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.

Diagram

The goal of the RMI library is, simply, that when the client calls a method on the stub, the skeleton calls exactly the same method on the server. If no network error occurs, it should appear as if the client made the call directly on the server, as if the server is a local object.

Remote Interfaces

A *remote interface* is simply a regular interface with the additional requirement that every method be marked as throwing a special exception, `RMIException`. This is because, when using RMI, a method may fail due to a network error, a protocol incompatibility, or for other reasons that are related only to RMI and have nothing to do with the functionality of the server object. These failures are, of course, signalled by throwing `RMIException`.

Example: File Server Based on RMI

1. Defining a remote interface

```
public interface Server
{
    public long size(String path) throws FileNotFoundException, RMIException;
    public byte[] retrieve(String path) throws FileNotFoundException, RMIException;
}
```

2. Defining a server class

```
public class ServerImplementation implements Server
{
    // Fields and methods.
    ...
```

```
    public long size(String path) throws FileNotFoundException, RMIException
    {
        // Some implementation of the size method, which probably
        // accesses the server machine’s local storage and does no
        // network I/O.
        ...
    }

    public byte[] retrieve(String path) throws FileNotFoundException, RMIException
    {
        // Some implementation of the retrieve method.
        ...
    }

    ...
}
```

3. Creating the server object and making it remotely-accessible

```
// Create the server object.

ServerImplementation    server = new ServerImplementation(...);

// At this point, the server object is a regular local object, and is not accessible
// remotely.

// Create the skeleton object. Note that the type Skeleton is parametrized by the
// remote interface for which the skeleton is being created. Unfortunately, due to
// a limitation of the Java language, the class object for the same interface must
// also be passed as the first parameter to the constructor, as shown below.

Skeleton<Server>        skeleton = new Skeleton(Server.class, server);

// Start the TCP server in the skeleton, making the server object remotely-accessible.

skeleton.start();
```

4. Accessing a server object remotely

```
// Create an RMI stub which will forward method calls to the remote object.

InetSocketAddress       address = new InetSocketAddress(hostname, port);
Server                  server = Stub.create(Server.class, address);

// Perform some method calls using the stub. In practice, RMIException and any
// other exceptions thrown by the methods in the Server interface must either be
// handled or declared by the method containing this code.

long                    file_size = server.size("/file");

...

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

Specification

You are to create, within the package `rmi`, two classes: `Skeleton` and `Stub`. Detailed specifications can be found in the Javadoc distributed with the starter code, which can be generated by executing `make docs` in your project directory. Brief overviews are given here.

`Skeleton` implements a multithreaded TCP server. The class is parametrized by the remote interface for which it accepts calls. Each of the `Skeleton` constructors requires the caller to provide a reference to a server object which implements that interface. The skeleton object must forward all valid call requests it receives to the server object thus specified. The skeleton object must also stop gracefully in response to calls to `stop`, must be restartable, and must call the `stopped`, `listen_error`, and `service_error` methods in response to the appropriate events.

`Stub` is a class factory which generates stub objects for remote interfaces. The class `Stub` itself cannot be instantiated. To repeat, it is important to note that stub objects are not instances of the `Stub` class, for reasons that should become clear after reading the implementation section.

Stubs must necessarily implement all the methods in their remote interface, and they must do this by forwarding each method call to the remote skeleton. Stubs should open a single connection per method call. Arguments should be forwarded as given, and results should be returned to the caller as they were returned to the skeleton from the server. If the remote method raises an exception, the stub must raise the same exception, with the same fields and the same stack trace. The stub may additionally raise `RMIException` if an RMI error occurs while making the method call.

Stubs must also additionally implement the methods `equals`, `hashCode`, and `toString`. Two stubs are considered equal if they implement the same remote interface and connect to the same skeleton. The `equals` and `hashCode` methods must respect this requirement. The `toString` method should report the name of the remote interface implemented by the stub, and the remote address (including hostname and port) of the skeleton to which the stub connects. Stubs must also be serializable.

All the constructors of `Skeleton` and all versions of `Stub.create` must reject interfaces which are not remote interfaces.

Implementation

The RMI library is very generic. `Skeleton` must be able to call any method in any remote interface. `Stub` must solve an even greater challenge: it must be capable of generating stub objects that *implement* any remote interface *at run-time*. You will need to use Java Reflection in order to achieve this level of flexibility. To start, consult the documentation for the `Class` and `Method` classes. Also see `InvocationTargetException`.

The generation of stubs that implement arbitrary remote interfaces at run-time is done by creating *proxy objects* (class `Proxy`). Every proxy object has a reference to an *invocation handler* (class `InvocationHandler`), whose `invoke` method is called whenever a method is called on the proxy object. The RMI library's stub objects will therefore be proxy objects, and the marshalling of arguments will be done in their invocation handlers. Please refer to the documentation of these classes.

Perhaps the easiest way to transmit arguments and results is by using Java's native serialization facility. For this, please refer to the documentation for the classes `ObjectInputStream` and `ObjectOutputStream`.

You may create whatever helper classes you need to implement `Skeleton` and `Stub`, but please put them in the `rmi` package and make them package-private.

Testing

The starter code is distributed together with some test cases in the `conformance/` subdirectory. These test the public interface of your RMI library for conformance with its specification, and part of your grade will depend on passing these tests.

You may create additional test cases in the `conformance/` subdirectory to test the public interface of your library. These tests are in the package `conformance.rmi`, so they can *only* access the public interface.

You may also create test cases in the `unit/` subdirectory. The tests in the `unit/` subdirectory are meant to share the package `rmi` with your library code, so you can test package-private classes and methods.

Run `make test` to run all tests. For detailed documentation on how to use the test library and write your own tests, run `make docs-all` and read the documentation for the package `test` in the Javadoc generated in the `javadoc-all/` subdirectory. The `Test` and `Series` classes are of particular interest. You should also look at the some of the tests distributed with the starter code as examples.

Notes and Tips

If you are creating both an `ObjectInputStream` and an `ObjectOutputStream` on two sides of a connection, you must ensure that the output stream is created first, and you must flush it before creating the input stream. This is because the `ObjectInputStream` constructor might not return until it receives a header from the peer's `ObjectOutputStream` object, and that object tends not to send it until it is either used to transmit something, or is flushed. If both peers create their input streams first, or fail to flush their output streams, then they will deadlock waiting for stream headers that will never come.

Be sure to gracefully handle invalid arguments (for example, `null`), and eagerly release any system resources your code acquires.

Note that the skeleton is multithreaded, so server objects must be thread-safe. Overlapping requests may, in principle, arrive from clients at any time.

Note also that there are several interesting cases: a server object may implement multiple remote interfaces, and different skeletons may be forwarding requests on those interfaces. Multiple skeletons may be forwarding requests on the same interface. There may also be no skeletons forwarding requests to a particular server. On the client side, the type of a stub is simply a remote interface. Nothing prevents a regular, local object from implementing this interface and "masquerading" as a stub. In fact, this allows a degree of flexibility — a user can choose between a remote or a local object at run-time.

There are two `Stub.create` methods, corresponding to two ways to create stubs. Why is this? The usefulness of one of the ways is obvious — the caller specifies the remote interface and the address of the remote host, and obtains a stub. The other way allows a stub to be created by taking the address automatically from a skeleton, which must therefore be running in the same virtual machine on which the stub is being created. This second method is useful when this stub will then be transmitted to a client. This allows callbacks to be implemented easily.

RMI stubs can be passed as arguments and return values in RMI calls. In this way, servers and clients can make each other aware of other servers by simply passing stubs through which the other servers can be accessed. Syntactically, passing stubs in this way appears the same as passing regular local references. This is also the reason for defining equality of stubs as done in the specification section.

References

Java Remote Method Invocation API

`http://docs.oracle.com/javase/tutorial/rmi/index.html`

`http://docs.oracle.com/javase/6/docs/api/java/rmi/package-summary.html`

Java Reflection API

`http://docs.oracle.com/javase/tutorial/reflect/`

`http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/package-summary.html`

Proxy objects and invocation handlers

`http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/Proxy.html`

`http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/InvocationHandler.html`

Object input and output streams

`http://docs.oracle.com/javase/6/docs/api/java/io/ObjectInputStream.html`

`http://docs.oracle.com/javase/6/docs/api/java/io/ObjectOutputStream.html`