Blockchain et crypto-monnaie
hashons les blocs à la chaîne

Janvier 2018

Table des matières

L'UE projet lance une crypto-monnaie nommée la DuckCoinCoin. Cette "monnaie" sera gérée au moyen d'une blockchain simplifiée inspirée de celle du Bitcoin. Tout sera géré en local sans aucune connexion ou gestion de réseau P2P. C'est dommage car on perd un des intérêts principaux de la blockchain mais la gestion d'un réseau P2P nous entraîne un peu trop loin. Ce document est divisé en plusieurs parties. La dernière partie est un cahier technique détaillé. La présentation est volontairement épurée pour ne pas noyer le lecteur dans un tsunami d'explications. Les choses seront plus détaillées dans la suite du document.

Vous êtes invités, pour bien profiter de ce document, à :

1. Vous détendre,
2. Lire le document jusqu'au bout (éventuellement en plusieurs fois)
3. Répéter le mantra : « c'est moins compliqué que ça en a l'air »

1 Présentation

1.1 Principe d'une blockchain

La blockchain ou chaîne de blocs est une structure de données informatique qui permet de sécuriser des données, supposées être publiques, contre toute tentative de modification ou altération. Pour cela on utilise un algorithme de hash code qui permet d'associer à une donnée une valeur numérique unique. Si H est la fonction qui implémente un de ces algorithmes, et x une donnée quelconque, on a :

— $H(x)$ donne toujours la même valeur pour le même x
— $H(x) \neq H(x')$ même si la différence entre x et x' est minime.
— connaissant $H(x)$ est très difficile de retrouver x.

L'algorithme de hash que nous utiliserons est le SHA256 (nombres de 256 bits). Nous allons manipuler son résultat sous forme d'une chaîne de caractères hexadécimaux. La longueur de la chaîne est fixe (32 octets, donc caractères).

La blockchain est constituée de blocs chaînés entre eux sous forme de liste. Jusque là ça reste une SD très classique en informatique. Pour éviter qu'un bloc disparaisse sans laisser de trace, on va ajouter (signer) chaque bloc avec le hash code de son contenu. Et pour qu'il laisse une trace en cas de disparition on va inclure dans le contenu du bloc le hash code de son prédécesseur. Et donc, le hash code du bloc va intégrer le hash code du bloc précédent dans la chaîne.

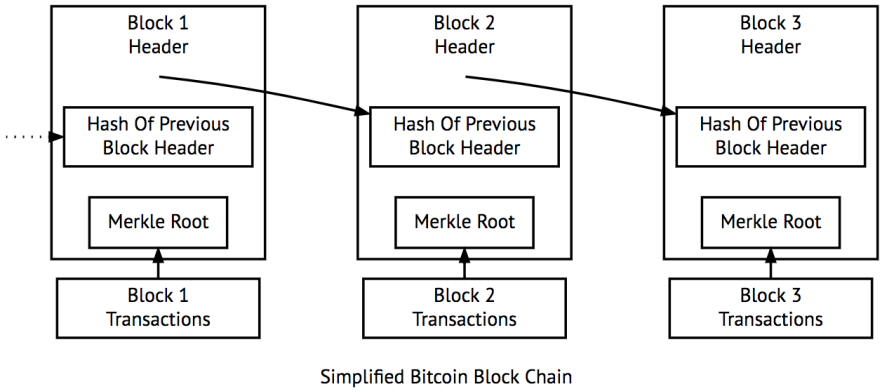


FIGURE 1 – La blockchain

Que se passe-t-il si on veut supprimer un bloc quelque part au milieu de la chaîne ? De toute évidence on perturbe la continuité des hash codes. Il faut donc recalculer le hash de tous les blocs suivant celui qui disparaît et refaire le chaînage. La sécurité de la chaîne tient ici à la quantité de travail à faire pour altérer la chaîne.

Même sur un grand nombre de block, on peut imaginer pouvoir le faire avec un gros PC. Ça reste jouable me direz-vous. On va donc pimenter le tout avec la "difficulté". Pour faire simple, disons que la difficulté est un critère que doit remplir le hash code et qui assure qu'il n'est pas trop facile à calculer même avec une machine performante. Dans la pratique (Bitcoin) la difficulté est adaptée à l'évolution de la puissance des machines. Elle est choisie pour qu'il faille 10mn pour la calculer. A itérer autant de fois qu'il y a de blocs à recalculer pour altérer la chaîne sans que cela ne se voit. Comme un nouveau bloc arrive toutes les 10mn, il faudrait recalculer l'ensemble de la chaîne en 10mn pour que cela passe inaperçu (ou presque car il y a aussi d'autres mécanismes dans un univers peer2peer où il y a autant de copies de la chaîne que de noeuds dans le réseau).

Nous allons simplifier ce mécanisme en choisissant une difficulté représentée par un entier d, et en demandant que le hash du bloc commence par d zéros dans sa représentation en ascii hexadécimal. Et si cela n'est pas le cas ?

1.1.1 Le minage

On introduit dans le contenu de chaque bloc un entier initialisé à zéro et appelé "nonce". Si le hash du bloc (incluant la nonce) ne satisfait pas le critère, on incrémente la nonce et on recommence le calcul du hash. On répète ce processus jusqu'à ce que le hash satisfasse le critère (la difficulté). Ce calcul est appelé «minage» [mining].

1.1.2 Contenu des blocs

Que met-on dans les blocs ? Ce que l'on veut en fait. Le Bitcoin gère des transactions d'argent (crypto-monnaie [crypto-currency]). Nous allons imiter ce principe bien qu'il soit annexe dans ce sujet. Donc il y aura des transactions de Duckcoincoin (Dcc) du genre «Astérix envoie 10 Dcc à Obélix». Il y a un nombre variable de transactions dans un bloc qui est normalement (Bitcoin) dépendant de la taille maximale du bloc et de celle des transactions. Nous allons fixer arbitrairement le nombre maximal de transactions (constante du programme). De plus, le bloc contient la date de création (timestamp).

Le premier bloc de la chaîne est particulier. Il ne peut pas intégrer le hash de son prédécesseur car il n'a pas de prédécesseur. On l'appelle bloc «génésis». Le hash de son prédécesseur est zéro, sa nonce reste à zéro, il a une unique transaction réduite à la chaîne « Genesis » [1]. Par contre il a bien un hash. Il n'est pas nécessaire de le miner.

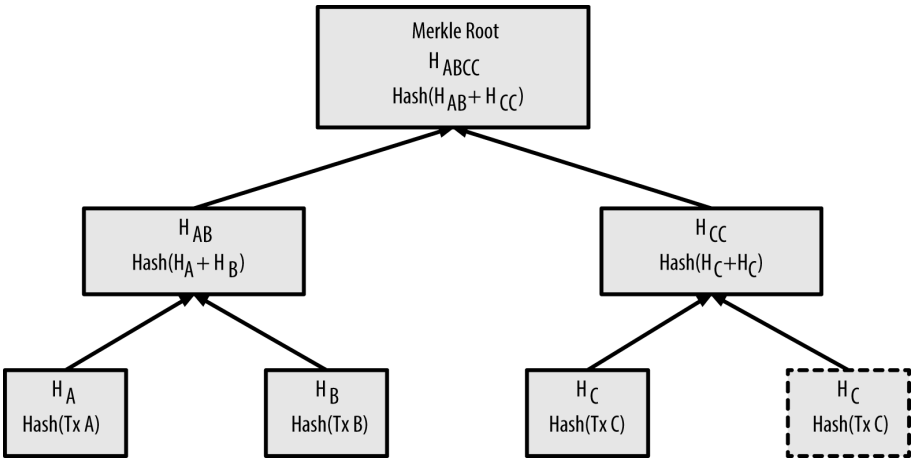
Que se passe-t-il si on veut altérer une transaction d'un bloc sans toucher aux hash de ce bloc ?

1.1.3 Arbre de Merkle [Merkle tree]

Si on altère une transaction d'un bloc sans toucher aux hash de ce bloc, le hash du bloc va quand même être modifier puisque qu'il prend en compte le contenu de tout le bloc, et la liste des transactions fait partie du contenu (pour notre sujet car le Bitcoin est légèrement différent).

Pour corser le tout on va calculer l'arbre de Merkle des hash des transactions :

1. supposons que nous ayons quatre transactions : $t1, t2, t3, t4$,
2. on calcule les hash de chaque transaction. On obtient $h1, h2, h3, h4 = H(t1), H(t2), H(t3), H(t4)$
3. puis on calcule le hash de $h1$ concaténé avec $h2$ et le hash de $h3$ concaténé avec $h4$. Soit $h1.2$ et $h3.4$ les résultats,
4. puis on calcule $H(h1.2$ concaténé $h3.4)$, soit $h1.2.3.4$ le résultat.



Le résultat final $h1.2.3.4$ est le racine de l'arbre [Merkle tree root]. C'est un hash code. Ce hash sera inclus dans le contenu du bloc.

Si le nombre de transactions (ou de hash intermédiaires) est impair, on duplique simplement la dernière transaction (ou hash).

1. Pour reprendre le nom donné à ce block par Satoshi Nakamoto (聡中本) l'inventeur du Bitcoin.

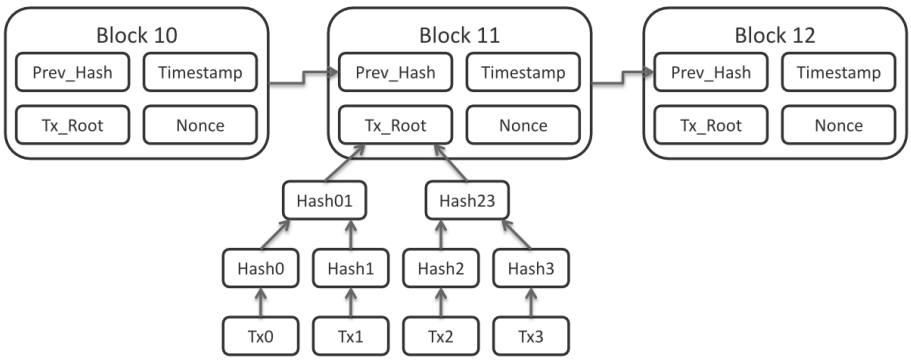


FIGURE 2 – Merkle tree

Connaissant les transactions qui sont écrites en clair on peut recalculer la racine de l'arbre et vérifier que c'est bien le même hash root que celui qui apparaît dans le bloc, et que donc, aucune transaction n'a été altérée

1.1.4 Sécurité de la blockchain

Quelques attaques (dans la cas de notre blockchain simplifiée) :

— Supprimer ou modifier une transaction : Cela modifie le merkleroot et donc le sha du block et provoque une inconsistance de la chaîne.

— Rejouer une transaction déjà présente. Normalement le timestamp et le numéro de tx doit empêcher ça car il sont dans le hash de la signature. Il faudrait modifier ces données. De plus cela modifie le merkle tree. Le test de hash root le détecte.

— Agir sur les blocks. Supprimer un block pour faire disparaître une transaction. Cela supprime aussi les autres transactions. Et le chaînage de hash est perturbé. Il faut tout recalculer.

1.1.5 Les adresses et les transactions

Pour créer une transaction, c'est à dire pour envoyer de l'argent, il faut avoir une adresse. Les adresses sont liées à deux clefs de cryptographie : une clef publique et une clef secrète. Comme son nom l'indique, la clef publique est faîte pour être diffusée. Elle sert à vérifier les signatures. La clef secrète ne doit être connue que de son propriétaire. Elle sert à signer les transactions. Cette partie sera gérée à partir de packages Java fournis. Chaque transaction intègre, de plus, une date (timestamp).

Dans un soucis de simplification on ne traitera pas des wallets, transactions loquées, ni de tout ce qui concerne le réseau.

1.1.6 Vérification

L'intégrité de la blockchain peut être vérifiée à tout moment.

1. On vérifie que la chaîne commence bien par le bloc génésis et que le chaînage des hash est valide, et que le hash du bloc est bien celui annoncé (vérification 1).
2. On vérifie pour chaque bloc que le hash Merkle root correspond bien aux transactions de ce bloc (vérification 2).
3. On vérifie quand on ajoute une nouvelle transaction que la signature est valide (voir Niveau 2).

1.1.7 Qui veut la peau de la blockchain ?

Bien sûr notre blockchain n'est pas inaltérable, ni celle du Bitcoin d'ailleurs. Puisque que la nôtre est centralisée on peut faire ce que l'on veut avec ce qui n'est pas le cas de celle du Bitcoin qui est répartie et soumise au principe du consensus (mais c'est une autre histoire).

Donc voila le programme : deux programmes, un en Java et un en C. Ils sont indépendants. Le programme Java va implémenter la blockchain, les blocks, le minage, les hash (avec un package fourni), un générateur aléatoire de transactions réduites, dans un premier temps, à leur plus simple expressions : des chaînes de caractères, et des systèmes de contrôle d'intégrité.

Dans un deuxième temps le programme (Java toujours) va implémenter une version améliorée de cette blockchain avec des transactions plus sophistiquées intégrant les adresses de l'émetteur et du destinataire ainsi qu'un montant. On intégrera les signatures. Tout ceci avec un package *ad hoc*.

En C on va faire le même programme de base avec en plus un *cheater*, c'est à dire un programme qui altère un bloc et recalcule les hash de tous les blocs impactés le plus vite possible. Le cheater devra être développé dans cet esprit tout en gardant un code propre et lisible.

En fait il va y avoir deux étapes de programmation :

1. Faire les deux programmes en Java et C avec des transactions simplifiées (chaînes de caractères).
2. Faire ensuite les transactions niveau 2 en Java, ou le lecteur JSON en C.

Avant de vous poser la question de savoir ce que vous choisissez dans le niveau 2, il faut être arrivé au bout du niveau 1.

2 Travail à faire

Le code à produire est en deux parties indépendantes :
— un code Java,
— un code C

Les deux niveaux :
— Niveau 1 : développer la même gestion de la blockchain dans les deux langages (faire deux fois la même chose ou presque),
— Niveau 2 : développer les modules complémentaires en Java et en C. Les deux langages servant à implémenter des choses différentes

2.1 Niveau 1 : les frères presque jumeaux

2.1.1 Cahier des charges JAVA

La programmation par objets se prête particulièrement à la programmation d'une blockchain. Java fournit des structures de données prédéfinies comme des *arraylist* génériques qui sont très efficaces. Les packages et classes fournis pour la lecture/écriture au format JSON et le calcul du SHA256 sont *plug and play*. Votre programme Java devrait se passer sans trop de soucis et aller assez vite.

1. Interface de gestion : pas de véritable contrainte. Malgré tout cette interface devra permettre de générer une blockchain en choisissant le nombre de blocs, la difficulté, le nombre maximal de transactions par bloc, sauver ou lire dans un fichier au format JSON, vérifier l'intégrité de la blockchain, sortir du programme, afficher la blockchain ou un bloc d'un numéro donné. Cette interaction devra être légère et efficace.
2. Structure de données pour la blockchain centralisée

3. Structure de données pour les blocs
4. Structure de données pour les listes de transactions (chaînes de caractères dans l'étape 1)
5. Calcul du hash256 d'un bloc
6. Minage des blocs
7. Test de validité 1 et 2 (présence du Génésis, les ash des blocks sont cohérents, le chaînage, aussi, les racines de Merkle aussi)
8. Calcul de l'arbre de Merkle d'une liste de transactions
9. Sauvegarde/lecture fichier JSON (en utilisant le package Google)
10. Générateur aléatoire de transactions simplifiés
11. Générateur aléatoire de blocks
12. Une classe main qui gère l'ensemble

2.1.2 Cahier des charges C

En C, vous l'avez déjà compris, il faut tout écrire. Le développement des structures de données pour la blockchain sera basé sur des *struct* et des pointeurs. La librairie de lecture d'un format JSON n'est pas vraiment *plug ang play*, on la laisse donc à l'étape 2. La fonction de calcul du SHA256 est moins contraignante mais reste sensible. L'avantage du C est la rapidité.

1. Structure de données pour la blockchain centralisée
2. Structure de données pour les blocs
3. Structure de données pour les listes de transactions simplifiées (chaînes de caractères)
4. Calcul du hash256 d'un bloc
5. Minage des blocs
6. Test de validité 1 et 2 (présence du Génésis, les ash des blocks sont cohérents, le chaînage aussi, les racines de Merkle aussi)
7. Calcul de l'arbre de Merkle d'une liste de transactions simplifiées
8. Générateur aléatoire de transactions simplifiées
9. Générateur aléatoire de blocks
10. Un main qui gère l'ensemble. Pas d'interface demandé, mais il faudra prévoir de générer une blockchain avec choix de difficulté, nombre de blocs et de transactions, mise en action ou non du cheater avec dans ce cas le choix d'un numéro de bloc et, en option, de transaction à supprimer. Ces paramètres pourront être passés en ligne de commande (recommandé).
11. **un cheater de block qui supprime un block dont le numéro est passé en paramètre et recalcule les hash de tous les blocks suivant pour effacer les traces avec affichage du temps de calcul final.**
12. **un cheater de transaction qui supprime une transaction dont le numéro est passé en paramètre ainsi que celui de son block et recalcule les hash de l'arbre de Merkle et de tous les blocks suivant pour effacer les traces avec affichage du temps de calcul final.**

    Remarque : Les timestamp sont un autre élément de sécurité permettant de détecter des anomalie dans la blockchain ou dans la liste des transactions. On ne les vérifiera pas.

2.2 Niveau 2 : de nouveaux objectifs pour tous les goûts

2.2.1 Cahier des charges Java

1. Définir une classe de transactions avec émetteur, destinataire, montant, signature numérique

2. Générateur de n adresses (avec le package Google), n étant un paramètre.
3. Signature numérique avec les adresses
4. Vérification de la signature d'une transaction
5. Générateur aléatoire de transaction niveau 2 choisissant au hasard l'adresse de l'émetteur et du destinataire dans la liste des adresses, ainsi qu'un montant.
6. Intégration dans la blockchain

2.2.2 Cahier des charges C

1. Lecture du fichier de blockchain JSON (celui créé par le programme Java au niveau 1) et chargement dans la structure de données C.
2. Intégration de ce sous-programme dans le code C du niveau 1 et ajouter un paramètre de ligne de commande pour gérer la possibilité de lire la blockchain dans un fichier JSON plutôt que de la calculer.
3. Sauver la blockchain modifiée par le cheater dans un fichier JSON (à la main).
4. Charger la blockchain précédente dans Java (test de validité)

3 Evaluation

Les programmes sont organisés en modules et en niveaux. On peut organiser les modules en un graphe qui fait apparaître les niveaux, dépendances et relations.

Le barème détaillé est téléchargeable sur Moodle. Les points du programme rendu seront attribués à l'équipe dans son ensemble. Vous êtes libres de vous les partager comme bon vous semble. Par défaut, les points seront divisés par le nombre de membres de l'équipe sauf décision contraire du tuteur.

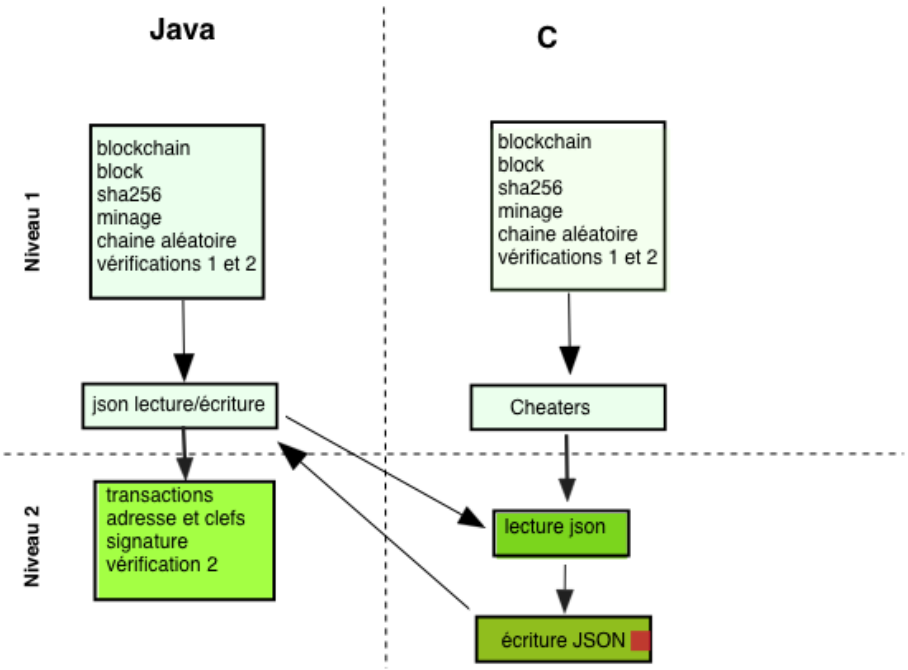


FIGURE 3 – Graphe du projet

4 Cahier technique

Cette partie apporte des précision sur la structure des différents objets. Une partie des documents, les programmes et liens cités sont disponible sur Moodle. Moodle sera par ailleurs le medium privilégié de communication de documents, de news, de mise à jour etc.

4.1 Spécifications générales

4.1.1 Structure de la blockchain

— Champs difficulté : entier = critère de difficulté pour le minage
— Champs : nombre de blocks : entier = nombre de blocks de la chaîne (en comptant le génésis).
— Champs liste des blocks : liste de blocks

4.1.2 Structure des blocks

— champs : index : entier = numéro du block dans la chaîne (le génésis est numéroté zéro)
— champs : timestamp : chaîne de caractères = la date au moment de la création
— champs : hash du block précédent : chaîne de caractères = hash du block précédent dans la chaîne
— champs : nombre de transactions : entier
— champs : liste des transactions
— champs : hash root de l'arbre de Merkle des transactions : chaîne de caractères
— champs : hash du block courant : chaîne de caractères
— champs : nonce : entier

4.1.3 Structure des transactions

Etape 1 :
Les chaînes de caractères représentant les transactions simplifiées seront composées de la concaténation de la chaîne fixe "Source-Destination :" et d'un nombre aléatoire compris entre 1 et une constante MAX_VALUE.

Etape 2 :
— champs : index : entier = numéro de la transaction dans la liste
— champs : timestamp : chaîne de caractères = la date au moment de la création de la transaction
— champs : émetteur : chaîne de caractères = adresse hexadécimale de l'émetteur
— champs : destinataire : chaîne de caractères = adresse hexadécimale du destinataire
— champs : montant : entier = montant de la transaction
— champs : signature émetteur : chaîne de caractère

4.2 SHA256

L'algorithme de hash que nous allons utiliser est celui qu'utilise la blockchain du Bitcoin, le sha256. Un programme réalisant ce calcul est fourni en Java comme en C. Il reçoit une chaîne de caractères et renvoie une chaîne contenant le hash en caractères hexadécimaux.

4.3 Le format JSON

Le format JSON vient du Javascript et est un format d'échange de données alternatif au XML. Wikipédia en parle très bien, je vous invite à lire la page :

`https://fr.wikipedia.org/wiki/JavaScript_Object_Notation`

En fait, il n'y a pas grand chose à savoir sur ce format. Il s'agit juste de l'utiliser. Des packages et librairies sont fournies pour cela.

4.4 Les packages Java

4.4.1 GSON : le package de lecture/écriture JSON de Google

`https://github.com/google/gson`

Le package $gson - 2.2.2.jar$ est à télécharger sur Moodle.

4.4.2 Classe sha256

La classe est à télécharger sur Moodle.

4.4.3 Le package google pour les adresses et les clefs

bitcoinj-core-0.14.5-bundled.jar sur Moodle

4.5 Programmes C fournis

4.5.1 sha256

Le calcul du sha256 en C se fera en utilisant le code fourni : sha256.h, sha256.c, sha256_utils.h, sha256_utils.c par Brat Conte et adapté aux besoins du projet.

Vous n'avez besoin d'appeler que la procédure :

```
void sha256ofString(BYTE * str,char hashRes[SHA256_BLOCK_SIZE*2 + 1])
```

où str est la chaîne de caractères dont on veut calculer le hash (entrée), et hashRes est la chaîne de 32 caractères représentant le hash sous sa forme de caractères hexadécimaux (sortie).

Les fichiers sha256.c et sha256.h sont nécessaires mais doivent être considérés comme des boîtes noires.

Les types BYTE et SHA256_BLOCK_SIZE sont déclarés dans sha256.h

Pour le timestamp en C, il y a dans *time.h* des fonctions pour récupérer la date et l'heure qu'on peut utiliser pour créer une fonction *getTimeStamp* :

```
char * getTimeStamp(){
    time_t ltime;
    time(&ltime);
    return ctime(&ltime);
}
```

Cette fonction renvoie une chaîne de caractères du type "Fri Jan 19 13 :40 :09 2018"

4.5.2 Lecture d'un fichier JSON

Nous allons utiliser les programmes *json-parser-master* créés par divers auteurs (cf le fichier AUTHORS pour plus de détails). Comme pour le sha256 nous allons considérer ce code comme une boîte noire à l'exception du fichier *test_json.c* de Mirko Pasqualetti que nous allons modifier (le fichier pas Mirko Pasqualetti).

Nous n'avons besoin que des fichiers : *json.c, json.h*, de la librairie *libjsonparser.a* et bien sûr du fichier *test_json.c* (voir Moodle).

4.6 Constantes des programmes et performances attendues

La difficulté sera de 4 pour les tests. Le nombre de transactions doit pouvoir être de 1 à 10 (nombre aléatoire pour chaque bloc). Le nombre de blocs n'exèdera pas de 10 pour les tests. Cependant, programme doit pouvoir supporter la génération de 100 blocks. A titre indicatif le temps CPU moyen est d'environ 80 secondes sur une machine ordinaire (laptop i5 bi-core 2.7Ghz) pour cheater 100 blocs en supprimant le bloc 3 et en recalculant tout le reste de la chaîne.

Les structures de données en C seront exclusivement dynamiques. Il est conseillé de programmer les listes de blocks et des transactions avec des listes doublement chaînées. On peut utiliser des pointeurs void qui pourront recevoir indifféremment des pointeurs de transaction ou des pointeurs de block.

Il est conseillé de faire des économies de mémoire (n'allouez que ce qui est nécessaire, faîtes des free de ce qui devient inutile. Attention toutefois à ne pas libérer une zone mémoire utilisée par ailleurs par une de vos structures de données).