15-498: Distributed Systems
Project #1: Design and Implementation of a RMI Facility for Java

Dates of Interest

Assigned: During class, Friday, January 26, 2007
Due: 11:59PM, Friday, February 13, 2007

Credits

Although this handout was prepared locally, this project was designed by Kohei Honda and published among the support materials for the Coulouris, et al textbook. Prof. Honda is a member of the Department of Computer Science, Queen Mary and Westfield College, University of London. He authored the source code provided on the project page.

Overview

This project asks you to design and implement a Remote Method Invocation (RMI) facility for Java. In other words, you are asked to provide a mechanism by which objects within one Java Virtual Machine (JVM) can invoke methods on objects within another JVM, even if the target object resides within a JVM hosted by a different, but network accessible, machine.

Pedagogy

This project is designed to reinforce your understanding of the basic challenges facing the developers of middleware, and the techniques used to overcome them. Specifically, we hope that this project will reinforce your understanding of the following:

- Naming objects (or programs) that reside among many hosts
- Locating objects (or programs) within many hosts
- Marshalling methods (or procedures) and their parameters
- Constructing a natural and largely transparent abstraction for the application developer using lower-level network abstractions, e.g. sockets

Since this project is Java-specific, we do hope that you'll take some time to consider Java's native JVM facility and to consider the design decisions made by its implementers. Needless to say, in many ways Java's own RMI might be a good inspiration for the RMI that you design and implement. By doing so, we hope that you'll gain some valuable insights:

- A hands-on and in-depth understanding of RMI in Java
- A considered and critical understanding of both the design decisions made by the developers of Java's RMI and the trade-offs present in these decisions

The Requirements

The requirements are, if open-ended, very straight-forward. Without using Java's RMI facility, which includes everything in the java.rmi package, design and implement an RMI facility for Java.

Once this is complete, prepare a professional-looking report that describes your design, the major design decisions, including trade-offs, and anything that is broken or incomplete. The report does not have to be large or fancy. Rather than counting pages or investing time in intricate figures, concentrate on effectively communicating to us those things that we have asked, with the fewest possible words and simplest figures.

You Can't Do It All

We recognize that there will be components of the facility that you simply won't have time to complete. In general we expect your solution to implement the following elements:

- the ability to name remote objects, e.g., remote object references
- the ability to invoke methods on remote objects, including those methods that pass and/or return remote objects references and those methods that pass and/or return references to local objects.
- The ability to locate remote objects, e.g. a registry service

We don't expect you to create all of the tools that would be part of a commercial package. For example, the following would be nice, but aren't required:

- A stub compiler (this is a bit time consuming but mechanical)
- The automatic retrieval of .class files for stubs (this isn't bad, if you've got a little extra time)
- A distributed garbage collector (we wouldn't even have time to think about this in the time provided)

Furthermore, Java's native RMI facility is not perfect. For example, you might try playing with the .equals, clone(), or hashCode(). You'll soon discover that these don't work. We certainly don't expect you to fix these – but we'd like for you to understand the limitations and the reasons that they exist.

For things you would have liked to have done, but did not, we would like you to do three things:

1) Ensure that these things are possible given the rest of your solution. For example, even if you don't implement a stub compiler, it should be possible to implement one – without magic (or intuition).

2) Provide a work-around so that we can test your project. For example, if you don't provide a stub compiler, prepare some examples for us for which you have hand-written the stubs.

3) Document the not-yet-implemented components in your report, with as much of a description of their design as you have prepared.

Grading

In grading your project, we will consider the design and implementation of the RMI facility, as well as the report. The most important factor in your grade is the quality of what you have built from the perspective of the application programmer. The next most important factor is the quality of your analysis of your design and implementation, as expressed in your report. The third major grading consideration is the quality of your implementation and documentation from the perspective of a maintainer, peer, or review committee.

A Suggested Framework

Although we are leaving the design completely up to you, we do want to suggest an approach for tackling this assignment. In particular, we'd like to suggest that you take a very careful look at Java's native RMI mechanism and understand how it works and the trade-offs the designers made. We'd like to further suggest that you emulate what you like and rework what you don't.

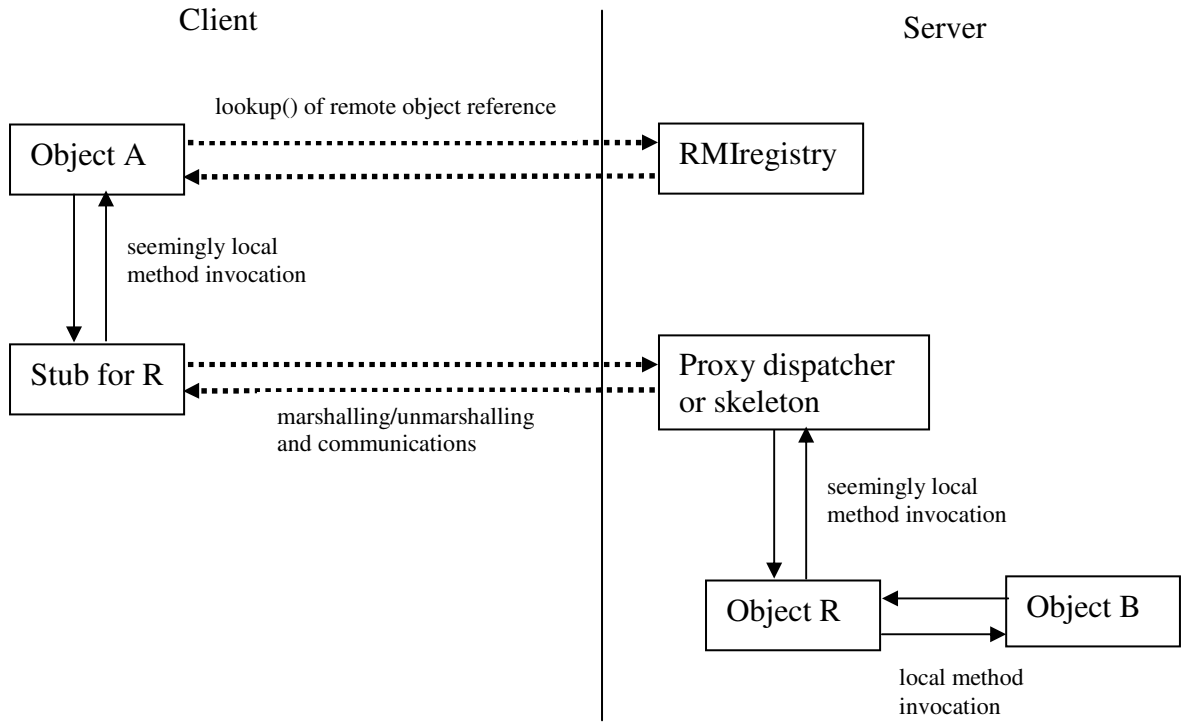
We suggest that you explore it not only through documentation, such as that provided by Sun, but also through experimentation, by writing RMI applications to illuminate the mechanisms and behaviors of Sun's RMI.

For those who take this approach, most of the rest of this document highlights several key aspects of Java's native RMI facility and offers some implementation ideas. It isn't an exacting definition – it does take some liberties. But, it is a good model for what Java's RMI actually does.

This document concludes with a plan of action that might be helpful for those following an implementation similar to this one.

The Big Picture

The figure below illustrates the model used by Java's native RMI facility. The components and interactions are described in the sections below.



Remote Object References

Java identifies objects using *references*. References are nothing more than names for objects. Typical references, such as those contained within Java's primitive *reference variables* are *local references*. That is to say that these references are capable of naming object only within a single JVM.

But, an RMI facility needs a way of naming objects that live within one JVM from another JVM. In otherwords, a *remote object reference* type is needed. Although this type is transparent from the application programmer's perspective, you probably want to create a class to represent it – it will be very useful internally.

The attributes of a remote object reference will vary with your design, but you might want to consider including the IP address and port number of the remote host, a local reference or other identifier, and the name of the interface implemented by the remote object.

Unless your remote object reference explicitly contains the local reference, you'll also need a mapping table on the server that maps between the remote object reference and the local object reference. And, actually manipulating Java references is harder than it seems.

Client-side Stubs

As discussed in class, Java represents objects to remote callers by placing a proxy object, known as a stub, locally within the caller's JVM. It is the job of this stub to handle the marshalling of the method invocation into a message, the delivery of the message to the communication module, and the reverse of this process, all the way to the client object, upon the methods return.

There is at least one instance of the stub class for each remote object in use within the JVM. If there are several remote objects, even if they are of the same type, there are several different instances of the stub class, one for each. Each instance of the stub class contains the remote object reference for the object that it represents.

In order to ensure that only one stub exist for each remote object, Java's RMI maintains a table that maps between the remote object reference and the local reference to the stub. If a stub has not already been created for a particular remote instance, it is created and registered in this table.

If the class for the stub is not already available on the client, it can be downloaded from the server via HTTP.

Server-side Skeletons

In the original version of RMI, there was a server-side compliment to the stub, known as the skeleton. The skeleton, like the stub, was responsible for marshalling. Java 2 eliminated the need for the server-side skeletons. It did this by factoring this functionality a common component, which I call the *proxy dispatcher*.

This was possible, because no part of the process is necessarily unique to the particular target object. The unmarshalling of the method call is a mechanical step which yields a local object reference, a method to invoke, and the parameters to this method. Once this is known, the process of invoking the method using the local reference is the same for all objects. And, the last step, the marshalling of the return value, is just as mechanical as the initial unmarshalling. The only trick is maintaining the opportunity for concurrency, without breaking anything, throughout the process.

We suggest eliminating the server-side skeletons – only if you are comfortable with doing so. It isn't a big deal if you do make use of skeletons in your design – and it might make the RMI mechanism a little less complicated.

Messages

We strongly suggest developing a general format for representing messages between classes. These messages can include method invocations, return values, and exceptions. In other words, develop a message class to represent the communication that will need to cross the network.

If you do this, you will be able to pass the necessary information into the constructor to create a new message, send this message object whole over the network to the other side, instantiate it, and ask it to unpack itself. If you don't do this, you'll have to worry about parsing data from the network in several different places – and this is no fun.

If you want to use three different message types, you might want to consider using inheritance to avoid block-copying the implementation of common behaviors.

Communications

We strongly suggest that you develop a class to handle communications. Depending on your design, the same one might work for either side of the pipe or you might develop two different classes, one for each side. If you do decide two develop two different classes, you might find inheritance useful.

If you've got time, you might want to consider developing a connection cache. It isn't uncommon to find that two different JVMs are chattering a good bit – and it can get expensive to build up and tear down the session with each network connection.

Pass by Value vs. Pass by Reference

In Java, parameters are passed into methods and returned from methods *by reference*. This is problematic for an RMI facility, because not all objects can be remote objects – not all JVMs are willing to expose any of their objects, never mind all of them. As a result, the RMI facility needs to determine which object can be passed by reference, and which can't. And, it needs to have some mechanism for handling those that can't be passed by reference.

To address the first concern, Java has a very simple rule. Any object that is to be remotely accessible must be an instance of a class that implements the Remote interface. Objects that implement the Remote interface are passed by reference into methods and when they are returned from methods. Other objects are passed *by value,* in other words by creating local copies.

Java passes object by value using a process known as *serialization*. Basically, this means that Java flattens out the object, copies it, and sends this copy to the other side. At the other side, the object is recreated from the serialized copy, and a reference to this recreated object is used. Java needs to have an object's .class file to reconstitute it from the serialized copy.

In order to recreate an object from a serialized copy, the object's .class file is needed. To facilitate this, Java sends the URL for the .class file along with the serialized copy. If the recipient doesn't already have the .class file, it can download it via HTTP using the provided URL.

The result of this process is that there are two copies – one on each side. The client's JVM has one copy and the server's JVM has another. Each acts on its own copy. The object has been passed by value.

When Java passes an object by reference, it does this by passing a remote reference to the object, along with the URL of the stub class. This enables the recipient to recreate the stub object just as it did objects passed by value. As before, if the recipient doesn't already have a copy of the defining .class file, it can download it using HTTP via the provided URL.

The process of recreating a remote object or stub on the local system is called *localization.* Pass by value localizes the remote object, pass by reference localizes a stub for the remote object.

You are welcome to use Java's serialization methods, writeObject() and ReadObject, which can be found in *ObjectInputStream* and *ObjectOutputStream*. But, these probably will not prove to be as effective as you would like, since they won't treat your remote object references specially.

Failure and Exceptions

Unlike local method calls, remote method calls can fail. As an example, the network could be down or partitioned. Java's native RMI handles this by requiring that all methods of remote objects throw RemoteException. This, in turn, requires that each use of a remote method catch the RemoteException.

Finding Remote Objects

Most remote objects are "found" when references to them are returned by methods invoked on other remote objects. But, for obvious reasons, this mechanism does not explain how all remote objects are found -- we need to find the first object somehow.

Java does this using a mechanism known as the RMIregistry. Servers that create remote objects designed to be the first point of contact by a client can register these remote objects using bind() or rebind(), which take a common, URL-style name and the local reference to the object.

Once that happens, a client can connect to the RMIregistry on that server and ask for an object by name. In return, the client will get a reference to the remote object. A client can also invoke the list() method on the RMIregistry, which will return an array containing the names of all of the registered objects. The RMIregistry isn't global, instead there is one per server. Clients need to connect to a particular server's RMIregistry, which can tell them only about the objects registered on the same server.

One Possible Plan of Action

1. Play with Java's RMI. Become comfortable with it. Test out example that include remote and local objects as parameters and return values.

2. Take a look at ObjectInputStream and ObjectOutputStream.

3. Write toy code capable of serializing an object, writing it to a file, recreating it, and using the new instance.

4. Write toy code capable of serializing an object, sending it over a socket, recreating it at the other side, recreating it, and using the recreated copy.

5. Write the RMIMessage class capable of encapsulating a method invocation. Test it out by marshalling a method invocation, unmarshalling it, and invoking it.

6. Enhance the RMIMessage class so it can handle return values, and then exceptions, if applicable to your design. Test this out after each step.

7. Write toy code which accepts a method invocation, marshals it using your RMIMessage, unmarshals it from your RMIMessage class, invokes the method on another class, marshals the return value, unmarshals the return value, and then returns the value

8. Develop your *RemoteObjectReference* class

9. Develop your *498Remote* interface, make sure you can use getInterfaces() to determine if an object implements this interface.

10. Write a sample class that implements the *498Remote* interface. Write a very simple sample client stub by hand. This stub should accept a local reference and marshal the method invocation to a local instance of the object.

11. If you are using server skeletons, write one for the sample remote class.

12. Write your communication modules. At this point, don't use the network. Use a file. This is easier to monitor for debugging purposes.

13. Get the whole process working without the network. Test, test, test, test – it doesn't get easier with the network – and there is no file to observe.

14. Eliminate the file interaction and add in full-blown network interaction. Once this is done, you've actually got a working RMI.

15. Implement a registry and add support for it.

16. If you are interested add support to download the .class files using HTTP.

17. If you are interested, add support for connection caching.

18. If you are interested, write the RMI compiler

19. Clean up everything

20. Organize some of your test code, or produce some examples for us that demonstrate your work.

21. Write your report.