cast for any polls that have entered the reveal phase — something they can do at any time. Once their tokens are unlocked, any tokens they have notionally received since their tokens became locked are added to their balance. This global lock prevents a scenario, for example, where one user would reveal their vote and then send tokens to a colluding user, who would then reveal their vote using the augmented token balance.

It is possible to achieve this locking in constant gas by storing all submitted secrets for votes in a sorted linked list indexed by `closeTime`. If the first key in this linked list is earlier than `now` when a user sends or would receive funds, then they find their tokens locked. Revealing a vote causes the key to be deleted (if the user has no other votes submitted for polls that closed at the same time). This will unlock the tokens so long as the next key in the list is a timestamp in the future. A more detailed description of our implementation can be found on the Colony blog [9].

Insertion into this structure can also be done in constant gas if the client supplies the correct insertion location, which can be checked efficiently on-chain, rather than searching for the correct location to insert new items.

Hybrid voting

A hybrid vote would allow both reputation holders and token holders to vote on a decision. We envision such a vote being used when the action being voted on would potentially have a sizeable impact on both reputation holders and token holders. This would include altering the supply of the colony tokens beyond the parameters already agreed (see Section 3.5.1) or when deciding whether to execute an arbitrary transaction (see Section 2.8).

In order for a proposal to successfully pass through a hybrid vote a majority of both reputation and token holders who vote must agree that the change should be enacted.

3.5 Miscellaneous

3.5.1 Token management

While **root** users can mint tokens at-will, in many cases it will be desirable to mediate this ability via an extension contract. Here we describe such an extension.

Token generation and initial supply

When the extension is deployed, the `TokenSupplyCeiling` and the `TokenIssuanceRate` are set. The former is the total number of colony tokens that will be created and the latter is the rate at which they become available to the root domain to assign to subdomains or expenditures. The number of tokens available to the root domain can be updated at any time by a transaction from any user (i.e. a public function will determine the pro-rated amount of tokens to generate since the last distribution).

Increasing the TokenSupplyCeiling

It is advised that new tokens not be generated without widespread consensus — especially if tokens have a financial value. Consequently, such decisions require a vote with high quorum and majority requirements involving both the token holders and reputation holders.

Changing the TokenIssuanceRate

The `TokenSupplyCeiling` represents the total number of tokens that the token holders have granted to the colony in order to conduct business: to fund domains and expenditures, and to incentivize workers and contributors.

The `TokenIssuanceRate` controls how rapidly the colony receives the new tokens. If the rate is ‘too high’, tokens will accumulate in the funding pot of the root domain (or other funding pots lower in the hierarchy). If the rate is too low, this signals that the colony has a healthy amount of activity and that the issuance rate has become a bottleneck. In such situations it may be desirable to increase the rate of issuance without necessarily increasing the maximum supply.

Increasing and decreasing the `TokenIssuanceRate` by up to 10% can be done by the reputation holders alone and this action can be taken no more than once every 4 weeks. Larger changes to the issuance rate should additionally require the agreement of existing token holders.

3.5.2 Compensation Modalities

Using expenditures, it is possible to implement many types of compensation modalities. In addition to the ‘task’ (see Section 3.1), we can conceive of a number of additional modalities of compensation.

Salaries

Tasks imply colony-worker relationships are mostly transactional. In many cases, it is desirable to represent more long-term relationships via a *salary*.

A salary can be as simple as a `recipient`, `amount`, and `period`, and `lastClaimed`. At any point, the recipient can ping the salaries contract, at which point the contract will create an expenditure made out to the recipient, with funding equivalent to the pro-rated amount since the last salary payout. If the recipient is willing to pay the gas costs, they could conceivably claim (a fraction of) their salary daily, or choose to claim weekly, monthly, or at whatever cadence suits them. It would be the responsibility of the colony to ensure that adequate funding is available in the domain out of which salaries are to be paid.

This extension would need the **funding** and **administration** permissions to manipulate tokens and expenditures on behalf of the recipients.

Recurring or Automatic Tasks

While for certain kinds of work, tasks are unique and must be individually scoped out and subjectively evaluated. However, we can imagine situations where work can be done by many people and/or evaluated automatically by computer (such as rewarding users for participating in a referral program). In these cases, a variation of the task which contains logic for evaluating the work, and allows anyone to submit the work, would be appropriate.

This extension would need the **funding** and **administration** permissions to manipulate tokens and expenditures on behalf of the recipients.

3.5.3 Awarding reputation for work not captured by tasks

All reputation decays over time, as described in Section 2.5. This prevents a permanent ‘reputation aristocracy’ and allows reputation to remain relevant even after major changes in the colony token’s value.

Reputation is awarded when a user receives payment of a colony’s internal token — most commonly as payout from an expenditure, but sometimes from motions resolution and, in the case of the Metacolony, from the reputation mining process. We can use the expenditure mechanism to award users extra reputation when there is consensus to do so.

Consider the scenario in which a founder, or an important early contributor to a colony has almost no reputation left by the time the colony starts earning revenue; perhaps the development of the product took a long time or perhaps the reputation decay rate was sub-optimally high for the particular colony.[15] Or perhaps the founder was doing a lot of intangible work to get the colony off the ground in the first place and so was never compensated properly on-chain. To get around the limitations of the reputation system and to re-integrate the founder (and make them eligible to receive their rewards), the colony can create an expenditure that is solely designed to award the reputation they are due. To qualify for the payout of tokens (and thereby the reputation), the user in question would have to give the same number of tokens back to the colony. Again, a good frontend abstraction could make such reputation awards easy and intuitive.

Another important scenario concerns absences due to maternity/paternity or illness – the reputation system should not implicitly discriminate against these users. While 'pausing' reputation decay is not a viable option, various mechanisms of giving reputation 'for free' can be used to ensure that these users retain their reputation during periods of unavoidable absence.

The important point is that any limitations imposed by the system can be weakened if there is consensus to do so. The system should not stand in the way of consensus, it should just provide conflict resolution mechanisms for those times in which there is dissent.

3.5.4 Motions by non-members

Having reputation is a prerequisite for creating a motion or staking an opposition. Therefore, if an outsider is hired by a colony to perform a task, they will not, on their own, be able to make a motion in defense of their work. However, a good colony frontend may allow them to create the template for a motion, effectively calling for members of the colony to support it and submit the motion to the colony network on-chain on their behalf.

This is analogous to a member staking only 10% of the required amount and waiting for further support from their peers (Section 3.4.2), with the difference being that without any third party support, the motion would never be processed on-chain.

[15]Finding an optimal decay rate for reputation in the network will depend on empirical data collected from early colonies. It is also possible that in the future reputation decay rates can be configured per-colony.

4 The Colony Network

The Colony Network is a collection of contracts on the Ethereum blockchain. At the core of the network is the `ColonyNetwork` contract. This contract is primarily responsible for managing the reputation mining process (see Section 5), but also for general management of the network: deploying new colonies, setting the fees associated with using the network, and releasing new versions of the `Colony` contracts. These actions will be mediated by a special colony, the Metacolony.

4.1 Revenue model

The Colony Network must be able to sustain itself. In particular, the Metacolony (which controls the Colony Network) maintains the contracts that underpin the network and develops new functionality for the network — development of which needs to be paid for. Long term, the development and maintenance of the network (including the reputation system) needs to be financed by the network itself.

4.1.1 The network fee

We propose a fee levied on expenditure and reward payouts. When a user claims a payout, some small fraction will be paid to the network. The fees are sent to either the Metacolony (if the payment was in Ether or another whitelisted 'currency' token) or the Colony Network contract (if it is any other ERC20-compatible token). A cartoon showing the revenue split is show in Figure 5.

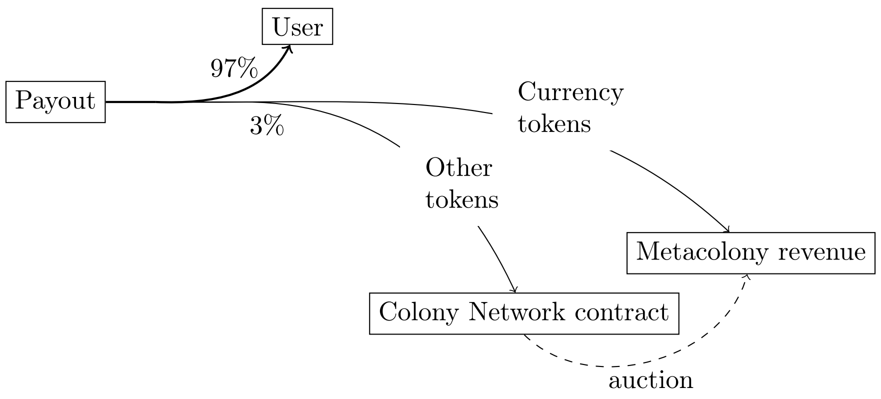

Figure 5: Summary of the revenue split upon payout for a task.

This idea of a fee is a little unusual for such a decentralised system. One of the appeals of decentralised systems on Ethereum is that other than gas costs, they do not seek rent and are free to use. However, the network fee is key to ensuring the game theoretic security of the Colony Network's reputation mining and governance processes, by providing underlying value to the CLNY held by Metacolony members. Importantly, this fee is not payable to any centrally controlled entity, but rather to the Metacolony. As anybody may contribute to the Metacolony, anyone may claim a share of these fees proportional to their contribution. We believe that the benefit of being part of a secure, well maintained network will be appealing enough that a small fee to pay for its existence will be acceptable.

The presence of this fee means we have to make some considerations which would otherwise be irrelevant. Primarily, we will need to make 'piggyback' contracts as hard as possible to make that might e.g. be used to pay out an expenditure payout when a expenditure was finalized, but without sending the fee.

4.1.2 The token auction

As the network fee may be denominated in any ERC20 token, there is a need for a mechanism to liquidate arbitrary bundles of tokens: the token auction. The tokens collected are auctioned off by the Colony Network Contract, with the auctions denominated in Colony Network Tokens, the proceeds of which are burnt. These auctions — one for each type of token collected — occur on a regular basis of once a month.

We believe such a mechanism will be beneficial for the Colony Network Token holders (whose tokens gain value by having an explicit use beyond reputation mining) and the Metacolony itself (by reducing the supply of Colony Network Tokens and thus making any future minting more valuable). It also provides an immediate mechanism of price discovery for a colony's internal tokens, which are unlikely to be traded on third-party exchanges until much later in the lifetime of the colony. By auctioning off the collected tokens, we also prevent the Metacolony collecting a large number of different tokens that it has to manage, which would prove cumbersome and annoying.

4.2 The Metacolony and CLNY

The Metacolony is a special colony which governs the Colony Network. Tokens in the Metacolony are known as CLNY and will be initially generated during the Colony Network distribution period.[16]

4.2.1 Role of CLNY holders and the Metacolony

CLNY holders have two primary roles. The first is to participate in the reputation mining process, described in Section 5. The second is management of the Colony Network itself. There will be permissioned functions on the Network Contract to allow fundamental parameters of the network to be set, which can only be called by the Metacolony. For these permissioned functions to be called by the Metacolony, a vote open to all CLNY and reputation holders must be conducted.

Management of the Colony Network also includes making updates to Colony contracts available to colonies. CLNY holders are not necessarily responsible for the development of these updates, but are required to vote to deploy them. They are therefore responsible for at least ensuring due diligence is done, either by themselves or by service providers, to avoid introducing security weaknesses or other undesirable behaviour. In return for the responsibility of the development and maintenance of the Colony Network, the Metacolony is the beneficiary of the network fee (see Section 4.1).

Reputation in the Metacolony can be acquired by earning CLNY tokens by via expenditure payouts just as in any other colony (see Section 2.5.2). Reputation in the Metacolony can also be earned by participating in the reputation mining process (see Section 5.8), which is unique to the Metacolony.

[16]The mechanism of this distribution are yet to be defined.

4.2.2 Handing off decision-making power to the Metacolony

Colony Network Token holders are responsible for reputation mining from the start, but decisions about the underlying properties of the network will initially be made by a multisignature contract controlled by the Colony team. As the network develops and is proved to be effective, control over these decisions will cede to the Metacolony.

Stage 1: Colony team multisig in control

Initially, the Network Contract's functions will be **root**-permissioned to only allow transactions from the multisig contract under the control of the Colony team to change these properties of the network.

Stage 2: Colony team multisig approval required

At a later date, an extension contract will be set up and given the **root** permission. This contract will allow the Metacolony (as a whole, via the governance mechanisms provided to all colonies) to propose changes to be made to the Colony Network Contract. The intermediate contract will have functionality such that all changes will have to be explicitly allowed by the account under the control of the Colony team. In other words, the Metacolony will be able to propose changes, but the team must sign them off.

Stage 3: Colony team multisig retains veto

The next stage will be a second extension contract operating similarly to the first, but after a timeout — with no interaction from the Colony team's account — the change will be able to be forwarded to the Colony Network Contract by anyone. The Colony team's role will be to block changes if necessary. Thus at this stage the Metacolony will be able to make changes autonomously, but the Colony team retains a veto. The proposal to move to this contract will have to come from the Metacolony itself.

Stage 4: Metacolony fully controls the network

Finally, the specialized extension contract will be removed and replaced with a generic voting extension (see Section 3.4), and the Metacolony will have direct control over the Colony Network Contract with no privileged control available to the Colony team other than that provided by any CLNY and reputation held.

5 Reputation Mining

The reputation system is a core component of any decentralised colony. By carefully balancing the rewards and penalties we aim to keep every users' incentives aligned with the colony and the colony network. Since reputation can only be *earned* and not transferred between accounts, the system fosters a more meritocratic form of decision making than pure token-weighted voting can hope to achieve. The continuous decay of reputation ensures that the influence conveyed by reputation is recently earned and up-to-date. As such, it prevents a reputation aristocracy and allows for a fluid passing of control from one set of contributors to another over time.

Due to the combined complexity of reputation scores across multiple colonies, domains, and skills, reputation scores cannot be stored or calculated on-chain. Instead, the calculations will all take place off-chain, the results of which will be reported to the blockchain by participating CLNY holders — in a process resembling a proof-of-stake blockchain consensus protocol. We call this procedure **Reputation Mining**.

The reputation calculation whose result the miners are submitting is determined by the activities that have taken place in the colonies and can be fully deterministically derived from the Ethereum blockchain. Game-theoretically the system is protected similarly to the off-chain calculations of TrueBit ([4]) in that, *while the calculation cannot be done on-chain and a correct submission can never be proved true, an incorrect calculation can always be proved to be wrong.*

5.1 Merkle-Patricia trees and proofs

This subsection contains only a summary of Merkle-Patricia trees ([5], [6]) and Merkle proofs in order to establish some terminology, and can be skipped if already familiar with them.

A Merkle-Patricia tree, or 'trie', is a key-value store with two special properties: efficient insertion and lookup, and a compact cryptographic state signature. Put succinctly, it is Patricia in the branches, and Merkle in the nodes – the branching of the tree is determined by the keys (in a way which avoids redundant tree traversal), while the values in the nodes are determined by recursively hashing the values inserted in the leaves.

Consider the tree shown in Figure 6, in which every values has a 4-bit key. The data leaves of the tree (1, 2, 3 and 4), which correspond to the keys (0000, 0010, 00111, and 1011), are each hashed individually to give A, B, C and D. These are then repeatedly hashed pairwise, following the branching structure determined by the keys, until only a single hash remains, indicated by G. The resulting structure is known as a Merkle-Patricia tree. In order to prove that the element `1` is in the tree with root `G`, one submits a Merkle proof containing the information `(0000, 1, [E,D], [1010])`. The first pair of arguments are the key-value pair whose existence is to be proved. The third argument is the array of node hashes ('siblings') that the leaf hash should be recursively hashed with. The last argument, the 'branch mask', is an array of `1`'s and `0`'s that indicate which bits of they key correspond to the branching points (in this case, the first and third most-significant bits). So to show that `3` was in the tree with root `G`, the proof would be of the form `(0011, 3, [B,A,D], [1011])`.

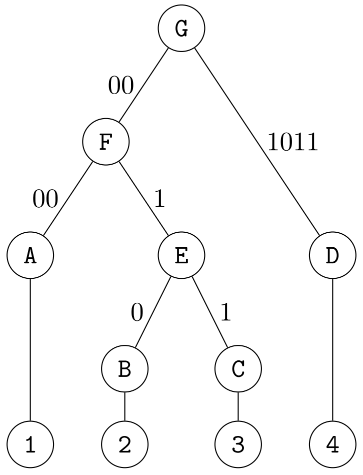

Figure 6: A simple Merkle-Patricia tree with 4-bit keys. Element A is the hash of value 1, with a key of 0000. Element E is the hash of B concatenated with C, and so on recursively up to the root G. Changing any leaf value will change the root, the essential property.

5.2 The Reputation Tree

The key-value pairs in the reputation tree are the reputations all users have in all skills, as well as the colony-wide totals. A single key-value pair consists of the following data:

$$k = \begin{cases} \texttt{colony} & \text{address of the colony the reputation is in} \\ \texttt{user} & \text{account address holding the reputation} \\ \texttt{skill} & \text{skill id of the reputation} \end{cases}$$

$$v = \begin{cases} \texttt{amount} & \text{numerical value of the reputation} \\ \texttt{nonce} & \text{unique per-leaf id (used in reputation mining)} \end{cases}$$

All individual reputations are assembled into the **Reputation Tree** which is a Merkle-Patricia tree of all individual reputations in a colony, as well as the total reputation of each type held by the users in each colony. The leaves that represent these colony-wide totals are indicated by setting `user` to zero. These leaves are then inserted into the tree as described in Section 5.1. We term the root hash of the resulting tree the `ReputationRootHash`, $\mathcal{RH}$.

The `ReputationRootHash` is the only data we record on the blockchain associated with users' reputations. It summarises the state of the whole reputation system and whenever a user wishes to make use of their reputation, they can submit a Merkle proof from the reputation $\mathcal{R}_i$ they wish to make use of and ending at $\mathcal{RH}$.

5.3 Calculating the new root hash

To calculate the new root hash, the miners begin with the last reputation state, and decay all reputations held by all users in all colonies, in the order of the leaf nonces. They then take the set

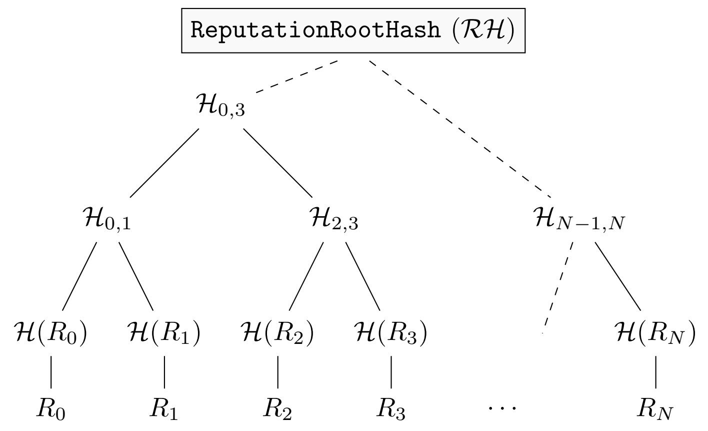

Figure 7: The Merkle tree of users' reputations with `ReputationRootHash` as the root. We use $\mathcal{H}$ to indicate the `keccak256` hash function.

of reputation gains or losses that were not in the last state submitted, and are to be included in the next state (the update log). They apply the reputation updates to each user in each colony, updating or adding leaves as necessary (following the process described in 5.5.1), to end up with a new list of reputations for all users and colonies. These new reputations are then hashed and assembled into a new Merkle-Patricia tree yielding an updated `ReputationRootHash`.

While the calculation is too large to be done on-chain due to technical and economic limitations (i.e. the block gas limit and the cost of gas, respectively), this calculation can easily be performed by a typical user's computer.

5.4 Submission of a new root hash

What is submitted?

The final `ReputationRootHash` is submitted to the contract by the miner along with the number of leaves in the tree (`NReputationLeaves`). The miner also submits the root hash of the Justification Tree (see Section 5.5.1), which will be used in the event of a dispute. These three properties uniquely identify a submission.

Who can submit a new root hash?

All Colony Network Token holders are eligible to become miners and participate in the reputation update process. Since any user can calculate the correct root hash locally, it should be possible for *any* miner to submit the hash to the contract.

It is however undesirable to have too many submissions for every update. We propose a mechanism that only allows some miners to submit results to begin with. To participate in the mining process, Colony Network Token holders must stake some of their tokens to become 'reputation miners'. A submission will only be accepted from a miner if

$$\texttt{uint256(keccak256(address, N, hash))} < \texttt{target}.[17]$$

[17] Note that internally the arguments are encoded using `abi.encodePacked` before being hashed.

At the beginning of the submission window, the target is set to 0 and slowly increases to $2^{256} - 1$ after 1 hour. We limit the total number of miners allowed to submit a specific hash to 12. In the unlikely event that no submissions are received before the 1-hour window has elapsed, exactly one submission will be accepted, whenever it is received.

The variable N that goes into the hash is some integer greater than 0 and less than the number of tokens the Colony Network Token holder account has staked divided by $2000 \cdot 10^{18}$ meaning that users with a large stake have a higher chance of qualifying to submit a hash sooner than smaller stake holders. The factor of $2000 \cdot 10^{18}$ is introduced to ensure that all hashes a user is eligible to submit can be calculated in a few seconds by the client. It also effectively creates a minimum number of tokens that must be staked to submit a hash. This puts a tangible cost on any attacks revolving around spamming known false submissions (see Section 5.7).

When a miner stakes, the timestamp of the stake is recorded.[18] In order to be eligible to submit a hash, the miner must have staked before the beginning of the current mining cycle.

Verifying a submission

If only one state is submitted by the end of the submission period, then the new state is accepted, and proposals of the next state can begin to be made. This is expected to be the most common occurrence.

If more than one state has been submitted, then either someone has made a mistake, or there is a malicious entity trying to introduce a fraudulent reputation change. In this event, the a challenge-response protocol can establish which state is incorrect (see Section 5.5).

Mining rewards

When a state is accepted, a number of (newly minted) Colony Network Tokens are made available for the users who submitted the correct state to claim as a reward. They also receive a corresponding amount of reputation in the Metacolony (in a special mining skill, which only users in the Metacolony can earn by performing this task). This reputation update is no different from any other, aside from the limitations of who is able to earn it, and will be included in the subsequent reputation update cycle. The size of the rewards and their distribution are described in Section 5.8.

5.5 Dealing with false submissions

We assume that the correct hash is one of the submitted hashes. This is a reasonable assumption, as only one out of all the miners is required to make a correct submission, and there is an incentive for them to do so (the reward defined in Section 5.8). Thus our task is not to validate the correct hash but to invalidate the false one(s).

We must prove all but one submission incorrect by having each submission prove that they calculated more correct reputation updates before getting one wrong (if any) than another submission they are being compared against. Anyone is able to respond to a challenge (and be rewarded), regardless of who submitted the original hash; this should ensure that the correct state is always defended, even if some miners go offline.

We consider the scenario where only two submissions are made, and one is correct. In the event of more than two submissions, this same pair-wise comparison described below is repeatedly run (in parallel, where possible) among remaining submissions until only one remains.

[18]If a miner adds additional stake, the timestamp is set to the weighted average of the existing and new timestamps.

5.5.1 The Justification Tree

A client submits a `JustificationRootHash` ($\mathbb{JRH}$) as part of the process of submitting a proposed new root hash. This is the Merkle root of the 'Justification Tree' – a Merkle-Patricia tree where each leaf represents not a single reputation value, but the root hash of an entire reputation state. The left-most leaf of the Justification Tree is the final accepted reputation state from the last update ($\mathcal{RH}_0$) concatenated with the number of leaves ($\mathcal{L}_0$) in the reputation tree $\mathcal{RH}_0$ is the root of. The right-most leaf of the Justification Tree is the `ReputationRootHash` they submitted ($\mathcal{RH}_N$) concatenated with the number of leaves it contains ($\mathcal{L}_N$). We denote these leaves as $\mathcal{RH}_0 \cdot \mathcal{L}_0$ and $\mathcal{RH}_N \cdot \mathcal{L}_N$.

The intermediate leaves represent the evolution of the global reputation state after applying some subset of the full sequence of reputation updates required. Each state represented by a leaf differs from the reputation states in neighboring leaves in *at most* a single reputation.[19] In order to do this in a consistent way, we must order all the updates in a reputation cycle. The canonical ordering for reputation updates is:

1. All decays of existing reputations, in order of `nonce`,
2. The entries in the reputation update log, in order of appearance in the reputation update log. A single one of these entries corresponds to at least two reputation updates with an unbounded upper limit (since there can be many parent and child updates). These updates are themselves ordered:
    (a) Colony-wide total of any impacted child reputations
    (b) Colony-wide total of any impacted parent reputations
    (c) The colony-wide total of the origin reputation
    (d) User-specific child reputations
    (e) User-specific parent reputations
    (f) User-specific origin reputation

    The origin reputation is defined to be the skill specified in the reputation log where the reputation is to be lost or gained. Where multiple child reputations are required to be updated as part of 2a or 2d, they are done so in the order they appear in the `children` property of the origin skill. Where multiple parent reputations are required to be updated as part of 2b or 2e, the immediate parent is updated first, then the immediate parent of that skill, and so on until no parents remain. We do the decay calculations first to give users the benefit of the doubt during reputation updates so they do not lose reputation they have only just earned to premature decay.

As a miner applies the reputation updates for a cycle in this order, they should take each intermediate `ReputationRootHash` values and build the Justification Tree by adding the intermediate `ReputationRootHash` values concatenated with `NReputationLeaves` to the tree, *with key equal to the update number*, starting from 0. Note that unlike in the Reputation Tree, the keys should not be hashed. This will become important further on in the dispute process, as we will need to be able

[19]It is possible for two adjacent leaves to be identical if the reputation update the transition between them represents does not result in a change in reputation – e.g. a reputation loss in a reputation that is already zero.

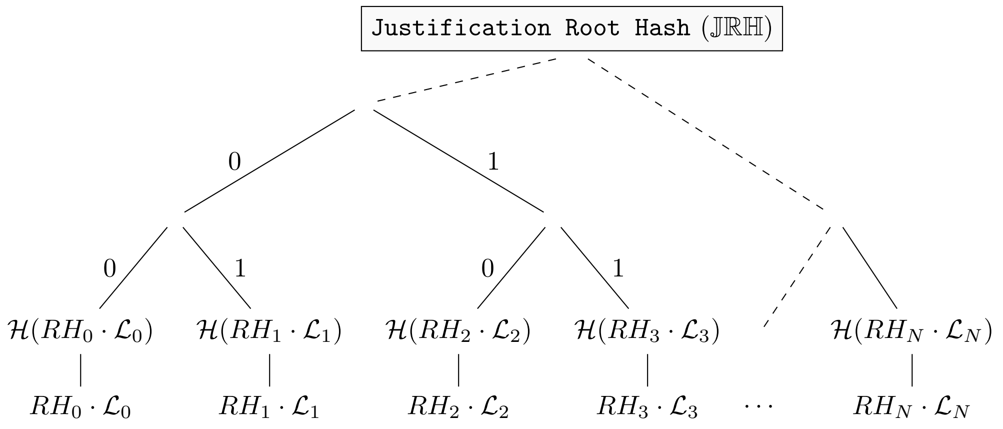
Justification Root Hash (JRH)
0
1
0
1
0
1
$\mathcal{H}(RH_0 \cdot \mathcal{L}_0)$
$\mathcal{H}(RH_1 \cdot \mathcal{L}_1)$
$\mathcal{H}(RH_2 \cdot \mathcal{L}_2)$
$\mathcal{H}(RH_3 \cdot \mathcal{L}_3)$
$\mathcal{H}(RH_N \cdot \mathcal{L}_N)$
$RH_0 \cdot \mathcal{L}_0$
$RH_1 \cdot \mathcal{L}_1$
$RH_2 \cdot \mathcal{L}_2$
$RH_3 \cdot \mathcal{L}_3$
$\cdots$
$RH_N \cdot \mathcal{L}_N$

Figure 8: The Justification Tree. The leaf containing $\mathcal{H}(RH_0 \cdot \mathcal{L}_0)$ – the last accepted reputation state – is found at key `...00`. The leaf containing $\mathcal{H}(RH_1 \cdot \mathcal{L}_1)$ – the reputation state with the decay of the existing skill with nonce 0 applied – is found at key `...01`. Every intermediate state is recorded in the tree up to $\mathcal{H}(RH_N \cdot \mathcal{L}_N)$, which is the proposed new reputation state with all reputation updates applied.

to identify sequentially adjacent reputation updates. The intermediate leaves of the Justification Tree represent the evolution of the reputation state, with $\mathcal{RH}_i$ corresponding to the reputation state after the first i reputation updates in this cycle have been applied. An example of such a tree is shown in Figure 8.

5.5.2 Resolving a dispute

If a dispute occurs, the first step is for each submission to verify the Justification Tree they submitted alongside their proposed new root hash.

1. Verifying the Justification Tree

For a justification tree to be deemed valid, it must:

- Have $\mathcal{H}(RH_0 \cdot \mathcal{L}_0)$ at key 0,
- Have $\mathcal{H}(RH_N \cdot \mathcal{L}_N)$ at key N,
- Have a plausible structure.

The first two items are relatively straightforward to prove via Merkle proof. The last requirement requires slightly more explanation. A Justification Tree with two leaves could meet the first two requirements, but we would be able to tell this tree wasn't plausible as a Justification Tree because the Merkle proofs supplied to prove the first two requirements would not be of the expected length.[20]

[20] We know that every reputation cycle will at least have log entries for rewarding those who submitted the previous reputation state successfully.

This length is calculable because we know how many leaves are meant to be in the tree and what keys they are expected to be at (every key between 0 and N). We are also able to determine the branchmasks that these proofs should have (the binary representations of $2^{\lceil \log_2(N+1) \rceil} - 1$, and N respectively), which further constrains the shape of a plausible tree.

Unfortunately, based on these two proofs, we are unable to guarantee the Justification Tree contains a leaf at every key from 0 to N. This is because we do not know how many leaves the siblings used in the Merkle proofs contain, and it is not feasible to request a proof for every key between 1 and $N-1$. This is why the constraint is limited to being 'plausible'. With the approach we have taken, however, we are at least able to guarantee that key N is the last in the Justification Tree, that key 0 is the first (though this is a trivial consequence of it existing), and there are at least some number of other keys in the tree.[21]

Since any two differing submitted states agree on the first leaf $\mathcal{RH}_0$ (the `ReputationRootHash` accepted at the end of the previous iteration of the mining cycle), and disagree on the last leaf $\mathcal{RH}_N$ (the hash they submitted), there must be a hash $\mathcal{RH}_i$ that they agree on, and a hash that $\mathcal{RH}_{i+1}$ that they do not. This is a reputation update where they agree on the starting state but disagree on the result. This transition is meant to be the effect of a single reputation update (the i^{th}), and this is the reputation update we will calculate on-chain to establish which submission is incorrect.

First, however, we must establish where the two submissions begin to differ.

2. Searching for the discrepancy

The contract requires both parties to submit repeated Merkle proofs to locate $\mathcal{RH}_{i+1}$, the first disagreement hash. We shall call the two parties A and B and we shall indicate which party made a submission by a superscript of A or B. Furthermore we introduce the simplifying notation of $\overline{h}$ to mean 'sibling of h' in the Merkle tree.

Along with their justification root hashes $\mathbb{JRH}^A$ and $\mathbb{JRH}^B$ both parties have already submitted proofs for the left-most leaf. Ignoring the branchmasks for simplicity, these proofs have the form:

$$\overline{\mathcal{RH}_0}^A, \overline{h_{0,1}}^A, \overline{h_{0,2}}^A, \ldots \overline{h_{0,2^k}}^A \qquad \text{terminating at } \mathbb{JRH}^A$$

and

$$\overline{\mathcal{RH}_0}^B, \overline{h_{0,1}}^B, \overline{h_{0,2}}^B, \ldots \overline{h_{0,2^k}}^B \qquad \text{terminating at } \mathbb{JRH}^B$$

where k is the largest integer such that 2^k is smaller than n.

When the first miner (say A) submits their proof the contract saves the values of $h^A_{0,2^k}$ and $\overline{h_{0,2^k}}^A$. When the second miner submits their proof the contract compares $h^A_{0,2^k}$ to $h^B_{0,2^k}$. If they are not equal, the contract saves both of these values (and forgets $\overline{h_{0,2^k}}^A$). If they are equal, the contract retains the values of $\overline{h_{0,2^k}}^A$ and $\overline{h_{0,2^k}}^B$ (forgetting $h^A_{0,2^k}$).

The rationale behind this behaviour is the following: If $h^A_{0,2^k} = h^B_{0,2^k}$ then the two justification trees are equal between $\mathcal{RH}_0$ and $\mathcal{RH}_{2^{k-1}}$ and the first discrepancy must lie in the right-hand subtree

[21] This number is $\lceil \log_2(N+1) \rceil + \#_1(N) - 2$, where $\#_1(N)$ is the binary logarithm of the Nth integer in Gould's sequence. To derive this expression, consider how many siblings each proof requires; each sibling corresponds to at least one other key. In our case, $\#_1(N)$ provides the number of set bits in the branchmask of the proof for key N i.e. how many siblings that the Merkle proof of key N has. The -2 accounts for the key-value pairs stored at 0 and N.

whose root is $\overline{h_{0,2^k}}^A$ for miner A and $\overline{h_{0,2^k}}^B$ for miner B. If on the other hand $h^A_{0,2^k} \neq h^B_{0,2^k}$, then the first discrepancy must lie in the left-hand subtrees given by $h^A_{0,2^k}$ and $h^B_{0,2^k}$. The situation is summarised by

$$h^A_{0,2^k} \neq h^B_{0,2^k} \implies \text{First discrepancy occurs at some } \mathcal{RH}_i \text{ with } 0 \leqslant i < 2^k$$
$$h^A_{0,2^k} = h^B_{0,2^k} \implies \text{First discrepancy occurs at some } \mathcal{RH}_i \text{ with } 2^k \leqslant i < n$$

The contract begins its search by picking an index j from within the range the first discrepancy in known to lie in (say always the smallest), and requiring both parties to provide a Merkle proof showing value of `ReputationRootHash` and `NReputationLeaves` at that key in the Justification Tree. The required target of this proof is no longer the $\mathbb{JRH}$ itself, but rather the retained value for $h_{0,2^k}$ or $\overline{h_{0,2^k}}$.

The process as before of comparing hashes and retaining the roots of either the left-side subtree or the right-side subtree is repeated. With each iteration, the range of possible values for the index of the first discrepancy is reduced by (on average) a factor of two and the length of the required Merkle proofs is reduced by at least one.

There are two ways this process can terminate. The first way the process terminates is when one party does not respond to a challenge in adequate time, either because they could not or chose not to. In this case the party not responding is deemed to be incorrect. The other way the process terminates is when it has reached the bottom of the tree and determined i, the disputed reputation update.

3. Confirming i

Once the contract has found the index i such that $\mathcal{RH}^A_i = \mathcal{RH}^B_i$ but $\mathcal{RH}^A_{i+1} \neq \mathcal{RH}^B_{i+1}$, the contract then requires each party to submit the value stored at the key $i+1$ in the Justification Tree to confirm that it is there. It is possible that this was already done during the binary search, but it is not guaranteed. The value of the `ReputationRootHash` and `NReputationLeaves` at this key are stored in the contract managing the dispute resolution. Again, if one party does not respond in time, they are assumed to be incorrect.

4. Deciding which submission performed the correct calculation

In most cases, only one of the two submissions will be able to perform the transaction corresponding to this step. There are several types of calculation and checks that might need to take place, depending on which transition the two submissions first disagree on, and the values of `ReputationRootHash` and `NReputationLeaves` they placed in their Justification Trees. All of the following must be performed on chain to verify the calculation under dispute.

- **Check the key** As part of the Merkle proofs submitted (see the next step), the `key` of the reputation under dispute is submitted. This is a concatenation of the colony address the reputation is in, the skill the reputation is in, and the account address holding the reputation (which is possibly the zero address if the total reputation in the colony is under dispute). Based on the value of i, which has already been established during the binary search, and the known canonical ordering of reputation updates, the contract is able to check these values. In the case where the update under dispute is due to a log entry (as opposed to normal decay), the index of the log entry under dispute is supplied by the user and verified on-chain, to avoid iteration over the unbounded log.

- **Check the claimed before and after values** The values of the reputation at `key` in $\mathcal{RH}_i$ and $\mathcal{RH}_{i+1}$ for the submission in question are proved via Merkle proof. If it is claimed an existing reputation is being updated, then the proofs' siblings and branchmask are required to be the same to guarantee no other reputations have been changed. If a new reputation is claimed to be being added (indicated by $\mathcal{L}_{i+1} - \mathcal{L}_i = 1$), the proof for the reputation in $\mathcal{RH}_i$ is not required as the reputation should not be in the tree, but an additional check will be performed later to confirm this.

- **Perform the calculation** This has two elements. The first is to check that the `nonce` for the reputation has not changed in the leaf if an existing leaf is being updated. If a new leaf is being added, the value of the `nonce` is checked to be one larger than the $\mathcal{L}_i$ that they have provided (as each `nonce` is given in sequence). The second is to check that the value of the reputation is correct. It can be a decay, or an update due to an entry in the log, the potential consequences of which are described in section 5.6.

- **Additional checks** Depending on the type of update being disputed, additional checks may be required

    1. Any additional reputation values needed for the calculation are proved to be in $\mathcal{RH}_i$ (which both parties agree on the value of) as required via Merkle proof. The canonical ordering of reputation updates ensures that any values required for the calculation under dispute are present in $\mathcal{RH}_i$ and have not yet been altered by the log entry under consideration.
    2. Similarly, if a new leaf is added, a Merkle proof for the reputation in $\mathcal{RH}_i$ with the previous `nonce` is checked to confirm the `nonce` assigned to the new node is plausible.
    3. If during the calculation, the user claims that a reputation they need for a calculation does not exist in $\mathcal{RH}_i$, and so they will use a value of 0, they must prove it does not exist in the tree.[22]
    4. If a new reputation value is being added to the tree, we also prove that the reputation that it ends up closest to in the tree exists in $\mathcal{RH}_i$. We are therefore able to deduce what the proof for that reputation in $\mathcal{RH}_{i+1}$ should be if only the new leaf is added. By checking this Merkle proof is valid, we ensure no other changes have been made to the reputation tree.

It is possible that both submissions will successfully submit the transaction that validates their submissions as described here. This can happen when a new leaf is being added, and the submissions have disagreed over the `nonce` the reputation should be given. Whichever has successfully awarded the highest `nonce` is deemed the correct submission. If neither party responds to a particular stage in this process in time, both are eliminated from consideration.

If the 1-hour mining cycle window has not elapsed by the time only one submission remains, the next window only opens when the current window has elapsed. If the 1-hour window has elapsed by the time the dispute process has finished, the next submission window opens immediately.

[22]They do this by providing a Merkle proof for an existing reputation in $\mathcal{RH}_i$ whose hashed key has the longest shared prefix with the hashed key of the reputation being added. By examining the branchmask of this proof, and the (hashed) keys of the proved reputation and the reputation being claimed to not exist, the contract is able to confirm the latter does not exist in $\mathcal{RH}_i$. If a key with a shorter shared prefix is used, the branchmask will already contain a branch point where they diverge and this proof will fail.

5.6 Calculating reputation updates

5.6.1 Keeping track of reputation changes

Fundamentally, there are two types of reputation update that occur:

- Decay of existing reputation.
- Addition or removal of reputation as a reward or punishment.

When a user earns reputation in a skill or domain, they also earn reputation in all parent domains, which corresponds to $2 \times (\texttt{n\_parents} + 1)$ reputation updates. Alternately, when a user loses reputation, they also lose reputation in all parents and all children representing a total number of updates of $2 \times (\texttt{n\_parents} + \texttt{n\_children} + 1)$. The factors of two here come from also updating the relevant colony-wide totals.

In Section 2.5.3, we asserted we store `n_parents` and `n_children` for all skills and domains. It is only by having access to the number of parents and children for each reputation and the reputation update log recording how many reputations have been updated already in this update cycle (via `n_updates`) that the resolution protocol is able to perform the binary search of the justification trees submitted by the disagreeing users. At the start of the challenge protocol, the contract can look up the last entry in the update table for the cycle under consideration, and work out how many updates have occurred in this cycle based on the number of updates prior and the number of parent and child reputations. After verifying that both submitted justification trees contain this exact number of leaves it can proceed to the binary search.

If the discrepant transition is a decay transition they must also supply a Merkle proof that the starting value assumed for the user corresponds to the value that user had at the end of the last update cycle. A decay transition is identified by the Merkle path corresponding to an index in the justification tree smaller than the number of leaves in the reputation tree at the end of the last successful update.

5.6.2 Earning reputation for the first time

When a user earns reputation in a new skill, at least one new leaf is added to the tree — if they have not earned reputation before in some of the parents, then they will also cause further new leaves to be added. Additional new leaves will be added if they are the first user in a colony to earn those particular skills, making the total reputation for that skill in the colony non-zero. During a dispute, when the user proves that they have included the update in the tree, it is not possible to check (efficiently) on-chain that they should not have added it to an existing leaf instead. However, because during the resolution process we are always comparing two submissions against each other, one of two things will be true:[23]

- Both submissions added a new leaf to the tree. If there was a discrepancy, then it is in the maths conducted on this leaf, not the addition of the leaf itself. The maths can be checked on-chain to establish which result is correct.
- One submission adds the new reputation to an existing leaf (the correctness of which can be checked on-chain easily). In this case, the user who added the leaf incorrectly is wrong.

[23]Assuming that one of the two submissions is correct.

5.6.3 Transfers of reputation between accounts

The most important quality of reputation that distinguishes it from a token is that it is tied to an account and cannot be transferred. However, in the event of disputes (Section 3.4) it can happen that one party to a dispute loses reputation while the other gains. This process has to be modelled as a 'reputation transfer' to ensure that reputation is never created in this process (i.e. the reputation lost by the loser is at least as much as the reputation gained by the winner).

If an entry in the reputation update log indicates that a dispute has occurred and been resolved, then there will be a number of transfers of reputation between users represented by a single entry. Each such transfer will have to accommodate the updates of all the parents of the reputation being gained by one user, and updates of all the parents and children of the reputation being lost by the other. However, we have to ensure that the user who is losing reputation still has the reputation to lose if another user is gaining it.

To achieve this, all the transactions that correspond to updating the reputations of the user gaining the reputation are done first. In the event such a transaction must be proved to be correct in the resolution protocol, the users can provide a proof of the losing user's reputation, prior to them losing it in this event in update cycle, and this can be compared to the amount of reputation intended to be gained. Whichever is smaller is used as the amount of reputation the user is gaining during the calculations.

Then, when calculating the reputation deduction to be applied to the losing user, the reputation that was used as the voting weight should be done last i.e. all the children and parents should be considered first, as it is the amount of the reputation that was eligible to vote that will determine the fraction lost of each of the child reputations.

For further details about reputation transfers and disputes, see Appendix A.1.

5.6.4 The reputation decay calculation

The reputation decay process was described above as being continuous. In practise, it will decay by a small, discrete amount during each reputation update cycle following an exponential decay. However, such a calculation is not possible to do accurately on-chain during the resolution protocol, so we must use an approximation. The details of the approximation we use, and a proof that this approximation is accurate and will not affect the running of (active) colonies can be found in Appendix A.2.

5.7 Denial of service attacks

In the event of multiple submissions, finding the correct one takes time — the timeout t for the challenge-response must be reasonable to allow the transaction defending a submission to propagate and be mined. A denial-of-service attack is therefore possible, whereby an attacker makes many false submissions. However, if these false submissions were random hashes, unable to be defended, then none would be defended correctly within the first timeout window, and the attack would quickly end. For pairings where neither submission is defended, any user can remove both submissions from consideration and claim the tokens that were staked to allow submission.

The denial of service attack (to delay a proper reputation update) can only be sustained when the false submissions are incorrect only in some leaves, and the majority of the justification tree is correct. In this scenario, the attacker successfully defends each of their submissions for as long as possible to delay the resolution of the reputation mining protocol as much as possible.

Any such attack is capped by the first round of pairings of submissions against each other. Even if the attacker made millions of submissions, only a finite number of those would be able to be successfully defended due to the block size — currently, no more than 4500 submissions would be able to be defended, even if the attacker used up all block space during the timeout.[24] With only 4500 submissions able to make it to the second round, the length of time the DoS attack would be sustained for is given by

$$t \times \lceil \log_2(4500) \rceil \times \lceil \log_2(N_{\text{updates}}) \rceil$$

where N_{updates} is the number of reputation updates that have been made in this update cycle. To arrive at this figure, we know there will need to be $\lceil \log_2(4500) \rceil$ rounds of comparison between submissions to eliminate all but one. Each round will require $\lceil \log_2(N_{\text{updates}}) \rceil$ interrogations of the justification tree to establish where the two submissions being compared differ. Finally, each interrogation can take up to t before it is considered to have timed out and one or both of the submissions is deemed invalid. The product of these three factors tells us how long this reputation update can be delayed by an attacker.

Long term, N_{updates} will be dominated by the decay transactions rather than by any updates that have occurred since the last reputation state was established. Even if the Colony Network were wildly successful, with 100000 colonies, each with 1000 users that had earned some reputation in 1000 different skills in each of the structural and skill hierarchies, and using 5 minutes as the value of t, the delay to the reputation updates would only be around 36 hours. Recent congestion on the Ethereum network has shown that we will need to be able to accommodate situations where block space is at a premium; the reputation mining client will need to recognise when this is occurring, and send transactions with higher gas prices as appropriate in order to meet the timeout deadline.

There would be little effect on the rest of the Colony Network in this time. Users would still be able to exercise their reputation from the previous reputation update, and continue to influence decisions with that reputation. Indeed, this shows what perhaps the main motivation for such an attack would be — if a user knew that they had been 'caught' behaving badly, and was due to lose all their reputation, they might try such an attack to eke out the last bits of influence they possibly could. However, decisions in the Colony network do not resolve quickly, and in a well-developed colony we would not expect any one person to have a large amount of reputation when compared to the rest of the colony. It therefore seems unlikely any one user would be able to unduly influence decisions significantly while conducting such an attack.

Assuming this attack continued, then the reputation mining protocol would effectively only update every 36 hours. Users staking would become more susceptible to variance in terms of the rewards, but otherwise little would change in the day-to-day functioning of any individual colony.

However, the attacker would lose *all* the Colony Network Token that they had staked (which would be around 4500 times the expected minimum stake) in order to perform the attack, and so would have to buy more to attack again making this attack exceedingly expensive.

Note: There is an edge case to consider in which the attacker is sending enough defending transactions to completely fill the blocks. In such a case however we assume that the defence of the legitimate state is always successfully included in block, as a one-off increase in gas costs will always be worthwhile to ensure the legitimate state is defended. Between now and the deployment of the Colony Protocol, we will carefully observe the Ethereum network to gather empirical data about

[24]These figures assume $1.5\pi \times 10^6$ gas in a block, and that each transaction is only 21000 gas for a worst-case-scenario calculation.

the cost and practicability of such an attack and will adjust the timeout parameter t accordingly. Longer timeout periods make this attack exceedingly difficult and expensive, but would also slow down the resolution protocol.

5.8 Costs and rewards of mining

In order to be involved in the reputation mining process, Colony Network Token holders must stake their tokens with the Colony Network Contract. This allows them to submit a reputation hash as part of the reputation mining process described above.

If they submit a new hash, this is recorded and they are noted as the first account to submit that hash. If they submit a hash that has already been submitted, they are appended to a list of users that have submitted that hash. The system allows for a maximum of 12 miners to be added to the list in each round. The same miner is allowed to appear on the list multiple times, but using different values of N in the inequality (5.4) on page 40.

If a hash is found to be incorrect, all those who submitted it lose some of their stake. If a hash is deemed correct, however, the miners who submitted it gain Colony Network Tokens and reputation.

The total amount of reputation earned by miners is not fixed, but varies along with activity in the Metacolony. The system tries to ensure that on average, 25% of Metacolony reputation comes from mining.

Suppose that the reputation earned in the Metacolony every hour due to all activity (mining included) is constant at h, then eventually the colony will reach a steady state in which the decay of reputation is balanced out precisely by the newly earned reputation and

$$R_{tot}\left(1-e^{-k}\right)=h \tag{4}$$

where k is the decay constant used in each update period (see Appendix A.2) and R_{tot} is the total reputation in the Metacolony. If one quarter of all reputation is to come from mining, then the hourly mining reward M in this situation should be given by

$$M=\frac{h}{4}=\frac{R_{tot}\left(1-e^{-k}\right)}{4}. \tag{5}$$

The *actual* mining rewards are calculated based on the above model and we *define* the total reputation to be earned by miners in a given hour to be given by equation (5).

Miners who make a submission in a given reputation update cycle are entitled to a share of this reward. When a miner makes their submission, their weighting for that submission is calculated and recorded, and this is added to the total weights of all submitters for this hash so far. The n^{th} submitter has a weight of

$$w_n=\left(1-\exp\left(\frac{-t_n}{T}\right)\right)\times\left(1-\frac{n-1}{N}\right) \tag{6}$$

where t_n is a number of seconds that the n^{th} miner has staked their tokens for and T and N are normalising constants. T is set to a number of seconds representing 90 days, and N is set to twice the length of the list of submitters — in our case $N=24$.

The first factor in equation (6) encourages users to stake their tokens for long periods of time when they register as miners. When locking tokens for T seconds, this first factor grows to 0.63, when locking for $2T$ it grows to 0.86, and the factor approaches 1 as the locking time approaches infinity. The second factor in equation (6) encourages miners to submit the hash as soon as possible, with this factor becoming smaller the later users submit; the first submission will have twice the weight of the last submission, all other factors being equal.

Once the submission window has expired, and either there was only one submitted hash, or all but one submitted hash has been proved to be wrong, any user can make a transaction to make this submitted hash the canonical reputation state used by the network until the end of the next update cycle. This transaction mints and transfers the CLNY reward to the miners, as well as adds reputation changes for the miners to the start of the reputation change log, where they will be included in the next update cycle.

The reward earned by each miner on the list is given by

$$m_n = M\frac{w_n}{W} \quad \text{where} \quad W = \sum_{n=1}^{12} w_n \tag{7}$$

i.e. the mining reward M is divided among the miners according to their relative weighting.

5.9 Emergency shutdown

Section 2.7 described a transaction from a whitelisted account can put a colony into 'recovery mode' during which the state can be edited, the effects of bugs can be corrected and upgrades can be made. Similarly, the reputation mining process will also have an emergency stop-and-repair mechanism (to begin with). This will allow the whitelisted accountes to revert the reputation root hash to a previous version and halt all updates to the reputation state until the issues have been resolved (which will likely involve a contract upgrade). The colonies will be able to continue operations as usual using the reputations of the last valid state, which will be temporarily frozen and not decay.

6 Conclusion

We have described and defined the Colony Protocol — an organisational operating system built on Ethereum. It provides a general purpose framework for the creation, management, and operation of decentralised organisations of various kinds.

The specification contained herein represents our current best description of the Colony Protocol. It is however, a working document, and it should be expected that the final specification will differ substantively, both as a consequence of the rapidly changing technological landscape, and the refinement of our understanding of the requirements of decentralised organisation through iterative cycles of development and user testing.

Acknowledgments

The authors would like to thank the following people for their comments, suggestions, and other contributions: Alexei Akhounov, Alex Amsel, and Sebastian Klier.

References

[1] R. H. Coase. The Nature of the Firm. *Economica*, 4(16):386–405, 1937. ISSN 1468-0335. `http://www3.nccu.edu.tw/~jsfeng/CPEC11.pdf`.

[2] ERC: Token standard #20. `https://github.com/ethereum/EIPs/issues/20`.

[3] Peter Borah. EtherRouter. `https://github.com/ownage-ltd/ether-router`.

[4] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. `http://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf`.

[5] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378. Springer-Verlag, London, UK, 1988. ISBN 3-540-18796-0. `https://people.eecs.berkeley.edu/~raluca/cs261-f15/readings/merkle.pdf`.

[6] Vitalik Buterin. Merkling in Ethereum. `http://blog.ethereum.org/2015/11/15/merkling-in-ethereum/`.

[7] J.M. Keynes. *The General Theory of Employment, Interest and Money*, chapter 12. Harcourt, Brace, 1936.

[8] Griff Green. The MiniMe Token: Open Sourced by Giveth. `https://medium.com/giveth/the-minime-token-open-sourced-by-giveth-2710c0210787`.

[9] Elena Dimitrova. Token-weighted voting implementation. `http://blog.colony.io/token-weighted-voting-implementation-part-1-72f836b5423b`.

A Appendices

A.1 Gas-Efficient Reputation Penalty in Dispute Resolution

Once a dispute has been raised and settled one way or the other, the users on the losing side will lose reputation and those on the winning side will gain it. If there is then a disagreement during the reputation mining mechanism, we must be able to calculate on-chain, in a gas-efficient way, a specific reputational consequence of the dispute being settled. A dispute may affect the reputations of many users, but all of these reputation changes are represented by only a single entry in the 'reputation update log', so it is necessary to expand upon the process used to resolve this.

Given that users are able stake small amounts on each motion, an arbitrarily large number of users could theoretically be involved. Gas limits dictate that we must therefore not have any (on-chain) loops in this implementation.

A.1.1 Staking

As noted in Section 3.4.2, a motion requires 0.1% of the reputation queried and 0.1% of the corresponding fraction of tokens to be staked' to be considered valid, but that just 10% of this amount is sufficient in order to create such a proposal. The exact numerical values of how much reputation and how many tokens are needed, is set at the time the proposal is first created. We proceed with a detailed example.

Let us consider a situation where a motion requires 600 tokens and 1200 reputation points to activate. User A initiates the motion and puts up a stake of 100 tokens. In order to do this, A must have at least 200 relevant reputation points at the time. Assume that users B and C support the motion, staking 200 and 300 tokens (and having 400 and 600 reputation points) respectively.

Table 1: A table of stakes showing part of what is recorded during the dispute process up to the point where the 'change' side has received enough support of 600 total tokens staked.

Stake #	**User**	**Staked Tokens**	Σ^+
1	A	100	100
2	B	200	300
3	C	300	600

For simplicity, the table does not contain entries for reputation. The corresponding amounts of reputation at risk are implied.

Once the cumulative backing (Σ^+) reaches the threshold required (600) the motion becomes active. Now we assume that two users (D and E) oppose the motion with matching funds of 150 and 450 respectively.[25] Once the cumulative backing on the keep side (Σ^-) reaches the required threshold (-600) a dispute is triggered.

We assume that, in the dispute, the initiating users (A, B and C) were found to have been wrong and so will lose some of their stake. To keep this example simple, let us pretend that they lose 50% of their staked tokens to the opposing side (D and E). They will also lose a corresponding amount of relevant reputation, or all of their relevant reputation, whichever is smaller.

[25]We write negative numbers in the table to denote *opposing* stake.

Table 2: A table of stakes showing part of what is recorded during the dispute process up to the point where both the 'change' and 'keep' sides have received 600 tokens of support.

Stake #	User	Staked Amount	Σ^+	Σ^-
1	A	100	100	0
2	B	200	300	0
3	C	300	600	0
4	D	-150	600	-150
5	E	-450	600	-600

We will assume that all users have the appropriate amount reputation to lose (i.e. A, B and C did not lose their reputation between the time of backing this proposal and the resolution of the dispute). We will also assume the dispute only affected domain reputation, not skill reputation.[26] There are four transfers of reputation that must occur here:

1. User A loses 100 reputation to User D
2. User B loses 50 reputation to User D
3. User B loses 150 reputation to User E
4. User C loses 300 reputation to User E

Indeed, in a group of m people where some owe the others a debt, the maximum number of transfers required to make everyone whole is equal to $m - 1$. If the reputation being lost has p parents and c children, there are up to $p + c + 1$ domain-totals to be updated (as some reputation is destroyed), $p + c + 1$ reputations for the losing user, and $p + 1$ totals for the gaining user (who does not receive any reputation in any child domains). Thus there are up to $3 + 3p + 2c$ reputation updates that must occur at each of these steps. There are therefore $(m-1) \times (3+3p+2c)$ reputation updates in total. In the event of a disagreement regarding the reputation state, we must be able to access the n^{th} update directly when calculating an update on-chain. This is made possible by additional logging of data when stakes are made.

When a user stakes and opposes some existing stake that does not yet have a counterpart, we record the stakes that it is matching against as well as any remainder.

Table 3: Table showing additional data recorded for stakes that match against earlier stakes on the opposite side.

Stake	Match From	Match To	Remainder	Tx # From	Tx # To
-150	1	2	50	1	2
-450	2	3	0	3	4

When staking, the user supplies the 'Match From' and 'Match To' arguments. These can be checked to be correct on-chain in constant gas by using the values of Σ^+ and Σ^- recorded alongside previous stakes, and the remainder from the previous match. Then, when a miner is asked to prove

[26] In the case of affecting both, the number of updates required is doubled.

a particular transaction has been included, they can point to the row in this log that contains that transaction without the contract having to iterate over an arbitrarily long list. The user's client is required to do this iteration locally to find the row, but this does not require any gas expenditure.

A.1.2 Exact matching

For the 'reputation update log' to work correctly, we must know exactly how many reputation updates we have to consider. In the above example, it was $4 \times (3 + 3p + 2c)$, which could be calculated and recorded easily in the update log. However, consider an example where the staked amounts were

Table 4: An example staking for each side where fewer than the upper limit of transfers for four people are required. Only two transfers are required to exactly balance the users.

Stake #	User	Staked Amount	Σ^+	Σ^-
1	A	100	100	0
2	B	200	300	0
3	C	-100	300	-100
4	D	-200	300	-300

Even though there are four people, only two transfers are required — from user A to user C, and from user B to user D. This is because the users have accidentally matched themselves exactly, and so one transaction makes two users 'whole'. In order to accommodate this possibility in the reputation update log, we insert dummy reputation transfers in the log whenever an exact match occurs:

Table 5: Table showing what is recorded for stakes that match against earlier stakes on the opposing side, in the case where some match exactly. The entries with 0 stake are used to ensure there are four transactions recorded, even if not all are needed.

Stake	Match From	Match To	Remainder	Tx # From	Tx # To
-100	1	1	0	1	1
0	0	0	0	2	2
-200	2	2	0	3	3
0	0	0	0	4	4

These dummy insertions occur whenever the remainder is 0 — i.e. when the new stake has exactly matched the first unmatched stakes. This ensures that this log always describes as many transactions as there are people (the last entry is always a dummy transaction as the final transaction will always make two users whole). This means that regardless of how the users have matched up against each other, the event that is recorded in the reputation update log will have a known number of transactions equal to the number of staking users, even if some of those are 'null' transactions.

A.2 Reputation Decay Calculation Details

Reputation in any skill decays by a factor of two every 90 days. At each update (i.e. after every 1 hour), the new decayed value (u_{new}) is calculated by

$$u_{\text{new}} = u_{\text{old}} \times \exp\left(-\frac{\ln 2}{2160}\right) = u_{\text{old}} \times \exp\left(-k\right).$$

This calculation is applied separately to each user's skill, as well as the number that represents the total of all of those skills in the colony. Due to rounding error with the integer representation on the blockchain, these numbers will drift away from each other. However, we can show the accumulated error will be negligible. The amount of reputation that will be incorrectly missing after the first iteration will be, on average. $0.5N$ reputation wei, where N is the number of users that have this skill.[27] The 0.5 is the average fractional part lost during each calculation.

After the second iteration, the amount of reputation that is incorrectly missing is

$$0.5N \exp\left(-k\right) + 0.5N.$$

The second term here is the incorrectly lost reputation from this second set of calculations. The factor of $\exp\left(-k\right)$ has been introduced to the term representing the incorrectly lost reputation from the first set of calculations because some of that incorrectly lost reputation would have correctly decayed away by this point, and so it shouldn't be considered incorrectly lost.

It is apparent that this is a geometric series, and after b cycles of reputation update have passed, the amount of reputation incorrectly missing (R_{m}) is

$$R_{\text{m}} = \frac{N}{2}\left(\frac{1-\exp\left(-bk\right)}{1-\exp\left(-k\right)}\right)$$

where we have used the standard result for the sum of a geometric series. If we started with R_0 reputation, then the ratio of the incorrectly missing reputation to the total the colony believes exists is

$$\frac{R_{\text{m}}}{R_0 \exp\left(-bk\right)}.$$

This ratio becomes 1 when

$$b = \frac{1}{k}\ln\left(\frac{2R_0}{N}\left(1-\exp\left(-k\right)\right)+1\right)$$

which, for conservative values of $R_0 = 200 \times 10^{18}$ and $N = 1000000$ occurs after 38011 iterations, or over 4 years for a 1 hour mining cycle. At this point, even though the colony believes some amount of reputation exists, no users have it, and no users can make decisions related to this type of reputation.

This is the end-of-life for an inactive colony; if no activity takes place in it for 4 years that is worthy of earning reputation, then the colony will be irrecoverable — no-one will be able to create

[27]This ignores the incorrectly lost reputation from rounding error introduced when decaying the colony-wide sum of the relevant reputation, but as there is only one total and many more users, ignoring it does not change our conclusions. We also note that there is an implicit assumption here that all users have the same amount of reputation; this is a worst-case assumption, as if it is not true then once some users have lost all their reputation the reputation incorrectly lost on each cycle will drop below $0.5N$.

tasks to earn further reputation. This seems like a reasonable failure mode for an inactive colony, and it would take longer to reach for smaller colonies (with fewer rounding errors).

We now consider the case of an active colony. If the colony is active and creates A new reputation at every update cycle, how does the ratio between the figure taken to be the total reputation and the incorrectly missing reputation change over time?

R_m remains the same in this situation, but the total reputation the colony believes exists increases by A each cycle. After b iterations, we can show that the total reputation the colony believes exists is

$$R_0 \exp(-bk) + A\frac{1-\exp(-bk)}{1-\exp(-k)}.$$

As b tends to infinity — which represents the regime of a colony in a steady state — the ratio between this and R_m tends to

$$\frac{N}{2A}$$

i.e. for the discrepancy to be small between what the colony thinks the total reputation inside it is and the sum of all users' reputations, the reputation earned in each cycle should, on average, be much larger than the number of users. Given that reputation will be expressed in terms of numbers on the order of 10^{18}, this seems assured.

For calculating the exponential decay, we will use the first-order Taylor expansion of the exponential decay i.e. we approximate $\exp(-k)$ as $1-k$. Given that k is small, this will be a good approximation — the second order term is on the order of 10^{-7}. This error will cause all reputations to decay slightly faster than an exponential, but otherwise will have no effect.

When calculating the decay, in order to accommodate the fact that we are multiplying by a value close to one and only integers are available in Solidity, we will multiply the user's reputation by $K(1-k)$ (calculated off-chain), for some large value of K, and then divide by the same large factor K.