Java RMI Tutorial

Java Remote Method Invocation (RMI)

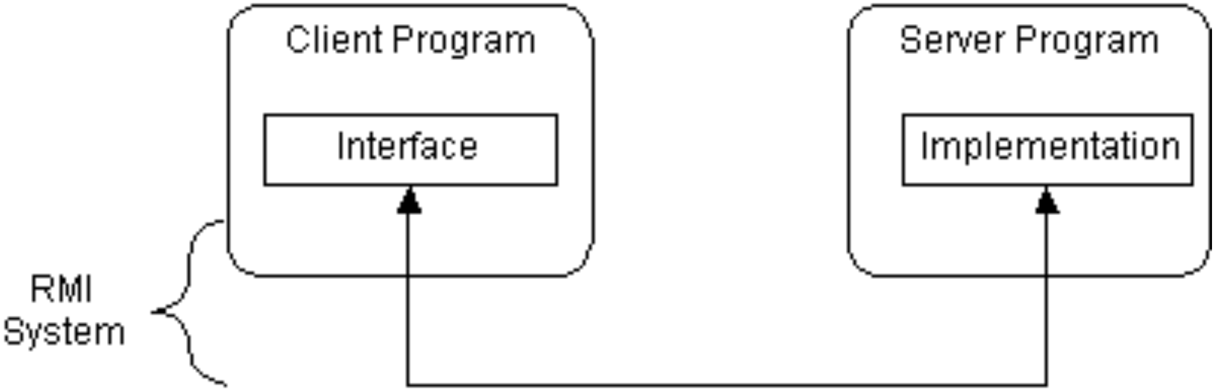
- Java RMI is a mechanism that allow one to invoke a method on an object that exists in **another address space**
  - Different address spaces can exist on the **same** or **different** machines

- Extend the Java object model to support programming with **distributed** objects
  - Make such distributed programming as easy as standard Java programming (recall *transparency* discussed in lectures)
  - Possible to invoke methods on remote objects using exactly the **same syntax** as for local objects

- Use of RMI is visible (explicit) to programmers
  - An object must be aware that it is making a remote call as it must handle `RemoteExceptions`
  - The implementer of a remote object is also aware of its status as the object must implement the `Remote` interface
  - The semantics of parameter passing are also different

Programming with interfaces

- RMI architecture builds on the concept of interfaces
  - Definition of remote object specified by its interface
  - Interfaces define **behaviour** and classes define **implementations**
  - The code defining the behaviour and the code that implements the behaviour can remain separate and run on **separate** JVMs
  - **Clients** are concerned about the **definition** of a service and **servers** are focused on **providing** the service



Parameter passing

- Parameters of a method invocation equate to **input** parameters and the result of the method is the **single output** parameter
  - Input parameters are **marshalled** and sent to the remote object
  - Result is **marshalled** and sent back to the calling object after the execution of the method

- `Serializable` class
  - Objects need to be `serializable` in order to be passed as parameters in RMI
  - Any object that implements the `serializable` interface, is marshalled and **copied by value**
  - Original object remains at the host site
  - The copy made of the object and the original may **diverge**

- However…
  - If the type of a parameter or the result is a *remote interface*, the corresponding argument or result is **passed by reference**

RMI registry

- A naming service that enables the programmer to locate remote interfaces

- A copy of this service must run on any computer offering remote interfaces

- RMIregistry maintains a table mapping URL-style names to interface references

- URL for names is of the form:
  - `rmi://<host_name> [:<name_service_port>] / <service_name>`
  - Default service port 1099; argument only needs to be specified in URL if port different from the default

Java reflection and activation

- Reflection (Java 1.2 and above)
  - Makes it possible to inspect **classes**, **interfaces**, **fields** and **methods** at runtime, without knowing the names of the classes, methods etc. at compile time
  - Reflection can simplify the server side: can be used to implement a **generic dispatcher** at this end of the connection, and **alleviate** the need for individual skeletons

- Activation (Java 1.2 and above)
  - Remote objects that are **not running** can be automatically **activated** on invocation
  - Previously, it was necessary for those objects to **execute continuously** in order to receive invocations
  - Uses `MarshalledObject` for passing **persistence** or **initialisation** data to `Activatable` objects

Simple RMI example

- Scenario
  - Create a simple distributed system that performs the functionality of a remote calculator service
  - Single client, single server
  - Server provides a set of arithmetic methods {add, subtract, multiply, divide & power} that can be **remotely invoked** by the client
  - The server **receives a request** from the client, **performs the arithmetic operation** and then **returns** the result back to the client

- RMI system composed of the following parts
  - 1. An **interface definition** of the remote services that are provided
  - 2. The **implementations** of the remote services
  - 3. **Stub** and **skeleton** files
  - 4. A **server** to **host** the remote services
  - 5. A **client program** that uses the remote services
  - 6. A RMI **Naming service** that allows clients to find the remote services

1. Creating the interface

- Signatures of methods provided by the remote calculator

```
public interface calculator extends java.rmi.Remote {

public long add(long a, long b) throws java.rmi.RemoteException;
public long sub(long a, long b) throws java.rmi.RemoteException;
public long mul(long a, long b) throws java.rmi.RemoteException;
public long div(long a, long b) throws java.rmi.RemoteException;
public long pow(long a, int b) throws java.rmi.RemoteException;
}
```

- Throwing `RemoteException` allows the client to **detect** when an exception is generated **due to** a communication-related problem in the remote call.

2. Implementation of the remote service

- Contains the implementation code for each of the methods identified in the interface
  - Class uses `UnicastRemoteObject` to **link** to the RMI system
  - **States** that this is a **remote object** whose references are only valid while the server hosting it is still alive
  - Must provide a constructor that declares that it **may throw** a `RemoteException` object
  - `super()` activates code in `UnicastRemoteObject` that performs the RMI **linking** and remote object **initialisation**

```
public class calculatorimpl
    extends java.rmi.server.UnicastRemoteObject implements calculator {

public calculatorimpl() throws java.rmi.RemoteException {
        super(); }

public long add(long a, long b) throws java.rmi.RemoteException {
        return a + b; }

public long sub(long a, long b) throws java.rmi.RemoteException {
        return a - b; }
```

2. Implementation of the remote service (cont.)

```
public long mul(long a, long b) throws java.rmi.RemoteException {
        return a * b;
    }

public long div(long a, long b) throws java.rmi.RemoteException {
        return a / b;
    }

public long pow(long a, int b) throws java.rmi.RemoteException {

       if (b==0)
               return 1;
       else
               return a*pow(a, b-1);
    }
}
```

3. Stub and skeleton files

- Java 1.5 and later, don’t need to generate stubs and skeletons
  - Stubs are **generated on the fly** by the server and sent to the client when needed
  - Skeletons **replaced** by a **generic dispatcher** on the server side based on reflection (since Java 1.2)

- When using older Java versions
  - Or when server needs to support clients written in older versions
  - Need to **explicitly** generate stubs with RMI compiler (`rmic`)
  - Manually install them in clients’ *classpath*, or
  - Make them **downloadable** by clients from the server

4. Create the host server

- After creating the implementation class for remote object that provides the arithmetic methods, need to create a server to **host** this object
  - Construct an **instance** of the object
  - **Bind** it to the naming service
  - In this demonstration we use the local machine to host both the client and the server
    - They can equally run on separate machines

4. Create the host server (cont.)

```
import java.rmi.Naming; //Import naming classes to bind to rmiregistry

public class calculatorserver {

public calculatorserver() {

//N.b. it is possible to host multiple objects on a server
//by repeating the following method.

try {
        calculator c = new calculatorimpl();
        Naming.rebind("rmi://localhost/CalculatorService", c);
     } catch (Exception e) {
       System.out.println("Server Error: " + e);
     }
} // end of calculatorserver constructor

 public static void main(String args[]) {
        new calculatorserver();
   }
}
```

5. Creating the client program

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class calculatorclient {

  public static void main(String[] args) {

   try {

// Create the reference to the remote object through the rmiregistry
        calculator c = (calculator)
               Naming.lookup("rmi://localhost/CalculatorService");

            // Now use the reference c to call remote methods
            System.out.println("3+21="+ c.add(3, 21) );
             System.out.println("18-9="+ c.sub(18, 9) );
             System.out.println("4*17="+ c.mul(4, 17) );
             System.out.println("70/10="+ c.div(70, 10) );
            System.out.println("2^5="+ c.pow(2, 5) );
         }
```

5. Creating the client program (cont.)

```
// Catch the exceptions that may occur – bad URL, Remote exception
// Not bound exception or the arithmetic exception that may occur in
// one of the methods creates an arithmetic error (e.g. divide by zero)

        catch (MalformedURLException murle) {
             System.out.println(“MalformedURLException");
            System.out.println(murle);
        }
        catch (RemoteException re) {
            System.out.println("RemoteException");
            System.out.println(re);
        }
        catch (NotBoundException nbe) {
            System.out.println("NotBoundException");
            System.out.println(nbe);
        }
        catch (java.lang.ArithmeticException ae) {
            System.out.println("java.lang.ArithmeticException");
            System.out.println(ae);
        }
    }
}
```

Running the RMI system

- Compile the classes
  - `javac *.java`
- Start the RMI naming service
  - UNIX: `rmiregistry &`
  - Windows: `start rmiregistry`
- Run the server and client programs
  - `java calculatorserver`
  - `java calculatorclient`

Java security

- One of the most common problems one encounters with RMI is a failure due to security constraints
  - A Java program may specify a **security manager** that determines its **security policy**
  - A program will not have any security manager unless one is specified
  - Security policy set by **constructing** a `SecurityManager` object and **calling** the `setSecurityManager` method of the System class

```
System.setSecurityManager(new RMISecurityManager());
```

- Certain operations **require** that there be a security manager
  - E.g. RMI will download a `Serializable` class from another machine **only** if there is a security manager
  - The security manager will have **to permit** the downloading of the class from that machine
  - A security manager also needs a security policy to act upon (specified in a ‘policy file’)

Java security (cont.)

- Default security manager uses a policy that is defined in a collection of policy files (usually in jre*/lib/security)
  - If we want to grant additional permissions, then we can specify them in a policy file and then request that they be loaded using runtime options such as the following:
  - `java -Djava.security.manager -Djava.security.policy=`*`policy-file`*` MyClass`
  - To override default security policy with own:
  - `java -Djava.security.manager -Djava.security.policy==`*`policy-file`*` MyClass`

- Example policy files

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

```
grant codeBase "file:C:/RMI/-" {
   // grant all permissions of any kind to code
  // residing in the RMI directory on the C: drive
  permission java.security.AllPermission;
  };
```

Dynamic code downloading (2)

- Dynamic code downloading through a `codebase`
  - A source, or a place from which to load classes into a JVM
  - `CLASSPATH` can be seen as a local `codebase`
  - The `java.rmi.server.codebase` property value represents one or more URL locations from which client can download stubs
  - Can be `http://`, `ftp://`, `file:// (/)` (generally requires client and server residing on same physical hosts, or over a DFS)

- Examples
  - `-Djava.rmi.server.codebase=http://webvector/export/`
  - `-Djava.rmi.server.codebase=http://webline/pub/stuff.jar`
  - `-Djava.rmi.server.codebase="http://webfront/myStuff.jar http://webwave/myOtherStuff.jar"`

- For more information you can look at
  - http://download.oracle.com/javase/1.4.2/docs/guide/rmi/codebase.html

TemplateIF.java

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

/**
 * template interface to remote object
 */

public interface TemplateIF extends Remote {
    public Type1 methodName1(Arguments1) throws RemoteException;
        *
        *
        *
}
```

TemplateIFServant.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

class TemplateIFServant extends UnicastRemoteObject
                        implements TemplateIF {

    public TemplateIFServant() throws RemoteException {
        // whatever initialization you must do for this object
    }

    public Type1 methodName1(Arguments1) throws RemoteException {
        // body of method
    }

        *
        *
        *
}
```

TemplateIFServer.java

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;

class TemplateIFServer {
    public static void main(String args[]) {
        try {
            TemplateIFServant tis = new TemplateIFServant();
            System.out.format("Created server, now advertising it\n");
            Registry reg = LocateRegistry.getRegistry("localhost", 1099);
            reg.rebind("templateIFServer", tis);
            System.out.format("Advertising completed\n");
        } catch (Exception e) {
            System.out.format("templateIFServer: an exception occurred”);
            System.out.format(“ when attempting to export the service –”);
            System.out.format(“ %s\n", e.getMessage());
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

TemplateIFClient.java

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;

class TemplateIFClient {

    TemplateIF tif = null;

    public static void main(String args[]) {
        try {
            System.out.format("Client starting\n");
            Registry reg = LocateRegistry.getRegistry("localhost", 1099);
            Object o = reg.lookup("templateIFServer");
            tif = (TemplateIF)o;
        } catch (Exception e) {
            System.out.format("Error in locating templateIFServer from”);
            System.out.format(“ registry\n");
            e.printStackTrace();
            System.exit(1);
        }
        // code to use tif
    }
}
```