Semester Project - P2P insurance supported by the blockchain

Hugo MOREAU and Paul NICOLET
{hugo.moreau ; paul.nicolet}@epfl.ch, EPFL
Laboratory For Cryptologic Algorithms (LACAL)
Under the supervision of Arjen LENSTRA and Benjamin WESOLOWSKI

May 5, 2017

Contents

Introduction

Insurance companies greatly suffer from insurance fraud. It seems to be explained by the fact that insurance companies and users have conflicting interests. It's no rocket science that insurance companies need to be profitable and therefore have a natural tendency to try to reduce the bill when it comes to reimbursing a customer. On the other hand, customers might have the feeling that they already contribute enough through their regular insurance fees to be covered when they need to. Therefore, it appears that as they feel a bit cheated by the company they react by themselves trying to cheat the company as well.

In order to go against that problem, the principal intuition would be to align insurance goals with users goal. By creating a company held and governed by the users such a compromise can be found.

Two problems remain :

- What kind of insurance is suitable for such a framework ?
- How to create such a community with a fair repartition of trust and power ?

First of all, we considered car deductibles to be the most suited value to insure. A deductible may be used to describe one of several types of clauses that are used by insurance companies as a threshold for policy payments. In the case of car insurance they are frequently used for responsible crash in the case of an "all-risks" insured car. The driver always has to pay from his own pocket the deductible part: if an insurance policy has a deductible of 100$ then all crashes requiring repairs costing less than 100$ won't benefit from any insurance help while a 400$ repair cost will receive a 300$ help from the insurance. This was considered as a great framework since it allows to ensure that the value insured per user remains bounded and relatively low. Also it would allow to have a consistent value to insure among users: most car insurance have the same value magnitude when it comes to deductibles.

Then, in order to create such community we need to have a common fund managed by all users. This is quite complicated to implement: how to coordinate multiple users without the help of a central referee[1]? This justifies the use of the blockchain. As it will be explained in later discussions, the blockchain technology provides a useful framework to create such a community.

Using such system, users could be insured by the community. All of them would contribute to a common fund and would vote whenever one of them needs a reimbursement. Every user is then implicitly encouraged to be careful on how he evaluates the legitimacy of each claim, including its own. This is due to the fact that once the common pot goes dry no user is insured anymore. Therefore for this pot to last as long as possible they need to carefully evaluate claims. Also, the statistics of the users may be used to show how often do they approve claims from other users so that when they claim a crash, voters can decide whether or not they play by the rule of the community, how generous they are with there votes and how often do they happen to ask for a reimbursement.

We will first explain the model that we created for such a community. Then different alternatives to implement the system will be described and all choices will be justified. Finally some implementation details and difficulties will be exposed to raise awareness about some limitations.

[1] The use of a referee is strongly discouraged given the fact that the goal of the system is to be user-owned and driven

Chapter 1

The System Model

This section presents the different underlying theories that support the system. It will also try to give the different solutions that were imagined and arguments to accept or reject them, when possible.

1.1 Claims

The project is about insuring members of the community. In order for a user to pretend to a reimbursement from the community, he needs to submit a **claim**. Each claim only concerns the member of the community at the time it is posted, i.e., a member votes and pays only for claims published at a time they belonged to the community. Each claim is submitted along with :

- Debating period : the amount of time voters have to decide.
- Amount of the claim : the amount of money requested by the claimer.
- Description : a short explanation about the claim.
- Documents information (*not yet implemented*) : URLs and hashes of official documents used to support the claim. The need for such information will be made clear later on.

1.2 Funding

This paragraph tries to explain how the system is funded. The basic idea is to have each user deposit at launch of the community a fixed amount of money which will be used to refund claims until the community cannot support claims anymore (this condition will be further described in the following definition). The money funded by each user is kept in his virtual **balance**.

Once the community cannot support a new claim anymore, a refunding is initiated. Users have the choice to either leave or stay in the community. If they decide to stay, using the statistics from the end of the last funding to the current date, called the **inter-funding period**, an estimated amount of claim per user in a year[1] is computed. Using the latter and the average price of a claim a new funding amount per user is computed. Mathematically it expresses as follows :

A **funding** is a period during which users of the community must contribute to common funds used to reimburse potential claims. This period has a given duration `FUNDING_DURATION` (denoted d_f in math equations). A user is insured only once he has contributed to the current funding. In this representation we will always denote by k the index of such funding. Therefore we will denote by t_k the time at which the funding starts. A funding period lasts some time in order for the user to be able to pay the designated amount.

The funding k happens for $t \in [t_k; t_k + d_f]$. We denote by T_k the amount that will be in every mem-

[1] The period for which the estimate is computed can be tuned in order to be realistic and depending on further implementation choices.

ber's balance at the end of the funding[2], that is, at $t = t_k + d_f$.

Let $x_{i,t}$ represent the balance for user i for $i \in \{1, ...n_t\}$ at time t, where n_t represents the number of users in the community at time t

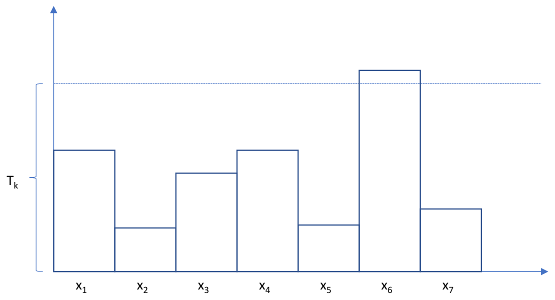

Figure 1.1: Balances at time t_k

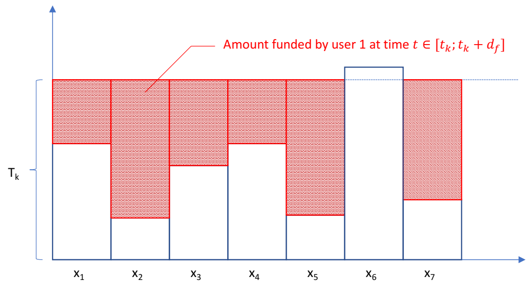

Figure 1.2: Balances at time $t_k + d_f$ when no user has joined during funding period

In Figures 1.1, 1.2 and 1.3, we characterise what is T_k. Note that the time at which the balances are considered is precised in the caption, so that we do not overload the graphic with the extra subscript. Fig. 1.1 shows the situation at the time where the funding is required. While Fig. 1.2 explains how each user tops-off its balance in order to stay in the community in the case where no user arrives between t_k and $t_k + d_f$ (which is explained in Fig. 1.3). An extra rule that can be observed is that user 6 already has a balance over T_k, therefore he doesn't need to top off.

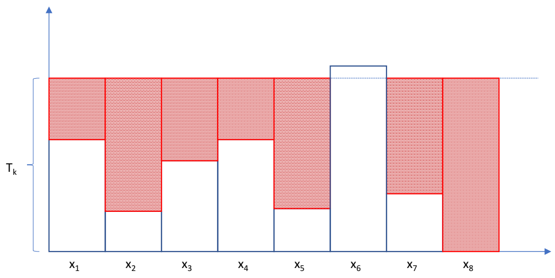

Figure 1.3: Balances at time $t_k + d_f$: user 8 has joined during the funding period

1.2.1 Initial Balance Value : $x_{i,t}$

- Genesis of the system: let P_{hist} be the average price of a claim and C_{hist} be the average number of claim per user in a year[3] based on historical data. Then :

$$\forall i \in \{1, ...n_0\} \quad x_{i,0} = C_{hist} \times P_{hist} = T_0 \quad (1.1)$$

- Upon arrival of a new user: Suppose a new participant arrives at time t_a. Then he will pay the same amount than the top-off value that had to be reached by other users upon last funding. To avoid mathematical difficulties denote by k the index of the funding that was initiated at t_k and suppose that there was no other funding initiated between t_k and t_a. Then if we denote by j the index of the new user:

$$x_{j,t_a} = T_k \qquad (1.2)$$

1.2.2 Condition Upon Which a New Funding is Required

While we first didn't consider that having a user with a negative balance would be a problem, it turns out

[2] Note that if a user hasn't paid by the end of the funding period he will not be considered as a member anymore.

[3] This number will have to be a bit exaggerated in order to cope with the possible differences between historical rate and observed rate.

that this makes it difficult to handle what to do when such user decides to leave the community. It seems clear that we cannot reach a point at which one user balance would become negative after reimbursing a claim.

Suppose t_{new} is the current time and corresponds to a new claim being added. We need to ensure that at this time the balance of every user can reimburse all claims that are pending as well as this new claim that is submitted. In Fig.1.4, one can observe that the current balance of the user depends on the number of user at the time of the claim, and on the price of the claim. In the simple example offered by Fig. 1.4, the condition that is checked at time t_{new} is :

$$\forall i \in \{1, ...n_{t_{\text{new}}}\} \quad x_{i,t_{new}} - \frac{p_1}{n_{t_1}} - \frac{p_2}{n_{t_2}} > \frac{p_{\text{new}}}{n_{t_{\text{new}}}} \tag{1.3}$$

where p_i denotes the price of the i$^{\text{th}}$ pending claim and t_i the time at which it was submitted.

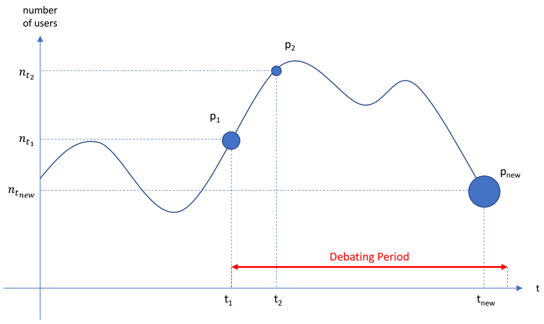

Figure 1.4: Illustration of the funding condition

In this example we see that we need to check that once each user has potentially reimbursed his share of the first two claims he will still be able to reimburse his share of the last claim.

It's important to note that by using the previous condition we are actually checking something stronger than we actually need : e.g. a user that arrives after t_1 isn't concerned by the first claim but we still check that he has enough to reimburse it on top of other claims. But if all balances of users that are concerned by claim 1 pass the condition then this latter user will also verify it. This is due to the fact that his balance is greater or equal to the member with the least amount in his balance (due to the computation of the top-off value). The reason we do not only check for users concerned by claims is to have a more systematic check that will turn out to be less computationally expensive.

1.2.3 System Behavior During a Funding Period

There are multiple ways to treat the claims which triggered the funding:

- The claim is still accepted and its deadline is extended by the funding duration. This is the best solution for the claimer's point of view, but this would not be very useful since we don't want users which did not fund to be able to vote. So the claim would stay idle during the whole funding period. Moreover, this would imply that if users choose to not fund the community, the amount needed for the claim might not be reached, whereas the claim would be active, which can lead the community in an inconsistent state.

- The claim is buffered until the end of the funding period. This solves the previous point since the claim is only instantiated when funding is finished. However, we can't provide the claimer with any information about the creation date of its claim. Indeed, even if the funding period has a defined deadline, the action of finishing the funding will not be taken until users show some activity on the community after this deadline[4]. Then the creation date of the claim is unknown to the claimer and can lead to confusion.

- The best alternative we thought of is the simplest: reject the claim. The claimer would be

[4] This is an implementation detail. We will see later on that there is no way to make scheduled calls in solidity. Therefore we use user interaction with the contract to check if we need to trigger a new action.

informed that the community is currently unable to support his claim, and that he should try again after the funding period.

Naturally, it should not be possible for users to deposit a claim during a funding period, since the size and funds of the community are in a transient state. However, we decided that it should still be possible to vote and execute pending claims, which had been instantiated before the current funding period. Indeed, these claims can be supported by the community, and we are guaranteed that users have the necessary amount in their balance. Therefore we only pause submission of new claims.

1.2.4 Computation of top-off value at each funding : T_k

One must keep in mind that the system must adapt to the long term and short term community. To do so, we will keep track of a **virtual user**. This user will save the amount paid by a user during each inter-funding period. Therefore we will have three state variables :

1. `PaidSinceLastFunding` (denoted `PSLF` in equations) which keeps track of what is paid by the virtual user since previous funding.

2. `PaidYearlyAcc` which accumulates the sum of all yearly payments computed from each inter-funding period.

3. `PaidYearly` which keeps track of what is paid by the virtual user between day 0 and current time on a yearly basis.

The update rule of those three variables are quite intuitive. Every time the community approves a claim and it gets reimbursed, the amount paid by a user is added to `PaidSinceLastFunding` This variable is reinitialised at the begining of each inter-funding period. Once a new funding k is initiated :

$$\texttt{PaidYearlyAcc}+ = 365 \times \frac{\texttt{PSLF}}{\texttt{days(}t_k - t_{k-1}\texttt{)}}$$

Then we can obtain the third one simply by :

$$\texttt{PaidYearly} = \frac{\texttt{PaidYearlyAcc}}{\#\text{revolved inter-funding period}}$$

This approach allows for `PaidYearly` to reflect what a user pays on a yearly basis depending on the historical community as this will average over each inter-funding period. Also it entirely isolates the variation of the number of user during inter-funding periods since the amount of the claim is divided among the number of members at the time the claim is published.

We compute the top-off value per user:

$$T_k = (\alpha \times \texttt{PaidYearly} + (1-\alpha) \times \texttt{PSLF}_{k-1}) \quad (1.4)$$

where $\texttt{PSLF}_{k-1}$ is the amount paid by the virtual user during the preceding inter-funding period.

Since T_k is the amount that a user must reach once they have topped off, users' funding amount is computed using the **Water-filling** solution as explained in Fig. 1.2 and 1.3 s.t.

$$\forall i, j \quad x_{i,t_k+d_f} = x_{j,t_k+d_f} = T_k \quad (1.5)$$

The reason for the weighted average in Equation 1.4 is due to the need for the system to dynamically adjust to the historical behavior since the beginning of the community as well as the possible recent changes in the community composition.

1.2.5 Incentive to Fund

Given the fact that the funding period will be made quite long we decided to exclude all users that haven't contributed to the common fund by the end of the period. This will cause the rest of their balance to be kept by the community.

1.2.6 Launching a new funding

In order to avoid the attack described in Section 3.3, we decided to never launch a new funding if the users all have enough money in their balance. That is if the new top off value is already reached by all users.

1.3 Voting

1.3.1 Fundamental Behavior

In order to determine whether a claim should be funded by the community or not, a vote is organised. It is not clear at first how to avoid having the user vote for every single claim but this behavior is required in order to be able to scale to a large community. The most appropriate solution resides in **random sampling**. On the same principle than jury duty in the United States, each claim will be voted by a small randomly selected sample of the users. The number of such users has to be defined and should heavily depend on the size of the community. Options that are available :

1. Relative : always consider that the jury to be representative of the community should be a given fraction of its members. But this is wrong for a big enough community as only a sample of a given size is necessary in order to achieve a given error margin (see following paragraph).

2. Absolute : Simply consider that the critical mass (see further explanation of the critical mass) is enough to decide and that the number of voters is fixed for any community size. This could turn out to be a good solution but would force us to have a large critical mass and therefore to wait for a given number of pre-registered users to launch the community.

3. Combination of both : use a threshold, where a relative value is used under this threshold and then an absolute value is used. It would ensure that the bigger the community the less frequently people have to vote. So far, this solution is the one we have considered.

1.3.2 Error Yielded by Random Sampling of Voters

In the following we will follow an article by Terence Tao [3]. We will also assume ideal conditions of sampling.

Assumptions: While the article needs to assume a lot of different perfect behavior, this is not necessary here. In fact we will explain why the only assumption we need to make is Simple random sampling.

- Simple question : voters are offered only two possible answers (accept or reject claim)

- Perfect response rate: all the following computation will be computed considering the size of the sample to be equal to to the minimum quorum and therefore the issue of the vote will not be considered if it isn't reached

- Honest Responses : we are not doing a poll but real vote therefore it is safe to assume that voters will vote following their real voting intentions.

- Fixed Poll Size : before doing the vote we will always use the same minimum quorum.

- Honest Reporting : From the source code of the system one can see that the result of the vote are always reported and used as they are.

Now the only remaining assumption is **Simple Random Sampling without replacement**. This means that each of the n sampled voters have been selected uniformly at random among the entire population independently of others except from the fact that we do not allow replacement. That is once a voter has been selected it can not be selected a second time [5].

Let X denote the set of the total population. Let A be the set of voters that accept the claim. We wish to compute the margin of error [7] (at the confidence interval 95% [5]). Let p be the proportion of voters that accept the claim:

$$p = \frac{|A|}{|X|} \tag{1.6}$$

Now suppose that we sample n users of the community to be representative of the the entire community

[5] It is important to note that by allowing replacement once can obtain joint independence and use Chernoff inequality to further reduce the margin of error.

(under the assumptions described above). Then we will have, $\overline{p}$, the proportion of sampled voters that vote to approve the claim :

$$\overline{p} = \frac{|1 \leq i \leq n : x_i \in A|}{n} \tag{1.7}$$

Finally using Markov's inequality,

$$\forall r > 0 \quad \Pr(|\overline{p} - p| \leq r) \geq 1 - \frac{1}{4nr^2} \tag{1.8}$$

Therefore if we compute a confidence interval of 95% we want :

$$\Pr(|\overline{p} - p| \leq r) \geq 0.95 \tag{1.9}$$

It is sufficient to show :

$$\begin{aligned} & 1 - \frac{1}{4nr^2} \geq 0.95 \\ \Rightarrow \quad & r = \frac{1}{\sqrt{4n(1-0.95)}} \\ \Rightarrow \quad & \Pr(|\overline{p} - p| \leq r) \geq 0.95 \end{aligned} \tag{1.10}$$

hence we compute the margin of error r depending on different values of n (the minimum quorum).

Minimum Quorum (n)	**Margin of error (r)**
200	15.811%
500	10.000%
1000	7.071%
5000	3.162%
$10'000$	2.236%

Table 1.1: Margin of error for different Minimum Quora

Choice of Minimum Quorum : it can be observed from Fig. 1.5 that we have a clear tradeoff between how well can we represent the community and how many people do we need to sample. We considered that choosing $n = 1000$ yielded a small enough margin of error.

Sampled size We will see later on that due to our incentive to vote, we need to sample more users than Minimum Quorum since we cannot assume that all the sampled users will vote every single time they are selected.

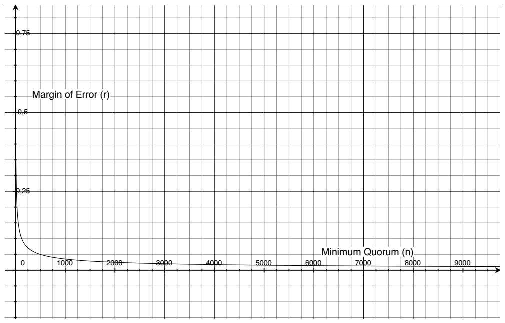

Figure 1.5: Tradeoff between the margin of error and minimum quorum

1.3.3 Incentive to Vote

It is important to note that we have not yet defined the incentive for users to vote but this is a very important notion. The whole community relies on voters:

- At first we considered that if a user selected for a given vote doesn't vote before the end of the debating period then he would be excluded from the community. But this seemed as a very unfriendly behavior for users.

- Finally it was decided to have a system of points. A given number of points are issued to each user[6]. Then whenever a voter fails to vote before the deadline, a point is taken away from him. Once a user reaches zero points he is excluded. Those points will reset after a given number of successful votes[7]. In order to implement such strategy, we introduce the notion of **Minimum Quorum**, which denotes how many voters from the randomly sampled users must vote in order for the voting result to be considered valid.

Once a user is excluded from the community he cannot vote, or submit any claim and he would have lost

[6] The number of points is considered as an implementation decision.

[7] Again this number doesn't really affect the theory.

all the abilities that were provided to him based on his fundings.

1.3.4 Critical Mass

Using the previous results on the size of the minimum quorum, we can determine what is the critical mass we wish to have. Since users are allowed to not vote some given number of times, we need to make sure that we are quite confident that among the sampled users, we are quite confident that we will reach the Minimum Quorum at the issue of the vote. It is quite intuitive using the expectation of a binomial distribution[8], that if each user doesn't vote approximately half of the time then the expecting number of selected users that will vote is close to half of them. Then we decided that choosing a Critical Mass of 2000 users would be significant to reach the minimum Quorum with high probability.

1.4 Leaving The Community : Handling Left-over Funds

When members leave the community willingly or not, it is possible that some amount of money remains in their balance. We need to define a policy about how these left-over funds are handled. First of all, they should not be directly considered as "free" funds because a user remains liable for all claim that have been submitted during his membership. Therefore we need to keep track of the money that is currently not usable because it will eventually be used to reimburse a claim and the money that is currently usable by the community to reduce the amount paid by each user to reimburse a claim.

We therefore introduced :

- `jackpot` a global variable that will count the amount of usable money. This money will be used to decrease the amount paid by users upon reimbursing a claim. If the `jackpot` is non zero when a claim must be reimbursed, it is entirely used to subtract from the claim amount before computing what the users must pay from their own balance.

- `leftOvers` a variable stored for each claim. Whenever a user leaves, its balance will be split. For each pending claim he was supposed to pay for, the amount he was suppose to pay will be taken from his balance to the claim `leftOvers`. Either the claim gets approved, in which case `leftOvers` are forgotten, or it turns out the claim does not get approved and `leftOvers` goes to the `jackpot`.

This money is only a virtual counter, it doesn't reveal the real amount of money that is held by the contract. In fact, rounding will often leave a very small amount of money which isn't accounted by the balances, but this amount will be only a very small fraction and is considered to be non-relevant (at least until the possible end of the community as it will be seen later on.)

1.5 Anonymity and Verication of Claims'Documents

1.5.1 Anonimity

The very first issue that came to our mind was **anonymity**. The blockchain technology relies on a public ledger, which means that any claim would be visible by anyone even-though they are not members of the community. Therefore, in our use (that is : car deductible), it might be desirable to have some sort of anonymity. But here are the reasons why we rejected the idea of having anonymous users :

- We need to be able to know what car is insured to avoid having a user paying only one registration and insuring multiple vehicles. To identify the vehicle we need a license plate and therefore will void any user anonymity.

- A central part of the vote will be the user profile, to keep statistics about the users (number of claims, age, percentage of positive votes...),

[8] this supposes that the simple random sampling is done without replacement. Even though it is not the case, it won't hopefully be too far from the reality

therefore if we do not identify the users, then those statistics can be reset as many time as the user wants.

- End users need to be able to verify that documents linked to the claim are legit. In the standard insurance it is usually required to have a police report of the crash and/or some witnesses. Those documents are heavily tied with the victim's identity. It would be hard to certify the documents without linking them to a person.

Therefore, judging by the risk at stake we thought that the fact that the system doesn't ensure anonymity isn't such a problem. We couldn't see many use cases where someone could wish to hide from the public an accident but not from its insurance company. Moreover, where regular insurance companies can decide to have the car inspected for damages in the event of a crash, the DAO doesn't have such "experts". Therefore it heavily relies on the documents that prove the crash existence and the profile of the claimer.

1.5.2 Document Verification

TODO : here we will explain how do we plan on verifying the documents (storage, hashing etc)

1.6 Community Suicide

We have seen so far that we need to have a critical mass for the system to be able to work. But then we also always need to check that we won't go below this lower limit once the system is online. Moreover, what should be the policy once we go below such limit ? We decided to **suicide the actual community**. In such scenario we would :

1. Take all remaining users and subscribe them to preregistration again. *(Not implemented yet : so far we simply kill the community forever)*

2. Reimburse the current amount of money contained in each user's virtual balance.

3. As we previously mention those balances do not represent the real amount of money held by the system. Therefore a user will be randomly selected[9] to receive all the eventual remaining funds of the system.

4. Delete all the current community to start from scratch if critical mass of preregistration is reached again in the future.

The process by which the user is randomly selected will be further described in Section 2

[9] To avoid targeted attacks

Chapter 2

Technical Alternatives Considered

Since many alternatives where considered when designing the system, it was decided to present the ”trial and error” protocol that was observed. This will hopefully explain decisions that were undertaken and there deep correlation with our line of thoughts. That is, each topic of the system will be presented depending on when the question showed up during the project development.

2.1 Using Bitcoin with Distributed Wallets

Given the financial aspect of the project, the first technology that presented itself as an evidence was to use Bitcoin.

2.1.1 Principle of Bitcoin

This paragraph presents a very simplified and not comprehensive explanation of the Bitcoin system. Bitcoin is a cryptocurrency where transactions take place between users without the need of a trusted third party. The blockchain represents a public ledger where transactions are recorded after they have been verified by nodes of the network. Transactions are grouped into blocks, forming the well-known Blockchain. This activity of verifying the transactions and adding them to blocks is called **mining**. Bitcoins are created as a reward for a competition into which miners try to constitute blocks with new transactions, and add them to the Blockchain. In order to be the winner and publish a block, a miner must perform a **Proof of Work**. This is what makes mining a hard task. It consists in incrementing a number until the hash of the block verifies some predefined criterias. In order to cheat the blockchain, since all blocks are linked to each other using hashes, one need to modify all subsequent blocks and recompute all proofs of work. Therefore, as the number of blocks increases, it becomes harder to modify a past block. This is how transactions don’t need to be verified by an intermediary. The network always consider the longest chain to be the valid one.[4]

2.1.2 Distributed Wallets

The collaborative funding aspect of the project meant that a common fund **managed by the community** needed to be created. Therefore our reflexion was oriented towards **distributed wallets**. Those wallets, to be opposed to regular Bitcoin wallets, use a multi-signature scheme. This enables to have N-out of-M participants controling the wallet. Hence, to be valid, a transaction needs to be signed by at least N out of the M wallet cosigners. (It is important to note that current multi-signature schemes are limited to $M = 15$ [1, p. 127]). These distributed wallets allow to have each user’s deposit protected by the community (or at least a portion of it). The cosigners would be randomly chosen to protect a deposit. Therefore no deposit can be withdrawn from the community without a consensus among at least N of the cosigners.

2.1.3 Problems with this alternative

Many problems were discovered about the solution using distributed wallets.

1. **Collusion** : if N users out the M cosigners decide to steal the deposit : nothing protects the community from doing so. The assumption that they won't is very weak as the maximum amount of cosigners is 15.

2. **Excluding users** : given the fact that the incentive to vote is to avoid exclusion from the community we need to ensure that when excluding someone from the community our protocol is well defined. But if more than $M - N$ cosigners get excluded from the community then access to the wallet is lost and the community suffers from this loss.

3. **Need actions from the users to unlock the money** : upon approval of a claim we need to unlock the money from each distributed wallet to reimburse the claimer. But suppose most cosigners on a wallet disapproved the claim. Then, they could decide not to unlock the wallet even though the community approved.

4. **Increased price of a claim reimbursement** : By definition of the blockchain, each transaction has to be mined and therefore each transaction has some extra fees to pay the miners. With one distributed wallet per user, that means that each reimbursement involves a transaction from each distributed wallet to the claimer. That means a lot of extra fees (that might even go beyond the reimbursement in itself making the system useless.)

2.1.4 Conclusion

At the sight of the previous problems encountered, it was decided to let go off the idea of distributed wallet. Nevertheless, much of the theoretical approach to the problem is kept and transposed to the next technical alternative.

2.2 Using Ethereum smart contracts to handle the common funds

Previous alternative demonstrated the difficulty of setting up a community fund using Bitcoin. This is where Ethereum came into play.

2.2.1 Principle

In Bitcoin, spending a UTXO requires to sign it with one's private key, to use the multi-signature scheme or finally to write small locking scripts that can only be unlocked by the corresponding unlocking scrips. But the simple stack based Bitcoin scripting language isn't Turing complete. Hence it is not possible to write a complex piece of code that executes whatever function is required for a specific use case.

Ethereum would theoretically allow to write complex script for our project. The idea is to create, using smart contracts, a Decentralized Autonomous Organization.

Ethereum is a blockchain-based distributed computing platform. It provides a virtual machine (EVM). This virtual machine can execute scripts using an international network of public nodes. Ethereum also provides a value token called "ether", that can be thought of as a bitcoin. Then in order to pay for computational power the notion of "gas" was introduced. This internal transaction pricing mechanism, is used to prevent spam on the network and allocate resources proportionally to the incentive offered by the request. [6]

It is composed of two kind of nodes :

- Smart contract : the guardian of the fund. It will represent the community by gathering the votes, creating the transactions when necessary to reimburse a claimer etc.

- EOA : externally owned accounts. Those are the end users. They will invest in the DAO's wallet, vote whenever they have to, and post claims.

2.2.2 Storing documents

We need to store documents related to each claim and contract, such as pictures of the car, police crash declaration, and any other documents that can help the community decide whether or not to reimburse the claim. Therefore we need to store these documents. Three alternatives were considered:

- Storing the documents in the contract storage : this turned out to be a very bad idea. To make it simple in February 2016 each GigaByte of data cost 76'000$ [2]. The only advantage of this would be that each claim document cannot be modified after the claim and would be available in the event of a new claim by the user to check multiple characteristics, such as checking if this is the same car or accident.

- Store the documents on a community paid server. This solution was the simplest for the end user because it would allow to offer in the application a "drag and drop" platform to store the documents. But this seemed a bit complex to set up and was not in the scope of our project.

- Finally, the option that remained was to have each user use its favorite online data storage platform. On the contract storage would only be stored a link to the archive containing the document and a hash of the archive. The hash would serve as a proof that the user doesn't alter the documents during the vote. But it will also store forever the fingerprint of the archive in the blockchain allowing voters to ask a user to provide the archives of previous claims in order to make the sub-mentioned verifications.

The third solution appeared to be much more scalable to a large community given that each user handles itself this aspect of the system. One of the drawbacks is that not all users will be familiar with any services of this type. This could be documented in the final system. Moreover, given the "community" aspect of the system, it could have a community forum where non expert users could ask for help.

Chapter 3

Implementation

Given the now well defined concept we decided to start the implementation and leave the remaining zones of shades for later.

3.1 Development Set-Up

Solidity was first proposed in 2014 and Ethereum was first released in 2015 [6]. This means that documentation is still quite sparse and that the development tools are not yet very established.

The main problem with Ethereum development is that testing on the live network is quite discouraged by the inherent costs. Both in term of time (one need to wait for miners to execute transactions) and in term of gas (one need to pay the miners for such computing power). Therefore we used widely available tools:

- `testrpc` : a *Node.js* based Ethereum client for testing and development. It uses *ethereumjs* to simulate full client behavior and make developing Ethereum applications much faster. It also includes all popular RPC [1] functions and features,like events, and can be run deterministically. It can be thought of as a ”local blockchain”.
- `truffle` : a development environment, testing framework and asset pipeline for Ethereum, aiming to make life as an Ethereum developer easier. In particular it allows to compile smart contracts, but also to test them using the *Mocha* and *Chai* testing frameworks. It also helped us get started with Solidity as it provides basic contracts along with their migration script[2], and their testing files.

Then we again quickly encountered some implementation issues.

3.2 Scheduled Calls

A claim has a voting deadline as well as a minimum quorum[3]. Once either of those is reached, the claim must be executed. The problem is that we cannot really call code at a given point in time directly from the contract (*e.g. when the deadline is past*). Therefore we considered multiple techniques :

- `ethereum-alarm-clock` : an externally developed smart contract that allows scheduled contract function calls. This seemed at first like a good idea but some problems remained. It allows to schedule code at a given block number but not at an exact date and time. This could be overcome since blocks are mined at a quite regular interval. But then still, development using this tool would be very complicated given that the contract can only be accessed on the live blockchain.

[1] RPC: Remote Procedure Call.

[2] Code specifying how the contract is posted on the blockchain.

[3] The number of voters required in order for the vote to be considered valid.

- Rely entirely on the user : after all, the claimer wants the money therefore it's not too much of an assumption to say that he can take care of asking for the system to rule the claim once the deadline is past.

- Use the client side application : we can also consider the option where the client side application (when launched) checks the current pending claim of the user and there deadline and ask the contract to execute the ones that are meant to be finished.

3.3 Attack on statistics

3.3.1 Idea of the attack

While implementing the system we realised we might have a problem. Suppose an attacker wants to affect the value of `PaidYearly` by engaging a new funding before any claim has been executed (reimbursed) for the current inter-funding period. Then we see in equation 1.4 that $\texttt{PSLF}_k$ will be zero. Moreover, when a new funding is launched we update the values of `PaidYearlyAcc` and `PaidYearly`. Here `PaidYearlyAcc` won't be changed and `PaidYearly` will decrease as the number of revolved inter-funding period will increase.

3.3.2 Prerequisites

In order for this attack to take place, we need the following conditions:

- No money was spent so far (that is no claim has been executed yet) in this inter-funding period
- The attacker post one or more claim such that the system cannot support it/them.

Since no claim has been executed yet, if we suppose that last funding was the k-th one, then each user has T_k in its balance.

3.3.3 How to avoid such attack - Effectiveness

In order to avoid the attack, we previously described the fact that no funding is launched if the users already have enough money in their balance. We will try to show how this avoids the mentioned attack. We want to show that in such situation $T_{k+1} < T_k$. This will ensure that no funding is launched and therefore will prevent `PaidYearly` to be updated.

Since $\texttt{PSLF}_k = 0$:

$$\begin{aligned} T_{k+1} &= [\alpha \times \texttt{PaidYearly} + (1-\alpha) \times \texttt{PSLF}_k] \\ &= \alpha \times \texttt{PaidYearly} \end{aligned} \tag{3.1}$$

Moreover, suppose that the attack is being performed for the first time. Then $\texttt{PSLF}_{k-1} \geq 0$

$$\begin{aligned} T_k &= [\alpha \times \texttt{PaidYearly} + (1-\alpha) \times \texttt{PSLF}_{k-1}] \\ &\geq \alpha \times \texttt{PaidYearly} = T_k \end{aligned} \tag{3.2}$$

Now we need to show that if the attack happens at the very beginning of the community then the argument holds. At the the initialisation of the community, $\texttt{PSLF}_{-1} = \texttt{PaidYearly}$ (we do not have any information about the composition of the community and therefore we can only use an "historical estimate" of the yearly price). Then the initial funding top-off value would be:

$$\begin{aligned} T_0 &= [\alpha \times \texttt{PaidYearly} + (1-\alpha) \times \texttt{PSLF}_{-1}] \\ &= [\alpha \times \texttt{PaidYearly} + (1-\alpha) \times \texttt{PaidYearly}] \\ &= \texttt{PaidYearly} \end{aligned} \tag{3.3}$$

and :

$$\begin{aligned} T_1 &= \alpha \times \texttt{PaidYearly} \\ &\leq \texttt{PaidYearly} \end{aligned} \tag{3.4}$$

We have proven that for any k

Bibliography

[1] Andreas Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies.* O'Reilly, 2015.

[2] Ethereum Stack Exchange. *What is the cost to store 1KB, 10KB, 100KB worth of data into the ethereum blockchain?* accessed online on the 19th of March, available at `http://goo.gl/nRyi8K`. 2016.

[3] Terence Tao. *Small Sample and the margin of error.* [Online; accessed 02-May-2017]. 2008. URL: `https://terrytao.wordpress.com/2008/10/10/small-samples-and-the-margin-of-error/`.

[4] Wikipedia. *Bitcoin — Wikipedia, The Free Encyclopedia.* [Online; accessed 25-April-2017]. 2017. URL: `https://en.wikipedia.org/w/index.php?title=Bitcoin&oldid=777113800`.

[5] Wikipedia. *Confidence interval — Wikipedia, The Free Encyclopedia.* [Online; accessed 2-May-2017]. 2017. URL: `https://en.wikipedia.org/w/index.php?title=Confidence_interval&oldid=777204670`.

[6] Wikipedia. *Ethereum — Wikipedia, The Free Encyclopedia.* [Online; accessed 31-March-2017]. 2017. URL: `https://en.wikipedia.org/w/index.php?title=Ethereum&oldid=772283953`.

[7] Wikipedia. *Margin of error — Wikipedia, The Free Encyclopedia.* [Online; accessed 2-May-2017]. 2017. URL: `https://en.wikipedia.org/w/index.php?title=Margin_of_error&oldid=774851130`.