Lecture 7: Java RMI

CS178: Programming Parallel and Distributed Systems

February 14, 2001
Steven P. Reiss

I. Overview

A. Last time we started looking at multiple process programming

1. **How to do interprocess communications**

B. We looked first at sockets as a basic mechanism

1. **Sockets provide the tools to communication**
2. **Two-way communications links**

C. But sockets aren't a logical approach

1. **Want to deal with messages rather than raw data**
2. **Message mechanisms can be built on sockets**

D. What type of messages to consider

1. **Informative messages -- process A informing process B that something happened**
2. **Data messages -- process A providing data to process B**
3. **Command messages -- process A asking process B to do something; process B sending the result of that request back to process A**

E. Central Message Server Architecture

1. **Tools describe the messages they want to receive**
    a) Either the commands they accept
    b) Or the information they want to know about
2. **Tools send messages to the server**
    a) Messages rebroadcast to interested tools
3. **Tools can reply to messages (especially commands)**
    a) Replies sent back to original sender

II. Remote Procedure Calls

A. If we are sending commands across sockets

1. Why not think of these as procedure calls

a) They are issued in one process

b) And executed in a second process

2. This is the notion of remote procedure calls

B. RPC availability

1. RPC has been around for quite a while

a) Execution model

b) Execution model with multiple threads

2. RPC tools

a) Automatic generation of STUBS and SKELETONS from C declarations

b) Libraries to do all the data encoding and decoding

c) Libraries to handle authentication, finding servers, ...

III.Remote objects

A. Motivation

1. As we move from procedural to OO programming

2. These should be method calls on remote objects

3. This is the basic idea in modern distributed message passing systems

B. What are the issues that arise here

1. Identifying the object to be called

2. Passing arguments to the routine/method

a) All the previous problems

b) But objects are generally pointers, how to deal

c) Also, how to deal with passing remote objects

3. Synchronous vs. asynchronous calls

4. Handling failure

C. Lets start with what remote objects would look like to a client and server program

1. Diagram of object in server

2. **What exists in the client**

   a) Client needs an object for the programming language to work

   b) If I call a method on this object what should happen

   c) This yields a STUB

3. **What exists in the server**

   a) First the object must exist (we have that already)

   b) Second, someone has to take the message describing the call and decode it, make the call, encode the result, and send a message back

   c) This is a all class dependent

   d) Managed by a SKELETON for that class

D. Representing remote objects in a language

1. **The object needs to be the same in the client and the server**

   a) But you have a stub in the client and a real object in the server

   b) But the two have different functionality

   c) What language construct provides for this

2. **Remote objects are typically represented as interfaces**

   a) This lets code ignore whether object is local/remote

   b) Allows for arbitrary implementation of the remote object

   c) This is done for CORBA, COM, and RMI in slightly different ways

      (1) COBRA, COM -- invent their own interfaces outside the language

      (2) Java RMI -- uses Java interfaces inside the language

IV. Java RMI

A. Representing Remote Objects

1. **A RMI remote object is described by an interface**

   a) That inherits from java.rmi.Remote

2. **Lets look at an example**

a) Suppose we want to implement a calculator that operates remotely

b) We define a calculator interface extending Remote

B. Handling Errors

1. If we deal with multiple processes connected by a network, we lose control of lots of things

a) Either process could be terminated or die

b) The network might go down

c) The processes might not agree on things (e.g. data formats)

d) The code being executed can throw an exception

2. We want to ensure that the program deals with these cases in a reasonable way

a) And make sure the programmer is aware that things can go wrong

b) Thus all remote methods must be declared to throw java.rmi.RemoteException

c) Or one of its supertypes (Exception, IOException)

3. Thus we have to change our example interface

C. Implementing the remote class

1. We next have to define the class that exists in the server

a) This will implement the remote interface

b) But it has to do more

(1) Slightly odd semantics for garbage collection, equals, hashCode

(2) Handle creating an object to export

(3) Server has to be registered so people can find the object

2. Thus the server has to inherit from RemoteServer

a) java.rmi.server.UnicastRemoteObject

b) Other choices

(1) Designed so you can implement replicated objects

(2) Designed so you can create objects that automatically start a server

3. **But method implementations are straightforward**

a) Throw remote exception, but otherwise as would be expected

b) Continue with our example

V. RMI Overall Architecture

A. In order to use RMI, the client must be able to find the objects in order to call them

1. **This is done by a separate process, the RMI Registry**
2. **Servers register remote objects with the registry**
3. **Clients lookup and get handles to these objects**

B. Starting the registry

1. **rmiregistry command starts a registry**
2. **This should be run as a background process**
3. **Started once on the machine is enough**
4. **Runs at port 1099 by default**

C. Then there are calls to register/lookup names

1. **Names are URL-like qualified items**

a) //host/dir/dir/item

b) Use your own scheme to ensure uniqueness

2. **Getting the registery**

a) You can access a registry explicitly

b) Or use the default registry (easier) via java.rmi.Naming

3. **Binding calls**

a) Naming.bind(String url,Remote obj) throws AlreadyBoundException, MalformedURLException, RemoteException

b) Naming.rebind(String url,Remote obj) throws MalformedURLException, RemoteException

4. **Lookup calls**

a) Remote Naming.lookup(String url) throws NotBoundException, MalformedURLException, RemoteException

b) String [] list(String url) throws ... -- provides a list of names bound at the given registry

D. Implementing the server itself

1. The server has to do a couple of things

a) Install a security manager if you are going to be working with applets, etc.

```
if (System.getSecurityManager() == null) {
   System.setSecurityManager(new RMISecurityManager())
 }
```

b) Create the server object

c) Register that object using Naming.(re)bind

2. Put this into our example as main() in server

a) Note that adding a remote object creates background threads that will keep running even after main exits

b) These threads wait for the objects

E. Implementing a client

1. The client needs to do a couple of things as well

a) Install a security manager for applets, etc.

b) Lookup the object to work with

c) Invoke it.

2. Find the remote object

a) Naming.lookup(url)

3. Invoke the remote object as you would normally

a) Handling RemoteException at all points

4. Add this to our example as a test class

VI. Next Time

A. What is happening behind the scenes

1. Skeletons and stubs

2. Generating skeletons and stubs

B. How arguments are passed

1. Object serialization and class loading

2. Passing remote objects

3. Receiving remote objects

C. **Server complexities**

   1. **Finding the proper server**
   2. **Handling multiple servers**
   3. **Creating a new server where appropriate**

D. **Other remote object models**

   1. **CORBA**
   2. **DCOM**