Myriam Mabrouki
Anna Oquidam
–
Groupe 4

LU2IN006 Projet

Blockchain appliquée à un processus électoral

SORBONNE UNIVERSITÉ

RAPPORT DE PROJET

TABLE DES MATIÈRES

CADRE DU PROJET

L'objectif de ce projet est d'implémenter un processus électoral par scrutin uninominal majoritaire à deux tours en maintenant la sécurité et la protection de l'élection, à travers différents protocoles et structures de données.

Pour cela, le projet est divisé en 5 parties, chacune centrée sur un aspect du projet. La première partie concerne la cryptographie asymétrique et, plus précisément, au protocole RSA. La deuxième partie s'intéresse aux déclarations, de vote ou de candidature, que peuvent effectuer les citoyens. La troisième partie se concentre sur la centralisation de ces déclarations ainsi que sur leur intégrété. La quatrième partie, quant à elle, s'ouvre à l'implémentation d'un mécanisme de consensus. Enfin, la cinquième partie s'intéresse à la décentralisation des déclarations.

Tout au long du projet, pour alléger les affichages, nous utiliserons l'affichage hexadécimal des entiers long, forme plus compacte et plus lisible.

Gestion des fichiers et Makefile

Chaque partie est regroupée dans un fichier contenant les fonctions, appelé « `partiei.c` », pour i le numéro de la partie, d'un fichier « `partiei.h` » associé, contenant les définitions de fonctions et de structures, ainsi que d'un fichier main contenant des jeux de test, appelé « `main_pi.c` ».
Chaque partie peut être compilée et exécutée individuellement à l'aide du makefile et de la commande « `make main_pi` ».

Liste des fichiers par partie :

Partie 1 :	`partie1.c`	`partie1.h`	`main_p1.c`
Partie 2 :	`partie2.c`	`partie2.h`	`main_p2.c`
Partie 3 :	`partie3.c`	`partie3.h`	`main_p3.c`
Partie 4 :	`partie4.c`	`partie4.h`	`main_p4.c`
Partie 5 :	`partie5.c`	`partie4.h`	`main_p5.c`

Les fichiers .txt sont les fichiers utilisés pour récupérer des données.

`sortie_modpow.txt` contient les données de temps d'exécution des fonctions `modpow` de la partie 1 et `commande_modpow.txt` les commandes nécessaires pour afficher la courbe associée à l'aide de gnuplot.

PARTIE 1

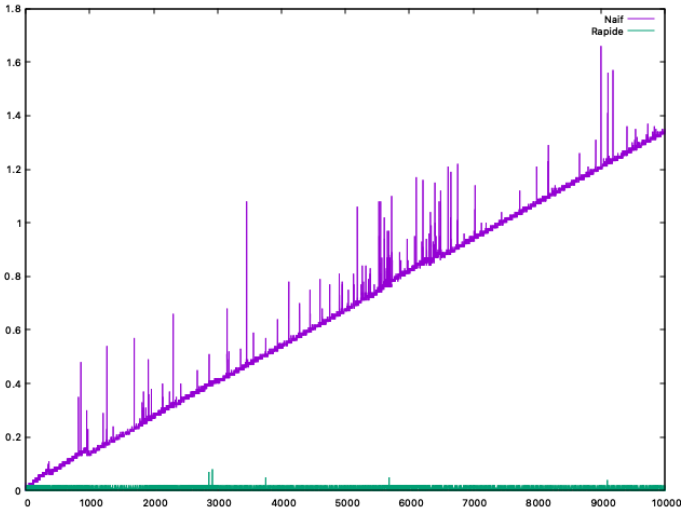
Développement d'outils cryptographiques

La première partie est centrée sur l'implémentation d'outils cryptographiques, en commençant par développer une méthode de génération de nombres premiers efficace, puis en réalisant le cryptage et décryptage de messages à l'aide de clés publiques et privées.

Résolution du problème de primalité

La première étape pour trouver un nombre premier est de déterminer si un nombre p est premier. Nous commençons par réaliser une fonction naïve, dont la complexité est en $O(p)$.
En utilisant cette fonction, le plus grand nombre premier que l'on arrive à tester en moins de 2 millièmes de secondes est de l'ordre de 200 000 (après deux lancements : 204061 en 2.042ms et 170539 en 2.018ms).

Puis, nous implémentons le test de Miller-Rabin, plus efficace pour calculer de grands nombres premiers.
La première fonction nécessaire pour ce test est une fonction calculant $a^b mod\, n$, que nous réalisons dans un premier temps de manière naïve également, et dont la complexité est en $O(b)$.
Après cette fonction naïve, nous implémentons une version plus efficace, et nous comparons les deux méthodes en fonction du temps de calcul. Après affichage des courbes obtenues à l'aide de gnuplot, nous obtenons le graphe ci-contre.



Nous observons alors que la méthode naïve a un temps de calcul croissant en fonction de m, tandis que la méthode rapide a un temps de calcul environ constant.

Le reste des fonctions nécessaires pour le test de Miller-Rabin est fourni, calculons une borne supérieure de la probabilité de l'algorithme :

Soit O l'ensemble des entiers entre 2 et $p - 1$.

Soit A l'ensemble des entiers de O qui ne sont pas témoins de Miller.
Soit a_i le numéro tiré au i-ème tour de boucle, pour i entre 1 et k.

Alors, on a : $P(échec) = P(a_1 \in A, a_2 \in A, \dots, a_k \in k)$
Les tirages sont indépendants, donc :

$$P(échec) = \prod_{i=1}^{k} a_i \in A$$

Les tirages sont aléatoires donc chacun des a_i suit la loi uniforme, ainsi :

$$P(a_i \in A) = \frac{card(A)}{card(O)}$$

$$P(échec) = \left(\frac{card(A)}{card(O)}\right)^k$$

Or, $card(A) \leq \left(1 - \frac{3}{4}\right) card(O)$ d'après l'énoncé, donc on a :

$$P(échec) \leq \frac{1^k}{4}$$

Donc la borne supérieure sur la probabilité d'erreur de l'algorithme est $\frac{1^k}{4}$.
Cela montre que pour des grands nombres, ce test est fiable.

Implémentation du protocole RSA

Une fois en capacité de générer un grand nombre premier, nous nous intéressons au chiffrement et déchiffrement de messages à l'aide du protocole RSA. C'est un algorithme de cryptographie asymétrique basé sur la génération de deux clés : une clé publique et une clé privée. De cette manière, on crypte le message avec notre clé privée, et le message peut être décrypté à l'aide de la clé publique, ou inversement.
Pour réaliser cela, nous commençons par générer le couple de clés publique/privé (fonction `generate_key_values`), à l'aide de deux nombre premiers p et q, générés aléatoirement, puis nous calculons :
$n = p \times q$
$t = (p - 1) \times q - 1)$
s tel que $PGCD(s, t) = 1$
u tel que $s \times u \; mod \; t = 1$
On a alors $pKey = (s, n)$ et $sKey = (u, n)$

Une fois ces deux clés générées, nous pouvons implémenter des fonctions permettant de crypter ou décrypter un message (qui est une chaine de caractères) à l'aide d'une clé (publique ou privée) :

Pour encrypter, il suffit d'allouer un tableau d'entiers longs de la même taille que le message, puis de calculer $m^s \; mod \; n$, pour m un caractère du message (converti automatiquement en entier).

Pour décrypter, le principe est le même à ceci près que le calcul est maintenant $m^u \; mod \; n$.

Chaque fonction peut être testée à l'aide du fichier `main_p1.c`.

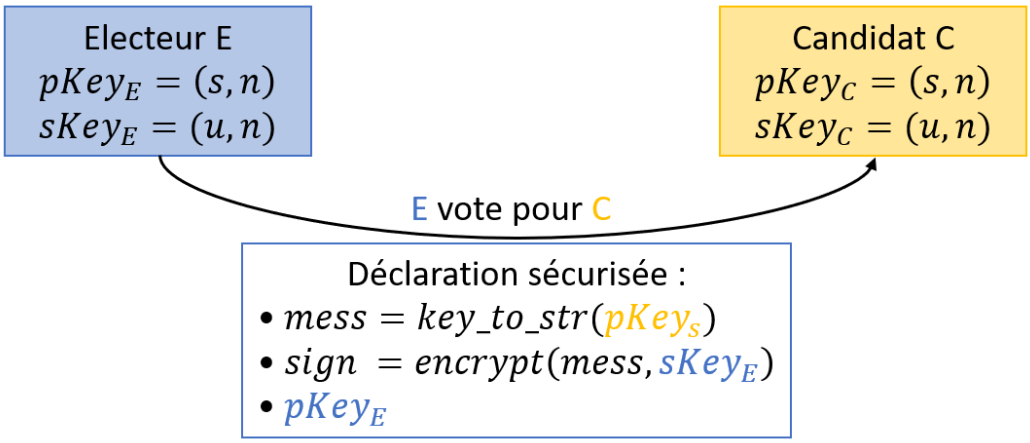
PARTIE 2

Déclarations sécurisées

La deuxième partie est centrée sur les déclarations sécurisées, en créant des structures adaptées : des clés publiques et privées représentant un électeur, des signatures permettant d'identifier l'électeur, et des déclarations sécurisées permettant à l'électeur de voter.

Manipulations de structures sécurisées

Pour créer des déclarations sécurisées, il faut tout d'abord modéliser l'objectif. Nous avons réalisé un schéma pour mieux visualiser.



Chaque personne, électeur ou candidat, possède une clé publique et une clé privée. Si un électeur veut voter pour un candidat, il doit réaliser une déclaration sécurisée, comportant la clé publique du candidat ($mess$), la version cryptée de cette clée, à l'aide de sa clé privée ($sign$), et enfin sa clé publique. De cette manière, si on décrypte $sign$ avec la clé publique de l'électeur, on devrait obtenir $mess$.

Pour réaliser cela, nous avons utilisé plusieurs structures différentes :
Une pour représenter les clés, une pour les signatures, et une pour les déclarations.

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

```
typedef struct signature
{
  long * content;
  int size;
} Signature;
```

```
typedef struct
protected {
  Key * pKey;
  char * mess;
  Signature * sgn;
} Protected;
```

Chaque structure comporte les mêmes types de fonctions :

• **`init_struct`** : alloue la taille de la structure et l'initialise avec les données en paramètre.
• **`key_to_str/str_to_key`** : transforme la clé en paramètre en chaine de caractères sous la forme « (u,n) » et inversement.
• **`signature_to_str/str_to_signature`** : transforme la signature (tableau d'entiers long) en chaine de caractères séparés par un « # » et inversement.
• **`protected_to_str/str_to_protected`** : transforme la déclaration en chaine de caractères sous la forme « pkey mess sign » et inversement.

Il y a également ces fonctions :

• **`init_pair_keys`** : génère une clé publique et une clé privée à l'aide de nombre premiers compris entre `low_size` et `up_size` bits et de la fonction `generate_key_values` vu à la première partie.
• **`verify`** : vérifie que la signature décryptée à l'aide de la clé publique corresponde bien au message de la déclaration à l'aide de la fonction `decrypt`, renvoie 1 si c'est le cas, 0 sinon.

De cette manière, nous pouvons à présent réaliser une déclaration sécurisée et l'afficher.

Création de données pour simuler le processus de vote

Maintenant que nous pouvons créer des déclarations, il est important de pouvoir les stocker, dans le but de les utiliser pour faire des simulations de vote.

Ainsi, la fonction generate_random_data permet cela :
Cette fonction prend en paramètres le nombre d'électeurs et de candidats.

Tout d'abord, elle génère aléatoirement autant de couple de clés publique/privée que d'électeurs, les stocke dans un tableau, et les écrit dans le fichier « keys.txt » au fur et à mesure (à l'aide de key_to_str).

Ensuite, elle sélectionne aléatoirement les candidats parmis les électeurs, stocke leur clés publiques dans un deuxième tableau et les écrit dans le fichier « candidates.txt ».

Enfin, pour chaque électeur, elle sélectionne aléatoirement un candidat pour qui voter, génère une déclaration sécurisée, la stocke dans un tableau et l'écrit dans le fichier « déclarations.txt ».

Chaque fonction peut être testée dans le fichier `main_p2.c`.

PARTIE 3

Base de déclarations centralisée

La troisième partie est centrée sur la gestion des électeurs, candidats et déclarations dans des fichiers, en transformant le contenu des fichiers en listes chainées, puis sur la détermination du gagnant de l'élection à partir de ces fichiers.

Lecture et stockage des données dans des listes chaînées

Nous voulons désormais pouvoir lire des données, telles que des clés ou des déclarations, dans des fichiers afin de les stocker dans des listes chaînées. Rappelons que les données lues seront soit des clés, soit des déclarations. Pour ce faire, nous devons d'abord créer des structures associées à une donnée. On obtient ainsi une structure pour ranger les clés et une autre pour y ranger les déclarations.

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

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

À partir de ces structures, nous voulons créer des fonctions afin de pouvoir créer un élément, ajouter un élément à une liste, afficher cet élément, supprimer cet élément, lire le fichier.... Ainsi, chaque structure comporte les mêmes types de fonctions :

- **`create_cell_struct`** : crée et alloue la taille de la cellule puis l'initialise avec la structure en paramètre.
- **`push_struct`** : crée une cellule correspondant à la structure en argument, à l'aide de la fonction précédente et l'ajoute à la liste en paramètre.
- **`read_public_struct`** : lit chaque ligne du fichier et crée une liste chainée de la structure correspondante avec les données lues.
- **`print_struct`** : affiche la structure en argument.
- **`print_list_struct`** : affiche la liste de structure passée en paramètre, chaque structure étant affichée grâce à la fonction précédente.
- **`delete_cell_struct`** : supprime et libère la mémoire d'une structure passée en argument.
- **`delete_list_struct`** : supprime et libère la mémoire d'une liste passée en argument, chaque élément de la liste étant supprimé grâce à la fonction précédente.

Ici, `struct = CellKey` ou `struct = CellProtected`

Détermination du gagnant de l'élection

Maintenant que nous avons des listes chainées permettant d'analyser le contenu de fichiers, nous voulons pouvoir déterminer quel est le gagnant de l'élection à partir des déclarations.

Pour cela, nous commençons par éliminer les déclarations frauduleuses, c'est-à-dire les déclarations où la signature n'est pas valide.
Il faut donc créer une fonction adaptée qu'on nommera ici : `delete_invalid_protected`. Pour cela, il nous suffit simplement de parcourir la liste de toutes les déclarations obtenues, vérifier la signature de chaque déclaration et supprimer la déclaration en question de la liste si sa signature n'est pas valide. On veillera bien sûr à faire la distinction entre un élément à supprimer en début ou en milieu de chaîne.

Une fois les déclarations frauduleuses retirées, nous pouvons chercher à déterminer le gagnant de l'élection. Pour cela, nous allons stocker les données des électeurs et candidats dans une table de hachage, représentée par les structures suivantes :

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

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

Cette table de hachage nous permettra d'accéder efficacement à chaque électeur ou candidat.

Nous commençons par créer la fonction de hachage, permettant de trouver la position absolue des éléments (en l'occurrence en fonction de la clé de la personne) dans le tableau. Nous avons choisi pour cela de faire la fonction suivante, pour (val, n) la clé, et $size$ la taille de la table de hachage :

$$hash((val, n)) = (val + n)\%\, size$$

Ensuite, nous avons créé des fonctions permettant de manipuler ces structures :
• **`create_hashcell`** : crée et alloue la taille de la cellule puis l'initialise avec la clé en paramètre, et la valeur à 0.
• **`find_position`** : cherche la clé en paramètres dans la table de hachage, si elle est trouvée elle retourne sa position, sinon elle retourne la position où elle aurait dû être (une case vide).
• **`create_hashtable`** : crée et initialise une table de hachage de la bonne taille, et la remplis à l'aide de la liste chainée en paramètre.
• **`delete_hashtable`** : supprime la table de hachage et toutes les cellules hachées associées.

Pour finir, nous implémentons la fonction `compute_winner`, qui calcule le vainqueur de l'élection en fonction d'une liste chainées de déclarations, une liste de candidats et une liste d'électeurs, ainsi que la taille des deux tables de hachage à créer.

En effet, nous utilisons deux tables de hachage différentes :
- La première contient la liste des candidats (H_c)
- La seconde contient la liste des électeurs (H_v)

Pour les deux tables, la case val des cellules hachées a une fonction différente :
Pour H_c, val contiendra le nombre de personnes ayant voté pour ce candidat.
Pour H_v, val contiendra l'information de si l'électeur a déjà voté (0 s'il n'a pas voté, 1 si il a voté).

Nous commençons donc par initialiser ces deux tables à l'aide des listes données en paramètres, puis nous supprimons toutes les déclarations frauduleuses de la liste de déclarations, et enfin nous parcourons cette liste en faisant voter chaque électeur, la table de hachage nous permettant d'accéder rapidement à la case de chacun.

Une fois ce parcours effectué, nous parcourons une dernière fois la table de hachage des candidats, pour déterminer celui comportant le plus de votes, et ainsi retourner sa clé.

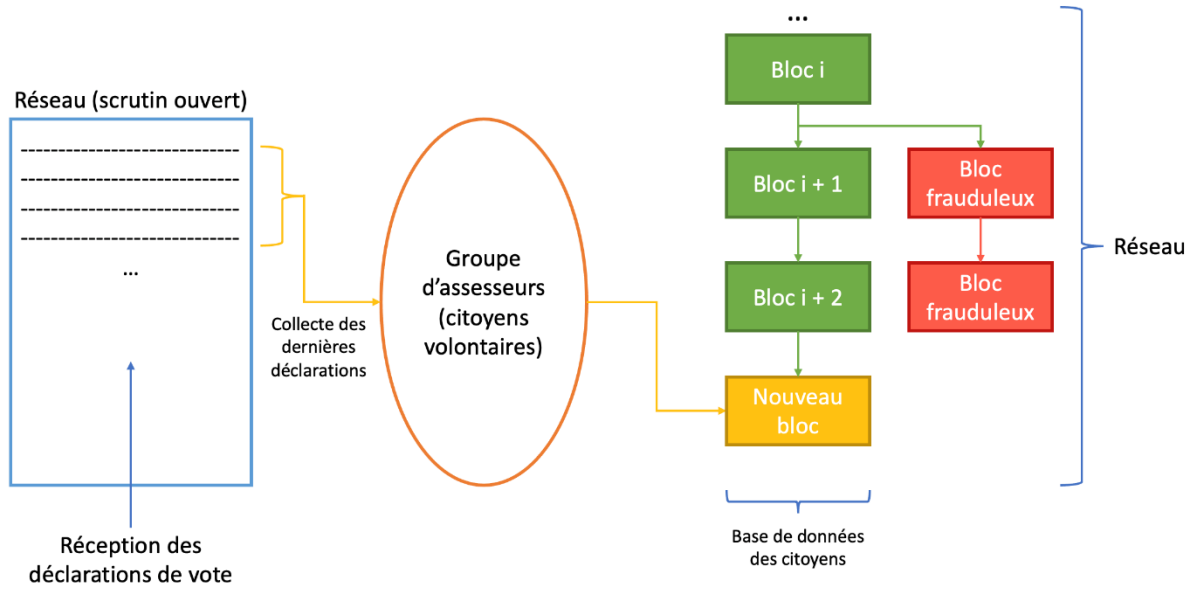
Chaque fonction peut être testée dans le fichier `main_p3.c`.

PARTIE 4

Blocs et persistance des données

La quatrième partie est centrée sur l'implémentation de blockchain, afin de permettre de créer un système de votes décentralisé. Les blocks contiennent les déclarations sécurisées et sont répartis dans une arborescence permettant de mieux gérer les fraudes.

Pour mieux visualiser le principe, nous avons réalisé un schéma :



Les citoyens envoient leurs déclarations sur le réseau, les assesseurs récupèrent les dernières déclarations puis, créent un nouveau bloc valide à l'aide d'une fonction de hachage cryptographique.
Ce bloc contient également la valeur de hachage du bloc précédent, permettant ainsi de déterminer l'apparition de blocs frauduleux (ils ont la même valeur de hachage du bloc précédent).
Comme le groupe d'assesseurs envoie les blocs à intervalles réguliers, les blocs frauduleux sont plus lents, donc on considère à tout moment que la branche la plus longue est la branche valide.
Les citoyens reçoivent tous les nouveaux blocs et enregistrent les blocs valides dans leur bases de données.

Structure d'un bloc et persistance

Nous commençons par développer l'utilisation des blocs, à l'aide de la structure suivante :

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

Un block est donc composé de :

- La clé publique de son auteur
- La liste de déclarations sécurisées
- La valeur hachée du bloc
- La valeur hachée du bloc précédent
- La preuve de travail

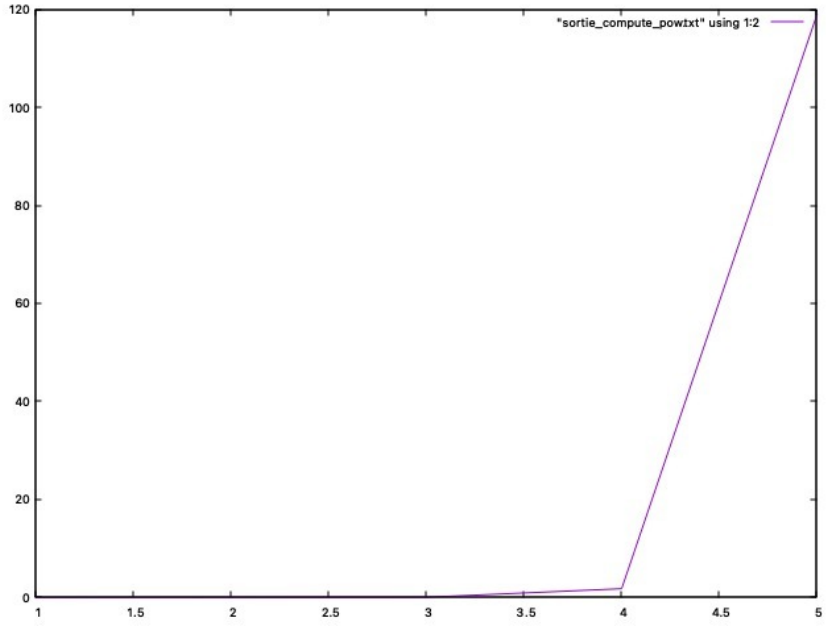
La preuve de travail est un entier permettant de montrer la qualité de la valeur de hachage.

A l'aide de cette structure, nous réalisons les fonctions suivantes :
• **`write_block`** : écrit un bloc dans un fichier.
• **`read_block`** : lit un bloc dans un fichier et le crée.
• **`block_to_str`** : génère une chaine de caractères représentant un bloc : la clé de l'auteur, la valeur hachée du bloc précédent, une représentation des votes, la preuve de travail.
• **`str_to_hash`** : transforme une chaine de caractères en valeur hachée à l'aide de l'algorithme `SHA256` fourni.
• **`compute_proof_of_work`** : calcule la valeur hachée du bloc en incrémentant `nonce` à chaque tour, jusqu'à obtenir `d` 0 successifs au début de la valeur hachée, `d` étant donné en paramètre de la fonction.
• **`verify_block`** : vérifie qu'un bloc est valide en vérifiant que sa valeur hachée commence bien par `d` 0 successifs.
• **`delete_block`** : supprime un block en libérant toute la mémoire associée à l'auteur et au contenu de la liste chainée de déclarations.

N.B. : Nous avons choisi de faire la fonction `delete_block` de cette manière et non pas comme il était indiqué de le faire dans l'énoncé car nous avons considéré qu'il était plus pertinent de libérer l'entièreté de la mémoire associée directement, plutôt que de risquer de créer des fuites mémoires par la suite.

Nous avons ensuite étudié le temps moyen de la fonction `compute_proof_of_work` en secondes, en fonction de la valeur de `d`. Ainsi, les valeurs obtenues à l'aide de gnuplot sont représentées sur la courbe ci-après.

Nous n'avons pas pu faire un grand nombre de valeurs de `d` car le temps de calcul devenait vite très long, mais nous pouvons constater sur le graphe la tendance exponentielle du temps de calcul de cette fonction.



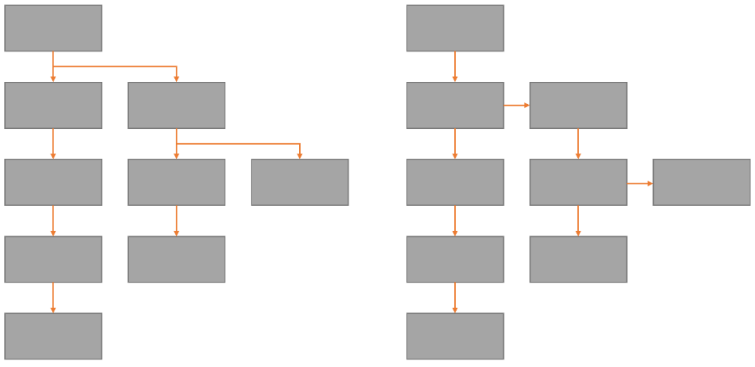
Nous pouvons également constater que la fonction dépasse une seconde de temps de calcul à partir de d = 4.

Structure arborescente

Nous allons à présent modéliser les chaines entre les blocs. Pour cela, nous prenons en compte la potentielle présence de fraude, impliquant qu'il y ait deux blocs avec le même bloc précédent. Cela crée alors une structure arborescente, que voici :

```
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;
```

Voici un exemple de représentation de l'arbre telle qu'on pourrait l'imaginer (gauche), et la représentation utilisée pour cette structure (droite) :



A l'aide de cette représentation, nous pouvons facilement accéder aux différents fils d'un bloc, et à la hauteur de l'arbre à partir d'un certain bloc.
Ensuite, nous implémentons les fonctions nécessaires à la manipulation de cette structure :
• **`create_node`** : crée et initialise un nœud de hauteur égale à zéro.
• **`update_height`** : met à jour la hauteur du nœud parent quand l'un des fils est modifié, retourne 1 si la hauteur est modifiée, 0 sinon.
• **`add_child`** : ajoute un fils à un nœud, et met à jour la hauteur de tous ses ascendants.
• **`print_tree`** : affiche un arbre : pour chaque nœud, la fonction affiche sa hauteur et la valeur hachée du bloc.
• **`delete_node (resp. tree)`** : supprime un nœud de l'arbre (resp. l'arbre).

Une fois ces fonctions de bases implémentées, il faut alors déterminer les fonctions permettant le vote. Pour cela, nous commençons par faire plusieurs fonctions pour obtenir des informations importantes sur l'arbre :

• **`highest_child`** : détermine et renvoie le nœud fils avec la plus grande hauteur.
• **`last_node`** : retourne le dernier bloc de la chaine la plus longue.
• **`fusion_declarations`** : fusionne deux listes chainées de déclarations sécurisées.
La complexité de la fonction `fusion_declarations` est en $O(n)$. Pour avoir une complexité en $O(1)$, il aurait fallu changer la structure actuelle de `CellProtected` en liste doublement chainée :

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

L'attribut `prev` est un pointeur vers la `CellProtected` précédente.

Maintenant que nous avons les outils nécessaires pour le parcours de la plus grande branche de l'arbre et la fusion de listes chainées, nous implémentons la fonction `protected_in_highest_child`, permettant de renvoyer la liste chainée des déclarations de la plus grande branche de l'arbre.

Chaque fonction peut être testée dans le fichier `main_p4.c`.

PARTIE 5

Simulation du processus de vote

La cinquième et dernière partie est centrée sur la mise en place de la simulation du processus de vote, en modélisant l'arrivée des votes et l'envoi des blocs à l'aide de répertoires et de fichiers :

Vote et création de blocs valides

Pour commencer, nous allons manipuler des fichiers et répertoires :

• **`Blockchain`** : répertoire contenant un fichier par bloc, représentant la blockchain qu'un citoyen a construit en local à partir des blocs reçus sur le réseau.
• **`Pending_block`** : fichier contenant le dernier bloc créé et envoyé par un des assesseurs, en attente d'ajout dans la blockchain.
• **`Pending_votes.txt`** : fichier contenant les déclarations de votes en attente d'ajout dans un bloc.

Puis, nous réalisons plusieurs fonctions permettant de modifier ces fichiers :

• **`submit_vote`** : soumet un vote en l'ajoutant à la fin du fichier `Pending_votes.txt`.
• **`create_block`** : crée un bloc valide contenant les votes en attente dans le fichier `Pending_votes.txt` à la suite de la plus longue chaîne de l'arbre, supprime le fichier lu puis écrit le bloc dans le fichier `Pending_block`.
• **`add_block`** : vérifie que le bloc du fichier `Pending_block` est valide, si oui, crée un fichier représentant ce bloc dans le répertoire `Blockchain`, puis supprime le fichier `Pending_block`.

De cette manière, nous pouvons simuler un processus de vote sécurisé à distance efficacement.

Lecture de l'arbre et calcul du gagnant

Maintenant que la gestion des fichiers est réalisée, nous allons nous occuper de transformer ces fichiers en données exploitables pour nos fonctions précédemment réalisées, notamment construire l'arbre correspondant aux blocs du répertoire Blockchain.

Pour cela, nous écrivons une fonction read_tree(), qui :

- Crée un nœud de l'arbre pour chaque block du répertoire, en les stockant dans un tableau.

- Parcourt ensuite le tableau de nœuds, pour chaque nœud, recherche tous ses fils et les ajoute à la liste des fils de ce nœud.
- Effectue un dernier parcours de l'arbre pour trouver la racine et la retourner.

Une fois cette fonction réalisée, nous avons tous les outils en main pour déterminer le gagnant dans la fonction `compute_winner_BT` : nous commençons par récupérer la liste de déclarations de la plus longue chaîne de l'arbre, étant donné que nous faisons confiance à la plus longue chaîne, puis nous déterminons le vainqueur à l'aide de la fonction compute_winner de la partie 4.

Pour finir, nous réalisons la fonction main mettant en œuvre toutes les fonctions créées jusqu'ici, disponible dans le fichier `main_p5.c`, et qui simule un processus électoral comportant 1000 électeurs et 5 candidats.

CONCLUSION

Avis et expérience

Dans le cadre d'un processus de vote, l'utilisation d'une blockchain est plutôt judicieuse, car efficace et sécurisée. Cependant, c'est un processus plutôt coûteux en temps et en ressources étant donné le peu de citoyens que nous utilisons, ainsi que les fonctions cryptographiques assez simplifiées. Il faudrait probablement aborder le projet différemment pour une utilisation à grande échelle.

De plus, le consensus consistant à faire confiance à la plus longue chaine ne permet pas d'éviter toutes les fraudes. En effet, il suffit de créer de nouvelles branches à l'approche de la fin de la période de vote pour obtenir deux branches de la même longueur, l'une étant celle valide, l'autre étant celle frauduleuse.

Pour conclure ce projet, nous souhaitons indiquer que nous avons pris beaucoup de plaisir à le réaliser, malgré la difficulté de certaines parties, et nous aimerions remercier l'équipe enseignante qui était très présente et nous a bien accompagné tout au long du projet.

Nous sommes également conscientes que certains éléments de notre version du projet ont des problèmes, notamment le fait qu'il n'y ait qu'en moyenne que 98% des personnes qui votent, alors qu'elles devraient toutes voter. Nous avons essayé de comprendre d'où venait le problème et n'avons pas su le résoudre.
Cependant, ce problème étant mineur, nous avons considéré que nous pouvions le laisser ainsi.