COMPSCI 335 Distributed Objects, Services and Programming
Week 4 Lab Work

Ian Warren

August 1, 2014

The purpose of this week's lab work is for you to gain practical experience with developing and deploying distributed applications.

The bulk of this week's work will be taken up with the main task – developing a Java RMI application. The secondary task simply involves building and deploying the Contacts Web service, discussed in lectures. The second task won't take long, but is worth doing as it's useful preparation for the main assignment.

Main task

A partial implementation of a Java RMI banking application is available from the course web pages on Cecil. Similarly to the Whiteboard application, the source code is structured into 3 packages: `client`, `common` and `server`. Code in the `client` and `server` packages is used exclusively by the client and server sides of the application respectively. The `common` package contains code required by both the client and server.

The client is responsible for loading a set of banking transactions, stored in a data file, and making RMI calls on `BankAccount` objects hosted by the server. The client employs a `Producer` thread to read from the file and deposit transactions into a buffer (in this case a `java.util.concurrent.BlockingQueue`). A number of `Worker` threads consume transactions from the buffer and make RMI calls on remote `BankAccount` objects as necessary. When all transactions have been processed, the client application requests and displays the final balances for each of the remote `BankAccounts` and exits. The server is responsible for hosting the remotely accessible `BankAccount` objects. Clients acquire proxy objects for the remotely accessible `BankAccounts` from the RMI Registry.

You will need to study parts of the source code but you need not understand the implementation details of some classes. Notably `BankCommandParser`, `CommandParser`, and `IllegalSyntaxException` implement the parsing of the transactions file and knowledge of this is not required to complete the tasks. You also don't need to make any changes to the way multithreading is used in the client – simply add your code where indicated.

Objective

Complete the banking application. Completing the project involves fleshing out particular methods and making minor changes as appropriate.

When you have finished coding, run the server and then the client. If your implementation is correct, the server should process 100,000 banking transactions and leave the account balances as follows:

- Account 67832189 $22,477.26
- Account 69826344 $4,438,479.12

- Account 61198701 $4,320,654.23

If your program does not result in the accounts having exactly the above balances, think about how the server-side middleware has to dispatch incoming invocations in order to maintain a reasonable level of throughput processing. The remedy to the problem lies in making a change to code in the `server` package.

Package client

1. Complete class `Client` by fleshing out method `lookupRemoteAccounts`. This method should contain the code to lookup, via the RMI Registry, the proxy objects for the `BankAccount` objects hosted by the server. `lookupRemoteAccounts` should return a `Hashtable` of (`String`, `BankAccount`) pairs, where the `String` index is the account number and the `BankAccount` value is a reference to a `BankAccount_Stub` instance.
2. Complete class `Worker` by fleshing out method `processCommands`. This method should inspect its argument, an array of `String`s where each `String` is a token and the tokens collectively describe a banking transaction. To process a transaction a RMI call must be made on a particular `BankAccount` object, via its proxy held in the hashtable created by `lookupRemoteAccounts`. The `String` array is structured as follows:
   - Element 0 identifies the type of transaction, one of “balance”, “name”, “deposit” or “withdraw”.
   - Element 1 stores the account number of the bank account to which the transaction applies.
   - Elements 2 and 3 store the amount involved for withdraw and deposit requests. Element 2 stores the number of dollars, element 3 the number of cents.

Package server

1. Modify class `BankAccountServant` so that its instances can be exported and remotely accessed. Note that you don’t need to make changes to the data fields or to the logic encoded in `BankAccountServant`’s methods.
2. Complete class `Server` by adding code to method `main` to register the 3 `BankAccount` instances with the RMI Registry. Each remotely accessible `BankAccount` object should be registered under its account number with the registry.

Package common

Make any changes to the source code in this package as you see fit. Hint: *one minor* change is required to *one* of the members of package `common`.

Food for thought

- Run the client on one machine, and the server and RMI Registry on a different machine. Does everything work as expected?
- Try running the RMI Registry on a different machine to the server. What happens when the server first attempts to bind to the registry? Suggest why the RMI Registry behaves in this way.
- Why does the Server process not terminate once it has registered proxy objects for its remotely accessible objects? Ordinarily, a process terminates once the last instruction within its `main()` method has executed. Think about *why* the RMI server process behaves differently to a regular program and *what* the middleware might be doing to yield this behaviour.