Java RMI:
Remote Method Invocation

January 2000

Nancy McCracken

Syracuse University

RMI

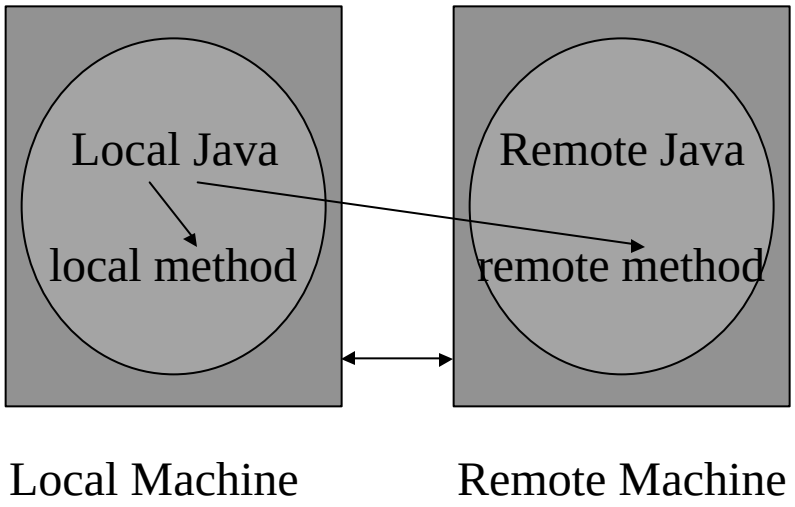
- Java RMI allows the programming of distributed applications across the Internet at the object level. One Java application or applet (the client in this context) can call the methods of an instance, or object, of a class of a Java application (the server in this context) running on another host machine.
- 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 2, Volume II - Advanced Features, Cay Horstmann and Gary Cornell, Prentice-Hall 2000.
  - Java RMI, Troy Bryan Downing, IDG books, 1998.
  - advanced Java networking, Prashant Sridharan, Sunsoft Press, 1997.
  - http:/ / www.javasoft.com/

RMI compared to other networking

- In Java, it is quite easy to establish a socket network connection to another host. Communication takes place via streams. You must program protocols for message interactions and formats yourself.
  - High-level compared with Unix sockets, but low-level compared to RMI.
- Remote Procedure Call (RPC) is a protocol to allow calling remote procedures.
  - Programmers register their application with a host port mapper.
  - Protocols for procedure parameter passing supports a limited number of primitive types.
- Remote distributed objects allow objects to be passed as parameters as well.
  - RMI, CORBA, DCOM (for ActiveX and other Microsoft applications).

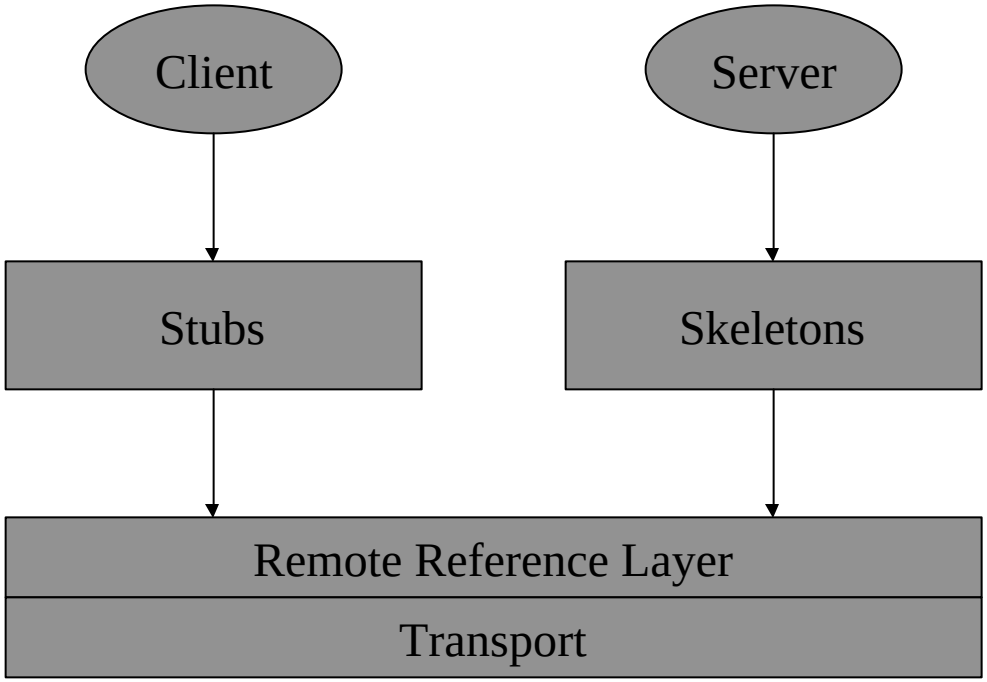
The Java RMI package

- 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.
- Important concepts
  - naming Registry - allows lookup to connect with remote objects
  - Remote interface - specification of remote methods
  - RemoteObjects - allows objects to be distributed
  - RMISecurityManager - to control the use of remote code
  - Serialization - protocol for representin all objects to be passed across the network.



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.

Remote Reference and Transport Layers

- The remote reference layer provides a Stream interface for communication between the stubs and skeletons.
  - It knows the local and remote objects and how to translate to the local and remote name space.
- The transport layer handles all the lower-level network issues.
  - It sets up a connection over a physical socket. This is not necessarily TCP/ IP, but may be UDP or other network protocol.
  - 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.
- The most important 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.
- 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.

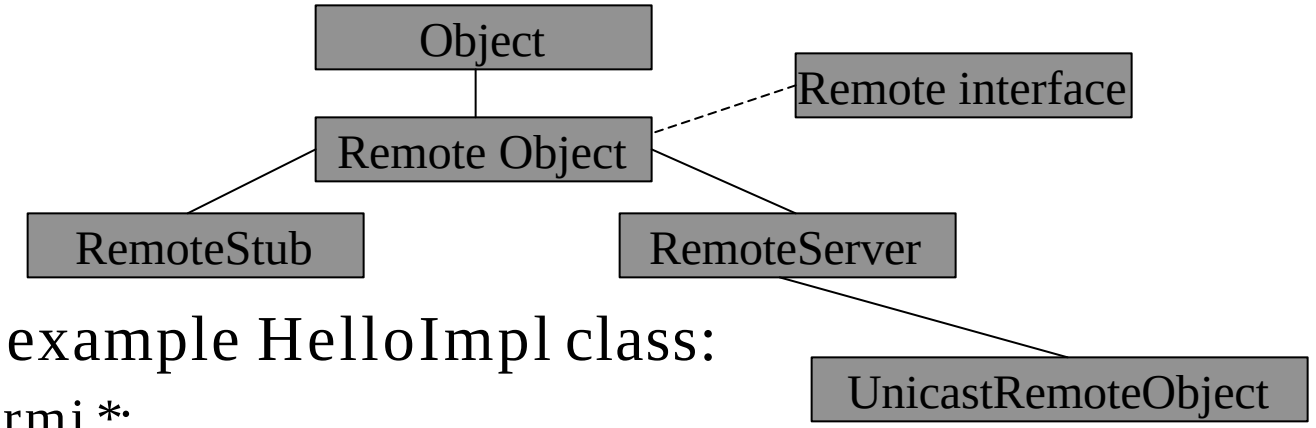
RMI Remote Interface

- 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 be 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.
-

```
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
}
```

RMI Security Manager

- During serialization of an object, the fields and methods are encoded by the protocol and transmitted as data. For the object to be used on the remote side, the class loader must be called to load the code for the methods of that object. Any java program that calls the class loader must have a security manager to check the classes for the security policy of that application.
  - For example, a security manager must make sure that any calls initiated by a remote client will not perform any “sensitive” operations, such as loading local classes.
- RMISecurityManager is the default for RMI - you can write your own security manager.

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 or bind methods. Clients can then use a lookup method to find a service.
- An object can be bound into a naming registry if it is a remote object
  - it must either extend UnicastRemoteObject or made into a remote object by calling UnicastRemoteObject.exportObject().
  - Remote Objects include the stub which will be passed to the client. For this reason, when you start the rmi registry server, you must give the directory where it can find stubs of the remote object.
- 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”.

The rmiregistry

- rmiregistry is the standard naming registry service provided with Java.
  - You can write your own service by implementing the java.rmi.registry interface, including the Naming class methods of bind, rebind, unbind, list, lookup, etc.
- 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
- 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");
```

- Exceptions to this call include
  - NotBoundException, MalformedURLException, RemoteException
- 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.
- 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.

RMIC

- The rmic program provided by Java takes a classfile or list of classfiles that have remote objects to export.
- The options to this program include
  - d the directory in which to place the stubs and skeletons
  - show popup window to show names of methods
  - O optimize (same as regular compiler)
  - keepgenerated keeps the .java file of the generated stubs and skeletons for you to see

Passing Remote Objects

- In the RMI server example, the remote object was registered in the naming registry and then passed to the client on lookup.
- More generally remote objects can be passed as either input or return parameters to remote methods, enabling quite general communications patterns.
- The RMI server example implemented a classic client/ server model where the server provides services encapsulated into methods in a remote interface. Clients initiate communication by calling a remote method.
- But the client can also provide a remote interface and pass itself as a remote object to the server. Then the server can also initiate communication by calling remote methods on the client.
- In remaining pages, we sketch such a collaboration server example. We omit details about setting up threads and synchronization and the server side RMI interface, in order to show the client side interface.

Server Remote Interface

- The collaboration server allows users to connect to the server. It keeps a list of users. When any user posts a message, it distributes it to all the other users.
- public interface ChatServer extends Remote
  {
  / / pass client object to server with id name
  public void register(Chat c, String name) throws RE

  public void postMessage (Message m) throws RE

  public String[] listChatters ( ) throws RE
  }
- Note that RE is an abbreviation for RemoteException.
- The Message class must implement Serializable.

Client Remote Interface

- When the server needs to distribute messages to users, it does so by calling a method in the client's remote interface.
-

```
public interface Chat extends Remote
{
  public void chatNotify ( Message m )  throws RE

  public String getName ( )  throws RE
}
```

Server Implementation

- public class ChatServerImpl extends UnicastRemoteObject
  implements ChatServer

```
{ private Vector chatters = new Vector();
  public ChatServerImpl() throws RE { . . . }

  public void register(Chat c, String name)
  {  chatters.addElement ( . . . c . . . .);   }
  public String[ ] listChatters ( ) { . . . }

  public void postMessage ( Message m)
       {  … c.chatNotify(Message m) …}

  public static void main(String[ ] args)
  {/ /  set security manager, bind registry . . .}
}
```

Client Implementation - Applet

- public class ChatImpl extends Applet implements Chat

```
{  public ChatImpl ( ) throws RE { . . . }
   public void init ( )
   { . . .
      try
      { UnicastRemoteObject.exportObject(this);
         cs = (ChatServer)Naming.lookup( . . .);
         cs.register(this, name);
     }. . .}
   / *  other applet methods for GUI, including calls to
     cs.postMessage(Message m) and cs.listChatters */
   public void chatNotify(Message m)  throws RE
   { / /  display message }
}
```

- use rmic to create stubs and skeletons on the client side