COLONY

Technical White Paper

20170920

Alex Rea, Aron Fischer, Jack du Rose

Introduction

Colony is the people layer of the decentralised protocol stack — an emerging collection of technologies built on Ethereum, which provide open, permissionless protocols for developers to create sophisticated products without having to build complex backend software. Augur provides a layer for prediction, 0x for exchange, Golem for computation, and Colony for human capital.

The Colony Protocol allows developers to integrate decentralised and self regulating division of labour, decision making, and financial management into their applications.

Colony brings about a new "Nature of the Firm" [1] by significantly reducing both the transaction costs of the market exchange mechanism for labour, and trust required for people to work together. This innovation makes pseudonymous, peer to peer organisations possible. Rather than centralised ownership and hierarchical management, smart contracts distribute ownership according to the value each individual contributes, and influence emerges from the bottom up through systematic peer review of contributed work.

This paper seeks to describe and define our best current understanding of the Colony protocol, but touches only lightly on use cases. Specifically, it discusses only on chain actions. In many areas, subject to use case, off chain functionality will be beneficial. This may include messaging or informal polling, but as it does not need to happen on chain, the specifics of how this occurs are beyond the scope of this document.

The Colony protocol is intended to be flexible and extensible. As it is the most challenging manifestation to consider, this permutation is designed with low trust, decentralised environments in mind. However, subsets of the functionality in different configurations would be applicable in different circumstances, such as where greater centralisation of trust is desired.

We envisage this protocol being integrated into a variety of applications. It could form the basis of a decentralised ridesharing service, claims handling in an insurance dApp, or to provide the framework by which a merchants' guild coordinates in a virtual world.

Contents

1 Overview

Section 2 outlines the makeup of the **Colony Network**, introducing the `ColonyNetwork` contract and the `Colony` contracts that it spawns. This section also includes a discussion on upgrading these contracts as the network develops, and how we will prevent bugs discovered in the system before it is mature from fatally affecting the network.

Section 3 introduces the Common Colony Tokens (**CLNY**) that will underpin the colony network. CLNY is the token of the **Common Colony** — a colony tasked with maintaining and developing the Colony Network. CLNY will be usable by all colonies, alongside Ether and their own native token, to pay for work. This section also describes how we plan for control of the Colony Network to be transferred to the Common Colony over time. The generalised equivalent of CLNY for every colony in the network is introduced in Section 5.

Section 4 describes the organisational structure of a colony. A colony is a tool to coordinate effort to achieve a collective goal. We believe that dividing this goal into more achievable units will be essential for the success of any colony; we call these units **tasks**. Tasks can be assigned as work to members of the colony. The creation, assignment and completion of tasks is the *raison d'être* of a colony. Successful colonies are likely to have many tasks open at any given time; in order to ease the management of tasks, colonies can be divided into **domains**. These make it easy to group related tasks together and separate them from other unrelated tasks in other domains.

Completing a task entitles the member to claim any **bounty** assigned to the task. Each colony is able to denominate bounties in its own token, Ether, CLNY, and other tokens that adhere to the ERC20 format [2] and are whitelisted by the Colony Network. Bounties are assigned to tasks before workers are, with the bounty held in escrow by the colony to ensure the bounty can be claimed when the work is completed. The funding allocation system of **Funding Proposals** is described in Section 8. Tokens are assigned to domains and tasks on a continuous basis; the funding flows are directed by the members of the colony, and are prioritised by the members' **reputation**.

The reputation system is introduced in Section 6. Reputation is a key feature of the colony network, and is required to create tasks and domains, as well as to fund them with tokens. Reputation is used to quantify the historical contributions of members to a colony, and to make sure they are justly rewarded. 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. 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. The details of this **"Reputation Mining"** process are the subject of Section 7.

Many decisions within a colony are made by consensus. Members are expected to monitor their colleagues actions, but hopefully will only rarely need to intervene. Intervention in this context means 'raising an objection' and is the subject of Section 9. Decision via vote is required infrequently within Colony because it is slow and carries a high coordination cost. A notable exception is in the case of dispute resolution. The **Dispute Resolution System** allows for any decision to be escalated to a vote of some or all members of the colony. Ballots are weighted meritocratically, according to voters' relevant reputation.

Colonies may be voluntary, non-profit, or for profit. A revenue generative colony may elect to pay out a portion of its revenue to its members. When the colony pays out rewards, the reward 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 rather than simply sitting on accumulated tokens. The details of the **Reward Payout Process** are contained in Section 10.

We of course want people to use Colony for as many different workflows as possible, even those that are not immediately apparent as being able to use Colony. Section 11 provides a brief outline of some more complex behaviours that we have envisaged being possible with the system described here, such as interacting with arbitrary contracts.

Throughout this document various numerical parameters are concretely specified. These values will be subject to empirical review when the colony network begins live operation and any parameter values proposed in this document should be seen as good-faith suggestions, not prescriptions for the final network. Similarly, nothing within this document should be understood as a guarantee that any given functionality will be either developed or deployed. In the event that any version of the Colony Network is live on the public Ethereum mainnet, it should be considered as is, and the reader should not be infer, irrespective of the content of this document, that any existing functionality will be upgraded beyond its current state.

2 The Colony Network on Ethereum

The Colony Network consists of a collection of contracts on the Ethereum blockchain. At the core of the network will be a `ColonyNetwork` contract. This contract is primarily responsible for managing the reputation mining process (described in Section 7), but also for general control of the network — for example, setting the fees associated with using the network, or deploying new versions of the `Colony` contract.

The instances of the `Colony` contract represent the individual projects being worked on in the Colony Network. These are originally deployed by the `ColonyNetwork` contract. These `Colony` contracts will also have an ERC20 compliant contract that represents the colony's own token deployed alongside them.

2.1 Contract upgradability

We want to ensure the future upgradability of the deployed system as we foresee the Colony Network being continuously developed. Providing an upgrade path is important to allow people to use Colony without preventing themselves 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 addresses 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 `delegatecall`s 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`.

We will ensure that in the case of a colony, the choice of upgrading the underlying `Colony` contract will never lie with 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.

2.2 Contract security

While we aspire to bug free contracts, the adoption of a 'defensive programming' mentality endeavours to limit the impact of any issues that manifest in the deployed contracts.

At launch, colonies will be able to be put into a 'recovery mode'. In this state, whitelisted addresses are able to access 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 addresses, the contract will then be able to be taken out of recovery mode once the contract has been returned to a rational state. Removal from recovery mode requires the approval of multiple whitelisted addresses. This ensures that a single whitelisted address 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 address signalling that the contract should enter recovery mode.
- Something that should always be true of the colony not being true — for example, after a task payout checking that the amount of funds promised to tasks 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 addresses to leave recovery mode must be reset whenever a variable is edited. A whitelisted address 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.[1]

The first whitelisted address is added at colony creation and is the creator of the colony. Whitelisted addresses can be added or removed by a simple majority vote of existing whitelisted addresses.

[1]We note that this is a loop only limited by the number of whitelisted addresses. An alternative implementation or a hard cap on the number of whitelisted addresses in each colony will therefore be required to ensure recovery mode can always be left.

3 CLNY Tokens and the Common Colony

3.1 Creation of CLNY Tokens

After the Colony Network Contract is deployed, the first colony created will be the Common Colony. Tokens in the Common Colony will be known as CLNY. CLNY will be initially generated during the Colony Network distribution period.[2]

Functionality will be added to the CLNY token contract as required, through the `EtherRouter` mechanism described in Section 2.1. The ability to add functionality is extremely powerful, and theoretically could allow the inclusion of additional functions to arbitrarily set balances or create additional tokens. This would destroy trust in CLNY, and thereby its underlying utility. For this reason, the ability to add new functionality to the CLNY token will never lie with the developers of Colony, but with a multisignature contract initially made up of members of the wider Ethereum community, and then subsequently the Common Colony itself.

3.2 Role of CLNY Holders and the Common Colony

CLNY holders have two primary roles. The first is the process termed **Reputation Mining**, described in Section 7.

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 Common Colony.

For these permissioned functions to be called by the Common Colony, a vote open to all CLNY and reputation holders must be conducted (see Section 11.4).

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 Common Colony is the beneficiary of the network fee (see Section 10.3).

Reputation in the Common Colony can be acquired by earning CLNY tokens completing tasks, and administrative and evaluative duties just as in any other colony (see Section 6.2). Reputation in the Common Colony can also be earned by participating in the reputation mining process (defined in Section 7.8), which is unique to the Common Colony.

3.3 Handing off decision-making power to the Common Colony

Common Colony 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 Common Colony.

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

Stage 1: Colony Team Multisig in control

Initially, the Network Contract's functions will be permissioned to only allow transactions from the multisig address 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 intermediate contract will be set up, to receive these permissions. This contract will allow the Common Colony (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 address under the control of the Colony team. In other words, the Common Colony 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 intermediate contract operating similarly to the first, but after a timeout — with no interaction from the Colony Team's address — 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 Common Colony 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 Common Colony itself.

Stage 4: Common Colony fully controls the network

Finally, the intermediate contract will be removed, and the Common Colony will have direct control over the Colony Network Contract with no control imbued to the Colony Team other than that provided by any CLNY held.

4 Structure and Hierarchy within a Colony

Colonies exist to enable collaboration between their members, and direct collective efforts towards some common goal(s). Facilitating effective division of labour is therefore, one of the most important functions of the Colony protocol.

Work is divided into discrete ‘tasks’, which can then be organised into ‘domains’.

4.1 Tasks

The smallest structural unit in a colony is the ‘task’. A task represents a unit of work requiring no further subdivision or delegation. A task has three roles associated with it:

- A manager — someone responsible for defining and coordinating the delivery of the task.
- A worker — someone responsible for executing the task.
- An evaluator — someone responsible for assessing whether the work has been completed satisfactorily.

These roles are assigned to addresses. While it is anticipated that roles within tasks will likely be assigned to individuals, there is nothing to prevent these addresses being contracts under the control of multiple people.[3]

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

- A due date.
- Bounties for each of the manager, the worker and the evaluator.
- A specification, or brief: to be used by the worker to guide the work, and by the evaluator for assessing the satisfactory completion of the task.

In order to create a task, the manager must possess sufficient reputation and must stake a number of Colony Tokens proportional to the amount of reputation within the domain. This stake is used to discourage spam, and provide a mechanism whereby the manager can be punished for bad behaviour.

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 8). A funding proposal can be made by anyone, but the manager is the natural person to do so. Payouts do not have to all be in the same currency, and a task’s payout can be made up of two or three different currencies.

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 are assigned to the domain (introduced in Section 4.2) that the task was created in. The manager’s stake is still only returned after a timeout period in order to ensure they can be punished if necessary.

Assigning the worker can only occur after the funds to satisfy the proposed payout have been received by the task, and both the manager and proposed worker have agreed to the assignment.

[3]With the protocol described in this version of the document, any reputation earned would be assigned to the contract in question and not able to be moved to the appropriate users. We would expect some further developed version of the Colony Network to be able provide this functionality to users.

The manager and the evaluator must also mutually agree to the assignment of the evaluator before the evaluator is assigned to the task. The brief and due date may be changed, the assigned worker removed, or the task cancelled, either by mutual consent, or via the dispute process.

After the task has been assigned, the worker may 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 **objections** to the final state of the task can be made and **disputes** can be initiated (see Section 9). Once three days have elapsed, no more objections or disputes can be raised. Once all pending disputes related to the task are resolved, the parties involved get punished or rewarded based on the final state of the task.

If the evaluator's rating for the work is changed as a result of a dispute, they get a reduced payout based on the discrepancy between their original score and the score that their peers determined to be more appropriate. If an objection has determined the manager or evaluator should be punished, they lose their stake, otherwise the stake is able to be reclaimed. The worker is paid based on the final score awarded to their work, taking into account the result of any disputes.

4.2 Domains

Of course, without structure, a large colony could quickly become difficult to navigate due to the sheer number of tasks —– domains solve this problem. A domain is like a folder in a shared filesystem, except instead of containing files and folders, it can contain sub-domains and tasks. This simple modularity enables great flexibility as to how an organisation may be structured. A toy example is shown in in Figure 1.

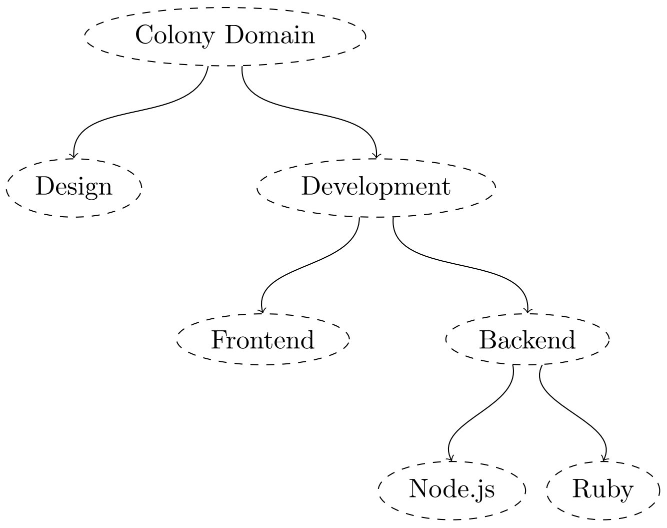

Figure 1: Parts of a possible domain hierarchy for a colony developing a web service.

This compartmentalisation of activity provides a tangible benefit to the colony as a whole. When objections are raised, they can be raised to a specific level in the colony's domain hierarchy.

This means that people with relevant contextual knowledge can be targeted for their opinion, and that when a dispute occurs, the whole colony is not required to vote in the dispute. Rather, only members with the relevant experience are asked for their opinion.

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 tasks together, or even multiple tasks that other colonies would consider a single task. We aim to provide a general framework that colonies may use however they see fit, and to only be prescriptive where necessary.

In order to create a domain, the creating user must have sufficient reputation in the domain that will become a parent, and must stake a small amount of colony tokens. These thresholds will be larger than for task creation, as the creation of a domain is more significant than the creation of a task. The token stake for creating a domain will be able to be claimed by the creator once a task has been successfully paid out inside the domain — this indicates the domain is legitimate.

In the event of a dispute being raised over the creation of the domain, if the domain is determined to be spurious, the stake is lost and the creator loses reputation.

5 The Colonies' own Tokens

Every colony has its own token. These are the tokens that, when earned as a task bounty, also create reputation for the receiver. What these tokens represent apart from this is up to the colony to establish. For example, they may have financial value, or they may be purely symbolic; some possible scenarios are outlined in Section 5.2.

5.1 Managing a colony's token supply

A colony is in control of changing the supply of its own tokens. More specifically, both reputation holders and token holders must agree to changes in the token supply, as both will be affected by it.

5.1.1 Token Generation and Initial Supply

When a colony is created, 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 colony-wide domain to assign to tasks or subdomains. The number of tokens available to the colony-wide domain can be updated at any time by a transaction from any user.

At colony creation, some tokens must also be assigned to addresses to allow users to stake tokens to create the first tasks. A one-off lump sum may also be created and made available to the colony-wide domain.

5.1.2 Increasing the TokenSupplyCeiling

It is crucial that new tokens cannot 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.

5.1.3 Changing the TokenIssuanceRate

The `TokenSupplyCeiling` represents the tokens that the token holders have granted to the colony in order to conduct business: to fund tasks and domains, and to hire workers and contributors. This is especially important during the early life of a colony when it has little-to-no revenue in other tokens to fall back on.

The `TokenIssuanceRate` controls how rapidly the colony receives the new tokens. If the rate is 'too high', tokens will accumulate in the pot of the top-level colony domain (or other pots lower in the hierarchy); usually this is not a big problem. 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.

5.2 Example of Token Usage

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 bounties for tasks that people may accept in the hope that the reputation earned by these token payments and the future revenue earned by the colony will eventually reap 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 tasks are paid in Ether, but include a number of the colony’s own tokens as well, equal to the expected number of hours worked on a task. The members of the colony would be responsible for assigning ‘correct’ token and Ether bounties to tasks. 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.

6 The Reputation System

6.1 What is Reputation?

Reputation is a number associated with an account which attempts to quantify the merit of a user's recent contributions to a colony. 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 (see Section 10.2). Because reputation is awarded to users by either direct or indirect peer approval of their actions, influence and rewards are distributed according to merit.

Colony aims to be broadly meritocratic. Consequently, the majority of decisions in a colony are weighted by the relevant reputation.

Unlike tokens, reputation cannot be transferred between addresses ; it represents an appraisal of the address'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 malfeasance; a description of how reputation is gained and lost is given in Section 6.2.

6.1.1 Reputation by Domain

The hierarchical structure of a colony was described in Section 4.2. 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. When reputation is lost in a domain with children, all child domains lose the same fraction of reputation. If a reputation update would result in a user's reputation being less than zero, their reputation is set to zero instead.

An example makes this clearer. Suppose a colony has a 'development' domain which contains a 'backend' domain and a 'frontend' domain, as in Figure 1 on page 10. 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 top-level domain of the colony. Reputation earned in the development domain will only increase the development and top-level 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 all of the Frontend, Backend, Node.js and Ruby domains).

6.1.2 Reputation by Skill

We envision 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 tasks corresponding to both design and development work, reputation earned by completing tasks 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 hierarchy is also maintained.

This global hierarchy of skills is available to all colonies, and is curated and maintained by the Common Colony. When a task is created, as well as being placed in a particular domain in the colony, it is also tagged with a skill from the skill hierarchy. When the worker earns reputation for successfully completing the task, they will earn reputation in the skill the task was tagged with and all parent skills. This is in addition to the reputation earned in the relevant domains. Conversely, if they are to lose reputation because their work is found inadequate, they will lose a proportional amount reputation from all child skills of the tag (if any exist), as is the case with the domain reputation. There is a top-level skill analogous to the top-level domain in a colony, of which all skills are descendants.

Even though the skill hierarchy is universal, reputation earned in the skill hierarchy 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.

6.1.3 Reputation by Colony

A user's total reputation in a colony is the sum of their reputation in the top-level skill and the top-level domain. This is the reputation they will be voting with in any decisions that require input from everyone in the colony. 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.

6.2 Earning and losing reputation

There are three ways to earn reputation in a colony.[4] The first is being involved with a successfully completed task. The second is through the dispute process. In both of these cases, the user has contributed usefully to the colony and is rewarded accordingly. The third way to earn reputation is upon the creation of a colony and the associated bootstrapping process (see Section 6.2.3).

Reputation losses can arise from a user being found responsible for a badly executed task, or being involved in the dispute process and the dispute being resolved against them. In addition, all reputation earned by users is exposed to a continual decay over time.

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

6.2.1 Reputation change from contributing to a task

Each task requires three roles to be assigned: the manager, the worker and the evaluator (as described in Section 4.1). If the bounty for the task is denominated in the colony's token, each of these roles are eligible to earn reputation when the task is completed as long as their work was well received.

The performance of the user who has completed the work is established when the work is submitted and then evaluated. At this point, the evaluator grades the work submitted by the worker, and the worker rates the manager's ability to coordinate delivery of the task[5] out of five stars.

[4]The Common Colony is a special case in which reputation may also be earned by Reputation Mining — see Section 7.

[5]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, their rating of their counterpart is assumed to be the highest possible and they receive a mildly negative rating.

In the case of the evaluator, a rating of 0-2 stars counts as them rejecting the work, and a score of 3-5 stars counts as accepting the work. Beyond that, we suggest the following guidelines for ratings:

0 stars: user submitted no meaningful work

1 star: user showed little activity relevant to the task, and remains far from completion on due date.

2 stars: user was unable to complete the task, but put in a reasonable amount of effort.

3 stars: user completed the task following the brief but there were issues during the work.

4 stars: user completed the task acceptably and there were no complaints.

5 stars: user completed the task to a higher standard than requested.

The actual number of reputation r earned by the worker for the completion of the task is then a function of this rating s and the token payout t:

$$r = t \times \frac{2s - 5}{3}.$$

Reputation lost or gained as a function of the star rating therefore varies linearly between $-\frac{5t}{3}$ and $\frac{5t}{3}$ for zero and five stars respectively, and a rating of four stars earns the user exactly t.

Similarly, the manager gets an amount of reputation based on their grading by the worker, but on a scale that only varies between $-t_{\mathrm{ev}}$ and t_{ev} (where t_{ev} is the manager's notional token payout for the task). They only earn this reputation in the current (and all parent) domains, not in the skill reputation hierarchy 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 reputation based on their token reward. There is no explicit rating of the evaluator, but as with all other payments and rewards, an objection can be raised before a payout occurs; if the evaluation is changed by such an objection, the evaluator's reward is reduced or turns in to a loss of reputation. For all participants, reputation updates occur and payouts are made available only *after* the objection window (described in Section 4.1) has closed and all disputes (described in Section 9) have been resolved at the end of the task. The reputation updates and payouts are based on the final state of the task and the difference, if any, between the initial gradings and final state of the task.

6.2.2 Reputation change as a result of Disputes

If a dispute occurs, causing a vote among some portion of the colony, each side will have had to stake some number of tokens. Those who staked on the side determined to be right gain their stake back, plus some tokens that have been lost by the losing side. There will also be a reputation change as a result — those on the losing side will lose some reputation, and some of that will be gained by the winning side. Section 9 provides a full description of the dispute mechanism and the amount of tokens and reputation each side loses and gains.

We note here that, unlike tokens, reputation is never staked during this process. A user must hold greater than a minimum threshold of reputation in order to be eligible to stake tokens and take certain actions, and this threshold only applied at the time the action is taken. If their stake is lost, they will lose some reputation as a consequence for being on the losing side. It is possible that the user will not have enough reputation to lose by the time this reputation update occurs due to bad behaviour elsewhere. This is acceptable from a game-theoretic perspective; the user has still had to stake and lose tokens, and these cannot be 'double-spent'.

6.2.3 Bootstrapping reputation

Since a colony's decision making procedure rests on reputation weighted voting, we are presented with a bootstrapping problem for new colonies. When a colony is new, no-one has yet completed any work in it and so nobody will have earned any reputation. Consequently, no objections can be raised and no disputes can be resolved as no-one is able to vote. Then, once the first task is successfully completed, 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 addresses to have initial reputation assigned to them in the colony-wide 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 colony-wide 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 task being paid out. Users receiving reputation are presumably the colony creator 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.

6.2.4 Reputation Decay

All reputation decays over time. Every 600000 blocks, a user's reputation in every domain or skill decays by a factor of 2. This decay occurs every hour, rather than being a step change every three months 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 task) do not permanently distort the distribution of reputation but instead serves to smooth out the effects of such fluctuations over time.

6.3 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 `rep_id` that is unique across the whole network. When a skill is created, additional properties are recorded and initialised.

$$\texttt{skill\_id} \rightarrow \begin{cases} \texttt{n\_parents} & \text{total number of parents} \\ \texttt{parent\_n\_id} & \text{the } \texttt{rep\_id} \text{ of the } n^{\text{th}} \text{ parent, where } n \text{ is an integer power} \\ & \text{of two larger than or equal to 1.} \\ \texttt{children}[\cdots] & \text{array of } \texttt{rep\_id}\text{s of all child skills} \\ \texttt{n\_children} & \text{total number of child skills} \end{cases}$$

Upon creation, `children[]` and `n_children` are empty. These two fields in all parents are updated with the `skill_id` of the new skill on creation.[6]

Storing these pieces of data on-chain is required, as they are used by the reputation mining protocol (see next section) and the procedures for escalating disputes (see Section 9). They are stored under the control of the `ColonyNetwork` contract.

6.4 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 suffering the reputation loss or gain.
- The amount of 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 `n_parents` and `n_children` for each skill and domain, as described in Section 6.3.
- 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 6.2.2), 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.

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 7), the `ColonyNetwork` contract can use this record to establish whether an update has been included correctly.

[6]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. Back-of-the-envelope calculations suggest this corresponds to a depth of around 80, which we don't believe our users will be limited by.

7 Calculating Reputation: Miners and Merkle Proofs

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 from 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.*

7.1 Merkle Trees and Proofs

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

Consider the tree shown in Figure 2. The data leaves of the tree (1, 2, 3 and 4) are each hashed individually to give A, B, C and D. These are then repeatedly hashed pairwise until only a single hash remains, indicated by G in Figure 2. In the event of a starting or an intermediate array being an odd number (which will always happen for a starting array that is not a power of two), the hash contained in the last element is hashed with itself.

The resulting structure is known as a Merkle tree. In order to prove that the element `1` is in the tree with root `G`, one submits a Merkle proof containing the information `1, [B,F], [l,l]`. The first argument is the element whose existence is to be proved. The second argument is the list of hashes that the hash of the first element should be hashed with in succession. The last argument is an array of `l`'s and `r`'s that indicates whether the hash calculated so far should be hashed on the left or the right of next element in question. So to show that `3` was in the tree with root `G`, the proof would be of the form `3, [D,E], [l,r]`.

Note also that the array of `l`'s and `r`'s nothing more than a binary representation of the leaf node's index in the tree. When expressed in this way, we refer to the index as the 'path' in the Merkle proof. We refer to the objects that get hashed along the way (i.e. `D` and `E`) as the 'siblings'.

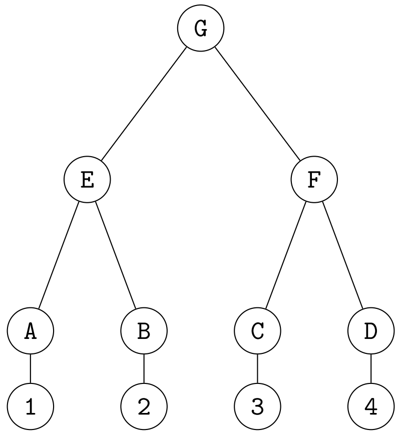

Figure 2: A simple Merkle tree. Elements A, B, C and D correspond to the hashes of 1, 2, 3 and 4. E is the hash of A concatenated with B.

7.2 The Reputation Tree and the ReputationRootHash

The data leaves in the reputation tree are the reputations all users have in all skills, as well as the colony-wide totals. Such a leaf consists of the following data:

$$R = \begin{cases} \texttt{rep\_id} & \text{the id of a skill or domain identifying the type of reputation,} \\ \texttt{colony\_id} & \text{the colony the reputation is held in,} \\ \texttt{user} & \text{the address holding the skill,} \\ \texttt{amount} & \text{the numerical value of the reputation.} \end{cases}$$

All individual reputations are assembled into the **"Reputation Tree"** which is a Merkle 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 put into a Merkle tree in the usual way (described in Section 7.1).[7] 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 $\mathcal{RH}$.

7.3 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 leaves in the tree. They then take the set of reputation gains or losses that were not in the last state submitted, and are to be included in the next state. They apply the reputation updates to each user in each colony, updating or adding leaves as necessary, to end up with a new list of reputations for all users and colonies.

[7]The ordering of the data leaves is only determined by when these reputations were first earned.

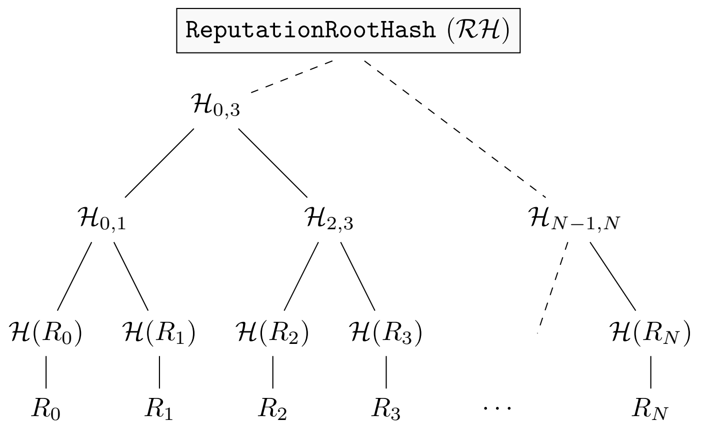

Figure 3: The Merkle tree of users' reputations with `ReputationRootHash` as the root. We use $\mathcal{H}$ to indicate the `keccak256` hash function. Above the second row, each element is the hash of the concatenation of its two children.

These new reputations are then hashed and assembled into a new Merkle 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.

7.4 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. Further, the miner also submits the IPFS or Swarm hash of a document containing the entire state tree, though this is only for convenience; any user can construct this locally based on the blockchain history.

Who can submit a new root hash?

All CLNY token holders are eligible to become miners and participate in the reputation update process, but since any user can calculate the correct root hash locally, it would 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, Common Colony Token holders must stake some of their tokens to become 'reputation miners'. A submission will only be accepted from a miner if

$$\texttt{keccak256(address, N, hash)} < \texttt{target}.$$

At the beginning of the submission window, the target is set to 0 and slowly increases to $2^{256} - 1$ after 150 blocks. We limit the total number of miners allowed to submit a specific hash to 12.

The variable N that goes into the hash is some integer greater than 0 and less than the number of tokens the Common Colony Token holder address has staked divided by 10^{15}, 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 10^{15} 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 7.7).

Miners will also be required to have staked their tokens for a few update cycles before they are eligible to submit or support a hash.

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 7.5)

Mining Rewards

When a state is accepted, a number of (newly minted) Common Colony 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 Common Colony (in a special mining skill, which only users in the Common Colony 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 7.8.

7.5 Dealing with false submissions

7.5.1 The Challenge-Response Protocol

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 7.8). Thus our task is not to validate the correct hash but to invalidate the false ones.

We must prove all but one submission incorrect by having each submission navigate a series of *challenges*. These challenges refer to events that happened in the colony network within the last update cycle that have an effect on reputation. The *responses* to the challenges are Merkle proofs that the corresponding reputation update was properly handled. Anyone is able to respond to a challenge, regardless of who submitted the original hash; this should ensure that the correct state is always defended.

We begin with the scenario where only two submissions are made, and one is correct.

First we consider the case in which the same hash has been submitted twice, but with a disagreement about the number of leaves it contained. In this situation, the users are required to submit a Merkle proof for the last leaf in their tree. We are able to tell if a submitted proof corresponds to the last leaf in the tree, as all sibling hashes must be hashed on the left. Noting the positions in

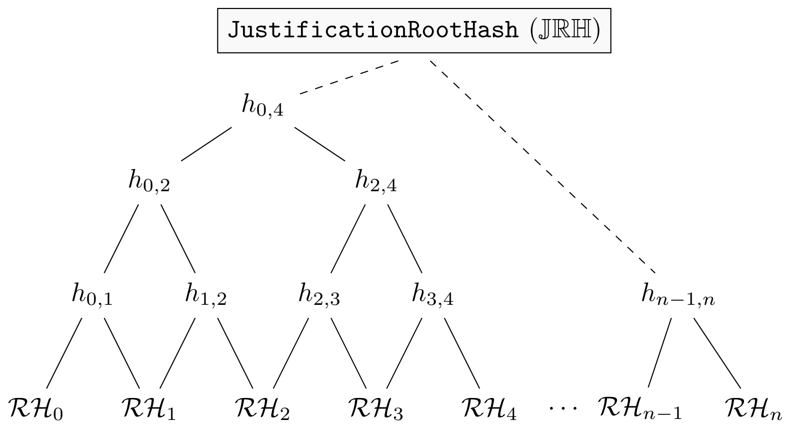

Figure 4: The Justification Tree. Note that each intermediate leaf $\mathcal{RH}_i$ is hashed into the tree twice.

the Merkle proof in which a hash is hashed with itself also allows us to tell the index of this last leaf; if it corresponds to either of the submitted states, that one is correct.

We now consider the more complicated case where two different hashes have been submitted.

1. The Justification Tree

The first step is for both parties to upload a justification of their submitted hash. This justification consists of a second Merkle root and two Merkle proofs.

The Merkle root in questions is the `JustificationRootHash` ($\mathbb{JRH}$); it is the root of the 'Justification Tree' – a tree where each leaf represents a complete reputation state i.e. each leaf is a `ReputationRootHash`.

The left-most leaf of the Justification Tree ($\mathcal{RH}_0$) is the final accepted reputation state from the last update. Both parties must submit a proof that their tree does indeed start at $\mathcal{RH}_0$. Note that in a Merkle proof for a left-most leaf, all siblings are hashed on the right.

The right-most leaf of the Justification Tree ($\mathcal{RH}_n$) is the `ReputationRootHash` they originally submitted. Both parties must submit a Merkle proof that their tree does indeed end with this hash. Note that in a Merkle proof for a right-most leaf, all siblings are hashes on the left; furthermore, by noting the steps at which a hash is hashed with itself, we can determine the index of the last leaf. This index is required to be n — the number of reputation updates in the log.

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 4. Note that in the first stage of the tree, every neighbouring pair of data leaves is hashed, and so any pair has a shared Merkle proof to the root. We do this so that each element in the first row of hashes (indicated by $h_{0,1}, h_{0,2} \ldots h_{n-1,n}$ in Figure 4) represents a transition between two states. It is these transitions that are the subject of the mining dispute resolution process.

Since any two differing submitted states agree on the first leaf $\mathcal{RH}_0$ (which is the `ReputationRootHash` accepted at the end of the previous iteration of the mining process), and disagree on the last leaf

$\mathcal{RH}_n$ (the hash they submitted), there must be a hash ($h_{i,i+1}$) that represents the transition from $\mathcal{RH}_i$ to $\mathcal{RH}_{i+1}$) 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})[8], 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 first differ.

2. Searching for the discrepancy

The contract requires both parties to submit repeated Merkle proofs that specific reputation updates were handled correctly until the first discrepancy between the two submissions is found.

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. Recall that if h does not have an immediate sibling, $\overline{h}$ is taken to be equal to h as per our rule of hashing these elements with themselves.

Along with their justification root hashes $\mathbb{JRH}^A$ and $\mathbb{JRH}^B$ both parties have already submitted proofs for the left-most leaf. These proofs have the simplified[9] form:

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

and

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

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 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

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

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 how the j^{th} reputation update from all updates to be applied this cycle (Section 6.4) was included in their 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 $\overline{h_{0,2^k}}$. Specifically this proof consists of

[8] We start counting at zero, i.e. the transition from $\mathcal{RH}_0$ to $\mathcal{RH}_1$ is the 'zeroth'.

[9] Since we know that the proofs concern the first and last leaf of the tree, no array of left-right information is needed.

(i) The root hash before the update was applied ($\mathcal{RH}_j$).

(ii) The root hash after the update was applied ($\mathcal{RH}_{j+1}$).

(iii) A Merkle proof showing the inclusion of $h_{j,j+1}$ in the relevant justification sub-tree.

The same 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 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 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 found $h^A_{i-1,i} = h^B_{i-1,i}$ while $h^A_{i,i+1} \neq h^B_{i,i+1}$. The process has thus determined that it is the i^{th} reputation update from the log where the two submissions differ (i.e. they agree on $\mathcal{RH}_i$ but not on $\mathcal{RH}_{i+1}$).

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

(iv) A Merkle proof for the reputation (R_j) affected by the i^{th} update transaction.

The Merkle proof that proves the inclusion of R_j (before the reputation update) in the reputation root hash $\mathcal{RH}_i$ and the Merkle proof of the inclusion of R_j (after the reputation update) in $\mathcal{RH}_{i+1}$ must only differ in the state of the reputation R_j itself. This is because the reputation trees with roots $\mathcal{RH}_i$ and $\mathcal{RH}_{i+1}$ differ in only a single leaf (that of R_j).[10]

When either party submits valid Merkle proofs, showing that the before and after state they claim to be correct is included in the $\mathcal{RH}_i$ and $\mathcal{RH}_{i+1}$ in their justification tree, the reputation update is also calculated on-chain to determine whether the submitting party has done their calculation correctly. If the calculation was not done correctly, the submission is identified as false.

The process does not terminate until *both* parties have submitted their proofs, or the timeout is reached. There is an edge case in which it appears as if the first to submit has done the calculation correctly, but in actual fact has erroneously added a new leaf in the reputation tree (see Section 7.6.2) even though no new leaf was required. This error can be identified when the other party submits a correct proof.

If there are more than two submissions for the new `ReputationRootHash`, this process is conducted in parallel between multiple pairs of submitted hashes, and then repeated, until only one submission remains.

If less than an hour elapses from submissions opening to only one submission remaining, the next submission window only opens when an hour has passed from the start of this window. If more than an hour has passed, the next submission window opens immediately.

Gas-cost Considerations and Alternative Implementations

There is a trade-off between the number of transactions needed until the first discrepancy between two submissions is found and the size of the storage required by the contract. As written above,

[10]In the case of the addition of a new leaf, the proof for an R_j before the update is replaced by a proof for the current last leaf in the tree. They also submit the merkle proof for the new R_j after the update. These two proofs submitted will differ in more than just R_j, but if valid must converge at a predictable location in the Merkle proof, dependant on the number of leaves in the tree.

the contract stores only the last hashes of the Merkle proof, thereby reducing the range of possible values of i by a factor of two at each step. If the contract stores more of the submitted hashes that constitute the Merkle proofs, the range can be reduced faster and fewer challenges will be required. Ultimately it will be a gas-cost analysis that determines the exact algorithm to use.

We also note at this point that a malicious submission, performing the attack described in Section 7.7, can carefully craft their false justification trees to require the maximum number of challenges to resolve. Such a worst-case submission can not be made in the alternative implementation in which the full[11] Merkle proofs are stored (and compared) on-chain, and in which challenges are issued for an index j picked pseudorandomly from within the range of possible values.

7.6 Calculating reputation updates

7.6.1 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 6.3, 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.

When the discrepant state transition is found, the users supply the index of the event in the on-chain log that corresponds to that reputation update. This means that the contract does not have to iterate over the whole list expensively, but the contract can simply check the correct reputation update is being considered, and then confirm that the calculation made corresponds to the correct reputation update. To do this, we assert that children are always updated in order, then parents, and then the reputation in question itself. In addition, all colony-wide totals of reputation are always updated in this order before any user-specific reputation.

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.

[11] Or as much of the head and tail of the Merkle proofs as the gas limits will allow.

Recording the number of leaves in the reputation tree is required to accommodate the decay calculations that must be done at the start of each update. Before any new reputation is earned or lost in an update cycle, all existing reputations owned by users decay (see Section 7.6.4). There is a decay calculation for every leaf in the previously accepted reputation tree. 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.

7.6.2 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:[12]

- 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.

7.6.3 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 9) 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

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

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.

7.6.4 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 B.

7.7 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.[13] 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

[13] 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.

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 Common Colony 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 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.

7.8 Costs and Rewards of Mining

In order to be involved in the reputation mining process, Common Colony 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 address 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 (7.4) on page 21.

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

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

Suppose that the reputation earned in the Common Colony 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

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

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

$$M=\frac{h}{4}=\frac{R_{tot}\left(1-e^{-k}\right)}{4}. \tag{2}$$

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 (2).

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

$$w_n=\left(1-\exp\left(\frac{t_n}{T}\right)\right)\times\left(1-\frac{n-1}{N}\right) \tag{3}$$

where t_n is a number of blocks that the n^{th} miner has staked their tokens for and T and N are normalising constants. T is set to a number of blocks representing 3 months, and N is set to twice the length of the list of submitters — in our case $N=24$.

The first factor in equation (3) encourages users to stake their tokens for long periods of time when they register as miners. When locking tokens for T blocks, 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 (3) 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 also adds the reputation changes for the miners to the start of the reputation change log, and will be included in the next update cycle. Finally, this transaction also makes the miners eligible to claim their CLNY token reward.

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

$$m_n=M\frac{w_n}{W}\quad\text{where}\quad W=\sum_{n=1}^{12}w_n \tag{4}$$

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

7.9 Emergency Shutdown

Section 2.2 described a transaction from a whitelisted address 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 addresses 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.

8 Finance: Managing Funds and Bounties

This section deals with the mechanisms by which a colony *allocates* financial resources to domains and tasks. The norm is for resources to be allocated from the general to the particular, and that all allocations with sufficient reputational 'backing' may proceed without a vote. As long as there is no disagreement, everything will run smoothly and automatically.

For how revenue earned by a colony is handled see Section 10.

8.1 Tokens and Ether

Every colony has its own native ERC20-compatible token. These tokens are under the control of the colony contract and may be used to pay for work done in the colony. Tokens only leave the control of the colony upon being paid out for completed tasks.[14]

To pay out tasks, in addition to local tokens, a colony may also use Ether, CLNY and other ERC20 tokens that have been explicitly whitelisted by the Colony Network.

8.2 Pots and Funding Proposals

All tokens and currencies are administered by the colony contract; it is responsible for all the bookkeeping and allocations.

Each domain and each task in a colony has an associated *pot*. A pot can be thought of as acting like a wallet specific to a particular domain or task. To each pot, the colony contract may associate any number of unassigned tokens it holds. Depending on context, the funds in a pot may be referred to as the bounty, the budget, the salary or working capital.

Funds are transferred between pots through *funding proposals*.

A Funding Proposal consists of the following data

- `Creator` – The person that created the proposal.
- `From` – Pot funds are coming from.
- `To` – Pot funds are going to.
- `TokenType` – The token contract address (0x0 for ether).
- `CurrentState` – The state of the proposal (i.e. inactive, active, completed, cancelled).
- `TotalPaid` – How much has been transferred along this funding proposal so far.
- `TotalRequested` – The maximum amount to transfer after which this funding proposal is considered 'completed'.
- `LastUpdated` – The time when the funding proposal was last updated.
- `Rate` – Rate of funding.
- `PunishCreator` – Whether the creator should be punished upon completion.

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 atypical circumstances present themselves. The basic funding proposal may start funding the target straight away, whereas

[14]Currently, the only way this rule can be broken is by the Colony conspiring to abuse the Arbitrary Transaction feature described in Section 11.4.

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 the funding rate should be otherwise controlled (e.g. in the case of paying a salary).

For either funding proposal, the assignment to pots associated with domains or tasks is purely a bookkeeping mechanism. From the perspective of the blockchain, ether and tokens are held by the colony contract until they are paid out when a task is completed.

8.2.1 Creating a Funding Proposal

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. They must stake an equivalent fraction 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.

8.2.2 From, To and TokenType

The purpose of a funding proposal is to move tokens of `TokenType` from a pot `From` to a pot `To`.

The `TokenType` may be any ERC20 token whitelisted for use in the network, Ether, CLNY or the Colony's own Token. The `From` and `To` fields must be pots associated with a domain or a task in the colony. If the funds are to move 'downstream' from a domain to one of its children, a basic funding proposal is often sufficient.

8.2.3 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 completed when its `TotalPaid` reaches `TotalRequested`. Any other state changes must be made through the dispute mechanism (see Section 9).

8.2.4 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.

8.2.5 Rate and LastUpdated

When a funding proposal is eligible to accrue funds (see Section 8.2.6) it does so at a specific `Rate`. 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 an update. 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` and the current time is recorded as `LastUpdated`.

`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.

8.2.6 The Funding Queue

Active Funding Proposals that share the same `From` pot are ordered in a queue. At the top of the queue are the priority funding proposals, followed by the basic funding proposals. PFPs are ordered by the total reputation in their domain[15] — while basic funding proposals are ordered by the reputation 'backing' them. The details of this procedure are outlined below.

8.2.7 Basic Funding Proposals

A basic funding proposal (**BFP**) is a funding proposal from some domain's 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`, or through the dispute mechanism. In either case, the stake is only able to be reclaimed if `PunishCreator` has not been set to true via a dispute by the end of a timeout period.

8.2.8 Ordering of BFPs

Basic funding proposals are ordered in the *Funding Queue*. Only one of them can receive funds at any one time. The proposals are ordered by 'amount of reputation backing the proposal'.

When created, a basic funding proposal gets placed at the bottom of the queue. Users can give a proposal 'backing' weighted by their reputation in the source domain[16] at the time of backing[17]. 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.

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 funding proposal at the top of the queue accrues funds.

8.2.9 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 the `TokenType`) per week. The goal is a steady and predictable allocation of resources directed collectively by the domain's (reputation weighted) priorities.

[15]The domain of a PFP is the domain that voted on it becoming active — this will be the last common ancestor of the source and target pot domains unless an escalation has occurred.

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

[17]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.

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. To rectify this, we will allow users to update another user's backing to reflect their updated reputation scores, but we don't expect this functionality to be used often. We would only anticipate it being used if a user lost a lot of reputation due to some very bad behaviour, and other users wanted to prevent a bad funding proposal backed by the same user from being completed before it could be cancelled by other means (i.e. via dispute, described in Section 9).

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%.

8.2.10 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 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.**

Once the the BFP has been fully funded, for a period of three days anyone can make a proposal that `PunishCreator` should be set to 'true', and the creator lose their stake. This proposal takes the form of an Objection (Section 9). If no such proposal is made, or the proposal fails to pass, then the creator can reclaim their stake. Once the fate of the stake is decided, and the creator either reclaims the stake or loses it and a matching amount of reputation, the BFP is set to the `completed` state.

8.2.11 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.

8.2.12 PFPs and the Funding Queue

Active Priority Funding Proposals take priority over Basic Funding Proposals and so they are placed at the top of the funding queue. They are ordered by the total reputation of the domain that voted to activate it and, in case there is a tie, by the actual amount of reputation that voted to activate. Thus PFPs that are higher in the domain hierarchy come before those lower down.

As with BFPs, any user can 'ping' an active PFP at the top of the queue to cause the contract to update the funds available to the recipient pot. `TotalPaid`, `LastUpdated` and `CurrentState` are updated as required.

8.2.13 The 24h waiting period for PFP updates

Priority Funding Proposals take precedence over Regular Funding Proposals. To avoid the situation in which long running PFPs block the BFP process entirely, a limit is placed on how often and PFP can be updated ('pinged'). We say a PFP can only be pinged when it is first activated[18] and when its `LastUpdated` time is at least 24 hours old.

The result of this rule is that fast payments are still possible — in such a case the PFP's `rate` is set very high and the proposal is fully funded at the initial ping, while also allowing long-term lower-rate PFPs that do not block the entire BFP process.

8.2.14 When is a Funding Proposal eligible to receive funding?

A Regular Funding Proposal may receive funds when pinged if it is active and at the top of the BFP funding queue and when the `LastUpdated` time of the PFPs are less than 24 hours old.

A Priority Funding Proposal may receive funds when pinged if it is active and all PFP ahead of it in the funding queue have been updated less than 24 hours ago.[19]

8.2.15 Editing Funding Proposals

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).

8.2.16 Cancelling Funding Proposals

The `creator` of a funding proposal may set its `CurrentState` to `cancelled`. 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 4.1). Beyond this it must be possible for a colony to cancel a funding proposal without the creator's involvement. This is done through the objection mechanism described in Section 9.

[18]In this initial update the time elapsed since last update is taken to be 24 hours.

[19]In order to avoid hitting the gas limit due to unbounded loops, it will be necessary to maintain two orderings for the PFPs, one by priority and one by `LastUpdated`.

When a task is cancelled, funding proposals that have that task's pot as their target (`To`) also enter the `Cancelled` state when they are next pinged, and no funds are reallocated. However, the funds that had already been transferred are not automatically returned; it will require a PFP to return the funds 'upstream'.[20]

8.3 Paying out a Task Bounty

Ultimately, via the mechanism described above, some tokens have found their way into a pot associated with a task. Upon completion of the specified work and approval by the evaluator, the worker has earned the contents of the pot. However, even once the work has been approved, there remains a period of time before the funds can be requested to be paid out by the receiving address. This is enforced to allow a period where a user of the colony can object to the payout, accusing it of fraudulence or similar.

If a user objects to the payout, they can raise an objection to void the task payout. The objection may be raised to a parent domain of the task in question — or even escalated to the colony domain itself. The choice of domain lies with the user making the objection. Users vote to resolve any resulting disputes with votes weighted by their reputation in the domain the objection was raised to. In the event that their dispute is upheld, the funds can be returned to the domain that contained the task. If the voters approve of the payout, no changes are made and the user will be able to claim their payout after all.

While a payout is under dispute, the timeout period continues to run. After the end of the dispute period, no more objections are able to be raised, but any existing objections are resolved before payout is allowed. This is to prevent users being able to continually raise disputes to prevent payout.

Once the tokens have been paid out, they are under the control of the user — 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.

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

9 Objections and Disputes

The most successful organisations are those which are able to effectively and efficiently make decisions, divide labour, and deploy resources. These are trust problems, which have traditionally been solved by management hierarchies. But colonies are intended to be low trust, decentralised, and pseudonymous — a hierarchy is not suitable.

The solution to collective decision making is usually voting, but Colony is designed for day to day operation of an organisation. 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 4.1), nor do basic funding proposals (Section 8) or any number of administrative actions throughout the Colony system.

The **Dispute System** provides a self regulating mechanism to provide a balanced set of incentives for users to keep their colony running harmoniously. It is there to resolve disagreements and to punish bad behaviour and fraud. The dispute mechanism allows colony members to signal disapproval and potentially **force a vote** on decisions and actions that would otherwise have proceeded unimpeded.

What are Objections?

When a member of a colony feels that something is amiss, they can *raise an objection.* By doing so, they are fundamentally proposing that a variable, or more than one variable, in the contract should be changed to another value. For this reason we call supporters of the objection the 'change' side and opponents the 'keep' side.

The user raising the objection must also put up a stake of colony tokens to back it up (see Section 9.2). In essence, they are challenging the rest of the colony to disagree with them. In the spirit of avoiding unnecessary voting, the objection will pass automatically *unless* someone else stakes on the 'keep' side and thereby elevates the objection to a *dispute.*

What are Disputes?

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

9.1 Raising Objections

The user raising an objection submits the following data:

- the data that should be changed
- 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 objection, and what the initiator believes the state should be.[21] The second and third points concern *escalation.*

[21]The exact structure of this is dependent on the method used to implement contract upgradability. The function

In Colony you cannot escalate a decision to higher management, you can only escalate to bigger groups of your peers.

For example, suppose that the objection concerns a task in the domain 'development of our website'. The objector could choose to have all 'development' reputation vote on it — we say the decision was 'escalated to the development domain'. In this example, the third point would be a proof that the domain 'development of our website' was indeed a subdomain of 'development'.

The highest domain any decision can be escalated to is that of the entire colony, where all domain reputation is entitled to vote. Similarly, an escalation to the all-encompassing top-level skill allows all skill reputation to vote.

Whenever an escalation occurs, we need to ensure that the reputation we are escalating to is a direct 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 skills and domains) when they are created, which includes pointers to at least the direct parent.[22] When a user creates an objection, instead of directly specifying the skill or domain they are escalating to, they provide the steps needed to get there from the skill or domain associated with the variable that is to be changed. This ensures that the skill or domain they escalate to are direct parents of those associated with the variable.

9.2 Costs and Rewards

9.2.1 Cost of raising an objection

To create an objection, 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 objection depends on the level they are escalating to; the 'higher up' the decision goes, the higher the reputation requirement. To be able to create an objection, the user must have 0.1% of the reputation queried and then must stake 0.1% of the corresponding fraction of tokens. Thus, if an objection appeals to 13% of total colony reputation, then objecting 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.[23] In this case the objection 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 objection 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 objection outright.

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.

[22] Each reputation type contains pointers to its parent `parent_id` and its "n^{th} parent" `parent_n_id` for all n that are powers of 2. See also Section 6.3

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

9.2.2 Cost of defending against a raised objection

Once enough tokens have been staked against an objection it becomes active and, barring any further user actions for three days, the suggested change will take place when the objection 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 raised.

If the 'change' side does not garner enough support in three days, the objection 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 and it occurs when 'pinged'.

9.2.3 Voting on Disputes

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

The duration of the poll is determined by the amount of reputation eligible to vote in the poll 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. To make a vote, the user submits a hash that is `keccak256(secret, option_id)`, where `option_id` 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, option_id` and the contract calculates `keccak256(secret, option_id)` 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. We suggest a (hash) of the consequence field of the poll signed with their private key. This is easily reproducible by a client at a later date with no local storage required.

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]. 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 top-level colony 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). We define quorum to be more than 10% of the reputation eligible to vote has done so. If quorum is not reached, no changes are made and all participants get their remaining staked tokens returned.[24]

9.2.4 Consequences of the vote

If quorum has been reached in a dispute vote, and the 'change' side won, then the variable in question is changed, but only if the reputation that voted for this outcome is more than previous votes on the same variable (see Section 9.2.5). If the 'keep' side won, then the variable is not changed. In

[24]Some of the staked tokens on the change side will have been used to compensate voters.

either case, alongside the variable that may or may not have been changed, 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.

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):

$$L = 1 - \frac{R}{3}. \tag{5}$$

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:

$$\Delta = 0.9 \times \min\left\{\frac{w - 0.5}{L - 0.5}, 1\right\} \tag{6}$$

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 objections 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 instigators of the dispute 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 disputes are invoked.

9.2.5 Repeated Disputes

In order to reduce the number of repeated objections 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.

To ensure that the variable can always be changed if necessary, this threshold for changing the variable is ignored if the dispute was raised to the top-level domain of the colony.

9.3 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 instigator.

9.3.1 Reputation weighted voting

Most votes in a colony will be due to disputes 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.

9.3.2 Token weighted voting

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 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.

Instead, once a vote enters the reveal phase, any user who has voted on that vote 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.

It is possible to achieve this locking in constant gas by storing all submitted secrets for votes in a sorted linked list indexed by `close_time`. 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.

9.3.3 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 5.1) or when deciding whether to execute an arbitrary transaction (see Section 11.4).

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

10 Revenue and Rewards

What is Colony Revenue?

A colony may sell goods and services in exchange for tokens, for Ether or for one of the ERC20 tokens whitelisted for use on the network. Whenever a colony receives such payments, we say that the colony has earned *revenue*.[25]

Revenue is distinct from a colony's working capital. The latter is the sum of all tokens held by the colony for use in funding requests i.e. the funds in the pot belonging to the colony-wide domain in the colony (see Section 8), while the former, revenue, has its own dedicated pot.

What are Colony Rewards?

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

It is expected that most of the revenue will *not* go towards rewards, but towards replenishing the working capital.

10.1 Processing Revenue

Revenue accumulates in a dedicated revenue pot. In order to be processed, a user has to make a special transaction, triggering the revenue distribution process. They define a token that they wish to process accumulated revenue for. This process has slightly different effects on a colony's own token on compared to Ether, CLNY and the whitelisted tokens.

Revenue earned in a colony's own token

When a colony earns back some of its *own* tokens as revenue, the revenue distribution process transfers them directly to the working capital, where they become part of the general fund allocation system of regular and priority funding proposals (Section 8.2).

Revenue earned in other currencies

When a colony earns Ether or other currencies as revenue, the revenue distribution system allocates some of them to be claimed as rewards. In particular, the special triggering transaction takes any such revenue that has accumulated since the last such transaction, and makes 90% available to the colony as working capital, while the remaining 10% is used to pay out rewards to users that hold both colony tokens and reputation in the colony. This split can be changed by the colony based on a colony-wide vote of tokens and reputation, requiring a majority of both tokens and reputation.

[25] A colony contract can of course receive any tokens, even those that are not whitelisted. Only payments that can be used within a colony for funding proposals and tasks payouts count as revenue.

10.2 Claiming Rewards from the Reward Pot

Rewards accumulate in the rewards pot. To trigger an actual payout to users (i.e. to make rewards claimable) a special type of proposal is made, proposing that all users should receive a payout based on the reward pot's holdings.

This reward payout proposal includes the specific currency that should be paid, and only one currency is handled at a time. In the event that the proposal is approved by vote of reputation, then all user's 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 (7). Locking is triggered by incrementing the colony's 'most recent payout' counter.

Our currency 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.

While it is of course up to the members of each individual colony to decide, it is advisable that these payout proposals should only be accepted sporadically to keep the gas costs low for the users claiming their payouts, as well as simply to not be a nuisance to the users continually finding their tokens locked.

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 (7) 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 a task, increasing their reputation).

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):

$$p_i = \left(\frac{t_i r_i}{T \times R}\right)^{\frac{1}{2}} \qquad \text{where} \quad T = \sum_{u_j \in \mathcal{C}} t_j \quad \text{and} \quad R = \sum_{u_j \in \mathcal{C}} r_j. \tag{7}$$

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 7) 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 (500000 blocks), 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 reward pot and become part of the next reward cycle.

10.3 The Revenue Model of the Colony Network

The Colony Network must be able to sustain itself. In particular, the Common Colony (which ultimately is in control of the Colony Network) maintains the contracts that underpin the network and develops new functionality for the network, the 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.

10.3.1 The Network Fee

We propose a fee that is levied on task payments made. When a user claims payment for a task they have done, and the funds leave the control of the Colony, some small fraction is paid to the network. A cartoon showing the revenue split is show in Figure 5.

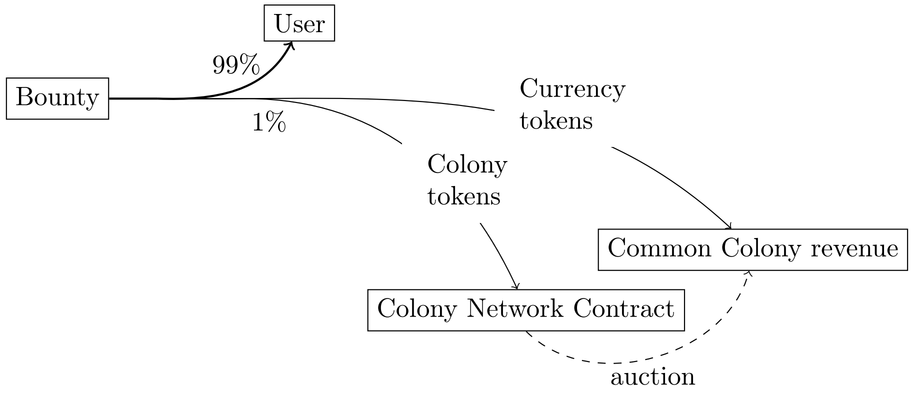

Figure 5: Summary of the revenue split upon payout for a task.

The fees thus collected are sent to either the Common Colony (if the payment was in Ether or another whitelisted token) or the Colony Network Contract (if it was in a colony's token).

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 vital in ensuring the game theoretic security of the Colony Network's reputation mining and governance processes by providing underlying value to the CLNY held by Common Colony members. Importantly, this fee is not payable to any centrally controlled entity, but rather to the Common Colony. Anybody may contribute to the Common Colony and claim a share of these fees proportional to their contribution. We believe that the benefit of being part of a secure, well supported network will ultimately 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 a task reward when a task was completed, but without sending the fee.

10.3.2 The token auction

The colony tokens collected are auctioned off by the Colony Network Contract, with the auctions denominated in Common Colony Tokens, the proceeds of which are given to the Common Colony as revenue. These auctions — one for each type of token collected — occur on a regular basis of once a month.

We believe such behaviour would be beneficial for the Common Colony Token holders (whose Common Colony Tokens gain value by having an explicit use beyond reputation mining) and the Common Colony itself (which turns the fees into Common Colony Tokens which it can then use to fund further development of the network without explicitly buying them back or creating more, diluting existing holders). It also provides an immediate mechanism of price discovery for the colony 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 Common Colony collecting a large number of different tokens that it has to manage, which would prove cumbersome and annoying for the colony.

11 Allowing more complex behaviour

The protocol described in this document is concerned with what happens on the Ethereum blockchain. Users of the network however are not expected to ever interact with the contracts manually; instead they will be using front-end applications that make the network's functionality easy to use.

In any colony application we expect a certain amount of **front-end abstraction** in which complex tools and concepts are presented for the users' convenience, and translated in the background into a sequence of contract interactions on the colony network.

Front-end abstraction lets us realise certain functionality that doesn't *seem* to be part of our protocol by combining the simple elements we have designed in inventive ways. The remainder of this section describes some such cases.

11.1 Salaried Positions

The work-for-payment model in the colony network is based around tasks, and on the surface this implies colony-worker relationships that are purely transactional. However the system is flexible enough to accommodate a wider range of employment models. One such example is a *salaried position.*

A salaried position could be realised by creating a special domain representing the position to be filled. The domain could be issued the salary through a priority funding proposal. The employee would be the only person with reputation within that domain and would be able to withdraw funds by creating and self-assigning placeholder tasks that are funded from the domain's pot. A good user interface could hide these implementation details from the users and render salaried positions differently from regular domains.

11.2 Awarding reputation for work not captured by tasks

All reputation decays over time, as described in Section 6. This prevents an eternal 'reputation aristocracy' and allows reputation to be meaningful even after major changes in the colony token's value.

Reputation is awarded when a user receives payment of colony tokens — most commonly as payout from a task, but sometimes from dispute resolution and, in the case of the Common Colony, from the reputation mining process. We can use the task payout mechanism to award users extra reputation provided 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.[26] 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 through the task system. 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 a special task that is solely designed to award 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.

[26]Finding an optimal decay rate for reputation in the network will depend on empirical data collected from early colonies.

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.

11.3 Objections by non-members

Having reputation is a prerequisite for creating an objection and triggering the dispute process. Therefore, if an outsider is hired by a colony to perform a task, they will not, on their own, be able to object to the evaluation of their work. However, a good colony frontend may allow them to create the template for an objection, effectively calling for members of the colony to support it and submit the objection 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 9.2), with the difference being that without any third party support, this 'objection' would never be processed on-chain.

11.4 Proposing an arbitrary transaction by the `Colony` contract

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. 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 high support thresholds.

Formally, proposing that a colony make an arbitrary transaction on the blockchain is no different from an objection; however the proposal is to change the value of a special variable from zero to the value of the transaction data of the proposed transaction. Such a proposal requires the entire colony to be able to vote (both token holders and reputation holders), as it concerns actions taken 'by the contract as a whole'. In the event the proposal is successful, the special variable is set. Another subsequent transaction — able to be made by anyone — is able to call a function that executes the transaction in the special variable, and resets it to zero if successful.

12 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.

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 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 objection, 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 Staking

As noted in Section 9.2, an objection 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 an objection requires 600 tokens and 1200 reputation points to activate. User A initiates the objections 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 objection, 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 objection becomes active. Now we assume that two users (D and E) oppose the objection with matching funds of 150 and 450 respectively.[27] 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.

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

[27]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

dispute). We will also assume the dispute only affected domain reputation, not skill reputation.[28] 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 nth 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 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.

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

A.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.

B Reputation decay calculation details

Reputation in any skill decays by a factor of two every 300000 blocks (roughly every three months). At each update (i.e. after every 300 blocks), the new decayed value (u_{new}) is calculated by

$$u_{\text{new}} = u_{\text{old}} \times \exp\left(-\frac{\ln 2}{1000}\right) = u_{\text{old}} \times \exp\left(-k\right).$$

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.[29] 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 5 years for 15 second blocks. 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.

[29] 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 $0.5N$.

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