Introduction to RMI

Mark Allen Weiss
Copyright 1999, 2000

Outline of Topics

- Basics of RMI
  - RMI vs other solutions
  - the bootstrap registry server
  - stubs and skeletons
- Serving a remote object
- Using the remote object
- Preview of advanced issues

RMI Involves “Advanced” Topics

- Reflection
- Serialization
- Threading
- Security
- Networking
- Classloading
- Garbage Collection
RMI Basics

- Allows you to use objects that are on remote machines.
- Supported starting with Java 1.1.
- 100% pure Java solution: the remote objects must be written in Java.
- Other solutions:
  - CORBA: Language and platform neutral; communicate via Java IDL. However, Java IDL is not standard yet.
  - DCOM: Language neutral, but works only in an MS world.

RMI Overview

- Remoteable objects implement the Remote interface.
  - Actually, abstract class RemoteObject implements Remote
  - Abstract class RemoteServer extends RemoteObject
  - Concrete class UnicastRemoteObject extends RemoteObject
- Typically, extend UnicastRemoteObject.
- All methods in a Remoteable object must declare that they throw RemoteException.

Remote Interfaces

- Remote objects are accessed through their interfaces only.
- Typically, need to define a remote interface and a remote object that implements the interface
- Remote interfaces
  - Must be public
  - Must extend the Remote interface
  - All methods must declare RemoteException in throws list
  - Any remote parameters/return values must be declared using their interface type
A Hello World Remote Object

```java
import java.rmi.*;
import java.rmi.server.*;

public interface Hello extends Remote {
    void print() throws RemoteException;
}

// In a separate file:
public class HelloImpl extends UnicastRemoteObject
    implements Hello {
    public HelloImpl() throws RemoteException {
    }
    public void print() throws RemoteException {
        System.out.println("Hello world!");
    }
}
```

Serving a Remote Object

- Need to generate a stub and skeleton for the remote class. Same idea as in native calls, and in CORBA. Use `rmic` utility on server side.
  - In Java 1.2, use `-v1.2` option to generate stub only
  - skeleton is implicit from stub
- Need to have a main method (or other function) that creates instances of the remote object.

What Stub and Skeleton Do

- Stub provides implementation of remote interface in the client VM.
- Stub serializes interface method calls and arguments to remote skeleton. (Marshalling)
- Skeleton deserializes method calls (unmarshalling), calls the desired method, and returns a value or a RemoteException, either of which is serialized back to the stub on the remote client.
- The client stub deserializes the return value or exception and rethrows any exception.
Class Loading

- The class representing the remote object must be loaded on the client machine.
- Client has the interface, but not implementation classes.
- Dynamic class loading can be used (like applets) to get any new remote class (implementations and stubs) that are needed; want the security manager to check that these dynamically loaded classes are ok.

Accessing Remote Objects

- If method returns remote object, can be accessed through interface to it.
- First server object is special: need to use the bootstrap registry service.

Registering Objects

- Use Naming.bind (or Naming.rebind).
- Client calls Naming.lookup (returns an Object, that client can cast to the interface type).
- Need to start the bootstrap registry. By default it runs on port 1099. From an MS-DOS window:
  ```
  start rmiregistry portnum
  ```
- If your application is the only program using the registry, can place, in main:
  ```
  LocateRegistry.createRegistry( portnum );
  ```
Server Main

```java
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class HelloServer {
    public static void main(String[] args) {
        try {
            HelloImpl h = new HelloImpl();
            Naming.bind("hello", h);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Summary of Server-Side

- Write an interface and implementation for the remote object.
- Run rmic to get the stub and skeleton.
- Write a main that creates the remote object, and (if needed) installs it in the bootstrap registry.
- Start the bootstrap registry.
- Start main.

The Client Side

- Must get the first remote object by using Naming.lookup. Subsequent object interfaces (that is, those returned as parameters) are obtained automatically by the dynamic class loader.
- Naming.lookup needs URL of server (possibly including the port number) and name stored in registry.
- Client accesses via the interface type only!
- Should install a security manager if really doing remote stuff.
Using the Hello Remote Object

```java
import java.rmi.*;
import java.rmi.registry.*;

public class HelloClient {
    public static void main(String[] args) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            Object obj = Naming.lookup(args[0] + "hello");
            (Hello) obj).print();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Method Parameters

- RMI uses Serialization to send remote references, objects, and primitives.
- Thus remote method arguments that are Objects must be either
  - Remote objects
  - Serializable
- This can be a pain:
  - Somethings cannot be passed (e.g. Graphics).
- Notice: Remote method semantics are different than local semantics when the parameter is a non-remote Object.

Larger Example With DCL

- Will have server have object that can be queried to give the host name and date.
- Client will be able to access this object.
- Client will have only
  - Client class file
  - Interface class file to remote object
- Client will not have
  - Stub or implementation of remote object; these will reside on the server and stub will be downloaded when needed
The Hello Interface

```java
import java.rmi.*;
import java.rmi.server.*;

public interface Hello extends Remote
{
    String getMessage() throws RemoteException;
}
```

Security

- Client will be downloading stub class of the remote object.
  - should be safe, since generate by rmic
  - but could be tampered with
  - need a security manager and policy file

```java
grant {
    // Allow everything for now;
    permission java.security.AllPermission;
}
```

The Implementation

```java
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
import java.util.Date;

public class HelloImpl extends UnicastRemoteObject implements Hello
{
    private static InetAddress me;
    static {
        try { me = InetAddress.getLocalHost(); } catch( java.net.UnknownHostException e) { }
    }

    public HelloImpl() throws RemoteException {
    }

    public String getMessage() throws RemoteException {
        Date now = new Date();
        System.out.println("Processing a request!");
        return "Hello from: " + me + " " + now;
    }
}
```
import java.rmi.*;
import java.rmi.registry.*;

public class HelloClient {
    public static void main(String[] args) {
        System.setProperty("java.security.policy", "all.policy");
        System.setSecurityManager(new RMISecurityManager());
        try {
            System.out.println("I AM THE CLIENT");
            String mach = args.length == 0 ? "" : args[0];
            Object obj = Naming.lookup("/" + mach + ":6000/hello");
            Hello hobj = (Hello) obj;
            System.out.println("Server returns: " + hobj.getMessage());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class HelloServer {
    public static void main(String[] args) {
        try {
            HelloImpl h = new HelloImpl();
            String mach = args.length == 0 ? "localhost" : args[0];
            Naming.rebind("/" + mach + ":6000/hello", h);
            System.out.println("I AM THE SERVER");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

Home Setup

- Compile HelloImpl.java
- Run rmic -v1.2 HelloImpl
- Compile the client and server
- Copy to a server (can use servletrunner to set up a webserver)
  - interface, stub, implementation, server class file
- Remove from client
  - stub, implementation, server class file (leave interface and client)
Running Everything

- On server:
  - unset CLASSPATH
  - run rmiregistry with port 6000 in parent directory
  - cd...
  - set CLASSPATH=
  - start rmiregistry 6000
  - go to server directory
  - start server, with CLASSPATH that includes . and define java.rmi.server.codebase=YourWebDir/
    java.io.class.path=. -Djava.rmi.server.codebase=http://localhost:8080/HelloServer

- On client
  - start the client

Garbage Collection

- Distributed objects are garbage collected
- VM uses idea of a lease that is renewed periodically by clients

Threading

- Server handles each request in separate thread
- Client thread is blocked until remote method returns
- Synchronizing on reference on client does not synchronize remote object; only synchronizes stub
Summary

- RMI basics not too complicated.
- Write an interface that can be used by clients.
- Server generates stub and skeletons via rmic.
- On server side, need to start a bootstrap registry, and bind at least one remote object.
- On client side, need to locate one remote object; dynamic class loading gets other remote classes.
- Client needs to run with at least an RMI security manager. Server should do the same.