MidTerm di Programmazione Avanzata

Francesco Landolfi

15 aprile 2017

1 Esercizio 1

`Hash` Questa classe (Listing 1) ha come attributi `hash`, un array di `byte` di 32 elementi, e `SHA256`, un `MessageDigest` statico, generalmente istanziato con algoritmo SHA-256. Un oggetto può essere creato con un costruttore senza argomenti, il quale inizializza a 0 gli elementi di `hash`, o con una lista di `Object`. In tal caso `hash` viene inizializzato con il digest prodotto da `SHA256` utilizzando la concatenazione degli hashcode degli argomenti del costruttore. Sono stati aggiunti i metodi `hashcode()`, per poter utilizzare correttamente un oggetto `Hash` come chiave in una `HashMap`, e `getShortHash()`, che restituisce un'abbreviazione leggibile di `hash` in un oggetto `String`.

`Transaction` Con questa classe (Listing 3) è possibile istanziare transazioni o *coinbase* (ovvero transazioni senza input) mediante il metodo statico `createCoinbase()`. In entrambi i casi vengono effettuati controlli di consistenza sugli argomenti dei costruttori per verificare che: 1. le chiavi del firmatario siano valide (utilizzando il metodo `checkKeyPair()`); 2. l'importo trasferito sia valido e coperto; 3. il firmatario usi denaro a lui intestato.

`Block` Con questa classe (Listing 6) è possibile creare blocchi destinati ad una block chain. Il metodo privato `rehash()` ricalcola l'hash del blocco chiamante. Viene utilizzato ogni volta che viene apportata una modifica al valore di un suo attributo e, in particolare, nel metodo `mine()`, il quale incrementa il valore di `nonce` ed esegue `rehash()` finché l'hash prodotto non inizia con tre 0.

2 Esercizio 2

`BlockChain` Con questa classe (Listing 7) è possibile creare una *block chain*. Il costruttore genera un blocco iniziale (chiamato anche *genesis block*) contenente una *coinbase* e l'hash del blocco precedente inizializzato a 0. Il metodo `isValidBlockChain()` controlla che ciascun blocco: 1. abbia un hash valido (ovvero che inizi con tre 0); 2. contenga l'hash del blocco precedente valido; 3. non contenga transazioni che utilizzino un importo già esaurito (ovvero, che non si verifichi un *double payment*). Quest'ultimo controllo viene eseguito scorrendo la block chain in maniera ordinata e, per ciascun blocco, viene salvato un `BitSet` con dimensione uguale al numero di output contenuti nella transazione del blocco. Se uno di questi output venisse speso in una transazione successiva, il bit di indice corrispondente a quell'output verrà impostato ad 1. Se una transazione dovesse spendere un output con bit già ad 1, il metodo ritorna `false`. Il controllo del valore delle transazioni non viene effettuato perché il design della classe `Transaction` non permette la creazione di transazioni con valori non consistenti. Il metodo `getBalance()`[1] restituisce l'importo totale posseduto da un dato utente (identificato da una `PublicKey`). Il calcolo di questo importo viene effettuato scorrendo a ritroso la block chain e, per ogni transazione contenente tra gli output la `PublicKey` dell'utente, l'importo a lui erogato viene sommato al totale. Questo metodo tiene conto degli output già spesi in maniera analoga al metodo precedente (utilizzando un `BitSet` per ogni transazione). Il metodo `mine()` esegue il metodo omonimo su un determinato blocco. Per ogni blocco successivo, viene aggiornato l'hash del blocco precedente e del blocco stesso.

3 Esercizio 3

`MidTerm` Questa classe (Listing 8) contiene il `main()` del progetto. Questo utilizza il metodo `init()` per generare una coppia di chiavi RSA per tre utenti fittizi, Alice, Bob e Carol, e ad ogni chiave associa un *alias* (ad esempio, la chiave privata e la chiave pubblica di Alice sono rispettivamente "SkA" e "PkA"). Questi vengono

[1] Ho modificato il nome di questo metodo da `printBalance()` a `getBalance()`, il quale non stampa a schermo ma restituisce un oggetto `String`. L'importo verrà poi stampato a schermo nel metodo `main()` della classe `MidTerm`.

utilizzati per ricavare le vere chiavi tramite una `HashMap`, mentre un'altra `HashMap` viene utilizzata per ottenere l'alias da una chiave. Il metodo termina generando la block chain iniziale, composta da un solo blocco (*non valido* – il *mining* dei blocchi deve essere eseguito sempre manualmente) contenente una coinbase di importo uguale a 10000 per Alice. Il metodo `main()` prosegue stampando a schermo la lista degli utenti con le loro chiavi, utilizzando il metodo `printUsers()`, e lo stato attuale della block chain, con il metodo `printBlockChain()`[2]. Successivamente attende l'inserimento di un comando da input. Oltre ai comandi che sono nella specifica del progetto, sono supportati anche i comandi `help`, che, tramite il metodo `printHelp()`, stampa a schermo i comandi supportati e come è possibile utilizzarli, `users`, che esegue il metodo `printUsers()`, e `status`, che esegue il metodo `printBlockChain()`. Gli altri comandi eseguono funzioni già offerte dalle altre classi ma alcuni di questi (ovvero `report`, `mine` e `transfer`) devono prima effettuare il parsing degli argomenti per poterle applicare.

Esempio di esecuzione

Partendo dalla block chain iniziale, effettuiamo una nuova transazione dall'output 0 del blocco 0 (ovvero dalla coinbase di Alice) per un importo di 10000 da destinarsi a Bob, eseguiamo una `mine` sul primo e sul secondo blocco, e controlliamo lo stato della block chain:

```
$> transfer --in=0,0 --out=10000,PkB --sign=SkA,PkA
< omissis >
$> mine 0
Found nonce value 8214.
< omissis >
$> mine 1
Found nonce value 7865.
< omissis >
$> check
The block chain is valid!
```

Adesso eseguiamo un'altra transazione, sempre dal primo blocco, verso Carol:

```
$> transfer --in=0,0 --out=5000,PkC --sign=SkA,PkA
< omissis >
$> mine 2
Found nonce value 1546.
< omissis >
$> check
The block chain is not valid.
```

A causa di un double payment la block chain non è più valida. Annulliamo quindi la transazione rimuovendo l'ultimo blocco:

```
$> remove
Last block removed.
< omissis >
$> check
The block chain is valid!
```

4 Esercizio 4

Una chiusura (*closure*) è una struttura dati che contiene un riferimento ad una funzione insieme al suo *ambiente lessicale*, ovvero all'insieme dei binding delle variabili (locali e non locali) nello scope della funzione. Questo permette di trattare le funzioni come oggetti di prima classe (*first-class objects*) e quindi associarle ad una variabile e ad effettuare chiamate alle funzioni anche al di fuori del contesto in cui sono state associate.

In C#, i `delegate` sono oggetti che rappresentano un insieme modificabile (anche vuoto) di chiusure di funzioni aventi tutte la stessa *firma*, ovvero il numero dei parametri, il tipo dei parametri, e il tipo di ritorno. Il `delegate` potrà poi essere utilizzato come un metodo, effettuando così una chiamata a ciascuna funzione ad esso associata, passando come loro argomenti gli stessi utilizzati dal `delegate`[3].

[2] Per migliorare la leggibilità dell'output, la block chain verrà stampata a schermo solamente dopo l'esecuzione di un comando che ne modifica il contenuto (ovvero dopo `transfer`, `mine` o `remove`).

[3] Riferimenti: Wikipedia, MSDN Microsoft.

5 Codice

Il progetto è stato implementato in Java™ utilizzando JDK 8 (Oracle®).

Listing 1: Hash.java

```
import java.nio.*;
import java.security.*;

public class Hash {
  private static MessageDigest SHA256 = null;
  private final byte[] hash;

  public Hash(Object... objects) {
    if (SHA256 == null)
      try {
        SHA256 = MessageDigest.getInstance("SHA-256");
      } catch (NoSuchAlgorithmException ignored) {}

    ByteBuffer buffer = ByteBuffer.allocate(objects.length*4);

    for (Object obj : objects) {
      buffer.putInt(obj.hashCode());
    }

    hash = SHA256.digest(buffer.array());
  }

  public Hash() { hash = new byte[32]; }
  public boolean isValid() { return (hash[0] | (hash[1] & 0xF0)) == 0; }
  public byte[] getHash(){ return hash; }

  public String getShortHash() {
    return String.format("%02x%02x%02x...%02x%02x%02x",
          hash[0], hash[1], hash[2], hash[29], hash[30], hash[31]);
  }

  @Override
  public int hashCode() {
    int code = 23;

    for (byte b : hash)
      code = 43*code + (int) b;

    return code;
  }

  @Override
  public boolean equals(Object obj) { return obj != null && obj instanceof Hash
    && equals(((Hash) obj).getHash()); }
  public boolean equals(byte[] hash) { return MessageDigest.isEqual(this.hash,
    hash); }
}
```

Listing 2: Entry.java

```
public class Entry<F, S> {
  final private F first;
  final private S second;

  public Entry(F first, S second) { this.first = first; this.second = second; }
  public F getFst() { return first; }
  public S getSnd() { return second; }
}
```

Listing 3: Transaction.java

```
import java.security.*;
import java.util.*;

public class Transaction {
  private ArrayList<Entry<Hash, Integer>> inputs;
```

```
  private ArrayList<Entry<Integer, PublicKey>> outputs;
  private byte[] signature;

  public static Transaction createCoinbase(PrivateKey signer, PublicKey payee,
                       int amount) throws InvalidKeyException {
    if (amount <= 0)
      throw new IllegalArgumentException();

    Transaction coinbase = new Transaction();
    coinbase.outputs.add(new Entry<>(amount, payee));

    try {
      Signature signature = Signature.getInstance("SHA256withRSA");
      signature.initSign(signer);
      signature.update((coinbase.inputs.toString() +
          coinbase.outputs.toString()).getBytes());
      coinbase.signature = signature.sign();
    } catch (NoSuchAlgorithmException | SignatureException ignored) {}

    return coinbase;
  }

  private Transaction() {
    this.inputs = new ArrayList<>();
    this.outputs = new ArrayList<>();
  }

  public Transaction(ArrayList<Entry<Transaction, Integer>> inputs,
            ArrayList<Entry<Integer, PublicKey>> outputs, KeyPair payer)
      throws AmountExceededException, InvalidKeyException,
      UnauthorizedTransactionException {
    this();
    checkKeyPair(payer);
    Integer total = 0;

    for (Entry<Transaction, Integer> in : inputs) {
      Entry<Integer, PublicKey> out = in.getFst().getOutputs().get(in.getSnd());

      if (!out.getSnd().equals(payer.getPublic()))
        throw new UnauthorizedTransactionException();

      this.inputs.add(new Entry<>(new Hash(in.getFst()), in.getSnd()));
      total += out.getFst();
    }

    for (Entry<Integer, PublicKey> e : outputs) {
      if (e.getFst() < 0)
        throw new UnauthorizedTransactionException();

      if((total -= e.getFst()) < 0)
        throw new AmountExceededException();

      this.outputs.add(e);
    }

    if (total > 0)
      this.outputs.add(new Entry<>(total, payer.getPublic()));

    try {
      Signature signature = Signature.getInstance("SHA256withRSA");
      signature.initSign(payer.getPrivate());
      signature.update((this.inputs.toString() +
          this.outputs.toString()).getBytes());
      this.signature = signature.sign();
    } catch (NoSuchAlgorithmException | SignatureException ignored) {}
  }

  private void checkKeyPair(KeyPair keys) throws InvalidKeyException {
    try {
      SecureRandom random = new SecureRandom();
      byte[] randomBytes = new byte[32];
      random.nextBytes(randomBytes);
      Signature signature = Signature.getInstance("SHA256withRSA");
```

```
      signature.initSign(keys.getPrivate());
      signature.update(randomBytes);
      byte[] sigBytes = signature.sign();
      signature.initVerify(keys.getPublic());
      signature.update(randomBytes);

      if (!signature.verify(sigBytes))
        throw new InvalidKeyException();
    } catch (NoSuchAlgorithmException | SignatureException ignored) {}
  }

  public ArrayList<Entry<Hash, Integer>> getInputs() { return inputs; }
  public ArrayList<Entry<Integer, PublicKey>> getOutputs() { return outputs; }
}
```

Listing 4: AmountExceededException.java

```
public class AmountExceededException extends Exception {}
```

Listing 5: UnauthorizedTransactionException.java

```
public class UnauthorizedTransactionException extends Exception {}
```

Listing 6: Block.java

```
import java.security.*;

public class Block {
  private int number;
  private int nonce = 0;
  private Hash previous, current;
  private Transaction data;

  public Block(int number, Hash previous, Transaction data) {
    this.number = number;
    this.data = data;
    this.previous = previous;
    rehash();
  }

  private void rehash() { current = new Hash(number, nonce, previous, data); }
  public int getNumber() { return number; }
  public int getNonce() { return nonce; }
  public void setPreviousHash(Hash hash) { previous = hash; rehash(); }
  public Hash getPreviousHash() { return previous; }
  public Hash getCurrentHash() { return current; }
  public Transaction getData() { return data; }

  public int mine() {
    while (!current.isValid()) {
      nonce++;
      rehash();
    }

    return nonce;
  }
}
```

Listing 7: BlockChain.java

```
import java.security.*;
import java.util.*;

public class BlockChain {
  private LinkedList<Block> chain = new LinkedList<>();

  public BlockChain(PrivateKey signer, PublicKey payee, int amount)
      throws InvalidKeyException { chain.add(new Block(0, new Hash(),
        Transaction.createCoinbase(signer, payee, amount))); }

  public boolean isValidBlockChain() {
```

```
    HashMap<Hash, BitSet> spentTransactions = new HashMap<>();
    Iterator<Block> it = chain.iterator();
    Block b = it.next();
    Transaction tx = b.getData();

    if (!b.getCurrentHash().isValid())
      return false;

    spentTransactions.put(new Hash(tx), new BitSet(tx.getOutputs().size()));

    while (it.hasNext()) {
      Hash previous = b.getCurrentHash();
      b = it.next();
      tx = b.getData();

      if (!b.getCurrentHash().isValid() ||
          !previous.equals(b.getPreviousHash()))
        return false;

      for (Entry<Hash, Integer> input : tx.getInputs()) {
        if (spentTransactions.computeIfPresent(input.getFst(), (k, v) -> {
          if (v.get(input.getSnd()))
            return null;

          v.set(input.getSnd());

          return v;
        }) == null)
          return false;
      }

      spentTransactions.put(new Hash(tx), new BitSet(tx.getOutputs().size()));
    }

    return true;
  }

  public int getBalance(PublicKey user) {
    HashMap<Hash, BitSet> spentTransactions = new HashMap<>();
    Iterator<Block> it = chain.descendingIterator();
    int balance = 0;

    while (it.hasNext()) {
      Block b = it.next();
      Transaction tx = b.getData();
      Hash hash = new Hash(tx);
      ArrayList<Entry<Integer, PublicKey>> outputs = tx.getOutputs();

      for (int i = 0; i < outputs.size(); i++)
        if (user.equals(outputs.get(i).getSnd()) &&
            (spentTransactions.get(hash) == null ||
            !spentTransactions.get(hash).get(i)))
          balance += outputs.get(i).getFst();

      tx.getInputs().forEach(in -> spentTransactions
          .computeIfAbsent(in.getFst(), v -> new BitSet()).set(in.getSnd()));
    }

    return balance;
  }

  public int mine(int index) {
    int nonce = chain.get(index).mine();

    for (int i = index + 1; i < chain.size(); i++)
      chain.get(i).setPreviousHash(chain.get(i - 1).getCurrentHash());

    return nonce;
  }

  public void addBlock(Transaction transaction) { chain.add(new
    Block(chain.size(), chain.getLast().getCurrentHash(), transaction)); }
  public boolean removeLast() { return chain.size() > 1 && chain.removeLast()
    != null; }
```

```
  public LinkedList<Block> getBlockChain() { return chain; }
}
```

Listing 8: MidTerm.java

```
import java.security.*;
import java.util.*;

public class MidTerm {
  private static KeyPairGenerator kpg = null;
  private static KeyPair a, b, c;
  private static BlockChain chain;
  private static HashMap<Key, String> aliases = new HashMap<>();
  private static HashMap<String, Key> keyring = new HashMap<>();

  private static void init() {
    try {
      kpg = KeyPairGenerator.getInstance("RSA");
      kpg.initialize(1024);
      a = kpg.generateKeyPair();
      b = kpg.generateKeyPair();
      c = kpg.generateKeyPair();
      aliases.put(a.getPrivate(), "SkA");
      aliases.put(a.getPublic(), "PkA");
      aliases.put(b.getPrivate(), "SkB");
      aliases.put(b.getPublic(), "PkB");
      aliases.put(c.getPrivate(), "SkC");
      aliases.put(c.getPublic(), "PkC");
      aliases.forEach((k, v) -> keyring.put(v, k));
      chain = new BlockChain(b.getPrivate(), a.getPublic(), 10000);
    } catch (NoSuchAlgorithmException | InvalidKeyException ignored) {}
  }

  private static void printUsers() {
    System.out.println("\nUsers:\n======\n" +
        "  Alice:\tPrivate Key: SkA\n        \tPublic Key: PkA\n\n" +
        "  Bob:  \tPrivate Key: SkB\n        \tPublic Key: PkB\n\n" +
        "  Carol:\tPrivate Key: SkC\n        \tPublic Key: PkC\n");
  }

  private static void printBlockChain() {
    System.out.println("\nCurrent BlockChain:\n" +
        "==================\n");

    for (Block b : chain.getBlockChain()) {
      System.out.print("  Block " + b.getNumber() + ":\tInputs:\n");

      if (b.getData().getInputs().size() == 0)
        System.out.println("          \t   None.");
      else {
        ArrayList<Entry<Hash, Integer>> inputs = b.getData().getInputs();

        for (int i = 0; i < inputs.size(); i++) {
          Entry<Hash, Integer> in = inputs.get(i);
          System.out.println("          \t   " + i + ". Hash: " +
              in.getFst().getShortHash() + ", Index: " + in.getSnd());
        }
      }

      ArrayList<Entry<Integer, PublicKey>> outputs = b.getData().getOutputs();
      System.out.println("          \tOutputs:");

      for (int i = 0; i < outputs.size(); i++) {
        Entry<Integer, PublicKey> out = outputs.get(i);
        System.out.println("          \t   " + i + ". Amount: " + out.getFst() +
            ", Payee: " + aliases.get(out.getSnd()));
      }

      System.out.println("          \tNonce value:   " + b.getNonce() +
          "\n          \tCurrent Hash:  " + b.getCurrentHash().getShortHash() +
          "\n          \tPrevious Hash: " + b.getPreviousHash().getShortHash() +
          "\n");
    }
  }
```

```
70
71  private static void printHelp() {
72    System.out.println("\nCOMMANDS\n========\n\n" +
73        "  help\n\t\tShow the list of possible commands.\n\n" +
74        "  users\n\t\tShow the list of users with their RSA key aliases.\n\n" +
75        "  status\n\t\tShow the current block chain.\n\n" +
76        "  mine INDEX\n\t\tDiscovers the nonce number of the block of index " +
77        "INDEX.\n\n" +
78        "  transfer --in=BLOCK,OUTPUT --out=AMOUNT,PAYEE --sign=PRIVATE," +
79        "PUBLIC\n\t\tPerforms a transaction from output number OUTPUT of " +
80        "block number\n\t\tBLOCK to the beneficiary with public key PAYEE " +
81        "with the amount\n\t\tof AMOUNT, then signs the transaction with " +
82        "the key pair\n\t\tPRIVATE,PUBLIC. All switches are mandatory and " +
83        "'--in' and '--out'\n\t\tmay be reused.\n\n" +
84        "  remove\n\t\tRemoves the last block of the chain.\n\n" +
85        "  check\n\t\tChecks that the chain is valid.\n\n" +
86        "  report PUBLIC_KEY\n\t\tReports the balance of the user having " +
87        "the public key PUBLIC_KEY.\n\n" +
88        "  quit\n\t\tQuits the program.\n\n");
89  }
90
91  public static void main(String args[]) {
92    String input;
93    String[] tokens;
94    Scanner reader = new Scanner(System.in);
95    init();
96    printUsers();
97    printBlockChain();
98    System.out.println("Type 'help' for a list of commands.\n");
99
100   do {
101     do {
102       System.out.print("$> ");
103       input = reader.nextLine().trim();
104     } while (input.equals(""));
105
106     tokens = input.split("\\s+");
107
108     switch (tokens[0]) {
109       case "quit":
110         break;
111
112       case "help":
113         printHelp();
114         break;
115
116       case "users":
117         printUsers();
118         break;
119
120       case "status":
121         printBlockChain();
122         break;
123
124       case "check":
125         if (chain.isValidBlockChain())
126           System.out.println("The block chain is valid!");
127         else
128           System.out.println("The block chain is not valid.");
129
130         break;
131
132       case "remove":
133         if (!chain.removeLast())
134           System.err.println("Error: you can not remove the first block.");
135         else {
136           System.out.println("Last block removed.");
137           printBlockChain();
138         }
139
140         break;
141
142       case "report":
143         if (tokens.length < 2)
```

```
144             System.err.println("Error: missing argument.");
145           else {
146             Key key = keyring.get(tokens[1]);
147
148             if (key == null)
149               System.out.println("User '" + tokens[1] + "' not found.");
150             else
151               System.out.println("Balance: " +
152                   chain.getBalance((PublicKey) key));
153           }
154
155           break;
156
157         case "mine":
158           if (tokens.length < 2)
159             System.err.println("Error: missing argument");
160           else
161             try {
162               int nonce, index = Integer.parseInt(tokens[1]);
163
164               nonce = chain.mine(index);
165               System.out.println("Found nonce value " + nonce + ".");
166               printBlockChain();
167             } catch (NumberFormatException e) {
168               System.err.println("Error: invalid block index.");
169             } catch (IndexOutOfBoundsException e) {
170               System.err.println("Error: index out of bounds.");
171             }
172
173           break;
174
175         case "transfer":
176           ArrayList<Entry<Transaction, Integer>> inputs = new ArrayList<>();
177           ArrayList<Entry<Integer, PublicKey>> outputs = new ArrayList<>();
178           KeyPair signer = null;
179           boolean error = false;
180
181           loop: for (int i = 1; i < tokens.length; i++) {
182             String[] entry = tokens[i].split("[=,]");
183
184             if (entry.length != 3) {
185               System.err.println("Error: invalid argument.");
186               error = true;
187               break;
188             }
189
190             switch (entry[0]) {
191               case "--in":
192                 try {
193                   inputs.add(new Entry<>(chain.getBlockChain()
194                       .get(Integer.parseInt(entry[1]))
195                       .getData(), Integer.parseInt(entry[2])));
196                 } catch (NumberFormatException e) {
197                   System.err.println("Error: invalid index.");
198                   error = true;
199                   break loop;
200                 } catch (IndexOutOfBoundsException e) {
201                   System.err.println("Error: index out of bounds.");
202                   error = true;
203                   break loop;
204                 }
205
206                 break;
207
208               case "--out":
209                 try {
210                   Key key = keyring.get(entry[2]);
211
212                   if (key == null) {
213                     System.err.println("Error: key '" + entry[2] +
214                         "' not found.");
215                     error = true;
216                     break loop;
217                   }
```

```
                    outputs.add(new Entry<>(Integer.parseInt(entry[1]),
                        (PublicKey) key));
                  } catch (NumberFormatException e) {
                    System.err.println("Error: invalid amount.");
                    error = true;
                    break loop;
                  }

                  break;

                case "--sign":
                  if (signer != null) {
                    System.err.println("Error: a transaction must have " +
                        "only one signer.");
                    error = true;
                    break loop;
                  }

                  PrivateKey sk = (PrivateKey) keyring.get(entry[1]);
                  PublicKey pk = (PublicKey) keyring.get(entry[2]);

                  if (sk == null || pk == null) {
                    System.err.println("Error: invalid key pair.");
                    error = true;
                    break loop;
                  }

                  signer = new KeyPair(pk, sk);
                  break;

                default:
                  System.err.println("Error: invalid argument.");
                  error = true;
                  break loop;
              }
            }

            if (!error && signer != null && outputs.size() > 0
                && inputs.size() > 0) {
              try {
                chain.addBlock(new Transaction(inputs, outputs, signer));
              } catch (AmountExceededException e) {
                System.err.println("Error: amount exceeded.");
              } catch (InvalidKeyException e) {
                System.err.println("Error: Invalid key.");
              } catch (UnauthorizedTransactionException e) {
                System.err.println("Error: unauthorized transaction.");
              }

              printBlockChain();
            }

            break;

          default:
            System.err.println("Command '" + tokens[0] + "' not found.");
            break;
        }
      } while(!tokens[0].equals("quit"));
    }
}
```