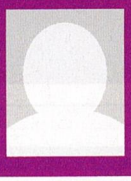


Sylvain SAUREL
Développeur Java / Android
https://www.ssaurel.com - sylvain.saurel@gmail.com

Création d'une Blockchain en Java : gestion des transactions

niveau 200

Partie 2

Au coeur du Bitcoin et des cryptomonnaies, la technologie Blockchain est une véritable révolution technologique. Nous vous avions ainsi présenté dans le numéro 216 de Programmez comment réaliser les fondations de votre propre Blockchain en Java. Dans cet article, nous passons aux choses sérieuses puisque nous allons y ajouter la gestion des transactions.

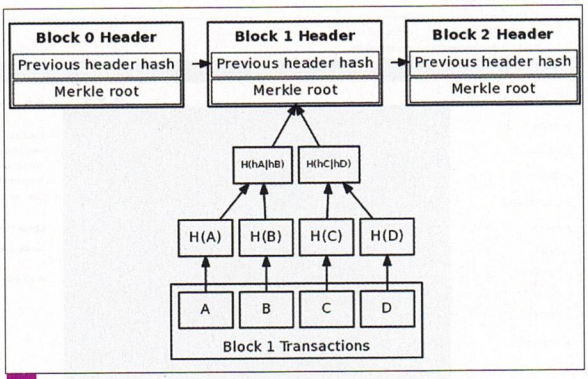
Ajout des transactions aux blocs

Dans la première partie de cet article, nous avions utilisé des blocs contenant de simples messages. Maintenant que notre Blockchain s'étoffe, nous devons y stocker des transactions complètes afin d'avoir un système de gestion des transactions parfaitement fonctionnel.

Nous pourrions remplacer la chaîne de caractères présente dans notre objet *Block* par une liste d'objets *Transaction*. Cependant, nous devons pouvoir ajouter jusqu'à 1000 transactions au sein d'un bloc ce qui va poser problème pour le calcul du hash d'un bloc. Afin de résoudre ce problème, les développeurs de la Blockchain Bitcoin ont choisi de recourir à une structure de données bien spécifique nommée Arbre de Merkle (figure 1). Egalement connue sous le nom d'arbre de hachage, cette structure permet de vérifier l'intégrité des données qu'elle contient sans qu'il soit nécessaire de toutes les avoir au moment de la vérification.

Nous ajoutons une méthode utilitaire *getMerkleRoot* à notre classe *Utils* prenant en entrée une liste de transactions et retournant en sortie la racine de l'Arbre de Merkle correspondant au format chaîne de caractères :

```
public static String getMerkleRoot(ArrayList<Transaction> transactions) {
 int count = transactions.size();
 ArrayList<String> previousTreeLayer = new ArrayList<String>();

 for (Transaction transaction : transactions) {
  previousTreeLayer.add(transaction.transactionId);
 }

 ArrayList<String> treeLayer = previousTreeLayer;

 while (count > 1) {
  treeLayer = new ArrayList<String>();

  for (int i = 1; i < previousTreeLayer.size(); i++) {
   treeLayer.add(applySha256(previousTreeLayer.get(i - 1) + previousTreeLayer.get(i)));
  }

  count = treeLayer.size();
  previousTreeLayer = treeLayer;
 }
```



1 *Arbre de Merkle appliqué aux transactions*

```
 String merkleRoot = (treeLayer.size() == 1) ? treeLayer.get(0) : "";
 return merkleRoot;
}
```

La classe Block peut maintenant être mise à jour. Elle contient désormais une liste de transactions qui sera transformée sous la forme d'un Arbre de Merkle au moment où le bloc sera miné. On ajoute une méthode *addTransaction* tentant de traiter la transaction avant de l'ajouter au bloc si le traitement a réussi. Au niveau de la méthode de minage du bloc, on effectue une modification afin de convertir la liste des transactions sous la forme d'un Arbre de Merkle. On place la racine de cet arbre dans le champ data au format chaîne de caractères utilisé dans le hash du bloc.

Ceci nous donne le code mis à jour suivant pour la classe *Block* :

```
public class Block {

 // ...

 public ArrayList<Transaction> transactions = new ArrayList<Transaction>();

 // ...

 public void mineBlock(int difficulty) {
  String merkleRoot = Utils.getMerkleRoot(transactions);
  data = (!"".equals(merkleRoot)) ? merkleRoot : data;
  nonce = 0;

  while (!getHash().substring(0, difficulty).equals(Utils.zeros(difficulty))) {
```

```
    nonce++;
    hash = Block.calculateHash(this);
   }
  }

  public boolean addTransaction(Transaction transaction) {
   if (transaction == null)
    return false;

   if (previousHash != null) {
    if (!transaction.processTransaction()) {
     System.out.println("Transaction non valide. Ajout annulé");
     return false;
    }
   }

   transactions.add(transaction);
   System.out.println("Transaction ajoutée avec succès au bloc");

   return true;
  }
}
```

Notre Blockchain en Action

Notre Blockchain permet désormais la réalisation de transactions entre utilisateurs qui seront ici représentés par des instances de *Wallet*. Nous allons donc pouvoir la mettre en action mais avant toute chose, il est nécessaire d'introduire des fonds dans notre Blockchain avant de permettre aux utilisateurs d'effectuer ensuite des transactions. Nous réalisons cela à la création de notre Blockchain en envoyant 100 coins au Wallet A dans le bloc genèse. Il est important également de stocker cette première transaction dans la liste *UTXOs* des transactions non dépensées de notre classe *Blockchain*. Une fois la transaction créée, on l'ajoute au bloc genèse et on mine ce bloc avant de l'intégrer à la Blockchain :

```
import java.security.Security;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
public class Blockchain {

 public static float minimumTransaction = 0.1f;
 private int difficulty;
 private List<Block> blocks;
 public static HashMap<String, TransactionOutput> UTXOs = new HashMap<String,
TransactionOutput>();
 public Wallet walletA;
 public Wallet walletB;
 public Transaction genesisTransaction;

 public Blockchain(int difficulty) {
  Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
  // Création des Wallets
  walletA = new Wallet();
  walletB = new Wallet();
  Wallet baseWallet = new Wallet();

  // Transaction genèse
  genesisTransaction = new Transaction(baseWallet.publicKey, walletA.publicKey, 100f, null);
  // Signature de de la transaction
  genesisTransaction.generateSignature(baseWallet.privateKey);
  genesisTransaction.transactionId = "0";
  genesisTransaction.outputs.add(new TransactionOutput(genesisTransaction.recipient,
genesisTransaction.value, genesisTransaction.transactionId));
  // Ajout à la liste UTXOs des transactions non dépensées
  UTXOs.put(genesisTransaction.outputs.get(0).id, genesisTransaction.outputs.get(0));

  this.difficulty = difficulty;
  blocks = new ArrayList<>();
  // Création du bloc genèse
  Block b = new Block(0, System.currentTimeMillis(), null, "Genesis Block");
  b.transactions.add(genesisTransaction);
  b.mineBlock(difficulty);
  // Ajout du Bloc
  addBlock(b);
 }

 // ...
}
```

Il est également intéressant de modifier la méthode *isBlockChainValid* vérifiant la validité de notre Blockchain. On rajoute ainsi certains contrôles sur les transactions portées par chaque bloc : signature valide, somme des montants des transactions entrantes égale à la somme des transactions sortantes ou encore cohérence entre expéditeur et destinataire sur les transactions.

Finalement, on peut passer au test de notre Blockchain en lui assignant une difficulté de 4 pour le minage des blocs. On vérifie que le Wallet A se retrouve bien avec un montant de 100 coins à la création de la Blockchain. Ensuite, on réalise un premier envoi de 40 coins du Wallet A au Wallet B avant d'ajouter la transaction à notre Blockchain. C'est d'ailleurs durant cet ajout que le travail de minage est réalisé. On vérifie alors le montant contenu dans chacun des Wallets A et B.

Un bon test consiste ensuite à envoyer plus d'argent que ce qu'un Wallet contient. Notre Blockchain ayant été bien conçue, la transaction ne peut alors être créée. Les montants des Wallets n'ont donc pas changé. On termine par 2 nouvelles transactions entre le Wallet B et le Wallet A et inversement. A la fin de notre test, le Wallet A doit avoir un montant de 70 coins en balance et le Wallet B un montant de 30 coins ce qui est confirmé dans la figure 2.

Finalement, on affiche le contenu de notre Blockchain pour constater qu'on a bien 3 blocs et que les données de chacun des blocs correspondent à la racine de l'Arbre de Merkle des transactions portées par le bloc. Ces différentes opérations sont réalisées au sein du point d'entrée main de la classe Blockchain :

```
public static void main(String[] args) {
 Blockchain blockchain = new Blockchain(4);
 System.out.println("\nWallet A montant : " + blockchain.walletA.getBalance());
```

```
System.out.println("Wallet A essaie d'envoyer 40 au Wallet B ...");
Block b = blockchain.newBlock("Block 2");
b.addTransaction(blockchain.walletA.sendFunds(blockchain.walletB.publicKey, 40f));
blockchain.addBlock(b);

System.out.println("Wallet A montant : " + blockchain.walletA.getBalance());
System.out.println("Wallet B montant : " + blockchain.walletB.getBalance());

System.out.println("\nWallet A essaie d'envoyer plus (100) que ce qu'il possède ...");
b = blockchain.newBlock("Block 3");
b.addTransaction(blockchain.walletA.sendFunds(blockchain.walletB.publicKey, 1000f));
blockchain.addBlock(b);

System.out.println("Wallet A montant : " + blockchain.walletA.getBalance());
System.out.println("Wallet B montant : " + blockchain.walletB.getBalance());

System.out.println("\nWallet B essaie d'envoyer 20 au Wallet A ...");
System.out.println("Wallet A essaie d'envoyer 10 au Wallet B ...");
b = blockchain.newBlock("Block 4");
b.addTransaction(blockchain.walletB.sendFunds(blockchain.walletA.publicKey, 20f));
b.addTransaction(blockchain.walletA.sendFunds(blockchain.walletB.publicKey, 10f));
blockchain.addBlock(b);

System.out.println("Wallet A montant : " + blockchain.walletA.getBalance());
System.out.println("Wallet B montant : " + blockchain.walletB.getBalance());

System.out.println();
System.out.println(blockchain);
System.out.println("Blockchain valide : " + (blockchain.isBlockChainValid() ? "Oui" : "Non"));
}
```

```
Console
<terminated> Blockchain [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (26 juil. 2018 à 16:42:26)

Wallet A montant : 100.0
Wallet A essaie d'envoyer 40 au Wallet B ...
Transaction ajoutée avec succès au bloc

Wallet A montant : 60.0
Wallet B montant : 40.0

Wallet A essaie d'envoyer plus (100) que ce qu'il possède ...
Manque de fonds pour créer la transaction

Wallet A montant : 60.0
Wallet B montant : 40.0

Wallet B essaie d'envoyer 20 au Wallet A ...
Wallet A essaie d'envoyer 10 au Wallet B ...
Transaction ajoutée avec succès au bloc

Transaction ajoutée avec succès au bloc

Wallet A montant : 70.0
Wallet B montant : 30.0

Block #0 [previousHash : null, timestamp : Thu Jul 26 16:42:27 CEST 2018, data : 0, hash : 0000a6b5f9e
Block #1 [previousHash : 0000a6b5f9e245559cba113125ae6ec6d93fd651ee93d71fb756a8eb664656f4, timestamp :
Block #2 [previousHash : 0000f2c3581a25345e78aae0deabbac429c704be0d6d5660c80dcdf9851c5c59, timestamp :

Blockchain valide : Oui
```

2 *Notre Blockchain en Action*

Conclusion

La première version de notre Blockchain nous avait permis de mettre en avant le fonctionnement basique d'une chaîne de blocs stockant de simples messages au format chaînes de caractères. En l'état, elle n'avait que peu d'utilité. Dans cet article, nous avons vu comment notre Blockchain pouvait passer à la vitesse supérieure en intégrant un système complet de gestion des transactions avec en prime l'utilisation de clés publiques et privées pour sécuriser les transactions. C'est avec une Blockchain de ce type que le Bitcoin fonctionne et demeure la crypto monnaie de référence. Cependant, notre Blockchain ne fonctionne qu'en local pour le moment ce qui est plutôt limitant vous en conviendrez. Dans un futur article, nous verrons donc comment nous pouvons mettre en réseau P2P notre Blockchain. •