Programming with Java RMI …an introduction...

CS 417

Distributed Systems

Paul Krzyzanowski

Java RMI

- Allows a method to be invoked that resides on a different JVM:
  - remote machine
  - same machine, different JVM (different address space)
- Object-oriented RPC

RMI: Remote Method Invocation

Java RMI is a mechanism to allow the invocation of methods that reside on different Java Virtual Machines (JVMs). The JVMs may be on different machines or they could be on the same machine. In either case, the method runs in a different address space than the calling process.

Java RMI is an object-oriented remote procedure call mechanism.

It differs from CORBA:

+ CORBA is language/machine independent

  - RMI was designed for Java running on a JVM

+ CORBA includes more mechanisms:

  server application starting

  managing persistent state

  support for transactions

+ Java RMI does not have an Object Broker

It differs from Sun RPC and DCE RPC:

+ RMI is not language/machineindependent
+ RMI supports the concept of classes with methods within a class
+ RMI supports polymorphism

  (different methods that share the same name but accept different parameters)

Participating processes

- Client
  - process that is invoking a method on a remote object
- Server
  - process that owns the remote object (local object to the server)
- Object Registry
  - name server that associates objects with names
  - objects are registered
  - URL namespace
    `rmi://hostname:port/pathname`
    e.g.: **`rmi://crapper.pk.org/MyServer`**

There are three entities involved in running a program that uses RMI:

client: this is the program that you write to access remote methods

server: this is the program that you write to implement the remote methods - clients connect to the server and request that a method be executed. The remote methods to the client are local methods to the server.

Object registry: this is a program that you use.

The object registry runs on a known port (1099 by default)

A server, upon starting, registers its objects with a textual name with the object registry.

A client, before performing invoking a remote method, must first contact the object registry to obtain access to the remote object.

Two kinds of classes

- Remote class (remote object)
  - instances can be used remotely
  - works like any other object locally
  - in other address spaces, object is referenced with an *object handle*
  - if a remote object is passed as a parameter, its handle is passed.

Two types of objects are useful when dealing with RMI.

A **remote object** is one whose instances can be used remotely. A handle to a remote object identifies where that object is located and how to contact it remotely (via rmi).

When used locally, it works like any other object.

When it is passed as a parameter, its handle is passed (as with ordinary java objects).

Two kinds of classes

- Serializable class (serializable object)
  - instances can be copied between address spaces
  - object that can be marshaled
  - can be passed as a parameter or be a return value to a remote object
    - value of object is copied
  - implements **java.io.Serializable** interface

A **serializable** object is one whose value can be marshaled. This means that the contents of the object can be represented in a pointerless form as a bunch of bytes and then reconstructed into their respective values as needed. This is useful, for example, in saving the contents of an object into a file and then reading back that object.

In the RMI case, it is useful if we want to pass an object as a parameter to a remote method or receive a result from a remote method. In this case, we *don't* want to pass object handles, because the pointers will make no sense on a different JVM. If an object is defined to implement java.io.Serializable, however, passing it as a parameter will allow us to pass the *value* of the object (marshaled) rather than the handle.

remote classes

- Remote class has two parts:
  - interface definition
  - class definition
- Remote interface
  - must be public
  - must extend **`java.rmi.Remote`**
  - every method must declare that it throws **`java.rmi.RemoteException`**
- Remote class
  - implements Remote interface
  - extends **`java.rmi.server.UnicastRemoteObject`**

There are two parts to defining a remote class: its interface and the actual class itself. The Remote Interface represents the type of an object handle and tells clients how to invoke remote methods. The remote class defines those objects.

The interface definition:

- must be public
- must extend the interface java.rmi.Remote
- every method in the interface must declare that it thows

java.rmi.RemoteException

(but other exceptions may be thrown as well)

The Remote class:

- must implement a Remote interface
- should extend java.rmi.server.UnicastRemoteObject class

Objects of this class exist in the address space of the server and can be invoked remotely. (there are other ways of defining a remote class - this is the easiest)

Objects in the remote class may have methods that are *not* in its remote interface. However, these can only be invoked locally.

Sample program

- Client invokes a remote method with strings as parameter
- Server returns a string containing the reversed input string and a message

Define the remote interface (`HelloInterface.java`)

```
import java.rmi.*;

public interface HelloInterface extends Remote {
    public String say(String msg) throws RemoteException;
}
```


We define an interface for a remote class that includes one remote method: *say*

say accepts a String as an input parameter and returns a String as a result

Note that the interface extends Remote and that each method (there's only one here) throws RemoteException.

These classes are defined in the java.rmi package, so we import it.

Define the remote class (`Hello.java`)

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

public class Hello
 extends UnicastRemoteObject
 implements HelloInterface {
    private String message;

    public Hello(String msg) throws RemoteException {
        message = msg;
    }

    public String say(String m) throws RemoteException {
        // return the input message - reversed input plus our standard message
        return new StringBuffer(m).reverse().toString() + "\n" + message;
    }
}
```


The remote class defines two methods:

Hello is our constructor and will be used by the server to set define the standard message that will always be sent back to the client.

say is a method that takes a string as the input. It returns a string containing the input with the characters reversed followed by a newline followed by our standard message (set in the constructor).

Note that:

- the class extends UnicastRemoteObject

  this will allow the methods to be invoked remotely

- the class implements the interface we defined

- each method must throw RemoteException

  because remote procedure calls might fail

- the say method accepts a String and returns a String. String is defined to be Serializable. If you were to accept or return an object you defined, it would have to be defined to implement Serializable.

The server (`HelloServer.java`)

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

public class HelloServer {
    public static void main(String argv[]) {
        if (System.getSecurityManager() == null)
            System.setSecurityManager(new RMISecurityManager());
        try {
            // instantiate the service with the standard message & tell registry
            Naming.rebind("Hello", new Hello("Hello, world!"));
            System.out.println("Server is running...");
        } catch(Exception e) {
            System.out.println("Hello Server failed: " + e);
        }
    }
}
```


The server is a plain program that is responsible for:

1. Creating and installing a security manager. We do this with:

   System.setSecurityManager(new RMISecurityManager());

The security manager makes sure that classes that get loaded do not perform operations that they are not allowed to perform.

2. Registering at least one remote object with the object registry.

   We do this with:

   Naming.rebind(object_name, object);

   where object_name is a String that names the remote object

   and object is the remote object that we are registering.

All that our server does is:

set the security maanger

create the remote object

register it

print a message saying “Server is running…”

The client (`HelloClient.java`)

```
import java.rmi.*;

public class HelloClient {
    public static void main(String argv[]) {
        try {
            if (args.length < 0) {
                System.err.println("usage: java HelloClient string …\n");
                System.exit(1);
            }
            HelloInterface hello =  (HelloInterface)Naming.lookup("//localhost/Hello");
            for (int i=0; i < args.length; ++I)
                System.out.println(hello.say(args[i]));
        } catch (Exception e) {
            System.out.println("Helloclient exception: " + e);
        }
    }
}
```

Paul Krzyzanowski • Distributed Systems

The client invokes the remote method.

To do this, the client must have a handle (reference) to the remote object.

The object registry allows the client to get this reference to a remote object.

To get the reference to the remote object, we need to know:

1. Internet name (or address) of machine that is running the object registry for which the remote object is registered (can omit if localhost)

2. Port on which the object registry is running (can omit if it is 1099 - default)

3. Name of the object within the object registry.

The Naming.lookup method obtains an object handle from the object registry running on localhost on the default port (we could have just looked up "Hello" in this case).

The result of **Naming.lookup** must be cast to the type of the Remote Interface

---

The remote invocation of the object is the call to **hello.say("abc")**. It returns a String which we print. Remember again, that we can get a String *only* because String is a Serializable class.

The policy (`policy`)

```
grant {
   permission java.security.AllPermission;
}
```


The policy file controls which clients have which permissions. The one in this example grants everyone global permission (*not a good one to use in a production environment!*).

The policy file is not needed when programming with JDK < 1.2

The Java security manager contains a number of *check* methods (e.g ***checkConnect***). In the default manager, these are implemented by calling a ***checkPermission*** method. The type of permission is specified by a parameter of type **Permission** that is passed to **checkPermission**. The method checks whether the permission is implied by a list of granted permissions.

To specify or change the security policy, there is a class named Policy. A program can get its policy by calling **Policy.getPolicy()** or set it (if it has permissions) with Policy.setPolicy(). The security policy is typically specified by a policy configuration file which is reaad when the program starts.

Compile

- Compile the interface and classes:

```
javac HelloInterface.java Hello.java
javac HelloServer.java HelloClient.java
```

- Create the class files for stubs:

```
rmic Hello
```

- rmic creates:

```
Hello_Skel.class   skeleton - server-side stub
Hello_Stub.class   client-site stub
```


We first compile the remote interface and classes using javac. Then we generate the class files for the client and server stubs with the rmi compiler:

rmic Hello

This produces two class files:

Hello_Skel.class is the skeleton (a name for a server-stub)

Hello_Stub.class is the client stub function

Run

- Start the object registry (in the background):

```
rmiregistry &
```

- Start the server (in the background):

```
java -Djava.security.policy=policy
HelloServer &
```

- Run the client:

```
java HelloClient testing abcdefgh
```

- See the output:

```
gnitset
Hello, world!
hgfedcba
Hello, world!
```


Now we're ready to run the program.

If the registry is not running, it must be started. Otherwise the server will not be able to register the object and the client will not be able to look it up.

We start it in the background with:

rmiregistry &

If we wanted it to listen on a different port than 1099, we would specify the port number on the command line. For example, to listen on port 5511:

rmiregistry 5511 &

Next we start the server (also in the background) by running:

java -Djava.security.policy=policy HelloServer &

where the -D flag sets the name of the file containing our security policy (our file is named *policy* and is in the current directory).

Finally, we can run the client with:

java HelloClient testing abcdefg

NOTE: be sure to kill the registry and server processes when you're done.

That's it!

RMI
A bit of the internals

Interfaces

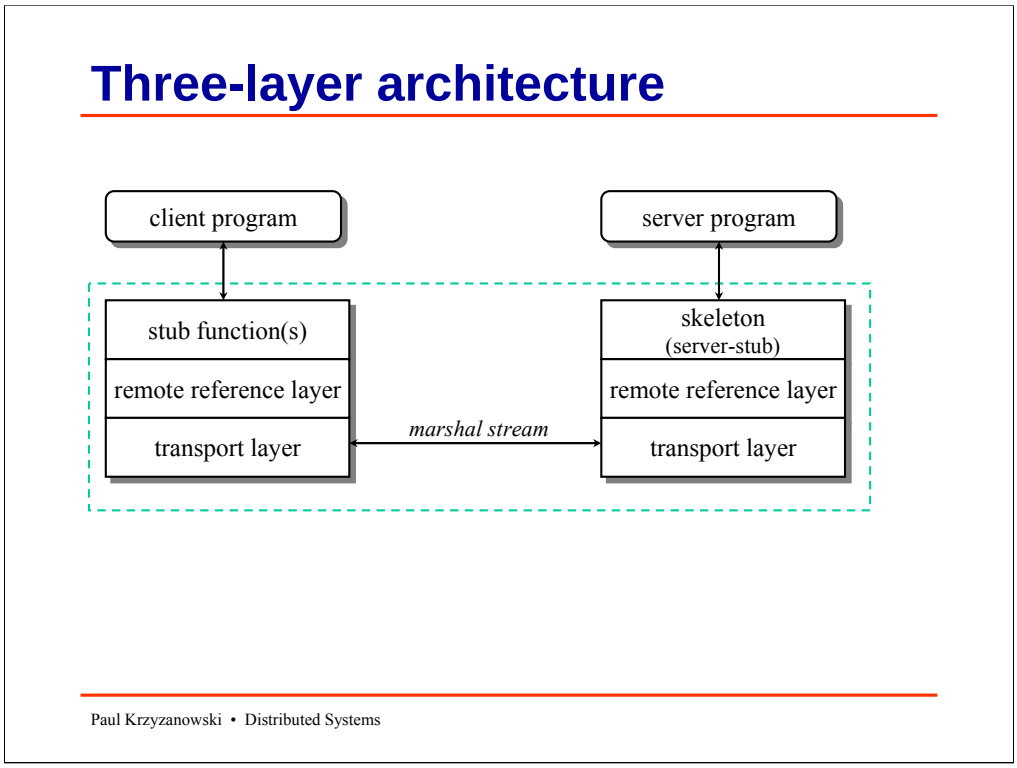
- Interfaces define behavior
- Classes define implementation
- RMI: two classes support the same interface
  - client stub
  - server implementation

When programming for RMI, we define an Interface: interfaces define the behavior - the API - of an underlying implementation. The classes themselves define the implementation.

In the case of RMI, two classes will end up supporting the same interface:

1. The client stub, generated via *rmi*c, provides a "proxy" to the server

2.. The server contains the actual code for the remote method.



RMI consists of three layers:

stubs & skeletons: this layer intercepts method calls made by the client to the interface reference and redirects them to a remote RMI service

remote reference layer: this layer understands how to interpret and manage references made from clients to the remote service objects.

In JDK 1.1, this layer connects clients to remote service objects that are running and exported on a server. The connection is a one-to-one (unicast) link.

In the Java 2 SDK, this layer was enhanced to support the activation of dormant remote service objects via *Remote Object Activation*. When a method call is made to the proxy for an activatable object, RMI determines if the remote service implementation object is dormant. If it is dormant, RMI will instantiate the object and restore its state from a disk file. Once an activatable object is in memory, it behaves just like JDK 1.1 remote service implementation objects. `

transport layer: based on TCP/IP. Provides connectivity via sockets. An internal protocol, the *Java Remote Method Protocol (JRMP)* is used for communication.

Server - 1

- **Server creates an instance of the server object**
  - extends UnicastRemoteObject
  - TCP socket is bound to an arbitrary port number
  - thread is created which listens for connections on that socket
- **Server registers object**
  - RMI registry is an RMI server (accepts RMI calls)
  - hands the registry the client stub for that server object
    - contains information needed to call back to the server (hostname, port)

The server has to create an instance of the server object

This object is defined to extend UnicastRemoteObject

The UnicastRemoteObject constructor:

- creates a TCP socket bound to an arbitrary port and set for listening

- creates a new thread that will listen for connections on that socket

The server then registers the object with the registry

- the registration process involves the server sending the registry the client stub for that server object. The stub will contain information needed to access the server (such as hostname, port) Note: the stub data is sent - not the code for the stub.

Java RMI does support the loading of remote stubs via the java.rmi.server.codebase property

- the RMI registry itself is an RMI server.

Client - 1

- **Client obtains stub from registry**
- **Client issues a remote method invocation**
  - **stub class creates a RemoteCall**
    - opens socket to the server on port specified in the stub
    - sends RMI header information
  - **stub marshals arguments over the network connection**
    - uses methods on RemoteCall to obtain a subclass of ObjectOutputStream
    - knows how to deal with objects that extend java.rmi.Remote
      - serializes Java objects over socket
  - **stub calls RemoteCall.executeCall()**
    - causes the remote method invocation to take place

The purpose of the RMI registry is to hold remote object stubs and send them to clients upon request.

The client connects to the registry and requests an RMI stub by name. The registry returns the appropriate stub.

The client then issues a remote method invocation on that object. The first step is for the stub to create a RemoteCall. This will cause a TCP socket connection to be created to the port specified in the stub and RMI header information to be sent to the server.

The client stub then marshals the arguments over the RMI connection using methods in the RemoteCall to obtain an output stream which is a subclass of ObjectOutputStream that knows how to deal with serializing Java objects over the socket.

The stub function then calls RemoteCall.executeCall() to actually have the remote method invocation take place

Server - 2

- **Server accepts connection from client**
- **Creates a new thread to deal with the incoming request**
- **Reads header information**
  - creates RemoteCall to deal with unmarshaling RMI arguments
- **Calls *dispatch* method of the server-side stub (skeleton)**
  - calls appropriate method on the object
  - sends result to network connection via RemoteCall interface
  - if server threw exception, that is marshaled instead of a return value

Back at the server…

the server accepted the connection from the client

It forked off a new thread to process the request. The original thread is still listening on the original socket so that other rmi calls can be made.

It reads the header information sent when the client established a *RemoteCall* and creates its own *RemoteCall* object to deal with unmarshaling the data it will receive from the client.

It then calls a *dispatch* method found in the server-side stub (the skeleton generated by *rmic*). This method causes a call to the appropriate method on the obect. That method finishes and the return result is send over the network connection via the *RemoteCall* interface (just like the client sent the parameters).

If an exception was thrown by the server code, instead of a return value, the exception is marshaled and sent back to the client.

Client - 2

- The client unmarshals the return value of the RMI
  - using RemoteCall
- value is returned from the stub back to the client code
  - or an exception is thrown to the client if the return was an exception

The client unmarshals the return value of the remote method invocation.

It uses the same RemoteCall interface that it used to marshal the incoming parameters.

If the return is an exception, then the stub causes an exception to be thrown back to the client. If the return is a value, then the stub returns the value to the invoking client code.