Rapport sur le projet C : Blockchains

CHREIM Charles, AKIN Yavuz

Introduction :

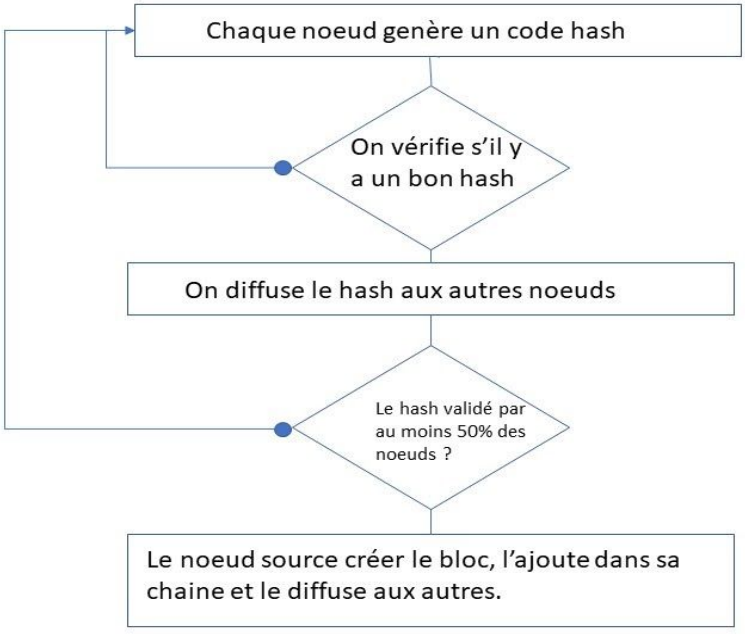
Pour pouvoir réaliser notre projet sur les blockchains, nous nous sommes inspirés de son application la plus courante : les crypto-monnaies. Notre projet se divise en 2 parties : la première s'intéresse à la construction et à la manipulation de blockchains. La seconde partie concerne la construction de la simulation d'un réseau P2P en s'inspirant de l'algorithme de Chandy-Misra vu dans le TD4.

Modélisation :

Pour modéliser la communication peer-to-peer on va utiliser une fonction qui calcule et trouve le bon hash, ensuite on va choisir aléatoirement un sommet qu'on considère le sommet qui a trouvé le bon hash en premier. Ce sommet va recevoir le nouveau bloc qui sera par la suite diffusé vers les autres sommets.

On a choisi Chandy-Misra car l'application réelle de cet algorithme est de transmettre des "*processus*", par exemple les résultats des fonctions qui tournent dans chaque noeud vers les autres noeuds. Un modèle plus réel serait de faire tourner la fonction du génération du hash dans chaque noeud.
On s'arrête dès qu'un noeud trouve un bon hash, ce qui est vérifié par un consensus qu'on pourrait modéliser par un message en plus avec Chandy Misra.

L'organigramme d'un programme plus complet :



Les structures employés :

Dans la première partie de l'algorithme, on utilise les structures suivantes pour manipuler les blockchains:

- **Blockchain :** La blockchain est une liste chaînée de Blocs, sa structure est composé de l'adresse de la tête de liste, et de la longueur de liste.

- **Block**: le bloc est la structure qui contient les messages que l'on veut transmettre aux autres utilisateurs. Dans le cas de le crypto-monnaie, le message s'agit d'une transaction.

  Le bloc est composé de :
  - Un index (donne sa position dans la liste)
  - Du nom de l'auteur
  - D'un timestamp (date d'écriture)
  - Du message
  - D'un POW (expliqué plus tard)
  - D'un hash
  - Du previous hash
  - L'adresse du prochain block

Pour former un réseau P2P on se sert des structures suivantes :

- **Graph:** Le graphe est représenté sous forme de liste d'adjacence, elle contient la taille du graphe, et le tableau des adresses des successeurs.

- **Node :** Elle contient l'identité du noeud, le poids de celui-ci, et son successeur.

- **ParamsAlgoChM :** Elle est inspiré du TD4, elle contient les paramètres de l'algorithme. C'est-à-dire:
  - La source qui envoie le message.
  - L'adresse du graphe.
  - Un tableau qui contient le nombre de messages non reconnus par les utilisateurs.
  - Le tableau qui contient l'adresse des blockchains des utilisateurs.

- Le tableau des prédécesseurs.

- **MessageList:** Elle correspond à la liste chaînée de messages.

- **Message**: Elle contient, l'indice du récepteur et du transmetteur, le bloc à envoyer, le statut du message, et le message suivant dans la liste de message.

Les fonctions utilisées :

1. Blockchain.h

1.1. **unsigned long POWGenerator () :** une fonction qui va générer un nombre aléatoire qui est au maximum de l'ordre de grandeur du timestamp, ce nombre correspond au proof of work.Le proof of work permet de modifier le hash.

1.2. **unsigned long HashGenerator (Block* block) :** Cette fonction calcule le hash à partir du contenu du bloc. La fonction se sert du contenu entier du bloc pour générer le hash . On relance cette fonction jusqu'à ce que qu'on ai le bon hash, la seule information qui varie à chaque fois est le POW : le proof of work. Au départ nous voulions générer le hash en s'inspirant de la famille des fonction de hachage SHA-2. Ces fonctions étaient trop compliqués, on avait du mal à les comprendre. Nous avons donc préféré l'option de créer les hash par nos propres moyens en essayant de rendre le processus le plus complexe possible (par exemple pour les chaînes de caractères dans les blocs on se sert de la longueur de ces chaînes multipliés par un nombre assez grand pour générer le hash code).

1.3. **int VerifyHash (unsigned long hash) :** Cette fonction vérifie que le hash contient deux 9 consécutifs. Les hash qui ne remplissent pas cette condition ne sont pas acceptés dans l'algorithme. L'objectif derrière cette condition est de ralentir la création de blocs. Cela nous permet d'éviter la fraude, parce que modifier un bloc de la chaîne signifie modifier tous les blocs qui suivent (d'après la définition du hash). Ainsi, rattrapper le retard sur les autres utilisateurs devient impossible car la génération de hash prend trop de temps (la fraude est alors démasqué par les autres utilisateurs). De plus on veut éviter que tous les utilisateurs puissent créer des blocs en même temps, par conséquent on ralentit le processus. Nous avons choisi la condition des deux 9 parce qu'en effectuant des tests, la

durée pour que l'ordinateur trouve un hash correct était de quelques secondes.

1.4. **Block* CreateBlock(unsigned long index, char author[20],unsigned int timestamp,char message[20],unsigned long previous_hash) :** Fonction qui va créer un block à partir des informations fournies, elle va ensuite envoyer son adresse.

1.5. **void InitializeChain (BlockChain * chain) :** Fonction qui va créer le bloc genesis et va l'insérer dans la chaîne.

1.6. **void InsertBlock (BlockChain * chain) :** Dans cette fonction on demande à l'utilisateur d'entrer les informations nécessaires pour créer un bloc, ensuite on créer le bloc grâce à une fonction CreateBlock, ensuite on l'insère dans la chaîne.

1.7. **void Print_Block(Block *block)** et **void Print_Blockchain(BlockChain * chain) :** Deux fonctions pour afficher respectivement un bloc et une chaîne.

Optimisations possibles : La difficulté de la condition sur le hash que l'on a créé ne dépend pas du nombre d'utilisateurs. Dans un cas réel, il se pourrait qu'au bout d'un moment il y ait tellement d'utilisateurs qui travaillent que la condition ne soit plus suffisante pour ralentir la création de blocs . Par exemple on aurait pu augmenter le nombre de 9 consécutifs désirables en fonction du nombre d'utilisateurs, le problème est que notre algorithme est une simulation et même si le programme marque qu'il y a plusieurs utilisateurs, finalement, le seul utilisateur qui calcule le hash est l'ordinateur lui-même, c'est pourquoi on a pas voulu ajouter cette adaptation de la difficulté de la condition du hash.

En deuxième lieu, on pouvait complexifier encore plus le hash code pour avoir une meilleur sécurité des données. Ceci étant dit complexifier plus le hash et la condition de sa validité pèse beaucoup sur nos machines, déjà avec les conditions actuelles nos ordinateurs mettent parfois une dizaines de secondes pour trouver un POW et par la suite un hash qui convient.

2. Graph.h

2.1. **void create_graph(Graph* G, const char* grapheFileName)** et **void print_graph(Graph* G) :** la première fonction crée la liste d'adjacence à partir d'un fichier texte et la deuxième l'affiche.

2.2. **void adapted_chandy_misra_algorithm(Graph* G,ParamsAlgoChM* paramsAlgo, int source) :** On a adapté l'algorithme de Chandy-Mistra du TD4 pour qu'à la place de trouver les distances les plus courtes d'un noeud à l'autre, l'algorithme soit capable de simuler un network P2P. En effet, on a modifié le contenu des messages de telle façon à ce que ce contenu correspond à des blocs plutôt qu'à des distances minimales. En gardant le squelette de l'algorithme intact (principe des acknowledges ...), on a modifié le reste de l'algorithme pour qu'elle soit adapté à son nouvel utilisation (par exemple, on a supprimé la phase 2 de l'algorithme car on a pas besoin de s'intéresser aux cycles négatifs). Ainsi, le noeud source (l'utilisateur qui a réussi à trouver le bon POW) propage le bon bloc dans le réseau grâce au fonctionnement de l'algorithme de Chandy-Misra modifié.

2.3. **void adapted_chandy_misra_initialize(ParamsAlgoChM* paramsAlg, Graph* G, int source) :** fonction adaptée qui initialise l'algorithme de Chandy-Mistra et qui place le bloc genesis dans chaque sommet du graphe et qui crée les paramètres de l'algorithme.

2.4. **void messageListInitialize(MessageList* list) :** Elle initialise la liste des messages à être transmis.

2.5. **void addMessageViaHead(MessageList *messageList, Message* newMessage):** Elle ajoute un message dans la liste de message par la tête de la liste.

2.6. **void sendReceiveMessage(ParamsAlgoChM* paramsAlgo, MessageList *messageList, Message* message, int* tabvisit):** Elle permet d'envoyer un message de la liste de message, dans certains cas elle renvoie aussi un acknowledge.

2.7. **int sendMessageToSuccessors(ParamsAlgoChM* paramsAlgo, MessageList* messageList,enum messageStatus status, int idNode, int* tabvisit):** Elle permet d'envoyer un message à ses successeurs.

2.8. **void sendReceiveAck(ParamsAlgoChM* paramsAlgo, Message* message):** Elle permet d'envoyer un acknowledgement à son prédécesseur.

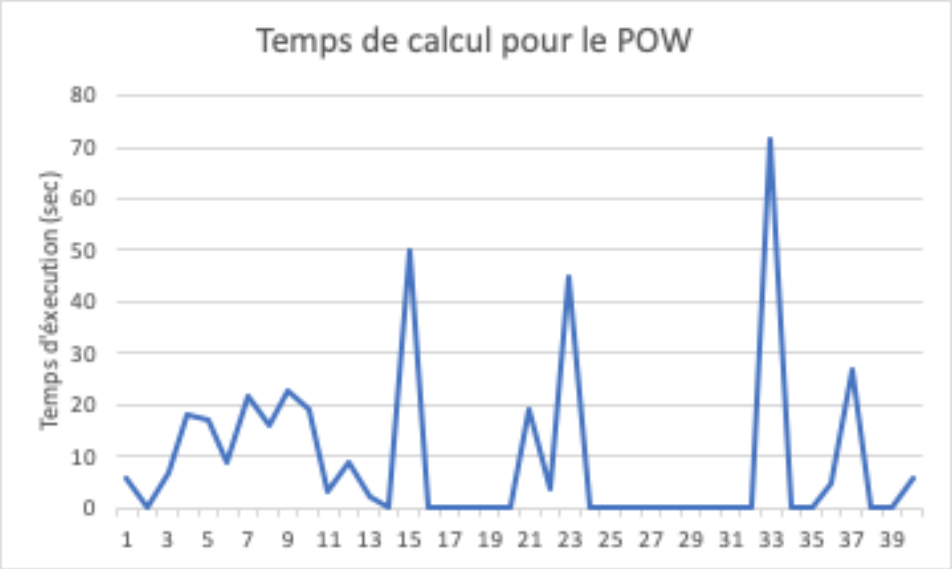
2.9. **void InsertNewBlock(ParamsAlgoChM* paramsAlgo,int source) :** fonction qui insère un nouveau bloc dans la source, ce bloc sera par la suite propagé vers les autres sommets.

2.10. **int random_node(Graph * G) :** fonction qui choisit un sommet aléatoire du graphe destiné à être la source.

2.11. **int generateRandomInRange(int lower, int upper) :** fonction qui génère un nombre entier aléatoire entre deux bornes [lower,upper].

2.12 **Message* extractRandomMessage(MessageList *messageList):** Elle extrait un message aléatoire de la liste de message.

Optimisations :



Ce graphique nous démontre que la résolution de la condition pour le hash est beaucoup trop instable. Il serait intéressant de pouvoir une fonction de hachage plus efficace ou une condition plus compliquée pour la confirmation du code hash.

Présentation :

Pour permettre à l'utilisateur d'interagir d'une manière souple et compréhensive avec notre programme, on a décidé de faire une interface qui permet à ce dernier de :

- Créer un bloc
- Afficher le graphe
- Afficher la blockchain d'un noeud
- Afficher toutes les blockchains

Conclusion :

Notre programme fonctionne bien, comme prévu. Ceci étant dit une connaissance de C plus approfondi est nécessaire pour qu'on puisse simuler un modèle plus réaliste.