Initiation à la blockchain au travers d'une crypto-monnaie (Ether) : Découverte d'Ethereum

Pour cette mise en pratique, nous allons utilser Geth (Go Ethereum), l'implémentation Go d'Ethereum (https://geth.ethereum.org/). D'autres implémentations d'Ethereum existent mais présentent certains désavantages : l'implémentation C++ est difficile à prendre en main et l'implémentation Python ne possède actuellement pas de version stable.

L'autre outil important ici est le compiler solidity (https://solidity.readthedocs.io/en/v0.4.21/), solc. Celui ci va nous permettre de compiler les fichiers Solidity utilisés pour l'implémentation de smart contracts avec Ethereum.

Pour ce qui est de l'installation de ces outils (Go, Geth et solc), celle ci pourra aisément être réalisée grâce au script `./install.sh`. S'ils ne sont pas installés sur cette machine, lancez donc ce script et vous devriez vous retrouver avec un environnement fonctionnel.

Ici, nous allons utiliser un environnement de test, et par conséquent une blockchain privée dans laquelle nous pourrons créer des utilisateurs, attribuer des ethers, réaliser des transactions, développer et tester des smart contracts de façon gratuite et rapide.

Premiers pas : Création d'un compte et premières transactions

---

1. Création d'un utilisateur

Pour pouvoir intéragir avec Ethereum, l'utilisation d'un compte Ethereum est essentielle. Celui ci possède un solde (un nombre d'ethers, *ether balance*), permet l'envoi de transactions, qu'il s'agisse de transactions monétaires ou de smarts contracts et enfin est contrôlé par une clé privée qui permet de le sécuriser. Il va ici être facile d'en créer un en lançant le script `./peer1/create_account.sh`.

Ceci en faisant appel à la commande*geth* va simplement demander la création d'un nouvel utilisateur ayant pour mot de passe le mot de passe stocké dans le fichier */peer1/ethereum_pwd.txt*. Une fois cet utilisateur créé une adresse va être générée, celle ci correspond à l'identifiant unique de notre utilisateur, stocké dans la blockchain et et qui permettra de l'authentifier lorsqu'il cherchera à intéragir avec la blockchain mais également d'identifier toutes les transactions qu'il aura réalisé (permettant par exemple de connaitre son solde à un moment donné).

2. Configuration

On va maintenant allouer une certaine somme d'Ether à cet utilisateur que nous venons de créer : *100000000000000000000* (bien sûr ceci n'est possible que dans le cas d'une blockchain privée...). Pour cela ouvrez le fichier `genesis.json`, et remplacez le champ *ADDRESS* par l'adresse que vous venez de générer :

```
 "alloc": {
   "ADDRESS": {
     "balance": "10000000000000000000"
   }
 }
```

Le *genesis block* (https://bitcoin.fr/bloc-genesis/) correspond au tout premier bloc d'une blockchain qui permet d'instancier la blockchain et de créer un premier lien à partir duquel pourra se construire l'ensemble de la chaine. Dans le cas réel, ce bloc hardcodé contient des données arbitraires, généralement une certaine somme d'argent à laquelle il sera impossible d'accéder par la suite.

Dans notre cas ce genesis block est créé grâce au fichier `genesis.json` qui correspond en quelque sorte à sa configuration. Celui ci comprend différents champs, notamment le champ *config* qui permet de définir certains paramètres de la blockchain tels que son ID. D'autres champs importants pour nous sont :

- *alloc* qui permet d'allouer un certain nombre d'ethers à une adresse donnée;
- la difficulté qui correspond à la complexité nécessaire à la résolution du "puzzle" permettant de miner le hash d'un bloc donné ; dans notre cas afin que le minage soit très rapide (automatique), cette valeur est définie à 0. Dans un cas réel, afin que celui ci soit maintenu à une vitesse constante, cette difficulté évolue en fonction du nombre de mineurs et de la puissance de calcul de la blockchain;
- le timestamp, moment d'émission du bloc.

3. Lancement

Maintenant que nous avons créé un utilisateur et modifié le fichier de configuration, nous allons accéder à la blockchain. Pour ceci il va falloir dans un premier temps lancer le script `./peer1/init.sh` qui va permettre d'initier la blockchain avec la configuration que nous venons de définir. Maintenant que ceci est réalisé, nous allons pouvoir accéder à la blockchain. Pour ceci il va falloir lancer le script `./peer1/run.sh`.

Nous voici maintenant dans la blockchain, à ce stade, on peut vérifier deux choses :

- que l'utilisateur que nous avons créé est bien dans la blockchain avec la commande `eth.accounts` qui liste l'ensemble des utilisateurs
- que la somme demandée a bien été créditée sur le compte de l'utilisateur avec la commande `eth.getBalance(eth.accounts[0])`

4. Création de deux autres utilisateurs

Nous allons maintenant créer deux nouveaux utilisateurs. Le premier de ces deux utilisateurs va nous permettre de découvrir le fonctionnement des transactions. Pour ce qui est du second, nous allons par la suite le définir comme *mineur* afin d'observer la rétribution de ceux-ci lors du minage de blocs.

- a. Créz deux nouveaux utilisateurs, en répétant deux fois la commande `personal.newAccount()`, il vous sera demandé de définir un mot de passe pour ces utilisateurs. Afin de nous simplifier la vie, nous pouvons définir ce mot de passe comme étant une chaine de caractères vide `""`, pour cela il vous suffit de presser la touche [Entrée].

- b. maintenant que ces deux comptes sont créés, grâce aux commandes utilisées dans la question 3, vérifiez que la création de ces deux comptes est effective et regardez le solde de ces deux nouveaux utilisateurs. Que vaut il ?

5. Lancement d'un mineur

Maintenant que ces deux utilisateurs sont créés, nous allons définir dans notre blockchain un compte mineur qui recevra l'argent perçu par le noeud pour le minage des blocs.

Pour cela il va tout d'abord falloir indiquer au système quelle est l'adresse du noeud mineur, pour ce faire, utilisez la commande `miner.setEtherbase("<ad du troisième user>")`. Vous pouvez vérifier que l'opération a fonctionné en utilisant la commande `eth.coinbase`.

Maintenant que ceci est réalisé, nous allons pouvoir lancer le minage. Normalement, en utilisant une blockchain privée avec ethereum, le minage est réalisé de façon continue, que des transactions soient en attente ou pas. Afin de ne miner que les blocs nécessaires, nous allons lancer le script `loadScript("./js_scripts/mineWhenNeeded.js")` qui vérifie que des transactions sont bien en attente avant de lancer le minage.

Vous pourrez constater qu'au premier lancement de ce script et donc au premier *minage* la ligne *Generating DAG in progress* s'affiche à de nombreuses reprises (environ une minute). Comme expliqué ici (https://github.com/ethereum/wiki/wiki/Ethash-DAG), ceci est dû au fait que pour que le système (consensus) de PoW (Proof of Work) fonctionne il est nécessaire qu'un nouveau client génère une certaine quantité de *travail*.

6. Une première transaction

Maintenant que l'environnement de travail est prêt (instanciation des comptes, lancement du minage) nous allons pouvoir effectuer une première transaction. Avant tout, nous allons observer le nombre de blocs actuellement présents dans la chaine grâce à la commande `eth.blockNumber`. On peut également observer le contenu de ce bloc, et retrouver les champs provenant du fichier `genesis.json` grâce à la commande `eth.getBlock(<numero_de_bloc>)`.

Une fois ceci réalisé, on peut réaliser une premier transaction grâce à la commande `eth.sendTransaction({from:eth.accounts[0], to:eth.accounts[1], value:5000000000})` qui contient trois informations : l'émetteur (premier utilisateur créé), le récepteur (second utilisateur créé) ainsi que la valeur de la transaction. Afin que cette commande soit possible, il sera très certainement nécessaire d'indiquer le mot de passe de l'émetteur, ce qui permettra de l'authentifier et de rendre possible la transaction : `personal.unlockAccount(eth.accounts[0], "pa55w0rd123")`. Ceci montre que sans la clé privée d'un utilisateur, il sera impossible d'interagir avec la blockchain en utilisant simplement son adresse.

On peut constater qu'après cette première transaction le mineur s'est mis en mouvement, en effet celui ci va miner cette transaction dans un nouveau bloc qu'il va ajouter à la blockchain. Une fois ceci effectué, on peut vérifier trois choses : que la transaction a bien été effectuée en regardant à la fois le compte de l'émetteur, du récepteur mais également celui du mineur (user 3). On peut également observer qu'un nouveau bloc a été ajouter en regardant la longueur de la blockchain `eth.blockNumber` mais également en regardant le contenu de ce dernier bloc `eth.getBlock(<NOMBRE>)` (il devrait bien entendu contenir la transaction que nous venons d'effectuer).

Note : Maintenant que nous avons miné notre premier bloc, en regardant les informations de ce bloc, on peut observer quels sont les composants essentiels d'un bloc, nécessaires à son fonctionnement : timestamp, hash, hash du bloc précédents, transactions, etc.

7. Ajout d'un second pair/noeud

On va maintenant chercher à agrandir notre réseau privé en créant un nouveau pair, c'est à dire un nouveau noeud s'ajoutant à celui déjà en fonctionnement. Pour cela il va falloir commencer par lancer le script `./peer2/init.sh`. En ouvrant ce script vous pourrez constater que la commande utilisée est quasiment la même que pour le premier noeud. En effet pour que nous puissions ajouter ce second noeud à notre réeau privée il est nécessaire que son bloc de génèse et sa configuration soient les mêmes que ceux du premier noeud. La seule différence entre ces deux scripts de lancement est le dossie de destination dans lequel nous souhaitons stocker les informations relatives à ce noeud (chaque dossier correspondant à un noeud). On va maintenant pouvoir lancer ce second noeud grâce à la commande `./peer2/run.sh` qui indique simplement un port différent de celui du pair (ainsi que toujours un dossier de stockage différent).
Ce pair lancé, on va vouloir récupérer ses informations pour pouvoir les transmettre au pair 1. Pour cela il suffit de lancer la commande `admin.nodeInfo`. Ce qui nous intéresse ici est la valeur de l'inode de ce noeud qui permet de l'identifier de façon unique. On va donc copier cette valeur et en retournant dans le terminal du noeud 1 taper la commande : `admin.addPeer(<ENODE 2>)`. Si l'ensemble du processus a fonctionné, on devrait donc avoir maintenant deux neouds appareillés au sein de notre blockchain. Pour le vérifier on peut utiliser la commande `admin.peers`, l'identifiant du pair 2 devrait s'afficher. Maintenant retournez dans le terminal 2 et affichez le numéro du bloc courant (ie dernier bloc présent dans la blockchain) grâce à la commande `eth.blockNumber`. Après cela, dans le terminal 1, effectuez une nouvelle transaction de l'utilisateur 1 vers l'utilisateur 2 : `eth.sendTransaction(...)`. Retournez à nouveau dans le terminal 2 et affichez à nouveau la valeur du bloc courant ? Comment l'expliquez vous ?

Note : Comme cela a été dit dans le cours, la blockchain la plus longue est celle sur laquelle le plus de mineur ont travaillé et par conséquent la blockchain la plus récente et la plus sure. Pour cette raison lorsqu'un noeud est confronté à une blockchain plus longue que la sienne, il va la télécharger en remplacement de la sienne.

A la fin de cette partie nous devriez être en capacité, de créer une nouvelle blockchain privée composée de plusieurs noeuds, d'y créer des utilisateurs, d'intéragir avec ces utilisateurs, de lancer un mineur et d'effectuer des transactions.

Deuxième temps : Smart contracts

On va maintenant essayer de mettre en place un smart afin d'en comprendre le fonctionnement.

Avec Ethereum ceci est possible grâce à Solidity, un langage haut niveau conçu pour l'implémentation de contrats. Afin de gagner du temps nous allons partir d'un code existant que nous allons essayer de comprendre et d'améliorer.

Vous pouvez regarder le code de ce contrat contenu dans le fichier `/smart_contract/tirelire.sol`. Ce contrat correspond à une tirelire dans laquelle il est possible de mettre de l'argent (fonction *donner*), d'en retirer de l'argent (*retier*) et d'afficher un message (*hello*). Un contrat est donc constitué de différents éléments, des éléments nécessaires (constructeurs et fonctions) et d'autres éléments qui peuvent s'avérer

utiles (variables). Une fois le contrat instancié, les fonctions et variables seront pour l'utilisateur le seul moyen d'intéragir avec le contrat.

1. Un premier contrat

Pour commencer, on va simplement essayer de compiler et de déployer le code actuel du fichier `tirelire.sol`. Pour cela dans un premier temps nous allons compilé ce fichier au travers de la génération de deux fichiers primordiaux : un fichier *bin* qui correspond au contrat compilé et donc au code qui va être placé dans la blockchain et un fichier *ABI* (Application Binary Interface), qui va nous permettre d'intéragir dans un format compréhensible avec le contenu hexadécimal du fichier *bin*.

Pour cette compilation, deux solutions s'offrent à vous : - si vous êtes sous Ubuntu, utilisez (dans un troisième terminal !) la commande `solc --abi --bin tirelire.sol` ; - si vous êtes sous Debian, le compilateur Solidity n'est pas installé sur votre machine eet par conséquent il va vous être nécessaire d'utiliser votre navigateur Web pour accéder à la page `https://chriseth.github.io/browser-solidity/#version=soljson-latest.js`, il s'agit d'un compilateur en ligne temps réel de solidity. Après être accédé à cette page dans un navigateur, collez le contenu du fichier `tirelire.sol` à la place du code présent sur l'interface. Celui ci devait être automatique compilé et il va maintenant vous être possible de récupérer l'ABI et le code bin générées sous les noms de variables `Bytecode` et `Interface`.

Note : si vous utilisez la dernière version du compilateur, il se pourrait que celle ci ne fonctionne pas. Aussi nous vous conseillons de modifier les paramètres de configuration pour utiliser une version fonctionnelle du compilateur telle que : *0.4.24+commit.e67f0147*.

Une fois ces deux fichiers générés, on va pouvoir passer au déploiement du contrat grâce aux lignes de commandes suivantes (dans le terminal du noeud 1 !) :

```
// Stockage du fichier bin, penser à ajouter 0x pour que celui ci soit bien interprété
// comme du contenu hexadécimal et à mettre des guillements autour !

tirelireBin = '0x' + "<BIN provenant du compilateur solidity!>"

// Simple stockage de l'ABI dans une variable (format JSON)

tirelireAbi = <abi provenant du compilateur solidity>

// Signifie à Geth que cette variable est un ABI : tranformer en ABI le JSON

tirelireInterface = eth.contract(tirelireAbi)

//On débloque l'utilisateur pour pouvoir publier le smart contract

personal.unlockAccount(eth.accounts[0], "pa55w0rd123")

// On publie le smart contract

var tirelireTx = tirelireInterface.new({from: eth.accounts[0],data: tirelireBin,gas: 100

// On récupère le hash

tirelireTxHash = tirelireTx.transactionHash

// Pour finir, on récupère l'adresse du contrat

publishedTirelireAddr = eth.getTransactionReceipt(tirelireTxHash).contractAddress

// il s'agit là de deux informations qui pourront nous être utiles par la suite
```

Maintenant que vous avez réalisé ces différentes commandes, le contrat devrait avoir été miné et il devrait être possible de l'utiliser ! Pour le vérifier, on va essayer d'utiliser les différentes fonctions définies dans le fichier `tirelire.sol`. On va donc commencer par mettre de l'argent dans notre tirelire avec la commande `tirelireTx.donner({from:eth.accounts[0], value:"50000000000000000"})`. Pour vérifier que cela a fonctionner, on peut regarder le solde actuel de cette tirelire avec la commande `eth.getBalance(tirelireTx.address)`. On peut maintenant à nouveau allouer cet argent à l'utilisateur avec la commande `tirelireTx.retirer({from:eth.accounts[0]})`. Enfin il est également possible d'afficher un message de présentation à l'écran avec la classique fonction `tirelireTx.hello()`.

Note : Vous pouvez constater qu'au moment de la publication du smart contract, on a indiqué un attribut nommé *gas*. Le Gas est une unité de mesure correspondant aux ressources en calcul nécessaire à l'accomplissement d'une tâche et en l'occurrence ici au ressources en calcul nécessaire à la réalisation d'une transaction, à l'instanciation d'un smart contract ou à l'interaction avec ce smart contract. Bien évidemment

certaines transactions sont bien plus simples que d'autres et par conséquent le calcul nécessaire à leur mise en place est bien plus léger. Un mineur consacrant son temps, son électricité et ses capacités de calcul à l'exécution de code pour la résolution d'une preuve et la mise en place d'une transaction doit être rétribué et c'est le *Gas* qui permet de déterminer à quelle hauteur celui ci doit l'être. Le Gas est donc la capacité de calcul nécessaire à l'exécution d'un smart contract ou d'une transaction. Plus la demande est complexe, plus le coût est élevé, il est donc possible de définir une *Gas limit*, il s'agit de la quantité de Gas maximale que l'on est prêt à dépenser dans une transaction/un contrat. En effet, en cas d'erreur dans le code ou de bug, la capacité de calcul pourrait être infinie et la définition d'une limite permet de se prémunir contre ce genre de problèmes. Si la limite est atteinte, la transaction est interrompue et annulée. Un complément d'information intéressant sur ce sujet est visible ici (https://masterthecrypto.com/ethereum-what-is-gas-gas-limit-gas-price/)

Pour tester le fonctionnement de ce smart contract sur le second noeud on va avoir besoin de deux choses : - récupérer l'abi du contrat (toujours sur le troisième terminal ou le navigateur) et le convertir en contrat; - récupérer l'adresse du contrat.

```
tirelireAbi = <abi provenant du terminal 2 !>
tirelireInterface = eth.contract(tirelireAbi)
tirelireInterface.at(<adresse du smart contract>).hello()
```

Note : Vous pouvez récupérer l'adresse du smart contract à l'aide de la commande `tirelireTx.address`.

Comme on peut le constater, un contrat est défini par deux choses : sont abi (liste d'interactions possibles) et son adresse et avec cela il est possible d'y accéder depuis n'importe quel endroit dans le réseau.

On peut également constater qu'un smart contract peut (tout comme le compute d'un utilisateur) contenir de l'argent.

Enfin comme tout type de transaction, l'instanciation et l'interaction avec le smart contract coûte de l'argent.

2. Définition de variables

On va maintenant essayer de pousser un peu plus loin le fonctionnement de notre smart contract en y ajoutant des variables.

Pour cela, commencez par copiez le fichier `tirelire.sol` dans un nouveau fichier `tirelire_2.sol` sur lequel vous travaillerez jusqu'à la fin de ce tp (afin de garder une base valide du fichier).

Maintenant, ouvrez le fichier et créez y une nouvelle variable `uint public nbAcces` que vous instancierez à `0` dans le constructeur. Cette fonction va nous permettre de savoir combien de fois on a mis de l'argent dans notre tirelire, il va donc falloir l'incrémenter à chaque appel de la fonction `donner`.

Une fois ces trois étapes passées (définition, instanciation et incrémentation), il va falloir répéter les étapes de la question précédente avec le fichier `tirelire_2.sol` pour pouvoir le mettre en place dans la blockchain.

Une fois le fichier mis en place (abi, bin, contrat), vérifiez que les modifications que vous venez d'effectuer fonctionnent bien. Pour cela, vérifiez tout d'abord la valeur de la variable `uint public nbAcces`, puis déposez de l'argent dans votre tirelire et affichez à nouveau la valeur de *nbAcces*, celle-ci devait avoir été incrémentée.

Cette partie doit vous permettre de constater qu'un smart contract perrmet l'instanciation de variables donc la valeur sera stockées dans la blockchain. Ces variables pourront être affichées mais également modifiées par l'appel de fonctions et de constructeurs du smart contract.

3. Mise en place de conditions

On va maintenant fixer une nouvelle valeur, qui va correspondre au but que l'on veut atteindre, au nombre d'ethers que l'on veut avoir stocké dans notre tirelire avant de pouvoir en retirer de l'argent.

Pour cela, toujours dans le même fichier, on va tout d'abord définir une nouvelle valeur `uint public objectif;` que l'on va instancier à `10000000000000000` dans le constructeur. Une fois ceci réaliser, on va ajouter une nouvelle ligne dans la fonction *retirer* : `assert(this.balance >= objectif)`. Ceci signifie que si la somme présente dans notre épargne n'est pas supérieure égale à notre objectif, il sera impossible de retirer de l'argent.

Après avoir à nouveau compilé et ajouté ce contrat à la blockchain, vérifiez que ce que l'on vient de mettre en place fonctionne : - effectuez un premier virement sur la tirelire `tirelireTx.donner({from:eth.accounts[0], value:"50000000000000000"})` et affichez le solde de celle ci; - essayez de retirer de l'argent de la blockchain, et affichez son solde, que constatez vous ? - effectuez un second virement de la même valeur, et essayez à nouveau de retirer l'argent présent dans la tirelire, que se passe t il cette fois ci ?

La définition de cette variable et de cette condition nous permet de constater que si les conditions d'un accord (quel quelles soient) ne sont pas remplies, celui ci n'ira pas jusqu'à son terme. En l'occurence ici, si l'argent n'est pas suffisant, on ne pourra pas en retirer.

Cette partie montre également qu'une fois une variable définie si aucune fonction ne permet de la modifier, il sera impossible d'interagir avec elle, pour par exemple dans notre cas la dimunier. Cette valeur est donc fixée de façon définitive, et cette condition sera toujours vraie et nécessaire.

La seule solution pour modifier cette valeur reviendrait à créer un nouveau smart contract avec une valeur de départ différente, toutefois ceci n'est qu'une fausse solution, en effet, l'argent mis sur notre première tirelire serait perdu...

4. Sécurisation

Après avoir versé de l'argent à un des autres users du noeud 1 ou 2, par exemple : `eth.sendTransaction({from:eth.accounts[0], to:eth.accounts[1], value:50000000000000000})` (sans argent, ce user ne pourra pas payer le mineur et par conséquent pas agir sur le contrat !), déposez de l'argent sur le contrat avec le user 1 (*eth.accounts[0]*) `tirelireTx.donner({from:eth.accounts[0], value:"10000000000000000"})` et essayez de vider le contenu du smart contract avec l'adresse de l'utilisateur vers lequel vous avez effectué la première transaction : `tirelireTx.retirer({from:eth.accounts[1]})` (une authentification sera peut être nécessaire). Quelle somme d'argent contient maintenant notre tirelire ?

Comme vous devriez pouvoir le constater, un smart contract est ouvert et quiconque possède l'adresse de ce contrat peut y accéder. Ainsi, tout utilisateur pourra interagir avec les fonctions publiques de notre contrat, et comme c'est le cas ici, il pourra donc être facile pour un être malveillant de dérober le contenu de notre

tirelire...

Pour cette raison on va mettre en place un peu de sécurité, seul l'émetteur du smart contract sera en mesure de le modifier, pour cela on crée et instancie un nouvel élément `address owner` à `msg.sender` dans le constructeur, et on ajoute dans la fonction `retirer` l'assert permettant de vérifier que l'adresse correspond bien à l'adresse du créateur du smart contract.

Une fois le contrat déployé, déposez à nouveau de l'argent dans notre tirelire : `tirelireTx.donner({from:eth.accounts[0], value:"10000000000000000"})` puis essayez d'y accéder avec un autre utilisateur, que constatez vous ?

Cette partie visait à montrer une chose évidente, la sécurité est un point primordial et contrôler les droits d'accès à nos smart contracts est un point essentiel.

Question ouverte

Imaginons maintenant que l'on ait un ensemble de routeurs qui reçoivent régulièrement des mises à jour de leur table du routage.

Par exemple, à 10:55:20 le routeur R1 reçoit {rule:"test",params:"parameters",id:"rule_id", time:"10:55:20"}

A noter que chacun des routeurs reçoit des mises à jours qui lui sont propres. C'est à dire que les mises à jours des différents routeurs sont indépendantes.

Pour assurer la sécurité du système, il serait important de vérifier que les règles actuellement déployées par ces routeurs correspondent bien aux règles qui leur ont été envoyées. En effet, un routeur malicieux, ou sous le contrôle d'une entité malveillante, pourrait chercher à modifier ces règles, perturbant ainsi le bon fonctionnement de la distribution des informations.

L'objectif de cette partie est de concevoir un mécanisme basé sur la blockchain (et des smarts contracts) permettant d'assurer la sécurité du système de communication. Pour cela des fonctions, par exemple de Mapping (https://coursetro.com/posts/code/102/Solidity-Mappings-&-Structs-Tutorial) pourront être utilisées.

Différents points devront être pris en compte:

1. Afin de maximiser les performances de la blockchain, il est important de minimiser la quantité de données stockées dans la blockchain, en hashant les données par exemple ou encore en rassemblant l'ensemble des règles en une seule (un seul hash pour l'ensemble). Toutefois, le système devra être en capacité de retrouver la règle non correspondante...
2. La génération d'un nouveau bloc dans la blockchain n'est pas automatique et peut prendre quelques secondes. Ainsi, la version des règles actuellement présente dans les routeurs et celle stockée dans la blockchain peuvent être différentes. Comment prendre en compte ces divergences ? La mise en place d'un système de versioning et l'utilisation du moment d'envoi des règles pourraient être utilisés dans cette solution.
3. L'accès au smart contract/la modification des données contenues dans le smart contract devra être

régulé pour éviter qu'une entité malveillante ne puisse perturber le fonctionnement du système

(pour plus de détails, venir me demander)

Pour aller plus loin

Si vous avez finies les parties précédentes, nous vous proposons des améliorations possibles pour découvrir de nouvelles fonctionnalités de la blockchain :

- Mise en pratique d'autres applications de Smart contract et plus particulièrement d'une gestion d'un processus d'élection transparent et sécurisé grâce au tutoriel suivant : https://solidity.readthedocs.io/en/latest/solidity-by-example.html?fbclid=IwAR1TvfUhoadmSGrg0DECsDZvsyJa3rOcxfsJLObJV0SCryiETg3fxpsD1gg#possible-improvements. Pour cela, vous aurez simplement à copier le code présent sur cette page, à le compiler et à le mettre en place sur la blockchain comme dans la partie précédente.
- Interconnexion de multiples machines pour étendre les capacités de votre réseau, pour ceci vous aurez besoin de différentes idées déjà abordées dans ce TP : utilisation d'un même fichier de génèse au moment de l'*init* et ajout de pair par l'administrateur `admin.addPeer("enode://address@ip:port")`, où *ip* et *port* correspondent aux informations provenant de la machine que l'on cherche à connecter.
- Un autre aspect intéressant pourrait être la protection de ressources sensibles grâce au hashage. Une bonne introduction à ceci est fournit ici : https://medium.com/talo-protocol/how-to-secure-sensitive-data-on-an-ethereum-smart-contract-77f21c2b49f5.