Java RMI: Remote Method Invocation

January 1998

Nancy McCracken

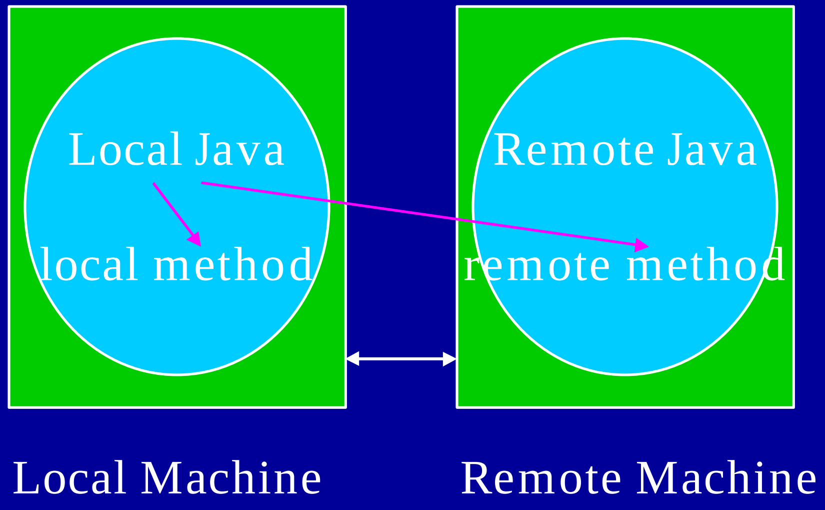
NPAC

RMI

- Java RMI allows the programming of distributed applications across the Internet. One Java application or applet (the client in this context) can call the methods of an instance, or object, of a class of a Java application (the server in this context) running on another host machine.
- An example of Distributed Object Programming - similar to CORBA, except that CORBA allows the remote objects to be programmed in other languages.
  - CORBA is a more general solution, but is not fully in place and has more overhead.
- References:
  - core Java 1.1, Volume II - Advanced Features, Cay Horstmann and Gary Cornell, Sunsoft Press, 1998.
  - advanced Java networking, Prashant Sridharan, Sunsoft Press, 1997.
  - http://www.javasoft.com/

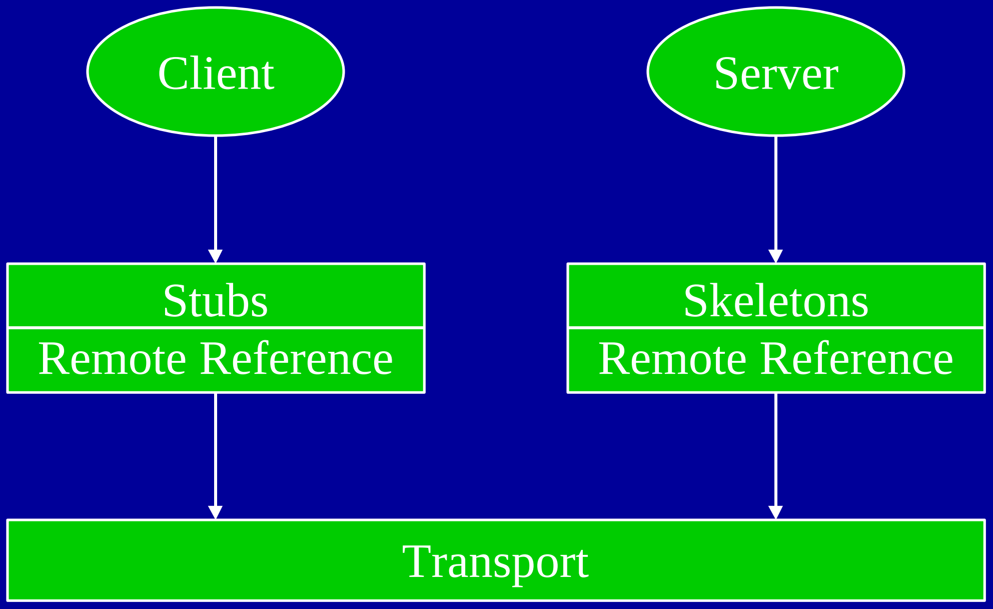
The Java RMI package

- Java RMI adds a number of classes to the Java language. The basic intent is to make a call to a remote method look and behave the same as local ones.
- Classes are added for
  - naming Registry - associates a name with a remote Java object
  - Remote interface - specification of remote methods
  - RemoteObjects
  - RMISecurityManager
  - RemoteExceptions



A Remote Method Call

- The architecture of a method call from the client to a method on the server.



Stubs

- To call a method on a remote machine, a surrogate method is set up for you on the local machine, called the stub.
- It packages the parameters, resolving local references. This is called marshalling the parameters:
  - device-independent encoding of numbers
  - strings and objects may have local memory references and so are passed by object serialization
- The stub builds an information block with
  - An identifier of the remote object to be used
  - An operation number, describing the method to be called
  - The marshalled parameters
- Stubs will also “unmarshall” return values after the call and receive RemoteExceptions. It will throw the exceptions in the local space.

Skeletons

- On the server side, a skeleton object receives the packet of information from the client stub and manages the call to the actual method:
  - It unmarshals the parameters.
  - It calls the desired method on the real remote object that lies on the server.
  - It captures the return value or exception of the call on the server.
  - It marshals that value.
  - It sends a package consisting of the return values and any exceptions.

Transport Layer

- The transport layer handles all the network issues.
  - It sets up a connection over a physical socket.
  - It knows the local and remote objects and how to translate to the local and remote name space.
  - It serializes objects as required.
  - It monitors the connection for signs of trouble, such as the remote server doesn¹t respond, and may throw RemoteExceptions.

Local vs. Remote Objects

- The goal is for local and remote objects to be semantically the same.
- For Java, an important issue is garbage collection, which automatically deallocates memory for local objects. Remote objects are also garbage collected as follows:
  - Remote reference layer on the server keeps reference counts for each object in Remote interface.
  - Remote reference layer on the client notifies the server when all references are removed for the object
  - When all references from all clients are removed, the server object is marked for garbage collection.
- One difference between local and remote method calls is that objects are passed to local method calls effectively by reference, whereas they are copied via the serialization technique to pass to remote method calls.

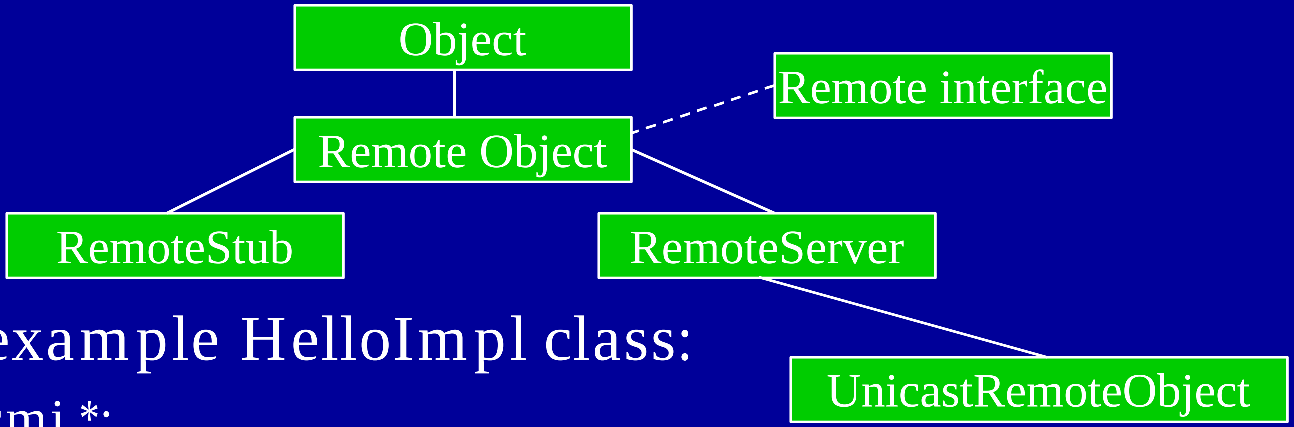
RMI Remote Interface

- In setting up an RMI client and server, the starting point is the interface. This interface gives specifications of all the methods which reside on the server and are available to be called by the client.
- This interface is a subclass of the Remote interface in the Java rmi package, and must available to the compiler on both the client and server.
- Example: A server whose object will have one method, sayHello(), which can be called by the client:

```
public interface Hello extends java.rmi.Remote
{
  String sayHello() throws java.rmi.RemoteException;
}
```

Server Implements the remote object

- All remote servers are a subclass of the class UnicastRemoteObject in the rmi.server package. Extending this class means that it will be a (nonreplicated) remote object that is set up to use the default socket-based transport layer for communication.
- This is the inheritance diagram of the server classes:



- Beginning of example HelloImpl class:

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloImpl extends UnicastRemoteObject
                                        implements Hello
```

Define the constructor for the remote object

- Creating an instance of this class calls the constructor in the same way as for a normal local class. The constructor initializes instance variables of the class.
- In this case, we also call the constructor of the parent class by using the keyword “super”. This call starts the server of the unicastremoteobject listening for requests on the incoming socket. Note that an implementation class must always be prepared to throw an exception if communication resources are not available.

```
private String name;       //instance variable
public HelloImpl (String s) throws java.rmi.RemoteException
   {  super();
      name = s;
    }
```

Provide an implementation for each remote method

- The implementation class must provide a method for each method name specified in the Remote interface. (Other methods may also be given, but they will only be available locally from other server classes.)
- Note that any objects to be passed as parameters or returned as values must implement the java.io.serializable interface. Most of the core Java classes in java.lang and java.util, such as String, are serializable.

```
public String sayHello() throws RemoteException
{
        return “Hello, World!  From” + name;
}
```

Main method: Create an instance and install a Security Manager

- This method will call the constructor of the class to create an instance.
- It will also install a security manager to make sure that any calls initiated by a remote client will not perform any “sensitive” operations, such as loading local classes.

```
public static void main (String args [ ])
{
System.setSecurityManager (new RMISecurityManager());
  try
  {  HelloImpl obj = new HelloImpl(“HelloServer”);
    . . .  / /  name registry code goes here
  } catch (Exception e) { . . . }  / / code to print exception message
}
```

Name Registry

- The rmi registry is another server running on the remote machine - all RMI servers can register names with an object by calling the rebind method.
- The name given to rebind should be a string of the form:

  Naming.rebind("//osprey7:1099/HelloServer", obj);

  where the machine name can default to the current host

  the port number can default to the default registry port, 1099
- For large distributed applications using RMI, a design goal is to minimize the number of names in the registry. The client can obtain the name of one remote object from the registry, and other remote objects from that rmi server can be returned as values to the client. This is called "bootstrapping".
- Example call to rebind for the main method in HelloImpl:

  Naming.rebind("HelloServer", obj);
- Look at full example Hello.java and HelloImpl.java

Client applet or application

- The client must also have a security manager. Applets already have one; applications will make the same call to System.setSecurityManager as the server did.
- The client will look up the server name in the name registry, obtaining a reference to the remote object:

```
String url=“rmi://osprey7.npac.syr.edu/”;
Hello obj = (Hello) Naming.lookup(url +
                                  “HelloServer”);
```

- Then the client can call any method in the interface to this server:

```
obj.sayHello();
```

- Look at Hello.html and HelloApplet.java

Summary of steps for setting up RMI

- 1. Compile the java code.
- 2. Place the interface class extending Remote on the server and the client.
- 3. Place the implementation class extending RemoteObject on the server.
- 4. Generate stubs and skeletons on the server by running the program rmic. Copy the stubs to the client.
- 5. Start the name registry server on the rmi server machine.
- 6. Start the program that creates and registers objects of the implementation class on the rmi server machine.
- 7. Run the client program.
- For more details, see the RMI tutorial at http://www.npac.syr.edu/projects/tutorials/JDBC