Project 1: Remote Method Invocation
15-440 Spring 2012

Assigned: Tuesday, 31 January
Due: Tuesday, 14 February

Overview

In this project, you will implement a *remote method invocation* (RMI) library. RMI forwards method calls over a network connection, permitting objects located on one Java virtual machine to call methods on another. Users of the RMI library write code in the usual Java style: remote methods appear, syntactially, as regular Java methods. The only difference is in how the objects on which the methods are invoked are obtained. The usage is roughly as follows:

Regular methods

```
// Create a (contrived) object which can read
// files on the local filesystem.

Storage storage = new Storage();

// Invoke a method on the local object.

byte[]  data = storage.retrieve("/dir/file");
```

Remote method invocation

```
// Create a "stub" object which "knows" how to
// communicate with the server. This object is
// generated for you automatically by the RMI
// library ("Stub" is a class in the RMI
// library) - you only need to define the
// interface "Server".

InetSocketAddress   address =
    new InetSocketAddress("127.0.0.1", 80);
Server  server =
    Stub.create(Server.class, address);

// Invoke a method on the remote object. This
// method is executed remotely. The data and
// return value are automatically marshalled
// and unmarshalled for you by the RMI library.
// Return values and exceptions are passed
// back to the local VM and can be handled in
// the usual Java fashion.

byte[]  data = server.retrieve("/dir/file");
```

A similar simplification takes place on the server side — the details are given later in the handout. As you can probably see, RMI greatly reduces the difficulty of writing networked Java code. After you complete the RMI library, you will use it to write a distributed filesystem in the next project.

Java already has a standard RMI library, but we will not be using it in this course. You may be interested in browsing its documentation to get more ideas about RMI. A link is given in the references section.

Logistics

You should work alone on this project. The starter code can be downloaded from the lab 1 section of the course Web page. When you are finished, submit your code in a single zip archive to `/afs/andrew/course/15/440-s12/handin/lab1/yourandrewid/`. Be sure to include your modified `Skeleton.java` and `Stub.java`, and files containing any helper classes you have created. `Stub`, `Skeleton`, and all helper classes must be in the package `rmi`. We will extract these classes and use them to test your implementation. Your RMI library should work on Java 6 virtual machines.

Detailed Description

RMI works, firstly, by exposing an object on one Java virtual machine as remotely-accessible, and secondly, by providing other virtual machines with a way to access this object. The remotely-accessible object can be thought of as a *server* in the abstract sense, since it provides some services through its remotely-accessible methods. Each Java virtual machine that accesses this object is then a *client*. Therefore, the RMI library has two major components: one that simplifies the task of making servers remotely-accessible, and another that simplifies the writing of the corresponding clients.

Note that it is important not to think of the remotely-accessible object as a "server" in the low-level (socket programming-level) sense. As you will soon see, a low-level TCP server is implemented in, and hidden by, the RMI library.

On the Server: Skeleton

The server is a regular Java object of a regular Java class, with some public methods. Some of these public methods are meant to be accessible remotely. To permit remote access, a *skeleton object* is created for the server. The skeleton object is implemented in the RMI library. It is a multithreaded TCP server which handles all the low-level networking tasks: it listens for incoming connections, accepts them, parses method call requests, and calls the correct methods on the server object. When a method returns, the return value (or exception) is sent over the network to the client, and the skeleton closes the connection.

Note that the server object itself need not perform any network I/O — this is all done entirely by the skeleton, within the RMI library. The server object does not even have to be aware of the existence of any skeletons that might invoke methods on it.

What determines which public methods of the server object are accessible remotely? The server object implements a certain kind of interface called a *remote interface*, which will be detailed later. The remote interface lists the remotely-accessible methods of the server object. The skeleton object is created for this interface, and only forwards calls to the methods which are listed in it.

On the Client: Stub

Clients use *stub objects* to access the server. Stub objects are created by the RMI library. Each one appears to implement a given remote interface. However, instead of implementing the interface in a direct manner, each stub object forwards all method calls to a server by contacting a remote skeleton.

When the client invokes a method on a stub object, the stub opens a connection to the skeleton, and sends the method name and arguments. As described in the section on skeletons, this causes the remote skeleton to call the same method on the server object. When the method finishes, the skeleton transmits the result, and the stub returns this result to the caller in the client virtual machine.

As with the skeleton, the user of the stub need not explicitly perform any network I/O — again, this is done entirely by the stub object, and implemented within the RMI library.

Diagra