Liquid Democracy
– A concrete proposal –

Fabrizio Romano Genovese
Quantum Group
University of Oxford
fabrizio.genovese@cs.ox.ac.uk

After some brainstorming sessions in which the main ideas have been laid down, we start drafting how a hypothetical Liquid Democracy implementation for Swarm could look in technical terms. We will use a typed approach, that should in principle be compatible with the concepts behind StateBox.

1 Introduction

2 Hierarchy Structure

We start laying down how the *social structure behind Swarm* works. There are four main entities we are interested in defining. These are

Swarm: This must be thought of as a financial market, since Swarm provides all the necessary infrastructure to host projects.

Project: An investment project on the Swarm platform. Projects must be thought of as the equivalent of companies. Creating a new project on Swarm is like putting a company on the stock market, allowing investors to put money into it.

Collective: Represents a group of people. Collectives must be thought of as group of investors sharing some interests. Collectives can also be defined locally in a project, and are useful to denote those people that actually make the project work. For instance, a collective called "marketing" in a project called "real estate" could include all those investors that are actually active in doing publicity to the project itself. Local collectives can only be defined within a project.

Person: The basic block of the Liquid Democracy environment. A Person must be thought of as an investor moving in the financial market (the Swarm platform). Investors can join or leave projects (the equivalent of buying/selling company stocks). A person can be part of multiple collectives and multiple projects.

When we want these social structures to interact, we can use their names as programming language types, as follows:

Example 1. Suppose, for instance, that there are two Swarm projects, call them X and Y. Some investors of X and Y want these two projects to merge into a unique project Z, and hence start drafting a series of smart contracts using StateBox that "entangle" all the regulation of X and Y providing a new set of regulations for Z. This can be seen as a procedure having type $\textit{merge} : \textbf{Project} \times \textbf{Project} \to \textbf{Project}$, meaning that it takes two projects in input and creates a new project as output.

This is particularly useful because this merging proposal will have to be voted by the investors of both X and Y to be approved. Using this typed mindset we can draft an automatic procedure to ask the relevant people involved to vote. In pseudocode, this may look like

Submitted to:
Swarm group

```
for all P: Project in domain(merge)
        for all u: Person in P: Project
                Forward.Vote(merge, u)
        end
end
```

Where *Forward.Vote* is some function that automatically asks user *u* to vote for the approval of the contract *merge*. We can implement our contract drafting tool to continuously scan for all the projects, collectives and people named in the contract, notifying the contract writer about who is going to receive the voting request in real time. When the proposal is submitted, an algorythm like the one above can be triggered to run automatically.

Example 2. Suppose that user *u* wants to join a collective *C*. The collective *C* has to approve this join request. Again, this amounts to have an algorithm

```
for all a: Person in C: Collective
        Forward.Vote(join(u,C), a)
end
```

That automatically asks the people in the collective if they want *u* in or not when *u* submits his join request.

Example 3. Suppose that a new change of rules for Swarm is proposed. In this case, our algorythm will look like

```
for all a: Person in SWARM
Forward.Vote(NewRules, a)
end
```

We can see here that **SWARM** is not considered as a variable type but as a *constant*. This makes sense because there is only one Swarm platform, and this constant is used every time there is some decision to be taken that involves all the users present.

The hierarchy structure, up to now, looks like this:

$$\textbf{SWARM} > \textbf{Project} > \textbf{Collective} > \textbf{Person}$$

Meaning that, for instance, some procedure that acts on a project *P* is relevant for all the collectives and people that are part of *P*.

3 Proposals and kinds

Now that we drafted these different social entities in Swarm, we have to provide the means to make them interact together. Interactions are mainly made by means of proposals, that are nothing but smart contracts. Proposals are classified by *kinds*, that are tags denoting the type of proposal considered. To each proposal can only be assigned one kind, to make the categorizing of smart contracts easier.

Sufficient precision is attained allowing kinds to be nestable together in a fashion similar to a filesystem folder structure on any PC, meaning that we should be able to define relevant subkinds if needed.

Kinds are Swarm, project or collective specific, meaning that within every of such social structures a different set of kinds can be defined. For instance, a collective *C* particularly focused on the environmental repercussions of a project *P* may create a specific kind $\textbf{environment}_C$ (the subscript means that this kind is specific for *C*) to classify all the proposals that deal with environmental issues in *P*.

Example 4. Kinds become especially useful to make people's life on Swarm easier. For instance, say that a user u is part of some collective C, but doesn't really care about all the fuss regarding the collective rules (requisites for joining or leaving the collective, for instance). In this case the user may want to delegate someone else to vote for him when changes of rules are proposed, or he may want to automatically abstain when any such proposal is submitted. This becomes easier endorsing some other user u' to vote for him every time a contract having kind $\textbf{rules}_C$ is proposed. In this way, vote endorsement becomes a very customizable process. As we said, kinds are organized as a filesystem folder structure, and this allows us to formalize endorsing conditions based on kinds by means of simple propositional logic statements.

For instance suppose that the kind $\textbf{rules}_C$ has two defined subkinds $\textbf{rules/voting.procedures}_C$ and $\textbf{rules/admission.criteria}_C$. A user u endorsing some other user v for **rules** automatically endorses v for $\textbf{rules/voting.procedures}_C$ and $\textbf{rules/admission.criteria}_C$ too, but u can refine this endorsing v for $\textbf{rules}_C \wedge \neg\textbf{rules/voting.procedures}_C$. In this case any proposal regarding changes of rules is endorsed to v, with the exception of proposals regarding changes of voting procedures specifically, that are instead brought to the attention of u (see Figure 1).

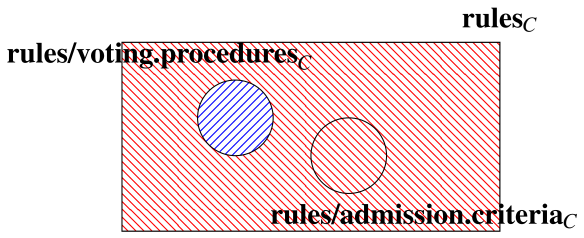

Figure 1: The expression $\textbf{rules}_C \wedge \neg\textbf{rules/voting.procedures}_C$ endorses the red area to v and the blue area to u.

3.1 The kind "rules" and the rules contract

If kinds can be customized to serve some needs, it makes sense for Swarm to propose a basic set of kinds to play with. In particular, the kind **rules** is maybe the most important one. In fact, every time a project or a collective is created, a "rules contract" must be configured, containing all the basic information about the administrative structure of the collective/project one is creating (admission criteria, minimum investment to enter, needed quorum to make a vote valid, minimum percentage of backers to forward a proposal to voting stage, ...).

In particular, the rules contract is modular and relies on other subcontracts, each managing a different aspect of the administrative structure of the social group being defined. These subsections are tagged with subkinds, such as the kinds $\textbf{rules/voting.procedures}_C$ and $\textbf{rules/admission.criteria}_C$ in Example 4.

As we briefly said, there are three types of rules contracts, depending on the social structure they refer to.

- The Swarm rules contract, drafted by the Swarm Team itself, and regulating the general social administration of the whole platform. Changes to this contract via liquid democracy are possible, but they should require high thresholds to be passed since said modifications would influence the behavior of the platform as a whole.

  Moreover, any change proposal to the Swarm rules contract should be carefully audited by the Swarm team that, in exceptional circumstances, should have the right to have it vetoed (for instance, if the contract implies a non-democratic change of behavior on the platform or if it results in some law infringement that may endanger Swarm as a whole).

When a change of the Swarm rules is proposed, Swarm takes it into consideration and produces an audit document that gets forwarded to all the platform users along with the proposed changes. The audit document contains an in-dept analysis performed by the Swarm team about what this change of rules should imply. Similarly, if a proposal is vetoed at the end of the audit, a document explaining to users why they will not be able to vote for the proposal should be produced, with a detailed analysis about why this apparent "violation of the democratic rules" has been necessary.

- For every project, there is a project rules contract. This contains then name of the project, a brief description of what the project wants to accomplish and the basic settings of all the relevant parameters for the project administration. Since Swarm is responsible for the projects it hosts, they must be audited as well to check if they don't violate Swarm's basic regulations. Again, a project may be blocked by the Swarm team in exceptional circumstances (for instance, if a Project is nothing more than a Ponzi Scheme), and an audit document should be produced.
- For every collective, there is a collective rules contract. This states what that collective is about, which are the requisites to join or leave the collective and the like. Since collectives are nothing more than congregations of users, collective rules and purposes should not be audited by Swarm (if Swarm goes well I'd expect collectives to be A LOT, so auditing any single collective rule contract would put a huge stress on the Swarm legal team).

You can think of rules contracts as the analogous of .conf files that regulate the behavior of some application (in our case a project, a collective or Swarm itself).

A very important thing to avoid a nightmarish mess is that all the parameters set in a rules contract are *proprietary*, meaning that these parameters cannot be modified by other contracts. This avoids conflicts between rules contracts and proposals of any other kind that may result in unpredictable behavior.

In particular, this means that the only way to change the administration rules for a social entity is to modify the rules contract itself, making some attack scenarios ineffective. This not being the case, one could perform an attack moving forward a proposal that overwrites some parameters of the rules contract, disguising it for some contract of a less important kind.

4 Methods

Methods have to be thought of as basic actions between proposals and social entities. More complicated methods can be created composing the basic ones, and some new (local) methods can be created along with a project.

4.1 Methods for Person

join(C : **Collective**): This method is called when a person p asks to join a collective C. First, it checks if p satisfies the needed requisites to join the collective (for instance, the collective rules may be such as users with less than a predetermined voting weight in a project have applications rejected by default), defined in the collective rules contract. If these requisites check out, it forwards an approval request to the collective. This approval request is ruled out according to the rules contract (it could be a vote, the collective may have a designated moderator, ...). If the request results into an acceptance, a smart contract denoting p as part of the collective C is produced. Otherwise, a smart contract reporting the failed join attempt by p is produced. This last case may be useful to enforce some other collective rules, such as forbidding to ask to join again if a predetermined amount of time has not passed.

leave(C : **Collective**): Checks if p is part of the collective C. If p is, it checks if according to the rules contract p has the needed requisites to leave (it may be that, for instance, the collective does not allow a member to leave if some conditions are not met). If p has, then it produces a smart contract registering that p has left C or, even better, it triggers some condition in the smart contract produced when p joined C deactivating it.

change(contract:**kind**,C : **Collective** | P : **Project** | **SWARM**): Any user can propose to change an existing contract defined by a collective, a project or Swarm itself. For instance, an user may propose a change of rules for the collective he is in (for instance, change the requirements to join it), for a project he is involved with, or for the Swarm platform itself. According to the relevant rules contract, a proposal may have to be endorsed by other users or collectives, meaning that there is a predefined window of time in which other people have to back up the proposal. If enough endorsement is reached, then the relevant crowd to which the contract is proposed will vote for it as defined by the relevant rules. If the vote passes, the contract becomes active, otherwise it doesn't.

This method is particularly useful to propose changes in the rules contract regulating the behavior of some social entity.

propose(contract:**kind**,C : **Collective** | P : **Project** | **SWARM**): Similar to the previous one, but used to propose a new contract that does not need the modification of some existing ones.

endorse(contract:**kind**): This method registers the fact that a given user has endorsed a proposal.

deendorse(contract:**kind**): Opposite to *endorse*, retires the endorsement of a proposal that had been endorsed previously.

invest(P:**Project**, t:**tokens**): If the necessary conditions in the rules contract of P are met (minimum amount of the investment and so on), invest a t amount of money in P.

cashback(p:**Project**): The users closes its investment in a project P, and receives his amount of tokens or other good as prescribed by the project itself.