Contrôle Final 2 - CSC3101 - 18 Mars 2022 - 2h

La durée de cet examen est 2h. Tout document écrit est autorisé.

Une blockchain est une base de données sûre, répliquée entre un grand nombre de machines. L'objectif de cet examen est d'implémenter une blockchain simplifiée dans le langage Java. Tout comme Bitcoin, notre blockchain, appelée `CCoin`, permettra de faire des transactions entre deux comptes afin de transférer une somme d'argent. Un exemple d'utilisation de la blockchain `CCoin` est proposé à la fin du sujet.

1. Classes de base *(~30 minutes, 7pt)*

Les premières classes dont nous avons besoin sont `Account`, `Database` et `Transaction`. La classe `Account` possède trois champs, à savoir un identifiant (`int id`), un solde (`int balance`) et une paire de clés pour le chiffrement (`KeyPair key`). Par simplicité, on suppose que la classe `KeyPair` du paquetage `java.security` est déjà importée.

[Q1a] *(2pt)* Donnez le code de la classe `Account`. Ajoutez un constructeur à la classe prenant en entrée trois paramètres pour initialiser les champs. (*NB: Dans cet examen, la visibilité des champs et méthodes d'une classe sont ceux par défaut.*)

La classe `Database` contient un seul champ, une collection de comptes (`Collection<Account> accounts`).

[Q1b] *(2pt)* Donnez le code de `Database`. Le constructeur de cette classe prendra un seul argument, une collection de comptes afin d'initialiser `accounts`.

Une `Transaction` permet de matérialiser un mouvement d'argent entre deux comptes À cet effet, cette classe contient trois champs, `from` et `to` de type `Account`, et `int amount` qui contient la quantité d'argent transférée. Une transaction possède aussi un identifiant unique (`String id`) ainsi qu'une signature (`byte[] signature`). Par ailleurs, chaque transaction est signée à l'aide de la clé du compte `from`. Pour ce faire, on suppose désormais que la classe `Account` contient deux méthodes:

- `byte[] sign(byte[] value)` génère une signature à partir du tableau d'octets `value`
- `boolean verify(byte[] value, byte[] signature)` vérifie que la `signature` de `value` est correcte.

[Q1c] *(3pt)* Donnez le code de `Transaction`. Écrivez un constructeur qui prend en entrée la valeur des champs `from`, `to`, `amount` et `id`. Dans ce constructeur, utilisez la méthode `sign` pour signer la transaction. Cette méthode prend en paramètre l'identifiant de la transaction. Vous utiliserez la méthode `getBytes()` de la classe `String` afin de récupérer sous forme d'octets (`byte[]`) l'identifiant de la transaction.

2. Exécution des transactions *(~50 minutes, 10pt)*

Un livre de comptes (*ledger* en anglais) tient à jour les modifications qui ont été exécutées sur un ensemble de comptes. Ces modifications sont ordonnées dans une liste. Cette liste constitue la blockchain à proprement parler.

Quand une nouvelle transaction est ajoutée à la blockchain, ses modifications sont appliquées. Avant de faire cela, on doit vérifier que la transaction est légale, et donc bien signée par le compte dont on retire de l'argent. Cette logique est implémentée dans la classe `Ledger` ci-après.

[Q2a] *(2pt)* Créez la classe `Ledger`. Cette classe contient la base de données des comptes (`Database db`) et la liste de transactions (`List<Transaction> blockchain`). Ajoutez un constructeur à la classe. Ce constructeur prend en entrée un base de données et initialise la variable `blockchain` à une liste vide.

[Q2b] *(4pt)* Ajoutez une méthode `boolean isLegit(Transaction t)` à `Ledger` qui vérifie la légitimité d'une transaction `t`. Pour être légitime, `t` doit conjointement *(i)* accéder à des comptes qui sont dans la base de données, et *(ii)* être correctement signée. On rappelle que la méthode `contains(E e)` permet de vérifier si l'élément `e` est dans une collection. Par ailleurs, pour vérifier la signature de `t`, vous utiliserez la méthode `verify` de la classe `Account`.

[Q2c] *(4pt)* Ajouter une méthode `void execute(Transaction t)` à `Ledger` afin d'exécuter une transaction. Dans le cas où la transaction est invalide, la méthode doit retourner une exception `IllegalArgumentException`.

3. Sécuriser la blockchain *(~20 minutes, 3+2)*

Dans un système réparti, chaque machine de la blockchain exécute une instance de `Ledger`. Elle reçoit des transactions d'utilisateurs, vérifie leur légitimités puis les appliquent. Les machines peuvent ne pas être d'accord sur l'ordre d'exécution des transactions. L'approche utilisée par Bitcoin est de choisir la blockchain la plus longue, afin d'assurer la convergence à terme. Toutefois ce mécanisme ne permet pas d'éviter les abus. Ainsi, un utilisateur peut proposer une transaction `t`, puis avant que `t` ne soit appliquer "partout", proposer une transaction `t'` réutilisant l'argent dépensé dans `t`. Pour éviter ceci, nous devons tout comme Bitcoin demander à une machine qui ajoute une transaction à la blockchain de résoudre un puzzle. Notre blockchain `CCoin` utilisera pour cela une décomposition en facteurs premiers de l'indice du bloc courant, problème difficile pour de grandes valeurs d'indice.

[Q3a] *(3pt)* Créez une classe `CCoin` qui hérite de `Ledger`. A ce stade, cette classe inclut uniquement un constructeur.

[Q3b] *(2pt)* Dans `CCoin` sur-définissez la méthode `void execute(Transaction t)`. Cette nouvelle méthode doit d'abord calculer la décomposition en facteurs premiers de `blockchain.size()`. Pour ce faire, elle utilise une méthode `List<Integer> primeFactors(int number)` de `CCoin` supposée déjà là. Une fois ce calcul fait, la transaction est exécutée via la méthode `execute` de `Ledger`.

Exemple d'utilisation

```
class Main{
    static int ACCOUNTS=10;
    static int TRANSACTIONS=10;
    public static void main(String[] args){

        // 1 - create ledger
        Collection<Account> accounts = new ArrayList<Account>();
        for(int i = 0; i<ACCOUNTS; i++){
            accounts.add(new Account(i,100,newKeyPair()));
        }
        Database db = new Database(accounts);
        Ledger ledger = new CCoin(db);

        // 2 - run transactions
        System.out.println(ledger);
        for(int i = 0; i<TRANSACTIONS; i++){
            List<Account> shuffled  = new ArrayList<>(db.accounts);
            Collections.shuffle(shuffled);
            Account from = shuffled.get(1);
            Account to = shuffled.get(2);
            Transaction t = new Transaction(Integer.toString(i),from,to,1);
            ledger.execute(t);
        }

        // 3 - check its content
        System.out.println(ledger);
    }

    static KeyPair newKeyPair(){
        KeyPair ret = null;
        try{
            KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
            kpg.initialize(2048);
            ret = kpg.generateKeyPair();
        }catch(Exception e){} // ignore
    return ret;
    }
}
```