BETHabase Report: A Decentralized Solution for Special Image Aquisition and Social Consensus

Ozan Ünal[1] (ouenal@ethz.ch)
Orhun Caner Eren[1] (oreren@ethz.ch)
Adrian Balla[1] (aballa@ethz.ch)
Clemens Volk[1] (volkc@ethz.ch
Ege Yılmaz[1] (yilmaze@ethz.ch)

[1]ETH Zurich, Switzerland

All members contributed equally to this report

Abstract

There is an ever growing demand for labeled datasets as machine learning finds its way into new areas and industries. Image labeling is not a trivial task and can be highly costly for specific requests. The aim of our project is to provide a solution to the problem of database collection by utilizing blockchain to establish a distributed marketplace for labeled image acquisition. BETHabase users are given the chance to upload images and verify the integrity of the accumulated data by forming a social consensus, that thrives on not only Ether gain but also a reputation system.

Contents

1 Introduction to Blockchain

A blockchain, is a linked growing list of records. In other words, the batches of records are stored in blocks which are chained together chronologically. Blockchain therefore enables a way of maintaining records of transactions.

In modern world, we have a central authority, a records keeper who would be designated the ability to authorize the occurring transactions, such as money exchange, medical records, media ownerships, land titles etc. With blockchain however, this necessity of the authority is eliminated, because blockchain network consists of nodes that each of them possesses a copy of the whole ledger that carries the records of transactions. By its nature, this distributed ledger aspect of blockchain yields a consensus of network. This gives the blockchain technology the potential to become a game changer.

1.1 Democracy and Collaboration with Blockchain

1.1.1 Decentralization, Trust and Transparency

When data is uploaded on blockchain, it gets distributed across a large network of computers. Since information is not stored on a single system, it is decentralized.

This decentralization of information reduces the possibility of data tampering. It creates trust in the data: Because the transactions are communicated via a whole network that is responsible for verifying or rejecting these, any modification must get the approval of the network. In case of a hacking attempt, it must be done simultaneously in every computer that is a part of the network, in other words the data on a blockchain is immutable.

Blockchain stores information in a non-destructive, chronological manner. Any changes made to a block is stored as a new entry to the chain with a timestamp.

One of the famous examples that needed this level of transparency to exist is 'Bitcoin'. The problem that the concept of cryptocurrency faced was the contingency of it getting printed in unlimited amounts, which then would lead to its devaluation. With blockchain governance, bitcoin transactions exist in an 'Open Ledger', meaning they are public and traceable. Thus bitcoin can not be arbitrarily created.

1.1.2 Data Confidentiality and Less Friction in Transactions

Today, in many cases we need to trust some third party institutions to protect and verify our personal information. By storing our data on a blockchain we can have ownership of our intellectual property or commodity without the need for other institutions to speak up for us in case of a violation/dispute against our ownership. This shift from institutional trust to computational trust can lead to less exposure of our information by narrowing the process chain and hence eliminate the potential risk in sharing our information with third parties and expecting them to maintain and authenticate it.

Securing information on the blockchain cuts out trusted intermediaries via smart contracts, which are self-executing scripts written into lines of code that carries out the agreements of the parties. This is done without the need for any legal system or external enforcement mechanism so that the loss of time and money caused by extra steps in the exchange is averted.

1.2 Blockchain and Sustainability

Preserving the biosphere, the social and economical conjuncture for generations to come is currently among the biggest concerns. The world faces issues such as global warming, inaccessibility to safe water, unsafe outdoor air-pollution, social inequality, marginalization and many others. Can blockchain technology help us solving some of these issues?

1.2.1 Impact of Transparency on Sustainability

Transparency is essential for the well-functioning of any type of institution and blockchain technology can help organizations become transparent and democratic. Transparency that blockchain enforces can help tackling sustainability issues in various ways:

- Informing the consumers about the details of a product's supply chain, e.g. where and how it is produced, which materials and chemicals are used in its production, its origin and movement until it reaches the store shelves, where the waste goes, allows them to choose environment friendly products.

  'Provenance' is a blockchain based application that gathers and shares the details about products' nutritional content, safety, quality and impact on the environment. It allows the shoppers to find brands that match their environmental criteria.

- Blockchain can be used to track carbon footprints, greenhouse gases or waste emissions from factories or a companys overall history of compliance to environmental standards. It allows companies or individuals to see direct effects of their actions and gives them th chance to take actions that will benefit the environment.

- Incentivizing environmental practices, such as recycling, is key to deal with sustainibilty issues we are facing. 'Plastic Bank', one of IBM's 'Cognitive Foundry' startups, is working on this topic. They incentivize people in third world countries to collect plastic waste by rewarding them in the form of blockchain secured digital tokens. This way not only plastic pollution but also social inequality is dealt with.

- It is difficult to track the outcome of environmental treaties as well as to what degree it is administered by governments. If the process is carried out on blockchain, governments and corporations are forced to be transparent on the matter, hence the chance of fraud or misreport or manipulation of data is eliminated. This would also cut down on administrative costs due to tracking of carbon credits and legal documents.

1.2.2 Impact of Decentralization on Sustainability

The world economy is based on extracting low entropy available energy in nature like metallic ore, fossil fuel and throughout value chains storing and shipping it, producing goods and services from it, consuming it and recycling it back to nature. While moving nature's resources through society, at every step of conversion we lose some of the energy that is supplied for the transit of goods and services to the next stage of what it becomes.

The aggregate efficiency of todays fossil fuel-based infrastructure has reached its sealing. Blockchain can be the technology that will allow us to engage each other directly on a global internet of things, bypassing vertically integrated organizations. This would increase aggregate efficiency and productivity, reduce ecological footprints and plunge marginal costs:

- When energy from a centralized source is distributed over long distances, some portion of it gets lost along the way.

  'Suncontract' is a blockchain powered peer-to-peer platform that incentivizes energy decentralization by allowing local producers to exchange their home generated renewable energy directly with their neighbours and local community.

  'Solar coin' is another example that incentivizes individuals to install green energy capturing devices by giving digital tokens to individuals for their energy production.

- Today we have big companies such as 'Uber', 'Airbnb', that aggregate services and then sell them or businesses that provide third party tools connecting intellectual properties with the consumers. Instead, a decentralized application using blockchain technology could do the same by identifying parties and bringing them together via smart contracts. This would lead to a redistribution of wealth and contribute to the overall prosperity by bringing the money concentrated in intermediaries to people.

- People who make or consider making donations often share concerns regarding if their money is put in good use. A system such as blockchain solves this by discarding centralized authorities, allowing donors to transact and track their money. This is an important step to incentivize people to give to charity for environmental causes.

2 The Concept

2.1 Verifying Datasets Based on Human Input

Artificial intelligence and machine learning have become not only a crucially important area of research but also an integral part of modern society with the rise of smart devices and increasing product demand, with the promise of self driving cars and a better, more efficient future. More and more we see machine learning, more specifically segmentation, localization and classification tasks being implemented in areas among many other: quality control, health and safety inspection and production waste reduction. Such tasks of segmentation, localization and classification that are widely used in industry are only possible by using supervised learning techniques which are machine learning applications that require a large *labeled* quantities of data in order to train the system. The larger and more diverse the dataset gets, the more robust and accurate the system can become.

In addition to programming and training a supervised learning algorithm, the generation or acquisition of a suitable dataset presents a real challenge. A researcher alone, no matter the insight, no matter the expertise in a given domain, can never acquire a diverse set of one million images to train his/her system. This inadvertently means that a small group of researchers can never compete with corporations owning data-generating networks or crawling systems that not only already posses billions of labeled images but have the resources of acquiring more when required.

There exists no mechanism for the broad masses that will incentivize the participation in image collection and labeling. Even if there were, there is no easy way to verify the accuracy of the labels. Only big cooperation are eligible of acquiring verified datasets, limiting democratisation of technological growth.

Our proposed method does not only enable such dataset collection using a social consensus, but would empower the free-flow of verified data, giving small research institutions the access to affordable datasets. Using a decentralized solution for the information exchange, we enable the acquisition of verified data through a the social consensus. Gamification through a reputation system introduces another incentive for honest collaboration that is enhanced by the reward of Ether given a correct submission.

2.2 Four Step Solution in Generating a Verified Labeled Dataset

The generation of a labeled dataset starts with a request by a client, provided images to the request by the uploaders, and the verification of said images using the social consensus. To make the system more understandable for the reader, we provide an example case that is woven after each explanation.

1. **Client:** The client sends a request by specifying the *keyword* or *tag* to which a dataset should be collected as well as the desired number of im-

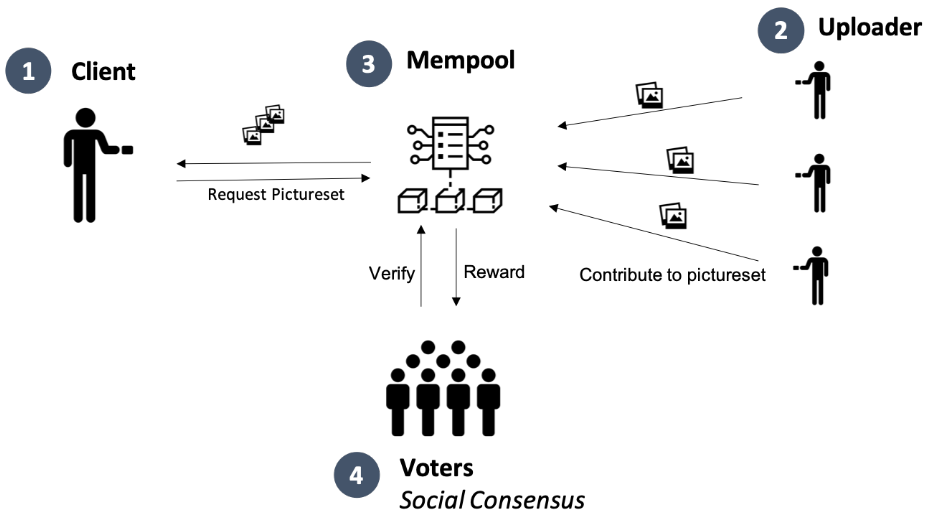


Figure 1: Overview of the BETHabase backend

ages. The request along with the upfront payment is submitted to the blockchain, where the payment is stored in the smart contract.

Let's assume that I wish to form my own start-up company that will collect recycled cans, extract the raw materials used and refurbish them by making shelters for the homeless. For this I would need to automatize the collection process, as shelters require high amounts of raw material, and it would be impossible for me to handpick each can. I wish to develop a system where I can easily identify recyclable cans. This is a simple classification task that can be easily solved using supervised learning and training over thousands of labeled images of recyclable cans. But as a solo entrepreneur I do not posses the endless pockets of giant corporations.

Using BETHabase, I can simply create a request for the tag "recyclable can" for 10 000 images. I would have to pay 0.1 ETH up front which is quite achievable for any individual like myself.

2. **Uploader:** The Uploaders can see every tags currently being requested. An image can be uploaded to any of the given tags at any time. The Interplanetary File System (IPFS) serves as actual storage system for the image data.

Going back to our example, let's assume that now I am a user of BETHabase. I love recycling cans and I always have one lying around waiting to get recycled. I open my app and request the available tags currently in the blockchain. In the list I see "recyclable can". Perfect! I click on the tag and take a picture of the can sitting on my desk. I click upload.

3. **Mempool:** The pictures which are uploaded by the uploaders get stored temporarely in the so called mempool. In other Blockchains, like the proof of work implementation of Bitcoin, the mempool is best described by a waiting room of unconfirmed transactions. Once a transaction happens, it is not immediately considered as a valid transaction. It is until miners find the work intensive solution to the proof of work riddle and the transaction is beeing processed, put into the respective block which was mined and removed from the mempool- the list of unverified transactions. At BETHabase we introduced a new mechanism of consensus mechanism for problems in social applications. We call it the Social consensus. Like in the Bitcoin network, uploaded images are stored in a list of unverified transactions until they get finally mined and confirmed. However it is not based on the energy intensive proof of work mechanism to come to a globally acting consensus. We rather have implemented a social consensus based on reputation and reward for correct voting power.

4. **Voters (Social Consensus):** A voter can at any given time request an image from the Blockchain and vote as to what he/she believes the answer is from a set of given possibilities. Giving the social consensus a choice between multiple answers, where one answer is the original tag to which the photo was uploaded, allows to elegantly solve the verification problem by matching tags. If the majority vote on the tag to which the image was uploaded, the image gets accepted and all parties involved (both voters and uploader) receive payment in means of Ether and reputation tokens. If an image receives more votes on tags other than what it was intended for it gets discarded. A reputation decrease takes place for all parties.

Let's assume I am now a voter. On the app I click the Vote tab and am presented with an image and 6 tags. The tags read "red car", "recyclable can", "Ultrasound machine", "pencil", "table", "pass". I see that the image is a recyclable can so I pick the option and vote.

If others have also voted for the recyclable can, the vote majority would match the original tag to which the image was uploaded. In return for my effort I receive Ether and a reputation token. I can at this point look at the reputation list and see how I am compared to my friends or everyone else in the world.

Along with myself, all other voters and the uploader also receive these rewards once the image is accepted by the blockchain.

2.3 Front End Proof of Concept

To interact with the smart contract and the respective backend, we implemented a Proof of Concept Front End to interact with the blockchain and the Interplanetary File System. The Front End is fully functional in its core functionalities excluding gamification mechanism. A sketch and the idea of the first client facing front end is showed in the following.

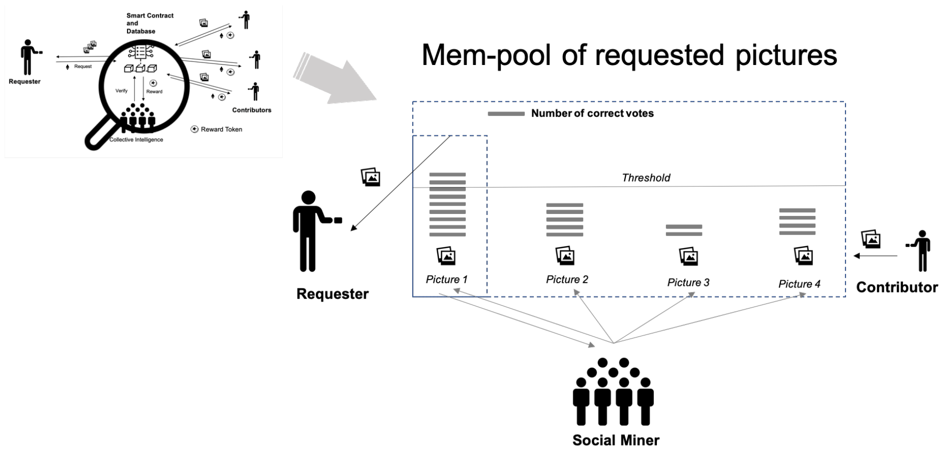


Figure 2: A closeup of the Mempool (3) from Fig. 1

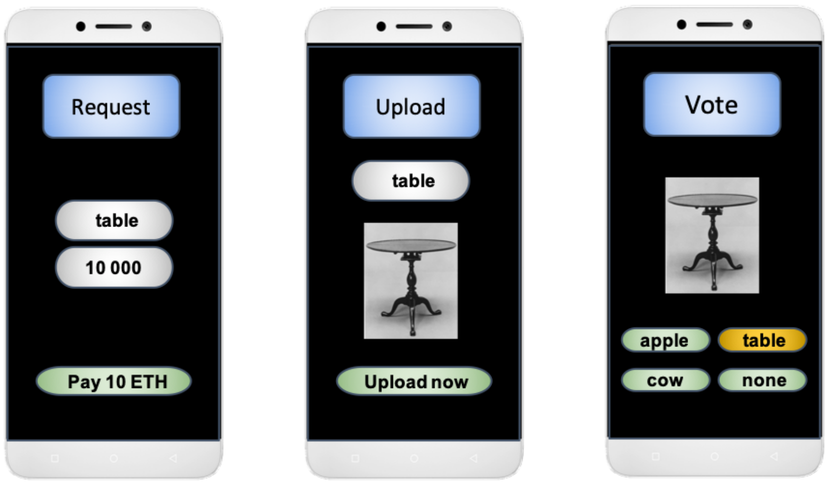
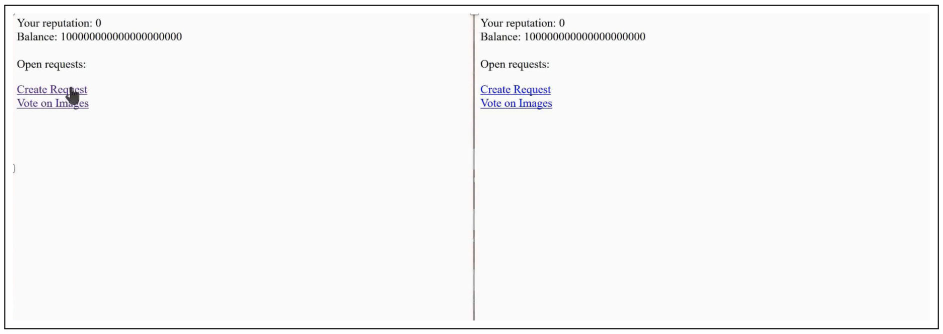


Figure 3: Client facing front end showing the three steps of requesting the type of pictures with the needed amount, the uploading page of the uploaders and a voting page voting for the respective truthful label.

This sketch is implemented in our Proof Of Concept Product as a working front end. The key steps of requesting and uploading pictures are shown in the following pictures.



Your reputation: 0
Balance: 100000000000000000000

Open requests:

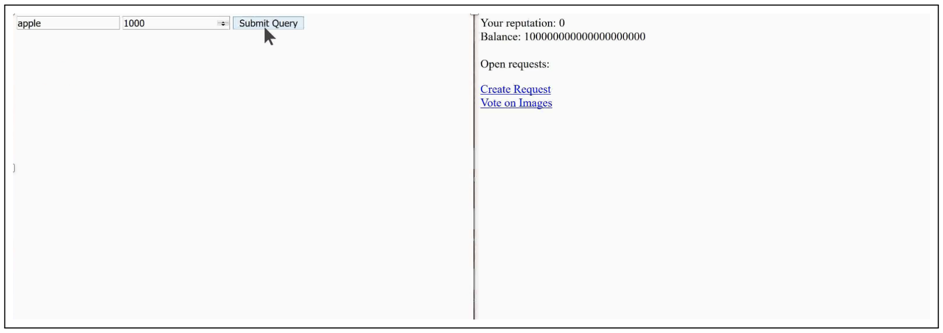
Create Request
Vote on Images

Your reputation: 0
Balance: 100000000000000000000

Open requests:

Create Request
Vote on Images

Figure 4: Two user interfaces are shown which displays their Ethereum account balance and their respective reputation balance. A user has the choice of either beeing the requester or a voter. In this case, the client on the left is the requester and is in need for a specific set of labelled pictures.



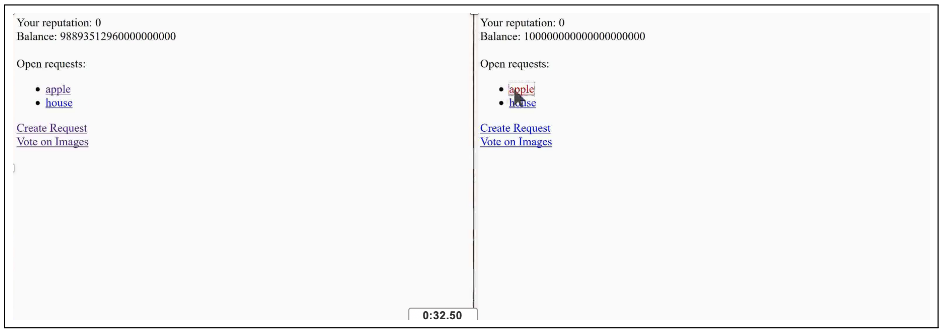
apple 1000 Submit Query

Your reputation: 0
Balance: 100000000000000000000

Open requests:

Create Request
Vote on Images

Figure 5: The requester sends out a request for 1000 pictures of an apple.



Your reputation: 0
Balance: 98893512960000000000

Open requests:

- apple
- house

Create Request
Vote on Images

Your reputation: 0
Balance: 100000000000000000000

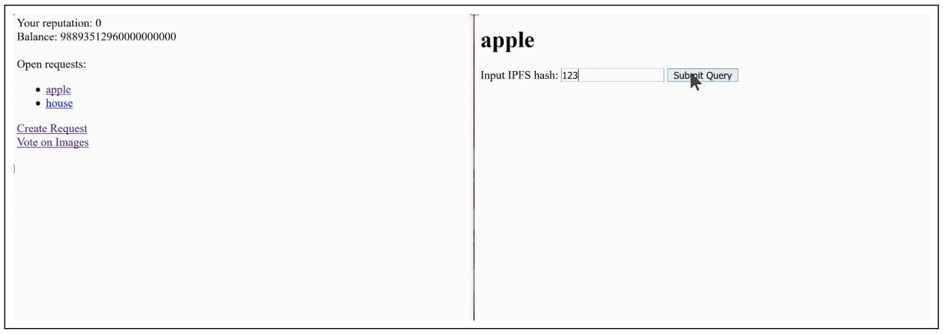
Open requests:

- apple
- house

Create Request
Vote on Images

0:32.50

Figure 6: The respective ETH balance is reducted by the corresponding fee which is paid into the smart contract automatically. The request is labelled and put into the public query list of open requests. An uploader can view the tags of requested images.


Your reputation: 0
Balance: 98893512960000000000

Open requests:

- apple
- house

Create Request
Vote on Images

apple

Input IPFS hash: 123

Figure 7: The uploader uploads the requested image to the Interplanetary File System and enters its hash which is then stored in the same tag-request database

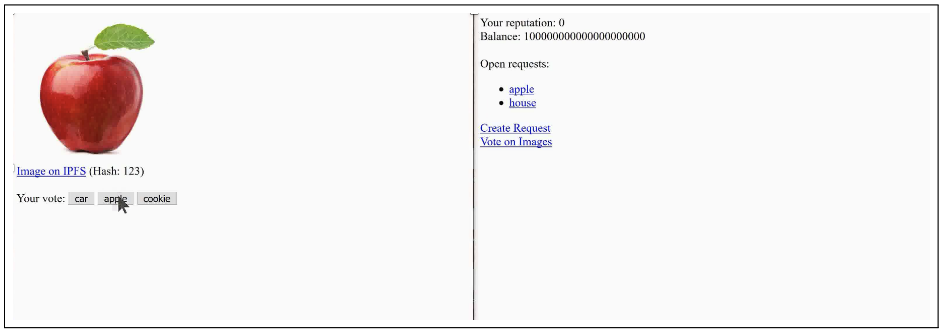


Image on IPFS (Hash: 123)

Your vote:

Your reputation: 0
Balance: 100000000000000000000

Open requests:

- apple
- house

Create Request
Vote on Images

Figure 8: The uploaded image is now stored in the mempool and waits to be declared as a valid tagged image. A voter sees all the open positions and has the chance to vote for the right label. The right label will then be depicted when a certain threshold is met. All voters with the correc t vote are rewarded with an reputation token. All incorrect voters are deducted one reputation token.

2.4 Incentivizing Honest Collaboration while Enabling Gamification

Honest collaboration is expected since the Uploader and the Voters must pay the gas prices for the transactions they call. They only earn back their investment and much more only when the image is accepted. We introduce a reputation token to further incentivize honest collaboration. Each correct vote and upload earns the user a reputation token. Each wrong vote reduces the amount by one with a lower bound of zero. A mechanism by which a higher balance in the reputation account corresponds to a greater voting power and therefore the possibility to earn a larger stake of the "transaction-fees", incentivizes on a monetary level to take part inside the ecosystem with best intentions. The more reputation token a social miner is holding, the bigger his earnings will be, once an image is accepted and declared as verified. On the contrary, a dishonest actor will disqualify himself as not valid. Every time a social miner is not taking part in the correct decision making process, he loses a reputation token. By losing reputation token, he loses future gains on correct votes. So on a monetary level it is more profitable to keep the honest voting power and gain on the respective honest collaboration. Besides the fact that honest collaboration is much more profitable, there are other factors which play an important role in the implementation of the reputation token system. It is the social factor which drives us humans to new frontiers. Humans are social beings and therefore thrive under the acceptance of other social companions. It is not to underestimate how importante gamification implementations are in a social engagement. First of all is the ranking list one of the most basic but important gamification tools to measure one effort inside a given environment, beeing it a score in an online game, or the ranking of honest collaborators. Gamification systems allow participants to compete with others in a fun way and enable the naturally building of a community. Around such a gamificated reward system, many more implementaions can be put into realization. The community could set up its own lottery system with its own goals and tasks to further incentivice honest collaboration. As an example an daily goal could be set up to reward the most active social miner or the most honest miner given a certain time frame.

What is yet to be implemented in the project is a reputation list with its gamificated tools. With a reputation list and gamification implementations online, users can compete with each other for the top seats both worldwide and among their friends.

3 Architecture & Design

This section is divided into two parts: First we describe the actual implementation done during the hackathon. Thereafter we explain a more sensible but complicated approach in theory, as the realization is more involved.

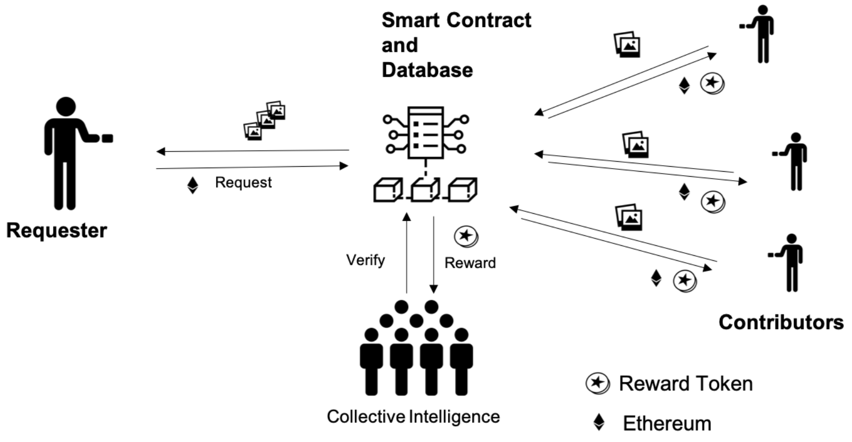


Figure 9: Reputation token flow

3.1 Design Overview of Implementation

A single contract oversees the entire process. Each type of user interacts to the same contract differently. Necessary information for the system to operate is governed by this main contract. This approach is relatively easy to design the system, however it is difficult to upgrade and maintain due to its bulky size.

To allow dynamic shape and size each necessary array is constructed as lists which is inherently dynamic in size such as list of datasets, list of members, list of voters etc. Images are always stored in IPFS (Interplanetary File System) which is a distributed approach to file storage. Hence knowing the address of last image in the list is sufficient to extract entire chain and centralize the dataset for the client once it is completed.

3.2 Hypothetical System Design

Naïve solution

The straightforward way of structuring the components is as follows: A Request object holds keyword, the requested amount of uploads and a list of Upload objects. These in turn comprise the IPFS hash and the votes (plus outcome).

There are several issues with this approach in terms of susceptibility to voting fraud and exploitation. We will address (un)desirable properties and discuss models and difficulties.
We aim for verification based on genuine user input. Hence blind copying or fake votes are unwanted. So the following actions should be inhibited or at least discouraged, for instance by imposing computational obstacles such that automation is harder. We consider a linear search through the database as sufficient.

1. *Voter predicts the majority vote of an upload without seeing its image.*
   There are multiple indicators for a poll to have a certain outcome. Previously submitted votes being one of them, the request's keyword is another.
   (a) *User voting on a particular upload accesses its previously submitted votes.*
   By waiting until an upload poll is nearly full one can determine the majority vote and stick with it, i.e. win dishonestly.
   (b) *User voting on a particular upload reads its corresponding request text.*
   Submitting the original keyword misses the point of verification and counts as fake vote.
   (c) *User voting on a particular upload accesses votes or outcome of an upload poll belonging to the same request.*
   Uploads for the same request are likely to receive similar votes, but votes for one have no verification value for the other.
2. *User casts duplicate votes.*
   An entity can control or at least influence the majority vote by registering the same vote using multiple accounts. In case of a takeover all winning voters gain reputation tokens, which provides them with more power and therefore is a potential positive feedback loop, but also will the upload be accepted irrespective of its image content/ground-truth keyword and the uploader will receive tokens too.
3. *Uploader re-submits an image already in the database.*
   A user could query the database for an image given a keyword and upload.

First, we will solve issue 1a:

Problem: Votes are public

When multiple users vote on a poll, one after the other, later voters will be able to inspect the published votes of others, which might influence the final outcome in an unwanted way. This is not how real-world voting is done after all.
In particular, voters would wait and finally copy the majority vote to become a winner.

Solution: Commit-Reveal

To keep everybody's vote secret until the poll has ended, one can make use of the *commitment scheme* where the process is divided into two phases: *commit* and *reveal*.
In the first phase every voter settles on a choice v and generates a random string r. These two compose the committed value $t = (v, r)$. The hash $h := H(t)$ is computed using the (secure) hash function H and then submitted.
After the poll has ended, the next phase starts and all voters must disclose their

original choice of t. A false revelation can be detected by comparing h with $H(t)$.
Now that all votes are public, the outcome can be determined.
It is infeasible to compute t based on h, so we can neglect the chance of a user successfully changing their vote after phase 1 or a user computing another's committed value before phase 2.

With 1b and 1c being more difficult to deal with, we will first discuss 2 and 3.
Problem 2 can be mitigated by assigning the upload polls to users, as done in the smart contract implementation attempt, rather than giving freedom of choice. Duplicate accounts cannot be detected however.
Action 3 can be detected when an exact copy of the image is uploaded: The IPFS hash stays the same. Using a hash set over IPFS hashes, images that were already added to the database can be rejected.

Regarding 1b) and 1c), we came up with the following advanced structure: Let `Request`, `Bucket`, `Upload` be object types. The corresponding objects hold data and point to each other as indicated in the following UML-like diagram.

When a request is submitted, a `Request` is created together with a `Bucket`. `Bucket` keeps track of the number of uploads for a request and at the same time serves as indirection between the `Request` containing the keyword and the `Upload`s belonging to it. To upload an image for a request, the user selects the `Request`, resolves the corresponding `Bucket`, commits to it and then an `Upload` is created given the IPFS hash and the commit-value.
After the upload poll hash finished, its uploader must notify its `Bucket`.
The `Request` is fulfilled when the number of `Upload`s to its `Bucket` reached the maximum. All `Upload` →`Bucket` pointers are revealed.
One the one hand, because the `Bucket` to which an `Upload` belongs, i.e. points to, is not revealed until the request is fulfilled (only committed), voters voting on a particular `Upload` cannot directly access the `Bucket` nor the `Request` (marks 1b off).
On the other hand, there is no direct way to obtain the respective `Upload`s from their `Bucket` before the request is fulfilled (1c).
Nevertheless, one could listen for transactions in which an upload is performed. These transactions involve both the uploader's ID (address) as well as a reference to the `Bucket` in order to increment the counter. Thus by tracking these values as well as building lookup tables for reversing the pointers from `Upload`, `Bucket` and also `Request` one could arrive at action 1b and 1c, however this requires a lot of effort and computation.

The stored IPFS hashes for a request can easily be iterated to finally retrieve the images.

4 Implementation

The work flow of the implementation can be explained as:

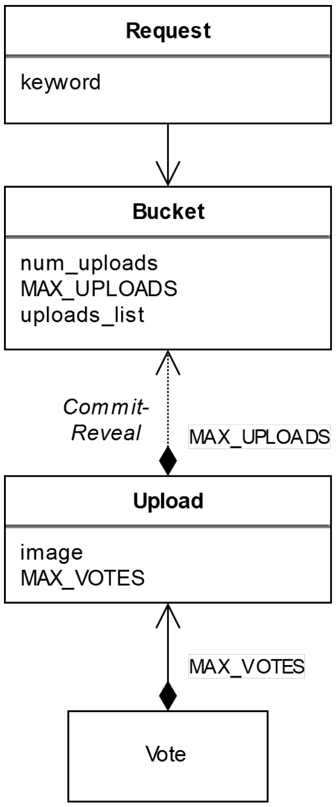


Figure 10: BETHabase Architecture

1. Clients request dataset via their application by calling a request function giving the tag name and number of images wanted for that given dataset. The function creates an empty array list for storing the approved images to be stored in IPFS and adds the tag of the dataset to array-list of tags, sets a boolean to false in order to check the status of completion, and assigns the address of the client as the owner of the dataset in order to enable access to it afterwards.

2. When uploaders open their apps they send a get available tags request to the contract, and contract transmits the current incomplete datasets to uploaders. Uploader only sees what contract transmits as result of the initial get function which are only incompleted datasets. If Uploader decides to upload a photo, he/she chooses a tag from app and takes a photo. The photo is uploaded to IPFS automatically and the address in the IPFS is transmitted to contract with the id of the tag that it is proposed as.

3. Once the IPFS address of the image and its proposed set is received from

the uploader, they are added to a link-list of all the proposed images and the account address of the uploader is also stored due to payment and rewarding if the validation is successful. This link list can be thought as a waiting queue for validation. Each link is stored here until it is validated by the voting procedure. A controlling step can be added here to check whether uploaded image is in the proposed dataset or not to prevent duplicate uploading. However this is not yet implemented.

4. When the app user decides to vote for images, the app sends a get request to the contract. The IPFS address and the proposed tag of the current image which is pointed by an ID pointer in the proposed image list is transmitted. Voting of a single user is handled by the application. This includes displaying multiple choice tags and doing a preliminary test of random images to increase credibility etc. However vote counting and validation of the results are done by the contract. Please note that when multiple voters at the same time request the image they all receive information about the same image, however when they vote only the votes before threshold are counted. The counting is done in pigeon nest manner with respect to 10 total voters. In other meaning, If 6 votes are required for acceptance, when "YES" vote count reaches 6, it is accepted or if "NO" votes reaches 5 first which mean on 10 scale the "YES" votes will be 5 at most, the image is rejected. This prevents paradigms such as no voting or lost votes. Only first 10 votes are counted in worst case 5 "YES" 4 "NO" votes which means there are multiple liars in each group which can be assume an event of low probability. In all cases the account addresses of the voters are stored in a list for rewarding and paying to majority which is assumed to be pointing the truth.

5. After voting process, the pointer of the current image is shifted by one. If the validation of the image is considered successful, i.e, "YES" vote count reaches 6 first. The IPFS address of the image is added to linked list of corresponding tag that it is being proposed. The dataset with approved tag is updated such as remaining number of images and completion status etc. And finally the payment is done to uploader and the voters who voted for "YES". In case of majority voting for "NO" no payment is done and reputation tokens are given to "NO" voters. A possible implementation for compensating the ETH cost for sending "NO" vote can be deducted from uploader, but this has not been implemented in here.

With this work flow, the datasets progress in parallel with totally voluntary efforts and client can always check the progress of his/her dataset at any given time, and once it is finished he/she can download it from IPFS addresses. And even though the dataset is complete since it is initial address is stored in the contract it can be used later for the cases such as expanding a dataset or selling the dataset to another client without creating another instance but just by sharing the IPFS addresses. The only down-side of this approach is that during improvements in the code or in patching copy of the entire system is required. Due to every flow is handled by single contract and every information is somewhat centralized by the contract itself.

4.1 Commit-Reveal for keeping votes and references secret

The commitment scheme was not integrated into the smart contract. A separate, self-contained demo is provided in the repository (`BETHabase/commitment_demo`). It consists of a contract with web frontend, from which a user can commit and reveal text, while stage transitions can be observed.
The contract (section 6.2) keeps track of the current stage using a value from the `enum` below.

```
    enum Stage {
        Commit,
        Reveal,
        Finished
    }
```

When committing a string from the user interface, the contract's `commit` function is called and similarly, for a revelation, the `reveal` function is called. The state diagram visualizes all this. Conditions for the state transitions are to the right of the arrows.

The limit for the number of participants (`MAX_VOTER_COUNT`) is set to 1, but may be changed any other value.

5 Conclusion and Outlook

As we approach to a third industrial revolution based on a sharing economy, internet of things and renewable energy sources, it can be expected that the presence of blockchain technology will increasingly gain significance.

A sharing economy of a sustainable society would need a platform, which allows fast communication among peers, ensures data security, tracks the big data flow of internet of things, incentivizes social entrepreneuralism and allows trade at near zero marginal cost. It looks promising that this platform would be based on blockchain technology due to the trust and transparency it provides, the way it allows to track data and its incentivization capacity.

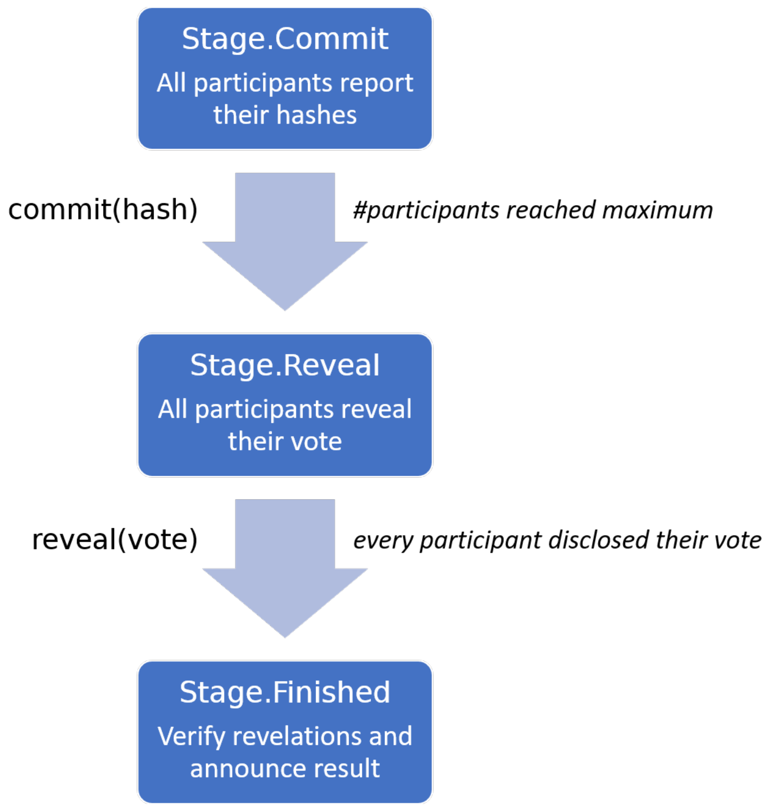

Stage.Commit
All participants report their hashes
commit(hash)
#participants reached maximum
Stage.Reveal
All participants reveal their vote
reveal(vote)
every participant disclosed their vote
Stage.Finished
Verify revelations and announce result

Figure 11: Commit-Reveal Diagram

6 Appendix

6.1 Backend

```
pragma solidity ^0.5.1;

contract Database
{
   //adding New user to Database
   struct Members {
       string FirstName;
       string LastName;
       uint RepBalance;
    }

   //globally define the value of a single image
   uint256 IMAGEVALUE = 10**12;

   //assign adress to each member and put an array of member
       accounts
   //allows look up a specific instructor with their Ethereum
       address
   mapping (address => Members) members;
   address[] public MemberAccts;

   //Adding a new member with an initial Reputation_Token balance 0
   //Mapp the public key of the sender account (the account who
       registers) to its corresponding struct Members
   //Add it to the last place in the overall MemberAccts array
    function setMember (string memory _FirstName, string memory
        _LastName) public {
       Members memory member = members[msg.sender];
       member.RepBalance = 0;

       member.FirstName = _FirstName;
       member.LastName =_LastName;

       MemberAccts.push(msg.sender) -1;
   }

   // Get members which are registered
   function getMembers() view public returns (address[] memory) {
       return MemberAccts;
   }

   //Get the balance of Reputation_Tokens for a specific address
   function getReputationTokenOfMember(address _ETHaddress) view
       public returns (uint256) {
       return members[_ETHaddress].RepBalance;
   }

   //Count total members on platform
   function countMembers() view public returns (uint) {
       return MemberAccts.length;
   }


```

```
    function getAddress() view public returns (address) {
        return msg.sender;
    }

   //Setting the variables for the request of images
   //@param nameOfTag Tag of the picture that is requested
   //@param aximumNumberOfImages Number of pictures needed
   //@param remainingNumberOfImages how many pictures are left till completion
   //@param perImagePrice Price which will be paid per accepted image
   //@param idOfRequestedDatabase Every Requested Database gets an ID assigned to track it later on
   //@param db Hash array of accepted images for given request
   //@param completed is set to 1 if request is done
   struct RequestedDatabase {
       string nameOfTag;
       uint maximumNumberOfImages;
       uint remainingNumberOfImages;
       uint perImagePrice;
       uint idOfRequestedDatabase;
       uint[] db;
       address ownerOfDatabase;
       bool completed;
   }

//The uploader proposes an image to the request which then needs to be verified
//@param idOfImageProposal unique id of the uploaded image
//@param hashOfImageProposal IPFS address of uploaded picture
//@param tagIdOfProposal is same as idOfRequestedDatabase (above) if the proposed image is voted correct
//@param uploaderAdress Address of uploaded image
//@param votersForCorrect array of all voters who voted for the proposed image fitting the requested pictures
//@param votersForWrong array of all voters who voted that the uploaded picture does not fit the request
//@param countCorrect #of answers for correct
//@param countFalse #of answers for false
//@param numberOfReceiversLeft for future development
//@param numberOfTotalReceivers for future develpment
   struct ImageProposal{
       uint idOfImageProposal;
       uint hashOfImageProposal;
       uint tagIdOfProposal;
       address payable uploaderAdress;
       address[] receiverAddresses;
       address payable[] votersForCorrect;
       address[] votersForWrong;
       uint countCorrect;
       uint countFalse;
       uint numberOfReceiversLeft;
       uint numberOfTotalReceivers;
       }

  //Events fired when NewRequestIsDone & NewImageIsUploaded
   event NewRequestIsDone(string nameOfTag, uint
       idOfRequestedDatabase, uint perImagePrice);
   event NewImageIsUploaded(uint hashOfImageProposal, uint
       idOfProposedTag);

  //not used yet
   mapping (uint => address ) private databaseIdToOwnerOfDatabase;

  //@param currentProposedImageIndex index pointer to current image
       proposal
  //Set up arrays for requests and image proposals
   uint public currentProposedImageIndex = 0;
   RequestedDatabase[] public requests;
   ImageProposal[] public proposedImages;

   //Get the balance of the smart contract
   function getBalance() public view returns (uint256) {
       return address(this).balance;
   }

   //Create new image Database request with the tag which describes
        the needed Dataset and the amlunt of pictures needed
   function createNewDatabaseRequest(string memory _name, uint
       _size) payable public {

       //Deposit a payment in ether which is later distributed upon
            completion to the uploader and voters
       uint _ETHPerImg = IMAGEVALUE;
       require(msg.value == _ETHPerImg*_size);

       RequestedDatabase memory request;
       request.nameOfTag = _name;
       request.maximumNumberOfImages = _size;
       request.remainingNumberOfImages = _size;
       request.perImagePrice = _ETHPerImg; //there is no double so
           coins must have soo many zeros
       request.ownerOfDatabase = msg.sender;
       request.completed = false;
       uint a = requests.push(request) - 1 ;
       requests[a].idOfRequestedDatabase = a;
       databaseIdToOwnerOfDatabase[request.idOfRequestedDatabase] =
            msg.sender;

       //emit the NewRequestIsDone event
       emit NewRequestIsDone(_name, request.idOfRequestedDatabase,
           request.perImagePrice);
   }

   //Returns current tags which are outstanding waiting for
       completion
   //Tag string is in same order of adding therefore ordering is
       important.
   function getNameOfAvailableTags() public view returns(string
       memory currentTags, uint[] memory currentTagIds){
       uint currentLength = requests.length;
       uint reducedLength = 0;

       //string memory currentTags;
       for(uint ii= 0 ; ii < currentLength; ii++){
           if(!requests[ii].completed){
               string memory currentString = string(abi.encodePacked(requests[ii].nameOfTag , string(","))) ;
               currentTags = string(abi.encodePacked(currentTags, currentString));
               reducedLength++;
           }
       }
       uint curInd = 0;
       currentTagIds  = new uint[](reducedLength);
       for(uint ii= 0; ii < currentLength; ii++){
           if(!requests[ii].completed){
               currentTagIds[curInd] = requests[ii].idOfRequestedDatabase;
               curInd++;
           }
       }
   }

   //handles the upload of Image Proposal
   function uploadImageProposal(uint _hash, uint _idOfProposedTag) public {
       ImageProposal memory proposedImage;
       proposedImage.hashOfImageProposal = _hash;
       proposedImage.tagIdOfProposal = _idOfProposedTag;
       proposedImage.uploaderAdress = msg.sender;
       proposedImage.numberOfTotalReceivers = 3;
       proposedImage.numberOfReceiversLeft = proposedImage.numberOfTotalReceivers; //It is value of receivers that judges
       uint a = proposedImages.push(proposedImage)-1;
       proposedImages[a].idOfImageProposal = a;
       proposedImage.countFalse = 0;
       proposedImage.countCorrect = 0;
       emit NewImageIsUploaded(proposedImage.hashOfImageProposal, proposedImage.tagIdOfProposal);
   }

   //when called it selects an uploaded image starting from
   function retrieveAnImageFromProposal() public returns(uint hashImage, string memory tagName){

       require(proposedImages[currentProposedImageIndex].numberOfReceiversLeft > 0);
       //check whether the sender is the uploader of next image or not
       require(msg.sender != proposedImages[currentProposedImageIndex].uploaderAdress);
       // check whether this address is already voted or downloaded the image before

       //Check the proposed Tag Id so that it is for a incomplete database.

       //add this address to receivers
       proposedImages[currentProposedImageIndex].receiverAddresses.push(msg.sender);
       hashImage = proposedImages[currentProposedImageIndex].hashOfImageProposal;
       tagName = requests[proposedImages[currentProposedImageIndex].tagIdOfProposal].nameOfTag;
   }

   //vote for an image whether the uploaded image fits to the requested tag
   function voteForImage(uint vote) public {

       //check whether receiverAddresses is not empty
       require(proposedImages[currentProposedImageIndex].numberOfReceiversLeft > 0);
       require(msg.sender != proposedImages[currentProposedImageIndex].uploaderAdress);
       // check whether messenger address is in the list of image receivers
       uint len = proposedImages[currentProposedImageIndex].receiverAddresses.length;
       bool existenceInReceiverSide = false;
       for(uint ii = 0; ii<len; ii++){
           existenceInReceiverSide = proposedImages[currentProposedImageIndex].receiverAddresses[ii]== msg.sender;
           if(existenceInReceiverSide== true){
               break;
           }
       }
       require(existenceInReceiverSide);

       //decrease the numberOfReceiversLeft
       proposedImages[currentProposedImageIndex].numberOfReceiversLeft--;

       if(vote == 1 ){
           //vote is assigned as correct
           proposedImages[currentProposedImageIndex].votersForCorrect.push(msg.sender);
           proposedImages[currentProposedImageIndex].countCorrect++;
       }
       else if(vote == 0){
           //vote is assigned as incorrect
           proposedImages[currentProposedImageIndex].votersForWrong.push(msg.sender);
           proposedImages[currentProposedImageIndex].countFalse++;
       }

       //Threshold until uploaded image counts as correct
       uint thresholdCorrect = proposedImages[currentProposedImageIndex].numberOfTotalReceivers / 2 + 1;
       uint thresholdFalse = proposedImages[currentProposedImageIndex].numberOfTotalReceivers - thresholdCorrect + 1 ;

       //Check for the correctness of the vote
       if(proposedImages[currentProposedImageIndex].countCorrect >=
            thresholdCorrect){
           //Add the proposedImage to the database corresponds to
               idOfProposedTag
           addCurrentImageToDatabase();

          address payable[] storage participants = proposedImages[
              currentProposedImageIndex].votersForCorrect;

          //anyone who voted right is put into a list
          participants.push(proposedImages[
              currentProposedImageIndex].uploaderAdress);

          //Reward everyone in this list with a reputation token
              and ether.
          for (uint i =0; i < participants.length; i++){
                  members[participants[i]].RepBalance++;
                  participants[i].transfer(IMAGEVALUE/participants.
                      length);
              }

          //deduct one reputation token for a false answer
          address[] memory participantsWrong = proposedImages[
              currentProposedImageIndex].votersForWrong;

          for( uint i=0; i< participantsWrong.length ; i++){
                  if(members[participantsWrong[i]].RepBalance >0)
                      members[participantsWrong[i]].RepBalance--;
              }

          //do the transaction to the vote for the correct
              receivers and uploader.

          deleteCurrentProposedImage();

       }else if(proposedImages[currentProposedImageIndex].
           countFalse >= thresholdFalse){

           //Discard the proposedImage as a false data and ignore
               it
           deleteCurrentProposedImage();

            address[] memory participants = proposedImages[
                currentProposedImageIndex].votersForWrong;
           //anyone who voted right + the uploader is rewarded

           //reward everyone in this list with a reputation token
               and ether.
           for (uint i =0; i < participants.length; i++){
               members[participants[i]].RepBalance++;
           }

           //deduct one reputation token for a false answer
           address payable[] storage participantsWrong =
               proposedImages[currentProposedImageIndex].
               votersForCorrect;
           participantsWrong.push(proposedImages[
               currentProposedImageIndex].uploaderAdress);

           for( uint i=0; i< participantsWrong.length ; i++){
               if(members[participantsWrong[i]].RepBalance >0)
                   members[participantsWrong[i]].RepBalance--;
           }

       }
   }

   //After True voting, the image is added to the database
   function addCurrentImageToDatabase() private{
       uint tagId = proposedImages[currentProposedImageIndex].
           tagIdOfProposal;
       uint imgAddress = proposedImages[currentProposedImageIndex].
           hashOfImageProposal;

       //check the number of remaining number of images
       require(requests[tagId].remainingNumberOfImages > 0);

       //add the image to the corresponding database
       requests[tagId].db.push(imgAddress);
       requests[tagId].remainingNumberOfImages--;

       //if the task is complete the database is removed from the
           visible tags
       //emits database complete signal
       //makes the database obtainable by the user
       if(requests[tagId].remainingNumberOfImages == 0 ){
           requests[tagId].completed=true;
       }
   }
   function deleteCurrentProposedImage() private{
       currentProposedImageIndex++;
       //if (currentProposedImageIndex >= proposedImages.length){
       //    deleteAllProposedImages();
       //}
   }
}
```

6.2 Commitment Demo

```
pragma solidity ^0.4.23;

contract CommitmentTest {
    enum Stage {
        Commit,
        Reveal,
        Finished
    }

    event NextStage(Stage);
    event LogEvent(bytes32);

    Stage public stage = Stage.Commit;
    uint voter_count = 0;
    uint constant MAX_VOTER_COUNT = 1;

    modifier atStage(Stage _stage) {
        require(
            stage == _stage,
            "Function cannot be called at this time. Wrong stage."
        );
        _;
    }

    function nextStage() internal {
        stage = Stage(uint(stage) + 1);
        emit NextStage(stage);
    }

    modifier transitionWhenFull() {
        _;
        if (voter_count == MAX_VOTER_COUNT)
        {
            nextStage();
            voter_count = 0;
        }
    }

    mapping(address => bytes32) commited_votes;
    mapping(address => string) revealed_votes;

    function commit(bytes32 hashedStr) public
    atStage(Stage.Commit)
    transitionWhenFull
    {
        commited_votes[msg.sender] = hashedStr;
        voter_count += 1;
    }

    function reveal(string revealedStr) public
    atStage(Stage.Reveal)
    transitionWhenFull
    {
        emit LogEvent(bytes32(commited_votes[msg.sender]));
        emit LogEvent(bytes32(keccak256(revealedStr)));
        require(commited_votes[msg.sender] == keccak256(revealedStr
            ),
            "CommitmentTest::reveal::Commit does not match reveal"
        );
        revealed_votes[msg.sender] = revealedStr;
        voter_count += 1;
    }

    function getCommit() public view returns (bytes32) {
        return commited_votes[msg.sender];
    }
    function getReveal() public view returns (string) {
        return revealed_votes[msg.sender];
    }

    // for client side:
    // https://web3js.readthedocs.io/en/1.0/web3-utils.html#
        soliditysha3
    // (you certainly would want to compute this on the client side
        , it just isn't correct yet)
    function hash(string str) public pure returns (bytes32) {
        return keccak256(str);
    }
}
```