Programming Assignment #4

CS–735/835

due 9 Apr 2017 11:59 PM

1 Presentation

1.1 A remote bank

This assignment uses RMI to implement a remote bank, somewhat similar in structure to the remote library presented in class. Banks are remote objects accessed through the remote interface `Bank` (Lis. 1). Via this interface, users can create and terminate accounts, deposit to and withdraw money from an account and consult an account's balance. All sums are expressed in *cents* using integers. Illegal operations (closing a closed account, depositing into a closed account, withdrawing more money than there is in the account, etc.) trigger an instance of `BankException`.

Accounts are identified by account numbers, which are positive long values. Method `OpenAccount` creates a new account and returns its number. The account's balance is zero. Conversely, `closeAccount` terminates an account and returns its current balance. Finally, `getAllAccounts` returns the numbers of all the (open) accounts in the bank, in increasing order.

1.2 Bank operations

Method `requestOperation` is used to request an operation on an account. There are three possible operations: deposit, withdrawal or consulting the balance. They are defined in a class `Operation` (Lis. 2). Each operation returns a record object, which is made of a message (string) and an amount in cents (integer) (Lis. 3). Operations and records are regular (non-remote) objects. They are immutable and serializable. Operations travel from clients to server; records from server to clients.

```
1  package cs735_835.remoteBank;
2
3  import java.rmi.RemoteException;
4
5  public interface Bank extends java.rmi.Remote {
6
7    long openAccount() throws RemoteException;
8    int closeAccount(long accountNumber) throws RemoteException;
9    long[] getAllAccounts() throws RemoteException;
10   Record requestOperation(long number, Operation operation) throws RemoteException;
11   RemoteAccount getRemoteAccount(long accountNumber) throws RemoteException;
12   BufferedAccount getBufferedAccount(long accountNumber) throws RemoteException;
13 }
```

Listing 1: Bank remote interface.

```
package cs735_835.remoteBank;

public abstract class Operation implements java.io.Serializable {

  public static Operation deposit(int cents) {...}
  public static Operation withdraw(int cents) {...}
  public static Operation getBalance() {...}
  ...
}
```

Listing 2: Bank operations.

```
package cs735_835.remoteBank;

public class Record implements java.io.Serializable {

  private static final long serialVersionUID = 64698026414748282 90L;

  public static final String DEPOSIT_RECORD_TYPE = "deposit to account #%s";
  public static final String WITHDRAWAL_RECORD_TYPE = "withdrawal from account #%s";
  public static final String BALANCE_RECORD_TYPE = "balance of account #%s";

  public final String type;
  public final int cents;

  @Override public String toString() {
    return String.format("%s: $%.2f", type, cents / 100.0);
  }

  Record(String type, int cents) {
    this.type = type;
    this.cents = cents;
  }
}
```

Listing 3: Transaction records.

1.3 Remote accounts

Instead of applying operations to the (remote) bank object, users can alternatively retrieve a (remote) account from the bank, then apply operations directly to this account. Method `getRemoteAccount` returns a stub on a remote object that implements the remote interface `RemoteAccount` (Lis. 4). This interface uses specific methods for deposit, withdrawals and balance consulting and does not rely on the `Operation` class. Records are produced by the server and returned as before.

1.4 Buffered accounts

When using `requestOperation` or `RemoteAccount`, clients need to connect to the bank server with every operation. For efficiency, it may be preferable, is some situations, to buffer operations on the client side in order to connect to the server less frequently. Class `BufferedAccount` (Lis. 5) implements such a strategy.

In this class, methods `accountNumber`, `deposit`, `withdraw` and `balance` *do not* connect to the server. Method `balance`, in particular, only reflects local deposits and withdrawals since the last connection to the server. Users need to request a server connection explicitly using method `sync`. If the aggregate of all operations since the last server connection is positive, a connection is made to the server to issue a deposit operation. If the aggregate is negative, a withdrawal operation is issued. Then, the server is consulted

```java
package cs735_835.remoteBank;

import java.rmi.RemoteException;

public interface RemoteAccount extends java.rmi.Remote {

  long accountNumber() throws RemoteException;
  Record deposit(int cents) throws RemoteException;
  Record withdraw(int cents) throws RemoteException;
  Record getBalance() throws RemoteException;
}
```

Listing 4: Remote accounts.

```java
package cs735_835.remoteBank;

import java.rmi.RemoteException;

public class BufferedAccount implements java.io.Serializable {

  public BufferedAccount(RemoteAccount account) throws RemoteException {...}
  public long accountNumber() {...}
  public void deposit(int cents) {...}
  public void withdraw(int cents) {...}
  public int balance() {...}
  public Record sync() throws RemoteException {...}
  ...
}
```

Listing 5: Buffered accounts.

again to obtain the current balance of the account (which reflects the aggregate operation, if any, as well as operations performed by other clients). The method returns a record of the deposit or withdrawal (or `null` is no deposit or withdrawal was issued).

Buffered accounts can be retrieved directly from the bank using method `getBufferedAccount` or clients can create them from a remote account using the public constructor of the class.

A buffered account needs to maintain a link to the server in order to implement the `sync` operation. One strategy would be to store a stub of the bank and to implement `sync` in terms of `requestOperation`. Another is to store a remote account and to implement `sync` in terms of the methods of this account. In this assignment, we use the second strategy because it makes it easier to build buffered accounts from existing remote accounts on the client side.

2 Implementation

An implementation is provided for the following classes:

- `BankException`: the class of exceptions thrown by illegal bank operations.
- `Record`: records of bank transactions.
- `BankServer`: a command-line application that starts a bank server. It can register the bank to an existing RMI registry or it can run its own.
- `TestClient`: a command-line application that creates accounts and applies operations to them using all three mechanisms (operations, remote accounts and buffered accounts).

These classes should not need any modification. Students must implement the following public classes:

- `Operation`: bank operations as serializable immutable instances.
- `LocalBank`: a `Bank` implementation that uses a `ConcurrentHashMap` to store the accounts. It relies on the properties of `ConcurrentHahsMap` for thread-safety (no additional locking).
- `BufferedAccount`: buffered accounts that internally use a `RemoteAccount` to connect to the bank.

Note that other classes are needed (in particular, an implementation of `RemoteAccount`), but they are not public.

3 Report

The report needs to discuss the following questions:

- How are atomicity and concurrency issues handled in the bank? What methods of `ConcurrentHashMap` are used for this purpose? How is correctness guaranteed if multiple clients act concurrently on the same account? Discuss specific scenarios, such as deposit vs withdrawal or deposit vs closing.
- How are atomicity and concurrency issues handled in remote accounts? How is correctness guaranteed if multiple clients act concurrently on the same account? Discuss specific scenarios, such as deposit vs withdrawal or deposit vs closing.
- Write a client that requests a large number of random operations on a bank account using all three mechanisms (operations, remote account and buffered account). Run this client and a bank server *on two different computers* connected by a network (wired or wireless) and compare the performance of the three mechanisms. Describe the experimental setup in the report (what machines/network were used).
- Consider the following scenario:

  ```
  RemoteAccount account = bank.getRemoteAccount(id);
  account.deposit(5);
  account.withdraw(10);
  ```

  Assume the initial balance of the account is zero (i.e., the withdrawal will fail). Describe *in detail* what objects are serialized and transfered between server and client (given the details of your implementation). Make sure you distinguish between remote objects and their stubs.
- Same question as above with the following scenario:

  ```
  BufferedAccount account = bank.getBufferedAccount(id);
  account.deposit(5);
  account.sync();
  account.withdraw(10);
  ```

- Same question as above with the following scenario:

  ```
  bank.requestOperation(id, Operation.deposit(5));
  bank.requestOperation(id, Operation.withdraw(10));
  ```

Notes:

- Public classes to be implemented are described at: `http://www.cs.unh.edu/~cs735/Programming/`. Public interfaces cannot be modified (i.e., you cannot add or remove a public or protected method, constructor, field or class, nor modify the checked exceptions thrown by a public or protected method or constructor). You can add private and package-private classes, methods, fields, etc.

- `LocalBank` relies on `ConcurrentHashMap` for its *own* thread-safety, but should delegate the thread-safety of individual accounts to the accounts themselves. The account implementation can rely on synchronized methods for its thread-safety. Alternatively, one can use atomic integers to implement accounts, but be aware that the atomicity requirements are non trivial and require using techniques discussed in chapter 15. Note also that buffered accounts are not thread-safe and can use non-synchronized methods and regular integers.

- Even though all three ways of applying a banking operation end up being applied to same bank account (stored in the bank map), they can have slightly different semantics. For instance, getting the balance of a closed account fails when using `Operation`, returns zero when using a `RemoteAccount` and produces a (possibly non-zero) local value when using a `BufferedAccount`.[1]

- Class `Record` defines three constant strings. The idea is to use them to generate `type` arguments for the constructor:

  ```
  new Record(String.format(DEPOSIT_RECORD_TYPE, number), cents)
  ```

  This helps ensure that strings are correctly formatted when testing/grading.

- Ideally, all testing should be done with clients and servers running on separate machines, but it is sometimes inconvenient to do so (e.g., when using TestNG). Alternatively, server, client and registry can run in the same JVM. An even simpler approach can be used for some tests that bypasses the registry entirely:

  ```
  Bank bank = (Bank) LocalBank.toStub(new LocalBank("TestBank"));
  ```

  This creates a stub as if this stub had been obtained via a registry.

- When accounts are closed, there might still exist stubs on them. For this reason, closed accounts should not be unexported. This allows stubs to produce meaningful results (zero balance, `BankException`) instead of failing with `RemoteException`. Closed accounts are simply removed from the map and become unreachable in the server JVM. Once all the stubs are gone, they will be garbage-collected.

  This garbage-collection can take a while. Consider the following program, started via `sbt`:

  ```
  public static void main(String[] args) throws Exception {
    Bank bank = (Bank) LocalBank.toStub(new LocalBank("TestBank"));
    long id = bank.openAccount();
    System.out.printf("account #%d created%n", id);
    RemoteAccount account = bank.getRemoteAccount(id);
    System.out.println(account);
    System.out.println(account.deposit(2016));
    System.out.println(account.withdraw(2000));
    System.out.println(account.getBalance());
  }
  ```

[1] Assuming, of course, that the `remoteAccount` and `BufferedAccount` were obtained before the account was closed.

It produces the following output:

```
> runMain cs735_835.TestTermination
account #1 created
Proxy[RemoteAccount,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:[132.177...
deposit to account #1: $20.17
withdrawal from account #1: $20.00
balance of account #1: $0.17
[success] Total time: 7182 s, completed Mar 22, 2017 12:10:05 PM
```

Note that `account` is not an instance of a user-defined class but a stub (proxy), with its own `toString` and `equals` methods. Note also the time it took for the program to terminate: 2 hours!

Termination issues with RMI can be tricky and we won't pay attention to them in this assignment. In the program above, the JVM does not terminate immediately after the main thread terminates because there are two exported remote objects listening for connections (the bank and the account). After one hour, RMI triggers a garbage collection. The stubs are discarded, but the remote objects are not because at the time of this garbage collection, the stubs still exist and use threads that have references to the remote objects. As they are discarded, the stubs notify their remote objects, terminate their threads and stop producing lease renewals messages. An hour later, the garbage collector runs again and this time the remote objects are available for garbage collection. Once they are discarded, there are no listening servers left and the JVM terminates.[2]

Adding these two lines in `build.sbt` changes the delay between garbage collections to 30 seconds:

```
fork in (Compile,run) := true,
javaOptions in (Compile,run) := Seq("-Dsun.rmi.dgc.server.gcInterval=30000"),
```

The system then terminates after one minute.

[2]This is an educated guess. Something along those lines is happening that requires two rounds of garbage collection.