Blockchain appliquée à un processus éléctoral

Rapport de Projet

Ce rapport présente le sujet du projet de manière globale en étudiant l'utilité des méthodes et fonctions appliquées et non leur fonctionnement avec précision sauf cas particulier. Il a ainsi pour but d'articuler les différentes parties de l'énoncé afin qu'elles représentent un tout expliquant clairement le principe de blockchain appliqué à un processus éléctoral et son but.

L'objectif de ce projet est de protéger les données concernant un processus éléctoral : les électeurs et leur vote ; et de ne les rendre accessible qu'à certaines personnes. Pour cela, nous pouvons distinguer trois parties :

Code Couleur : → Structures

→ Fonctions

→ Nom de fichier

Cryptage des Données

> La **cryptographie** est une des disciplines de la cryptologie s'attachant à protéger des messages (assurant confidentialité, authenticité et intégrité) en s'aidant souvent de *secrets* ou *clés*.
>
> Wikipedia

Si le chiffrement se basant sur une même clé pour chiffrer et déchiffrer un message semble plutôt sûr, la transmission de celle-ci à son correspondant n'est pas infaillible. Pour résoudre ce problème, la cryptographie asymétrique a été mise au point dans les années 1970.
Elle se base sur le principe de deux clés : -une publique, permettant le chiffrement ;
-une privée, permettant le déchiffrement.
C'est celle-ci que nous utiliserons et plus particulièrement le chiffrement RSA qui se base sur les congruences sur les entiers et le petit théorème de Fermat pour générer ces clés. Ainsi nous implémentons la fonction modpow(long a, long b, long n) permettant de calculer a^b mod n pour l'utilisation de ce théorème.

Le problème principal est que son application nécessite des nombres premiers. Nous allons donc voir comment générer des nombres premiers aléatoirement choisis dans une fourchette de taille d'entiers.

Génération de nombres premiers aléatoires

Afin de démontrer la primalité d'un nombre, nous utiliserons le test de primalité de Miller-Rabin qui étant donné un nombre entier, indique qu'il est soit, de façon certaine, un nombre composé, soit qu'il est probablement premier avec une probabilité d'erreur de log(k/4), k étant le nombre de test effectués sur un même nombre. En utilisant les fonctions données, nous créons random_prime_number(int low_size, int up_size, int k) permettant de générer un entier de taille comprise entre les variables *low_size* et *up_size* premier aléatoirement. Il nous reste donc à savoir à quoi va servir cet entier.

Génération d'une paire (clé publique, clé secrète)

A l'aide de la fonction précédente, nous générerons deux nombres premiers *p* et *q* afin d'obtenir quatre autres entiers :

$n = p * q$
$t = (p-1) * (q-1)$
s est généré aléatoirement jusqu'à trouver PGCD(*s*,*t*) = 1
u est obtenu par l'équation $s * u \bmod t = 1$

Ces quatres valeurs sont obtenus via la fonction generate_key_values(long p, long q, long* n, long *s) et vont servir à former les clés de cryptage formées ainsi :
Clé publique = (s,n) / Clé secrète = (u,n) .

Utilisation des clés

Nous n'allons pas expliquer ici le principe de chiffrement et déchiffrement de messages puisque l'énoncé le fait pour nous, en revanche, nous allons étudier les fonctions encrypt(char* chaine, long s, long n) et decrypt(long* crypted, int size, long u, long n) qui appliquent ce principe. La première applique le modpow() sur le message à crypter en utilisant la clé publique (donnée dans les paramètres) caractère par caractère. La seconde fonction applique sur une chaîne le modpow() du message crypté à l'aide cette fois-ci de la clé secrète. Ces deux fonctions qui sont la base de toute future donnée cryptée permettent donc de retrouver le message donné à encrypt() à la sortie de decrypt().

Bien que nous ayons désormais la capacité de crypter des données, il faut maintenant l'apliquer au projet. C'est pour cela que nous allons représenter nos citoyens, candidats et déclarations de vote sous la forme de données à crypter.

Représentation des Données

Dans cette partie, nous allons créer multiples structures représentant notre modèle éléctoral. Tout d'abord, les citoyens et candidats seront exprimés par la structure **Key** prenant deux entiers (s,n) pour une clé publique et (u,n) pour une clé secrète. Elle pourra être transformé en chaîne de caractères afin d'être stocké dans un fichier grâce à la fonction key_to_str(Key* key). Nous avons ensuite besoin de transformer les votes de chaque personne en données. Pour cela, nous utilisons la structure **Signature** prenant le vote crypté d'un citoyen. Pour unifier ces deux structures et donc créer une déclaration de vote valide nous créons une troisième structure **Protected** qui regroupe la clé d'un citoyen, son vote et son vote crypté en **Signature**. Tout comme la clé ces deux structures peuvent être mises en chaîne de charactères via les fonctions signature_to_str(Signature *sgn) et protected_to_str(Protected* pr).

Maintenant que les bases ont été posés, il faut faire vivre les fondations. Nous allons utiliser toutes ces fonctions et structures pour réaliser un véritable vote en créant la fonction generate_random_data(int nv, int nc), *nv* étant le nombre de citoyens et *nc* celui de candidats. Celle-ci permet de générer dans des fichiers txt les clés citoyennes et candidates, de définir les votes de chaque citoyens et de les écrire eux-aussi dans un fichier txt. Pour cela, on vérifie bien que le nombre de candidats ne soit pas supérieur à celui des citoyens et nous utilisons l'aléatoire basique via la fonction rand() pour choisir les candidats parmi les citoyens et le vote des ces derniers. Une fois la fonction exécutée, nous nous retrouvons avec trois fichier : keys.txt / candidates.txt / declarations.txt contenant toutes nos données **écrites**. Le nouveau problème se posant à nous est qu'il faut manipuler ces données ... et en masse.

C'est pour cela que les structures **CellKey** et **CellProtected** sont créées : l'une contient une liste chaînée de clés citoyennes et l'autre une liste chaînée de déclarations de votes. Ces structures permettent la création des fonctions read_public_keys(FILE* f) et read_protected(FILE* f) qui vont stocker les données des fichiers mis en paramètres dans une cellule de clés si ce sont des clés et dans une cellule de déclarations si ce sont des déclarations. Pour finir, il ne nous reste plus qu'à comptabiliser le nombre de voix de chaque candidat afin de déterminer le gagnant de l'élection.

La structure HashCell est donc créée afin de stocker les clés citoyennes et si elle est candidate son nombre de votes, sinon un booléen indiquant si le citoyen a déjà voté (0 pour non / 1 pour oui). La structure **HashCell** va elle-même être stocké dans un tableau situé dans la structure **HashTable** servant à ce stockage.

Quelques Problèmes

A partir d'ici, beaucoup de problèmes de compréhension ont empêchés la complétion totale du projet principalement aux exercices 6, 7 et 9. Ils seront énumérés petit à petit dans la suite de ce rapport.

La fonction hash_function(Key* key, int size) retourne la position d'un élément dans la table de hachage, or nous ne savons pas comment sont ajoutés les clés dans une table. J'ai supposé que nous devions imaginer un calcul afin de trouver un indice pour gérer le probing linéaire de la question suivante. J'ai donc simplement pris l'addition des deux valeurs composant la clé modulo la taille de la table. N'étant pas sûr de ce calcul les clés ont été ajoutés une par une dans le tableau et sont retrouvés via le parcours de ce même tableau.

Le manque de données ont rendus certaines fonctions intestables, principalement dans la suite mais pour la fonction compute_winner(CellProtected* decl, CellKey* candidates, CellKey* votes, int sizeC, int sizeV) aussi. Étant donné que je n'ai pas fini le projet, je n'ai pas écrit de fonction d'actualisation de vote permettant de remplir pour les candidats l'entier *val* dans la structure hashcell ce qui empêche la vérification du gagnant de l'élection.

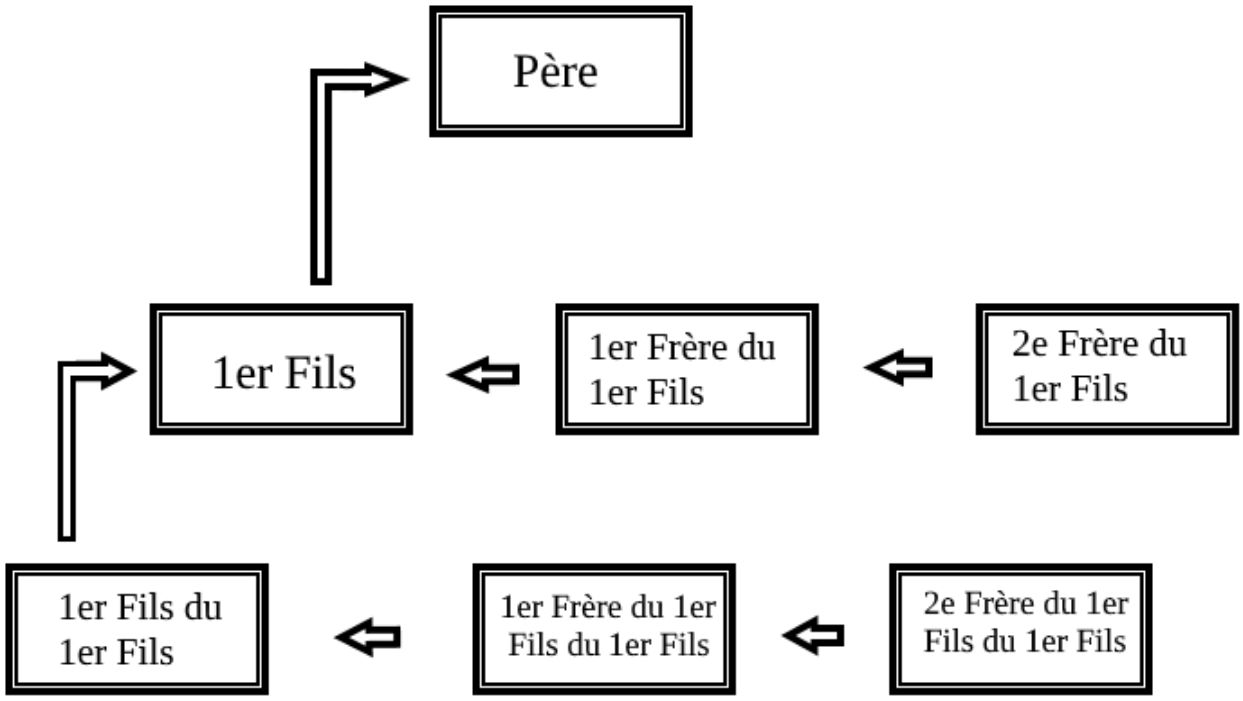
Création du Blockchain et son Application

Une fois toutes nos structures créées, il faut arriver à les regrouper pour les sécuriser puisque même si les votes sont cryptés, les clés pour les déchiffrer sont aussi donné. Une méthode répandue mondialement est celle du blockchain qui ne permet la lecture d'un bloc de données que si l'on possède l'accès au bloc le précédant. Ainsi, la structure **Block** fait son apparition contenant la clé publique de son créateur, une liste de déclarations de vote, la valeur hachée du bloc et du bloc précédent ainsi qu'une preuve de travail.

Pour hacher les blocs nous utilisons une fonction de cryptologie donnée par l'algorithme SHA256. Survient alors mon second problème, la création de blocs valides ne fonctionnent pas car la valeur hachée d'un bloc testé ne passe jamais le test de validité. Ce qui a rendu pour la suite du projet, les valeurs hachés inutilisables.

Structure Arbre

Ces blocs sont stockés dans un **CellTree** prenant cette forme :



Cet arbre peut s'étendre à l'infini et chaque block peut avoir un fils. Il permet en suivant les fils appartenant à la plus longue chaîne d'enfants de distinguer les fraudes des vrais blocs.

Conclusion du Projet

Bien que toutes les fonctions n’aient pas été faites et qu’il ne soit pas possible de déterminer un gagnant, nous pouvons tout de même tracer une conclusion. Les arbres représentent donc la block chain permettant d’être assuré de ne pas comptabiliser les blocs fraudeux en prenant la plus longue chaîne supposé être la totalité des votes d’un pays. Si le principe est très bien développé, avec l’évolution des technologies, le choix de faire confiance à la plus longue chaîne va, si ce n’est pas déjà être le cas, devenir dangereux puisqu’il suffit qu’une personne invente un nombre suffisamment élévé de votes frauduleux pour dépasser la longueur de la chaîne principale pour que ceux-ci soient comptabilisés à la place.

Bien que la méthode BlockChain possède des failles, l’utilisation des tables de hachage et d’arbre la rend très efficace en temps de calcul et peut rester utile à petite échelle.