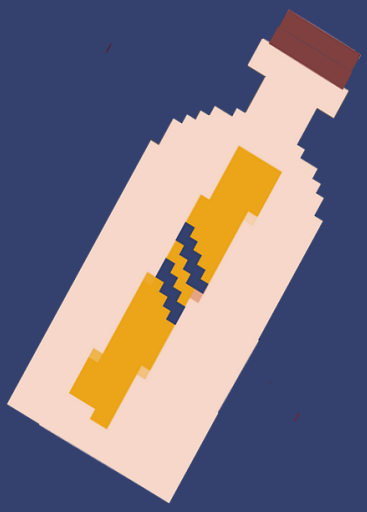
BOTTLE

RAPPORT PROJET C BLOCKHAIN



1er Avril 2020

LORRIAUX Tristan

MAISONNAVE Lucas

MINES Saint-Étienne

Objectifs du projet

Il s'agit d'abord d'un travail de documentation et de recherche : le premier objectif demeurait de nous renseigner sur la blockchain et de comprendre comment elle fonctionne.

Le second objectif résidait dans l'implémentation d'un programme simple en reprenant les bases vues en travaux pratiques, utilisant la base des blockchains : une pseudo-liste chainée, une structure pour les blocs

Le troisième objectif était le stockage des blocs, ainsi que leur sécurisation par hachage et l'implémentation d'une proof of work.

Enfin le dernier objectif qui subsistait était interface interactive et d'imaginer une utilisation commerciale possible pour notre application.

I°) Documentation

Ce travail s'appuie sur :

Les consignes et le syllabus du Projet C, Blockhain, 2020 :

https://campus.emse.fr/pluginfile.php/68815/mod_resource/content/1/2029_algo_prog_II_projet_final%20%281%29.pdf

La vidéo YouTube *« Comment fonctionne une blockchain »* - *Expliqué simplement* (traduit) de Simply Explained – Savjee :

https://www.youtube.com/watch?v=SSo_EIwHSd4

Les travaux et le Git de Lauri Hartikka, développeur finlandaise :

https://lhartikk.github.io/

Le git de DGSKI, développeur C++/Python, ayant développé une cryptomonnaie en C, le Noincoin :

https://github.com/dgski/blockchain-in-c

II°) Fonctionnalités de l'application, mécanisme et fonctionnement du code

Cette application est une simulation de messagerie qui fonctionne grâce aux blockchains. Assurant ainsi la sécurité des messages transmis l'application permet aussi la création et la gestion de comptes dont les informations confidentielles sont chiffrées.

L'application est codée en C. Elle se décompose en 3 parties

1. **blockhain.c et blockhain.h**

Ils contiennent l'essentiel du code, les bases de l'application. Commençons par le header.

Celui-ci déclare :

- Les macros :

N (taille maximale de le blockhain), MaxMessage (taille maximale d'un message, DIFFICULTY (permettant de régler la dificulté de la PoW[1]), FileNameBC (qui contient le nom du fichier texte où se trouve stockée les données de la blockchain).

HASH_SIZE (taille du hachage qui sort de Hash256[2]), HASH_HEX_SIZE (taille du hachage une fois convertit en hexadécimal), BINARY_SIZE (sa taille en binaire), BLOCK_STR SIZE (taille après conversion d'un bloc en chaîne de caractères).

- Les structures :

donnee : qui contient toutes les données stockées dans la blockhain utiles à la lecture, l'envoi ou la réception du message : émetteur, destinataire, date, et le message.

bloc : qui contient le hachage du précédent bloc, un index renseignant sur sa position, la donnée, une variable once utile pour la PoW ainsi que le lien vers le bloc suivant (pointeur) et son propre hachage.

genesis : premier bloc qui pointe vers la suite de la blockchain[3]

[1] **Proof of Work**
[2] **Cf blockchain.c**
[3] **Fonctionnement identique aux listes chaînées**

- La déclaration de toutes les fonctions du .c

```c
#include <stdbool.h>
#ifndef BLOCKCHAIN_H

    #define BLOCKCHAIN_H

    #define N 200           //Taille max de la blockchain
    #define MaxMessage 150          //Taille max des messages
    #define DIFFICULTY 4    //Difficulté de la proof of work
    #define FileNameBC "SaveBC.txt"

    #define HASH_SIZE 32
    #define HASH_HEX_SIZE 65
    #define BINARY_SIZE (HASH_HEX_SIZE*4+1)
    #define BLOCK_STR_SIZE 100

    typedef struct{
        char message[MaxMessage];
        char exp[20];
        char dest[20];
        char date[20];
    }donnee;

    struct bloc{
        char precHash[HASH_HEX_SIZE];
        char Hash[HASH_HEX_SIZE];
        int index;
        donnee* donnee;
        int nonce;      //utile pour la Pow
        struct bloc *lien;
    };

    struct genesis
    {
        struct bloc *premier;
        int taille;
    }*Genesis;

    void ajout_block(donnee* message);
    char *toString(struct bloc *blocks, char *str);
    void printBlock(struct bloc *blocs);
    void printAllBlock(void);
    void init_Data(donnee* data);
    bool HashMatchesDifficulty(char Hex[HASH_HEX_SIZE]);
    void hexToBinary(char input[HASH_HEX_SIZE], char output[BINARY_SIZE]);
    void hash256(unsigned char *output, const char *input);
    bool IsValidBlock(struct bloc* newBlock, struct bloc* previousBlock);
    void calculHash(struct bloc* Block);
    char *Hex_Hash(struct bloc *Bloc, char *output);
    void LoadBlockChainFromFile(char* filename);
    void SaveBlockChain(char* filename);
    void initGenesis();
#endif
```

Vient ensuite le *.c .

Nous incluons les bibliothèques qui nous serons utiles (math.h au cas où, stdio.h bien sûr, stdlib.h et string.h pour la gestion des chaînes de caractère), ainsi que des bibliothèques issues de OpennSSL, qui nous serons utiles pour le cryptage à diverses reprises des informations : sha.h et crypto.h (on ne charge pas tout OpenSSL pour éviter de surcharger la mémoire).

L'utilisation de la bibliothèque OpenSSL nécessite son installation au préalable à l'aide de la commande Ubuntu : sudo apt-get install libssl-dev [4]

```c
#include "blockchain.h"
#include "openssl/sha.h"
#include "openssl/crypto.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

Détaillons maintenant le code qui se décompose en deux parties, les fonctions dites « utilitaires » et les fonctions dites « principales ».

[4] Instruction différente pour Windows, non référencée ici

FONCTIONS UTILITAIRES

- **toString** prend en entrée un bloc et une chaîne de caractère et renvoie la conversion d'un bloc en une chaîne de caractère concaténant toutes les infos.
- **printBlock** prend en entrée un bloc et imprime ses composantes
- **printAllBlock** imprime tous les blocs de la blockchain
- **init_Data** initialise une donnée prise en entrée
- **hexToBinary** prend en entrée une chaîne de caractère hexadécimale idéalement et une chaîne de caractère de sortie où sera stockée la conversion en binaire
- **hash256** prend elle aussi en entrée deux chaînes de caractère, la première étant non hachée et la deuxième servant à stocker le hachage après le passage de SHA256 (fonction de OpenSSL). Notons que la sortie est en vérité inexploitable et il faudra alors effectuer d'autres opérations pour récupérer un hachage décent
- **Hex_Hash** prend en entrée un bloc et une chaîne de caractère

```
char *toString(struct bloc *bloc, char *str)    //Conversion d'un block en type "string
{
    if(bloc == NULL)
        return NULL;

    char block_string[BLOCK_STR_SIZE] = {0};
    char b[3]= {0};
    char c[32]= {0};
    sprintf(b, "%d", bloc->index);
    strcpy(block_string, b);
    strcat(block_string, bloc->precHash);
    strcat(block_string, bloc->donnee->date);
    strcat(block_string, bloc->donnee->dest);
    strcat(block_string, bloc->donnee->exp);
    strcat(block_string, bloc->donnee->message);
    sprintf(c, "%d", bloc->nonce);
    strcat(block_string, c);


    strcpy(str, block_string);
    return str;
}


void printBlock(struct bloc *blocx)     //Imprime un block
{
    printf("bloc %d : ", blocx->index);
    printf("precHash : ");
    printf(" %s", blocx->precHash);
    printf(" [%s]->", blocx->donnee->exp);
    printf("[%s] ", blocx->donnee->dest);
    printf("%s : ", blocx->donnee->date);
    printf("%s //", blocx->donnee->message);
    printf("Hash : ");
    printf(" %s", blocx->Hash);
    printf("  %p\n\n\n",blocx->lien);
}


void printAllBlock(void)        //Imprime tous les blocks de la blockchain
{
    struct bloc * curr = Genesis->premier;
    while(curr)
    {
        printBlock(curr);
        curr = curr->lien;
    }
}


void init_Data(donnee* data)        //Initialise le message, l'expéditeur, le destinat
{
    strcpy(data->date, "");
    strcpy(data->dest, "");
    strcpy(data->exp, "");
    strcpy(data->message, "");
}


void hexToBinary(char input[HASH_HEX_SIZE], char output[BINARY_SIZE])   //Conversion h
{
    long int i = 0;
    while (input[i])
    {

        switch (input[i])
        {
            case '0':
                strcat(output, "0000");
                break;
            case '1':
                strcat(output, "0001");
                break;
            case '2':
                strcat(output, "0010");
                break;
            case '3':
                strcat(output, "0011");
                break;
            case '4':
                strcat(output, "0100");
                break;
            case '5':
                strcat(output, "0101");
                break;
            case '6':
                strcat(output, "0110");
                break;
            case '7':
                strcat(output, "0111");
                break;
            case '8':
                strcat(output, "1000");
                break;
            case '9':
                strcat(output, "1001");
                break;
            case 'A':
            case 'a':
                strcat(output, "1010");
                break;
            case 'B':
            case 'b':
                strcat(output, "1011");
                break;
            case 'C':
            case 'c':
                strcat(output, "1100");
                break;
            case 'D':
            case 'd':
                strcat(output, "1101");
                break;
            case 'E':
            case 'e':
                strcat(output, "1110");
                break;
            case 'F':
            case 'f':
                strcat(output, "1111");
                break;
            default:
                printf("Caractère hexadécimal invalide %c\n",
                       input[i]);
        }
        i++;
    }
}
```

de sortie, qui sera renvoyée à la fin de la fonction, celle-ci contenant le hachage du bloc cette fois-ci en hexadécimal (exploitable)

- **initGenesis** qui initialise la tête de la blockchain

FONCTION PRINCIPALES

ajout_block, principale fonction du programme qui permet l'ajout d'un nouveau bloc à la blockchain. Elle prend en entrée une donnée, initialise éventuellement si nécessaire la blockchain, teste la validité du bloc, et si le bloc est valide l'ajoute à la blockchain. Notons que le parcours de la blockchain est similaire au parcourt d'une lise chaînée

- **IsValidBlock** prend en entrée un bloc et son prédécesseur, et renvoie un booléen, fonction du test la validité d'un bloc : index exact, bon hachage du précédent bloc et recalcul du hachage du bloc testé
- **calculHash** permet le calcul du hachage d'un bloc pris en entrée, implémentant en plus lors du calcul de ce hachage la résolution d'un puzzle (PoW) qui lorsqu'il est effectué modifie le hachage
- **HashMatchesDifficulty** prend en entrée un hash et vérifie que celui-ci possède le bon nombre de 0 (principe de notre PoW)

Enfin nous ne nous étendrons pas sur les diverses fonctions SaveBlockChain et LoadBlockChainFromFile qui permettent de rendre l'application pérenne dans le temps en sauvegardant la blockchain ou en la chargeant depuis un fichier texte.

```c
/*             Fonctions Principales              */

void initGenesis()
{
    Genesis->premier = NULL;
    Genesis->taille = 0;
}

void ajout_block(donnee* message)       //Pour l'ajout d'un nouveau block
{
    struct bloc *currentbloc = Genesis->premier;
    /*while(currentbloc->lien != NULL)    //Idem aux listes chainées : on parcourt tout
    {
        currentbloc=currentbloc->lien;
    }*/

    struct bloc *nouv_bloc = (struct bloc *)malloc(sizeof(struct bloc));
    nouv_bloc->donnee = (donnee*)malloc(sizeof(donnee));
    nouv_bloc->lien = NULL;
    nouv_bloc->donnee = message;

    if(currentbloc != NULL)
    {
        nouv_bloc->index = currentbloc->index + 1;
        strcpy(nouv_bloc->precHash, currentbloc->Hash);
    }
    else
    {
        nouv_bloc->index = 1;
        strcpy(nouv_bloc->precHash, "");
    }
    calculHash(nouv_bloc);

    if(IsValidBlock(nouv_bloc,currentbloc))     //Test de validité du block
    {
        printf("yo\n");
        nouv_bloc->lien = currentbloc;          //Ajout du block au débutde la blockchain
        Genesis->premier = nouv_bloc;
        Genesis->taille++;
    }
    else                                        //Bloc Invalide
    {
        printf("\nBloc invalide, veuillez retenter s'il vous plaît\n");
    }
}


bool IsValidBlock(struct bloc* newBlock, struct bloc* previousBlock)    //Test de la v
{
    char hash_to_test[HASH_HEX_SIZE]={0};

    Hex_Hash(newBlock, hash_to_test);
    if(previousBlock != NULL)
    {
        if (previousBlock->index + 1 != newBlock->index)    //Index exact?
        {
            printf("\nBlock invalide : indexs invalide (%d + 1 != %d).\n", previousBlo
            return false;
        }

        else if (strcmp(previousBlock->Hash, newBlock->precHash) != 0)  //Bon hachage
        {
            printf("\nBlock invalide : hash précédent invalide.\n");
            return false;
        }
    }

    if (strcmp(hash_to_test, newBlock->Hash) != 0)     //Recalcul du hachage
    {
        printf("\nBlock invalide: hashs différents (%s != %s)\n", hash_to_test, newBlo
        return false;
    }

    return true;
}


void calculHash(struct bloc* Bloc)  //Calcul du Hash d'un block
{
    Bloc->nonce = 0;        //Mise en place d'une variable nonce qui sera celle qui ch
    char hash_hex[HASH_HEX_SIZE];

    Hex_Hash(Bloc, hash_hex);
    strcpy(Bloc->Hash, hash_hex);

    while(!HashMatchesDifficulty(hash_hex))
    {
        Bloc->nonce++;
        Hex_Hash(Bloc, hash_hex);       //Conversion du hash en hex
    }
    strcpy(Bloc->Hash, hash_hex);       //Mise à jour du hash

    return;
}


bool HashMatchesDifficulty(char Hex[HASH_HEX_SIZE])     //Vérifie que le hash possède
{
    char binaryHash[BINARY_SIZE] =  {0};
    hexToBinary(Hex, binaryHash);

    // Creation de la chaine de caractère "0"*difficulty"
    char *prefix = malloc(DIFFICULTY + 1);
    memset(prefix, '0', DIFFICULTY);
    prefix[DIFFICULTY] = '\0';

    char *checker = NULL;
    checker = strstr(binaryHash, prefix);
    return checker == binaryHash;
}

/*----------------------------Sauvegarde----------------------------------*/

void SaveBlockChain(char* filename)
{
    FILE* file = fopen(filename, "w");
    struct bloc* current = Genesis->premier;
    while(current != NULL)
    {
        fprintf(file, "%s %d %d %s %s %s %s %s\n", current->precHash,current->nonce,
        current = current->lien;
    }

    fclose(file);
}

void loadBlockChainFromFile(char* filename)
{
    FILE* file = fopen(filename, "r");
    struct bloc *current = (struct bloc *)malloc(sizeof(struct bloc));
    current->donnee = (donnee*)malloc(sizeof(donnee));
    while(fscanf(file, "%s %d %d %s %s %s %s %s", current->precHash,&current->nonce, &
    {
        printf("hello\n");
        current->lien = Genesis->premier;
        Genesis->premier = current;
        current = (struct bloc *)malloc(sizeof(struct bloc));
        current->donnee = (donnee*)malloc(sizeof(donnee));
    }
    fclose(file);
}
```

2. **compte.c et compte.h**

Ils contiennent le code permettant la gestion sécurisée d'un compte, la connexion, l'inscription, et l'affichage des messages liés à ce compte. Commençons par le header.

Celui-ci déclare :

- Les macros :

  COMPT_H (variable de contrôle permettant d'éviter plusieurs fois le même code), MAX_WORD_LENGHT (taille maximale d'un mot utile pour les ID et les mots de passe), FileNameID (nom du fichier texte sauvegardant les comptes), NbMessages (permettant de paramétrer le nombre de messages affiché pour un compte).

- Les structures :

  Explicites : Identifiant et TabID (tableau des identifiants, contenant tous les identifiants et les mots de passe).

```c
//Permet de gérer la création et la gestion des comptes

/*-------------------------------------------------------

1/  -Créer nouveau Compte
    -Se connecter
2/ Une fois connecté:
    -Choix de l'utilisateur avec qui connecter
3/ Affichage de de la conversation et possibilité d'envoi de messages

-------------------------------------------------------*/

#include <stdbool.h>

#ifndef COMPTE_H

    #define COMPTE_H


    #define MAX_WORD_LENGHT     29      // Maximum word length à réajuster
    #define NbID 20             // Nombre maximal d'ID
    #define FileNameID "SaveID.txt"
    #define NbMessages 10       //Nb de messages à afficher

    struct Identifiant
    {
        char username[MAX_WORD_LENGHT];
        char password[65];
    };

    typedef struct
    {
        int taille;
        struct Identifiant* ID;
    }TABID;


    //void menu(HashTable *hashTab);
    void LoadTabIDFromFile(TABID* TabID, const char* TabIDFileName);
    void insertElementToTabID(TABID* TabID, struct Identifiant* Element);
    void initTabID(TABID *TabID);
    bool checkExistenceElementInTabID(TABID* TabID, struct Identifiant* Element);
    void printTabID(TABID* TabID);
    void SaveTabID(TABID* TabID, const char* TabIDFileName);
    bool SignIn(TABID* TabID, char* exp);  //S'identifier, renvoie true si l'identification s'est bien passé
    bool SignUp(TABID* TabID);  //Créer un nouveau compte, renvoie true si la création de compte s'est bien passé
    char *CryptPassword(struct Identifiant *Element, char *output);
    char *DisplayUsers(TABID* TabID, char* choice);
    void SendMessage(char* dest, char* exp);
    void DisplayMessages(char* dest, char* exp);

#endif
```

Vient ensuite le *.c .

Nous incluons les bibliothèques qui nous serons utiles (math.h au cas où, stdio.h bien sûr, stdlib.h et string.h pour la gestion des chaînes de caractère), ainsi que blockchain.h bien sûr pour récupérer les fonctions de cryptage.

- **initTabID** prend en entrée le tableau TabID et l'initialise
- **LoadTabIDFromFile** prend en entrée le fichier texte où sont sauvegardées les données et TabID, mettant les données stockées du premier dans le second
- **SaveTabID** est l'opération inverse
- **insertElementToTabID** prend en entrée un compte et l'insère dans TabID, mis en entrée lui aussi
- **checkExistenceElementInTabID** prend en entrée un compte et TabID et vérifie si le compte est contenu dans TabID : elle renvoie un booléen
- **CryptPassword** prend en entrée un Identifiant et une sortie et renvoie la sortie contenant le mot de passe crypté
- **PrintTabID** imprime le tableau TabID contenant tous les comptes
- **SignIn** prend en entrée une chaîne de caractère exp et TabID et permet la connexion et la mise à jour de la variable expéditeur
- **SignUp** prend en entrée TabID et permet la création d'un nouveau compte
- **DisplayUsers** prend en entrée un choix et TabID, renvoie ce choix concernant le destinataire d'un message
- **SendMessage** prend en entrée deux chaînes de caractère, destinataire et expéditeur et permet l'envoi d'un message entre les deux partis
- **DisplayMessages** idem mais affiche les messages échangés entre les deux partis

3. **compte.c et compte.h**

Nous ne nous focaliserons pas sur cette partie du code. Le lecteur pourra étudier sur cette partie du code, peu complexe, qui utlise simplement les divers outils de la bibliothèque SDL permettant l'affichage de fenêtres, la créeation de boutons, etc… en bref le nécéssaire pour la création d'une simple interface graphique en C.

III°) Possibles failles de sécurité, exploitation et possibles améliorations

La première faille majeure demeure dans la vulnérabilité des fichiers de sauvegarde .txt. Même en .bin, ils demeurent lisibles. Les mots de passe sont certes cryptés, mais si un utilisateur le souhaite, il peut modifier le cryptage et mettre à mal le fonctionnement de l'application. Il ne pourra cependant pas accéder aux vrais mots de passe.

La seconde faille demeure l'absence de vérification entre le chargement des fichiers et l'exécution du programme. Il y a alors pu avoir une possible modification des blockchains, sans qu'on le sache. La solution réside dans l'utilisation de la fonction IsValidBlock sur chaque bloc lors du chargement des blocs.

Les voies d'améliorations de cette application sont multiples : l'application à un réseau d'ordinateur (fin d'une messagerie locale, en réalité peu utile), voire son extension à internet (nécessitant alors la communication avec un serveur). Il faut noter par ailleurs qu'une étape de sécurité est manquante ici : la décentralisation des blockchains sur un réseau P2P, où chaque utilisateur vérifie à chaque ajout de block la validité de celui-ci.

Enfin la dernière amélioration possible serait l'envoi d'images cryptées, mais le cryptage des images nécessite d'autres techniques (clés) que nous n'exploitons pas ici.

Des applications comme Bottle existent déjà sur le marché, les messageries sécurisées ne manquent pas: on pourrait par exemple citer Telegram. Celles-ci sont grandement téléchargées du fait de la sécurisation des messages (terrorisme, espionnage, ou simple besoin de sécurité).