

h e g

Haute école de gestion
Genève

Apports des *Smart Contracts* aux *Blockchains* et comment créer une nouvelle crypto-monnaie

Travail de Bachelor réalisé en vue de l'obtention du Bachelor HES

par :

Jimmy PARIS

Conseiller au travail de Bachelor :

Rolf HAURI, professeur

Secrétariat, 29 septembre 2017

Haute École de Gestion de Genève (HEG-GE)

Filière informatique de gestion

Déclaration

Ce travail de Bachelor est réalisé dans le cadre de l'examen final de la Haute école de gestion de Genève, en vue de l'obtention du titre Bachelor of Science HES-SO en Informatique de gestion.

L'étudiant atteste que son travail a été vérifié par un logiciel de détection de plagiat.

L'étudiant accepte, le cas échéant, la clause de confidentialité. L'utilisation des conclusions et recommandations formulées dans le travail de Bachelor, sans préjuger de leur valeur, n'engage ni la responsabilité de l'auteur, ni celle du conseiller au travail de Bachelor, du juré et de la HEG.

« J'atteste avoir réalisé seul le présent travail, sans avoir utilisé des sources autres que celles citées dans la bibliographie. »

Fait à Genève, le 28/09/2017

Jimmy Paris



Remerciements

En tout premier je tiens à remercier M. Hauri pour sa disponibilité et pour ses conseils éclairés. Merci de m'avoir donné un cadre précis et clair qui a grandement facilité mon travail de recherche.

Je tiens aussi à remercier M. Priftis pour m'avoir confié ce projet qui, je sais, lui tient à cœur. Merci de m'avoir fait confiance et permis de mettre en pratique mes connaissances.

Je tiens également à remercier ma famille pour sa patience et bienveillance à mon égard. Merci de m'avoir encouragé et supporté pendant ces longs mois de travail à la maison.

Résumé

Dans ce travail de recherche, est expliqué ce qu'est la *Blockchain* et son fonctionnement, mais aussi ce qu'est une crypto-monnaie et un *Smart Contract*. En particulier, quelles sont les possibilités et les problématiques émergentes avec l'arrivée des *Smart Contracts*. De plus, ce travail aboutit à la réalisation pratique d'un projet, utilisant des *Smart Contracts* pour créer une crypto-monnaie locale *Business to Business*, permettant aussi de faire des prêts à 0% d'intérêt. Ce projet est également conçu comme une aide montrant comment programmer des *Smart Contracts*. Il apportera les outils et les connaissances nécessaires dans le but de pouvoir créer, soi-même, sa propre crypto-monnaie ou toute autre application décentralisée.

Table des matières

Déclaration.....	i
Remerciements	ii
Résumé	iii
Liste des figures.....	vii
1. Introduction.....	1
2. Apports des <i>Smart Contracts</i> aux <i>Blockchains</i>.....	4
2.1 Qu'est-ce qu'une <i>Blockchain</i>?	4
2.1.1 Définition générale	4
2.1.2 Caractéristiques essentielles.....	5
2.1.2.1 Réseau pair à pair distribué	5
2.1.2.2 Transparence	6
2.1.3 Fonctionnement général.....	6
2.1.4 Définition d'une crypto-monnaie	8
2.1.4.1 Particularités du marché du Bitcoin.....	8
2.1.5 Mécanismes de consensus	9
2.1.5.1 Practical Byzantine Fault Tolerance (PBFT).....	9
2.1.5.2 Proof of Work (PoW) - Bitcoin	11
2.1.5.3 Proof of Stake (PoS) - PeerCoin	12
2.1.5.4 Delegated proof of stake (DPoS) - BitShares	14
2.1.5.5 Gestion des attaques et pannes.....	14
2.1.5.6 Remarques finales - mécanismes de consensus.....	15
2.1.6 Avantages, risques et limites.....	15
2.1.6.1 Principaux avantages	15
2.1.6.2 Principaux risques	16
2.1.6.3 Problématique de la gouvernance.....	17
2.1.6.4 Limites de <i>scripting</i> des <i>Blockchains</i>	18
2.2 Qu'est-ce qu'Ethereum?	20
2.2.1 Définition générale et innovations	20
2.2.2 Fonctionnement général.....	21

2.2.2.1	Casper – mécanisme de consensus	22
2.3	Qu'est ce qu'un <i>Smart Contract</i>?	24
2.3.1	Histoire des <i>Smart Contracts</i>	24
2.3.2	<i>Smart Contract</i> dans Ethereum	25
2.3.2.1	Programmable	26
2.3.2.2	Le code c'est la loi	27
2.3.2.3	Idéalement indépendant.....	28
2.3.3	Bénéfices actuels	29
2.3.3.1	Héritage des avantages de la <i>Blockchain</i>	29
2.3.3.1.1	Suppression d'intermédiaires	29
2.3.3.1.2	Inarrêtable.....	29
2.3.3.1.3	Borderless.....	29
2.3.3.1.4	Open source	29
2.3.3.2	Coder des conditions de paiement divers	30
2.3.3.3	Créer des <i>Tokens</i>	30
2.3.3.4	<i>KickStarter</i>	30
2.3.3.5	Lancer sa propre ICO	31
2.3.3.6	Créer sa propre organisation décentralisée et autonome (DAO).....	31
2.3.3.7	Multiplés utilisations alternatives	33
2.3.3.8	Plus performant que les contrats traditionnels	33
2.3.4	Quels sont les problèmes actuels?.....	34
2.3.4.1	Faibles de sécurité	34
2.3.4.2	Difficile de ne pas recourir à des services externes.....	36
2.3.4.3	Participations aux ICO difficilement régulables	36
2.3.4.4	Pas de garantie légale.....	37
2.3.4.5	Compréhension du code et détection des vulnérabilités	38
2.3.4.6	Rompres un engagement.....	38
2.3.4.7	Trop rigide pour correspondre à la loi actuelle.....	38
2.3.4.8	Mécompréhension des <i>use cases</i> réellement réalisables	38
3.	Comment créer une nouvelle crypto-monnaie?	39
3.1	Bonnes pratiques du programmeur en Solidity	39
3.1.1	Se préparer à l'échec	39

3.1.2	Passer en production par étapes et détecter les bugs en amont	39
3.1.3	Garder le contrat simple	39
3.1.4	Se tenir informé.....	40
3.1.5	Se méfier des propriétés de la <i>Blockchain</i>	40
3.2	Projet Chablex.....	41
3.2.1	Description du projet	41
3.2.1.1	Problématiques.....	42
3.2.1.2	Détails des solutions proposées.....	43
3.2.1.2.1	Standard d'échangeabilité (ERC20)	43
3.2.1.2.2	Distribution fairplay et exclusivement locale.....	43
3.2.1.2.3	Système autogéré par ses membres.....	44
3.2.1.2.4	Demande d'emprunt	45
3.2.1.2.5	Maximum empruntable	45
3.2.1.2.6	Demandes traitées automatiquement (FIFO).....	45
3.2.1.2.7	Prêt et remboursement.....	45
3.2.1.2.8	Récompense (augmentation du maximum empruntable)	47
3.2.2	Manuel développeur (lien GitHub)	48
3.2.3	Manuel utilisateur (lien GitHub)	48
3.2.4	Bilan.....	48
3.2.4.1	Etat du projet	48
3.2.4.2	Améliorations possibles.....	48
4.	Conclusion.....	49
	Bibliographie	51
	Annexe 1 : Manuel Développeur	55
	Annexe 2 : Manuel utilisateur.....	81
	Annexe 3 : <i>Smart Contracts</i>	90

Liste des figures

Figure 1 : Architectures de réseau.....	5
Figure 2 : Fonctionnement de la <i>Blockchain</i>	7
Figure 3 : Scénario 1 (<i>Practical Byzantine Fault Tolerance</i>)	10
Figure 4 : Scénario 2 (<i>Practical Byzantine Fault Tolerance</i>)	11
Figure 5 : Évolution du maximum empruntable (optimiste)	48

1. Introduction

En novembre dernier, lors du Forum de l'économie numérique, une dizaine d'exposants s'intéressaient "à l'impact de la *Blockchain* sur les PME genevoises : quelles en sont les applications concrètes et comment les appréhender." (Service de la promotion économique de Genève, 2016). Pour beaucoup d'étudiants de l'École Supérieure d'Informatique de Gestion, ce fut une toute première approche de ce qu'est une *Blockchain* et de ses possibles utilisations. Le sujet Ethereum a été abordé, sans cependant recevoir toute l'attention qu'il méritait.

Alors très récent et peu connu du grand public, Ethereum va par la suite exploser et révolutionner la *Blockchain* en introduisant de nouvelles possibilités. Avec Ethereum il est possible, non seulement de faire ce que la monnaie Bitcoin proposait déjà, mais offre également la possibilité de profiter des nœuds du réseau pour exécuter des programmes, appelés des *Smart Contracts*. Ceux-ci permettent de créer des applications basées sur un système décentralisé. Un grand nombre de développeurs se sont alors mis à étudier Ethereum et les *Smart Contracts*, en participant activement à la prospérité de cette communauté.

Depuis cette fin 2016, le monde de la crypto-monnaie s'est consolidé et est entré dans la cour des grands. Le prix du Bitcoin a atteint des records, son prix a plus que doublé depuis le début de l'année, et pourtant, il risque de ne plus être le numéro un des monnaies digitales. Le prix de l'Ether, la monnaie virtuelle qui prend vie grâce à la Blockchain Ethereum, est monté de manière fulgurante jusqu'à croître de 4'500 pourcents depuis le début de l'année. (Popper, 2017)

De nombreuses applications ont été créées grâce aux *Smart Contracts* de la *Blockchain* Ethereum, et certaines *initial coin offering* (se référant au terme boursier officiel "*inital public offering*") ont récolté des sommes astronomiques. Par exemple, le projet Bancor a levé 150 millions de dollars en une semaine, ou encore le projet *Basic Attention Token* a levé 35 millions de dollars en 30 secondes. Cette échelle d'investissement, ainsi que le nombre de nouveaux projets liés aux crypto-monnaies, ont atteint des records historiques. (Keane, 2017)

C'est précisément dans ce contexte que s'insère ce travail. Pour un informaticien de gestion, l'émergence des *Smart Contracts* semble prometteuse, en effet, Bitcoin n'offrait pas cette possibilité, mais Ethereum (avec Solidity) permet désormais de passer à l'action et de créer de nouvelles formes d'organisation monétaire.

Tout d'abord, ce travail de recherche permet au lecteur de mieux comprendre cet environnement novateur et d'appréhender le vaste éventail des possibilités qu'offrent les *Smart Contracts*. Plus encore, à partir de la mise en pratique de cette théorie à travers un projet concret, il est amené à se poser de vraies questions quant à la manière dont la société fonctionne, qu'est-ce qu'une communauté locale, comment promouvoir la confiance et comment faire pour que chacun y trouve son compte. Le lecteur pourra ainsi se forger une connaissance solide de l'environnement de la crypto-monnaie et apprécier par lui-même toute la force de ce mouvement.

Plus précisément, ce projet apporte une réflexion nouvelle sur le monde de la *Blockchain*. En règle générale, la *Blockchain* n'est pas prévue pour se prêter, étant donné que pour cela il faut se faire confiance. C'est justement cette question que soulève la création d'une plateforme de prêt à 0%, le challenge étant d'avoir un *Smart Contract* qui crée cet environnement de confiance. Grâce à cette plateforme, la communauté impliquée se rassemble, les gens se réunissent et le boulanger se trouve lié avec le maraîcher, le chef d'usine avec le petit entrepreneur. C'est dans cette bulle créée par le *Smart Contract* que cette expérience commune peut avoir lieu et qu'il est possible de dynamiser une localité.

Dans ce travail, le but est, dans un premier temps, de comprendre ce qu'est un *Smart Contract*, puis de savoir comment les créer et les utiliser. Ensuite, c'est également d'apprendre à créer une monnaie locale avec ses fonctions primaires d'envoi d'argent, en y ajoutant une logique de prêt pour les entreprises : transferts, partie membre, conditions à la participation ou encore facteur confiance.

Le but n'est pas de prétendre à une parfaite réalisation de ce projet, mais de poser les premières pierres. Au niveau de sa programmation, c'est ici une toute première version qui prétend poser les idées sous forme de code, tout en étant conscient du manque de temps pour la réalisation suffisante de tests. L'idée n'est donc pas de faire un projet parfait et prêt à l'utilisation, mais bien de poser les bases et d'établir une liste de ce qui reste à faire, pour qu'un jour il puisse (peut-être) voir le jour.

Pour cela, ce document se divise en deux parties :

La première partie vise à expliciter ce qu'est une *Blockchain* et à montrer les nouveautés apportées par les *Smart Contracts*. Pour cela, le point de départ est l'exemple du Bitcoin, ainsi qu'une explication sur les mécanismes de consensus populaire qui en ont découlé. Ensuite, une introduction à la *Blockchain* Ethereum et aux *Smart Contracts* permettra de comprendre autant leurs avantages en tant qu'évolution technique et leur place dans la société actuelle, que les problèmes qui en découlent.

La deuxième partie est consacrée au travail pratique, et donc au développement d'une première version d'une monnaie locale grâce à laquelle il serait possible de prêter et d'emprunter de l'argent à 0% d'intérêt. Ensuite, les problématiques sous-jacentes de ce projet seront explicitées, ainsi que leurs solutions proposées, développées et documentées.

Pour conclure, il sera proposé un bilan de l'état actuel du projet, dans le but de pouvoir le concrétiser par la suite mais aussi pour le rendre accessible et réutilisable pour des applications futures.

2. Apports des *Smart Contracts* aux *Blockchains*

2.1 Qu'est-ce qu'une *Blockchain*?

2.1.1 Définition générale

Une *Blockchain* est un registre décentralisé qui est géré entre pairs et sans l'intervention d'une autorité centrale, garantissant ainsi la transparence et la sécurité du système. Afin de visualiser son fonctionnement, on peut y voir un grand registre - ou un grand livre - où tout est noté jusqu'au moindre détail. Il est donc possible de garder une traçabilité de tous les échanges, et de s'assurer de l'état des comptes, de la même manière que le ferait un audit comptable. La seule différence est qu'une *Blockchain* n'est pas un grand livre, mais autant de registres que de participants, assurant tous la sécurité et la disponibilité de l'information.

Un participant du réseau de la *Blockchain* est considéré comme un "nœud". Un "*full node*" est un nœud dont la particularité est de détenir l'information du registre dans son intégralité, alors qu'un "*partial node*" ne détient pas d'information, mais passe par un intermédiaire *full node* afin de réaliser sa transaction, ou tout simplement de consulter l'information contenue dans la *Blockchain*.

Une *Blockchain* est donc une base de données contenant tout l'historique des échanges réalisés depuis sa création, et partagée par ses différents utilisateurs. Chaque participant possède une copie de la *Blockchain*, permettant au système de marcher de pair à pair (*peer to peer*) et donc d'éliminer l'autorité centralisée. L'absence d'intermédiaire et la distribution de l'information sont les garants de la sécurité de la chaîne.

Il existe une différence entre une *Blockchain* publique et privée. La première est accessible par tout le monde et il est possible de consulter l'intégralité de l'information. La seconde ne permet pas la lecture complète de certaines informations (plus de confidentialité, mais moins de transparence et donc de sécurité), ou limite le nombre d'acteurs qui possèdent les droits nécessaires.

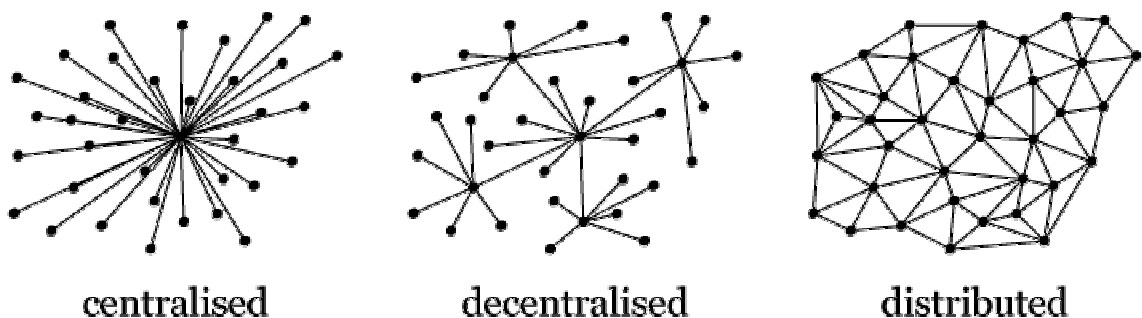
2.1.2 Caractéristiques essentielles

2.1.2.1 Réseau pair à pair distribué

L'autorité centrale du système est supprimée du réseau, permettant ainsi de distribuer cette fonction entre les participants. Le réseau fonctionne ainsi de pair à pair, de sorte que personne n'est au-dessus des lois, sans qu'il soit nécessaire de faire confiance aveuglément aux autres participants. Nul ne peut effacer ou falsifier les informations de la *Blockchain*, car le réseau pair à pair va veiller à ce que la *Blockchain* respecte certaines règles cryptographiques garantissant son contenu. De plus, ce type de configuration permet de supprimer les intermédiaires et donc de réduire les coûts de transfert, étant donné que moins de personnes toucheront une commission sur chaque opération.

Un réseau distribué a également l'avantage d'être difficilement arrêtable. Si on le compare à un grand filet, lorsque le nœud d'une intersection est défait, tous les autres resteront liés entre eux, et le réseaux continuera à fonctionner sans problème.

Figure 1 : Architectures de réseau



(Krawisz, 2013)

Finalement, un réseau pair à pair distribué permet de favoriser la disponibilité de l'information (quelle soit accessible et cela rapidement), étant donné que le contenu n'est pas stocké dans un serveur central. Le choix de favoriser la disponibilité et de distribuer ou de décentraliser l'information empêche de garantir la cohérence des données à un instant précis. Ceci est induit du théorème CAP qui définit qu'entre la disponibilité, la cohérence des données et la tolérance au partitionnement, il n'est possible de maximiser que deux de ces trois caractéristiques. (Gilbert et Lynch, 2002)

2.1.2.2 Transparence

Une *Blockchain* est généralement open source, garantissant à la communauté que celle-ci fonctionne dans les règles énoncées et qu'elle est accessible à tous. En d'autres termes, n'importe qui devrait avoir le droit d'utiliser et de maintenir un réseau *Blockchain*.

Une *Blockchain* est aussi accessible à tous pour garantir la sécurité de ses participants. Un exemple cité lors du Forum Economie Numérique "Transition numérique : la *Blockchain* et ses opportunités pour les entreprises", présenté en Novembre 2016, est celui d'une ville où l'injustice règne et où même la police (autorité centrale) brutalise souvent les personnes interpellées au commissariat. Plus personne ne peut leur faire confiance, alors, afin de redonner aux citoyens le sentiment de sécurité, une idée serait de créer un commissariat entièrement en verre pour que tout le monde puisse voir ce qui se passe à l'intérieur et observer directement une éventuelle forme d'injustice. (Epié et Loubet, 2016) Cette idée s'applique également à la *Blockchain*, où il est accessible à tous de demander l'historique des transactions.

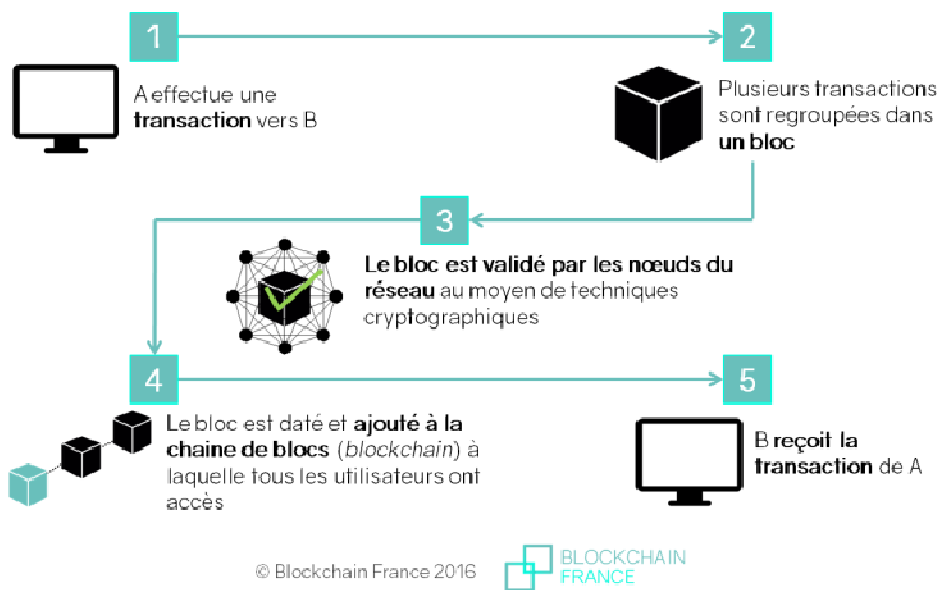
La *Blockchain* est donc transparente; quiconque peut "l'auditer", c'est-à-dire vérifier son état à tout moment. Une affirmation telle que "la transaction a été envoyée de l'adresse A vers l'adresse B" est vérifiable dans la *Blockchain*, ce qui permet de régler d'éventuels conflits. Par exemple, si on prétend avoir envoyé de l'argent sans le faire en réalité, tout le monde peut voir que le destinataire n'a rien reçu.

2.1.3 Fonctionnement général

Il est maintenant temps de détailler le fonctionnement d'une *Blockchain*, afin de poursuivre cette recherche en ayant construit une base de compréhension commune. Dans un premier temps, "les transactions effectuées entre les utilisateurs du réseau sont regroupées par blocs. Chaque bloc est validé par les nœuds du réseau appelés les "mineurs", selon des techniques qui dépendent du type de *Blockchain*." (Blockchain France, 2016)

Ce qui est ici appelé mineur correspond aux *full nodes*, ces utilisateurs qui détiennent l'intégralité de l'information afin de pouvoir valider la véracité de la ou les transaction-s regroupée-s dans le futur bloc. Cette étape nécessaire passée, ce dernier va être ajouté à la chaîne de blocs déjà existante.

Figure 2 : Fonctionnement de la *Blockchain*



(Blockchain France, 2016)

Lorsqu'une transaction est ajoutée à la *Blockchain*, il n'est pas possible de la modifier et l'information est très difficilement falsifiable. Cette base de données décentralisée ne peut donc pas être piratée ou manipulée, ou en tout cas pas aussi facilement que si elle était gérée de manière centralisée.

Par contre, pour rendre cette organisation accessible à tous, en préservant l'anonymat de chacun, il n'existe pas de contrôle d'accès qui contrôlerait l'identité des utilisateurs. Il est, en principe, impossible de savoir si celui-ci est une personne, une organisation ou un programme informatique. Chaque nœud peut, potentiellement, soumettre une transaction (c'est à dire ajouter de l'information sur la *Blockchain*) sans savoir, au préalable, si l'opérateur est digne de confiance ou non.

C'est parce que nous ne sommes pas sûrs de pouvoir faire confiance aux participants que le consensus est primordial. Avant que l'information soit ajoutée de manière permanente à la *Blockchain*, elle doit être étudiée et validée par les nœuds distribués dans le réseau. Car c'est seulement lorsque le consensus est atteint que le "bloc" peut être ajouté à la "chaîne".

2.1.4 Définition d'une crypto-monnaie

Il est essentiel de comprendre que la naissance même des *Blockchains* provient de l'émergence du Bitcoin. Créé par une mystérieuse personne surnommée Satoshi Nakamoto, le Bitcoin est une monnaie virtuelle, c'est-à-dire qu'elle n'existe pas physiquement (comme l'or ou les billets de banque). Contrairement à l'argent sur un compte bancaire géré par un serveur centralisé, cette monnaie dite numérique est gérée par un système *Blockchain*, profitant ainsi de toutes ses caractéristiques précédemment détaillées.

Une crypto-monnaie est une part de valeur dans une *Blockchain* qui est donc garantie par des règles cryptographiques. Elle est la force motrice de la *Blockchain* car elle motive les participants à la réalisation de certaines tâches nécessaires, (comme par exemple pour le maintien du réseau, appelé "minage"), mais elle sert également à rémunérer les développeurs, les chasseurs de bugs, etc.

2.1.4.1 Particularités du marché du Bitcoin

Pour mieux comprendre comment est déterminé le prix de chaque crypto-monnaie, il faut commencer par comprendre comment se régule le marché du Bitcoin.

Le Bitcoin n'appartient à personne, à aucun gouvernement ni à aucune banque, c'est un outil commun dont la gestion est assurée collectivement par ceux qui l'utilisent. Le prix du Bitcoin dépend de l'offre et de la demande et se situe au point d'équilibre. Si la demande augmente le prix augmente et si elle diminue le prix diminue (de manière semblable au cours d'une action en bourse).

La masse monétaire est strictement encadrée, le nombre total de Bitcoins est fixé à 21 millions, ce nombre faisant partie du protocole lui-même. D'un point de vue économique, le Bitcoin se comporte comme l'or, parce qu'il se base sur des ressources épuisables. Cependant, contrairement à une monnaie classique, le Bitcoin a l'avantage de pouvoir être divisé, non seulement en centimes, mais en infimes unités, la plus petite étant le Satoshi valant 0,00000001 BTC.

Créer de l'inflation dans un système comme celui-ci est très simple, car la monnaie est *unbacked*, c'est à dire que pour 1 Bitcoin en circulation il n'y a pas de garantie de valeur en or ou en dollars dans un coffre. Créer de l'inflation permet de rémunérer en nouveaux Bitcoins les *full nodes* qui participent activement au maintien du réseau (appelés de ce fait des mineurs).

Le système a tenté de réduire cette inflation en introduisant des frais de transactions. C'est aussi grâce à ces frais que vont pouvoir être en partie rémunérés les mineurs qui ont remporté le droit de créer un "bloc". En effet le simple fait d'éviter de mettre trop de nouveaux Bitcoins en circulation permet de limiter l'inflation. (Bitcoin.org, s.d.)

Il existe de nombreuses crypto-monnaies alternatives au Bitcoin. Certaines sont des reprises de Bitcoin incluant des innovations significatives, d'autres sont de vulgaires copies inutilisées et considérées comme des escroqueries servant à faire gagner de l'argent à ses créateurs et investisseurs initiaux. La plupart des crypto-monnaies et des plateformes où elles sont échangées sont indexées sur le site internet www.coinmarketcap.com.

2.1.5 Mécanismes de consensus

Dans une *Blockchain*, ainsi que dans tout type de système décentralisé, il existe quatre méthodes de base pour arriver à un consensus : l'algorithme de "*practical byzantine fault tolerance*" (PBFT), "*proof-of-work*" (PoW), "*proof-of-stake*" (PoS), et finalement "*delegated proof-of-stake*" (DPoS). (Hammerschmidt, 2017)

2.1.5.1 Practical Byzantine Fault Tolerance (PBFT)

Le "*byzantine fault tolerance*" (BFT) a d'abord été décrit dans un article publié par Leslie Lamport, Robert Shostak et Marshall Pease en 1982, où ils proposent plusieurs solutions au problème des généraux byzantins. Ce dernier part de l'allégorie suivante :

« Imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must decide on when to attack the city, but they need a strong majority of their army to attack at the same time. The generals must have an algorithm to guarantee that (a) all loyal generals decide upon the same plan of action, and (b) a small number of traitors cannot cause the loyal generals to adopt a bad plan. The loyal generals will all do what the algorithm says they should, but the traitors may do anything they wish. The algorithm must guarantee condition (a) regardless of what the traitors do. The loyal generals should not only reach agreement, but should agree upon a reasonable plan. » (Lamport, Shostak et Pease, 1982)

Bien que plusieurs solutions aient été apportées au problème, c'est en 1999 que Miguel Castro et Barbara Liskov (2002) introduisent l'algorithme "*Practical Byzantine Fault Tolerance*" (PBFT), utilisé pour atteindre un consensus entre tous les "généraux" ou participants. Sans prétendre expliquer l'algorithme dans toute sa complexité, la solution proposée implique que chaque "général" garde un registre personnel de tous

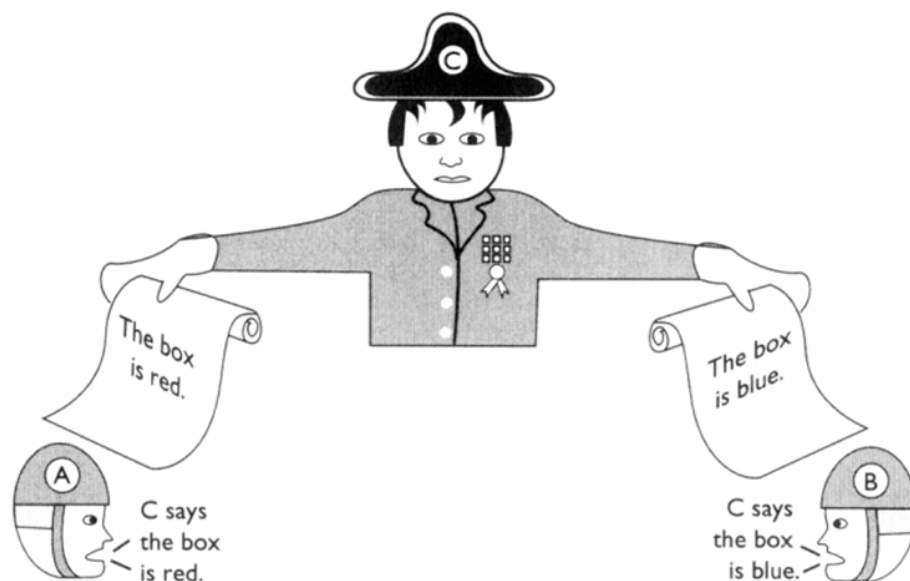
les échanges d'information, et lorsqu'un nouveau message est reçu ("attaquer" ou "se retirer", dans sa version simple), le "général" participant confronte l'information à son registre personnel selon un système d'évaluation qui lui dira quoi faire. L'information sera ensuite soumise aux autres "généraux" participants, permettant ainsi d'atteindre un consensus basé sur la totalité des informations soumises dans le système. (Hammerschmidt, 2017).

Pour faire le lien entre cette description et la *Blockchain*, chaque général représente un nœud; certains généraux sont loyaux (ils travaillent dans les règles pour le bien de la communauté) et d'autres sont des traîtres (ils tentent de faire échouer le consensus). Un algorithme de type byzantin doit garantir que, quel que soit le vote des traîtres, les validateurs se mettent d'accord sur la validation d'un block "raisonnable".

Voici, en image, les 2 scénarios possibles quand, parmi 3 généraux, se trouve un traître. Dans ces 2 cas, aucune décision ne peut être considérée comme valide.

Scénario 1 : le général C (qui représente le nœud qui propose le bloc) est un traître.

Figure 3 : Scénario 1 (*Practical Byzantine Fault Tolerance*)

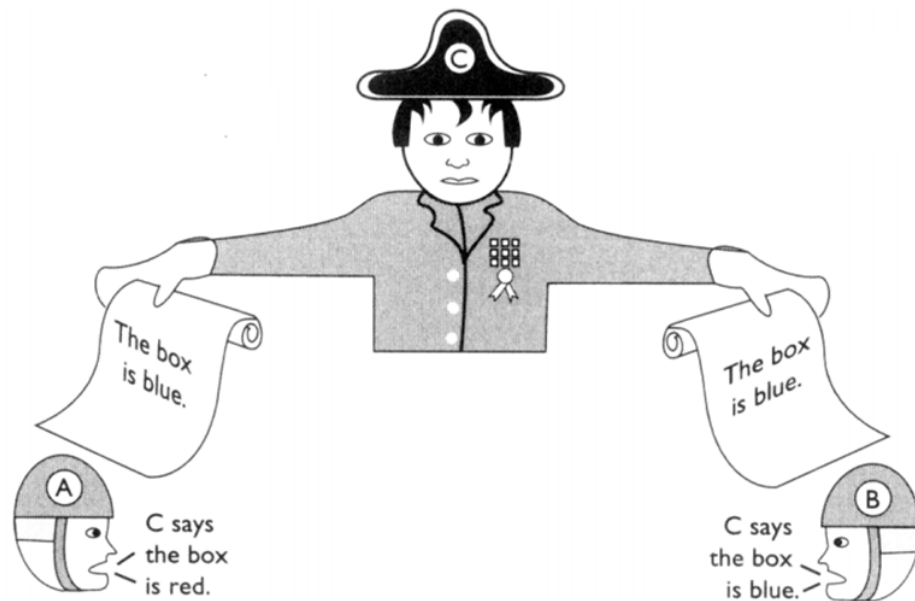


(Lesert, 2002)

Le général C a réussi à empêcher les autres généraux de prendre une décision raisonnable. Il leur faudra dans ce cas au moins un général de plus pour déterminer si la boîte est rouge ou bleue.

Scénario 2 : même problème si le général A (qui représente le validateur) est un traître.

Figure 4 : Scénario 2 (*Practical Byzantine Fault Tolerance*)



(Lesert, 2002)

On peut donc en conclure qu'il suffit qu'une personne sur trois soit un traître pour que la décision prise ne soit pas valide. On peut donc considérer que, si le nombre de traîtres est supérieur ou égal à $1/3$ du nombre total des généraux, aucune décision ne peut être prise de manière fiable. Cette conclusion sera déterminante dans la lutte contre les pannes et les différentes attaques des systèmes *Blockchain*. (Lesert, 2002)

2.1.5.2 Proof of Work (PoW) - Bitcoin

Bien que certaines *Blockchains* se basent sur le PBFT, c'est le système ou protocole "*Proof-of-Work*" (PoW) qui est le plus connu et utilisé par Bitcoin. À la différence de PBFT, PoW permet aux participants de ne pas avoir besoin des autres participants pour arriver au consensus. PoW utilise une fonction de hachage (*hash*) pour déterminer si un participant (aussi appelé mineur) a le droit d'annoncer aux autres sa conclusion sur la transaction, puis de l'ajouter ou non à son registre. Les autres participants vont, de manière indépendante, valider cette conclusion et l'ajouter ou non. C'est ce qui fait toute la différence avec PBFT où il suffisait qu'au moins $1/3$ des participants soient des traîtres pour que la prise de décision soit impossible; alors qu'avec PoW, il faudrait qu'au moins $1/2$ de la totalité de puissance de calcul (*Central Processing Unit* - CPU) soit utilisée par des traîtres.

PoW est prévu de telle sorte que les participants récupèrent toujours la plus grande chaîne de blocs et donc le registre des participants les plus performants.

C'est pour cela que la fonction de *hash* a été introduite et rend difficile de corrompre la *Blockchain*. Elle est utilisée pour donner un résultat de plus en plus dur à trouver et vérifiable rapidement en termes de puissance de CPU, c'est ce qu'on appelle "la difficulté".

Pour s'assurer que cette compétition de puissance CPU ne soit pas utilisée pour corrompre la *Blockchain*, une garantie supplémentaire est prévue : l'introduction d'une motivation pour les mineurs (un revenu en Bitcoin pour tout participant gagnant qui obtient le droit de conclure une transaction et de la faire valider par chaque nœud du réseau indépendamment). Les mineurs ont donc tout intérêt à gagner de l'argent en minant pleinement dans les règles et à garantir ainsi la sécurité du système. (Young, 2017)

2.1.5.3 Proof of Stake (PoS) - PeerCoin

Il existe aussi une troisième alternative, elle aussi bien connue, qui est le *Proof-of-Stake*. Elle vise à remplacer la fonction de *hash* par une simple preuve de possession de valeur. Dans ce cas, les mineurs ne sont pas rémunérés en fonction de leur puissance CPU, mais plutôt en proportion du montant qu'ils possèdent et mettent à disposition pour miner virtuellement (valider des blocs). Il suffit donc, dans cette version, de posséder un *wallet* (un porte-monnaie virtuel) ; même un ordinateur peu puissant peut faire l'affaire, ce qui a l'avantage de consommer moins d'énergie.

En d'autres mots, ce n'est donc plus les participants qui font une compétition de puissance de calcul dans l'unique but de définir qui va être autorisé à valider le bloc comme dans PoW, cette méthode étant extrêmement énergivore à cause de tous ces ordinateurs travaillant juste dans le but de remporter le challenge. Mais cette pratique est remplacée par PoS qui fonctionne comme une loterie (celui qui a plus de tickets aura plus de chances de remporter le lot). Ce mécanisme autorisera le nœud gagnant, que l'on peut considérer comme mineur virtuel, à créer un bloc. Donc les chances d'être élu sont proportionnelles au montant que le nœud possède. (Preuve d'enjeu, s.d.)

PoS est initialement utilisé par le protocole *PeerCoin* où le minage virtuel est appelé "*minting*" au lieu de "*mining*" et où les mineurs virtuels sont rémunérés par la création de nouveaux *Peercoin* en fonction de l'ancienneté des *coins* qu'ils possèdent en tant que *full nodes*.

Cette rémunération se base sur le temps écoulé depuis que ses *coins* sont dans le *wallet* du *full node*, de telle sorte qu'il reçoit au total 1% d'intérêt par an. (King et Nadal, 2012). Certaines *Blockchains* garantissent d'autres retours sur investissement pour les mineurs virtuels selon ce que les développeurs de la *Blockchain* en question ont décidé.

Cependant, ce mécanisme sensé être robuste et garantir l'incorruptibilité du système n'est, en fait, qu'un mécanisme qui favorise le profit des acteurs déjà fortement impliqués, portant préjudice aux nouveaux participants.

Ce système reste donc manipulable par de riches acteurs qui, puisque leur puissance est proportionnelle à leur fortune et non plus à la puissance de leurs ordinateurs, ont pu s'affranchir des contraintes physiques liées à l'achat de machines coûteuses qui, selon la loi de Moore, perdent chaque 18 mois la moitié de leur valeur jusqu'à devenir obsolètes. (Moore, 1965)

C'est pourquoi d'autres algorithmes ont été mis au point par les adeptes de PoS afin de forcer la dispersion du pouvoir, et ainsi d'éviter les cartels, tout en gardant l'idée de motiver les grands possesseurs de la monnaie par une rémunération. Le but recherché étant d'être toujours plus rentable et plus écologique par rapport au PoW, sachant que le coût électrique pour les mineurs en PoW de Bitcoin est estimé à 1/3 du coût total du minage, ce qui non seulement coûte cher, mais a une lourde répercussion sur l'environnement. (Gubik, 2017)

2.1.5.4 Delegated proof of stake (DPoS) - BitShares

DPoS (utilisé par exemple par la *Blockchain* BitShares) est une version qui est, à la base, prévue pour limiter la prise de pouvoir par ceux qui possèdent de grandes parts de valeur, et ainsi obtenir une meilleure répartition entre tous les acteurs. C'est dans ce but que le système DPoS a introduit la possibilité de déléguer son pouvoir de vote à un tiers. Cela permet aux petites parties prenantes de s'associer afin d'avoir un plus grand poids et représenter un tout décisionnel pouvant concurrencer les autres parties prenantes plus fortes. Malheureusement, il est facile de comprendre le désavantage de ce mécanisme de consensus qui permet aux nouveaux arrivants de s'associer, prenant, ainsi, vite de l'importance, accélérant au final encore plus la centralisation de la prise de décision. Ce problème était déjà présent dans PoW, puisque les petits s'associaient en *pool* de minage.

2.1.5.5 Gestion des attaques et pannes

Il existe, malheureusement, plusieurs types d'attaques et de pannes pouvant nuire au système de la *Blockchain*.

Une attaque de type 51 % est lorsqu'un mineur, ou le plus souvent un *pool* de minage, possède plus de la moitié de la puissance de calcul du réseau. L'attaquant est alors en mesure de tricher sur la valeur des transactions en en créant de nouvelles, ou alors en excluant ou modifiant l'ordre. L'attaquant peut ensuite construire secrètement une chaîne plus longue que celle produite par la communauté des mineurs. Une fois rendue publique, puisque le protocole impose de choisir la chaîne la plus longue, la chaîne de l'attaquant prendra alors la place de celle constituée par les mineurs travaillant dans les règles. Les *hackers* peuvent ainsi créer et confirmer leurs propres blocs (qui sont frauduleux) et ceci avec une telle rapidité que le reste du réseau verraient leurs blocs (résultants d'un travail honnête) invalidés ; ces derniers pourraient se retrouver en perte, de pouvoir de décision (sur l'acceptation des transactions) et financièrement. (Schneider, Durand-Garçon, Masurel et Lorcery, 2016)

Dans PoW, pour tricher il faudrait que le ou les attaquant-s possède-nt plus de la moitié de la puissance de calcul s'ils veulent avoir le pouvoir de créer un bloc sur une mauvaise chaîne. La punition pour un tel comportement est implicite au fonctionnement même de PoW: les attaquants doivent acquérir de nouvelles machines ou louer du matériel supplémentaire s'ils veulent posséder plus de 50% de la CPU. Or, selon la loi de Moore, les ordinateurs perdent la moitié de leur puissance environ tous les deux ans, ce qui rend une attaque de type 51% de plus en plus coûteuse.

Dans PoS, même si quelqu'un venait à posséder 51 % de la crypto-monnaie, ce ne serait pas du tout dans son intérêt d'attaquer le système dans lequel il a investi la majorité de sa fortune. Il serait vraiment improbable qu'une personne soit intéressée par acheter plus de la moitié des parts mises en jeu et ne soit pas intéressée de les voir fructifier le plus rapidement possible.

2.1.5.6 Remarques finales - mécanismes de consensus

Il est important de comprendre ces mécanismes utilisés par les différentes crypto-monnaies et leurs fonctionnements, pour pouvoir participer, en bénéficier et concevoir de nouveaux mécanismes de consensus.

Ces mécanismes servent principalement à distribuer les parts entre les participants de manière équitable en fonction de leurs efforts, mais aussi de leur permettre de participer à la prise de décision sur une *Blockchain* (principalement ici sur le nouveau bloc à ajouter à la chaîne, mais cela peut varier selon l'objectif).

Remarquez que, dans chaque cas, les participants ont une motivation spéciale à participer dans les règles, car il est prévu qu'ils reçoivent de nouvelles parts de valeur s'ils jouent le jeu.

2.1.6 Avantages, risques et limites

2.1.6.1 Principaux avantages

Les principaux avantages de la *Blockchain* sont l'immutabilité et la transparence des informations inscrites dans cette grande base de données distribuées.

Ainsi, les informations sont toujours disponibles, vérifiables, infalsifiables et ineffaçables. De part le nombre de nœuds (et donc de registres) dans un réseau *Blockchain*, l'information devient accessible rapidement. Grâce à la disponibilité de la totalité des transactions, il est possible à tous de consulter l'état des comptes et de réaliser un audit complet. Ce système n'est pas dirigé par une autorité centrale et fonctionne indépendamment des frontières, il garantit l'anonymat des utilisateurs et participants du réseau qui, grâce à leur fonctionnement *peer-to-peer*, permet de supprimer les intermédiaires, réduire les frais de transactions et sécuriser les échanges.

Corrompre le système impliquerait une coordination de plus de la majorité de la puissance CPU total des mineurs pour arriver au consensus, cette situation pourrait être rendue encore plus difficile en combinant différents mécanismes de consensus.

2.1.6.2 Principaux risques

Malheureusement, les systèmes *Blockchains* tels qu'ils sont réalisés aujourd'hui comptent bien des failles, des cases d'ombres et des vulnérabilités. Cependant, bien que certaines failles persistent dans les systèmes *Blockchains*, il est préférable de voir la *Blockchain* tel un château fortifié impénétrable, et que celui qui voudrait attaquer ses hôtes devrait leur tendre un piège sur le chemin arrivant au château, car une fois à l'intérieur ils ne risquent plus rien. Le moment le plus vulnérable est donc lorsqu'on utilise des services extérieurs à la *Blockchain*, le seul moyen d'avoir une garantie est donc de se référer à la réputation du fournisseur de service.

Il faut, cependant, rester vigilant, car certaines personnes mal intentionnées copient un site officiel où ils changent uniquement l'adresse publique pour la transaction grâce à laquelle ils proposent une demande d'envoi de Bitcoins et ainsi ils détournent aisément ces transactions.

Il existe aussi d'autres escroqueries basiques telles que de faux services recevant des Bitcoins, mais qui, en réalité, n'ont nullement l'intention d'offrir un service en retour et proposent souvent de fausses garanties sur investissement, etc.

Ces difficultés rendent les utilisateurs de *Blockchains* méfiants et en alerte permanente car ils ne savent jamais si ce qui leur est proposé est un service qui tient la route ou une escroquerie.

Il est également important de savoir que la plupart des plateformes d'échanges contrôlent les clés privées des *wallets* à la place de leurs utilisateurs. Cela leur permet, dans un système PoS, de voter à la place d'autres utilisateurs ou encore de récupérer les envois non tolérés. Mais surtout, en possédant les clés privées de leurs utilisateurs, ils peuvent prendre l'argent à leur place, ce qui, dans un système aussi sécurisé que la *Blockchain*, laisse fortement à désirer. D'innombrables piratages ont déjà eu lieu (mtGox, btc-e, etc...).

Trop vite associés au *darknet* et à ses mauvais usages (et surtout après la fermeture du site internet de trafics en tous genres nommé *SilkRoad*) ces services illégaux payés en crypto-monnaies ont donné des arguments à ceux qui soutiennent une politique anti-Bitcoin. L'argument est qu'il est difficile d'identifier réellement les utilisateurs d'adresses publiques de Bitcoins.

Il n'est pas à exclure que la *Blockchain* puisse tout de même être un jour corrompue, d'une manière ou d'une autre, en fonction du mécanisme de consensus utilisé (par exemple 51% pour PoW).

2.1.6.3 Problématique de la gouvernance

Venant du latin *gubernare*, gouverner reviendrait à “diriger le navire” et désignerait donc l'ensemble des règles et organes de décisions permettant d'assurer le bon fonctionnement de l'organisation en question. Ce sont aussi les mécanismes mis en place afin de faire respecter ces règles de surveillance comme de répression, qui s'assurent du contrôle et du respect des lois.

Mais, dans un réseau décentralisé *Blockchain*, qui se charge de gouverner? Savoir qui fait les règles est tout aussi important de savoir qui les fait respecter ; par exemple le protocole SWIFT est appliqué par SWIFTNet, utilisant un serveur centralisé et dirigé par un conseil d'administration. Ceci soulève la question suivante : la *Blockchain* est-elle libre de toute gestion humaine contrairement aux systèmes centralisés?

Le professeur Vili Lehdonvirta soutient qu'il y a un paradoxe dans la gouvernance de la *Blockchain*, car celle-ci se veut régie par des règles totalement impartiales, alors que les règles elles-mêmes peuvent être changées (2016). En effet, il est difficile de prétendre pouvoir changer les règles tout en assurant que ces règles sont applicables indéfiniment. La réalité de ce qui se passe dans Bitcoin montre que l'influence des mineurs est considérable, et qu'un désaccord peut aboutir à un “*hard fork*”, ou séparation lors de la modification des règles. Le système Bitcoin est reconnu comme étant limité dans la flexibilité pour être robuste dans la sécurité. En effet, changer des règles nécessite une majorité quasi absolue, ce qui rend les règles quasiment interchangeable, à moins de devenir un *hard fork*.

On peut ici citer comme exemple ce qui s'est passé lorsque certains ont décidé de changer les règles du Bitcoin, en créant une nouvelle chaîne branchée sur la chaîne actuelle du Bitcoin. Ce sont deux différentes branches qui ont le même passé mais sont séparées, avec chacune ses propres règles. Lors de cette séparation (*hard fork*), la nouvelle chaîne a pris le nom de Bitcoin Cash et a vu la taille de ses blocs passer de 1 MB à 8 MB (Renard, 2017). Ceux qui possédaient 1 Bitcoin possèdent désormais aussi 1 Bitcoin Cash, mais, bien qu'ils aient un passé commun, les comptes, les mineurs et les règles sont bien distinctes. Bitcoin n'a donc pas totalement éliminé la gestion humaine de son protocole. Comme nous allons le voir par la suite, il sera laissé une plus grande liberté aux utilisateurs, comme de programmer de nouveaux

mécanismes de consensus et de définir ses règles. Les humains gouvernent encore bien l'ensemble des règles mises en œuvre par cette technologie, dont le respect définit le réseau lui-même.

Il convient donc de garder un esprit critique quant au problème de la gouvernance d'un système décentralisé. Sur un ton provocateur, le professeur Vili Lehdonvirta écrit:

« Regardless of the model, my point is that Blockchain technologies cannot escape the problem of governance. Whether they recognize it or not, they face the same governance issues as conventional third-party enforcers. You can use technologies to potentially enhance the processes of governance (eg. transparency, online deliberation, e-voting), but you can't engineer away governance as such. All this leads me to wonder how revolutionary Blockchain technologies really are. If you still rely on a Board of Directors or similar body to make it work, how much has economic organization really changed? » (2016)

Le paradoxe de la gouvernance dans une *Blockchain* implique qu'il est nécessaire, à priori, d'avoir confiance en quelqu'un qui crée les règles, et ensuite qu'un jeu complexe d'influences fasse appliquer ces règles et/ou les modifie. S'attaquer au problème de la gouvernance réduit le caractère purement mathématique de la *Blockchain* souvent mis en avant. Et maintenant, si celle-ci n'est qu'une remplaçante de l'organe de gestion en charge des règles du jeu, est-elle vraiment nécessaire?

Le professeur Vili Lehdonvirta soutient que *"once you address the problem of governance, you no longer need Blockchain, (...) once you master it, you no longer need it"* (2016). Peut-être que la révolution annoncée n'est pas si significative. Ou peut-être que cette technologie n'est pas si différente des autres et, bien qu'elle apporte de grands avantages, il convient de l'aborder comme un outil qui, obligatoirement, est utilisé pour et par des personnes. Le consensus atteint à travers le réseau décentralisé *Blockchain* n'est pas une simple formule mathématique, mais suit un processus qui dépend des règles mises en place, et est donc forcément d'une gestion humaine.

2.1.6.4 Limites de *scripting* des *Blockchains*

Les *Blockchains*, avant l'arrivée des *Smart Contracts*, étaient utilisées seulement pour effectuer des transferts de valeur tel que Bitcoin. Il est possible d'écrire des scripts pour Bitcoin en FORTH, un langage totalement basé sur le mécanisme de la pile et qui est donc écrit dans une notation dite "polonaise inverse", ce qui est l'ordre naturel des opérations à exécuter pour une machine. Ce langage est, en revanche, moins naturel à lire pour l'être humain. Par exemple au lieu d'écrire $3 * 2 + 1$ on devra écrire $3 2 * 1 +$, ce qui peut vite devenir complexe si le script est volumineux. (Script, s.d.)

D'un côté, ce système permet de créer, par exemple, un compte multi-signatures où 3 personnes devront toutes autoriser un transfert pour que le transfert soit émis. D'un autre côté, ce langage ne permet pas d'écrire des boucles, offrant la garantie de ne pas se retrouver dans une boucle infinie (fort pratique pour les mineurs qui veulent exécuter ce script pour, par exemple, une transaction). Par contre, ce langage est bien moins pratique lors de la conception d'un algorithme, car, s'il faut 256 itérations, alors il est nécessaire de l'écrire 256 fois dans le script, ce qui le rallonge considérablement et le rend encore plus difficile à lire.

Le *scripting* connaît le problème appelé "*value-blindness*" car il ne permet pas de contrôler de manière précise le montant à retirer. L'unique astuce est de multiplier les UTXO (*Unspend Transaction Output*) afin de répondre à tous les cas envisagés (par exemple un pour retirer 10 Bitcoins, un autre pour retirer 20 Bitcoins, etc...).

Il manque également fortement d'états (*lack of state*). En effet un UTXO ne peut que être *unspend* ou *spend*, ce qui pose problème. Il serait préférable qu'une valeur en crypto-monnaie puisse passer dans d'autres états intermédiaires tels qu'être "gelée un certain temps" ou être "en attente de confirmation" dans un système de confirmation en deux phases. Il existe autant d'états possibles et imaginables que de cas d'usage et il est parfois difficile de s'en priver.

Les méta-protocoles permettent de faire plus, tout en inscrivant les données dans des transaction Bitcoins, ce système est donc dépendant de l'implémentation du Bitcoin et risque donc d'avoir des problèmes de maintenabilité. De plus, ces méta-protocoles sont difficilement implémentables à cause des problèmes de *value-blindness* et de *lack of state*. De plus, ils stockent la totalité de l'histoire de la *Blockchain* en se basant souvent sur un intermédiaire de confiance, pratique qui est normalement à proscrire dans l'univers *Blockchain* si on veut ne pas devoir faire confiance à une entité spécifique.

Le scripting peut donc être standardisé sans trop de difficultés, mais il reste un outil très limité, alors que les méta-protocoles permettent d'aller plus loin mais rencontrent des difficultés pour pouvoir monter en charge (être massivement utilisé).

C'est donc la raison pour laquelle Vitalik Buterin a décidé de créer une nouvelle *Blockchain* appelée Ethereum; une solution qui prendra du temps, mais bénéficiera de possibilités illimitées dans sa conception et son futur fonctionnement. (Buterin, 2013a)

2.2 Qu'est-ce qu'Ethereum?

2.2.1 Définition générale et innovations

Ethereum est un protocole de *Blockchain* imaginé par Vitalik Buterin publié dans son *whitepaper* en décembre 2013. Cette *Blockchain* est lancée depuis le 30 juillet 2015 et a fait beaucoup parler d'elle depuis ce jour. C'est une *Blockchain* qui permet de réaliser des transactions comme le fait le protocole Bitcoin, mais avec une crypto-monnaie appelée Ether, aussi symbolisée par ETH.

Mais Ethereum va au-delà de cela, en offrant un langage intégré Turing complet appelé Solidity. Avec ce langage il est possible à quiconque de coder et de le publier dans la *Blockchain* Ethereum, créant ainsi des *Smart Contracts* et des *decentralized autonomous organization* (DAO), dont les caractéristiques seront détaillées par la suite.

Ethereum introduit plusieurs innovations majeures, dont celle qui vise à créer un bloc idéalement, non pas toutes les 10 minutes comme Bitcoin, mais, en moyenne, toutes les 15 secondes. (Dannen, 2017)

Ethereum introduit également la notion d'état en partant du principe qu'entre chaque transaction l'état de la *Blockchain* change. Chaque bloc, en plus de la liste des transactions, contient aussi l'état le plus récent. Cet état est enregistré sous forme d'un arbre appelé le Patricia *Tree* qui permet d'effectuer efficacement des changements similaires à ceux nécessaires lors de transactions. Un *full node* n'aura donc besoin de stocker que l'état le plus récent et non pas la totalité de la *Blockchain*, ce qui d'après Vitalik Buterin ferait économiser 5 à 20 fois plus d'espace de stockage.

« One common concern about Ethereum is the issue of scalability. Like Bitcoin, Ethereum suffers from the flaw that every transaction needs to be processed by every node in the network. With Bitcoin, the size of the current Blockchain rests at about 15 GB, growing by about 1 MB per hour. If the Bitcoin network were to process Visa's 2000 transactions per second, it would grow by 1 MB per three seconds (1 GB per hour, 8 TB per year). Ethereum is likely to suffer a similar growth pattern, worsened by the fact that there will be many applications on top of the Ethereum Blockchain instead of just a currency as is the case with Bitcoin, but ameliorated by the fact that Ethereum full nodes need to store just the state instead of the entire Blockchain history.

The problem with such a large Blockchain size is centralization risk. If the Blockchain size increases to, say, 100 TB, then the likely scenario would be that only a very small number of large businesses would run full nodes, with all regular users using light SPV nodes. In such a situation, there arises the potential concern that the full nodes could band together and all agree to cheat in some profitable fashion » (Buterin, 2013b)

2.2.2 Fonctionnement général

Dans Ethereum, il y a deux types de comptes : les comptes externes d'utilisateurs gérés grâce à leurs clés privées, et les comptes des contrats (les *Smart Contracts* qui seront traités dans le prochain chapitre). Un compte externe peut créer et signer une transaction, mais ne possède pas de code, alors que le *Smart Contract*, lui, en possède un.

Dans le white paper d'Ethereum, Vitalik conclut :

« The Ethereum protocol was originally conceived as an upgraded version of a cryptocurrency, providing advanced features such as on-blockchain escrow, withdrawal limits, financial contracts, gambling markets and the like via a highly generalized programming language. The Ethereum protocol would not "support" any of the applications directly, but the existence of a Turing-complete programming language means that arbitrary contracts can theoretically be created for any transaction type or application. » (Buterin, 2013c)

Ethereum permet donc de faire bien plus qu'une simple crypto-monnaie.

« Protocols around decentralized file storage, decentralized computation and decentralized prediction markets, among dozens of other such concepts, have the potential to substantially increase the efficiency of the computational industry, and provide a massive boost to other peer-to-peer protocols by adding for the first time an economic layer. » (Buterin, 2013c)

Des applications peuvent être décentralisées et profiter pour la première fois d'une base monétaire *peer to peer*. D'autres applications peuvent aussi être décentralisées sans avoir pour autant un but monétaire.

« The concept of an arbitrary state transition function as implemented by the Ethereum protocol provides for a platform with unique potential; rather than being a closed-ended, single-purpose protocol intended for a specific array of applications in data storage, gambling or finance, "Ethereum is open-ended by design, and we believe that it is extremely well-suited to serving as a foundational layer for a very large number of both financial and non-financial protocols in the years to come. » (Buterin, 2013c)

2.2.2.1 Casper – mécanisme de consensus

Casper est un mécanisme de consensus planifié pour Ethereum.

Ethereum, qui fonctionne à la base en PoW, a voulu passer en PoS, bien que conscient du risque de perte de pouvoir des anciens mineurs. Un bon compromis serait un mélange de PoW et de PoS. La communauté d'Ethereum et son créateur Vitalik Buterin travaillent sur un système appelé Casper. (Hertig, 2017)

Hybride entre PoW et PoS, Casper va être un premier pas vers le PoS en introduisant le concept de mineur "virtuel" (il est virtuel car dispensé d'avoir une puissance de calcul) pouvant être quiconque possédant de l'Ether. Il sera possible de déposer de l'Ether dans le système Casper, ce montant sera gelé pour pouvoir miner virtuellement (valider un ou plusieurs blocs), être rémunéré pour ceci tout en acceptant de les perdre si on ne suit pas les règles.

Cette manière de fonctionner offre l'avantage de limiter le risque que certaines personnes, en possédant une grande partie des parts, ne puissent fausser les transactions dans la *Blockchain*. La meilleure garantie de sécurité vient surtout du fait que l'investissement est proportionnel à l'efficacité du minage virtuel. En effet, celui qui a investi beaucoup d'argent a un grand pouvoir de vote, mais aussi tout intérêt à le faire fructifier en étant rapide et efficace, alors qu'une autre personne, même si elle possède beaucoup de comptes, ayant investi presque rien, n'a que très peu de pouvoir de vote.

Les validateurs pourront valider un bloc tous les 100 blocs, celui-ci sera donc appelé un point de contrôle. Le validateur sera récompensé s'il a validé le nouveau point de contrôle sans mentir sur celui-ci ni sur le précédent (ces deux points de contrôle doivent avoir été validés par plus de 2/3 de la valeur en jeu).

Le but, pour chaque validateur, est d'arriver à consensus afin d'ajouter et de valider des blocs de la plus longue chaîne. Pour ceux qui n'auront pas misé sur la bonne chaîne (représentant moins de 1/3 du montant déposé), leurs mises seront retirées pour être redistribuées aux 2/3 des validateurs restants. Cela permet de ne pas créer de nouveaux Ethers donc de ne pas créer d'inflation. (Floersch, 2017)

Il serait toujours possible de brouiller les votes en les contrebalançant pour qu'ils puissent arriver à égalité et faire échouer le consensus. Par exemple: deux blocs sont en cours de validation pour être le nouveau point de contrôle, les mineurs ayant validé à 40% le bloc A et 60% le bloc B, un acteur malveillant pourrait vouloir faire échouer le

consensus en votant sur le bloc A, mais aussi sur le bloc B afin de s'assurer que A et B arrivent à 50%. Cependant, il est rassurant de savoir que, d'un certain point de vue, le minage virtuel PoS utilisé dans Casper n'est pas conceptuellement si différent de PoW. En effet, ce dernier possède aussi une punition en cas de participation à la mauvaise chaîne qui est: l'achat de nouveaux matériels, frais de location ou d'électricité nécessaire pour pouvoir miner sur les deux blocs simultanément. Le participant qui tente de valider un bloc sur deux chaînes simultanément (aussi appelé Slasher en hommage à la stratégie Slasher développée par Iddo Bentov) ou bien vote sur la mauvaise chaîne (appelée dunkle dans Casper) sera déterminé comme tricheur. Il verra ses parts brûlées automatiquement (ce qui n'est pas le cas de la première implantation dans PeerCoin mais qui le sera dans Casper la future version de Ethereum). (Bentov, Gabizon, et Mizrahi, 2017)

Une question est souvent posée : est-ce que de verrouiller des Ethers pendant une certaine période, les empêchant d'être utilisés et de fructifier, ne reviendrait-il pas au même que, avec PoW, d'investir dans du *hardware* (matériel informatique) et de l'utiliser au fil du temps?

La réponse donnée dans le document officiel à propos de PoS et de Casper est : non cela ne revient pas du tout au même et ceci pour plusieurs raisons. Les fonds verrouillés seront libérés après avoir permis de passer la période de validation (environ 4 mois), alors que, selon la loi de Moore, la puissance des machines double tous les 2,772 ans, ce qui rend rapidement tout *hardware* obsolète et donc à renouveler (ce qui occasionne des frais). Il faut bien comprendre que, dans PoW, il serait donc intéressant de ne plus être dépendant des lois physiques mais de pouvoir, disons, les optimiser virtuellement. Par exemple, si la difficulté de PoW venait à doubler à cause d'une nouvelle arrivée de supers mineurs, le même phénomène contrôlé virtuellement par PoS pourrait faire en sorte que la difficulté triple.

Le but recherché étant d'économiser de l'énergie; de dissoudre les cartels de mineurs actuels pour éviter une attaque de type 51%; de rendre l'accès au minage plus équitable et surtout moins centralisé.

2.3 Qu'est ce qu'un **Smart Contract**?

La suite de ce document parlera exclusivement de la *Blockchain* Ethereum pour se concentrer sur ce qu'est un *Smart Contract* ou une DAO et quels sont les fondamentaux pour en créer soi-même.

2.3.1 Histoire des **Smart Contracts**

En général, le droit des contrats est un secteur en constante évolution et très dynamique, s'adaptant constamment à son environnement. À chaque culture et chaque époque correspond une forme de contrat distincte. Dans une économie principalement agricole les contrats seront négociés différemment que dans une économie industrielle ou de services. Une société dominée par l'information penchera en faveur d'une minimisation de l'implication humaine, autant pour la définition des termes contractuels que lors de son exécution.

Les contrats dits "intelligents" vont dans cette direction et correspondent à cette évolution constante du droit contractuel. Selon Nick Szabo, pionnier et précurseur dans ce domaine,

« a smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives of smart contract design are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitration and enforcement costs, and other transaction costs. » (1994)

Cependant, cette définition ne permet pas de se démarquer d'autres technologies, par exemple les distributeurs automatiques, dont les premiers exemples remontent à l'époque romaine lorsque Hero Ctesibius d'Alexandrie expliquait, en 62 av. J.Ch., qu'en échange d'une pièce de 5 Drachma déposée dans sa machine, un peu d'eau sainte serait amenée automatiquement dans les temples égyptiens. (Savelyev, 2016). Outre les distributeurs automatiques, à la bourse, la plupart des opérations financières ne sont plus réalisées par des *traders*. En 2014, sur les bourses américaines de New York Stock Exchange et NASDAQ, plus de 75% des opérations étaient réalisées par un système de *trading* automatique (*automated trading system - ATS*). (Wikipedia, s.d.). Alors, qu'y a-t-il de vraiment nouveau dans un *Smart Contract*?

Le *Smart Contract* peut être considéré comme un changement de paradigme parce qu'il permet d'automatiser le processus contractuel des deux côtés. Et lorsque les

performances de toutes les parties sont totalement automatisées, de nouvelles qualités contractuelles apparaissent. (Savelyev, 2016)

Dans un article datant de 1997, Nick Szabo donnait l'exemple d'un contrat de leasing pour un véhicule. « *Si le propriétaire cesse d'effectuer les versements, le Smart Contract peut invoquer un protocole qui rend automatiquement le contrôle de la clé du véhicule à la banque* », écrivait-il. Les *Smart Contracts* combinent ainsi :

« protocols, users interfaces, and promises expressed via those interfaces, to formalize and secure relationships over public networks. This gives us new ways to formalize the digital relationships which are far more functional than their inanimate paper-based ancestors. Smart contracts reduce mental and computational transaction costs, imposed by either principals, third parties, or their tools. » (Szabo, 1997)

Certains annoncent d'ores et déjà l'obsolescence des services juridiques des entreprises , alors que d'autres se posent la question de la régulation. "Ces contrats entreront-ils dans le cadre de la législation existante, ou faudra-t-il légiférer pour prendre en compte leur rôle dans la société ? L'enjeu d'innovation va porter sur la Cyber Law : comment lier les contrats « crypto » et les contrats « fiat », terme qui regroupe tout ce qui a trait à l'environnement juridique traditionnel." (Deloitte, s.d.)

2.3.2 Smart Contract dans Ethereum

Il est donc nécessaire de s'éloigner de la description mécanique du *Smart Contract*, comme le distributeur automatique qui va, sur demande et par des mécanismes physiques, faire respecter un contrat (p.e. délivrer une barre de chocolat et rendre la monnaie). Dans ce sens, Greenspan fournit une définition plus précise: "A *Smart Contract is a piece of code which is stored on an Blockchain, triggered by Blockchain transactions, and which reads and writes data in that Blockchain's database*". (2016)

Le *Smart Contract* est ainsi inscrit dans la *Blockchain*. Dans le cas d'Ethereum, celui-ci est comparable à un programme qui va permettre à la communauté de l'utiliser ou d'écrire d'autre programmes qui héritent de celui-ci, chacun définissant sa propre logique pour répondre à un plus grand éventail d'utilisations possibles. Dans le *whitepaper* d'Ethereum, Vitalik écrit :

«Smart contracts, cryptographic "boxes" that contain value and only unlock it if certain conditions are met, can also be built on top of the platform, with vastly more power than that offered by Bitcoin scripting because of the added powers of Turing-completeness, value-awareness, blockchain-awareness and state.» (Buterin, 2013d)

Afin de mieux comprendre l'innovation apportée par Ethereum dans l'application des *Smart Contracts*, il convient de détailler chacune des caractéristiques principales.

2.3.2.1 Programmable

Un *Smart Contract* peut être écrit en Solidity, puis publié sur la *Blockchain* Ethereum.

Solidity est un langage dit *Turing* complet parce qu'il va permettre de programmer un plus grand éventail de fonctionnalités que le permet le langage Scripting sur Bitcoin, comme par exemple des boucles, prenant de fait moins de place de stockage lorsque les fonctionnalités sont programmées sous forme de *Smart Contracts*.

Un *Smart Contract* exécute du code à chaque fois qu'il reçoit une transaction, afin de lire et d'écrire des données en interne, d'envoyer un message et plus encore.

Les fonctions codées dans un *Smart Contract* sont appelables (si celles-ci sont publiques ce qui est le cas par défaut). Chaque appel prend la forme d'une transaction qui, une fois ajoutée à un bloc, est téléchargée puis exécutée par tous les nœuds du réseau, car cela fait partie intégrante de la transition d'état et de la validation du bloc. (Vitalik, 2013b)

Il est également possible de stocker des valeurs, introduire des variables et donc de faire évoluer le *Smart Contract*, ayant pour résultat un grand nombre d'états possibles, dont les comptes sont tenus au niveau du *Smart Contract*.

Pouvoir gérer nativement le temps grâce à des mots clés simples en Solidity, qui de manière transparente va se référer à la durée moyenne d'ajouts de blocs d'Ethereum et aux *blockhashs* pour gérer le temps. Il est donc possible de coder des conditions temporelles. Il est possible d'utiliser des sources de données externes pour obtenir des informations nécessaires provenant de bases de données, d'utilisateurs ou d'autres *Smart Contracts*. C'est ce qu'on appelle avoir recours à un oracle. Ceci est très utile pour connaître, par exemple, le prix du cours d'une devise ou des résultats sportifs, ce qui peut être requis dans de nombreux cas d'utilisation.

Finalement, il est possible de procéder à des tests avant la publication officielle du *Smart Contract*. La *Blockchain* Ethereum officielle est appelée *mainNet*, car il existe d'autres réseaux alternatifs (appelés *testNet*) où il est possible de faire des tests sans dépenser de vrais Ethers.

2.3.2.2 Le code c'est la loi

Grâce à la *Blockchain* et aux *Smart Contracts*, on se rend compte qu'il est possible de coder des règles qui, à elles seules, feront autorité une fois publiées dans la *Blockchain*. Les *Smart Contracts* ne sont pas là seulement pour exécuter des lois prédéfinies de manière automatique, mais permettent de définir du code qui a comme valeur intrinsèque d'être l'unique loi en vigueur.

Selon Nick Szabo (1997), les *Smart Contracts*, en plus d'être rapides et efficaces, permettent de supprimer l'ambiguïté des contrats traditionnellement textuels en remplaçant le texte par du code formel, interprétable par une machine. De plus, les *Smart Contracts* sont transnationaux et donc accessibles à tout le monde. Tout en étant une force, il est difficilement concevable de les faire reconnaître par les lois de chaque pays, posant alors la question de la légitimité et de la responsabilité des gouvernements. Les *decentralized autonomous organisations* (DAO) paraissent inarrêtables, ce qui pose problème aux États qui n'ont pas la possibilité d'imposer leur autorité.

Le "*Code is Law*" est un concept qui reste très problématique et pourrait avoir des conséquences non désirées. Des injustices pourraient être commises, d'un point de vue légal, le code n'étant pas prévu pour les éviter. Bien que le besoin d'accélérer, d'automatiser et de clarifier corresponde aux besoins d'autogérer des réseaux autonomes pair à pair, il n'en est pas moins que le fait d'automatiser les lois pourrait conduire à la réduction de nos droits individuels.

L'idée de permettre à quiconque de déployer un programme décentralisé utilisable et imposant ses propres lois est un concept démocratique très en vogue et qui, certainement, implique de nombreux avantages. Elle pourrait bien aussi déboucher sur un système inarrêtable et totalitariste, un rêve de cyber-libertaires qui pourrait tout aussi bien tourner au cauchemar décentralisé, laissant place à la surveillance et aux manipulations massives. (Filippi & Hassan, 2016)

Selon Savelyev, le danger est le suivant :

« The person expresses its consent with the terms of the contract and mode of their execution at the moment of conclusion of the contract. Taking into account that such person won't be able to influence the execution of the agreement, once it is entered to, there should be a certain trust in place, which gives rise to a kind of "fiduciary" relations in Smart contract. But in contrast to classic contract where trust is put in the personality of the other party to the contract, in Smart contracts such trust is put in the computer algorithm standing behind the agreement ("trustless trust"). » (2016)

2.3.2.3 Idéalement indépendant

Idéalement, un *Smart Contract* ne devrait pas être dépendant de systèmes externes afin de fournir un service déterminé, ce qui est malheureusement très difficile lors de son application concrète. Bien souvent, il est nécessaire de faire recours à un oracle.

Un oracle est un service informatique, externe au *Smart Contract*, qui est utilisé pour acquérir de nouvelles données.

Toutefois il convient d'être vigilant, car faire appel à un oracle implique de courir le risque de devoir gérer les cas où les données ne seraient plus atteignables, ou bien seraient faussées. En effet, les propriétés propres à la *Blockchain* sont que celle-ci est irrévocable et infalsifiable. Ceci se trouve être en contradiction avec le recours aux oracles, surtout si ceux-ci sont centralisés, rendant donc le contrat vulnérable.

C'est pourquoi un bon système d'oracle pourrait être fait par consensus dans un *Smart Contract*, afin que les résultats soient garantis par le plus grand nombre.

Un *Smart Contract*, s'il avait besoin d'un oracle, pourrait, pour plus de fiabilité, en utiliser plusieurs différents et ainsi, s'ils fournissent, en majorité, la même information, cette dernière pourrait être considérée comme valide.

Ce genre de procédé rendrait l'information fiable sur la base des participants qui l'ont validée, mais le processus est long et coûteux financièrement. En effet, il faudrait tout de même rémunérer les participants, il est aussi à prévoir que ce processus prend un certain temps (délai entre le moment où l'information a été divulguée et le moment où elle est validée par votation).

2.3.3 Bénéfices actuels

2.3.3.1 Héritage des avantages de la *Blockchain*

Un *Smart Contract* hérite des avantages et forces de la *Blockchain* qui est donc publique et auditable, étant donné que celui-ci est programmé et stocké sur la *Blockchain*.

2.3.3.1.1 Suppression d'intermédiaires

Tout comme la *Blockchain*, le *Smart Contract* permet d'automatiser un processus, tout en supprimant les intermédiaires. Un grand nombre de domaines pourraient en bénéficier comme le domaine légal, dans la musique, les réseaux sociaux, la publicité et bien d'autres. Cette technologie permettant de réduire les coûts tout en repensant la manière de participer et d'être récompensé pour cela.

2.3.3.1.2 Inarrêtable

L'intégralité du code est immuable, dans le sens que son code est publié et donc inscrit dans une transaction de la *Blockchain* Ethereum. Ce qui rend l'application inattaquable et inarrêtable.

2.3.3.1.3 Borderless

L'avantage d'être inarrêtable et sans autorité de contrôle est que le contrat n'est plus dépendant des frontières, de sa localisation physique et de la juridiction de son pays.

2.3.3.1.4 Open source

Accessible et donc vérifiable (au minimum sous forme compilée ce qui suffit pour être renversable par *retro engineering* et obtenir le code source). Il est donc préférable que le code source soit également fourni afin de permettre à la communauté de comprendre la logique du contrat sans avoir à passer par une étape de *retro engineering*.

De plus, une fois rendu public, il est possible de profiter pleinement d'un système de primes en ligne permettant à la communauté de trouver des bugs lors des tests. Malgré cela, certains préfèrent dissimuler leur savoir-faire et les failles non découvertes en rendant le code source difficile d'accès.

2.3.3.2 Coder des conditions de paiement divers

À travers un *Smart Contract* il est possible de définir ses propres règles, entre autres d'écrire les conditions pour déverrouiller des *assets* cryptographiques stockés dans la *Blockchain*. Un exemple d'application est la création de mécanismes de *wallets* multi-signatures qui, au lieu de nécessiter la signature de tous ses membres, peut être redéfini selon la volonté du programmeur. Ainsi, il est possible de demander la signature de deux personnes sur trois, 60% des membres ou ce qui semble pertinent. (Szabo, 2016)

À cela s'ajoute la possibilité d'y intégrer des conditions temporelles, grâce à l'accès aux dates des blocs visibles de manière transparente en Solidity. Par exemple, il pourrait être possible de retirer 20% des fonds tous les mois, ou de recevoir de l'Ether pendant 10 jours puis d'arrêter le *crowdfunding* après cette période.

2.3.3.3 Créer des Tokens

Un *Token* sur une *Blockchain* peut avoir de multiples applications, allant d'une crypto-monnaie valant un certain prix, jusqu'au simple jeton n'ayant aucun lien avec une valeur réelle, utilisé pour inciter à la participation. De plus,

« Token systems are surprisingly easy to implement in Ethereum. The key point to understand is that a currency, or token system, is just a database with one operation: subtract X units from A and give X units to B, with the proviso that (i) A had at least X units before the transaction and (ii) the transaction is approved by A. All that it takes to implement a token system is to implement this logic into a contract. » (Buterin, 2013e)

2.3.3.4 KickStarter

En règle générale, un *Smart Contract* "KickStarter" a pour but de demander des donations afin d'aider une organisation à concrétiser un projet, il permet de recevoir des fonds de plusieurs donateurs et de les stocker dans un conteneur dans la *Blockchain*. Il permet donc de connaître le montant total reçu lors de ce levé de fond initial, et de le distribuer aux bénéficiaires. De la même manière que le système de *crowdfunding*, si le contrat expire sans avoir atteint le seuil minimal, les fonds seront retournés aux donateurs.

2.3.3.5 Lancer sa propre ICO

Un *Smart Contract* permet d'initier sa propre vente de *Tokens* initiale, ou ICO, suivant le modèle dont il a été programmé. La communauté Ethereum peut donc désormais envoyer de l'Ether au *Smart Contract* de l'ICO, et ainsi se procurer des parts de ces *start-ups* en acquérant les tous premiers *Tokens* mis en circulation.

Lors de la programmation, toutes les règles de la vente sont fixées, telles que le nombre de *Tokens* à vendre; le prix à l'unité; la durée de la vente; d'éventuelles réductions et bien plus. Ces règles rendues publiques par l'organisme fondateur des *Tokens* sont généralement explicitées dans le *whitepaper*, en plus de figurer sous forme de code compilé dans la *Blockchain* Ethereum.

À travers ce processus, les participants seront récompensés au moyen d'acquisitions d'actions, ou de parts du projet "kickstarté" sous forme de *Tokens*. Fin juillet 2017, le Financial Times écrivait que plus de \$1.3 milliard de dollars ont été investis dans ces *start-ups* à travers les ICO rien qu'en 2017. (Arnold, 2017)

Comme il a été expliqué auparavant, il est préférable que le code source soit disponible afin de permettre à la communauté de comprendre la logique du contrat. Quand bien même celui-ci serait difficile d'accès, il peut être retrouvé par *reverse engineering*, mais gare aux pirates, il est primordial que le code soit sans faille pour que l'ICO se déroule comme prévu.

2.3.3.6 Créer sa propre organisation décentralisée et autonome (DAO)

Selon Savelyev, une DAO,

« is nothing more than a set of long lasting "Smart" contracts as opposed to a regular "Smart contract" having specific purposes and coming to an end once they are achieved. The organizational theorist Arthur Stinchcombe once wrote that contracts are merely organizations in miniature, and by extension all organizations are just complexes of contracts. Firms are created using a series of contractual agreements, ranging from employment contracts and employee benefits, to deals with vendors and suppliers and obligations to its customers, to building leases and sales & purchases of equipment. » (2016)

L'organisation en question sera régie par ses participants selon les règles et mécanismes mis en place dans le *Smart Contract*. Elle bénéficiera par la suite de nombreux avantages de la *Blockchain* et des *Smart Contracts* telles que la transparence et la disponibilité des informations relatives à l'organisation. (Blockchain France, 2016)

Une DAO est composée d'un ou plusieurs contrats déployés sur la *Blockchain* Ethereum permettant de se procurer des fonds à travers des ICO. Les investisseurs ou les membres du projet de la DAO, peuvent ensuite par exemple gérer ces fonds communs (représentés par des *Tokens*). La DAO peut distribuer des dividendes ou des salaires pour les meilleurs développeurs.

Contrairement aux *Smart Contracts* destinés à exécuter une tâche fixe pendant un certain temps, puis à s'arrêter, une DAO est définie telle une organisation de nature inarrêtable. Elle met à disposition des mécanismes d'évaluation du niveau de confiance des participants ainsi que de leurs pouvoirs dans l'organisation, pouvoirs d'action et de décisions, les participants sont rémunérés en Ether ou autres crypto-monnaies ou alors en parts (*Tokens*) de la DAO en question. L'avantage est que l'organisation bénéficie de la transparence et de l'immutabilité du *Smart Contract*. La transparence des règles met tous les participants au même niveau et assure qu'une tiers personne ne peut pas obtenir d'autres privilèges que ceux indiqués dans le contrat. Les valeurs du contrat ne peuvent pas être modifiées sans interagir avec celui-ci selon ses règles.

Il est possible de coder des mécanismes de consensus qui permettent de piloter une organisation de manière décentralisée, d'avoir un droit de vote sur les futures modifications de l'organisation, le grade de chacun ou les futures actions à accomplir. Les participants seront plus motivés si on leur permet de voter au lieu de se voir imposer des choix faits par un seul acteur privilégié.

Pouvoir piloter les fonds investis, c'est l'idée du projet "The DAO". Celui-ci a, d'ailleurs, mal fini à cause d'une faille de sécurité créant un boom médiatique et idéologique sur le moyen de rendre justice, ce qui a abouti à un *hard fork* créant les chaînes Ethereum (principale) et Ethereum Classic. (CryptoCompare, 2017, 5 juillet). "The DAO" est un projet qui s'est fait piraté, à ne pas confondre avec le terme DAO dans sa généralité.

2.3.3.7 Multiples utilisations alternatives

Les possibilités sont nombreuses, par exemple utiliser un *Smart Contract* pour créer un système décentralisé d'échange de crypto-monnaie avec lequel peuvent interagir d'autres *Smart Contracts* ou de simples utilisateurs à l'aide d'une application. (CryptoCompare, 2017, 13 avril). Ceci permettrait de garantir que l'argent ne sera pas détourné par un échange centralisé qui possède souvent l'accès à votre clé privée. L'avantage est aussi que la *Blockchain* est un registre distribué, ce qui rend un *Smart Contract* correctement programmé invulnérable face aux attaques de type DDoS, alors que les systèmes totalement ou quasiment centralisés eux le sont.

Créer des plateformes de paris sur des événements sportifs ou autres (tels que les projets Augur et Gnosis).

Il existe de nombreux cas d'utilisation prometteurs tels que: dans les réseaux sociaux, la publicité, la gestion de fortune, la location de bandes passantes ou d'espaces de stockage inutilisés et bien d'autres.

2.3.3.8 Plus performant que les contrats traditionnels

S'il y a un domaine qui est directement touché par l'émergence des *Smart Contracts* c'est bien le domaine légal, car les contrats traditionnels sont coûteux. Désormais, grâce au *Smart Contract* il est possible de définir les règles sous forme de code, le tout protégé de manière cryptographique. Ces lois sont ainsi exécutables rapidement, et à moindre frais.

«Courts, lawyers, judges and investigators all form this system of contract enforcement. With a blockchain-based «Smart» contract, however, much of these costs are greatly reduced or eliminated. This promises to make blockchain-based organizations more efficient, cost-effective, and competitive compared to traditional firms in the marketplace.» (Savelyev, 2016)

Lors d'un séminaire sur le futur de l'économie globale en 2016, Nick Szabo proposait l'illustration suivante: imaginez des hôtels qui, en cas de mauvais temps, perdent la moitié de leur chiffre d'affaire, ils paieraient donc un *Smart Contract* qui les dédommagerait automatiquement grâce à des informations provenant d'une *Blockchain* de prévision météorologique. Ce *Smart Contrat* pourrait donc à lui seul faire le travail d'une assurance de manière plus économique.

2.3.4 Quels sont les problèmes actuels?

2.3.4.1 Failles de sécurité

"The DAO" est un projet qui a rassemblé un grand nombre d'investisseurs et qui a atteint jusqu'à 150 millions de dollars lors de leur ICO. Une des idées phares était de permettre à la communauté de voter sur des projets d'investissements et d'y participer de manière autogérée sans autorité centrale.

Malheureusement "The DAO" fut piraté le 17 juin 2016, l'attaquant a exploité une faille dans le *Smart Contract* et, ainsi, détourné 3,6 millions d'Ethers ce qui, à l'époque, valait 50 millions de dollars.

Une faille a permis à l'attaquant de retirer de l'Ether du *Smart Contract* de "The DAO", non pas vers lui même, mais vers son propre *Smart Contract*, puis de rappeler le contrat de "The DAO" pour recommencer l'opération, ceci a court-circuité la suite du premier appel et, ainsi, lui permettait de retirer de multiples fois sans laisser le temps au contrat de le lui déduire de son compte. Cette erreur est devenue l'exemple classique de faille de sécurité dans ce domaine.

Cette attaque a créé beaucoup de réactions de la part de la communauté Ethereum qui était partagée entre les deux idées suivantes :

1) Ignorer les futures transactions du voleur pour qu'il ne puisse jamais retirer (contraire aux fondement de la *Blockchain*), ce qui aurait débouché sur un *soft fork* (ne nécessitant pas la création d'une nouvelle branche).

2) Publier une nouvelle version de la *Blockchain* pour permettre au voleur de récupérer son argent, aussi appelé *hard fork*, car l'erreur était dans le *Smart Contract* et a été faite par les développeurs de celui-ci. Techniquement, cela se résume en créant un nouveau *Smart Contract* où l'attaquant pourra seulement retirer ses parts de "The DAO" contre de l'Ether. Cette solution a été jugée correcte d'après l'idéologie de la justesse du code. Vitalik Buterin a aussi soutenu cette idée du fait que le code du *Smart Contract* était tout simplement faux et que l'attaquant, lui, a au moins le mérite de l'avoir vu.

Ce *hard fork* est mal vu par une partie de la communauté qui, elle, n'a pas accepté de reverser l'argent au voleur, ce qui à créé deux états différents des comptes dans Ethereum. Ceux qui n'ont pas accepté de rendre l'argent ont donc continué à faire fonctionner la *Blockchain* Ethereum sans ce changement pour débloquer l'argent de

"The DAO", se sont donc retrouvés avec une deuxième *Blockchain* différente de l'Ethereum qui s'appelle l'Ethereum Classic.

« *Ethereum Classic is not a new cryptocurrency, but instead a split from an existing cryptocurrency, Ethereum. Both Blockchains are identical in every way up until block 1920000 where the hard-fork to refund The DAO token holders was implemented, meaning that all the balances, wallets, and transactions that happened on Ethereum until the hard-fork are still valid on the Ethereum Classic Blockchain. After the hard-fork, the Blockchains were split in two and act individually.* » (CryptoCompare, 5 Juillet 2017)

On parle souvent de la faille de "The DAO", mais il existe en réalité énormément de failles possibles et de bugs à éviter, il est donc primordial de se renseigner régulièrement sur ce sujet.

Il existe un document très pratique qui en cite quelques unes :

<https://github.com/ConsenSys/smart-contract-best-practices> (ConsenSys, 2016)

La sécurité du contrat est primordiale pour garantir qu'aucune partie prenante ne soit biaisée à terme ou en cours de participation. C'est pourquoi il est nécessaire de considérer le développement d'un *Smart Contract* différemment d'un processus standard, car il requiert un code d'une qualité supérieure. Chaque ligne de code, ainsi que son emplacement, est extrêmement important, et doit être régulièrement testé afin de faire ses preuves. Une seule erreur est une porte ouverte à une multitude d'attaques diverses et variées.

2.3.4.2 Difficile de ne pas recourir à des services externes

Les cas autorisés et rejetés doivent être totalement cernés et maîtrisés. Les *Smart Contracts* ne doivent (si possible) pas être dépendants d'un système externe tel qu'un oracle. Le but étant de garantir la sécurité du contrat en question.

De ce fait, les oracles restent des points sensibles pouvant mettre en péril le *Smart Contract*, il est cependant difficile de s'en passer, car il faudrait pour cela que vous possédiez toutes les informations nécessaires dans le *Smart Contract* sans faire appel vers l'extérieur.

Un des moyen utilisés, dans la mesure du possible, est d'inclure le rapatriement de ces informations lors des appels de fonction du *Smart Contract* et de les valider, par exemple, en utilisant un mécanisme de consensus. Néanmoins, cela rajoute de la complexité et de la lenteur dans le *Smart Contract*, c'est pourquoi certains préféreront s'appuyer sur des systèmes tiers.

Ce problème pourrait aussi être réglé à terme, étant donné que de nombreux *Smart Contracts* feront office d'oracle, par exemple pour valider l'identité (projet Civic) d'une personne ou pour fournir des résultats sportifs (projet Gnosis) ou autres. L'information serait validée par consensus et donc indépendante de tout système centralisé. Ceci pourrait être un bon compromis malgré le fait qu'il restera toujours une certaine dépendance à un tiers, même si celui-ci est un autre un *Smart Contract*.

2.3.4.3 Participations aux ICO difficilement régulables

Tout le monde devrait avoir sa place, mais certaines personnes détiennent de multiples *wallets* et vont tenter de capitaliser une grande partie des parts (*Tokens*).

À l'heure actuelle peu d'ICO passent par une étape de validation de l'identité du participant afin de lui fixer une limite d'achat, de nouveaux projets récents comme Civic vise à répondre à ce problème. (Civic, 2017)

2.3.4.4 Pas de garantie légale

Lors de la participation à des ICO, ou même en utilisant des *Smart Contracts*, rien ne garanti la sécurité de l'investisseur, sauf les règles du contrat et les acteurs qui interagissent avec celui-ci.

Les personnes qui possèdent des droits supérieurs dans le contrat, comme par exemple son créateur, pourraient récupérer l'argent investi ou posséder de nombreux avantages, et c'est bien souvent le cas, car les ICO sont créées pour lancer des projets et les financer (*kickstarters*), il est donc bien normal que les créateurs retirent une partie des fonds investis.

Aucune garantie sur la performance, ou sur les promesses qu'ont faites les créateurs de cette prévente. Comme lors de l'arrivée de milliers de crypto-monnaies différentes qui sont majoritairement non utilisées et principalement créées à titre purement spéculatif, les *Tokens* mis en préventes par *Smart Contracts* sont eux aussi victimes de ce phénomène, de ces faux projets utilisés uniquement pour récolter des fonds puis trop vite abandonnés à la communauté.

Le code fait autorité et non la loi, même si l'intention n'était pas de permettre ou d'interdire une certaine action, au final c'est le code du *Smart Contract* qui sera l'arbitre de ce qui est ou non faisable.

D'autres idées très convoitées comme le fait qu'un *Smart Contract* pourrait très bien faire autorité sur la possession immobilière, faire foi de notaire électronique pour un coût ridicule. Ceci a fait débat sachant que, actuellement, les prestations d'un notaire sont loin d'être données.

Malheureusement, le gouvernement et les lois sont loin d'évoluer aussi vite que la technologie *Blockchain*.

« Le transfert de propriété d'un bien matériel (immeuble par exemple) passe par le registre foncier. Le contrat de transfert doit revêtir la forme authentique (notaire) et être vérifié par le conservateur du registre foncier. Une transaction immobilière qui ne passe que par la Blockchain n'a pas de valeur. »

« En cas de litige, " bonne chance si vous êtes devant un juge de district ou de commune en lui disant que le contrat a été conclu sur la Blockchain. La compétence des tribunaux est en retard " (Farine, 2017), avance l'avocat. A l'évidence, il ne faut pas s'attendre à de grands changements avant plusieurs années. » (Garessus, 2017)

2.3.4.5 Compréhension du code et détection des vulnérabilités

Il est prévu, dans le code civil (art. 1163), que les parties prenantes d'un contrat doivent être capables de comprendre les engagements qu'ils prennent. Cela pose problème dans les *Smart Contracts*, car il est très difficile de détecter une éventuelle faille de sécurité pouvant aboutir à ce que des personnes soient lésées. (Croiseaux, 2016)

2.3.4.6 Rompre un engagement

Un autre problème est que les *Smart Contracts*, sont difficiles à rompre. Une fois l'argent envoyé à celui-ci, il n'est plus possible de rompre l'engagement pour récupérer son argent, à moins que cela soit prévu dans le *Smart Contract* lui-même.

Contrairement à un contrat classique, une personne lésée ne pourra pas faire appel à un tribunal pour obtenir réparation. (Croiseaux, 2016)

2.3.4.7 Trop rigide pour correspondre à la loi actuelle

Le code ne change pas, alors comment les organisations, peuvent-elles coder, par exemple, ce qu'est un accident de voiture alors qu'ils ne sont toujours pas d'accord sur sa définition. Vouloir purement coder une version figée, qui ferait autorité automatiquement, serait, certainement, bien éloignée d'une décision longuement réfléchie, prise par des êtres humains après avoir pris le temps d'analyser tous les faits. En effet, une prise de décision purement automatisée est certes rapide, mais risque de perdre toute la souplesse, la finesse d'analyse, l'aspect éthique et moral dont seul l'être humain est capable.

2.3.4.8 Mécompréhension des *use cases* réellement réalisables

Il existe certains problèmes que les *Smart Contracts* ne peuvent pas résoudre de par leurs fonctionnements. Premièrement, contacter un service externe est une forme de perte de contrôle qu'un *Smart Contract* ne devrait pas permettre. Ensuite, un *Smart Contract* ne peut initier lui-même une transaction à un instant précis, c'est toujours un acteur extérieur qui doit déclencher une action du *Smart Contract*. Finalement, penser qu'un *Smart Contract* peut rendre privées (et donc confidentielles) certaines informations en les cachant de l'extérieur (par encapsulation) est une erreur. Ceci n'est en effet pas suffisant, car les informations sont stockées dans chaque *full node* de la *Blockchain* et sont accessibles localement par ces derniers. (Greenspan, 2016)

3. Comment créer une nouvelle crypto-monnaie?

3.1 Bonnes pratiques du programmeur en Solidity

Ethereum n'est arrivé que tout récemment avec ses *Smart Contracts*, ces derniers ne peuvent donc être perçus que comme hautement expérimentaux. Les bonnes pratiques doivent évoluer dès qu'un nouveau bug est découvert.

Programmer des *Smart Contracts* se rapproche plus de ce qui se fait dans le système bancaire ou d'aviation que d'autres applications classiques pour web ou mobiles. Prétendre se tenir au courant des nouvelles failles et corrections ne suffit pas, il faut adopter une nouvelle philosophie de développement. (ConsenSys, 2016)

3.1.1 Se préparer à l'échec

Il faut être bien conscient que tout *Smart Contract* va, tôt ou tard, échouer, avoir un bug ou être piraté. Il est donc primordial de se préparer à cette éventualité en mettant en place un moyen permettant de stopper l'utilisation du *Smart Contract*, comme, par exemple, un *circuit breaker*. Il faut aussi prévoir des moyens de mettre à jour un *Smart Contract*, ce qui est proposé dans la partie "lifecycle" de la librairie Open Zeppelin. (Open Zeppelin, 2016)

3.1.2 Passer en production par étapes et détecter les bugs en amont

Passer en production par étapes c'est prendre soin d'augmenter le nombre de tests et d'utilisations à chaque nouvelle étape. Il faut prévoir de tester le contrat intégralement et d'ajouter les tests nécessaires avec chaque correctif. Il faudra, également, penser à fournir des primes pour ceux qui détecteront un bug dans le contrat.

3.1.3 Garder le contrat simple

Une trop grande complexité augmente le risque d'erreurs, réduit la compréhensibilité du code et rend plus difficile sa maintenance. C'est pourquoi il faut garder un contrat avec une logique simple, pour cela il est judicieux de créer des modules. La clarté du code doit, si possible, passer avant sa performance. Eviter de réinventer la roue, utiliser des outils qui sont déjà prévus (ex: génération de nombres aléatoires). Un *Smart Contract* doit seulement représenter la partie du système qui nécessite d'être décentralisée.

3.1.4 Se tenir informé

Il est très important de se tenir informé en vérifiant les contrats dès qu'un nouveau type de bug est découvert. Il faut, également, tenir à jour, dès que possible, les bibliothèques utilisées. Finalement, il est primordial de veiller à ce que toutes les nouvelles bonnes pratiques de sécurité qui ont fait leur preuve soient mises en place.

3.1.5 Se méfier des propriétés de la *Blockchain*

Quand on veut faire appel à des contrats externes, il faut être prudent, car ils pourraient exécuter du code malicieux et changer le cours d'exécution. De plus, comme expliqué précédemment, les données seront de toute façon publiques donc accessibles à tous. Une des principales difficultés est de prendre en compte le fait que le code exécuté a un prix en gaz (Ether), ne serait-ce que pour adapter la limite de gaz des participants.

3.2 Projet Chablex

3.2.1 Description du projet

Le projet est de développer une crypto-monnaie locale "*business to business*" (B2B) permettant d'accélérer les échanges, mais aussi de faire des emprunts B2B à un taux de 0%.

Priftis Athanasios m'a proposé de réaliser une première version de ce projet, qui est, dans le cadre de ce travail de bachelor, une opportunité de, concrètement, réaliser des *Smart Contracts* et une application associée.

Ma mission est donc, tout d'abord, de créer les *Smart Contract* (côté serveur) pour la réalisation de cette crypto-monnaie. Puis, dans un second temps, de développer une application (côté client) permettant d'utiliser les fonctionnalités principales de ce projet.

Priftis Athanasios a exprimé le souhait que le prix de 1 *Token* soit figé, pour toujours, à 1 CHF. Ceci afin de ne pas être sujet à une quelconque fluctuation de prix (autre que la fluctuation du Franc Suisse). Il est vrai que ceci offrirait une grande stabilité au *Token*, car le Franc Suisse est moins volatil qu'une crypto-monnaie dont le prix est défini directement par le libre échange. Ceci permettrait de garantir que le taux du prêt soit bien de 0% en CHF et en *Tokens*. Par exemple un emprunt de 100 *Tokens* valant 100 CHF, devra être remboursé à hauteur de 100 *Tokens* valant toujours 100 CHF.

Malheureusement, verrouiller le prix de 1 *Token* à 1 CHF dépasse l'ambition de ce projet en raison du temps disponible et pourrait faire l'objet d'un autre travail de recherche.

Dans cette première version le prix du *Token* ne sera donc pas fixe, celui-ci sera défini par l'institution qui s'occupera de l'adhésion des membres et de la distribution initiale des *Tokens*. On pourrait tout de même partir du principe que la vente initiale se fasse sur la base de 1 CHF égal à 1 *Token*. Une fois cette prévente terminée, aucun autre *Token* ne sera émis. Ne pas figer le prix du *Token* pourrait aussi être avantageux pour les membres initiaux. En effet, il y a un grand avantage à pouvoir être les premiers à posséder des parts de valeur, car elles vaudront probablement plus cher par la suite. Cela favoriserait les premiers possesseurs de la monnaie et créerait un attrait à rejoindre la communauté.

3.2.1.1 Problématiques

Ce projet soulève différentes problématiques. Certaines sont répertoriées ci-dessous, avec pour chacune d'entre elles, une proposition de solution.

a) Comment rendre cette monnaie acceptée de tous, transférable rapidement à moindre frais et, surtout, prêtable et empruntable de manière transparente et sécurisée, tout cela dans le respect des règles communes?

Solution : évidemment, le *Smart Contract* (utilisant le standard ERC20)

b) L'idée principale est de permettre à ce réseau local de, surtout, rester local. C'est pourquoi il est important de le prévoir de telle sorte: qu'il ne puisse pas être manipulé de l'extérieur et, aussi, qu'il ait la possibilité de s'étendre au fil du temps. Cela peut sembler un peu paradoxal, en effet, comment va-t-on garder une vigilance de proximité si on lui permet de s'étendre?

Solutions : Distribution fairplay et exclusivement locale, système de membres autogérés

c) Comment éviter que les membres ne prêtent que aux personnes de leur choix et forment des cartels, mettant ainsi de côté certains autres membres *fairplay*?

Solution : Demandes traitées automatiquement (FIFO)

d) Quelle serait l'intérêt des participants pour oser prêter de l'argent à 0%?

Solution : Augmentation du maximum empruntable

e) Comment faire pour éviter qu'un membre emprunte de l'argent, puis sorte du système sans jamais rembourser sa dette ?

Solution : Récompense (augmentation du maximum empruntable)

f) Comment éviter qu'un membre puisse escalader le système en remboursant sa dette précédente avec de l'argent nouvellement emprunté ?

Solution : Délai minimal entre les augmentations du maximum empruntable

3.2.1.2 Détails des solutions proposées

3.2.1.2.1 Standard d'échangeabilité (ERC20)

Une option serait de rendre le *Token* compatible avec le standard ERC20 et donc de le faire adopter plus facilement sur les plateformes d'échanges. Une autre option serait de le rendre intrinsèquement échangeable selon l'offre et la demande et capable de définir par lui même son prix, ce qui n'est pas le cas dans cette première version. Cette seconde option est déjà explorée dans le projet *Intrinsically Tradable Token* (ITT). (o0ragman0o, 2016)

3.2.1.2.2 Distribution fairplay et exclusivement locale

Dans ce projet il serait préférable, pour des raisons de simplicité de mise en œuvre , de ne pas utiliser de *Smart Contract* pour effectuer la prévente des *Tokens* initiaux (ICO).

Il serait, en effet, favorable que la prévente des *Tokens* soit faite manuellement par un groupe de personnes autorisées et suivies par un huissier. Cela garantirait aux habitants locaux (dans notre cas du Chablais) de ne pas être, dès le départ, gouvernés par une institution non locale. Le travail de cette autorité de démarrage serait essentiel au bon fonctionnement futur de cette crypto-monnaie et nécessiterait un esprit *fairplay* de la part des personnes impliquées.

Si on voulait faire ce travail de démarrage à l'aide d'un *Smart Contract* de type ICO, il faudrait développer un système permettant de valider l'identité des entreprises, leurs localisations et aussi vérifier qu'elles ne soient pas fictives, tout ceci pouvant vite devenir très difficile. C'est pourquoi cette possibilité n'a pas été retenue dans cette version du projet.

Lors du démarrage officiel, le *Smart Contract* du *Token* pour le Chablais sera publié sur la *Blockchain* en utilisant un *wallet* multi-signatures, garantissant que les personnes autorisées ainsi que l'huissier doivent arriver à se mettre toutes d'accord avant de pouvoir effectuer une quelconque action. A partir de cet instant, le *Smart Contract* prendra vie et entrera dans la période initiale de 90 jours (appelée période Q1).

La période Q1 permet aux créateurs d'ajouter des membres grâce à leurs clés publiques Ethereum. De manière légitime, ils choisiront les nouveaux membres selon des critères comme : le domaine de l'entreprise, sa localisation, ses concurrents et autres. Cette procédure permettra de garantir que les participants sont des entreprises

connues, locales et actives. Cela permettra, également, d'éviter le favoritisme de sorte que chaque entreprise locale aura le droit de s'enregistrer.

Les créateurs pourront, après avoir ajouté des membres, leur vendre des *Tokens* au prix de 1 CHF par *Token*. Dans ce projet, il n'a pas été clairement défini comment l'argent reçu sera utilisé, il pourrait très bien permettre d'ouvrir une banque locale de *Tokens* par exemple.

Les membres ne pourront effectuer aucune action (tel que le transfert de *Token*, les demandes d'emprunt et les prêts, les propositions et élections de nouveaux membres) durant la période Q1. Cela afin de garantir que les membres initiaux ne puissent pas prendre de l'avance sur ceux qui sont encore en phase d'inscription.

3.2.1.2.3 *Système autogéré par ses membres*

C'est pourquoi cette crypto-monnaie va être distribuée localement pendant une certaine période, puis sera laissée sous l'unique contrôle des membres locaux initiaux. L'admission de nouveaux membres sera soumise au vote, ceux-ci seront acceptés seulement si la majorité (définie à plus des deux tiers) donne son approbation.

C'est un moyen évolutif (parce qu'un mécanisme de consensus est mis en place pour admettre de nouveaux membres), mais non permissif. Il permet, tout de même, de garantir qui a le droit d'utiliser certaines fonctionnalités dans le *Smart Contract*.

Ce système, réservé aux membres, empêche aussi que des adresses soient créées massivement pour emprunter. Par exemple, un individu pourrait vouloir se créer mille fois un nouveau compte pour chaque fois retirer le maximum initial (500 CHF) et quitter de suite le système.

3.2.1.2.4 Demande d'emprunt

Une demande d'emprunt peut être émise par un membre dans le but d'obtenir un prêt de *Tokens* à 0% d'intérêt.

Dès la fin de la période Q1 , il est possible d'effectuer une demande d'emprunt à hauteur du maximum empruntable (voir chapitre ci-dessous). Lorsque qu'un membre a emprunté son maximum empruntable, trois solutions s'offrent à lui :

- Rembourser intégralement son emprunt et obtenir une récompense (le *Smart Contract* prendra acte qu'il est une personne et lui permettra donc d'emprunter plus d'un seul coup en augmentant son maximum empruntable)
- Rembourser une partie (par exemple 10 *Tokens*), et obtenir le droit de réemprunter 10 *Tokens* par la suite s'il venait à traverser une mauvaise passe.
- Ne jamais rembourser et partir avec la somme empruntée, ce qui l'obligerait à quitter définitivement le système et le priverait de la possibilité de réemprunter.

Un membre n'est pas contraint à effectuer des demandes d'emprunts, cependant cette fonctionnalité peut être vraiment utile à une entreprise, il est donc très peu probable qu'elle ne souhaite pas en bénéficier.

3.2.1.2.5 Maximum empruntable

Le maximum empruntable permet, comme il est dit précédemment, de ne pas autoriser un membre à emprunter plus qu'un certain nombre de *Tokens*.

Le maximum empruntable est initialement défini à 500 CHF.

3.2.1.2.6 Demandes traitées automatiquement (FIFO)

Au départ, la fonction de prêt devait permettre de choisir à qui on voulait prêter et donc de choisir quelle demande serait traitée. Ce fonctionnement pose problème, car, dans un système où chaque demande qui est légitimement émise dans les règles doit être traitée, il n'est pas juste de permettre à un membre de rejeter certaines demandes.

C'est pourquoi ,il est prévu que les demandes de prêt soient traitées en faisant la queue (*First in first Out* ou *FIFO*). Les prêts seront accordés à la demande la plus ancienne, forçant, ainsi, les membres à répondre à toutes les demandes légitimes.

3.2.1.2.7 Prêt et remboursement

Un prêteur pourrait manquer de chance en ayant prêté à un membre qui est malhonnête. Ceci est d'autant plus problématique dans notre cas, puisque les demandes sont traitées automatiquement, il est donc impossible de choisir à qui l'on veut prêter.

Afin de garantir que les remboursements se fassent dans tous les cas, le prêteur reçoit, en prêtant, le droit de réemprunter. Cette mesure est plus efficace, car le prêteur aura plus de chance de recevoir un prêt par un autre membre (bon payeur) de la communauté, que s'il devait compter sur le remboursement fait par celui qui lui a concrètement emprunté.

Le prêt et le remboursement vont donc tous deux être utilisés pour donner des *Tokens* aux demandeurs, ces deux fonctionnalités ne sont donc en réalité qu'une seule et unique fonctionnalité. En résumé, on ne peut que demander un prêt (recevoir des *Tokens*) ou prêter (donner des *Tokens*) à la communauté.

Tout ce qui compte est que ceux qui ont donné peuvent ensuite recevoir et ceux qui ont reçu sont contraint à donner à la communauté s'ils espèrent avoir le droit de redemander des *Tokens*.

Effectuer un prêt à la communauté est comptabilisé par le *Smart Contract*, ce qui permet de réemprunter, en tout cas à hauteur de la somme prêtée. Cependant les membres risquent de ne pas avoir envie de prêter à 0% d'intérêt, surtout si cela est juste motivé par la crainte de se retrouver dans le besoin et de devoir compter sur la générosité d'un autre membre. Afin de garantir que les membres prêtent à la communauté, toute action de prêter (rembourser) permet d'obtenir une augmentation de son maximum empruntable.

Un membre ne peut pas être contraint à rembourser car cela reviendrait à geler l'argent qu'il a emprunté et le rendre non utilisable (il n'y a aucune utilité à emprunter de l'argent si on ne peut pas l'utiliser). Cependant un membre a tout de même intérêt à rembourser grâce à l'introduction de cette récompense.

Un problème est qu'il n'est pas possible de prêter s'il n'y a pas de demande d'emprunt en attente. Heureusement, même si personne dans le système n'est autorisé à émettre une nouvelle demande car personne ne souhaite rembourser (ce qui est très improbable en vue des récompenses), de nouvelles demandes peuvent toujours être émises après l'élection d'un nouveau membre.

3.2.1.2.8 Récompense (augmentation du maximum empruntable)

La récompense pour le prêt est l'augmentation de son maximum empruntable qui se voit doublé. Les membres seront donc motivés à prêter, car, en plus de pouvoir à leur tour emprunter de l'argent, ils pourront emprunter le double.

De même, le membre aura intérêt à rendre l'argent qu'il a emprunté, car il pourra demander le double par la suite. Imaginons qu'un partenaire me prête 500 CHF, pourquoi ne pas les lui rendre si cela me permet d'emprunter non pas 500 mais 1000 CHF le mois prochain. De plus, le fait d'être détecté comme tricheur par ses partenaires locaux, ne peut être que négatif pour les affaires.

Le maximum empruntable équivaut à un indice de confiance, seuls les membres qui réintroduisent de l'argent (à hauteur de ce qu'ils ont emprunté) sont déclarés de confiance. Ceux qui n'ont pas tout remboursé n'auront pas de récompense. Ils pourront toujours réemprunter à hauteur de leur remboursement, mais seront tout de même limités par un petit maximum empruntable et donc une petite marge de manœuvre.

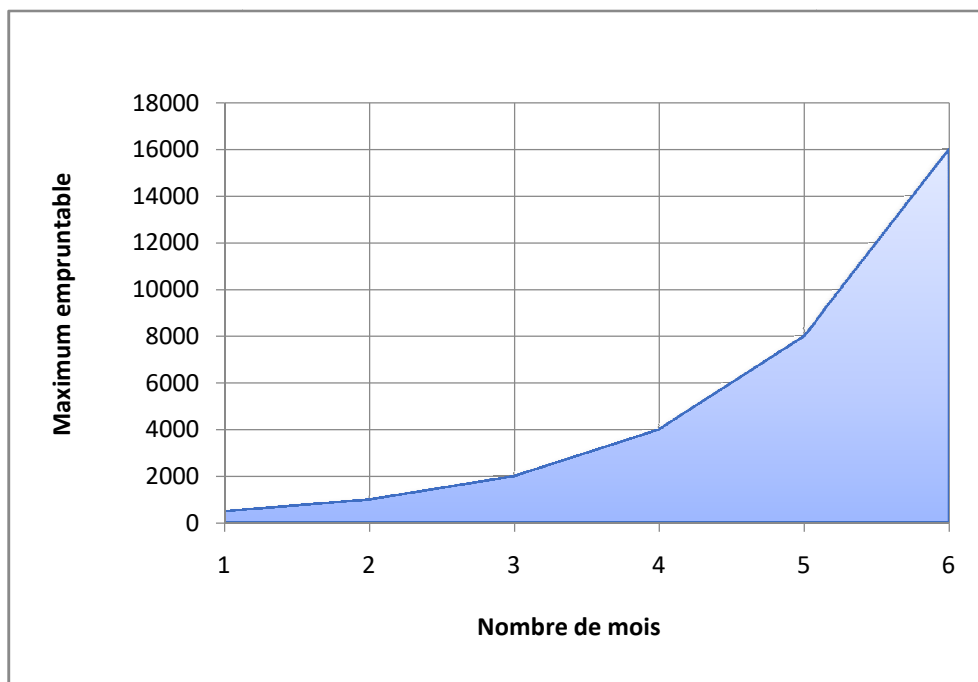
Par exemple :

Le membre A qui a emprunté 500 *Tokens*, mais qui n'en a remboursé que 200, il ne pourra réemprunter que 200 *Tokens*. Cependant, s'il décide de réemprunter, il devra tout de même rendre les 500 initiaux avant de bénéficier d'une augmentation de son maximum empruntable.

Alors que le membre B, lui qui a aussi emprunté 500 *Tokens*, mais les a entièrement remboursés, voit donc son maximum empruntable doublé, ce qui lui permet d'émettre une demande d'emprunt de 1000 *Tokens*.

Cette récompense ne peut se produire que si le prêteur a bien remboursé ses emprunts et n'a pas bénéficié d'une récompense depuis au moins 1 mois (ce délai est nécessaire afin de ne pas obtenir de la confiance trop rapidement).

Figure 5 : Évolution du maximum empruntable (optimiste)



(Création personnelle, 2017)

3.2.2 Manuel développeur (lien GitHub)

Disponible en annexe ou sur GitHub à l'adresse :

<https://github.com/jimy74/Chablex/blob/master/Documentation/manuelD%C3%A9veloppeur.pdf>

3.2.3 Manuel utilisateur (lien GitHub)

Disponible en annexe ou sur GitHub à l'adresse :

<https://github.com/jimy74/Chablex/blob/master/Documentation/manuelUtilisateur.pdf>

3.2.4 Bilan

3.2.4.1 Etat du projet

Toutes les solutions proposées ont été implémentées.

L'état actuel de ce projet est disponible sur GitHub à l'adresse :

<https://github.com/jimy74/Chablex>

3.2.4.2 Améliorations possibles

Les améliorations possibles sont décrites à la fin du manuel développeur.

De manière générale, il reste à implémenter une gestion du cycle de vie des *Smart Contracts*, des modifications pré-déploiement, ainsi que des tests automatisés.

4. Conclusion

Au départ il y avait le Bitcoin qui était sécurisé par sa propre *Blockchain*. C'est un outil très peu souple et, parce qu'il est verrouillé, il n'a pas été prévu pour permettre de créer, à l'intérieur, d'autres choses que ce pourquoi il a été conçu. Si on voulait, par exemple, créer une autre crypto-monnaie, il fallait, à partir d'une copie du protocole du Bitcoin, créer une autre *Blockchain* en ajoutant quelques modifications spécifiques. Cela nécessitait de vraiment bien comprendre son fonctionnement et aussi d'attirer beaucoup de mineurs si on voulait l'amener à un niveau de sécurité optimal. La plupart des premières crypto-monnaies ont été créées de cette manière.

Depuis le lancement de la *Blockchain* Ethereum, il n'est désormais plus nécessaire de recréer sa propre *Blockchain*. Grâce à Ethereum, il est possible de créer son propre programme, qui est un *Smart Contract*. Il n'est plus utile de repenser tout le fonctionnement de la *Blockchain* puisque les *Smart Contracts* vont venir s'inscrire directement dans celle-ci, offrant l'avantage de se concentrer uniquement sur le programme à décentraliser. Les *Smart Contracts* vont permettre de créer, de manière souple et ouverte, de nouvelles crypto-monnaies, mais aussi n'importe quelle application nécessitant d'être décentralisée. En effet, ce système permet de gérer, non pas seulement de la monnaie, mais aussi des informations. Il devient très facile et simple, en apprenant le langage de programmation adéquat (Solidity), de créer sa propre monnaie (*Token*) ; avec l'avantage de pouvoir rajouter des règles, comme dans le projet Chablex. La création et la mise en pratique de tels projets est simplifiée et accessible à tous. De plus, cela coûte moins cher et profite d'une sécurité optimale, car basée sur la *Blockchain* Ethereum (qui a fait ses preuves). Les *Smart Contracts* accélèrent donc la créativité dans le domaine de la *Blockchain*.

Tous les jours, de nouveaux projets sont financés par des *Smart Contracts* de type ICO. Les internautes n'attendent pas que les gouvernements légalisent les *Tokens* pour investir. Ils ne peuvent compter que sur leur instinct, car, les changements arrivant très rapidement, il n'existe aucune loi pouvant les guider ou les protéger. Malgré ce vide juridique, le marché des crypto-monnaies n'a fait que progresser. Bien que, ces dernières années, de nombreuses licences pour l'échange de Bitcoin aient été accordées, on en est encore dans certains pays à se poser la question de savoir si le Bitcoin est légal ou non. Certains pays (comme le Japon) reconnaissent le Bitcoin et autorisent même les paiements dans les magasins. (Keirns, 2017) Alors que d'autres gouvernements se lancent actuellement dans une mise en garde contre l'utilisation des

Smart Contracts (comme par exemple la Chine qui déclare les ICO illégales). (BBC, 2017)

Dans le futur, les *Smart Contrats* vont prendre inévitablement de plus en plus d'ampleur. La question sera de savoir comment rendre légal un *Smart Contract*? . Il faudrait reconnaître qu'il reste du chemin à faire dans ce domaine et non pas se contenter d'interdire ou de mettre en garde, mais d'instaurer des mécanismes de sécurité pour éviter à certains de perdre beaucoup d'argent. Il est néanmoins nécessaire de reconnaître que certains projets de création de *Tokens* sont novateurs et offrent de nouvelles perspectives pour le système monétaire de demain.

Il conviendrait donc, soit de redéfinir la notion de contrat pour que le *Smart Contract* puisse y trouver sa place, soit de définir de manière légale ce qu'est un *Smart Contract*. Ceci permettrait de le faire accepter et adopter par la société. On pourrait alors créer des *frameworks* utilisables par n'importe quel *Smart Contract*, qui permettrait de le rendre légal en lui implantant des règles basiques de bonne conduite. (Croiseaux, 2016) On pourrait même envisager que des *Tokens* soient reconnus légalement comme l'est la bourse.

Il est vrai qu'on a voulu s'affranchir des anciennes pratiques du système monétaire classique. Les *Smart Contracts* sont très permissifs, ils permettent de tout inventer, de rêver à un monde parfait, complètement décentralisé, impossible à pirater où toutes les règles sont fixées et respectées. Il faut cependant rester vigilant car les *Smart Contracts* sont des outils puissants qui doivent tout de même être utilisés à bon escient et en connaissance de cause (risques légaux, failles de sécurité). Il conviendrait aussi de définir plus de standards pour protéger ces *Smart Contracts*, de manière simple, efficace et durable.

Idéalement, il faudrait que la loi se marie avec les *Smart Contracts* afin que ceux-ci puissent fonctionner de manière parfaite pour les utilisateurs qui pourraient, en toute légalité, profiter de tous les avantages qu'offrent cette nouvelle et très prometteuse technologie.

Bibliographie

- Arnold, M. (2017). *Tech start-ups raise \$1.3bn this year from initial coin offerings*. Financial Times. Repéré à <https://www.ft.com/content/1a164d6c-6b12-11e7-bfeb-33fe0c5b7eaa>
- BBC. (2017, 5 septembre). *China bans initial coin offerings calling them 'illegal fundraising'*. Repéré à <http://www.bbc.com/news/business-41157249>
- Bentov, I., Gabizon, A. et Mizrahi, A. (2017). *Cryptocurrencies without Proof of Work*. Repéré à <https://arxiv.org/pdf/1406.5694.pdf>
- Bitcoin.org (s.d.) *Pourquoi les bitcoins ont-ils de la valeur*. FAQ. Repéré à <https://bitcoin.org/fr/faq#pourquoi-les-bitcoins-ont-ils-de-la-valeur>
- Blockchain France (12 mai 2016). *Qu'est-ce qu'une DAO?* Repéré à <https://blockchainfrance.net/2016/05/12/qu-est-ce-qu-une-dao/>
- Buterin, V. (2013 a). *History*. *Ethereum White Paper*. GitHub. Repéré à <https://github.com/ethereum/wiki/wiki/White-Paper#history>
- Buterin, V. (2013 b). *Ethereum State Transition Function*. *Ethereum White Paper*. GitHub. Repéré à <https://github.com/ethereum/wiki/wiki/White-Paper#ethereum-state-transition-function>
- Buterin, V. (2013 c). *Conclusion*. *Ethereum White Paper*. GitHub. Repéré à <https://github.com/ethereum/wiki/wiki/White-Paper#conclusion>
- Buterin, V. (2013 d). *Ethereum*. *Ethereum White Paper*. GitHub. Repéré à <https://github.com/ethereum/wiki/wiki/White-Paper#ethereum>
- Buterin, V. (2013 e). *Applications*. *Ethereum White Paper*. GitHub. Repéré à <https://github.com/ethereum/wiki/wiki/White-Paper#applications>
- Civic. (2017). *Whitepaper*. Repéré à <https://tokensale.civic.com/CivicTokenSaleWhitePaper.pdf>
- ConsenSys (2016). *Ethereum Contract Security Techniques and Tips*. GitHub. Repéré à <https://github.com/ConsenSys/smart-contract-best-practices>
- Croiseaux, F. (2016, 14 décembre). *Blockchain et gouvernance. Pourquoi un Smart Contract n'est pas un Contrat*. Repéré à <https://blog.intech.lu/index.php/2016/12/14/blockchain-et-gouvernance-pourquoi-un-smart-contract-nest-pas-un-contrat/>
- CryptoCompare. (2017, 5 juillet). *What is Ethereum Classic*. Repéré à <https://www.cryptocompare.com/coins/guides/what-is-ethereum-classic/>
- CryptoCompare. (2017, 13 avril). *What is a decentralized exchange*. Repéré à <https://www.cryptocompare.com/exchanges/guides/what-is-a-decentralized-exchange/>
- Dannen, C. (2017). *Mining Ether (chapitre 6) Dans Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. New York, USA: Apress. doi : 10.1007/978-1-4842-2535-6
- Deloitte (s.d.) *Mais à quoi servent les smart contracts ?* Repéré à <http://equationdelaconfiance.fr/decryptage/mais-quoi-servent-les-smart-contracts>

- Epié, C. et Loubet, N. (2016, novembre). *Démystification de la blockchain (ce qu'est la blockchain et ce qu'elle n'est pas)* [Atelier 1]. Forum de l'économie numérique: la blockchain et ses opportunités pour les entreprises, Genève.
- Farine, M. (2017, 26 janvier). *La blockchain fait son entrée dans la finance suisse*. Le temps. Repéré à <https://www.letemps.ch/economie/2017/01/26/blockchain-entree-finance-suisse>
- Filippi, P., & Hassan, S. (2016). *Blockchain technology as a regulatory technology: From code is law to law is code*. First Monday, 21(12). doi:<http://dx.doi.org/10.5210/fm.v21i12.7113>
- Floersch, K. (2017, 30 juin). *Casper & Smart Contract Consensus* [Vidéo en ligne]. Repéré à <https://media.consensys.net/casper-smart-contract-consensus-7be6cfa6f7ec>
- Garessus, E. (2017, 6 février). *Un vide juridique pèse sur la technologie blockchain*. Le Temps. Repéré à <https://www.letemps.ch/economie/2017/02/06/un-vide-juridique-pese-technologie-blockchain>
- Gilbert S. et Lynch N. (2002). *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. ACM SIGACT News, 33 (2) 51-59. Repéré à <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.1495&rep=rep1&type=pdf>
- Greenspan, G. (2016, 12 avril). *Beware of the Impossible Smart Contract*, Blockchain news. Repéré à <https://www.multichain.com/blog/2016/04/beware-impossible-smart-contract/>
- Greenspan, G. (2016, 17 avril). *Smart Contract Use Cases Are Simply Impossible*. Repéré à <https://www.coindesk.com/three-smart-contract-misconceptions/>
- Gubik, M. (2017). *Proof of Stake FAQ*. GitHub. Repéré à <https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ>
- Hammerschmidt, C. (2017). *Consensus in Blockchain Systems*. In Short. Medium. Repéré à <https://medium.com/@chrshmmmr/consensus-in-blockchain-systems-in-short-691fc7d1fefe>
- Hertig, A. (2017). *Ethereum's Big Switch: The New Roadmap to Proof-of-Stake*. CoinDesk. Repéré à <https://www.coindesk.com/ethereums-big-switch-the-new-roadmap-to-proof-of-stake/>
- iurimatias. (s.d.) *What is Embark*. GitHub. Consulté le 25 septembre 2017 à <https://github.com/iurimatias/embark-framework>
- Keane, J. (2017). *\$35 Million in 30 Seconds: Token Sale for Internet Browser Brave Sells Out*. Coin Desk. Repéré à <https://www.coindesk.com/35-million-30-seconds-token-sale-internet-browser-brave-sells/>
- Keirns G. (2017, 31 mars). *Japan's Bitcoin Law Goes Into Effect Tomorrow*. CoinDesk. Repéré à <https://www.coindesk.com/japan-bitcoin-law-effect-tomorrow/>
- King, S. et Nadal, S. (2012). *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake*. PeerCoin. Repéré à <https://peercoin.net/assets/paper/peercoin-paper.pdf>
- Krawisz, D. (2013). *Crypto-Anarchy and Libertarian Entrepreneurship Chapter 1: The Strategy*. Nakamoto Institute. Repéré à <http://nakamotoinstitute.org/mempool/crypto-anarchy-and-libertarian-entrepreneurship-1/#selection-9.6-13.24>
- Lamport, L., Shostak, R. et Pease, M. (1982). *The Byzantine Generals Problem*. ACM Transactions on Programming Languages and Systems, 4/3, 382-401

- Lehdonvirta, V. & Robleh, A. (2016). *Governance and Regulation*. In: M. Walport (ed.), *Distributed Ledger Technology: Beyond Blockchain*. London: UK Government Office for Science, pp. 40-45. Repéré à <http://vili.lehdonvirta.com/wp-content/uploads/2016/05/Lehdonvirta-Ali-2016-Distributed-ledger-governance-regulation.pdf>
- Lesert, A. (2002). *Tolérance aux pannes : Algorithme des généraux Byzantins*. Repéré à <http://aymeric.lesert.pagesperso-orange.fr/expose/dea/byzantin/algorithmes.pdf>
- Moore, G. E. (1965). *Cramming More Components Onto Integrated Circuits*. *Electronics*, vol. 38. Repéré à <https://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf>
- Open Zeppelin. (s.d.) *Zeppelin Solidity*. GitHub. Consulté le 26 septembre 2017 à <https://github.com/OpenZeppelin/zeppelin-solidity>
- o0ragman0o. (s.d.) *ITT - Intrinsically Tradable Token*. GitHub. Consulté le 26 septembre 2017 à <https://github.com/o0ragman0o/ITT>
- Popper, N. (2015, 15 Mai). *Decoding the Enigma of Satoshi Nakamoto and the Birth of Bitcoin*. *The New York Times*. Repéré à <https://www.nytimes.com/2015/05/17/business/decoding-the-enigma-of-satoshi-nakamoto-and-the-birth-of-bitcoin.html>
- Popper, N. (2017, 19 juin). *Move Over, Bitcoin. Ether Is the Digital Currency of the Moment*. *The New York Times*. Repéré à <https://www.nytimes.com/2017/06/19/business/dealbook/ethereum-bitcoin-digital-currency.html?mcubz=3>
- Preuve d'enjeu. (s.d.) Dans *Wikipédia, l'encyclopédie libre*. Consulté le 26 septembre 2017 à https://fr.wikipedia.org/wiki/Preuve_d%27enjeu
- Renard, J.P. (2017, 18 août). *Le "Bitcoin Cash" annonce-t-il la fin du "Bitcoin"?* La tribune. Repéré à <http://www.latribune.fr/opinions/tribunes/le-bitcoin-cash-annonce-t-il-la-fin-du-bitcoin-747305.html>
- Savelyev, A. (2016). *Contract Law 2.0: «Smart» Contracts As The Beginning Of The End Of Classic Contract Law*. Working paper No. WP BRP 71/LAW/2016. National Research University Higher School of Economics (HSE). Repéré à https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2885241
- Service de la promotion économique de Genève (2016). *Forum Économie Numérique - Transition Numérique: La Blockchain et ses opportunités pour les entreprises*. Repéré à <https://ge.ch/ecoquichetpmepmi/forum2016>
- Script (s.d.) Dans *Bitcoin Wiki*. Consulté le 26 septembre 2017 à <https://en.bitcoin.it/wiki/Script>
- Schneider, A., Durand-Garçon, K., Masurel, F. et Lorcery, P. (2016). *Qu'est-ce qu'une "attaque des 51%"*. Les Dossiers de CryptoFR. Repéré à <https://dossiers.cryptofr.com/quest-ce-quune-attaque-des-51/>
- Szabo, N. (1994). *Smart contracts in Essays on Smart Contracts, Commercial Controls and Security*. Repéré à <http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwiinterschool2006/szabo.best.vwh.net/smart.contracts.html>
- Szabo, N. (1997). *Formalizing and Securing Relationships on Public Networks*. First Monday, 2(9). doi:<http://dx.doi.org/10.5210/fm.v2i9.548>

Szabo, N. (2016). *Web conference "Blockchains and Smart Contracts"*. Repéré à <https://www.youtube.com/watch?v=tWuN2R2DC6c>

TheEthereumWiki (s.d.) *ERC20 Token Standard*. Consulté le 21 septembre 2017 à https://theethereum.wiki/w/index.php/ERC20_Token_Standard

Vitiko. (s.d.) *Congress*. GitHub. Consulté le 18 août 2017 à <https://github.com/vitiko/solidity-test-example/blob/master/contracts/Congress.sol>

Wikipedia (s.d.) *Automated Trading System*. Consulté le 28 août 2017 à https://en.wikipedia.org/wiki/Automated_trading_system

Young, J. (2017). *Harvard Researcher: Based on Moore's Law, Bitcoin Will Hit \$100,000*. The Merkle. Repéré à <https://themerke.com/harvard-researcher-based-on-moores-law-bitcoin-will-hit-100000/>

Annexe 1 : Manuel Développeur

Table des matières

1. Introduction.....	56
2. Environnement	56
2.1 Smart Contracts	56
2.2 Application côté client.....	56
2.3 Problèmes rencontrés	57
3. Structure des Smart Contracts.....	60
4. Implantation de la logique de base et des standards.....	60
4.1 SafeMath.....	60
4.2 Rendre le <i>Token</i> transférable.....	61
5. Implantation de la logique métier.....	62
5.1 Retirer les droits des créateurs après 90 jours.....	62
5.2 Distribution des <i>Tokens</i>	65
5.3 Crypto-monnaie pour des membres autogérés	66
5.4 Autoriser une demande de prêt	69
5.5 Demander un prêt	71
5.6 Autoriser un prêt.....	72
5.7 Emettre un prêt.....	72
5.8 Récompenser les prêteurs	74
5.9 Prêts automatiquement accordés aux demandeurs.....	75
6. Plateforme Chablex	76
6.1 Structure du projet.....	77
6.2 Interactions avec les <i>Smart Contracts</i> (concept).....	78
6.3 Interactions avec les <i>Smart Contracts</i> (code).....	78
7. Améliorations possibles	79

7.1	Côté <i>Smart Contracts</i>	79
7.2	Côté application (plateforme).....	80
7.3	Créer un plan marketing et financier	80
7.4	Recruter une équipe de développeurs aguerris.....	80

1. Introduction

Ce document vise à expliquer le fonctionnement des *Smart Contracts* et de la plateforme développés pour le projet Chablex. Il permet aussi d'expliquer l'environnement utilisé pour ce projet (encore en mode simulation). L'ensemble du projet est disponible sur le site <https://github.com/jimy74/Chablex>.

2. Environnement

2.1 *Smart Contracts*

Outils de développement des *Smart Contracts* : Ethereum Remix

Mise en route :

- a) Aller sur le l'IDE en ligne pour développer des *Smart Contracts* :
<https://remix.ethereum.org/>
- b) Définir l'environnement JavaScript VM

2.2 Application côté client

Système d'exploitation : Ubuntu (sur Windows avec virtual box).

Framework : Embark 2.5.2

Outil de développement (IDE) : Atom

Navigateur utilisé pour les tests : Firefox

Mise en route :

1. Installer Embark

```
$ npm -g install embark
```

2. Installer le simulateur

```
$ npm -g install ethereumjs-testrpc
```


3. Créer un projet de démonstration

```
$ embark demo  
$ cd embark_demo
```

4. Démarrer le simulateur

```
$ embark simulator
```

5. Lancer Embark

```
$ embark run
```

6. Ouvrir Atom

```
$ atom .
```

Mise à jour :

```
$ npm uninstall -g embark-framework  
$ npm install -g embark
```

Puis procéder à la mise en route ci-dessus.

La mise en route officielle d'Embark (si nécessaire) :

Iurimatias. (s.d.) *What is Embark*. Consulté le 25 septembre 2017 à <https://github.com/iurimatias/embark-framework>

2.3 Problèmes rencontrés

Problème : Gaz insuffisant pour que Embark compile les *Smart Contracts*

Solution : Augmenter le gaz limite dans le fichier

Dans le fichier `embark_demo/config/contracts.json` :

```
{  
  "default": {  
    "gas": "auto",  
    "contracts": {  
      "MonContract1": {"gas":3000000, "args":[10000]},  
      "MonContract2": {"gas":4500000, "args":[10000]},  
    }  
  }  
}
```

```
"MonContract3": {"gas":3000000, "args":[10000]}  
  
}  
  
}  
  
}
```

Problème : Embark indique qu'il ne trouve pas de *Blockchain*

Solution :

- a) Arrêter Embark en pressant les touches CTRL + C
- b) Arrêter le simulateur en pressant les touches CTRL + C
- c) Taper la commande `embark blockchain` puis l'arrêter après qu'il initialise la *Blockchain* en pressant les touches CTRL + C
- d) Puis relancer le simulateur avec la commande `embark simulator`
- e) Dans une autre console, relancer Embark avec la commande `embark run`

Problème : Une transaction coûte trop cher en gaz et ne peut être exécutée

Solution : Augmenter la limite de gaz lors de l'appel de cette transaction

Par exemple si dans `MonContract.sol` on veut appeler `maFonction` prenant deux paramètres :

```
MonContract.maFonction([param_1],[param_2],{gas:4712388,gasPrice:10^11}).then  
(  
  
  function(value) {  
  
    console.log("Action accomplie :" + value);  
  
  }).catch(function(error){  
  
    console.log("Erreur :" + error);  
  
  });
```

Problème : En mode simulation, on est le seul à faire des transactions sur la *Blockchain* locale, si on ne fait pas de transaction le temps reste figé (temps obtenu par l'instruction *now* dans un *Smart Contract*)

Solution : Ecrire un *timer* en JavaScript qui, tant que le temps n'a pas évolué depuis x secondes, va mettre à jour le temps localement (temps inchangé - x).

Problème : Comment tester l'application en simulant plusieurs comptes différents

Solution : Embark simulator permet d'utiliser 10 comptes de tests, ceux-ci sont reconnus automatiquement par web3. Il suffit ensuite de choisir l'un de ces comptes et de le passer en paramètre lorsqu'on appelle une fonction du *Smart Contract*.

```
var monCompte = web3.eth.accounts[0]; //Le premier des 10 comptes de simulations

MonContract.maFonction([param_1],[param_2],{from:monCompte}).then(

  function(value) {

    console.log("Action accomplie :" + value);

  }).catch(function(error){

    console.log("Erreur :" + error);

  });
```

Problème : Embark ne veut pas instancier un *Smart Contract* servant d'interface (par exemple ERC20.sol et ERC20Basic.sol de la librairie Open Zeppelin)

Solution : Réaliser cette interface dans vos *Smart Contracts* concrets et supprimer l'interface (par exemple dans BasicToken.sol et StandardToken.sol de Open Zeppelin)

3. Structure des *Smart Contracts*



(Création personnelle, 2017)

4. Implantation de la logique de base et des standards

4.1 SafeMath

Eviter de vérifier les bugs d'*over flow* en utilisant, par exemple, à la place de l'opération "+" la méthode `.add()`.

Les quatre opérations élémentaires sont présentes : `add()`; `sub()`; `mul()`; `div()`.

Il suffit donc d'écrire en début de *Smart Contract* :

```
using SafeMath for uint256;
```

Puis il sera possible de faire :

```
uint256 a = 1;  
  
uint256 b = 3;  
  
uint256 resultat = a.add(b); //Le résultat vaut 4
```

Ces fonctions sont implantées dans le *Smart Contract* "SafeMath.sol"

4.2 Rendre le *Token* transférable

Rendre le *Token* transférable en respectant le standard ERC-20

Le *Token* doit implémenter les fonctions suivantes :

```
// https://github.com/ethereum/EIPs/issues/20

contract ERC20Interface {

    // Get the total Token supply

    function totalSupply() constant returns (uint256 totalSupply);

    // Get the account balance of another account with address _owner

    function balanceOf(address _owner) constant returns (uint256 balance);

    // Send _value amount of tokens to address _to

    function transfer(address _to, uint256 _value) returns (bool success);

    // Send _value amount of tokens from address _from to address _to

    function transferFrom(address _from, address _to, uint256 _value) returns
    (bool success);

    /* Allow _spender to withdraw from your account, multiple times, up to the
    _value amount. */

    /* If this function is called again it overwrites the current allowance with
    _value. */

    // this function is required for some DEX functionality

    function approve(address _spender, uint256 _value) returns (bool success);

    // Returns the amount which _spender is still allowed to withdraw from
    _owner

    function allowance(address _owner, address _spender) constant returns
    (uint256 remaining);

    // Triggered when tokens are transferred.

    event Transfer(address indexed _from, address indexed _to, uint256 _value);

    // Triggered whenever approve(address _spender, uint256 _value) is called.
```

```
event Approval(address indexed _owner, address indexed _spender, uint256
_value);
}
```

TheEthereumWiki (s.d.) *ERC20 Token Standard*. Consulté le 21 septembre 2017 à https://theethereum.wiki/w/index.php/ERC20_Token_Standard

Ces fonctions sont implantées dans les *Smart Contracts* "BasicToken.sol" et "StandardToken.sol".

De plus, le *Token* doit posséder les informations suivantes :

```
string public constant symbol = "FIXED";

string public constant name = "Example Fixed Supply Token";

uint8 public constant decimals = 18;

uint256 _totalSupply = 1000000;
```

TheEthereumWiki (s.d.) *ERC20 Token Standard*. Consulté le 21 septembre 2017 à https://theethereum.wiki/w/index.php/ERC20_Token_Standard

Ces constantes sont définies dans le *Smart Contract* "ChabToken.sol".

5. Implantation de la logique métier

5.1 Retirer les droits des créateurs après 90 jours

L'adresse utilisée pour publier le contrat proviendra d'un porte-monnaie multi-signatures et dépendra donc de l'autorité de plusieurs personnes dont un huissier. Leurs privilèges, en tant que créateurs, par exemple l'ajout de membres et la distribution des *Tokens*, seront inaccessibles après la période de lancement de 90 jours (appelée Q1).

La période Q1 permet de définir correctement les membres de la communauté (de la monnaie locale) qui doivent correspondre à certains critères tels que: posséder une entreprise et appartenir à la localité de cette nouvelle crypto-monnaie. Cette période permet aussi de mettre en circulation les tout premiers *Tokens* vendus aux membres

au prix de 1 CHF, ceux-ci seront mis en sécurité dans le coffre d'une banque Suisse locale.

Après la période Q1, ce sont les membres eux-mêmes qui pourront voter sur l'adhésion des nouveaux membres, mais aussi pourront commencer à transférer, demander ou faire des prêts.

Cette règle est implantée dans le *Smart Contract* "MomentallyOwned.sol" comme ceci :

```
contract MomentallyOwned is StandardToken {

    address public owner;

    uint public creationTime; /*Not a constant because "now" should not be
    initialize in compile-time */

    uint public constant periodQ1 = 90 days;

    function MomentallyOwned() {owner = msg.sender; creationTime = now;}

    modifier onlyOwner {require(msg.sender == owner);_;}

    modifier onlyInQ1 {require(now <= creationTime.add(periodQ1));_;}

    modifier onlyAfterQ1 {require(now > creationTime.add(periodQ1));_;}

    /* _; is used to indicate to the function modified where it should be
    inserted */

    //The continuation of the contract ...
}
```

Source initiale : "Ownable.sol" de la librairie Open Zeppelin :

Open Zeppelin, (s.d.) *Zeppelin Solidity*. Consulté le 26 septembre 2017 à

<https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/ownership/Ownable.sol>

5.1.1.1.1 *Inscription initiale des membres*

Un membre est une structure (simple *record*) définie comme ci-dessous:

```
struct Member {

    address member; //Ethereum public key of the member

    string name; //Official name of the member
}
```

```
uint256 memberSince; //Date and time when the member is added  
}
```

La liste des membres est donc faite dans un simple tableau de membres comme ceci :

```
Member[] public members;
```

Les créateurs du *Token* pourront ajouter des membres pendant la période initiale Q1.

Un membre se constitue d'un nom, identifiant l'entreprise locale, et de sa clé publique Ethereum.

Pour ajouter un membre il suffit donc d'appeler la fonction `addMember` :

```
function addMember(address targetMember, string memberName)  
  
onlyOwner onlyNotMembers(targetMember) onlyInQ1 {  
  
    uint256 id;  
  
    memberId[targetMember] = members.length;  
  
    id = members.length++;  
  
    members[id] = Member({  
  
    member: targetMember, memberSince: now, name: memberName});  
  
    MembershipChanged(targetMember, true);  
  
}
```

En cas d'erreur de la part des créateurs, lors de la période Q1, ils pourront aussi supprimer un membre durant cette phase.

```
function removeMember(address targetMember)  
  
onlyOwner onlyInQ1 onlyMembers(targetMember) returns (bool){  
  
    for (uint256 i = memberId[targetMember]; i < members.length - 1; i++){  
  
        members[i] = members[i+1];  
  
        memberId[targetMember] = 0;  
  
    }  
  
}
```



```

balances[targetMember] = 0;

delete members[members.length-1];

members.length--;

MembershipChanged(targetMember, false);

}

```

Il est possible de contrôler si une adresse fait bien partie des membres grâce aux deux *modifieurs* suivants :

```

//Modifieur that allows only members

modifieur onlyMembers(address addr) {

    require(memberId[addr] != 0);

    _; //Indique où insérer le code de la fonction appelante

}

//Modifieur that allows only not members

modifieur onlyNotMembers(address addr) {

    require(memberId[addr] == 0);

    _;

}

```

Le *onlyMembers* est utilisé sur toute les fonctions liées aux *Tokens* et aux votations de nouveaux membres alors que *onlyNotMembers* est utilisée pour être sûr qu'on n'ajoute pas deux fois la même adresse dans la liste des membres, car cela reviendrait à lui doubler le nombre de vote qu'il peut émettre pour l'élection de nouveaux membres.

Ces fonctions et *modifieurs* sont définis dans le *Smart Contract* "CongressOwned.sol".

Source Initiale :

Vitiko. (s.d.) *Congress*. Consulté le 18 Août 2017 à

<https://github.com/vitiko/solidity-test-example/blob/master/contracts/Congress.sol>

5.2 Distribution des *Tokens*

Les créateurs peuvent distribuer des *tokens* aux membres seulement pendant la période Q1. Cette fonctionnalité permet aux créateurs de vendre des *Tokens* aux nouveaux membres, le prix initial étant défini à 1 CHF par *Token*.

Cette fonction est définie dans le *Smart Contract* "MintableToken.sol" (initialement de la librairie *Open Zeppelin*) de la manière suivante :

```
contract MintableToken is CongressOwned {

    event Mint(address indexed to, uint256 amount);

    /** @dev Function to mint tokens

     * @param _to The address that will receive the minted tokens.

     * @param _amount The amount of tokens to mint.

     * @return A boolean that indicates if the operation was successful. */

    function mint(address _to, uint256 _amount) onlyInQ1 onlyOwner
    onlyMembers(_to){

        totalSupply = totalSupply.add(_amount);

        balances[_to] = balances[_to].add(_amount);

        Mint(_to, _amount);

        Transfer(0x0, _to, _amount); /* Detail : 0x0 Because we create new tokens in
        the minting process (there is no sender of the transfer in term of loss)*/

    }
}
```

5.3 Crypto-monnaie pour des membres autogérés

Les actions possibles comme: le vote de nouveaux membres, le transfert, mais aussi la demande de prêt sont accessibles uniquement par des membres.

Il sera possible, pour les membres après la période Q1, de proposer de nouveaux membres et de voter pour ceux qui semblent de confiance (car faisant partie de la localité ou d'une région proche). Une proposition de nouveau membre est retenue si elle a reçu au moins 67% de votes positifs avant la fin du temps imparti (disons un mois). Ces paramètres sont des constantes du *Smart Contract* "CongressOwned.sol" ,

ils ne peuvent être définis, uniquement, avant la compilation et donc forcément avant sa publication sur la *Blockchain* Ethereum.

La proposition d'un nouveau membre est implantée dans le *Smart Contract* "CongressOwned.sol" comme suit :

```
function newProposal( address candidateAddress,string candidateName)
onlyAfterQ1 onlyMembers(msg.sender) onlyNotMembers(candidateAddress)
returns (uint256 proposalID) {
    //Sender did not make a proposal for a while
    require(now >= timeLastProposal[msg.sender] + periodEnterProposal);
    //Update the time of his last proposal from the sender
    timeLastProposal[msg.sender] = now;
    //Create a new proposal :
    //Set the id of the new proposal and (after) increase the proposals array
    proposalID = proposals.length++;
    Proposal storage p = proposals[proposalID]; //Set the pointer
    p.id = proposalID; //Set the id of this proposal
    p.candidateAddress = candidateAddress; //Set the candidate ETH address
    p.candidateName = candidateName; //Set the candidate firm identifier
    p.votingDeadline = now + debatingPeriod; //Set the deadline of this proposal
    p.executed = false; //Set the proposal to unexecuted
    p.proposalPassed = false; /*Set the result of the proposal to false (unused
if not executed) */
    //Vote for my own proposal :
    Vote storage v = p.votes[p.votes.length++]; //Get a new vote structure
    v.voter = msg.sender; //Set the voter
    v.inSupport = true; //Set the stat of his vote (accepted or rejected)
    v.justification = "Creator's vote"; //Set the justification
```

```

    p.voted[msg.sender] = true; //Sender has voted for this proposal

    p.numberOfVotes = 1; //Set the count of votes

    p.currentResult = 1; //Set the count of acceptations

    numProposals = proposalID +1; //Update the number of proposals

    ProposalAdded(proposalID, candidateAddress, candidateName);

    return proposalID;

}

```

Les membres doivent être avertis que, une fois qu'un nouveau membre est accepté par votation, il n'est plus possible de le renier de la liste des membres. Ceci est bien entendu pour garantir la diversité et éviter les renversements de pouvoir où des membres bienveillants seraient supprimés par de nouveaux membres malveillants.

La possibilité de voter (d'accepter ou de rejeter) une proposition de nouveau membre est implantée dans le même *Smart Contract* comme ceci :

```

function vote(uint256 proposalID, bool supportsProposal, string
justificationText) onlyAfterQ1 onlyMembers(msg.sender) returns (uint256 voteID)
{

    Proposal storage p = proposals[proposalID]; //Get the proposal

    require(p.voted[msg.sender] == false); //If has already voted, cancel

    Vote storage v = p.votes[p.votes.length++]; //Get a new vote structure

    v.voter = msg.sender; //Set the voter

    v.inSupport = supportsProposal; //Set the vote stat (accepted or rejected)

    v.justification = justificationText; //Set the justification

    p.voted[msg.sender] = true; //Set this voter as having voted

    p.numberOfVotes++; //Increase the number of votes

    if (supportsProposal) //If they support the proposal

        p.currentResult++; //Increase score

    // Fire Events

```

```

        ProposalTallied(proposalID,p.currentResult,p.numberOfVotes,p.proposalPassed;
    }

    function executeProposal(uint256 proposalID) onlyAfterQ1 {

        Proposal storage p = proposals[proposalID];

        require(now >= p.votingDeadline); //Has the voting deadline arrived?

        require(!p.executed); //Has it not been already executed

        require(p.numberOfVotes >= minimumQuorum); //Has a minimum quorum?

        //If difference between support and opposition is larger than margin

        if ( p.currentResult * 100 / p.numberOfVotes >= majorityMinPourcent) {

            //Add the member

            addElectedMember(p.candidateAddress,p.candidateName);

            p.proposalPassed = true;

        } else {

            p.proposalPassed = false;

        }

        p.executed = true; //Note the proposal as executed

        //Fire this event

        Voted(proposalID, supportsProposal, msg.sender, justificationText);

        return p.numberOfVotes;

    }

```

Il suffit donc à quiconque sur le réseau Ethereum de demander l'exécution d'une proposition pour que cette proposition puisse être conclue sur une acceptation ou non du membre proposé. Une proposition ne peut être exécutée que si elle existe depuis plus d'une semaine et qu'elle a reçu au moins 3 votes (défini dans les constantes du *Smart Contract* "CongressOwned.sol").

5.4 Autoriser une demande de prêt

Le maximum empruntable est le nombre de *Tokens* maximum qu'un membre peut emprunter. Si le maximum empruntable est atteint, c'est-à-dire qu'il a emprunté jusqu'à

atteindre son maximum empruntable (initialement fixé à 500 *Tokens*), il ne peut, dès lors, plus faire de demande à moins de rembourser, par la fonction de prêt, la totalité de ses emprunts à la communauté.

Une règle nommée "peutDemander" est implantée en Solidity en utilisant un *modifier*.

Celle-ci stipule que l'on peut demander une valeur X si ces 5 conditions sont réunies :

- Le demandeur a remboursé ses emprunts
- La valeur X demandée est supérieure à 1 *Token*
- On ne peut pas emprunter plus de *Tokens* que le maximum empruntable
- Le total des demandes + la demande ne dépasse pas le total des remboursements + le max
- Le total demandé par la communauté ne dépassera pas le 1/3 du nombre total de *Tokens*

Mais attention, d'autres règles, telles que "*onlyMember*" et "*onlyAfterQ1*", sont aussi là pour protéger contre une demande incongrue.

La règle "peutDemander" est implantée sous forme d'un *modifier* dans le *Smart Contract* "ChabToken.sol" de la manière suivante :

```
modifier peutDemander(uint256 _value) {

    require(_value >= 1); // La valeur demandée est au moins 1 token

    //Définir le maximum empruntable ou sa valeur initiale

    uint monMaxEmpruntable = getMaxEmpruntable(msg.sender);

    //N'emprunte pas plus de tokens que le max

    require(_value <= monMaxEmpruntable);

    /* Le total des demandes plus la demande, ne dépassent pas le total des
    remboursements plus le maximum empruntable */

    require(demandes[msg.sender].add(_value)<=
    remboursements[msg.sender].add(monMaxEmpruntable));

    //Le total demandé est inférieur à 1/ratio du nombre de tokens total

    require((_value + demandesEnCours.getTotalValue()) *
    minRatioCirculent < totalSupply);

    _; //Indique où insérer le code de la fonction appelante
```

```
}
```

Remarquez que ce *Smart Contract* est codé et commenté en français car il a été conçu à partir de rien, alors que les autres contrats proviennent tous de bibliothèques ou d'autres ressources internes puis ont été modifiés. Le *Smart Contract* "ChabToken.sol" est donc le cœur métier du projet Chablex.

5.5 Demander un prêt

Pour tenter d'obtenir un prêt, il faut être autorisé, comme vu dans le chapitre ci-dessus, de plus, comme dit précédemment, il faut être membre et que la période Q1 soit terminée.

La fonctionnalité d'emprunt est implémentée dans le *Smart Contract* "ChabToken.sol" de la façon suivante :

```
function demander(uint256 _value) public
onlyAfterQ1 onlyMembers(msg.sender) peutDemander(_value) {
    //Augmente le total demandé
    demandes[msg.sender] = demandes[msg.sender].add(_value);
    //Ajoute la demande à la file d'attente
    demandesEnCours.addRequest(msg.sender, _value);
    //Déclenche l'évènement
    Demander(msg.sender, _value);
}
```

On observe, au passage, que le code devient plus court grâce aux différents *modifieurs* tels que "onlyAfterQ1", "onlyMembers" et "peutDemander".

5.6 Autoriser un prêt

On peut prêter la valeur X si ces 3 conditions sont réunies :

- La valeur X prêtée est supérieure à 0 *Token*
- Le prêteur possède suffisamment
- La valeur X du prêt ne dépasse pas le total des demandes en cours

Cette règle est implantée dans le *Smart Contract* "ChabToken.sol" comme ceci :

```
modifier peutPreter(uint256 _value) {  
  
    // La valeur prêtée est supérieure à 0 token  
  
    require(_value > 0);  
  
    // Le prêteur possède suffisamment  
  
    require(balances[msg.sender] >= _value);  
  
    /* La valeur du prêt ne dépasse pas le total des demandes en  
    cours, et les demandes complétées n'appartiennent pas au prêteur */  
  
    require(demandesEnCours.containMinValueFromOther(_value, msg.sender));  
  
    _;  
  
}
```

5.7 Emettre un prêt

Il est possible, pour un membre après la période Q1, de faire un prêt à la communauté, cela sera comptabilisé comme un remboursement de sa part et induira, peut-être (conditions expliquées dans le chapitre suivant), à une récompense (en augmentant son maximum empruntable).

La fonction "prêter" est définie dans le *Smart Contract* "ChabToken.sol" comme suit :

```
function preter(uint256 _value) public  
  
onlyAfterQ1 onlyMembers(msg.sender) peutPreter(_value){  
  
    // Déduit en amont la balance du prêteur  
  
    balances[msg.sender] = balances[msg.sender].sub(_value);  
  
    uint256 pretRestant = _value; // Initialise le montant qu'il reste à prêter
```



```

while (demandesEnCours.queueLength() > 0 && pretRestant > 0){

    /* Récupère la demande la plus vieille sans la dépiler
    (demandeur,valeur) */

    var (demandeur,valeur) = (demandesEnCours.copyPopRequest());

    // Si le prêt restant peut recouvrir cette demande intégralement

    if (pretRestant >= valeur){

        demandesEnCours.popRequest();

        //Augmente les emprunts total du demandeur

        emprunts[demandeur] = emprunts[demandeur].add(valeur);

        //Augmente la balance du demandeur

        balances[demandeur] = balances[demandeur].add(valeur);

        //Augmente les remboursements total du prêteur

        remboursements[msg.sender] =
        remboursements[msg.sender].add(valeur);

        // Réduire la somme encore prêtable

        pretRestant = pretRestant.sub(valeur);

        Preter(demandeur,valeur);

    } else { //Sinon, répondre partiellement à cette demande

        emprunts[demandeur]=emprunts[demandeur].add(pretRestant);

        balances[demandeur]=balances[demandeur].add(pretRestant);

        remboursements[msg.sender] =
        remboursements[msg.sender].add(pretRestant);

        /* Réduit la valeur de cette demande mais la laisse dans
        la file d'attente */

        demandesEnCours.replaceInFrontRequest(demandeur,
        valeur.sub(pretRestant));

        PreterUnePartie(demandeur, pretRestant);

```

```

        // Arrête la boucle car tout prêté

        pretRestant = 0;

    }

}

/* Et ici, donner selon certaines conditions, la récompense

    ce qui est expliqué dans le chapitre suivant ... */
}

```

5.8 Récompenser les prêteurs

Si le prêteur, une fois le prêt effectué correctement, se trouve dans la situation où il y a, en tout, prêté autant qu'il a emprunté, qu'il a déjà demandé jusqu'à sa limite maximale et que celle-ci n'a pas changé depuis 30 jours, alors le maximum qu'il peut emprunter la prochaine fois double.

Cette règle se trouve à la fin de la fonction de prêt dans le *Smart Contract* "ChabToken.sol", et est implantée comme ceci :

```

if (//Si le prêteur a remboursé ses emprunts

    remboursements[msg.sender] >= emprunts[msg.sender] &&

    //Et est arrivé à son maximum

    remboursements[msg.sender] >= getMaxEmprunable(msg.sender) &&

    //Et que sa dernière augmentation date de au moins 30 jours

    now >= dateChangementMax[msg.sender].add(tempsMinChangeMax)

) {

    dateChangementMax[msg.sender] = now;

    //Multiplie (par 2) le maximum emprunable lors du prochain emprunt

    maxEmprunable[msg.sender] =

    getMaxEmprunable(msg.sender).mul(facteurChangeMax);

    /* Dans ce cas il peut aussi faire un nouvel emprunt

```

```
(voir la première ligne du modifier peutDemander) */
```

```
}
```

5.9 Prêts automatiquement accordés aux demandeurs

Les prêts sont accordés grâce à une pile (First In First Out) qui est codée dans le *Smart Contract* "QueueDemande.sol". C'est un outil technique, il n'est donc pas nécessaire de l'analyser dans les détails pour comprendre le fonctionnement ChabToken. Cependant, la queue de demandes reste un point névralgique du système, elle reste une partie sensible qui mériterait d'être plus amplement revue et testée pour garantir la sécurité du système.

Cette fonctionnalité aurait besoin d'être améliorée, en tout cas pour vérifier que la pile (qui est en l'occurrence circulaire) ne finisse par réécrire sur elle même. Ceci n'est pas encore réalisé car le projet arrive à son échéance, une idée serait de tester la taille de la queue avant d'ajouter une nouvelle demande.

Vous retrouverez le *Smart Contract* "QueueDemande.sol" et tous les autres sur le GitHub du projet Chablex.

6. Plateforme Chablex

La plateforme Chablex est encore simulée dans un environnement de test en utilisant le *framework Embark* (voir la mise en route dans le chapitre Environnement).

Celui-ci simule une *Blockchain* localement et permet de manipuler plusieurs comptes et d'utiliser de faux Ethers afin d'éviter de payer réellement les transactions.

Les *Smart Contracts* doivent être mis dans le dossier des contrats, Embark va les compiler et les rendre accessibles en passant par du JavaScript.

Embark permet aussi de faire un nouveau projet de démonstration ou figure un exemple de code pour interagir avec un *Smart Contract* basic. Pour cela, il suffit de taper la commande "embark demo".

Cette plateforme respecte la structure d'un projet généré par Embark.

Cette plateforme coté client est donc codée en utilisant les langages bien connus des développeurs Web tel que HTML, CSS, et JavaScript (ainsi que jQuery et AJAX).

Pour mieux comprendre les rouages de cette plateforme, il est essentiel d'avoir compris cette structure (décrite ci-dessous).

6.1 Structure du projet



1. Dossier ou se trouvent les Smart Contracts à utiliser

2. Fichier CSS pour changer le style de la plateforme

3. Fichier JavaScript où on interagit avec les *Smart Contracts*

4. Fichier HTML contenant la structure de la plateforme

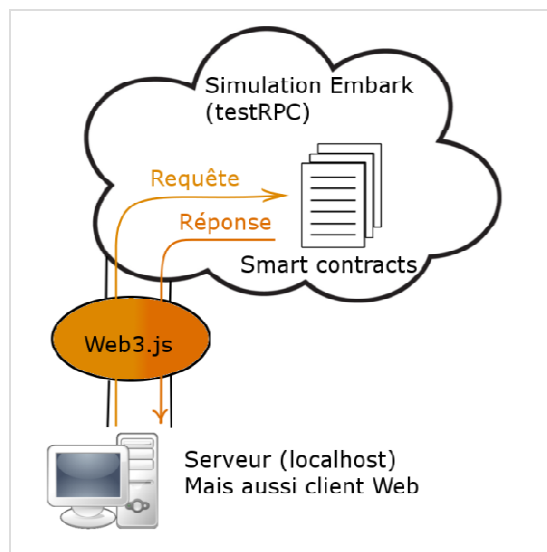
5. Fichier JSON permettant de modifier le gaz à utiliser pour compiler les Smart Contracts

6.2 Interactions avec les *Smart Contracts* (concept)

Les *Smart Contracts* sont pour, l'instant, publiés par Embark, dans une *Blockchain* de simulation.

Pour appeler une fonction d'un *Smart Contract*, le serveur (aussi simulé en local), va utiliser Web3.js pour effectuer des transactions sur la *Blockchain*. Une fois cette fonction appelée (en JavaScript), Web3.js va ouvrir ce qui s'appelle une "*promise*", et transmettra de manière asynchrone le résultat obtenu suite à cette requête.

Communication entre la plateforme et les *Smart Contracts*



(Création personnelle, 2017)

Si une erreur imprévue dans le *Smart Contract*, telle que la rencontre de l'instruction "*throw*", le résultat n'est pas reçu, mais une exception est déclenchée.

6.3 Interactions avec les *Smart Contracts* (code)

Voici un exemple de code qui permet de visualiser comment interagir avec un *Smart Contract* (dans un projet Embark) :

```
MonContract.maFonction([param_1],[param_2]).then(  
  
  function(value) {  
  
    console.log("Action accomplie :" + value);  
  
  }).catch(function(error){  
  
    console.log("Erreur :" + error);  
  
  });
```

7. Améliorations possibles

7.1 Côté *Smart Contracts*

- a) Utiliser des *Smart Contracts* de gestion de cycle de vie

La librairie Open Zeppelin propose trois *Smart Contracts* réutilisables (Pausable.sol, Migration.sol et Destructible.sol). Ils permettraient aux créateurs, ou aux membres par votation (après ajout de quelques lignes de code), de stopper le *Smart Contract* en cas d'attaque afin d'éviter une catastrophe.

Cela faciliterait la légalisation du projet Chablex car il offrirait un moyen d'arrêter si le gouvernement l'ordonnerait, réduisant le risque d'interdiction d'un tel projet.

Mais cela permettrait surtout de mettre à jour le *Smart Contract* ChabToken, car bien qu'on ne le prévoie pas forcément dès le départ, cela peut devenir très vite une nécessité en cas de bug ou d'attaque.

Tout ceci reste à faire, pour l'instant le *Smart Contract* ChabToken n'offre, après la période initiale, aucun moyen de l'arrêter, mais aussi aucun moyen de le mettre à jour (de manière transparente pour l'application coté client).

- b) Publier le *Smart Contract* sur un testNet Officiel

Cela permettrait d'accéder plus facilement au *Smart Contract* pour le tester, mais permettrait également de tester l'application coté client en utilisant cette fois MetaMask (un *wallet* d'Ether pour le navigateur Chrome).

- c) Proposer des récompenses pour les chasseurs de bugs

Ce serait le début d'un *Smart Contract* sécurisé, malheureusement cette étape est coûteuse et nécessiterait un moyen de financement qui resterait à définir.

Une solution serait d'utiliser les Francs Suisses acquis pendant la vente initiale.

- d) Tester le *Smart Contract* étape par étape

Ceci permettrait de, petit à petit, monter en exigence en réduisant le nombre de tests et donc le risque de failles de sécurité.

- e) Publier la version finale du *Smart Contract* sur la *Blockchain* Ethereum

Cela se ferait sûrement sur le testNet car, pour l'instant, on n'utilise pas d'Ether dans les *Smart Contracts* (ce qui réduit les frais de transaction). Une autre solution serait de le publier sur le mainNet qui a, au moins, l'avantage d'être sécurisé par l'existence d'un grand nombre de mineurs.

7.2 Côté application (plateforme)

- a) Trier les tableaux qui sont désordonnés à cause de l'asynchronicité des requêtes.
- b) Afficher le nombre de *Tokens* en circulation.
- c) Afficher le temps restant pour une proposition avant quelle puisse être exécutée.
- d) Effectuer des messages d'alertes plus fines.
- e) Créer un graphique pour visualiser l'évolution de l'état d'un compte dans le temps.
- f) Changer le code JavaScript pour que Web3.js se connecte au *Smart Contract* définitivement publié sur la *Blockchain*.
- g) Changer le code JavaScript pour que web3.js afin qu'il se lie à MetaMask.

7.3 Créer un plan marketing et financier

- a) Créer un site officiel
- b) Mettre en place des moyens de communication
- c) Rencontrer les entreprises locales
- d) Trouver des moyens de financement

7.4 Recruter une équipe de développeurs aguerris

Cela serait une des meilleure garantie pour la réussite du projet, autant sur le plan technique que marketing. Il faudrait qu'une telle équipe puisse être rémunérée dans le cadre de ce projet (à moins qu'ils acceptent de travailler gratuitement, ce qui est tout de même peu probable). Une deuxième solution, plus économique, mais avec malheureusement moins de garantie, serait qu'un autre étudiant développe une version 2 du projet Chablex dans le cadre de son futur travail de Bachelor.

Annexe 2 : Manuel utilisateur

Table des matières

1. Introduction.....	82
2. Interface utilisateur	82
3. Fonctionnalités de simulation	85
3.1 Changer de compte Ethereum	85
3.2 Recommandations pour simplifier les tests	85
4. Fonctionnalités des créateurs.....	85
4.1 Ajouter un membre	85
4.2 Supprimer un membre.....	86
4.3 Distribuer des <i>Tokens</i>.....	86
5. Fonctionnalités des membres	86
5.1 Actions financières	87
5.1.1 Transférer des <i>Tokens</i>	87
5.1.2 Demander un prêt en <i>Tokens</i>	87
5.1.3 Prêter des <i>Tokens</i>	88
5.2 Actions communautaires	88
5.2.1 Proposer un nouveau membre	88
5.2.2 Voter pour une proposition	88
5.2.3 Clore la votation pour une proposition	89
5.2.4 Afficher l'historique des votes pour une proposition.....	89

1. Introduction

Ce document est destiné aux futurs utilisateurs du Chablex, il est conçu pour expliquer comment interagir avec la future plateforme. Ce document peut aussi être utile pour tester la première version de cette plateforme qui fonctionne pour l'instant en mode simulation. Pour lancer réellement ce projet, aller au chapitre "Environnement > Application coté client > Mise en route" du manuel développeur.

2. Interface utilisateur



1. Liste d'adresses permettant de sélectionner celle à utiliser

Il est donc possible d'avoir plusieurs adresses, ceci est fort pratique pour cette simulation. En effet, jouer le rôle des créateurs et des membres permet d'effectuer des tests manuellement.

2. Affiche combien de temps il reste avant que la période Q1 soit terminée

C'est une information importante, car les droits des créateurs prennent fin dès que la période Q₁ est terminée. Les membres, par contre, doivent attendre la fin de cette période avant de pouvoir effectuer des actions sur cette plateforme.

La période Q1 ce termine dans : 0 H 1 Min 29 Sec

Ou, si la période Q1 est terminée :

La période Q1 est terminée

3. Onglet *wallet* (par défaut)

Dans cet onglet, on retrouve, sur la gauche, les informations concernant l'état financier du compte :

Mon compte : [Rafraîchir](#)

Balance : 2500 CHT

Demandses : 0 CHT

Emprunts : 0 CHT

Remboursements : 0 CHT

Max empruntable : 500 CHT Etat initial

Dans cet onglet, sur la droite, sont proposées les fonctionnalités financières telles que: le transfert, la demande d'emprunt et le prêt en *Tokens*. Une liste des demandes en cours y est aussi affichée. Voici, ci-dessous, la disposition de ces éléments :

Adresse ETH CHT

CHT

CHT

Liste des demandes en cours : [Rafraîchir](#)

Position	Adresse ETH	Montant
0	0x034a611d306f3b1a9813bf867e71ac8c13bb6821	500
1	0x34debc5bfd7a08e85996361f00df6442beb6b4e5	1000
2	0x7a992758381ebe985e0b60ede95c0284d219d5d6	200

4. Onglet membres

Dans la partie gauche de cet onglet, se retrouvent les fonctionnalités des créateurs (invisible pour les membres), qui permettent d'ajouter, de supprimer ou de distribuer des *Tokens* à un membre. Ces fonctionnalités ne seront plus autorisées après la

période Q1. En dessous se trouve la liste des membres (elle reste visible). Voici la disposition de cette partie :

Actions de démarrage

Ajouter un nouveau membre :

Supprimer un membre :

Distribuer de nouveaux CHT :

CHT

Liste des membres [Rafraîchir](#)

Nom	Adresse	Balance
Entreprise XX	0x7a992758381ebe985e0b60ede95c0284d219d5d6	2500 CHT
Entreprise XY	0x034a611d306f3b1a9813bf867e71ac8c13bb6821	2500 CHT

Dans la partie droite, se retrouvent les fonctionnalités que seuls les membres peuvent utiliser après la période Q1. Ces fonctionnalités permettent de proposer de nouveaux membres ainsi que de voter (Accepter ou Refuser) les propositions existantes. La liste des propositions est aussi affichée. Voici la disposition de cette seconde partie :

Actions communautaires

Proposer un nouveau membre :

Voter pour une proposition :

Accepter Refuser

Clore une votation

Liste des propositions [Rafraîchir](#)

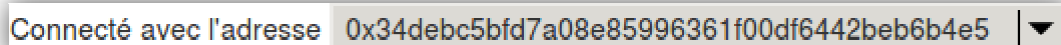
ID	Nom	Adresse	Etat	Votes reçus	% Acceptation
0	Entreprise Z	0xd1d3f736c509836e1b95b0224efb0d3b0d6fc1f	En cours	3	33 %

3. Fonctionnalités de simulation

Avant tout il est primordial que l'environnement soit bien configuré (voir la documentation développeur) et d'être connecté en *localhost* sur la plateforme Chablex.

3.1 Changer de compte Ethereum

Une fois sur la plateforme, une liste propose 10 adresses Ethereum. En cliquant dans cette liste, il est possible de sélectionner l'adresse choisie pour effectuer des transactions, et donc interagir avec la plateforme.



Connecté avec l'adresse 0x34debc5bfd7a08e85996361f00df6442beb6b4e5 ▼

Cette fonctionnalité est utilisée pour tester l'interaction entre différents acteurs alors, qu'en réalité, vous êtes seul et isolé dans cette simulation.

3.2 Recommandations pour simplifier les tests

En mode simulation, il est avantageux de changer certaines variables temporelles des *Smart Contract* afin de ne pas devoir attendre réellement la fin de la période Q1 (90 jours). En effet, c'est seulement après la clôture de la période Q1 que les fonctionnalités des membres deviennent actives, il est donc recommandé de réduire ce temps à, par exemple, 3 minutes. Ceci peut être fait, si nécessaire, sur toutes les variables temporelles, tout changement sur un *Smart Contract* va recompiler celui-ci et le remettre dans son état initial.

4. Fonctionnalités des créateurs

Ces fonctionnalités sont visibles uniquement par l'adresse Ethereum créatrice des *Smart Contracts* (par défaut la première de la liste des adresses proposées). Ces fonctionnalités seront dans tous les cas inutilisables après la période Q1.

4.1 Ajouter un membre

Dans l'onglet "Membre" à gauche, il est possible d'ajouter un membre en indiquant le nom (identifiant) de cette entreprise ainsi que son adresse Ethereum. Il suffit ensuite de cliquer sur le bouton "Ajouter".

Ajouter un nouveau membre :

Nom de l'entreprise	Adresse ETH	Ajouter
---------------------	-------------	---------

4.2 Supprimer un membre

Dans l'onglet "Membre", à gauche, il est possible de supprimer un membre en indiquant son adresse Ethereum, puis en cliquant sur le bouton "Supprimer".

Supprimer un membre :

Adresse ETH	Supprimer
-------------	-----------

4.3 Distribuer des *Tokens*

Dans l'onglet "Membre", à gauche, il est possible de distribuer des *Tokens* à un membre. Pour cela, il suffit d'indiquer l'adresse Ethereum de celui-ci ainsi que le nombre de *Tokens* à lui distribuer, puis cliquer sur le bouton "Envoyer".

Distribuer de nouveaux CHT :

Adresse ETH	0.01	CHT	Envoyer
-------------	------	-----	---------

Des nouveaux *Tokens* seront créés par le système et transférés au membre désigné. Puis la liste des membres se met à jour, ce qui permet de vérifier le nombre de *Tokens* que possède désormais ce membre.

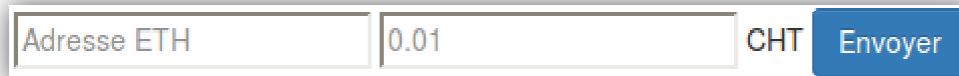
5. Fonctionnalités des membres

Les fonctionnalités des membres ne seront seulement accessibles par ceux-ci qu'à la fin de la période Q1. Une adresse Ethereum ne faisant pas partie de la liste des membres ne peut pas avoir accès à ces fonctionnalités.

5.1 Actions financières

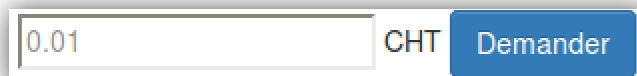
5.1.1 Transférer des *Tokens*

Dans l'onglet "*Wallet*", à droite, il est possible de transférer des *Tokens* à un autre membre. Pour cela il suffit d'indiquer l'adresse Ethereum du destinataire ainsi que le montant de *Tokens* à lui transférer puis cliquer sur le bouton "Envoyer".



5.1.2 Demander un prêt en *Tokens*

Dans l'onglet "*Wallet*", à droite, il est possible de demander un prêt de *Tokens*. Pour cela il suffit d'indiquer le montant à emprunter puis de cliquer sur le bouton "Demander".



Le système va, si le membre a encore le droit d'emprunter, prendre en compte sa demande de prêt et l'afficher dans la liste des demandes en cours.

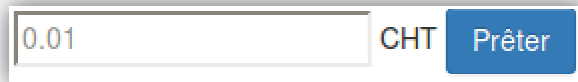
Liste des demandes en cours : [Rafraîchir](#)

Position	Adresse ETH	Montant
0	0x034a611d306f3b1a9813bf867e71ac8c13bb6821	500
1	0x34debc5bfd7a08e85996361f00df6442beb6b4e5	1000
2	0x7a992758381ebe985e0b60ede95c0284d219d5d6	200

Le prêt ne sera réellement accordé que lorsque la demande arrivera en haut de la file d'attente et qu'un membre fasse un prêt à la communauté.

5.1.3 Prêter des *Tokens*

Dans l'onglet "*Wallet*", à droite, il est possible de prêter des *Tokens*. Pour cela il suffit d'indiquer le montant à prêter puis de cliquer sur le bouton "Prêter".



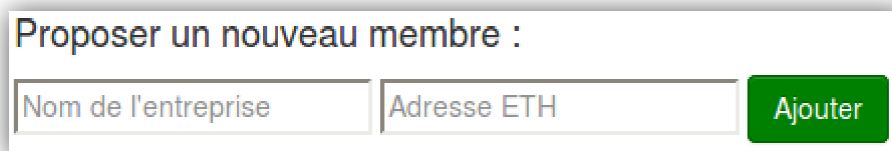
0.01 CHT Prêter

Le prêt sera accordé à la plus ancienne des demandes émises par les membres de la communauté (celle en haut de la liste des demandes en cours).

5.2 Actions communautaires

5.2.1 Proposer un nouveau membre

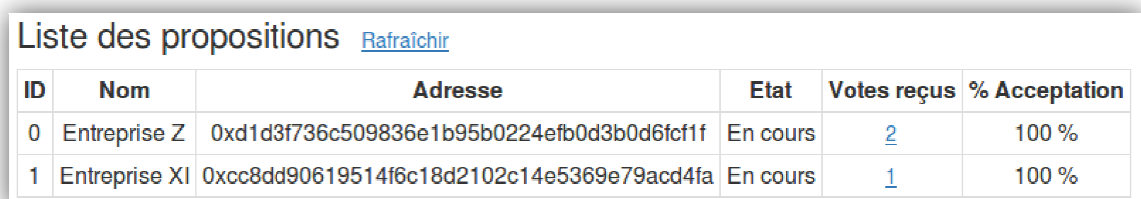
Dans l'onglet "Membres", à droite, il est possible de proposer un nouveau membre. Pour cela il faut indiquer le nom (identifiant) de cette entreprise ainsi que son adresse Ethereum, puis cliquer sur le bouton "Ajouter".



Proposer un nouveau membre :

Nom de l'entreprise Adresse ETH Ajouter

En proposant un nouveau membre, le système comptabilise en même temps un vote pour celui-ci. Ensuite, le système met à jour la liste des propositions.



Liste des propositions [Rafraîchir](#)

ID	Nom	Adresse	Etat	Votes reçus	% Acceptation
0	Entreprise Z	0xd1d3f736c509836e1b95b0224efb0d3b0d6fc1f	En cours	<u>2</u>	100 %
1	Entreprise XI	0xcc8dd90619514f6c18d2102c14e5369e79acd4fa	En cours	<u>1</u>	100 %

5.2.2 Voter pour une proposition

Dans l'onglet "Membre", à droite, il est possible de voter pour une proposition en cours. Pour cela, indiquer le numéro (identifiant) de la proposition, choisir d'accepter ou de refuser la proposition et donner une explication textuelle (qui est optionnelle). Finalement cliquer sur le bouton "Voter".

Voter pour une proposition :

Accepter Refuser

Le vote sera comptabilisé par le système qui va ensuite mettre à jour la liste des propositions en tenant compte du vote.

5.2.3 Clore la votation pour une proposition

Dans l'onglet "Membre", à droite, il est possible de clore une proposition (si celle-ci a reçu au moins 3 votes et est créée depuis au moins une semaine). Pour clore une proposition, il suffit d'indiquer son numéro (identifiant) et de cliquer sur le bouton "Clore".

Clore une votation

La proposition, une fois close, sera ensuite acceptée ou refusée, si elle est acceptée le candidat devient automatiquement membre.

5.2.4 Afficher l'historique des votes pour une proposition

Dans la liste des propositions (onglet "Wallet", à gauche), cliquer sur le nombre de votes de cette proposition. L'historique s'affiche ensuite comme ci-dessous :

Liste des propositions [Rafraîchir](#)

ID	Nom	Adresse	Etat	Votes reçus	% Acceptation
0	Entreprise Z	0xd1d3f736c509836e1b95b0224efb0d3b0d6fc1f	En cours	3	66 %
1	Entreprise X	Adresse	Avis	Justification	100 %
		0x7a992758381ebe985e0b60ede95c0284d219d5d6	Acceptation	Creator's vote	
		0x034a611d306f3b1a9813bf867e71ac8c13bb6821	Acceptation	ok	
		0xbd3bc2125c0099d27bf11fcdcbc8e7158a251539	Refus	Not trusted	

Annexe 3 : *Smart Contracts*

Table des matières

1. SafeMath.sol	91
2. BasicToken.sol	92
3. StandardToken.sol	93
4. MomentalyOwned.sol.....	95
5. CongressOwned.sol.....	96
6. MintableToken.sol	101
7. ChabToken.sol.....	102
8. QueueDemande.sol	106

1. SafeMath.sol

```
pragma solidity ^0.4.11;

// Initially from : https://github.com/OpenZeppelin

/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */

library SafeMath {
    function mul(uint256 a, uint256 b) internal constant returns (uint256) {
        uint256 c = a * b;
        assert(a == 0 || c / a == b);
        return c;
    }

    function div(uint256 a, uint256 b) internal constant returns (uint256) {
        //assert(b > 0); // Solidity automatically throws when dividing by 0
        uint256 c = a / b;
        //assert(a == b * c + a % b); // There is no case in which this doesn't hold
        return c;
    }

    function sub(uint256 a, uint256 b) internal constant returns (uint256) {
        assert(b <= a);
        return a - b;
    }

    function add(uint256 a, uint256 b) internal constant returns (uint256) {
        uint256 c = a + b;
        assert(c >= a);
        return c;
    }
}
```

2. BasicToken.sol

```
pragma solidity ^0.4.11;
// Initially from : https://github.com/OpenZeppelin
import './SafeMath.sol';
/**
 * @title Basic token
 * @dev Basic version of StandardToken, with no allowances.
 */
contract BasicToken {
    using SafeMath for uint256;
    uint256 public totalSupply;
    event Transfer(address indexed from, address indexed to, uint256 value);
    mapping(address => uint256) balances;
    /**
     * @dev transfer token for a specified address
     * @param _to The address to transfer to.
     * @param _value The amount to be transferred.
     */
    function transfer(address _to, uint256 _value) returns (bool) {
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(_value);
        Transfer(msg.sender, _to, _value);
        return true;
    }

    /**
     * @dev Gets the balance of the specified address.
     * @param _owner The address to query the the balance of.
     * @return An uint256 representing the amount owned by the passed address.
     */
    function balanceOf(address _owner) constant returns (uint256 balance) {
        return balances[_owner];
    }
}
```

3. StandardToken.sol

```
pragma solidity ^0.4.11;

// Initially from : https://github.com/OpenZeppelin

import './BasicToken.sol';

/**
 * @title Standard ERC20 token
 *
 * @dev Implementation of the basic standard token.
 * @dev https://github.com/ethereum/EIPs/issues/20
 * @dev Based on code by FirstBlood:
 *      https://github.com/Firstbloodio/token/blob/master/smart_contract/FirstBloodToken.sol
 */
contract StandardToken is BasicToken {
    event Approval(address indexed owner, address indexed spender, uint256 value);
    mapping (address => mapping (address => uint256)) allowed;

    /**
     * @dev Transfer tokens from one address to another
     * @param _from address The address which you want to send tokens from
     * @param _to address The address which you want to transfer to
     * @param _value uint256 the amount of tokens to be transferred */
    function transferFrom(address _from, address _to, uint256 _value) returns (bool) {
        var _allowance = allowed[_from][msg.sender];

        /* Check is not needed because sub(_allowance, _value) will already throw if
        this condition is not met */
        require (_value <= _allowance)
        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(_value);
        allowed[_from][msg.sender] = _allowance.sub(_value);
        Transfer(_from, _to, _value);
        return true;
    }

    /*@dev Approve the passed address to spend the specified amount of tokens on
    behalf of msg.sender.

    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint256 _value) returns (bool) {
        // To change the approve amount you first have to reduce the addresses`

```

```

// allowance to zero by calling `approve(_spender, 0)` if it is not
// already 0 to mitigate the race condition described here:
// https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
require((_value == 0) || (allowed[msg.sender][_spender] == 0));
allowed[msg.sender][_spender] = _value;
Approval(msg.sender, _spender, _value);
return true;
}

/* @dev Function to check the amount of tokens that an owner allowed to a
spender.
* @param _owner address The address which owns the funds.
* @param _spender address The address which will spend the funds.
* @return A uint256 specifying the amount of tokens still available for the
spender.
*/

function allowance(address _owner, address _spender) constant returns (uint256
remaining) {
    return allowed[_owner][_spender];
}

/* approve should be called when allowed[_spender] == 0. To increment
* allowed value is better to use this function to avoid 2 calls (and wait
until the first transaction is mined)
* From MonolithDAO Token.sol */

function increaseApproval (address _spender, uint _addedValue)
returns (bool success) {
    allowed[msg.sender][_spender] =
allowed[msg.sender][_spender].add(_addedValue);
    Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    return true;
}

function decreaseApproval (address _spender, uint _subtractedValue)
returns (bool success) {
    uint oldValue = allowed[msg.sender][_spender];
    if (_subtractedValue > oldValue) {
        allowed[msg.sender][_spender] = 0;
    } else {
        allowed[msg.sender][_spender] = oldValue.sub(_subtractedValue);
    }
    Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    return true;
}
}
}

```

4. MomentallyOwned.sol

```
pragma solidity ^0.4.11;
/* Initially from : https://github.com/OpenZeppelin - Changed by : Paris Jimmy
   I changed this code for my needs and decided to rename it "MomentallyOwned"
   In this version the owner can be allowed to keep some rights,
   but he will still lose some special rights after a period of time (called
   here the "Q1 period"). */
import './StandardToken.sol';
contract MomentallyOwned is StandardToken {
    event TransferOwnership(address indexed newOwner);
    address public owner;
    uint public creationTime;//Don't initialize time in compile time, not constant
    uint public constant periodQ1 = 90 days;
    function MomentallyOwned() {
        owner = msg.sender; creationTime = now;
    }
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
    modifier onlyInQ1 {
        require(now <= creationTime.add(periodQ1));
        _;
    }
    modifier onlyAfterQ1 {
        require(now > creationTime.add(periodQ1));
        _;
    }
    function transferOwnership(address newOwner) onlyOwner {
        require(newOwner != address(0));
        owner = newOwner;
        TransferOwnership(newOwner);
    }
    function getTimeLeftInQ1() constant returns (uint256){
        uint256 timePicture = now;
        uint256 endOfQ1 = creationTime.add(periodQ1);
        return timePicture >= endOfQ1 ? 0 : endOfQ1.sub(timePicture);
    }
}
```

5. CongressOwned.sol

```
pragma solidity ^0.4.11;

/*
    Initially from : https://github.com/vitiko/solidity-test-example/blob/master/contracts/Congress.sol
    Changed by : Jimmy Paris
    Changed to be accessible only for a liste of the members
    Members can propose new members who will be accepted or not by votations.
    Note that in this version the creator can add the first members in a inital
    period.
*/

import './MomentalyOwned.sol';

contract CongressOwned is MomentalyOwned {
    /* constants, variables and events */
    // Minimum acceptation + rejection
    uint256 public constant minimumQuorum = 3;
    uint256 public constant debatingPeriod = 7 days;
    // Minimal delay to close a votation
    uint256 public constant majorityMinPourcent = 67;
    // Minimal delay enter 2 proposals from the same sender
    uint256 public constant periodEnterProposal = 7 days;

    Proposal[] public proposals;
    uint256 public numProposals;
    mapping (address => uint256) public timeLastProposal;
    mapping (address => uint256) public memberId;
    Member[] public members;
    event ProposalAdded(uint256 proposalID, address candidate, string
        candidateName);
    event Voted(uint256 proposalID, bool position, address voter, string
        justification);
    event ProposalTallied(uint256 proposalID, uint256 result, uint256 quorum, bool
        active);
    event MembershipChanged(address member, bool isMember);
    struct Proposal {
        uint256 id;
        address candidateAddress;
        string candidateName; //Name of the candidate
        uint256 votingDeadline;
        bool executed;
        bool proposalPassed;
        uint256 numberOfVotes;
        uint256 currentResult;
    }
}
```



```

    Vote[] votes;
    mapping (address => bool) voted;
}
struct Member {
    address member;
    string name;
    uint256 memberSince;
}
struct Vote {
    bool inSupport;
    address voter;
    string justification;
}
/* Modifier that allows only members */
modifier onlyMembers(address addr) {
    require(memberId[addr] != 0);
    _;
}
/* Modifier that allows only not members */
modifier onlyNotMembers(address addr) {
    require(memberId[addr] == 0);
    _;
}
/* First time setup */
function CongressOwned() {
    // It's necessary to add an empty first member (as a sentinel)
    addMember(0, '');
    //And let's add the founder (anyway he can do it later)
    addMember(owner, 'founder');
}

```

```

function getMembersCount() constant returns (uint256){
    return members.length -1;
}

/* Make a new member */
function addElectedMember(address targetMember, string memberName) onlyAfterQ1
onlyNotMembers(targetMember) private {
    uint256 id;
    memberId[targetMember] = members.length;
    id = members.length++;
    members[id] = Member({member:targetMember,memberSince:now,name:memberName});
    MembershipChanged(targetMember, true);
}

/* Make a new "early" member */
function addMember(address targetMember, string memberName) onlyOwner
onlyNotMembers(targetMember) onlyInQ1 {
    uint256 id;
    memberId[targetMember] = members.length;
    id = members.length++;
    members[id] = Member({member:targetMember,memberSince:now,name:memberName});
    MembershipChanged(targetMember, true);
}

function removeMember(address targetMember) onlyOwner onlyInQ1
onlyMembers(targetMember) returns (bool){
    for (uint256 i = memberId[targetMember]; i<members.length-1; i++){
        members[i] = members[i+1];
    }
    memberId[targetMember] = 0;
    balances[targetMember] = 0;
    delete members[members.length-1];
    members.length--;
    MembershipChanged(targetMember, false);
}

function getTime() constant returns (uint256){
    return now;
}

```

```

/* Function to create a new proposal */
function newProposal( address candidateAddress,string candidateName)
  onlyAfterQ1 onlyMembers(msg.sender) onlyNotMembers(candidateAddress) returns
  (uint256 proposalID) {
  //Sender did not make a proposal for a while
  require(now >= timeLastProposal[msg.sender] + periodEnterProposal);
  //Update the time of the last proposal from the sender
  timeLastProposal[msg.sender] = now;
  //Create a new proposal :
  /*Set the id of the new proposal and (after) increase the proposals array*/
  proposalID = proposals.length++;
  Proposal storage p = proposals[proposalID]; //Set the pointer
  p.id = proposalID; //Set the id of this proposal
  p.candidateAddress = candidateAddress;//Set the ETH address of the candidate
  p.candidateName = candidateName; //Set the candidate firm identifier
  p.votingDeadline = now + debatingPeriod;//Set the deadline of this proposal
  p.executed = false; //Set the proposal to unexecuted
  /* Set the result of the proposal to false (unused if not executed) */
  p.proposalPassed = false;
  //Vote for my own proposal :
  Vote storage v = p.votes[p.votes.length++]; //Get a new vote structure
  v.voter = msg.sender; //Set the voter
  v.inSupport = true; //Set the stat of his vote (accepted or rejected)
  v.justification = "Creator's vote"; //Set the justification
  p.voted[msg.sender] = true; //Sender has voted for this proposal
  p.numberOfVotes = 1; //Set the count of votes
  p.currentResult = 1; //Set the count of acceptations
  numProposals = proposalID +1; //Update the number of proposals
  ProposalAdded(proposalID, candidateAddress, candidateName);
  return proposalID;
}

function vote(uint256 proposalID,bool supportsProposal,string
  justificationText) onlyAfterQ1 onlyMembers(msg.sender) returns (uint256
  voteID) {
  Proposal storage p = proposals[proposalID]; //Get the proposal
  require(p.voted[msg.sender] == false); //If has already voted, cancel
  Vote storage v = p.votes[p.votes.length++]; //Get a new vote structure
  v.voter = msg.sender; //Set the voter
  //Set the stat of his vote (accepted or rejected)
  v.inSupport = supportsProposal;
  v.justification = justificationText; //Set the justification
  p.voted[msg.sender] = true; //Set this voter as having voted
}

```

```

    p.numberOfVotes++; // Increase the number of votes
    if (supportsProposal) { // If they support the proposal
        p.currentResult++; // Increase score
    }
    // Create a log of this event
    Voted(proposalID, supportsProposal, msg.sender, justificationText);
    return p.numberOfVotes;
}

function executeProposal(uint256 proposalID) onlyAfterQ1 {
    Proposal storage p = proposals[proposalID];
    require(now >= p.votingDeadline); // Has the voting deadline arrived?
    require(!p.executed); // Has it been already executed or is it being executed?
    require(p.numberOfVotes >= minimumQuorum); // Has a minimum quorum?
    /* If difference between support and opposition is larger than margin */
    if ( p.currentResult * 100 / p.numberOfVotes >= majorityMinPourcent) {
        // Add the member
        addElectedMember(p.candidateAddress,p.candidateName);
        p.proposalPassed = true;
    } else {
        p.proposalPassed = false;
    }
    p.executed = true; //Note the proposal as executed
    ProposalTallied(proposalID, p.currentResult, p.numberOfVotes,
    p.proposalPassed); //Fire event
}

function getVoteFromProposal(uint256 idProposal, uint256 idVote) constant
returns (address, bool, string) {
    Proposal memory p = proposals[idProposal];
    Vote memory v = p.votes[idVote];
    return (v.voter, v.inSupport, v.justification);
}
}

```

6. MintableToken.sol

```
pragma solidity ^0.4.11;

/*
    Initially from : https://github.com/OpenZeppelin
    Changed by : Paris Jimmy

    The Minting is only allowed in the period Q1
    A part of the initial code became useless and been deleted
*/

import './CongressOwned.sol';

contract MintableToken is CongressOwned {

    event Mint(address indexed to, uint256 amount);

    /**
     * @dev Function to mint tokens
     * @param _to The address that will receive the minted tokens.
     * @param _amount The amount of tokens to mint.
     * @return A boolean that indicates if the operation was successful.
     */
    function mint(address _to, uint256 _amount) onlyInQ1 onlyOwner
        onlyMembers(_to) {
        totalSupply = totalSupply.add(_amount);
        balances[_to] = balances[_to].add(_amount);
        Mint(_to, _amount);
        Transfer(0x0, _to, _amount); /* Detail : 0x0 Because we create new tokens in
        the minting process (there is no sender of the transfer in term of loss) */
    }
}
```

7. ChabToken.sol

```
pragma solidity ^0.4.11;
/* Auteur : Jimmy Paris */
Logique métier du projet Chablex (prêt et emprunt)
*/
import './MintableToken.sol';
import './QueueDemande.sol';

contract ChabToken is MintableToken {

    event Demander(address indexed addr, uint value);
    event Preter(address indexed _to, uint256 _value);
    event PreterUnePartie(address indexed _to, uint256 _value);

    //Trois constantes pour les info. du token (standard ERC20)
    string public constant symbol = "CHT";
    string public constant name = "ChabToken";
    uint8 public constant decimals = 18;

    uint256 public constant initialEmprutable = 500; /* Utiliser comme
    maxEmprutable lors de la première demande de prêt */
    uint256 public constant facteurChangeMax = 2; /* Utiliser pour multiplier le
    max actuel et définir le nouveau max */
    uint256 public constant tempsMinChangeMax = 30 days; /* Temps obligatoire
    entre les changements du maxEprutable */
    uint256 public constant minRatioCirculent = 3; /* Fixe le maximum total des
    demandes en cours à 1 / ratio */

    QueueDemandesEnCours demandesEnCours = new QueueDemandesEnCours();

    mapping (address => uint256) public demandes; // Demandes d'emprunts (total)
    mapping (address => uint256) public emprunts; // Emprunts passés (total)
    mapping (address => uint256) public remboursements; // Remboursements ou
    prêts passés (total)
    mapping (address => uint256) private maxEmprutable; // Maximum emprutable
    mapping (address => uint256) public dateChangementMax; // Date du dernier
    changement du Max emprutable

    function getMaxEmprutable(address addr) constant returns (uint){
        if (memberId[addr] == 0) // si ce n'est pas un membre
            return 0;

        return initialEmprutable >= maxEmprutable[addr] ? initialEmprutable :
            maxEmprutable[addr];
    }
}
```

```

}
function getDemande(uint256 pos) constant returns (address, uint256){
    var (demandeur,valeur) = demandesEnCours.get(pos);
    return (demandeur,valeur);
}

function getNbDemandes() constant returns (uint256){
    return demandesEnCours.queueLength();
}

modifier peutDemander(uint256 _value) {
    require(_value >= 1); // La valeur demandée est au moins 1 token
    // Définir le maximum empruntable ou sa valeur initiale
    uint monMaxEmpruntable = getMaxEmpruntable(msg.sender);
    //N'emprunte pas plus de tokens que le max
    require(_value <= monMaxEmpruntable);
    /* Le total des demandes + la demande ne dépasse pas le total des
    remboursements + le max */
    require(demandes[msg.sender].add(_value) <=
    remboursements[msg.sender].add(monMaxEmpruntable));
    // Le total demandé est inférieur à 1/ratio du nombre de tokens total
    require( (_value + demandesEnCours.getTotalValue()) *
    minRatioCirculent < totalSupply);
    _; // Indique où insérer le code de la fonction appelante
}

function demander(uint256 _value) public onlyAfterQ1 onlyMembers(msg.sender)
peutDemander(_value) {
    // Augmente le total demandé
    demandes[msg.sender] = demandes[msg.sender].add(_value);
    // Ajoute la demande à la file d'attente
    demandesEnCours.addRequest(msg.sender, _value);
    Demander(msg.sender, _value);
}

modifier peutPreter(uint256 _value) {
    require(_value > 0); // La valeur prêtée est supérieure à 0 token
    require(balances[msg.sender] >= _value); // Le prêteur possède suffisamment
    /* La valeur du prêt ne dépasse pas le total des demandes en cours, et les
    demandes complétées n'appartiennent pas au prêteur */
    require(demandesEnCours.containMinValueFromOther(_value, msg.sender));
    _;
}

```

```

/* La fonction suivante est utilisée pour faire un prêt à la communauté, cela sera comptabilisé comme un remboursement de n'importe lequel de vos prêts et augmentera votre maxEmprutable. */
function preter(uint256 _value) public onlyAfterQ1 onlyMembers(msg.sender)
    peutPreter(_value){
    // Déduit en amont la balance du prêteur
    balances[msg.sender] = balances[msg.sender].sub(_value);
    uint256 pretRestant = _value; // Initialise le montant qu'il reste à prêter
    while (demandesEnCours.queueLength() > 0 && pretRestant > 0){
        // Récupère la plus vieille demande sans la désempiler (demandeur, valeur)
        var (demandeur,valeur) = (demandesEnCours.copyPopRequest());
        // Si le prêt restant peut recouvrir cette demande intégralement
        if (pretRestant >= valeur){
            demandesEnCours.popRequest();
            // Augmente le total des emprunts du demandeur
            emprunts[demandeur] = emprunts[demandeur].add(valeur);
            // Augmente la balance du demandeur
            balances[demandeur] = balances[demandeur].add(valeur);
            // Augmente le total des remboursements du prêteur
            remboursements[msg.sender] = remboursements[msg.sender].add(valeur);
            // Réduire la somme encore prètable
            pretRestant = pretRestant.sub(valeur);
            Preter(demandeur,valeur);
        }
        else { // Sinon, répondre partiellement à cette demande
            emprunts[demandeur] = emprunts[demandeur].add(pretRestant);
            balances[demandeur] = balances[demandeur].add(pretRestant);
            remboursements[msg.sender] =remboursements[msg.sender].add(pretRestant);
            // Réduit la valeur de cette demande et la laisse dans la file d'attente
            demandesEnCours.replaceInFrontRequest(
                demandeur, valeur.sub(pretRestant)
            );
            PreterUnePartie(demandeur, pretRestant);
            pretRestant = 0; // Arrête la boucle car tout prêté
        }
    }
    //Si le prêteur a remboursé ses emprunts
    if (remboursements[msg.sender] >= emprunts[msg.sender]
        // Et est arrivé à son maximum
        && remboursements[msg.sender] >= getMaxEmprutable(msg.sender)

```



```

// Et que sa dernière augmentation date de au moins 30 jours
&& now >= dateChangementMax[msg.sender].add(tempsMinChangeMax)
{
    dateChangementMax[msg.sender] = now;
    // Double le max empruntable lors du prochain emprunt
    maxEmpruntable[msg.sender] =
    getMaxEmpruntable(msg.sender).mul(facteurChangeMax);

    /* A noter que dans ce cas il peut aussi faire un nouvel emprunt (voir la
    première ligne du modifier peutDemander)*/
}
}

//Seuls les membres peuvent envoyer ou recevoir ce token
function transfer(address _to, uint256 _value) onlyAfterQ1
    onlyMembers(msg.sender) onlyMembers(_to) returns (bool) {
    return super.transfer(_to, _value);
}

function approve(address _spender, uint256 _value) onlyAfterQ1
    onlyMembers(msg.sender) onlyMembers(_spender) returns (bool) {
    return super.approve(_spender, _value);
}

function transferFrom(address _from, address _to, uint256 _value) onlyAfterQ1
    onlyMembers(_from) onlyMembers(_to) returns (bool) {
    return super.transferFrom(_from, _to, _value);
}

function increaseApproval(address _spender, uint _addedValue) onlyAfterQ1
    onlyMembers(msg.sender) onlyMembers(_spender) returns (bool success) {
    return super.increaseApproval(_spender, _addedValue);
}

function decreaseApproval (address _spender, uint _subtractedValue)
    onlyAfterQ1 onlyMembers(msg.sender) onlyMembers(_spender) returns (bool
    success) {
    return super.decreaseApproval(_spender, _subtractedValue);
}
}
}

```

8. QueueDemande.sol

```
pragma solidity ^0.4.2;

/*
  Initially from : https://github.com/chriseth/solidity-examples/blob/master/queue.sol
  Changed by: Jimmy Paris
  Changed to be a queue of Demand (a record of the asker's address and the
  asked value)
*/

contract QueueDemande
{
  struct Demande{
    address demandeur;
    uint256 valeur;
  }

  struct Queue {
    Demande[] data;
    uint256 front;
    uint256 back;
  }

  /// @dev the number of elements stored in the queue.
  function length(Queue storage q) constant internal returns (uint256) {
    return q.back - q.front;
  }

  /// @dev the number of elements this queue can hold
  function capacity(Queue storage q) constant internal returns (uint256) {
    return q.data.length - 1;
  }

  /// @dev push a new element to the back of the queue
  function push(Queue storage q, Demande dem) internal
  {
    if ((q.back + 1) % q.data.length == q.front)
      return; // throw;
    q.data[q.back] = dem;
    q.back = (q.back + 1) % q.data.length;
  }
}
```

```

/// @dev put a new element to the front of the queue
function replaceInFront(Queue storage q, Demande dem) internal
{
    if (q.back == q.front)
        return; // throw;
    q.data[q.front] = dem;
}

/// @dev remove and return the element at the front of the queue
function pop(Queue storage q) internal returns (Demande dem)
{
    if (q.back == q.front)
        return; // throw;
    dem = q.data[q.front];
    delete q.data[q.front];
    q.front = (q.front + 1) % q.data.length;
}

/// @dev copy and return the element at the front of the queue
function copyPop(Queue storage q) internal returns (Demande dem)
{
    if (q.back == q.front)
        return; // throw;
    dem = q.data[q.front];
}

function containMinValueFromOther(Queue storage q, uint256 _minValue, address
    exceptAddr) internal returns (bool){
    uint256 valeurComptee = 0;
    uint256 i = q.front;
    while(i < q.front + length(q) && valeurComptee < _minValue){
        if (exceptAddr == q.data[i].demandeur) return false;
        valeurComptee += q.data[i].valeur;
        i++;
    }
    return valeurComptee >= _minValue;
}

```

```

function getTotalValue(Queue storage q) internal returns (uint256){
    uint256 valeurComptee = 0;
    uint256 i = q.front;
    while(i < q.front + length(q)){
        valeurComptee += q.data[i].valeur;
        i++;
    }
    return valeurComptee;
}

/// @dev remove and return the element at the front of the queue
function get(Queue storage q, uint pos) internal returns (Demande dem)
{
    if (pos >= length(q))
        return; // throw;
    dem = q.data[q.front + pos];
}

contract QueueDemandesEnCours is QueueDemande {
    Queue requests;

    function QueueDemandesEnCours() {
        requests.data.length = 200;
    }

    function addRequest(address demandeur, uint256 valeur) {
        push(requests, Demande(demandeur,valeur));
    }

    function replaceInFrontRequest(address demandeur, uint256 valeur) {
        replaceInFront(requests, Demande(demandeur,valeur));
    }

    function popRequest() returns (address, uint256) {
        var d = pop(requests);
        return (d.demandeur,d.valeur);
    }

    function copyPopRequest() returns (address, uint256) {
        var d = copyPop(requests);
        return (d.demandeur,d.valeur);
    }

    function queueLength() returns (uint256) {
        return length(requests);
    }
}

```

```

function get(uint256 pos) returns (address, uint256) {
    var d = get(requests, pos);
    return (d.demandeur,d.valeur);
}
/* @dev check if a minimum value is asked in the queue, return false if it's
asked by a specific address */
function containMinValueFromOther(uint256 _minValue, address exceptAddr)
returns (bool) {
    return containMinValueFromOther(requests, _minValue, exceptAddr);
}
//Get the total of the asked value in the queue
function getTotalValue() returns (uint256){
    return getTotalValue(requests);
}
}

```