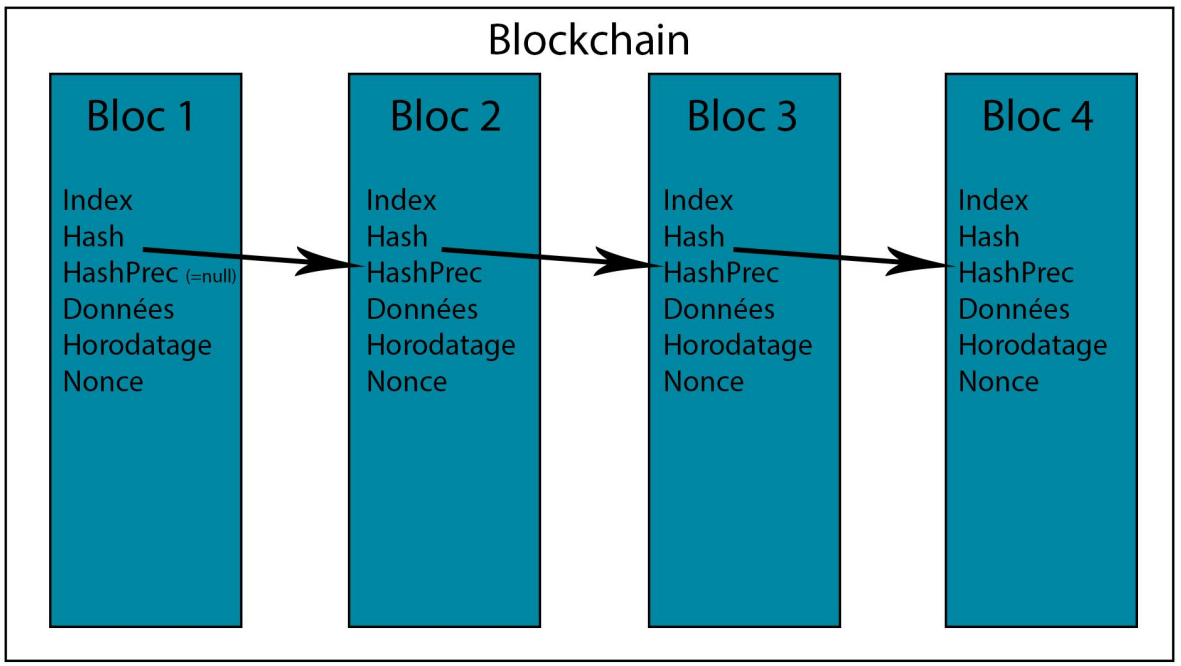
DUVAUX Maxime

TP - Sécurité des Systèmes Distribués

Master 1 Cyber-Défense et Sécurité de l'Information
Université Polytechnique des Hauts de France - Année Universitaire 2022-2023

Représentation de notre blockchain:



Notre blockchain est une chaîne de blocs agissant comme une base de données. Chaque bloc contient des données insérées par l'utilisateur lorsqu'il définit la taille de sa blockchain. Un bloc est aussi composé d'un index que l'on incrémente à chaque fois pour éviter les doublons d'index mais également la corruption de la blockchain avec un bloc non valide au milieu d'autres blocs valides. On retrouve également un hachage, le hachage générant une chaîne de caractères de longueur fixe (généralement 32 caractères) à partir d'un ensemble de données de n'importe quel volume. Pour lier les blocs entre eux, on utilise le hachage du bloc précédent (défini à null pour le premier bloc puisqu'il n'y a pas de bloc avant), c'est également un bon moyen de vérifier que les données sont liées entre elles. On y ajoute un horodatage/timestamp pour vérifier la date de minage du bloc et son insertion dans la blockchain. Enfin, on utilise un nonce qui servira de preuve de travail, c'est-à-dire qu'il s'agira d'un entier qui permet d'éviter les comportements indésirables au sein de la blockchain.

Utilisation de Java:

Java est un langage avantageux pour la création de blockchain pour plusieurs raisons:

- Prend en charge une interface de programmation d'application (API)
- Utilisation de la programmation orientée objet par les développeurs de blockchain (existe aussi en C++ ou en Python par exemple)
- Interface sécurisée, pas d'utilisation de pointeurs

À titre d'exemple, la blockchain Ethereum a été développée en Java.

Mise en place de notre blockchain:

On commence d'abord par modéliser ce qu'est un bloc selon la représentation que nous avons défini plus haut:

```
public class Bloc {
    protected int index;
    protected String hash;
    protected String hashPrec;
    protected String données;
    protected long timeStamp;
    protected int nonce;
```

On définit ensuite le constructeur de notre classe Bloc.

```
public Bloc(int index, String hashPrec, String données, long timeStamp) {
    this.index = index;
    this.hashPrec = hashPrec;
    this.données = données;
    this.timeStamp = timeStamp;
    this.hash = calcHash( bloc: this);
    nonce = 0;
}
```

Nonce est défini à 0 puisque c'est un paramètre qu'on ne souhaite pas influencer lors de la création d'un bloc et d'une blockchain. Le hachage sera lui calculé automatiquement par une fonction que l'on présentera plus tard.

```
public String calcHash(Bloc bloc) {
    if (bloc!=null) {
        MessageDigest digest = null;
        try {
            digest = MessageDigest.getInstance("SHA-256");
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
            return null;
        }
        String aHasher = hashPrec + timeStamp + nonce + données;
        byte[] bytes = digest.digest(aHasher.getBytes(StandardCharsets.UTF_8));
        StringBuilder aRetourner = new StringBuilder(); //On utilise StringBuild
        for (byte b: bytes) aRetourner.append(String.format("%02x",b)); //Nous c
        return aRetourner.toString();
    }
    return null;
}
```

Pour développer cette fonction, nous importons la classe Java.security.MessageDigest qui contient des fonctions de hachage dont on va se servir ici.
MessageDigest.getInstance renvoie un objet de type MessageDigest et applique l'algorithme mis en paramètre, ici on applique l'algorithme SHA-256. On renvoie une exception définie par la classe importée si l'algorithme que l'on décide d'implémenter n'existe pas.
Ensuite on définit quelles données on souhaite hasher et on les définit comme une chaîne de caractères (équivalent à utiliser toString() sauf que celle-ci a été redéfinie dans cette classe)
On souhaite ensuite que les bits qui composent le hachage soient des chiffres ou bien des lettres alors on applique la norme UTF-8 qui permet de coder l'ensemble des caractères du « répertoire universel de caractères codés ». On stocke tout ça dans un tableau de type byte[] que l'on appelle bytes et afin de construire notre hachage, on parcourt ce tableau et l'on ajoute chaque bit à une chaîne de caractères, sous le format "%02x", ce qui nous donne à chaque ajout à la chaîne de caractères une chaîne de 2 caractères qui commence obligatoirement par un 0. On utilise ensuite la fonction toString() redéfinie pour qu'elle soit du type String et non StringBuilder (qui nous sert pour utiliser la fonction append() qui ajoute des caractères à une chaîne de caractères).

```
public String toString() { //Permet de retourner notre bloc comme une chaine de caractères - Redéfinition de la fonction toString
    return "N° de bloc: "+index+"\nHash: "+hash+"\nHash precedent: "+hashPrec+"\nDate: "+(new Date(timeStamp))+"\nContenu du bloc:
}
```

On redéfinit la fonction toString, ce qui nous permet de gérer l'affichage d'un bloc. Plutôt que de retourner un bloc avec une suite d'informations, on décide de le mettre en page et de l'afficher comme une chaîne de caractères.

```
public void mineBloc (int i) {
    nonce = 0;
    while (!(getHash().substring(0,i).equals(GestionZeros.gererZeros(i)))) {
        nonce++;
        hash = calcHash( bloc: this);
    }
```

Le minage de bloc consiste en la création d'un bloc valide au sein d'une blockchain.

Voyons maintenant comment représenter la blockchain en fonction de notre classe Bloc.

```
public class Blockchain {
    protected List<Bloc> blocs = new ArrayList<>();
    protected int difficulte;
```

On définit la classe Blockchain et la blockchain prend la forme d'une liste de blocs, blocs que l'on crée grâce à la classe Bloc que l'on a conçue précédemment. La difficulté représente, quant à elle, la puissance et le temps nécessaires au hachage de chaque bloc. C'est ce que l'on a appelé i dans la fonction mineBloc ci-dessus.

```
public Blockchain (int difficulte) {
    this.difficulte = difficulte;
    Bloc b = new Bloc ( index: 1, hashPrec: null, données: "Test",new Date().getTime());
    b.mineBloc(difficulte);
    blocs.add(b);
}
```

On initialise notre blockchain avec un premier bloc déjà inséré pour éviter la corruption de cette dernière. Cependant on aurait également pu créer une fonction permettant de créer ce premier bloc que l'on aurait appelé afin que l'utilisateur puisse déjà saisir des données dans le premier bloc plutôt que de perdre un bloc et ainsi gagner du temps à la mise en place de sa blockchain. (On aurait déjà défini les autres paramètres de la même manière avec l'index 1 ou 0 et un hashPrec initialisé à null)

```
public void creerNouveauBloc() {
    if (blocs.size() > 0) { //On ne souhaite pas utiliser cette fonction si aucun bloc n'a été créé au préalable -> Plus simple p
        Scanner sc = new Scanner(System.in);
        System.out.println("Quelles sont les données que vous souhaitez insérer pour le bloc n°"+(getDernierBloc().getIndex()+1)+
        String aInserer = sc.nextLine(); //Plus simple pour l'utilisateur d'avoir une interface avec un scanner que de passer par
        Bloc dernierBloc = getDernierBloc(); //Permet de ne pas vérifier la validité d'un bloc à chaque fois - pratique pour l'in
        Bloc b = new Bloc( index: dernierBloc.getIndex() + 1, dernierBloc.getHash(), aInserer, new Date().getTime());
        b.mineBloc(difficulte);
        blocs.add(b);
    }
}
```

On crée notre fonction creerNouveauBloc qui permet à l'utilisateur d'ajouter des données à sa blockchain. On vérifie d'abord qu'il ne s'agisse pas du premier bloc afin d'éviter les erreurs sur le hachage précédent. Nous n'avons normalement pas besoin de cette vérification puisque nous construisons d'office un premier bloc dans le constructeur, à l'initialisation de la blockchain. Elle serait plutôt nécessaire si on décidait de créer une fonction pour initialiser ce premier bloc.

On utilise ensuite la fonction scanner qui permet à l'utilisateur de saisir des données à l'aide de la console. La fonction scanner hérite de la classe Java.util.Scanner. On définit les données à insérer comme une chaîne de caractères mais on aurait pu également décider de lire des entiers ou même un fichier. Pour se faire, on aurait pu créer un fichier .txt et le lire dans le scanner: `try (Scanner sc = new Scanner(new File(nom_du_fichier)))` puis afficher les différentes lignes de ce fichier comme une chaîne de caractères en donnée de chaque bloc.
Pour éviter les corruptions de notre blockchain et s'assurer que nos index soient uniques et suivis, on incrémente à chaque fois l'index en se basant sur l'index du dernier bloc miné.

Afin de récupérer le dernier bloc, on crée la fonction getDernierBloc qui consiste à aller chercher le dernier élément présent dans notre ArrayList.

```
public Bloc getDernierBloc() {
    return blocs.get(blocs.size()-1);
}
```

Dans notre ArrayList, l'index du dernier élément correspond à la taille de cette Array List-1.

```
public String toString() {
    StringBuilder builder = new StringBuilder();
    for (Bloc b : blocs) {
        builder.append(b).append("\n");
    }
    return builder.toString(); //On retourne notre blockchain comme une chaine de caractères
}
```

On redéfinit également la classe toString pour afficher notre liste de blocs comme une chaîne de caractères dont les blocs sont séparés à chaque fois par un saut de ligne.

On veut maintenant tester ces fonctions et afficher notre blockchain.

```
public class Main {
    public static void main (String[] args) {
        Blockchain blockchain = new Blockchain( difficulte: 5);
        for (int i=1; i<blockchain.difficulte; i++) blockchain.creerNouveauBloc();
        System.out.println(blockchain);
    }
}
```

On définit une blockchain composée de 5 blocs. Plutôt que d'appeler à chaque fois la fonction et de changer le nombre de fois auquel on y fait appel, on crée une boucle for pour répéter à chaque fois cette opération, quitte à ce que notre programme soit plus lent.

Test:

```
Quelles sont les données que vous souhaitez insérer pour le bloc n°2 ?
Ceci est le deuxieme bloc
Quelles sont les données que vous souhaitez insérer pour le bloc n°3 ?
Ceci est le troisieme bloc
Quelles sont les données que vous souhaitez insérer pour le bloc n°4 ?
Ceci est le quatrieme bloc
Quelles sont les données que vous souhaitez insérer pour le bloc n°5 ?
Ceci est le cinquieme bloc
N° de bloc: 1
Hash: 000001c874240d1beff2629e83d3abe6765a76a751a0d1e6dd9ea4e0abeac150
Hash precedent: null
Date: Thu Jan 05 23:28:24 CET 2023
Contenu du bloc: Test

N° de bloc: 2
Hash: 0000086520abf0a5607fe53ac0e193b0f6486840aba8bf0408fa1cf24ca3d42d
Hash precedent: 000001c874240d1beff2629e83d3abe6765a76a751a0d1e6dd9ea4e0abeac150
Date: Thu Jan 05 23:29:00 CET 2023
Contenu du bloc: Ceci est le deuxieme bloc

N° de bloc: 3
Hash: 0000016fbdd8edb7bd5a1cc105bf5454697ee396bcb8ebd7a3a2bfc8a3e46fb7
Hash precedent: 0000086520abf0a5607fe53ac0e193b0f6486840aba8bf0408fa1cf24ca3d42d
Date: Thu Jan 05 23:29:34 CET 2023
Contenu du bloc: Ceci est le troisieme bloc

N° de bloc: 4
Hash: 00000b2d0d029a35d8a350ffb69f2b048d1a296746e2193f09ad089f920c0e77
Hash precedent: 0000016fbdd8edb7bd5a1cc105bf5454697ee396bcb8ebd7a3a2bfc8a3e46fb7
Date: Thu Jan 05 23:29:57 CET 2023
Contenu du bloc: Ceci est le quatrieme bloc
```

```
N° de bloc: 5
Hash: 000007fae131a34b508d9ce2249e4d6b66af2c82e2ea24bb7cb644a5c5391fd2
Hash precedent: 00000b2d0d029a35d8a350ffb69f2b048d1a296746e2193f09ad089f920c0e77
Date: Thu Jan 05 23:30:46 CET 2023
Contenu du bloc: Ceci est le cinquieme bloc
```