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 20, 2017

Contents

Introduction

Insurance companies greatly suffer from fraud. According to the Coalition against insurance fraud, fraud accounts for 5-10 percent of claims costs for U.S. and Canadian insurers. They estimate the loss for insurers across different business lines to be as high as $80 billion a year [5]. It seems to be explained by the fact that they have conflicting interests with their customers. It is 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 feel they are entitled to get full coverage given they pay their fees. Therefore, it appears that as they feel a bit cheated by the company and their natural reaction is to try to cheat the company as well (e.g. by declaring more damage than what was actually caused).

In order to go against this conflict of interest, the principal intuition would be to align insurance's goals with users'. 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?

Throughout our project, we considered car deductibles to be the most suited value to insure. A deductible may be used to describe one of the several types of clauses that are used by insurance companies as a threshold for policy payments. Hence we will consider that the community will reimburse deductibles. In the case of car insurance, they are frequently used for a responsible crash in the case of an "all-risks" insured car. The driver always has to pay from his/her own pocket the deductible part: if an insurance policy has a deductible of $100 then all crashes requiring repairs costing less than $100 will not 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 having a consistent value to insure among users: most car insurances have the same value magnitude when it comes to deductibles[1].

Then, in order to create such community we need to have a common fund managed by all users. It would be simple to implement using a trusted third

[1]Therefore, claim price might be different but will remain in the same order of magnitude.

party but how can we avoid using such a central referee. Most people will agree that trust is everything. Yet, so far, building trust online is a true challenge. The Blockchain tries to solve this issue by providing a simple way to enable collaboration among users. Built as a public ledger verified by multiple nodes of the network, it uses multiple cryptographic tools at different levels of its architecture in such a manner that it does not have to rely on a trusted third party. Those tools ensure that verifiers can be trusted, get paid for their work, but mainly it guarantees that the content of the ledger can be trusted at any point in time. This justifies why the technology was foreseen as the heart of the system, bringing trust among strangers but also the tools to realise financial transactions.

Using such a system, users could be insured by the community. All of them would contribute to a common fund and would vote whenever a member needs a reimbursement. One could wonder why users would be willing to pay an additional fee to cover their deductible : thanks to the large community, this fee would be relatively small compared to the potential deductible. Every user is then implicitly encouraged to be careful on how he/she 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, it is in their own interest to make the pot last as long as possible. Furthermore, 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 their votes and how often do they happen to ask for a reimbursement. This policy will not depend on the fee paid by the claimer as all users will pay the same fee for a given period.

We first explain the model that was iteratively created for this system. Next two technological alternatives to implement the system are described along with the reasons why one was chosen over the other. After that, some implementation details and difficulties are exposed to raise awareness about some limitations. Then two attacks that were discovered on the system are exposed. Finally, we present the improvements that we have thought of should the project be continued.

Chapter 1

The System Model

In this chapter we discuss the model of the system. That is, the theoretical design of the community and the mathematics supporting it. It will be as decoupled as possible from the underlying implementation.

1.1 Claims

The project is about insuring members of the community. In order for a user to indicate that he/she wants to receive a reimbursement from the community, the claimer needs to submit a **claim**. Each claim only concerns members 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. At the creation of the community we will choose the **Debating period**, an amount of time voters have to decide to approve or reject a claim. Each claim is submitted along with:

- **Amount of the claim**: the amount of money requested by the claimer.
- **Description**: a short explanation about the claim.
- **Documents information**: URLs and hashes of official documents used to support the claim. The need for such information is explained in 1.5.2.

Since each user is insured for only one car, a user only make one claim at a time.

1.2 Funding

This section explains how the system is funded. The basic idea is to have each user deposit at the 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 is further described in the 1.2.2). The money funded by each user is kept in his/her **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. Using statistics from the end of the last funding to the current date, called the **inter-funding period** and since the beginning of the community, an estimated entry price to be insured for a year is computed. Using the latter and the average price of a claim a new funding amount per user is computed. A more precise description 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). A user is insured only once he/she has contributed to the current funding cycle. The funding will last enough time so that it can be expected that users can contribute.

For $k \in \mathbb{Z}_+$ and τ_k denoting the start of the k^{th} funding period, the k^{th} round occurs during the time period $[\tau_k; \tau_k + d_f]$. We denote by T_k (**the top-off value**) the amount that will be in every member's balance at the end of the funding[1], that is, at $t = \tau_k + d_f$.

We will call **inter-funding period** the periods $\{t \in [\tau_k; \tau_{k+1}[: k \in \mathbb{Z}_+\}$.

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

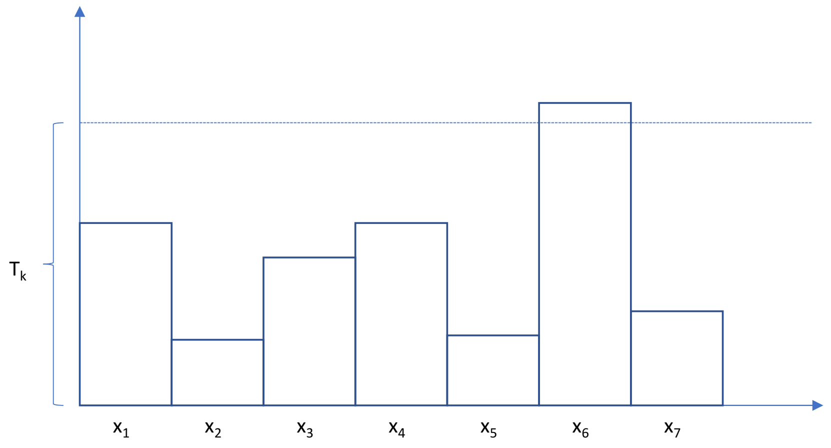

Figure 1.1: Balances at time τ_k

In Figures 1.1, 1.2 and 1.3, we graphically represent what is T_k. Note that the time at which the balances are considered is made clear in the caption so that we do not overload the graphic with the extra subscript. Figure 1.1 shows the situation at the time where the funding is required, while Figure 1.2 explains how each user top-offs its balance in order to stay in the community in the case where no new user arrives between τ_k and $\tau_k + d_f$. Should one or more new user arrive during this time period, then they will have to fund exactly T_k as Figure 1.3 suggests. An extra rule that can be observed is that user 6 already

[1]The scenario where a user did not fund by the end of the period is explained in 1.2.5.

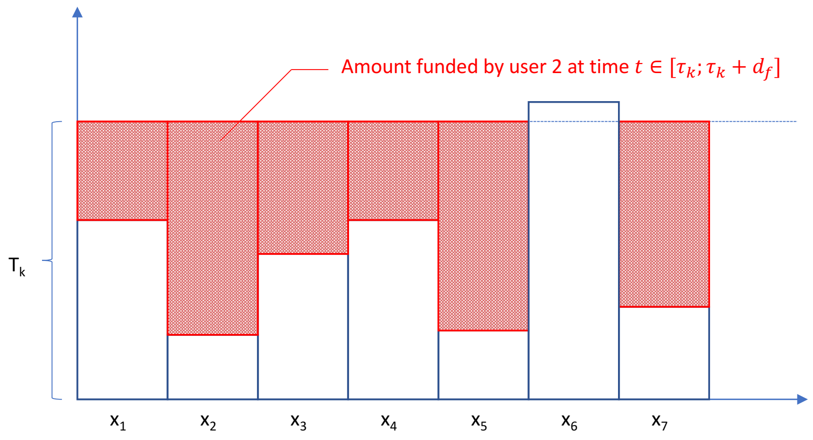

Figure 1.2: Balances at time $\tau_k + d_f$ when no user has joined during funding period

has a balance over T_k, therefore the user does not need to top off[2].

Generally speaking most users will have the same balance at the end of the k^{th} funding period, and during the following reimbursement period they will pay equal amounts. The Figures show high variations that should only appear when the community experiences many modifications of its community during inter-funding periods.

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

- **Genesis of the system:** since the composition of the community is not yet known we will have to rely on historical data. Let P_{hist} be the average price of a deductible and C_{hist} be the average number of crashes per user in a year that require them to pay a deductible[3] based on historical data. Then :
$$\forall i \in \{1, ..., n_0\} \quad x_{i,0} = C_{hist} \times P_{hist} = T_0 \tag{1.1}$$

- **Upon arrival of a new user:** Suppose a new participant arrives at time t_a. Then he/she will pay the top-off value that had to be reached by other users upon last funding. Suppose that there was no other funding initiated between τ_k and t_a i.e. $t_a \in [\tau_k ; \tau_{k+1}[$. Then if we denote by j the index of the new user:
$$x_{j,t_a} = T_k \tag{1.2}$$

[2]This can happen if the next top-off value is smaller than before and few or no claims have been paid

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

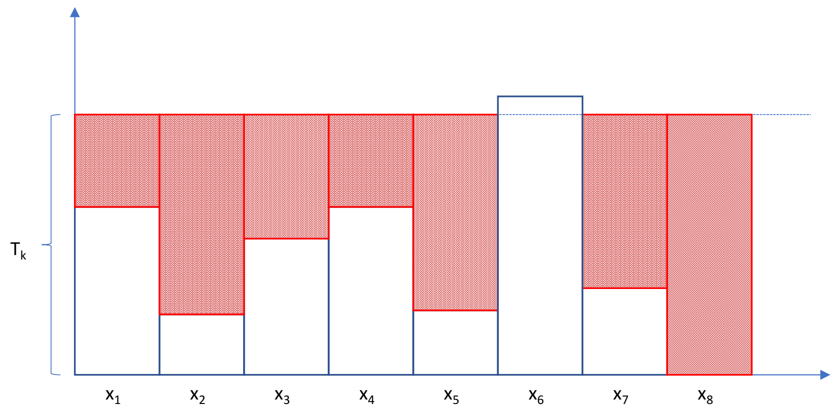

Figure 1.3: Balances at time $\tau_k + d_f$: user 8 has joined during the funding period

1.2.2 Condition Upon Which New Funding is Required

While we first did not consider that having a user with a negative balance would be a problem, it turns out that this makes it difficult to handle what to do when such a user decides to leave the community. It seems clear that we cannot reach any point at which one user's balance would become negative after reimbursing a claim.

Suppose that there are currently l pending claims, and a new claim is received. We denote the current time by t_{l+1} (time at which this new claim is received). If we ensure that the balance of every user can reimburse all claims that are pending as well as this new claim that is submitted then we will have no problem honoring all claims that we have accepted. In Figure 1.4, one can observe that the amount that has to be paid per user depends on the number of user at the time of the claim, and on the price of the claim. The condition that is checked at time t_{l+1} is :

$$\forall i \in \{1, ..., n_{t_{l+1}}\} \quad x_{i,t_{l+1}} - \sum_{j=1}^{l} \frac{p_j}{n_{t_j}} \geq \frac{p_{l+1}}{n_{t_{l+1}}} \tag{1.3}$$

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

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

It is important to note that by using condition 1.3 we are actually checking something stronger than we actually need: e.g. a user that arrives after t_1 is not involved in the reimbursement of the first claim but we still check that he/she has enough to reimburse it on top of other claims. But if all balances of users

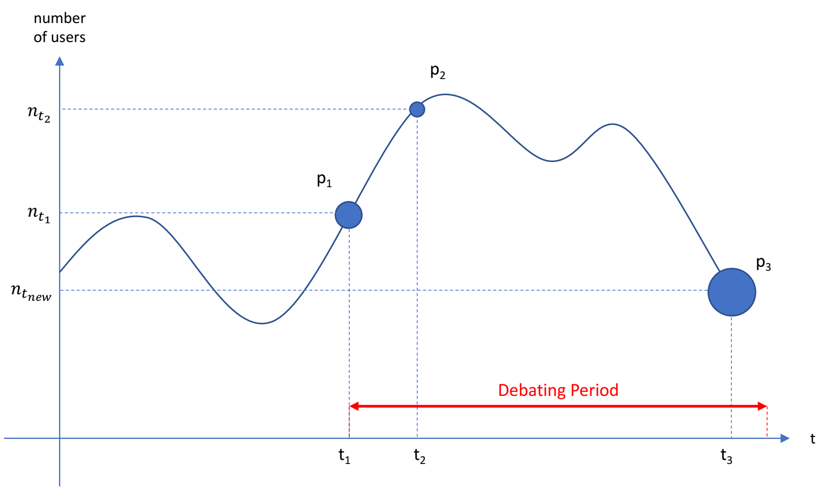

Figure 1.4: Illustration of the funding condition with only 2 pending claims when a new claim is submitted

that are involved in claim 1 pass the condition then this latter user will also verify it. This is due to the fact that his/her balance is greater or equal to the balance of one of the member with the least amount of money in his/her 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.

In order to avoid the attack described in Section 4.1, we also 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. One can also note that this approach is intuitive for the model, why should the system launch a new funding if the money is already here. The claim that caused the funding to be launched will, therefore, be refused. We refuse it because it either means that we currently have too many pending claims (and we want to artificially limit this number) or that the claimer has posted a claim which amount cannot be supported.

In conclusion a funding is launched if :

$$\begin{cases} \forall i \in \{1, ..., n_{t_{l+1}}\} \quad x_{i,t_{l+1}} - \sum_{j=1}^{l} \frac{p_j}{n_{t_j}} < \frac{p_{l+1}}{n_{t_{l+1}}} \\ \exists i \in \{1, ..., n_{t_{l+1}}\} \quad x_{i,t_{l+1}} < T_{k+1} \end{cases} \tag{1.4}$$

where implicitly $t_{l+1} \in [\tau_k; \tau_{k+1}[$

1.2.3 System Behavior During a Funding Period

When a claim is received, the system checks if the community is currently capable to "support" it (as described in subsection 1.2.2). If the claim cannot be supported then we need to decide how we deal with such claim:

- The claim is still accepted and the debating period for this particular claim is extended by the funding duration. This is the best solution from the claimer's point of view. But, we would have to pause any action related to this claim until the end of the funding because we want to restrict voting to people that have funded the community. 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 cannot 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 (c.f. 3.2.2 and 5.3.3). Therefore, the claimer will be notified that the system is not yet capable of informing him on a possible debating period end date but that he/she will later receive a notification that debating has started. This option seemed to be the most appropriate for our use-case.

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 global values that represent an **ideal user**. That is, a user that has been part of the community since the very start. This user will reflect 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 has been paid by the ideal user since previous funding.
2. `PaidYearlyAcc` which accumulates the sum of all yearly payments computed from each inter-funding period.
3. `NumberFundings` which keeps track of the number of finished fundings. (Initialised to one at the very beginning of the community as all members will have funded the first T_0)

As well as variables that are computed whenever needed :

1. $\texttt{PaidYearly}_{\texttt{hist}}$ which keeps track of what is paid by the ideal user between day 0 and current time on a yearly basis.
2. $\texttt{PaidYearly}_{\texttt{recent}}$ which keeps track of what was paid by the ideal user during last inter-funding period on a yearly basis.

Our goal in the following computation is to try to have users pay on a yearly basis (i.e. an inter-funding period should ideally last a year). The update rule of those three variables is quite intuitive. Every time a claim gets reimbursed, the amount paid by a user is added to `PaidSinceLastFunding`. This variable is reinitialized at the beginning of each inter-funding period.

We first introduce how the system estimates $\texttt{PaidYearly}_{\texttt{recent}}$ at τ_k:

$$\texttt{PaidYearly}_{\texttt{recent}} = 365 \times \frac{\texttt{PSLF}_{k-1}}{\texttt{days}(\tau_k - \tau_{k-1})} \tag{1.5}$$

where $\texttt{PSLF}_{k-1}$ is the amount paid by the ideal user during the preceding inter-funding period. Such that $\texttt{PSLF}_{-1} = \texttt{PaidYearly} = C_{Hist} \times P_{hist}$. Then :

$$\texttt{PaidYearlyAcc += } \texttt{PaidYearly}_{\texttt{recent}}$$

Then we can obtain the third one simply by :

$$\texttt{PaidYearly}_{\texttt{hist}} = \frac{\texttt{PaidYearlyAcc}}{\texttt{NumberFundings}}$$

This approach allows for $\texttt{PaidYearly}_{\texttt{hist}}$ 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 users 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:

$$T_k = \alpha \times \texttt{PaidYearly}_{\texttt{hist}} + (1 - \alpha) \times \texttt{PaidYearly}_{\texttt{recent}} \tag{1.6}$$

Since T_k is the amount that a user must reach once users have topped off, their funding amount is computed using the **Water-filling** solution as explained in Figure 1.2 and 1.3 s.t.

$$\forall i, j \quad x_{i,\tau_k+d_f} = x_{j,\tau_k+d_f} \geq T_k \tag{1.7}$$

The reason for the weighted average in Equation 1.6 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. The choice of α will depend on how variable is the community population expected to be[4].

[4]When we talk about population variability we are referring to the fact that some users are more likely than others to submit a claim. If a large part of the population changes and goes from very good drivers to accident-prone drivers then we need to adjust the top-off value

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 have not 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.

1.3 Voting

1.3.1 Fundamental Behavior

In order to determine whether a claim should be funded by the community or not, a vote is organized. To scale up, it would be desirable to avoid having all users vote for each and every claim. The most appropriate solution resides in **random sampling**. On the same principle as 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. 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 the 1.3.3).
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 turns out to be a good idea. We will require a critical mass that is large enough. This solution was kept anyway because the more users the more the idiosyncratic risk of each member gets diluted over the community. Therefore to have a low community risk we need to have a large enough number of user.

1.3.2 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 does not vote before the end of the debating period then he/she 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[5]. Then whenever a voter fails to vote before the deadline, a point is taken away from him/her. Once a user reaches zero points he/she is excluded. Those points will reset after a given number of successful votes[6]. In order to implement such strategy, we introduce the notion of **Quorum**, which denotes how many voters from the randomly

[5] The number of points is considered as an implementation decision.
[6] Again this number does not really affect the theory.

sampled users must vote in order for the voting result to be considered valid.

Once a user is excluded from the community he/she cannot vote nor submit any claim and he/she would have lost all the abilities that were provided to him/her based on his/her fundings.

1.3.3 Error Yielded by Random Sampling of Voters

In the following, we will follow an article by Terence Tao [7]. We will also assume ideal conditions of sampling.

Assumptions: While the article needs to assume a lot of different perfect behaviour, this is not necessary here. In fact, we will explain why the only assumption we need to make is Simple random sampling. Here are the assumptions that were made by the article along with the justification why the assumption is verified in our case.

- **Simple question**: voters are offered only two possible answers (accept or reject claim)
- **Perfect Response Rate**: a vote is considered as valid only if the Quorum is reached, therefore, we are ensured that all people we considered to be our sample will have responded since the sample has size Quorum. This will introduce a bias since the system considers only the voters that answered in time. But nevertheless, we consider this assumption to be reasonnable.
- **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**: We will always use the same 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 has been selected uniformly at random from the entire population independently from others except the fact that we do not allow replacement for a given vote. That is once a voter has been selected it cannot be selected a second time[7].

Let X denote the set of the total population. Let A be the subset of the population that would accept the claim. We wish to compute the margin of error [11] (at the confidence interval 95% [9]). Let p be the proportion of users that accept the claim:

$$p = \frac{|A|}{|X|} \tag{1.8}$$

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

Now suppose that we sample n users of the community to be representative of the entire community (under the assumptions described above). Then we will have, $\overline{p}$, the proportion of sampled voters that vote to approve the claim:

$$\overline{p} = \frac{|1 \leq i \leq n : x_i \in A|}{n} \tag{1.9}$$

Finally using Markov's inequality,

$$\forall r > 0 \quad \Pr(|\overline{p} - p| \leq r) \geq 1 - \frac{1}{4nr^2} \tag{1.10}$$

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

$$\Pr(|\overline{p} - p| \leq r) \geq 0.95 \tag{1.11}$$

It is sufficient to show :

$$1 - \frac{1}{4nr^2} \geq 0.95 \Rightarrow \quad r = \frac{1}{\sqrt{4n(1-0.95)}} \Rightarrow \quad \Pr(|\overline{p} - p| \leq r) \geq 0.95 \tag{1.12}$$

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

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 Quora

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

Sampled size We have seen in 1.3.2 that we need more people than Quorum in our sample of users to vote.

1.3.4 Critical Mass

Using the previous results on the size of the 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 Quorum at the issue of the vote. It is quite intuitive using the expectation of a binomial

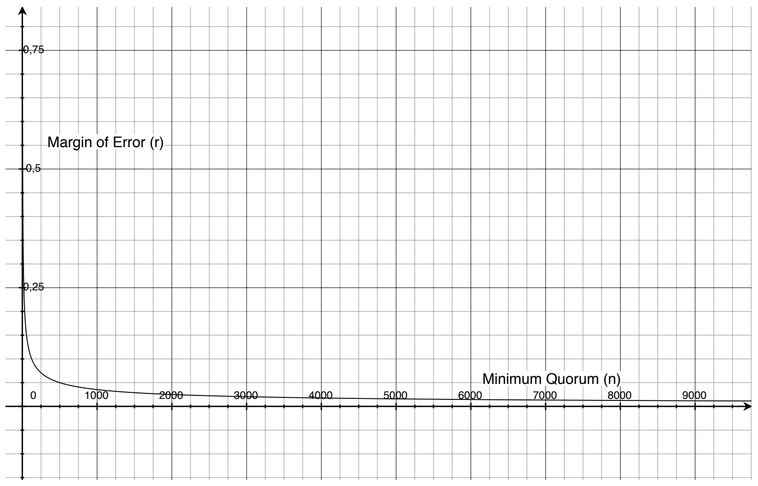

Figure 1.5: Tradeoff between the margin of error and Quorum

distribution[8], that if each user does not vote at least half of the time then the expected 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 Quorum with high probability. We plan to reach this critical mass before launching the community using a preregistration phase (discussed in section 5.1).

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 claims that have been submitted during his/her 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.

[8]This supposes that the simple random sampling is done without replacement. Even though it is not the case, it will not be too far from the reality given the large community size.

1.4.1 Jackpot

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 nonzero 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, his/her balance will be split. For each pending claim he/she was supposed to pay for, the amount he/she was supposed to pay will be taken from his/her balance to the claim's `leftOvers`. Everything that remains after that will be added to `jackpot`. 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 does not reveal the real amount of money that is held by the contract. In fact, rounding[9] will often leave a very small amount of money which is not 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.). The two counters are used solely for fund tracking purposes (this explains why we simply forget about `leftOvers` once the claim is reimbursed, we do not need to keep track of this amount anymore).

1.4.2 Unfairness

One could point out that this method is not fair. In fact it is not : new user could benefit from funds left by long gone users. But we would like to argue that this situation cannot be mitigated.

Dividing the money among current community It is important to understand that the amount in a user's balance is relatively small compared to the size of the community. Therefore rather than putting the money in `jackpot`, if we would divide it among users then this might end up dividing a small value by a large one and therefore making a lot of rounding errors (c.f. Section 3.2.3). Also, what happens with `leftOvers` should the claim not be accepted ? Then we would need to store somewhere the ids of the user that are entitled to receive those funds. This turns out to be inefficient.

Having time dependent jackpot On the other hand, one could think of keeping different jackpots in order to be fair. This again could yield a very computationally heavy solution, it would potentially require to keep track of too much variables.

[9]The reason for rounding will be further explained in 3.2.3

Considering the fact that users form a commmunity and that to enter new members need to pay, then it does not seem so bad that new users benefit from the long-run community. One can see that new user also benefit (or suffer) from the behavior of long gone users through the weighted average defined in Equation 1.6.

1.5 Anonymity and Verification of Claims' Documents

1.5.1 Anonymity

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 of claimants. 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's profile, to keep statistics about users (number of claims, age, the percentage of positive votes...), therefore if we do not identify users, then those statistics can be reset as often as the user wants.
- Voters need to be able to verify that documents linked to the claim are legitimate. 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 does not ensure anonymity is not such a problem. We could not 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 community does not have such "experts". Therefore it heavily relies on the documents that prove the crash existence and the profile of the claimer. Nevertheless, there might be ways to ensure anonymity in some ways (e.g. only the jury can access the documents) but it was decided that it was not in the scope of our project.

Finally we decided not to provide any form of anonymity for anything that happens on the system : claim, members' identity, ect.

1.5.2 Document Integrity Verification

Voters are supposed to base their decision for a claim on the details that are provided by the claimer. The most important are the claim's documents (*e.g. police statement, witnesses letters, pictures of the damages,...*). We need to ensure:

- Documents are not changed by the claimer once the claim has been broadcasted to voters.
- Documents from the previous claims are still available in the future (in order to allow the voters to check that the new claim is valid: it does not concern another car, has not been reimbursed yet, ...)

Storing documents on the blockchain is very inefficient (as explained in 3.2.1). Therefore we decided to verify the integrity of the document using the hash of the archive to de-correlate the document sharing policy with the blockchain.

With each claim will be stored a `url` where an archive[10] of the documents can be downloaded. The voters will be able to check that the archive of documents they obtain is the same as obtained by every other voter since its hash can be recomputed and compared to the value stored in the claim.

1.6 Community Death

We have seen and formalised in Subsection 1.3.4 the need to have a critical mass for the system to be able to work. But then we also always need to check that we will not go below this lower limit once the system is online. Moreover, what should be the policy once we go below such limit? We also want to emphasize the fact that this limit will be detected as soon as one user will leave the community and the remaining number of user is under the limit. Therefore, the community will still have `critical mass - 1` users. We decided to **kill the actual community**. In such scenario we would:

1. Reimburse the current amount of money contained in each user's balance once all pending claims have been processed.
2. As we previously mentioned those balances do not represent the real amount of money held by the system. Therefore a user will be randomly selected[11] to receive all the eventual remaining funds of the system.
3. Force all remaining users to preregister again *(not implemented yet: so far we simply kill the community forever).* And would pause all actions except the ones concerning pending claims.

[10] A compressed folder.
[11] To avoid targeted attacks.

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

Chapter 2

Technical Alternatives Considered

Since many alternatives were considered when designing the system, it was decided to present the ”trial and error” protocol that was experienced. This will hopefully explain decisions that were undertaken and their 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 a possibility was to use Bitcoin.

2.1.1 Principle of Bitcoin

This paragraph presents a very simplified and not comprehensive explanation of the Bitcoin system. More will be explained about the Blockchain technology in 2.2. 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 criteria. In order to cheat the blockchain, since all blocks are linked to each other using hashes, one needs 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 do not need to be verified by an intermediary. The network always considers the longest chain to be the valid one [8].

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, as opposed to regular Bitcoin wallets, use a multi-signature scheme. This enables to have N-out-of-M participants controlling 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 having 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

Many problems were discovered about the solution using distributed wallets.

- **Collusion**: if N users out the M cosigners decide to steal the deposit: nothing protects the community from doing so. And it is a huge assumption to consider that no user will use this vulnerability. Given the fact that the maximum number of cosigners is 15, the resulting security is very weak.
- **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.
- **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.
- **The 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

Given these problem it was decided not to use distributed wallets. 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 Common Funds

The 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 (unspent transaction output) 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 scripts. But the simple stack based Bitcoin scripting language is not 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 writing complex scripts for our project. The idea is to create, using smart contracts, a Decentralized Autonomous Organization (DAO).

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 [10]. This high-level explanation is further refined in subsection 2.2.3.

2.2.2 Ethereum Smart Contracts

Ethereum is composed of two kinds 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.

Using this duality we can see that creating a community is simply a matter of creating the correct behaviour for the Smart contract, make an access control policy for interactions between the smart contract and EOA that are part of the community. Once the source code of the contract has been written it can be coupled with a front-end that will make the whole system transparent for the user.

2.2.3 Ethereum Components

We will first present the different components of the system and will then explain how those interact together.

Gas vs. Ether As we have explained very superficially in previous section, Ethereum provides a virtual machine that runs on multiple international nodes (called **livenet**). Their are two values of interest in the system:

- **Ether**: Comparable with bitcoin, it is a cryptocurrency. It can be exchanged between users (either as a financial transaction or in exchange for ressources).
- **Gas**: a unit used to measure computational use. It is not very different from the way we pay for electricity in our houses. We use the kW as a measure of the amount of electricity used, which once multiplied with the cost of a kW gives the price to pay.

Ethereum Transactions Users use transaction to exchange data on the network. A **transaction** (denoted `tx`) is a signed data package that stores a message to be sent from an EOA. It can be a fund transfer or a call to a smart contract function. It contains:

- The recipient of the message
- A signature identifying the sender
- The amount of ether to transfer from the sender to the recipient
- A data field (optionnal)
- `startgas`: represents the maximum number of computational steps the transaction execution is allowed to take
- `gasprice`: represents the fee the sender pays per computational step

Accounts An account is composed of:

- A nonce: a counter used to ensure that all transactions are processed only once
- The ether balance
- The account's contract code (if it exists, that is if the account is the account of a smart contract)
- The account's storage (it it exists, empty by default)

State of Ethereum Ethereum Blockchain has a state. This state is simply composed of all the accounts. Using the transactions that are broadcasted in the network, peers can update the state of the blockchain they locally store. State transitions are direct transfers of value and information between accounts.

Suppose we are at state `S`, we describe the transition function `APPLY(S,tx)` that yields the new state `S'`:

1. Check that `tx` is valid (checks the signature, the number of fields, the correctness of the nonce according to the sender's nonce,...)
2. Compute `gasprice*startgas`, increment the sender's nonce and subtract the fee from the sender's balance (this allows the miner to ensure that the funds are locked while he/she tries to process the transaction)
3. Transfer the transaction value from the sender's account to the receiving account. If the receiving account is a contract, run the contract's code either to completion or until the execution runs out of gas.
4. If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all state changes except the payment of the fees, and add the fees to the miner's account.
5. Otherwise, transfer to the miner's account the transaction fees (this time the real number of steps are paid at `gasprice`) and refund the rest to the caller (`gasprice*startgas - trans_fees`).

This state transition function will later be used to explain how transaction are processed.

Blocks A **block** is a package of transaction. The network is intended to create a block every 15 seconds. A block contains a timestamp, a nonce, the hash of the previous block, a list of all transactions that have taken place since the previous block, and the most recent state (the last state yielded by the list of transactions). This creates what is known as the **blockchain** given that each block are "chained" to the preceding one using hashes.

Miners A **miner** is a node of the network that provides ressources in exchange for ether. They do not only process transactions but also process blocks: called **mining**. Given the fact that it uses a decentralized system to keep track of the state, it needs the order of the transactions to agree on a state. This will be done thanks to blocks that are ordered.

Blockchain This chain of blocks is therefore describing an order of blocks (that define themselves a transaction order). But it is not clear how do we decide which blockchain is valid : miners are competing against each other to mine blocks. Therefore, the network always considers the blockchain with highest difficulty[1] to be the authentic one.

[1]Later on, the notion of difficulty is explained.

Block Validation Once a block has been mined, it is broadcasted to all other nodes of the network that will need to check that it is valid. To check the validity of a block before adding it to the blockchain each node has to:

1. Check if previous block referenced is valid and exists.
2. Check that the timestamp is greater than that of the previous block and less than 2 hours in the future.
3. Check that the proof of work is valid (explained later on in this subsection).
4. Check that from the state of the previous block, applying the ordered transactions using the state transition does not return any error. Following Figure 2.1, let `S[0]` be the state from previous block and suppose there are n transactions in the current block. `S_FINAL` is the state once the block miner has been paid for his mining.
5. Check that the computed `S_FINAL` is identical to the one provided in the block's header.

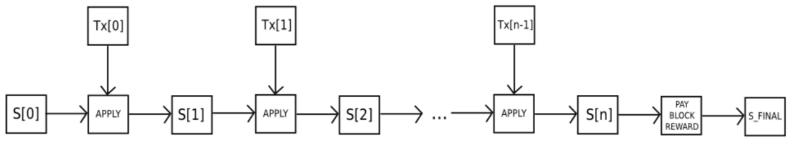

Figure 2.1: Verification of state transition to build block

In this protocol, the miner encloses in the block as the last transaction a transfer of 5 ether. If he/she is the first one to succeed mining the block and that it gets accepted by the network as the authentic new blockchain then he/she will have earned the 5 ether.

Proof-of-Work The final step that we miss is how does the miner produces the proof of work that is described in the block validation protocol. This proof of work is used for two reasons:

- Make ether creation increasingly hard: each mined blocks provides a fixed amount of ether to the successful miner. The original amount of ether being fixed and later split among users, the value of a given amount of ether need to increase as the community grows. Therefore mining a block must be harder and harder.
- Avoid attacks: if mining a block was an easy task, an attacker could falsify the whole blockchain and get any desired amount of ether (since possession of ether is based on what transaction from the entire blockchain

have yielded in each account's balance). Then, if more than 50% of the network is honest, the computing power provided is always greater than the computing power of a potential attack, and the faithful part of the network will always produce the longest blockchain, as mining speed is directly proportional to computing power.

In order to provide a proof of work the miner will try incrementing values for the block's nonce. For each of these values, he/she tries to compute the hash of the block (that contains this nonce as well as the hash of the previous block etc.). This hash must be greater than a given target. This target is dynamically set to adjust the difficulty. Since hashes are designed to be unpredictable, the miner needs to try many nonces before he/she can find a valid hash. Producing the nonce is a hard task but verifying the proof of work is a simple hash computation.

Call vs. Transaction A call is a local invocation of a contract function that does not broadcast or publish anything on the blockchain. It is a read-only operation and will not consume any Ether. It simulates what would happen in a transaction, but discards all the state changes when it is done. It is **synchronous** and the return value of the contract function is returned immediately.

A transaction is broadcasted to the network, processed by miners, and if valid, is published on the blockchain. It is a write-operation that will affect other accounts, update the state of the blockchain, and consume Ether (unless a miner accepts it with a gas price of zero). It is **asynchronous**, because it is possible that no miners will include the transaction in a block (for example, the gas price for the transaction may be too low). Therefore the return values are passed to the caller using special objects called events that can be listened to by users.

2.2.4 Putting the Pieces Together

Now that we have defined those components, we would like to illustrate the system using an example. Assume Bob wants to call a contract that only has one function that stores its name in the contract's storage. The contract is at $\texttt{address}_\texttt{c}$.

Bob creates a transaction :

- Recipient : $\texttt{address}_\texttt{c}$
- Signed using his private key
- he does not wish to send ether to the contract therefore the amount to transfer to the recipient is set to 0.
- the function takes as input the user's name : 'BOB' is stored in the data field.
- `startgas` is set to an arbitrary value `s` Wei[2]).

[2]Because floats cannot be represented in Solidity, different units are used : 1 ether = 10^{18} Wei

- `gasprice` is set according to the market value `g` Wei such that the transaction is fairly priced

He then broadcasts the transaction to the rest of the network. Miners will receive the transaction. Some of them will eventually consider that it is worth mining it given its `gasprice`. Therefore, they will try to include it in a block and to compute the proof of work. For the sake of this example we will assume that Alice and Eve, two miners, have successfully mined the block.

Both Alice and Eve broadcast to their neighbours the mined block and Bob will receive an event indicating that the function has be ran. For Alice and Eve it is not clear who won the mining competition. In fact, this will never be entirely sure. It is considered that after 40 blocks that have been appended over the mined block, it is very unprobable that another longer blockchain will later be discovered.

Since the notion of ether balance of each user is encapsulated into the blockchain this is not a major issue. After these 40 blocks, Alice still has not received a longer blockchain and can consider she is the final winner.[3]

This concludes our example. It could remain unclear to an attentive reader of where is the code execution taking place. As seen in the previous paragraphs, code execution is part of state transition, which itself is part of block validation. Therefore, if a transaction is placed in block `B`, its code will be executed by all nodes that download and validate the block `B` in the future. But only the miner that creates the block obtains the reward for mining the block as well as the gas paid for code execution.

2.2.5 Conclusion

This justifies why smart contract can be trusted. Every node in the network will run the code that is supposed to be run. Therefore a miner cannot cheat the execution of a transaction.

Then to avoid all the problems that were discovered with the bitcoin alternative, the challenge is to design the smart contract to be fair and without loopholes.

This also explains why so much work was put into the system design because it is as important as the underlying technology. The blockchain only ensures that the execution will be correct but it does not prevent any exploit on the source code itself.

[3]Turns out that Eve received a longer blockchain containing Alice's mined block and therefore could not consider her version valid anymore.

Chapter 3

Implementation

This chapter aims to present some details about the system implementation. A reader which is not directly interested in the implementation can skip it. It was included in the report in order to point out implementation decisions that were undertaken and to provide help to someone potentially deciding to continue the system's development.

3.1 Development Workflow

3.1.1 Tools

Language

As seen in Section 2.2, Ethereum provides the Ethereum Virtual Machine to network nodes. Programmers can develop smart contracts to be run on this virtual machine using the Solidity[1] programming language. Solidity is a contract-oriented language with a syntax highly similar to JavaScript, statically typed, it supports libraries and complex user-defined types. The described system is therefore entirely written in this language.

Although Solidity appeared at first as another programming language whose one simply needs to learn the syntax and principles. But it's youth made development difficult. Indeed, the language was first proposed in 2014, meaning that the documentation is still sparse, and the developers community very small compared to other well-established languages. We faced multiple limitations concerning, for example, return values, arrays or variables storage location. However, workarounds are always possible to accomplish one's goal at the price of either more lines of code, more storage, or less efficiency, which could be problematic for a large contract given that users pay for computation (c.f. 2.2.3).

JavaScript is also used throughout the project to interact with the contract, write tests, and implement the user interface (along with usual web development languages as HTML and CSS).

[1]`https://solidity.readthedocs.io/en/develop/`

Blockchain Development Framework

The Ethereum development framework `truffle`[2] has been used throughout the system implementation. This framework aims to facilitate the life of Ethereum programmers by providing useful tools to compile, migrate and test contracts but also to interact with them.

Network

Compiled smart contracts need to be deployed on a network in order to be tested. As it does not appear reasonable to migrate the contract on the Ethereum network each time a small change is made, we used `testrpc`[3] to simulate such a network. It is possible to describe `testrpc` as an in-memory network, simulating transactions processing, block creation and blockchain growth. It is important to note that `testrpc` does not introduce any delays in transaction processing. Those are directly included into blocks and contract calls are run instantly as opposed to the real Ethereum network on which it takes around 15 seconds for code to be run and contract state to be updated.

`testrpc` also facilitates development by providing programmers with fake accounts filled with ether, and the possibility to manually define some options as the number of accounts, the mining time (0 by default as said earlier), the gas price etc.

3.1.2 Testing and Debugging

When starting the implementation, we directly decided that testing should not be neglected. The contract being written step by step, adding features incrementally, it was important for us to make sure we did not break any previous functionality when adding a new one. The obvious mean to achieve such a goal was to write unit tests for a given feature and the feature itself in a simultaneous way.

However, achieving 100% code coverage by testing is time consuming, and would compromise a lot the development pace. Full code coverage is typically applied when a dedicated testing team is working on the project. Then we decided to focus on feature coverage, and tried to achieve 100% coverage on important features, ensuring they all survive to the project growth.

Truffle provides contract programmers with a complete testing environment based on `mocha`[4], a JavaScript testing framework, and `chai`[5], a JavaScript library providing assertions. Using those libraries required some learning, but allowed for a relatively easy test devellopment.

Debugging smart contracts is not an easy task. This is due to the lack of integrated debugger on the platform, or even simply the possibility to print statements from Solidity code. Then, the main debugging strategy we used

[2]`http://truffleframework.com`
[3]`https://github.com/ethereumjs/testrpc`
[4]`https://mochajs.org`
[5]`http://chaijs.com`

consisted in analysing events. Solidity events are small chunks of data which are sent back to the contract caller during execution on the EVM. It is then possible to infer execution path by looking at events with Truffle.

Globally, we can safely say that testing and debugging have been the two most time consuming parts of the implementation.

3.1.3 Development phases

We can split the implementation into multiple phases during which we worked on different parts of the system. These phases are listed below.

Contract

Most of the time has been spent developing the smart contract, testing and debugging it. The contract has been written incrementally, firstly because we needed to learn the technology and tools, and secondly to make sure we would not crash into the wall by directly writing a complex contract.

The first part of the contract consisted in implementing a simple community where users are able to join, post claims, and vote. Then we added the funding features with the possibility to pay, manage money, reimburse users when a claim is approved, and fund the community when needed. The third and last part consisted in implementing leftovers management, incentive to vote, random sampling, as well as small details and optimisations.

It is important to note that random sampling is experimental at the time of writing and could not really be tested at large scale. Indeed, to efficiently test the scalability of such an operation, we would need at least a community of the size of the critical mass defined earlier in the model, which is not possible to get unless we deploy the project and open it to public beta testing. The contract is clearly not mature enough to reach this stage.

User Interface

One of the goal was also to develop a basic user interface to be able to easily interact with the contract. We wanted to provide potential users with an intuitive and easy to use interface, in order to make their experience as pleasant as possible.

Include screenshot ?

Demo

The very last part of development consisted in writing a small script, which coupled with the interface, would provide a demonstration of the basic system features, following a precise scenario.

3.2 Implementation challenges

3.2.1 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. Three alternatives were considered:

- **Storing the documents in the contract's storage**: this turned out to be a very bad idea. To make it simple in February 2016 each GigaByte of data costs $76'000 [2]. The only advantage of this would be that each claim documents 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 (which can also be achieved using an integrity verification see 1.5.2).

- **Store the documents on a community paid server**: this solution was the simplest for the end user because it would allow offering 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. (Moreover it might really go against our disintermediation goal)

- **User Managed Storage**: finally, the option that remained was to have each user use its favorite online data storage platform. On the contract, the system would only store a link to the archive containing the document and a hash of the archive (c.f. 1.5.2). If the user claims to have lost some documents of previous claims it will be the voters responsibility to determine wether or not he/she has enough informations to rule the claim.

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.

3.2.2 Scheduled Calls

Some actions have a time limit (e.g. voting has a deadline, funding as well). Once a time limit is reached, we need to execute some code. 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 quite regular intervals. But then still, development using this tool would be very complicated given that the contract can only be accessed on the live blockchain.

- Rely entirely on the user: after all, the claimer wants the money, therefore, it is not too much of an assumption to say that he/she can take care of asking for the system to rule the claim once the deadline is past. But this seems very unfriendly from the user's point of view.
- 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 their deadline and ask the contract to execute the ones that are meant to be finished.

This last option was the only one considered valid for our system. Nevertheless, since the client-side application will not be developed entirely, it remains a theoretical assumption.

3.2.3 Lack of Float Representation in Solidity

Solidity does not yet support floating point representation or operations. Therefore, all divisions are rounded integer divisions. This is why we avoid as much as possible to have multiple divisions in a row so that the relative rounding does not propagate. So far, Ethereum uses different unit magnitudes to ease development (e.g. ether and wei).

3.2.4 Random Sampling

A challenge in this application was also the fact that randomness is very hard to obtain in Solidity [4].

Randomness in Ethereum

Nowadays random number generation for Ethereum is still being researched by Solidity enthusiasts. The only way we found to do this properly was to have each user generating a random number and to use a commitment scheme to submit it to the contract. When numbers are revealed they are all `XOR`ed together to obtain the random number.

The commitment scheme is used because everything that the contract sees an attacker can potentially see. Therefore if the number were sent in clear, a potential attacker could choose his/her number in order to get the number computed by the contract to be any number of his/her choice.

That would be the "clean way" of doing things. The problem is that it goes against some principles of our model:

- We need each and every user to submit a random number whenever we do a sampling. But the idea of the sampling was to avoid having every user to vote for every claim.

- Each interaction with the contract cost gas, therefore we want to avoid all those interactions that would cost a lot of money.

Therefore we decided to use a very poor source of randomness and to protect against the possible flaws of an attacker using this weakness.

Using Block Hashes as a Source of Randomness

Before explaining how we use the block hash to generate a pseudo-random number, we want to emphasize the fact that this technique could certainly be greatly improved. This was not the scope of our project and could be the subject of a whole semester project.

Block hashes are unpredictable and contained in the blockchain. This makes them easy to access. However, this is not entirely safe. The problem lies in the fact that miners have (limited) control over them. In fact, they can choose to publish or not a block. By withholding a block they could influence the random function.

This was taken into account already in 4.2.1. In order to obtain a pseudo-random number $r \in \{0, ..., N-1\}$:

$$r = H(\texttt{PreviousBlock}) \mod N \tag{3.1}$$

where we assume that the Hash of the previous block is an integer. This method will also yield a non uniform r depending on the value of N[6]. But again the poor randomness properties can be handled and it is not our goal to provide a perfect Pseudo Random Number Generator.

As we have already said, an attacker can never cheat the community since he/she can only get as much as he/she pays for to be part of the community with his/her fake accounts.

3.3 Conclusion

Ethereum development is still very new and we can easily feel it by the lack of presence on the web. Furthermore, the development process is globally more painful than classical ones given the number of parties interacting with each other and their respective immaturity. However there is clearly a growing community around the platform and it should become much easier for new developers to dive into Blockchain development as time goes. Also, tools like Truffle or testrpc are really essential to get an experience as pleasant as possible and the number of such tools is permanently growing.

[6]if N is a prime number then $\mathbb{Z}_N$ would be a field and r would be uniform.

Chapter 4

Attacks

This section will present some attacks that were imagined during the implementation phase. It is not (and does not aim to be) a comprehensive proof of the security of the system. An overall security analysis is still lacking due to the remaining possible deep modifications of the model.

4.1 Attack on statistics

4.1.1 Idea of the attack

While implementing the system we realized we might have a problem. Suppose an attacker wants to affect the value of $\texttt{PaidYearly}_{\texttt{hist}}$ by engaging a new funding before any claim has been executed (reimbursed) for the current inter-funding period. This attacker would then be able to influence the top-off value of the system. Since α is static, he/she can make the top off value arbitrary small and provoque a waterfall of fundings that will only end once the statistics as taken back a coherent value. Then we see in equation 1.5 that $\texttt{PSLF}_{k-1}$ will be zero, hence so will $\texttt{PaidYearly}_{\texttt{recent}}$ Moreover, when a new funding is launched we update the values of `PaidYearlyAcc` and $\texttt{PaidYearly}_{\texttt{hist}}$. Here `PaidYearlyAcc` will not be changed and $\texttt{PaidYearly}_{\texttt{hist}}$ will decrease as the number of finished inter-funding periods will increase.

4.1.2 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 posts one or more claim such that the system cannot support it/them[1].

[1]he/she would need to use fake accounts as user are limited to one claim per account.

Since no claim has been executed yet, if we suppose that last funding was the k^{th} one, then each user has at least T_k in his/her balance.

4.1.3 Avoiding Such Attacks - 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 $T_{k+1} \leq T_k$. This will ensure that no funding is launched (according to 1.2.2) and therefore will prevent $\texttt{PaidYearly}_{\texttt{hist}}$ to be updated.

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

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

$$\begin{aligned} T_k &= [\alpha \times \texttt{PaidYearly} + (1-\alpha) \times \texttt{PSLF}_{k-1}] \\ &\geq \alpha \times \texttt{PaidYearly} = T_{k+1} \end{aligned} \tag{4.2}$$

Now we need to show that if the attack happens for $t \in [\tau_0; \tau_1[$. At 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 a "historical estimate" of the yearly price). Then the initial funding top-off value would be:

$$\begin{aligned} T_0 &= [\alpha \times \texttt{PaidYearly} + (1-\alpha) \times \texttt{PSLF}_{-1}] \\ &= [\alpha \times \texttt{PaidYearly} + (1-\alpha) \times \texttt{PaidYearly}] \\ &= \texttt{PaidYearly} \end{aligned} \tag{4.3}$$

and :

$$\begin{aligned} T_1 &= \alpha \times \texttt{PaidYearly} \\ &\leq \texttt{PaidYearly} \end{aligned} \tag{4.4}$$

We have proven that for any $k \geq 0$ denoting the funding before the attack, the condition $T_k \geq T_{k+1}$ holds and therefore no funding is launched. This ensures that `fundingNumber` is not updated and therefore the statistics are not affected by the attack.

4.2 Denial of Service Attack

We also noticed that a potential attacker can make the system unusable by creating a big claim or by creating fake users and posting claims using each account such that the accumulative amount launches a new funding[2]. Therefore we thought of introducing a limit on the amount of a claim.

[2]Recall that a user can only post one claim at a time

4.2.1 Claim Amount Upper Bound

In order to choose this upper bound, we observed that we wanted to avoid the very unlikely case where an attacker has many fake accounts and all voters randomly selected for a claim he/she posted are fake accounts he/she holds. In such case, he/she would be able to decide to get reimbursed for his/her own claim.

Suppose last funding was the k^{th} one. To avoid that situation a simple countermeasure is to fix the maximum amount of a claim to be equal to $T_k \times$ Quorum. This would mean that the attacker cannot be reimbursed for each claim more than what he/she paid for in order to create his/her fake accounts. Moreover, the randomisation of the voters adresses the issue that he/she could still be reimbursed for the other fake claims.

4.2.2 Effectiveness

The previous upper bound allows to ensure that no user can post a big claim to launch a funding. Also, since the number of pending claim is auto-regulated (c.f. 1.2.2), then even with a great amount of fake accounts, the attacker cannot use multiple accounts to launch a new funding. This upper bound will also serve as a good protection against the poor randomness offered by our system (see 3.2.4).

Chapter 5

Further Improvements

This section aims at treating the subjects that we considered in our design but could not implement in the time frame that was allocated for the project. It should serve as a basis to restart the project in the future.

5.1 Preregistration

The notion of **Critical Mass** was introduced in 1.3.4. This number is defined as the minimum number of users for the community to behave as designed. So far we have considered that this number would be reached whenever the community was created but we did not discuss how that could be achieved.

The idea is to use an external pre-registration platform. This would allow having users register before we launch the platform. There are several possibilities for such platform. It could either be implemented as a smart contract on the blockchain or be completely external.

5.1.1 Idea

Each user that wishes to participate in the community will have the possibility to pre-register. In order to do so, he/she will need to provide the initial Top-Off Value T_0. The preregistration system will hold on to this amount.

Then there are several possibilities that can be imagined.

- The system keeps track of an expiry date for each registration: it refunds the user after that.
- The system can refund users whenever they ask to cancel their pre-registration which they can do only during the pre-registration period, they will not be able to get their money back once the community has been created.

5.1.2 Using Another Smart-contract

The very intuitive idea would be to use the blockchain to implement a system that can manage the funds during the pre-registration. The main justification for this is that we want a decentralised system. Also, this would allow us to automatically go back to preregistration whenever we go below the critical mass without the need for external support. Moreover, by using an external platform, we might need to change the system source code if this platform disappears in the future. Therefore we believe that it would be a good idea to use another smart contract to do pre-registration. It could be created as many times as necessary by the original system. As long as the original system is up and running it means that the Ethereum blockchain is still functional and that we can go back to pre-registration.

5.2 Community Rules

By community rules we mean the rule that all voters should follow in order to base their vote on same criteria. Those were not yet discussed because they would require surveying potential users to discover in what types of situations they wish to be insured and to insure others.

We will simply discuss examples of potential rules that could apply and could be proposed to users:

- In the case of a claim concerning a responsible crash[1], a proof should be produced that the claimer has not consumed any drugs and/or alcohol.
- The community covers repairs for car body work not exceeding $\$x$.
- We require at least k witnesses that are not affiliated to the claimer.

Those rules can be further developed and their scope has no limit. One could also think about another policy to update those rules based on a community vote with the possibility for users who do not agree with rules that they did not sign up for, to leave the community. This would allow adapting to changing regulations or community.

5.3 Extra Cost Management

5.3.1 Explanation of Extra Costs

Following the explanation of gas from 2.2.3, we know that what contract callers specify as `gasPrice` is used by miners to rank transactions for inclusion in the blockchain. It is the price in Wei of one unit of gas, in which operations are priced. Therefore each interaction with the contract costs some fees to the user.

We considered that those extra costs are no different from what one could experience in real life. Let us think about the regular means of communication with a traditional insurance. To send documents to an insurance it is necessary

[1]Responsible crash: a crash caused by the claimer and that can only be linked to his/her wrong actions

to pay for a stamp. Different stamp price will yield different services (reliability, tracking, delivery speed etc.). Here those fees are not set upfront but established as a negotiation between miners and the user. When proposing a transaction to the transaction pool, the user specifies how much he/she is willing to pay for each computation power unit. And also specify what is the limit he/she does not want to exceed when executing the transaction. Based on this, miners can decide to execute the transaction or not.

5.3.2 Storage - Less of a Concern

Interestingly, storage is also paid by users. Whenever a call to the contract results in a modification of the contract storage requirements, it will be translated in the gas consumption of this contract call. But the code does not contain any user-callable function that would cause important storage modification. Therefore, the storage costs should be evenly split between users. Each user will basically pay for storage added because of his/her activity in the community.

5.3.3 Consequences For Implementation

The main concern around those extra costs is that we wish to avoid having users unfairly paying more than others. That can happen because of the design decisions that were made for the system. Because of the lack of scheduled calls explained in 3.2.2 we decided to use contract interaction as triggers.

Whenever a user interacts with the contract, the contract checks that there is nothing to be executed (e.g. executing a claim, finishing a funding, ...). But in the event when something has to be executed, the gas needed to do so will have to be paid by the user who interacted. Some of those actions are very heavy and will need a lot of gas.

Since this is a very early version of the system, implementation did not handle extra cost consideration. But this has to be done. The risk would be that users get discouraged by such high costs and would let the community die.

5.4 Migration Toward the Ethereum Network

`testrpc` has allowed us to develop easily, providing us with a local blockchain, some fake accounts and a total transparency concerning transaction mining. Deploying the contract on the real blockchain will have to happen at some point in time.

It is important to note that once this is done, there is no going back. The system will be live and anyone will be able to interact with it.

However, before migrating to the real Ethereum network, it is possible to take an intermediary step with a public `testnet`[2]. Ethereum `testnets` are real networks, with nodes running across continents, which can be used to simulate

[2]Note that a `testnet` can also be private, which require setting up hardware and running Ethereum clients to simulate the blockchain activity.

a contract at large scale. The advantage over `testrpc` is that delays[3] for block mining and transaction validation are almost identical to the ones experienced on the real network, and multiple users can use the contract, as it is available on a public network. Such networks are `ropsten`, `kovan` or `rinkeby`[3], which differ from the client you can use to access them, and the type of consensus algorithm. Whereas these networks are practical for development purpose, they often lack miners and computing power and are very vulnerable to attackers[6]. Although, it is often not a major problem for users (except for the lack of availability during an outage), since ether used on these networks is fake ether. Users can get it either by mining the network, or using special links.

Once the contract proves its reliability on a public testnet, with a large number of beta users, we could consider moving it to production on the real public Ethereum network.

This is why migration to the real network should not be done before the system architecture is finished and bullet proof.

[3]In our system, these delays would affect the responsiveness of the system. E.g. it would be possible that a claim submission is not mined immediately therefore causing the claimer to experience a small delay. Nevertheless, those delays can be entirely absorbed by the client-side application that could handle re-submission etc.

Chapter 6

Conclusion

The blockchain technology seems is used as a buzzword in the industry. As we experienced it ourselves, its inner-workings are not trivial. Developing blockchain based applications is expensive and time-consuming. We wanted to emphasise on this point given that we experienced it ourselves : using the blockchain to build an application is a decision that must be well thought and should be avoided if there exist any other solution to implement the application.

The blockchain should not be used without seriously considering the advantages it provides. For our system, we believe that the choice is justified as we have already explained it.

Because we are convinced of the usability of the blockchain for our use case, we are hoping that the model will be further refined and will hopefully be launched one day. The project was stopped at such point because it will now require more knowledge of car insurances. But it will also require a bit more research on the potential market.

With these pieces of information, the system could be entirely specified (the historical statistics to use, the interface to put in place to interact with the contract, the rules to define for the community). One should also note that before publishing the contract on the blockchain it might be interesting to ask a jurist if the contract is in accordance with the laws of the nation into which the users are insured, but also to ensure that the system designer cannot be held responsible for any problem relative to the model.

More on the personal side, this project was a very rich experience. The LACAL lab let us be very autonomous in our work. Even though this proved to be quite a challenge to handle everything by ourselves, we think it made us grow more mature. We experienced for the first time the design and implementation of a system from beginning to end. We had the possibility also to organise our time and our roles as we wished. It enabled the two of us to really discover our strengths and weaknesses in a team and how to overcome them.

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] EtherScan. *EtherScan, The Ethereum Block explorer*. [Online; accessed 9-May-2017]. URL: `https://testnet.etherscan.io`.

[4] Ethereum Stack Exchange. *How can I securely generate a random number in my smart contract?* [Online; accessed 05-May-2017]. 2016. URL: `https://ethereum.stackexchange.com/questions/191/how-can-i-securely-generate-a-random-number-in-my-smart-contract`.

[5] Coalition Against Insurance Fraud. *By the numbers: Fraud Statistics*. accessed online on the 16th of May, available at `http://www.insurancefraud.org/`.

[6] Jim Manning. *Ropsten To Kovan To Rinkeby: Ethereum's Testnet Troubles*. [Online; accessed 9-May-2017]. URL: `https://www.ethnews.com/ropsten-to-kovan-to-rinkeby-ethereums-testnet-troubles`.

[7] 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/`.

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

[9] 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`.

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

[11] 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`.