Liquid Democracy Voting System for the Swarm Platform

Fabrizio Romano Genovese
Swarm Team
Quantum Group
University of Oxford
fabrizio@swarm.fund

Jelle Herold
Swarm Team
jelle@swarm.fund

Here we highlight how the voting system is going to work. We will start briefly reviewing existent solutions, we will proceed sketching the general idea for the voting system on the Swarm Platform and, finally, we will dive into details producing a working algorithm.

1 Pre-existent solutions

The need to define a voting system is not a new problem for the blockchain community. For instance, if some altcoin is going to hard fork, the developing team will often ask the user base to express a preference about which branch to adopt. It is obvious, then, that in a situation like this some kind of procedure has to be devised to allow people to express this preference, that is, to allow people to vote.

One of the most common ways to deal with this problem is by issuing some voting tokens and creating two addresses that stand for *yes* and *no*. Every user receives a number of tokens equal to the number of coins owned and sends them to one of these two addresses, expressing a preference.

Albeit very simple to implement, this method has many fundamental flaws that make it not suitable for a system that has democracy – and hence voting – at its center:

1. First of all, a voting procedure like the one highlighted above is very similar to majority voting and, as such, it is cursed by many downsides. One of the biggest achievement of social choice theory is proving that nearly all the voting systems that people intuitively consider "fair" are not fair at all, meaning that they are prone to any sort of manipulation or can even be dictatorial [1–3]. Voting with tokens as stated above is no exception.

2. In the very same moment someone votes, the tokens are transferred to the address expressing a preference, and can be publicly seen. This means that a sufficiently skilled user is able to see in real time who is winning the vote. This exposes the system because gives away data that can be employed for any kind of manipulation.

3. "Real" tokens can be exchanged while the voting ones remain in one's wallet. This means that voting power is not really related to the quantity of money one owns: In a transaction happening after the voting tokens are issued, the person receiving the real tokens has no voting power whatsoever while, on the other hand, the other person retains the voting power without having real stake.

   For instance, someone with a lot of real tokens could sell them all, and then use the voting tokens he retained to vote for a very bad decision for the platform. At this point the tokens he just sold will be depreciated because of this bad decision and the person will be able to buy them back at a fraction of the price, making a profit.

   This means that a very powerful user on the platform could exploit the democratic infrastructure to make direct profits.

Submitted to:
Swarm community

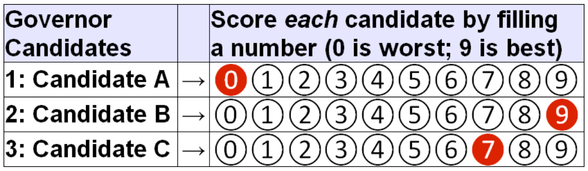
4. There is no reputation system. This means that a user voting power is only proportional to the amount of coins owned. A lot of important parameters (how active that user is on the platform, how much money one moves every month, how much that user helped with proposals, ...) are completely disregarded.

5. Vote delegation is messy, or even not impossible.

For these reasons we deemed necessary to steer away from all the pre-existing solutions and design an ad-hoc voting environment that is reasonably fair (according to social choice theory) and suitable to be used on a daily basis.

2 Range Voting

The voting method we choose for the platform is *range voting*. Range voting is one of the fairest voting systems available. In particular, being it a *cardinal* and not an *ordinal*[1] voting system, many of the mathematical results about unfairness and dictatorship, such as [1, 3], do not apply.



Governor Candidates		**Score *each* candidate by filling a number (0 is worst; 9 is best)**
1: Candidate A	→	⓿①②③④⑤⑥⑦⑧⑨
2: Candidate B	→	⓪①②③④⑤⑥⑦⑧❾
3: Candidate C	→	⓪①②③④⑤⑥❼⑧⑨

Figure 1: Range voting ballot [source: Wikipedia]

In range voting, each voter can rank each choice giving a score, for instance from 0 to 9, where 0 stands for "I absolutely don't like this choice" and 9 stands for "I absolutely love it" (see for instance Figure1). When the voting closes, the scores for each choice are summed, and the choice with more points is the one that wins.

There are many groups that actively advocate for range voting to be adopted in electoral laws all around the world, and in addition to this range voting is also considerd one of the best voting procedure available in company administration (see, for instance [4, 5]).

To conclude, in choosing the right voting procedure for the Swarm platform we consulted decades of research and study on voting systems to find the best alternative available, and we are fairly convinced that range voting is the one.

2.1 Reputation

Taking into account "modifiers of the voting power" is also very easy when one adopts range voting. Let's suppose, for instance, that we want to have a system where the power of a voter is proportional to the number of coins he owns. If voter A owns n coins, then it will be sufficient to multiply by n all the scores that A gives to each candidate. So, for instance, if A owns 10 coins and ranks a candidate with a score of 5, that candidate will receive $10 * 5 = 50$ points.

To be precise, we want to take into account more parameters than just the quantity of money a voter has to determine the voting power, but now we do not need any new concept to do this. For each user A, we will just calculate a multiplier, denoted with ρ_A and called *reputation of* A, considering all the parameters that we deem relevant. Again, everything we will need to do to make A's votes proportional to his reputation is to multiply all the scores that A assigns by ρ_A.

[1] These are technical terms of voting theory. You can safely ignore them if you don't know what they mean.

2.2 Drafts

Clearly, it may happen that a vote leads to a draft. In range voting this means that there are two or more candidates at the top of the points list having the same score. For instance, imagine we have to elect three people for a company board and there are ten candidates, with their ranking in terms of points as follows:

Candidate	**Points**
Candidate 7	247
Candidate 2	133
Candidate 5	102
Candidate 8	102
Candidate 10	84
Candidate 1	58
Candidate 6	37
Candidate 3	29
Candidate 9	14
Candidate 4	3

Since we are selecting only three members for the board, we have to pick the three best performing candidates. This obviously includes candidates 7 and 2, but then we are stucked because candidate 5 and 8 have the same amount of points. So who who among the two do we choose?

In classic range voting, at this point, another voting round is held. We discussed at length about implementing such a system for Swarm, and we decided that it wasn't a good idea, first because it puts much more stress and responsibility on users that will have to vote two times, and second because it would make the actual code much more complicated and difficult to audit. So we went for a system in which no second round is needed, as follows:

- If there is an ex-aequo situation, we will prefer the candidate with the highest reputation.
- If this doesn't solve the problem (there are multiple winning candidates with the same reputation), then the choice is pseudorandom.

3 The voting algorithm – Intuitive explanation

3.0.1 Desiderata

Now that we know what we want, we have to implement it. There are some fundamental desiderata we require:

- The voting should be transparent. This means that all the relevant information should be stored on the blockchain, and every user should be able to calculate the vote outcome independently, as well as to check that is preference was correctly submitted.
- The voting should be encrypted. As we said in Section 1, one problem of voting on the blockchain is that people can know who is winning before the voting window closes. To avoid such a situation, every vote should be encrypted and readable only when the voting window closes.
- Being an active member of the Swarm democracy should increase a member reputation. Swarm values participation, and it makes sense that "the more you vote, the more your vote becomes powerful". This is also a way to value the experience and time spent on the platform.

- The reputation should also be proportional to the amount of Swarm tokens owned. More specifically, if one owns zero Swarm tokens then his voting power should be zero (meaning that has no influence whatsoever).
- Delegation must be possible. The word "delegation" is actually the reason why we talk about *liquid democracy* and not just about *deocracy*. The idea here is that any Swarm member can delegate some other member to vote in his place. This is particularly useful when someone is asked to take a decision, but does not feel to have enough experience or understanding to vote. Instead of wasting his preference, in such situation a user can delegate someone else to vote in his place. Imagine, for instance, that Bob is a Swarm user. Bob has to vote to elect members of the administration Board. He reads all the proposals and what every member wants to do, but it is all incredibly technical and out of his skill set. Bob, on the other hand, knows that Alice, a professional trader which he trusts, is on Swarm too. What Bob can do is delegating Alice to vote for him, so that his preference will not be wasted.

3.0.2 User experience

From the point of view of a Swarm user, voting should be simple. The idea is that you just visit a webpage, and use the ethereum address you have your tokens on as a login. At this point, you are able to see your reputation and all the open voting windows. To vote, it is sufficient to rank each candidate on the webpage and sign your preference with your private key. At this point, the preference is submitted. To improve easy of use, we are working to make such webpage compatible with Nano Ledger S and Trezor, so that signing your petition will just require a couple of clicks.

When the voting window closes and the outcome is calculated, the user should will be able to see it on the very same web page.

3.0.3 Voting phases

There are two different kinds of votes on Swarm. The first kind is when the user base has to express a preference. The board may ask, for instance, "Do you prefer option A or option B?" and in this case everything that is needed is voting.

The second kind is when the vote is about electing people to occupy a position, for instance when the user base has to elect members for the Swarm board. In this case, we have to implement a "campaign phase" before the vote itself, where people are actually able to propose themselves for the position.

What we want then is as follows:

- A phase I (optional, depending on the kind of vote) to manage candidatures.
- A phase II that is the vote itself.

3.1 The Voting Algorithm

3.1.1 Phase I

During phase I (if the kind of vote comprises it), users will have a time window to propose themselves for the election. This is what happens:

- Phase I time window opens. A smart contract called **CANDIDATES** is created. It will contain information about all the candidates running for the election.

- Any user, logging in with his ethereum address, can see this on the webpage. He can, if he wishes to, decide to run for the election. To do this the user has to fill a couple of fields: One is a “bio” field where the user says something about himself. The other is a “letter of intent” where the user states why he wants to run for the election. The candidature can be submitted signing it with the user private key.
- Every time a candidature is submitted, the **CANDIDATES** contract checks that no other candidature has been already submitted by that ethereum address. This is because, obviously, one user can run only once for the same vote, and also to prevent spamming. If this is the case, the candidate is not added to **CANDIDATES** and the submission is rejected with an error message. Otherwise, it is.
- When the Phase I time window closes, it is not possible to add further candidates to **CANDIDATES**.

3.1.2 Phase II

During Phase II the actual voting happens, as follows:

- Phase II voting window opens. A smart contract called **VOTE** is created. It will contain information about all the candidates running for the election.
- A Swarm trusted party[2] creates a couple of public/private keys. The private key is kept secret, while the public key is pushed into the **VOTE** contract.
- Every user can see the open vote on the web page, with all the set of choices/candidates to rate. If Phase I was present, the set of candidates is automatically derived from the **CANDIDATES** smart contract.
- The user can vote as follows: He can rank each candidate giving him a score (from -3 to 3, the default being 0). He can moreover specify an ethereum address to delegate his vote to someone else.

  Note that these two things are not mutually excluding: If for some reason the delegation shouldn’t work (for instance if your delegate forgets to vote), then the algorithm will immediately fall back on the scores the user gave. It makes sense then that you express a vote even when you specify a delegate to vote for you.
- When the user pushes the “submit” button, all his voting preferences and delegate addresses are encrypted using the public key found in **VOTE**. The user is then prompted to sign his submission with his private key, as usual.
- When the Phase II voting window closes, it is not possible to push more voting preferences to **VOTE**.

3.1.3 Outcome

The outcome of the vote is then calculated, as follows:

- Shortly after the Phase voting window closes, the Swarm trusted party appends the secret key that was generated in the previous phase to **VOTE**. At this point, all the information contained in **VOTE** becomes accessible to everyone.

[2] The Estonian non-profit, for instance

- **VOTE** is decripted, and the outcome is calculated offline. This is still trustable since every user can repeat this procedure on his own computer (this functionality will be provided on the web page, and every user will be able to check that the outcome was calculated correctly with client-side).
- First of all, the algorithm check for repeated votes, that is, users that submitted their vote more than once. This being the case, the algorithm just cancels all the submitted votes but the most recent one.
- After this, the algorithm checks that no vote comes from the Swarm foundation address, and this being the case, cancels that voting preference from the contract. This is for ethical reasons. In fact, Swarm founders could use the Swarm foundation to express a vote. Since the Swarm foundation holds 33% of the total Swarm tokens, its would be so high to turn Swarm into a dictatorship. Swarm team members and partners are obviously still able to vote using their personal wallets, as any other member.
- The algorithm pushes information into the **HISTORY** contract, that registers how many times a user expressed a vote. This paramenter is used to calculate the reputation of a user, since it representative of the user activity and contribution to the platform
- The algorithm solves delegation errors. In fact, it may be that a user called Bob delegated Alice to vote for him, but Alice forgot to vote. It could also be that Bob delegated Alice, Alice delegated Charlie and Charlie delegated Bob. As you can see, this is a loop and it is difficult to say who is deciding for who. The algorithm addresses this situation falling back to the personal preferences. For instance, if Bob delegated Alice and Alice forgot to vote, then the algorithm uses the scores that Bob gave. In the worst case, Bob left everything as it was, meaning that he had scored 0 every candidate.
- The algorithm calculates the reputation of every voter. To do this, it checks how many tokens the voter holds, checks if there are more reputation modifiers referring to that user in the **HISTORY** contract, and puts everything together. At this point two things can happen:
  - If the user voted for himself, the algorithm multiplies the scores that the user gave by his reputation.
  - If the user delegated someone else, the algorithm transfers the user reputation to that someone to calculate the scores.
- Finally, the algorithm calculates the winning choice/candidate summing all the scores together. As we pointed out in Section 2, if there are ex-aequo candidates then the algorithm checks who has the bigger reputation to calculate the winner[3]. If this is not enough, the algorithm chooses between the winning alternatives pseudorandomly.

  Should be this the case, the user will receive a notification warning him: Since the final choice is pseudorandom, clearly the outcome calculated in locale may be different from the outcome calculated by Swarm team.
- The Swarm team publishes the result of the vote, that is displayed in the user web page. As we said, the user can independently check that the outcome has been calculated correctly.

[3] Note that if the voting is about choices and not candidates, this is not possible since choices do not correspond to single users and hence have no reputation. In this case this step is just skipped

4 The Algorithm, detailed explanation

Note: This section is technical and requires some basic familiarity with mathematics and coding. The following algorithms are not even written as pseudocode, and use evil instructions like “goto”. This choice has been made to improve readability and give an idea of what are the actions that the algorithm has to perform. The real modules used will be most likely coded using functional programming. Please refer to our github repo to check the actual source code.

4.1 Definitions

Definition 1 *A* string *s is a finite sequence of characters or digits $s_1 s_2 \cdots s_n$. If s is a string, we will denote with $s[n]$ the* n-th *character of s, and with $|s|$ the* length *of s. We will abuse this notation and we will often use* strings of strings*, that is, some string s such that $s[n]$, for some n, is a string too. In this case, we will denote with $s[n][m]$ the m-th entry of the n-th entry of s.*

If s is a string of numbers and k is a number as well, then $k * s$ is the string $(ks[1], \dots, ks[|s|])$.

Definition 2 *We will denote with $\mathscr{C}$ the* ordered *set of candidates or choices that can be voted for in a given vote. We will denote with $\mathfrak{R}$ the set of possible scores a user can assign to each candidate. For instance, if we have three possible choices A, B, C and the range of possible scores goes from -3 to 3, then $\mathscr{C} := \{A, B, C\}$ and $\mathfrak{R} := \{-3, -2, -1, 0, 1, 2, 3\}$. We will moreover denote with $|\mathscr{C}|$ and $|\mathfrak{R}|$ the number of elements that these two sets have. In our example, $|\mathscr{C}| = 3$ and $|\mathfrak{R}| = 7$.*

Definition 3 *We can express the voting of a user as a string $(a, b, c, \dots)$, where $a, b, c \in \mathfrak{R}$. a represents the rating given to candidate A, b the rating given to candidate B and c the rating given to candidate C and so on. This string is called* preference string*, and its length is clearly equal to $|\mathscr{C}|$.*

Definition 4 *We denote an user A delegating his/her vote to a user B with $A \to B$.*

4.2 Contracts

Here we highlight the fundamental pieces of information that will be specified in the three contracts we mentioned in Section 3.1.

4.2.1 The CANDIDATES contract

In every **CANDIDATES** contract, there will be a reference to the correspondent **VOTE** contract, on which the preferences of the users will be registered. This piece of information is specified when the contract is created and allows the user to reconstruct where he has to send his preferences once the voting window opens.

The data in the **CANDIDATES** contract is list of strings that look like this:

$$c := (\text{candidate ethereum address} \mid \text{hash of the bio} \mid \text{hash of the statement of purpose})$$

Every time some candidate pushes his candidature string c to the chain, it is appended to this list.

CANDIDATES does not do any special computation, aside of the following:

- If c is pushed to **CANDIDATES**, check the status of the time window.
- If the time window is closed, reject c.

- Check that $c[1]$ corresponds to the same address that is trying to append the string. If not, someone is running with a fraudulent address, and c is rejected.
- Otherwise, check if there is some c' such that $c'[1] = c[1]$.
- If this is the case, c' is deleted and c is added at the bottom of the contract (this allows a candidate to modify his statement of purpose or bio until the window stays open).

This means that we avoid double candidatures, make sure that the candidatures are not fraudulent and allow any candidate to modify his bio/statement of purpose until the time window stays open.

The $c[2]$ and $c[3]$ fields of the string guarantee that the candidate bio and statement of purpose are not altered. They can now be stored on an independent server minimizing the space consumed and still guaranteeing that the platform system stays trusted.

4.2.2 The CHOICES contract

As we mentioned, there are two kind of votes on the Swarm platform. The first kind allows for a Phase I where people can propose themselves as candidates, while the second does not, giving people the possibility to express a preference in a set of pre-determined choices (this is the case for a typical “yes/no” voting, for instance).

The **CANDIDATES** contract addresses the voting of the first kind, allowing users to add their candidature in it. The **CHOICES** contract is used for the votes of the second kind, and contains the set of choices the users will be able to vote.

Note that since a vote can have or not have a Phase I, these two contract are *mutually eclusive*: A voting relies on a **CANDIDATES** contract or on a **CHOICES** contract, but not on both at the same time.

As in the previous case, in every **CHOICES** contract there will be a reference to the correspondent **VOTE** contract, on which the preferences of the users will be registered. This piece of information is specified when the contract is created and allows the user to reconstruct where he has to send his preferences once the voting window opens.

The data in the **CHOICES** contract is a list of strings that look like this:

$$c := (\text{choice} \mid \text{hash of the choice description})$$

$c[1]$ represents the kind of choice we are going to vote (“yes” or “no”, for instance), while $c[2]$ is the hash of a description of what that choice means (for instance “Voting this you are approving the following board resolution: ...”).

This contract doesn’t perform any computation whatsoever.

The $c[2]$ field of the string guarantees that the description of a given choice can be stored on an independent server minimizing the space consumed and still guaranteeing that the platform system stays trusted.

4.2.3 The VOTE contract

The data in the **VOTE** contract is of different kinds, as follows:

Secret key: This is specified in the contract when it is created. It is a RSA (or similar) secret key, denoted with *pub* and generated by a Swarm trusted party.

Instructions: These are again specified when the contract is created, and are used to calculate the voting outcome. They consist of two parameters: an integer, representing the number of choices that have to be declared as winners (for instance, “the three best candidates have to be picked”), and the range of possible values used for range voting, denoted with $\mathfrak{R}$ (for instance $\{-3,-2,-1,0,1,2,3\}$[4]).

Encrypted block: This is a list of encrypted strings, that look like this when decrypted:

$$v := (\text{user address} \mid \text{preference string} \mid \text{delegation address})$$

Such strings are called *voting strings*, and have following specification:

user wallet address is the ethereum address of the voter;

preference string is a string defined as in Definition 3, where $\mathscr{C}$ (see Definition 2) is the set of all the addresses in **CANDIDATES**. To be specific, the first entry of the voting string refers to the first candidate in **CANDIDATES**, the second to the second candidate in **CANDIDATES** and so on.

delegation address is an eth address.

When the voting window opens, every user can encrypt his voting string with *pub* and add it to the encrypted block. Note that since these stings will be added in an encrypted form, their integrity can’t be checked by the contract itself, and will have to be verified offline (see module 4.3.7).

When the voting window closes, the users will not be able to add more strings to the contract. The list of all the encrypted strings as above will be called *encrypted block*.

Secret key: This is added to the contract by the same trusted party that generated *pub* after the voting window closes. This secret key, denoted with *sec*, allows to decrypt the encripted block.

Plain block: This is just the encrypted block itself, decrypted with *sec* and purged of incorrect specifications and delegation loops, according to module **??**. It will be added to the contract by a Swarm tusted party after the voting window closes.

Winner: The list of winner(s) of the election, again added to the contract by a Swarm trusted party when the voting window closes. This is just a list of numbers bigger or equal to 1, referencing to the **CANDIDATES** or **CHOICES** contract. If this list is, for instance, $(3,6,2)$, this means that the 3^{rd}, the 6^{th} and the 2^{nd} candidate (choice) listed on **CANDIDATES** (**CHOICES**) are the ones who won.

To get the corresponding winning candidate (choice) value it is sufficient to fetch the first value of the string in the contract. For instance, in the previous example, if our contract is of type **CANDIDATES** then the winning ethereum addresses are “3^{rd} listed string”[1], “6^{th} listed string”[1] and “2^{nd} listed string”[1].

This contract is very simple. It is just a big database, with an encrypted and a plain text part. The only real computation that the database does is checking if the user can or cannot add a voting string to it verifying if the voting window is open or not and, in a latter stage, authorizing only the Swarm trusted party to add information to it.

Note: On the user web page, before any preference is specified, vote strings and delegation addresses are set to the defaults values of 0 for $v[3]$ and $(0,0,\dots,0)$ for $v[2]$. In this way, if the user submits his vote without specifying any preference, its vote will still be submitted as well formatted.

Note: The reason why the plain block is added to the contract is to make the outcome verification process faster. On the user web page, in fact, the user will have the possibility to check independently the outcome of a vote. To do this, some code will be run client-side. The user will be able to select between three modes:

[4] By definition, we require 0 to be always present in the range, and the range to be symmetric, meaning that the values can go from $-x$ to x, for some integer x.

- A "thorough" mode that obtains the plain block applying module 4.3.4, verifies that this coincides with the plain block added to **VOTE** by the Swarm trusted party and then calculates the vote outcome, that is again compared with the list of winners attached to **VOTE**. With this option nearly every step of the calculation is executed locally.
- A "fast" option that skips the first part, and just uses the plain block provided on the **VOTE** contract to calculate the outcome. This will undoubtably be faster, since all the decryption and purging will be skipped.
- A "paranoid" option that works as the "thorough", with the exception that now module 4.3.13 does not use the plain block of previous contracts to calculate reputations, but invokes module 4.3.4 every time. This will require **a lot** of decryption and purging, and will probably be very very slow to run. On the other hand, here **every** step of the calculation is executed locally, and nothing is given for granted.

4.2.4 The HISTORY contract

The data in the **HISTORY** contract is just a list of references to all the **CANDIDATES** and **CHOICES** contracts produced.

Every time a new voting is open, a new **CANDIDATES** (only if the vote has a phase I) or **CHOICES** contract is produced. A Swarm trusted party adds a reference to these contracts to **HISTORY**.

This contract is useful for two reasons:

- Since both **CANDIDATES** and **CHOICES** contracts always contain a reference to their corresponding **VOTE** contract, every user is able to follow the references to see how many votes and candidatures an address submitted. This information is used to calculate the reputation of a user by module 4.3.13.
- The **HISTORY** contract is also used by the user web page to fetch the history of all the votes performed by the platform. Following the references any client can display which votes have been performed and for which ones Phase I and Phase II time windows are still open.

4.3 Modules

Here we formalize some algorithms that will perform various operations to get a vote outcome, such as getting rid of delegation loops. These algorithms will be invoked to perform the voting, both locally and by the Swarm trusted party.

4.3.1 Display votes

This algorithm is used to display all the votes that happened on the platform, including the open ones. It is used in particular in the front end environment.

- Scroll the **HISTORY** contract and follow the references, building a history of voting contracts. Every reference also gives the relevant time window for a given contract, that then can be displayed as open or closed on the user web page.
- When the user clicks on a given vote, fetch from an independent server all the relevant information. Verify that the hashes are correct on the correspondent **CANDIDATES** or **CHOICES** contract.

- At this point, depending on the time window, a user will be able to vote, run for an election or verify the outcome of a vote. Again, Following the reference on the corresponding **CANDIDATES** and **CHOICES** contracts, the user will be able to know which **VOTE** contract has to be consulted to manipulate this information.
- For instance, if a user wants to verify the outcome of a vote, he follows the unique identifier in the corresponding **CANDIDATES** (**CHOICES**) contract and applies module 4.3.5 to this. If the final result corresponds to the winner(s) displayed in **VOTE**, well. Otherwise, display an error.

4.3.2 Submit candidature

This is the algorithm to submit a candidature for a vote.

- Check if the time window allows you to submit your candidature to **CANDIDATES**. If not, return with an error.
- Calculate the hash of the "bio" form, call it *bio*.
- Calculate the hash of the "statement of purpose" form, call it *statement*.
- Upload these two forms on the independent server.
- Form the string $c :=$ (your ethwallet | *bio* | *statement*)
- Add it to **CANDIDATES** signing with your private key.

4.3.3 Submit vote

This is the algorithm to submit a vote.

- Check if the time window allows you to submit your vote to **VOTE**. If not, return with an error.
- Form the string $v :=$ (your ethwallet|preference string | delegation address)
- Fetch the public key *pub* from **VOTE**.
- Encrypt v with *pub*.
- Add it to **VOTE** signing with your private key.

4.3.4 Get decrypted block

This is the algorithm to decrypt the encrypted block in a **VOTE** contract and purge it by any kind of error.

- Use the secret key in **VOTE** to decrypt the encrypted block. Get a list of decrypted entries called **VOTE**' back. (module 4.3.6).
- Check that every string in **VOTE**' is well-formed (module 4.3.7).
- Ban the Swarm foundation address (module 4.3.8).
- Delete repeated votes (module 4.3.9).
- Get rid of hanging delegations (module 4.3.10).
- Get rid of delegation loops (module 4.3.11).
- Return **VOTE**'.

4.3.5 Check vote outcome

This module verifies that the outcome of a vote is the one provided by the Swarm trusted party.

First of all, fetch the number of winner(s) from the instructions of the **VOTE** contract. Fetch the number of candidates (choices) from the **CANDIDATES** (**CHOICES**) contract, call it $|\mathscr{C}|$. If $|\mathscr{C}| \leq w$, declare everyone a winner. Otherwise:

Fast mode: This is the fastest mode available.

- Read the plain block from **VOTE**. Copy it to a local file **VOTE**'.
- Assign reputation to every user (module 4.3.12, **fast** option enabled).
- Calculate the winner(s) (module 4.3.14).

Thorough mode: This mode will be slower, but the only thing that won't be entirely checked are user reputations.

- Apply module 4.3.4 to **VOTE**.
- Read the plain block from **VOTE**. Check that the two coincide. If not, display an error.
- Assign reputation to every user (module 4.3.12, **fast** option enabled).
- Calculate the winner(s) (module 4.3.14).

Paranoid mode: This mode will be painfully slow, but will check everything locally.

- Apply module 4.3.4 to **VOTE**.
- Read the plain block from **VOTE**. Check that the two coincide. If not, display an error.
- Assign reputation to every user (module 4.3.12, **slow** option enabled. This will take time).
- Calculate the winner(s) (module 4.3.14).

Finally, check that the output result coincides with the one in **VOTE** (module 4.3.15).

4.3.6 Decrypt block

This module just uses the the secret key *sec* in **VOTE** to decrypt the encrypted block.

- Fetch *sec* from **VOTE**.
- Fetch the encrypted block from **VOTE**. Copy it to a file **VOTE**'.
- Denote with $\mathbf{VOTE}'[n]$ each entry in **VOTE**'. Decrypt it:
  - If $\mathbf{VOTE}'[n]$ doesn't decrypt for some n (probably it has been encrypted with the wrong public key), delete it.
  - Otherwise, replace $\mathbf{VOTE}'[n]$ with its decrypted version.
  - Return **VOTE**'.

4.3.7 Check that every string is well formed

This algorithm checks that every voting string in **VOTE**' is in the right form.

- For every n, consider $\mathbf{VOTE}'[n][1]$. Check that this is a valid ethereum address.
- Next, check that $\mathbf{VOTE}[n]$ was pushed to **VOTE** by $\mathbf{VOTE}'[n][1]$. This not being the case, it means that some ethereum address added a vote to **VOTE** using a false identity. This being the case, erase the line $\mathbf{VOTE}'[n]$.

- Then, check how many candidates (choices) are in **CANDIDATES** (**CHOICES**). This will be equal to the number of candidate (choice) strings present there. To keep consistent with our notation, denote this number with $|\mathscr{C}|$. Fetch moreover the range parameter $\mathfrak{R}$ from **VOTE**. Check that, for every n, $\mathbf{VOTE}[n][2]$ is a string of integers of length $|\mathscr{C}|$ with values ranging in $\mathfrak{R}$. If this is not the case, again delete $\mathbf{VOTE}[n][2]$.
- Finally, for every n, check that $\mathbf{VOTE}'[n][3]$ is a valid ethereum address, or 0. Again, if this is not the case, delete $\mathbf{VOTE}[n][2]$.
- Return **VOTE**'.

4.3.8 Ban foundation address

This algorithm ignores scores assigned by the Swarm foundation address.

- Scroll **VOTE**': Each entry $\mathbf{VOTE}'[n]$ Will be a voting string s. Start from $\mathbf{VOTE}'[1]$.
- If $\mathbf{VOTE}'[n][1]$ is the Swarm foundation address, delete the entry.
- Keep going: $n = n+1$.
- Eventually the bottom of **VOTE**' is reached and the voting string coming from Swarm foundation address is purged.
- Return **VOTE**'.

4.3.9 Delete repeated votes

This algorithm deletes repeated votes coming from the same address, considering only the most recent one.

1. Denote with N the number of entries in $\mathbf{VOTE}'$. Scroll **VOTE**' bottom up, starting from $\mathbf{VOTE}'[N]$.
2. For n going from $N-1$ to 1, check if $\mathbf{VOTE}'[n][1] = \mathbf{VOTE}[N][1]$.
3. If yes, delete $\mathbf{VOTE}'[n][1]$. Otherwise keep going.
4. $n = n-1$.
5. When the for loop ends, we know that there are no addresses m such that $\mathbf{VOTE}[N][1] = \mathbf{VOTE}[m][1]$, aside for the case $m = N$.
6. Set $N = N-1$ start again from step 2.
7. Eventually $N = 1$ and the top of **VOTE**' is reached. All the repeated votes have been purged.
8. Return **VOTE**'.

4.3.10 Get rid of hanging delegations

This algorithm takes care of situations like $A \to B$, but B did not submit a vote. In this case the algorithm de-activates the delegation to B and falls back to A's preferences.

1. Denote with N the number of entries in $\mathbf{VOTE}'$. Scroll **VOTE**' from top to bottom, starting from $n = 1$.
2. If $\mathbf{VOTE}'[n][3] = 0$, then no delegation has been specified, go to step 6.

3. For each m, check if $\mathbf{VOTE}'[n][3] = \mathbf{VOTE}'[m][1]$.
4. If there is one such n, then the delegation is well formed, skip.
5. If not, then the delegate didn't vote. Set $\mathbf{VOTE}'[n][3] = 0$.
6. Set $n = n + 1$ and repeat from step 2.
7. Eventually $n = N$ and the bottom of the contract is reached. All the hanging delegations have been purged.
8. Return **VOTE**'.

4.3.11 Get rid of delegation loops

This algorithm gets rid of situations like $A \to B \to C \to A$. It proceeds looking for such situation, and if it finds one, it sets all the delegation addresses in the loop to zero, falling back to A, B, C own preferences.

1. Denote with N the number of entries in $\mathbf{VOTE}'$. Scroll **VOTE**' from top to bottom, starting from $n = 1$.
2. We form a string called *check* as follows:
   (a) The first entry of *check* is $n_0 = n$.
   (b) If $\mathbf{VOTE}'[n_0][3]$ is 0 then delegation was not set for n_0. Go to step 4.
   (c) If not, we scroll **VOTE**' until we find an n_1 such that $\mathbf{VOTE}'[n_1][1] = \mathbf{VOTE}'[n_0][3]$. We are sure this n_1 exists if we get rid of hanging delegations using module 4.3.10 before running this algorithm. We add n_1 to *check*.
   (d) Every time we add a new entry to *check* we look for repeated entries. If there is a repeated entry, *check* will look like
   $$(n_0, n_1, \dots, n_i, \dots, n_i)$$
   For some i. Now, check represents a delegation sequence: The first entry is the line on **VOTE**' we started from, the second the line corresponding to the address delegated by the first one and so on. For instance, the string above corresponds to the delegation sequence
   $$\mathbf{VOTE}'[n_0][1] \to \mathbf{VOTE}'[n_1][1] \to \dots \to \mathbf{VOTE}'[n_i][1] \to \dots \to \mathbf{VOTE}'[n_i][1]$$
   - Suppose we find a repeated entry, say n_i. The first occurence of n_i will be at entry $i+1$ in *Check*. For instance, if $Check := \{1, 6, 2, 8, 21, 2\}$, then $i = 3$. We get rid of the delegation loop just setting
     $$\mathbf{VOTE}'[check[i+1]][3] = \mathbf{VOTE}'[check[i+2]][3] = \dots = \mathbf{VOTE}'[check[|check|]][3] = 0$$
     If this is the case, we solved the problem and return to step 4.
   - If we don't find a repeated entry, then we go back to step (b) using n_1 in place of n_0 and n_2 in place of n_1. At every iteration, the n_is that we are using in step (b) will be incremented to n_{i+1}.
     At some point, we either find a repetition, or that $n_{i+1} = 0$ for some i. This is again guaranteed by the application of module 4.3.10 before running this algorithm.
     In any case, this loop will terminate at some point. when this happens, we are guaranteed that the line $\mathbf{VOTE}[n_0]$ doesn't lead to a delegation loop (if it did, we got rid of it).
3. Set $n = n + 1$. Go to step 2.
4. Eventually $n = N$ and the bottom of the contract is reached. It is now purged of delegation loops.
5. Return **VOTE**'.

4.3.12 Assign reputation to every user

This algorithm multiplies the scores given by a user with his reputation, or sets those scores to zero and transfers the reputation to a delegate if this was specified.

1. Denote with N the number of entries in $\mathbf{VOTE}'$. Create a string of numbers *weights* having length N. Set every entry of *weights* to 0.
2. Create another string of numbers called *reputations* having lenght N.
3. For each n, Calculate the reputation of $\mathbf{VOTE}'[n][1]$ via module 4.3.13, and store it in $reputations[n]$. To keep things readable we will denote this with ρ_n.
4. Scroll **VOTE**' from top to bottom, starting from $n = 1$.
5. if $\mathbf{VOTE}'[n][3] = 0$, go to step 5.
6. if $\mathbf{VOTE}'[n][3]! = 0$, then it is some ethereum address (we can be sure about this if we applied module 4.3.7 before running this algorithm). We follow the delegation three. Set $n_0 = n$:
   - Create a number variable T and set it to 0. This will represent the sum of reputations that will be accumulated while following the delegation tree.
   - Set $T = T + \rho_{n_0}$.
   - Set every entry in the string $\mathbf{VOTE}'[n_0][2]$ to 0 (since the user used a valid delegation, we set his personal preferences to 0).
   - Find the n_1 such that $\mathbf{VOTE}'[n_1][1] = \mathbf{VOTE}'[n_0][3]$.
   - Set $\mathbf{VOTE}'[n_0][3]$ to 0 (We read the delegation information, so now we discard it).
   - Consider $\mathbf{VOTE}'[n_1][3]$. If it is 0, then $weights[n_1] = T$ and go to step 5.

     Otherwise, find again a n_2 such that $\mathbf{VOTE}'[n_2][1] = \mathbf{VOTE}'[n_1][3]$; then set $\mathbf{VOTE}'[n_2][2] = 0$, $\mathbf{VOTE}'[n_1][3] = 0$ and $T = T + \rho_{n_1}$. Reiterate this step until you find an n_i such that $\mathbf{VOTE}'[n_i][3] = 0$. Then go to step 5.

     Note: To be guaranteed that at some point you will find some n_i such that $\mathbf{VOTE}'[n_i][3] = 0$, and thus that this loop terminates, it is fundamental to apply modules 4.3.10 and 4.3.11 before applying the algorithm described here.
7. Set $n = n + 1$. Go to step 3.
8. At this point, for every n, $\mathbf{VOTE}'[n][3] = 0$ and, for all the n that delegated someone else, every entry of $\mathbf{VOTE}'[n][2]$ will be 0 as well.
9. Again, scroll **VOTE**' from top to bottom, starting from $n = 1$.
10. For each n, have $weights[n] = weights[N] + \rho_n$.
11. Finally, for each n set $\mathbf{VOTE}'[n][2] = weights[n]\mathbf{VOTE}'[n][2]$. Note that the right hand side of this assignment is a multiplication of a string ($\mathbf{VOTE}'[n][2]$) by a scalar ($weights[n]$).
12. Now we have our scores weighted. Note that the previous step works well because we set $\mathbf{VOTE}'[n][2]$ equal to a string of 0s in case of delegation. Thus, even if we multiply this string by the scalar representing the reputation of n, the overall result stays 0. This guarantees us that the only scores accounted for will be the ones of the corresponding delegate, whose reputation will be the sum of all the reputations of the users that delegated him.
13. Return **VOTE**'.

4.3.13 Get reputation for every user

This algorithm calculates the reputation of every user showing up in the vote. After this, it multiply all the scores by the given reputation. Reputation is made of two components: The amount of tokens held and the activity on the platform, measured considering how many times the user voted or ran for an election. These two parameters are measured with respect to the time when the voting window closed.

- Given an eth address h and a date d go to the token contract snapshot corresponding to that date and check how many tokens are in the h wallet.
- Initialize two integer variables, called γ_{tot} and γ_h, to 0. These will denote the number of Phase I votes and the number of Phase I votes in which h ran, respectively, held before d.
- For each contract referenced in **HISTORY**, check if it is a **CANDIDATES** contract. If yes, check if **CANDIDATES** voting window closed before d. If yes:
    - Set $\gamma_{tot} = \gamma_{tot} + 1$.
    - Check if h is listed among the candidates. If yes, set $\gamma_h = \gamma_h + 1$.
- Initialize two integer variables, called η_{tot} and η_h, to 0. These will denote the number of Phase II votes and the number of Phase II votes in which h voted, respectively, held before d.
- For each contract referenced in **HISTORY**, reach the corresponding **VOTE** contract. If the voting window closed before d, then:

  – Set $\eta_{tot} = \eta_{tot} + 1$.

  Fast mode For each of such contracts, fetch the plain block and check if there is any voting string n such that $n[1] = h$. If yes, set $\eta_h = \eta_h + 1$.

  Slow mode For each of such contracts, decrypt the encrypted block using module 4.3.6. Then check if there is any voting string n such that $n[1] = h$. If yes, set $\eta_h = \eta_h + 1$.

  Note: As time progressess, the Slow mode will become increasingly slow, since the quantity of votes to decrypt will grow at least linearly.

- Now, corresponding to h, we have three parameters: σ_h, the amount of tokens held. η_h, the number of votes to which h took part in, and γ_h, the number of times h run for an election. The reputation is calculated as:

$$\rho_h = \left\lfloor \sigma_h \left(k\frac{\eta_h}{\eta_{tot}} + k'\frac{\gamma_h}{\gamma_{tot}} + 1 \right) \right\rfloor$$

  k and k' are two parameters, that at the moment are set as $k = 0.5, k' = 0.1$, and $\lfloor\rfloor$ is the floor function. The reason why k' is counterintuitively less rewarding than k is to prevent flooding: People may try to propose to every election en masse in the hope to boost their voting power on the platform.

  Note that with this formula, a user not holding any tokens has reputation equal to zero, meaning that he has no say whatsoever in Swarm governance. Also, supposing that someone runs for every election and votes at every vote, he can boost his reputation by a maximum of 1.6 with respect to the amount of tokens held.

4.3.14 Calculate the winner(s)

This algorithm sums all the scores in a list of strings **VOTE**’ and picks the best ones as prescribed by the instructions on the **VOTE** contract. It returns two arrays of numbers: The first represents the sure winner(s), the second represent the remaining winner in case pseudorandom selection is needed.

1. Denote with N the length of **VOTE**'. Calculate $S := \sum_{n=1}^{N} \textbf{VOTE}'[n][2]$, where with $\sum$ we are denoting the component-wise sum of strings of numbers.
2. Initialize two empty strings, call them K and P. They represent the set of winner(s) and the set of possible winner(s) that will have to be pseudorandomly chosen, respectively. Note that P may be empty.
3. Fetch from the **VOTE** contract instructions the number of winning candidates (choices) we have to pick. Suppose this number is w.
4. Find all the n not in K such that, for any n' not in K, $S[n] \geq S[n']$. Add these integers to P.
5. Proceed by cases:
   - If $|K| + |P| = w$, then append all the entries of P to K, set P to the empty string and go to step 6.
   - If $|K| + |P| < w$, then append all the entries of P to K, set P to the empty string and go to step 4.
   - If $|K| + |P| > w$, then we have too many elements in P. If the vote comes from an election, we can further select these elements in P until we hit the number w. If not, then we need to select alternatives in P pseudorandomly. Proceed by cases: If **VOTE** is referenced in a **CHOICES** contract, go to step 6. If **VOTE** is referenced in a **CANDIDATES** contract, then:
     (a) Form a string of the same length of P and initialize it to 0, call it R. For every p, fetch $P[p][1]$ in the **CANDIDATES** contract and calculate the reputation $\rho_{P[p][1]}$.[5] Set $R[p] := \rho_{P[p][1]}$.
     (b) Initialize an empty string P'.
     (c) Find all the p such that, for any p', $R[p] \geq R[p']$. Add these numbers to P'.
       - If $|K| + |P'| = w$, then for each p in P' append $P[p]$ to $|K|$, set P to the empty string and go to step 6.
       - If $|K| + |P'| < w$, then for each p in P' append $P[p]$ to $|K|$, purge the p-th entry from $|P|$ and go to step (a).
       - If $|K| + |P'| > w$, then purge all the $P[p]$ such that p is not in P' and go to step 6.
6. Return K and P.

4.3.15 Check that winner coincides with the one in VOTE

This algorithm takes the couple of strings of integers K and P returned by module 4.3.14 and checks if they coincide with the winner(s) specified in the relevant contract.

- Fetch the winner(s) from **VOTE**. Form a string with these numbers, call it W.
- If $|W| = |K|$, check that $W = K$. If not, produce an error.
- If $W \geq K$, check that every entry of W is in K or in P. If not, produce an error.

[5] This is again done invoking module 4.3.13. Again, fast or slow mode will have to be selected according to the type of check we are performing.

References

[1] K. J. Arrow. *Social choice and individual values*. Monograph (Yale University. Cowles Foundation for Research in Economics). New York: Wiley, 1951 (cit. on pp. 1, 2).

[2] A. Gibbard. “Manipulation of Voting Schemes: A General Result”. In: *Econometrica* 41.4 (1973), p. 587. ISSN: 00129682. DOI: `10.2307/1914083`. arXiv: `arXiv:1011.1669v3` (cit. on p. 1).

[3] M. A. Satterthwaite. “Strategy-proofness and Arrow’s conditions: Existence and correspondence theorems for voting procedures and social welfare functions”. In: *Journal of Economic Theory* 10.2 (1975), pp. 187–217. ISSN: 10957235. DOI: `10.1016/0022-0531(75)90050-2` (cit. on pp. 1, 2).

[4] *The Center for Election Science*. URL: `http://www.electology.org/` (visited on 09/24/2017) (cit. on p. 2).

[5] *The Center for Range Voting*. URL: `http://rangevoting.org/` (visited on 09/24/2017) (cit. on p. 2).