Java RMI

RMI = Remote Method Invocation

Allows a method to be invoked that resides on a different JVM (Java Virtual Machine):

- Either a remote machine
- Or same machine, different processes

- Each process runs on a different Java Virtual Machines (JVM)
- Different address space per process/JVM

RMI provides object-oriented RPC (Remote Procedure Calls)

Participating processes

- **Client**
  - Process that is invoking a method on a remote object
- **Server**
  - Process that owns the remote object
  - To the server, this is a local object

**Object Registry** ([rmiregistry](http://hostname:port/pathname))

- Name server that associates objects with names
- A server registers an object with rmiregistry
  - URL namespace
    - `rmi://hostname:port/pathname`
    - E.g.: `rmi://crapper.pk.org:12345/MyServer`

Classes & Interfaces needed for Java RMI

- **Remote**: for accessing remote methods
  - Used for remote objects
- **Serializable**: for passing parameters to remote methods
  - Used for parameters

Also needed:

- **RemoteException**: network or RMI errors can occur
- **UnicastRemoteObject**: used to export a remote object reference or obtain a stub for a remote object
- **Naming**: methods to interact with the registry

Remote class

- **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**
    - The handle identifies the location of the object
  - If a remote object is passed as a parameter, its handle is passed

Serializable interface

**java.io.Serializable** interface (serializable object)

- Allows an object to be represented as a sequence of bytes
- Allows instances of objects to be copied between address spaces
  - Can be passed as a parameter or be a return value to a remote object
  - Value of object is copied (pass by value)
- Any objects that may be passed as parameters should be defined to implement the **java.io.Serializable** interface

**Good news**: you rarely need to implement anything

- All core Java types already implement the interface
- For your classes, the interface will serialize each variable iteratively
Remote classes

Classes that will be accessed remotely have two parts:
1. interface definition
2. class definition

Remote interface
- This will be the basis for the creation of stub functions
- Must be public
- Must extend java.rmi.Remote
- Every method in the interface must declare that it throws java.rmi.RemoteException

Remote class
- implements Remote interface
- extends java.rmi.server.UnicastRemoteObject

Super-simple example 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 (SampleInterface.java)

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface SampleInterface extends Remote {
    public String invert(String msg) throws RemoteException;
}
```

- Interface is public
- Extends the Remote interface
- Defines methods that will be accessed remotely
  - We have just one method here: invert
- Each method must throw a RemoteException
  - In case things go wrong in the remote method invocation

Define the remote class (Sample.java)

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

public class Sample extends UnicastRemoteObject implements SampleInterface {
    public Sample() throws RemoteException {}  
    public String invert(String m) throws RemoteException {
        // return input message with characters reversed
        return new StringBuffer(m).reverse().toString();
    }
}
```

- Defines the implementation of the remote methods
- It implements the interface we defined
- It extends the java.rmi.server.UnicastRemoteObject class
  - Defines a unicast remote object whose references are valid only while the server process is alive

Next...

- We now have:
  - The remote interface definition: SampleInterface.java
  - The server-side (remote) class: Sample java
- Next, we’ll write the server: SampleServer.java
- Two parts:
  1. Create an instance of the remote class
  2. Register it with the name server (rmiregistry)

Server code (SampleServer.java)

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

public class SampleServer {
    public static void main(String[] args) throws RemoteException {
        try {
            Sample sample = new Sample();
            Naming.rebind("Sample", new Sample());
            // Your server code here...
        } catch (Exception e) {
            // Handle exception
        }
    }
}
```

- Create the object
  ```java
  new Sample()
  ```
- Register it with the name server (rmiregistry)
  ```java
  Naming.rebind("Sample", new Sample())
  ```
- rmiregistry runs on the server
  - The default port is 1099
  - The name is a URL format and can be prefixed with a hostname and port: "//localhost:1099/Server"
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class SampleServer {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage: java SampleServer rmipayrport");
            System.exit(1);
        }
        try {
            System.out.println("binding " + //localhost: + port + "/Sample");
            Naming.rebind(url, new Sample());
            System.out.println("server " + url + " is running...");
        } catch (Exception e) {
            System.out.println("Sample server failed:" + e.getMessage());
        }
    }
}

try {
    // first command-line arg: the port of the rmiregistry
    int port = Integer.parseInt(args[0]);
    // create the URL to contact the rmiregistry
    String url = "//localhost:" + port + "/Sample";
    // register it with rmiregistry
    Naming.rebind(url, new Sample());
    System.out.println("server " + url + " is running...");
} catch (Exception e) {
    System.out.println("Sample server failed:" + e.getMessage());
}

grant {
    permission java.security.AllPermission;
};

public class SampleClient {
    public static void main(String[] args) {
        try {
            // basic argument count check
            if (args.length < 3) {
                System.err.println("usage: java SampleClient hostname port string...");
                System.exit(1);
            }
            int port = Integer.parseInt(args[1]);
            String url = "//" + args[0] + ":" + port + "/Sample";
            System.out.println("looking up " + url);
            SampleInterface sample = (SampleInterface) Naming.lookup(url);
            for (int i = 2; i < args.length; ++i)
                System.out.println(sample.invert(args[i]));
        } catch (Exception e) {
            System.out.println("SampleClient exception: " + e);
        }
    }
}

/ *
 * The example above has a single security policy file. This policy file
 * grants all permissions.
 * 
 * This can be accomplished by adding the following to the security policy file:
 * 
 * grant {
 *     permission java.security.AllPermission;
 * };
 *
 * In the absence of this, the client will fail to connect to the server.
 */

import java.rmi.*;
public class SampleClient {
    public static void main(String[] args) {
        try {
            // basic argument count check
            if (args.length < 3) {
                System.err.println("usage: java SampleClient hostname port string...");
                System.exit(1);
            }
            int port = Integer.parseInt(args[1]);
            String url = "//" + args[0] + ":" + port + "/Sample";
            System.out.println("looking up " + url);
            SampleInterface sample = (SampleInterface) Naming.lookup(url);
            for (int i = 2; i < args.length; ++i)
                System.out.println(sample.invert(args[i]));
        } catch (Exception e) {
            System.out.println("SampleClient exception: " + e);
        }
    }
}
Compile

- Compile the interface and classes:
  ```
  javac SampleInterface.java Sample.java
  ```
- And the client...
  ```
  javac SampleClient.java
  ```
(you can do it all on one command: `javac *.java`)

- Note – Java used to use a separate RPC compiler
  - Since Java 1.5, Java supports the dynamic generation of stub classes at runtime
  - In the past, one had to use an RMI compiler, rmic
- If you want to, you can still use it but it’s not needed

Run

- Start the object registry (in the background):
  ```
  rmiregistry 12345 &
  ```
  - An argument overrides the default port 1099
- Start the server (giving it the port of the rmi registry):
  ```
  CLASSPATH=. java -Djava.security.policy=policy SampleServer 12345
  ```
- Run the client:
  ```
  java SampleClient svrname 12345 testing abcdefgh
  ```
  - Where `svrname` is the name of the server host. For example, `java SampleClient localhost 12345 testing abcdefgh`
  - `12345` is the port number of the name server, rmiregistry, not the actual service!
- See the output:
  ```
  gnitset hgfedcba
  ```

Interfaces

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

Three-layer architecture

- Client program
- 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)

<table>
<thead>
<tr>
<th>Stub functions</th>
<th>Application interaction: marshaling &amp; unmarshaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote reference layer</td>
<td>Handles the creation &amp; management of remote objects. Deals with the semantics of remote requests (how they behave)</td>
</tr>
<tr>
<td>Transport layer</td>
<td>Setting up connections and sending/receiving data</td>
</tr>
</tbody>
</table>
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
  - stub calls RemoteCall.executeCall()
  - causes the remote method invocation to 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

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 end