Rapport Du Projet

Projet de Structure de Données S3-2022

MEKHAIL Paul
REBOURS Paulin

Table des matières:

Introduction au sujet

Le but de ce projet a été de créer un système de vote décentralisé en utilisant une structure de blockchain qui est massivement utilisé et a été rendu populaire grâce au crypto monnaie tel que le bitcoin.

Nous avons utilisé le Proof of work qui c'est le principe le plus simple à implémenter contrairement au Proof au stake qui peut être plus compliqué.

Pour pouvoir avoir une élection décentralisée, la meilleure solution fût l'utilisation de la structure blockchain(même si comme nous allons le voir à la fin, cette structure a quelques défauts qui font qu'elle est déconseillé pour des élections très importantes). En utilisant cette structure, chaque personne pourrait vérifier indépendamment la validité de vote donc la sécurité croît avec le nombre de vérificateurs.

Pour organiser notre dossier, nous avons :

-Le répertoire src, contient le main et un répertoire lib contenant tout les fichiers .c et leurs headers dans le repertoire headers ainsi que le main.

-Le makefile nous permet de compiler l'ensemble des fichiers du répertoire lib ainsi que le main avec la commande « make all » puis de l'executer en tapant « ./main ».

-Le repertoire bin contient tout les fichiers .o créés par notre makefile.

-Un fichier keys.txt avec l'ensemble des couples clés privé/publics générés par la fonction generate_random_data

-Un fichier candidates.txt avec les couples des candidats choisis par la fonction generate_random_data

-Un fichier déclaration.txt avec l'ensembles des couples et leur déclaration de vote.

-un fichier pending_votes.txt avec les votes en attente d'ajout dans la blockChain.

Cryptographie

Nous avons tout d'abord implémenté des fonctions permettant de trouver des nombres premiers (witness ; is_prime_naive ; is_prime_miller ; random_prime_number ; extended_gcd) et nous utilisons ensuite des nombres premiers pour trouver un couple de clé publique et privé avec generate_key_values .

Nous allons ensuite utiliser énormément la puissance modulaire pour chiffrer de déchiffrer des messages. Et pour faire cela, nous avons la fonction

-long modpow(long a, long m, long n); qui renvoie a puissance m modulo n qui utilise un algorithme dichotomique pour économiser du temps de calcul

-long modpow_naive(long a, long m, long n); est une version naive du premier modpow sans l'utilisations de l'algorithme dichotomique.

Par la suite, avec dans le fichier encryption.c nous avons implémenté 2 fonction:

-char *encrypt(long *crypted, long u, long n); qui permet de crypter une chaîne de taille size avec les entiers u et n.

-char *decrypt(long *crypted, int size, long u, long n); qui permet de décrypter une chaîne de taille size qui est cryptée à partir de u et n.

Key

```
typedef struct _key{
   long val;
   long n;
}Key;
```

La première structure de notre projet est la Key. Elle représente les clés (privées ou publique) et contient les valeurs de celle qu'elle représente. Pour en créer ou en manipuler, nous avons différentes fonctions :

-`void init_key(Key* key, long val, long n);`

Créé et initialise une clé avec les valeurs données en argument.

-`void init_pair_keys(Key* pkey, Key* skey, long low_size, long up_size);`

Initialise un couple de clés privées compris entre deux bornes données en argument.

-`char* key_to_str(Key* key);`

Renvoie une chaîne de caractères contenant les valeurs de la clé.

-`Key* str_to_key(char* str);`

Crée un clé et l'initialise avec les valeurs contenu dans la chaîne de caractères données en argument.

CellKey

```
typedef struct cellKey{
   Key *data;
   struct cellKey *next;
}CellKey;
```

Cette structure est simplement une liste chaînées de Key. Des fonctions sont également là pour en créer ou en modifier :

-CellKey *create_cell_key(Key *key);

Créer une cellKey avec comme premier élément la clé donné en argument.

-void ajout_en_tete(CellKey *key, CellKey **liste);

Ajoute en tête de la CellKey « liste » la Key « key ».

-CellKey *read_public_keys(char *nomFic);

Renvoie une CellKey contenant toutes les Keys trouvées dans le fichier « nomFic ».

-void print_list_keys(CellKey* LCK);

Affiche toutes les clés de la CellKey « LCK ».

-void delete_cell_key(CellKey *c);

Supprime la CellKey c de la liste chaînées a la quelle elle appartient en reliant son précédent et son suivant.

-void delete_liste_key(CellKey *c);

Supprime toute une liste chaînées de CellKey « c ».

```
-int len_cellkey(CellKey *c);
```

Renvoie la taille d'une liste c de CellKey.

HashCell

```
typedef struct hashcell{
    Key *key;
    int val;
}HashCell;
```

```
typedef struct hashtable{
    HashCell **tab;
    int size;
}HashTable;
```

Nous avons crées deux structures. La première est la table de hachage. Cette structure contient la table sous forme d'un tableau de pointeur sur des cellules et elle contient également sa taille. La deuxième structure est la cellule de la table. Cette structure contient une clé et un entier qui nous servira pour manipuler le tableau et calculer le gagnant. Une fois de plus des fonctions permettent de les créer et de les manipuler:

-`HashCell *create_hashcell(Key *key);`

Crée et initialise une cellule de la table de hachage

-`int hash_function(Key *key, int size);`

C'est une fonction de hachage utilisant la méthode de Knuth avec le nombre d'or réduit de un. Elle renvoi alors la cash de la clé en fonction de la taille du tableau. Elle hache l'entier val de la clé donnée en paramètre.

-`int find_position(HashTable* t, Key* key);`

La fonction retrouve la position de la clé dans le tableau t en prenant en compte un probing linéaire. Si le tableau est plein, elle renvoie la position dans laquelle devrait être la clé.

-`HashTable *create_hashtable(CellKey *keys, int size);`

La fonction crée une table de hachage et l'initialise avec la liste des clés donnée en paramètre et avec la size donnée en paramètre.

-void delete_hashtable(HashTable* t);

La fonction la libère toute la mémoire occupé par la table de hachage t.

-Key* compute_winner(CellProtected* decl, CellKey* candidates, CellKey* voters, int sizeC, int sizeV);

Elle revoie la clé de la personne qui a le plus de vote dans CellProtected. Pour cela, la fonctions vérifie d'abord les fraudes puis comptabilise tous les votes en utilisant l'entier de la cellule. Si une personne a déjà votée, alors son entier est égal à 1 sinon il est égal à 0. Pour chaque vote pour un candidat, l'entier de ce candidat sera incrémenté.

Structure Protected

```
typedef struct _protected{
   Key *pKey;
   char *mess;
   Signature *sign;
}Protected;
```

Nous avons créé un structure protected qui contient une déclaration : la clé public, la signature et le vote d'un citoyen. Pour la manipulé, nous avons ensuite implémenter différentes fonctions :

-Protected *init_protected(Key *pKey, char *mess, Signature *sign);

Initialise un protected avec la clé le vote et la signature en argument.

-int verify(Protected *pr);

Vérifie la validité du protected avec sa clé et sa signature.

-char *protected_to_str(Protected *pr);

Créé une chaîne de caractères contenant toutes les informations de la déclaration.

-Protected *str_to_protected(char *chaine);

Créé une déclaration (protected) à partir d'une chaîne de caractère qui contient les informations de cette déclaration.

-void delete_protected(Protected *p);

Supprime une déclaration et libère la mémoire alloué.

Structure CellProtected

```
typedef struct cellProtected{
   Protected *data;
   struct cellProtected *next;
} CellProtected;
```

Nous avons ensuite créé une structure CellProtected qui est une liste chaînées de déclarations protected. Idem, des fonctions permettent de les manipuler :

-CellProtected *create_cell_protected(Protected *pr);

Créé une cellProtected avec la déclaration pr en tête de liste (et sans suivant).

-void ajout_en_tete_protected(CellProtected *pr, CellProtected **liste);

Ajoute la cellProtected pr en tête de la liste (les 2 donnés en paramètre).

-CellProtected *read_protected(char *nomfic);

Créé un cellProtected avec comme protected un protected obtenu à partir d'une chaîne de caractères contenu dans le fichier « nomfic ».

-void afficher_cell_protected(CellProtected *liste);

Affiche toutes les déclarations contenues dans une liste de CellProtected (avec protected_to_str).

-void delete_cell_protected(CellProtected *c);

Supprime une cellProtected et libère sa mémoire allouée.

-void delete_liste_protected(CellProtected *c);

Idem pour toutes une liste (avec delete_cell_protected).

-void verification_fraude(CellProtected **liste);

Enlève les déclarations invalide d'une liste (avec delete_cell_protected).

-void fusion(CellProtected **l1, CellProtected **l2);

Fusionne 2 listes de CellProtected.

Structure Block

```
typedef struct block{
   Key* author;
   CellProtected* votes;
   unsigned char* hash;
   unsigned char* previous_hash;
   int nonce;
}Block;
```

Une autre structure que nous avons créé est le block. Il contient la clé de l'auteur de la déclaration, un cellProtected qui contient les déclarations, un entier nonce qui est une preuve de travail, et enfin deux chaînes de caractère hashées, une représentant le block et une représentant le block précédent (comme dans une liste chaînées). Un block sera dit valide si sa chaîne hachée commence par d zéros (L'entier d reste à choisir, dans notre main nous avons pris 4). Une fois de plus des fonctions permettent de les manipuler :

-`void ecrire_block(Block *b, char *nomFic);`

écris un block avec son contenu sous formes de chaînes de caractères dans le fichier « nomFic ».

-`Block *lire_block(char *nomFic);`

Inversement, crée un block grâce à une chaîne de caractères contenu dans le fichier « nomFic »;

-`char *block_to_str(Block *block);`

Retranscris les informations contenues dans un block dans une chaîne de caractère.

-unsigned char *miaou256(const char *mess);

Crypte un message en paramètre grâce à la fonction sha256 (de openssl).

-void compute_proof_of_work(Block *b, int d);

Initialise le nonce du block à 0 et l'incrémente tant que ce block n'est pas valide pour d zéros (la chaîne doit commencer par d zéros).

Nous avons remarqué qu'à partir de 6 zéros, la foncions met plus d'une minute pour réaliser le calcul. Malheureusement nous n'avons pas le graphe(peut être que je le mettrais si j'arrive). Donc le nombre de zéros qu'il faudrait utiliser est 5 tout en ayant un temps de calcul correct.

-int verify_block(Block* b, int d);

vérifie également la validité du block pour d zéros mais cette fois sans changer le nonce, et renvoie donc seulement 1 si valide et 0 sinon.

-void delete_block(Block *b);

Supprime un block et sa mémoire allouée.

Structure CellTree

```
typedef struct block_tree_cell{
   Block* block;
   struct block_tree_cell* father;
   struct block_tree_cell* firstChild;
   struct block_tree_cell* nextBro;
   int height;
}CellTree;
```

Finalement, notre dernière structure est CellTree. Chaque CellTree contient un block , une hauteur (un entier), et 3 pointeurs vers une autre CellTree : un père, un premier fils et un frère. De cette manière, nous avons un arbre complet, avec chaque cellTree n'ayant qu'un seul premier fils mais chaque fils permettant de trouver le frère d'après, jusqu'à ce que le pointeur soit NULL (dernier frère et donc dernier fils). De la même manière on peut remonter à la racine en suivant les pointeurs père. Une fois de plus, des fonctions sont la pour les manipuler :

-`CellTree* create_node(Block* b);`

Créé une CellTree avec comme block b, son père son premier fils et son frère le plus proche seront initialisés à NULL.

-`int update_height(CellTree* father,CellTree* child);`

Actualise la taille du père par rapport à celle de son fils. Renvoie 1 si la taille à été modifié et 0 sinon.

-void add_child(CellTree* father, CellTree* child);

Rajoute child en fils a father, remonte la liste des fils jusqu'au dernier si il a déjà un firstChild.

-void print_tree(CellTree* arbre);

Affiche l'arbre arbre donné en paramètre.Pour chaque noeuds, on affiche sa hauteur et sa valeur hashée.

-CellTree* racineCellTree(CellTree* node);

Fonction non demandée, rajoutez par nous. Renvoie la racine de l'arbre auquel appartient node.

-void delete_node(CellTree* node);

Supprime un noeuds de l'arbre et libère la mémoire alloué. Lie également père fils et frère de ce noeuds pour éviter les « trous » dans l'arbre.

-void delete_tree(CellTree* arbre);

Supprime un arbre entièrement et libère sa mémoire allouée.

-CellTree* highest_child(CellTree* cell);

Renvoie le fils avec la plus grande hauteur parmi les fils de cell.

-CellTree* last_node(CellTree* tree);

Renvoie la feuille avec la plus grande hauteur de l'arbre (= le bout de la chaîne la plus longue).

-CellProtected *longest_node_votes(CellTree *cell);

Renvoie la liste obtenu par fusion des votes de la plus longue chaîne de l'arbre

Fin du projet et Main

Finalement ce projet fût très intéressant pour pouvoir étudier la mise en place du vote électronique et surtout le vote par blockchain.

Nous pensons que cette structure possède quelques problèmes qui sont:

- Attaque des 51%: une personne qui possède 51% de la puissance de calcul pourrait fausser les résultats à son avantage
- Il faut qu'une personne fournissent les clés au personne qui sont autorisés à voter

Donc le consensus consistant à faire confiance à la plus longue chaîne peut être attaqué si une personne possède plus de 50% de la puissance de calcul qui est alloué à la blockchain.

Donc nous pensons qu'une blockchain pourrait être utilisé pour un processus de vote d'importance mineure comme entre amis ou autre. Cependant pour une élections tel que les présidentielles, elle serait fortement déconseillé compte tenu des risques et des enjeux majeurs.

Dans le dernier main, nous avons crées un jeu de données aléatoires, puis nous avons crées les blocs(chaque bloc contient 10 déclarations donc il y a 10 bloc). Nous les avons ensuite tous enregistrés puis nous les avons relu avec la fonction read tree pour vérifier son bon fonctionnement et nous avons affiché cet arbre et avons cherché à avoir le gagnant puis de l'afficher dans la console.

Nous avons donnée à la fonction compte_winner_BT 10 pour la taille de al table des candidats et 1010 pour la table des votants, et cela semble suffire.

Nous avons également décidés de réaliser les tests avec un 1 seule zéro pour le compute Proof of work car cela reste assez rapide à calculer et également parce que l'ordinateur de test arrêtait automatiquement les tests au bout d'un certain temps. Puis pour pouvoir tester avec valgrind car celui–ci reste quand même très lent pour les tests.

Nous avons ensuite libérés la mémoire. Nous avons 0 fuites de mémoires et 0 erreurs.

Pour compiler le programme , il suffit de lancer la commande: « make all » puis de lancer avec « ./main ». Tous les fichiers compilés sont dans le répertoire bin.

Tous les fichiers source sont dans le répertoire src avec les fichiers d'en tête dans le répertoire headers dans le répertoire src.

Description fichier de test et lancement du prog

Il y a un block contenu dans le fichier block.txt qui est récrit dans le fichier uiui.txt avec le bon nonce, les votes, votants et candidats sont crées grâce à generate random data et comme demandé, il y a 1000 votants et 5 candidats. Les autres fichiers sont crées en lançant le programme. Une clé aléatoire est crée pour l'auteur des blocs. Il y a plusieurs main de la main.c, ceux qui ne sont utilisés sont commentés. Le premier main correspond à la première partie du projet avec la cryptographie. Le deuxième main correspond aux tests de la deuxième et troisième parties. Le troisième main est le test de la quatrième partie. Finalement le dernier main est le test de la dernière partie du projet.

Pour tester une partie ou une autre, il suffit de décommenter cette partie et commenter les autres main puis d'exécuter et tout fonctionnera sans aucun problème.