School of Computer Engineering and Science SHU
Blockchain Technology and Data Economics
(08696017 1000)

Practical Report

Practice 6
ScroogeCoin

Name: Lin Yijun ID: 18120189

Date: 13 May. 2020

Contents

1 Introduction

In ScroogeCoin (Lecture 2), the central authority Scrooge receives transactions from users. You will implement the logic used by Scrooge to process transactions and produce the ledger. Scrooge organizes transactions into time periods or blocks. In each block, Scrooge will receive a list of transactions, validate the transactions he receives, and publish a list of validated transactions.

Note that a transaction can reference another in the same block. Also, among the transactions received by Scrooge in a single block, more than one transaction may spend the same output. This would of course be a double-spend, and hence invalid. This means that transactions can't be validated in isolation; it is a tricky problem to choose a subset of transactions that are together valid.

You will be provided with a ***Transaction*** class that represents a ScroogeCoin transaction and has inner classes ***Transaction.Output*** and ***Transaction.Input*** .

A transaction output consists of a value and a public key to which it is being paid. For the public keys, we use the built-in Java ***PublicKey*** class.

A transaction input consists of the hash of the transaction that contains the corresponding output, the index of this output in that transaction (indices are simply integers starting from 0), and a digital signature. For the input to be valid, the signature it contains must be a valid signature over the current transaction with the public key in the spent output.

More specifically, the raw data that is signed is obtained from the ***getRawDataToSign(int index)*** method. To verify a signature, you will use the ***verifySignature()*** method included in the provided file Crypto.java:

```
public static boolean verifySignature(\textbf{\textit{PublicKey}} pubKey, byte
    [] message,byte[] signature)
```

This method takes a public key, a message and a signature, and returns true if and only signature correctly verifies over message with the public key pubKey .

Note that you are only given code to verify signatures, and this is all that you will need for this assignment. The computation of signatures is done outside the Transaction class by an entity that knows the appropriate private keys.

A transaction consists of a list of inputs, a list of outputs and a unique ID (see the **getRawTx()** method). The class also contains methods to add and remove an input, add an output, compute digests to sign/hash, add a signature to an input, and compute and store the hash of the transaction once all inputs/outputs/signatures have been added.

You will also be provided with a **UTXO** class that represents an unspent transaction output. A **UTXO** contains the hash of the transaction from which it originates as well as its index within that transaction. We have included equals , hashCode , and compareTo functions in **UTXO** that allow the testing of equality and comparison between two **UTXO**s based on their indices and the contents of their **txHash** arrays.

Further, you will be provided with a **UTXOPool** class that represents the current set of outstanding **UTXO** s and contains a map from each **UTXO** to its corresponding transaction output. This class contains constructors to create a new empty **UTXOPool** or a copy of a given **UTXOPool**, and methods to add and remove **UTXO** s from the pool, get the output corresponding to a given **UTXO**, check if a **UTXO** is in the pool, and get a list of all **UTXO** s in the pool. You will be responsible for creating a file called TxHandler.java and API is provided with frame code in the zip pack.

Your implementation of **handleTxs()** should return a mutually valid transaction set of maximal size (one that can't be enlarged simply by adding more transactions). It need not compute a set of maximum size (one for which there is no larger mutually valid transaction set).

Based on the transactions it has chosen to accept, **handleTxs** should also update its internal **UTXOPool** to reflect the current set of unspent transaction outputs, so that future calls to **handleTxs()** and **isValidTx()** are able to correctly process/validate transactions that claim outputs from transactions that were accepted in a previous call to **handleTxs()**.

Extra Credit: Create a second file called MaxFeeTxHandler.java whose **handleTxs()** method finds a set of transactions with maximum total transaction fees – i.e. maximize the sum over all transactions in the set of (sum of input values - sum of output values).

2 Materials and Methods

2.1 Re-Understand Scrooge Coin

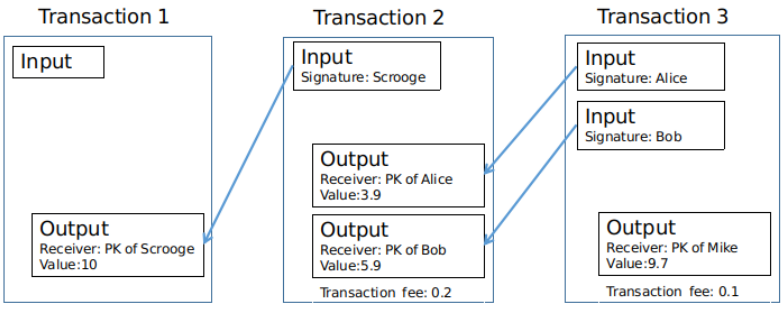

Figure 1: A clear figure shows Scroogr Coin.

This is a figure that helps me understand Scrooge Coin deeper.

Every transaction has a set of inputs and a set of outputs. An input in a transaction must use a hash pointer to refer to its corresponding output in the previous transaction, and it must be signed with the private key of the owner because the owner needs to prove he/she agrees to spend his/her coins.

Every output is correlated to the public key of the receiver, which is his/her ScroogeCoin address.

In the first transaction, we assume that Scrooge has created 10 coins and assigned them to himself, we don't doubt that because the system-Scroogecoin has a building rule which says that Scrooge has right to create coins.

In the second transaction, Scrooge transferred 3.9 coins to Alice and 5.9 coins to Bob. The sum of the two outputs is 0.2 less than the input because the transaction fee was 0.2 coin.

In the third transaction, there were two inputs and one output, Alice and Bob transferred 9.7 coins to mike, and the transaction fee was 0.1 coin.

2.2 TxHandler

2.2.1 Included Class

```
import java.util.HashSet;
import java.util.Set;
```

2.2.2 Completed Code with explaination

```
public class TxHandler {
    // current UTXOPool
    private UTXOPool utxoPool;
    /**
     * Creates a public ledger whose current UTXOPool (collection of unspent
         transaction outputs) is
     * {@code utxoPool}. This should make a copy of utxoPool by using the
         UTXOPool(UTXOPool uPool)
     * constructor.
     */
    public TxHandler(UTXOPool utxoPool) {
        this.utxoPool = new UTXOPool(utxoPool);
    }
```

```java
/**
 * @return true if:
 * (1) all outputs claimed by {@code tx} are in the current UTXO pool,
 * (2) the signatures on each input of {@code tx} are valid,
 * (3) no UTXO is claimed multiple times by {@code tx},
 * (4) all of {@code tx}s output values are non-negative, and
 * (5) the sum of {@code tx}s input values is greater than or equal to the
     sum of its output
 *     values; and false otherwise.
 */
public boolean isValidTx(Transaction tx) {
    UTXOPool uniqueUTXOs = new UTXOPool();
    double previousTxOutSum = 0;
    double currentTxOutSum = 0;
    // iterate over all inputs of transaction tx
    for (int i = 0; i < tx.numInputs(); i++) {
        Transaction.Input in = tx.getInput(i);
        UTXO utxo = new UTXO(in.prevTxHash, in.outputIndex);
        // previous output of input transaction
        Transaction.Output output = utxoPool.getTxOutput(utxo);
        // check if the transaction is in current UTXOPool
        if (!Crypto.verifySignature(output.address, tx.getRawDataToSign(i)
            , in.signature))
            return false;
        // check if the transaction is claimed only once
        if (uniqueUTXOs.contains(utxo))
            return false;
        // add transaction to uniqueUTXOs
        uniqueUTXOs.addUTXO(utxo, output);
        // add previous output value i.e. input value
        previousTxOutSum += output.value;
    }
    // iterate over all transactions output
    for (Transaction.Output out : tx.getOutputs()) {
        // check if the output value are non-negative
        if (out.value < 0)
            return false;
        // add current output value
        currentTxOutSum += out.value;
    }
    // check if the sum of the input value is less than sum of output
       value
    return previousTxOutSum >= currentTxOutSum;
}
/**
 * Handles each epoch by receiving an unordered array of proposed
    transactions, checking each
 * transaction for correctness, returning a mutually valid array of
    accepted transactions, and
 * updating the current UTXO pool as appropriate.
 */
public Transaction[] handleTxs(Transaction[] possibleTxs) {
```

```
        // set to store all the valid transactions
        Set<Transaction> validTxs = new HashSet<Transaction>();
        // iterate over all possible transactions
        for (Transaction tx : possibleTxs) {
            // check if the transactions are valid
            if (isValidTx(tx)) {
                // add transaction to set of valid transaction
                validTxs.add(tx);
                // iterate over inputs of transactions to remove them from
                    xurrent UTXOPool
                for (Transaction.Input in : tx.getInputs()) {
                    UTXO utxo = new UTXO(in.prevTxHash, in.outputIndex);
                    utxoPool.removeUTXO(utxo);
                }
                // iterate over outputs of transactions to add them to current
                     UTXOPool
                for (int i = 0; i < tx.numOutputs(); i++) {
                    Transaction.Output out = tx.getOutput(i);
                    UTXO utxo = new UTXO(tx.getHash(), i);
                    utxoPool.addUTXO(utxo, out);
                }
            }
        }
        // convert set of transactions to array of transactions
        Transaction[] validTxArray = new Transaction[validTxs.size()];
        return validTxs.toArray(validTxArray);
    }
}
```

2.3 MaxFeeTxHandler

2.3.1 Included Class

```
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;
import java.util.Collections;
```

2.3.2 Modified Code based on TxHandler

```
/**
 * Caculate Tx Fee
 * @param tx
 * @return sumInput - sumOutput
 */
private double TxFee(Transaction tx) {
    double sumInput = 0;
    double sumOutput = 0;
    for (Transaction.Input in : tx.getInputs()) {
        UTXO utxo = new UTXO(in.prevTxHash, in.outputIndex);
        if (!utxoPool.contains(utxo) || !isValidTx(tx)) continue;
        Transaction.Output txOutput = utxoPool.getTxOutput(utxo);
        sumInput += txOutput.value;
```

```
        }
        for (Transaction.Output out : tx.getOutputs()) {
            sumOutput += out.value;
        }
        return sumInput - sumOutput;
    }
    /**
     * Handles each epoch by receiving an unordered array of proposed transactions
        , checking each
     * transaction for correctness, returning a mutually valid array of accepted
        transactions, and
     * updating the current UTXO pool as appropriate.
     */
    public Transaction[] handleTxs(Transaction[] possibleTxs) {
        // set to store all the valid transactions
        Set<Transaction> txSortedByFee = new TreeSet<Transaction>((tx1, tx2) -> {
            double tx1Fee = TxFee(tx1);
            double tx2Fee = TxFee(tx2);
            return Double.valueOf(tx2Fee).compareTo(tx1Fee);
        });
        Collections.addAll(txSortedByFee, possibleTxs);
        // iterate over all possible transactions
        Set<Transaction> acceptedTx = new HashSet<Transaction>();
        for (Transaction tx : txSortedByFee) {
            // check if the transactions are valid
            if (isValidTx(tx)) {
            // add transaction to set of valid transaction
                acceptedTx.add(tx);
            // iterate over inputs of transactions to remove them from xurrent
               UTXOPool
                for (Transaction.Input in : tx.getInputs()) {
                    UTXO utxo = new UTXO(in.prevTxHash, in.outputIndex);
                    utxoPool.removeUTXO(utxo);
                }
                // iterate over outputs of transactions to add them to current
                   UTXOPool
                for (int i = 0; i < tx.numOutputs(); i++) {
                    Transaction.Output out = tx.getOutput(i);
                    UTXO utxo = new UTXO(tx.getHash(), i);
                    utxoPool.addUTXO(utxo, out);
                }
            }
        }
        // convert set of transactions to array of transactions
        Transaction[] validTxArray = new Transaction[acceptedTx.size()];
        return acceptedTx.toArray(validTxArray);
    }
```

2.4 Main

```
public class Main {
    public static void main (String[] args) {
        // TODO Auto-generated method stub
```

```
        Transaction tx = new Transaction();
        UTXOPool uxpool = new UTXOPool();
        TxHandler txhandler = new TxHandler(uxpool);
        System.out.println(txhandler.isValidTx(tx));
        Set<Transaction> validTxs = new HashSet<Transaction>();
        Transaction[] validTxArray = new Transaction[validTxs.size()];
        System.out.println(txhandler.handleTxs(validTxs.toArray(validTxArray))
            );
        // TODO Auto-generated method stub MaxFeeTxHandler
        Transaction tx2 = new Transaction();
        MaxFeeTxHandler maxFeeTxHandler = new MaxFeeTxHandler(uxpool);
        System.out.println(maxFeeTxHandler.isValidTx(tx));
        Set<Transaction> acceptedTx = new HashSet<Transaction>();
        Transaction[] maxValidTxArray = new Transaction[acceptedTx.size()];
        System.out.println(maxFeeTxHandler.handleTxs(acceptedTx.toArray(
            maxValidTxArray)));
    }
}
```

3 Results

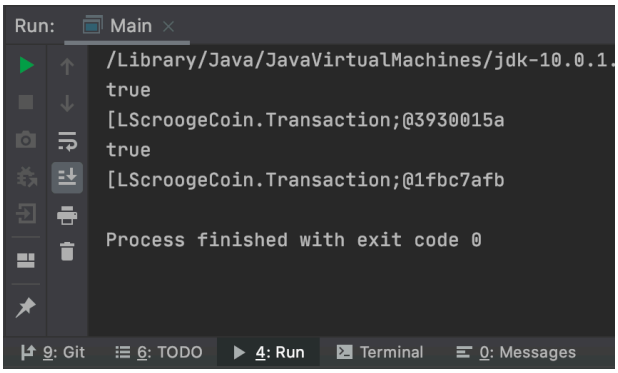

Figure 2: Results

4 Conclusion

In this practise, I tried to implement the logic used by Scrooge to process transactions and produce the ledger. I use Jetbrains IntelliJ IDEA as IDE for the first time.

According to the practice guide and the tutorial by TA, I understood the whole frame and tried my best on Java Programming as it's my first time to use Java. Since I'm not quite familiar with Java, I took such great amount of time to search for info and learnt it.

During the practice, I reviewed Scrooge Coin. And I found some review notes by other learners on the Internet, which gave me a lot of assist and motivation. By the way, I tried IntelliJ IDEA for the first time. I have heard about its strength on coding convenience and I finally tried it today.

In this practise, I have a deeper understanding about Scrooge Coin. Successfully complete the practical section this week.

5 References

PDF document: Practical 6 Guide File.

Website: https://zhaohuabing.com/2018/05/20/cryptocurrency_week1_scroogecoin/

PDF document: How to Write a Practical/Laboratory Report—Learning Guide.