JAVA RMI

- Remote Method Invocation (RMI)
  - This is a Java-Based mechanism for distributed object computing.
  - RMI enables the distribution of work to other Java objects residing in other processes or on other machines.
  - The objects in one Java Virtual Machine (JVM) are allowed to seamlessly invoke methods on objects in a remote JVM.
  - To call a method of a remote object we must first get a reference to that object.

JAVA RMI

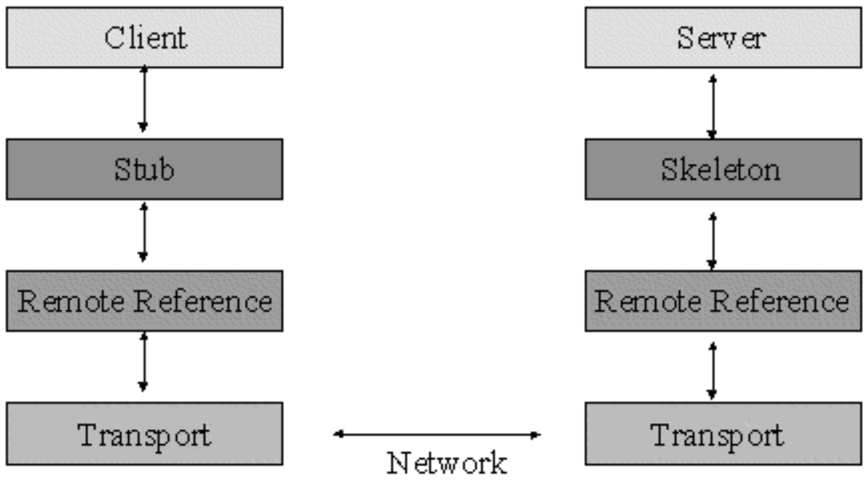
- This reference may be obtained:
  - From the registry name facility.
  - By receiving the reference as an argument or return value of a method call.
- Clients can call a remote object in a server that itself is a client of another server.
- Parameters of method calls are passed as serialised objects.
  - Types are not truncated - therefore, object-oriented polymorphism is supported
  - Parameters are passed by value (deep copy) - therefore object behaviour can be passed

JAVA RMI

- The Java Object Model is still supported with distributed (remote) objects.
- A reference to a remote object can be passed to or returned from local and remote objects.
- Remote object references are passed by reference - therefore the whole object is not always downloaded:
  - Objects that implement the Remote interface are passed as a remote reference.
  - Other objects are passed by value (using object serialisation).

JAVA RMI

Java RMI Architecture



JAVA RMI

- The client obtains a reference for a remote object by calling:
  - *Naming.lookup(//URL/registered name)*
  - A method which returns a reference to another remote object.
- Methods of the remote object may then be called by the client:
  - This call is actually to the stub which represents the remote object.
  - The stub packages the arguments (marshalling) into a data stream (to be sent across the network).

JAVA RMI

- On the implementation side:
  - The skeleton unmarshals the argument, calls the method, marshals the return value and sends it back.
  - The stub unmarshals the return value and returns it to the caller.
- The RMI layer sits on top of the JVM and this allows it to use the following functionality:
  - Java Garbage Collection of Remote Objects.
  - Java Security - a security manager may be set for the server.
  - Java Class Loading.

JAVA RMI

» Steps to creating an RMI application

- Define the interfaces to your remote objects.
- Implement the remote object classes.
- Write the main client and server programs (some examples follow).
- Create the stub & skeleton classes by running the *rmic* compiler on the remote implementation classes.
- Start the *rmiregistry* (if not already started).
- Start the server application.
- Start client (which obtains some initial object refs.)
- The client application/applet may then call object methods in the remote (server) program.

JAVA RMI

» Example Program

```
// Remote Object has a single method that is passed
// the name of a country and returns the capital city.
import java.rmi.*;

public interface CityServer extends Remote
{
    String getCapital(String Country) throws
            RemoteException;
}
```

JAVA RMI

» Server Implementation

```
import java.rmi.*;
import java.rmi.server.*;
public class CityServerImpl
        extends UnicastRemoteObject
        implements CityServer
{
    // constructor is required in RMI
    CityServerImpl() throws RemoteException
    {
        super();     // call the parent constructor
    }
```

JAVA RMI

```
// Remote method we are implementing!
public String getCapital(String country) throws
        RemoteException
{
       System.out.println("Sending return string now
        - country requested: " + country);
       if (country.toLowerCase().compareTo(“usa")
        == 0)
           return "Washington";
        else if
        (country.toLowerCase().compareTo(“ireland")
        == 0)
        return "Dublin";
```

JAVA RMI

```
          else if
            (country.toLowerCase().compareTo(“france")
            == 0)
          return "Paris";
          return "Don't know that one!";
 }

// main is required because the server is standalone
 public static void main(String args[])
 {
     try
     {
```

JAVA RMI

```
// First reset our Security manager
System.setSecurityManager(new
 RMISecurityManager());
 System.out.println("Security manager set");

// Create an instance of the local object
CityServerImpl cityServer = new
 CityServerImpl();
 System.out.println("Instance of City Server
             created");

// Put the server object into the Registry
```

JAVA RMI

```
            Naming.rebind("Capitals", cityServer);
            System.out.println("Name rebind completed");
            System.out.println("Server ready for
             requests!");
        }
        catch(Exception exc)
        {
            System.out.println("Error in main - " +
             exc.toString());
        }
    }
}
```

JAVA RMI

» Client Implementation

```
public class CityClient
{
   public static void main (String args[])
   {
            CityServer cities = (CityServer)
                    Naming.lookup(“//localhost/Capitals”);
            try {
                    String capital = cities.getCapital(“USA”);
                    System.out.println(capital); }
            catch (Exception e) {}
}  }
```

JAVA RMI

- » Class RemoteException
  - – No distributed system can mask communication failures:
    - ● Method semantics should include failure possibilities.
    - ● Every RMI remote method must declare the exception *RemoteException* in its throw clause.
    - ● This exception is thrown when method invocation or return fails.
    - ● The Java compiler requires failures to be handled (no choice here).

JAVA RMI

» Implementing a Remote Object

- Implementation class usually extends the RMI class *UnicastRemoteObject*:
  - This indicates that the implementation class is used to create a single (nonreplicated) remote object that uses RMI's default sockets based transport for communication.
- If you choose to extend a remote object from a nonremote class:
  - You need to explicitly export the remote object by calling the method *UnicastRemoteObject.exportObject()*.

JAVA RMI

- » Security Manager
  - – The main method of the service first needs to create and install a security manager:
    - Either the RMISecurityManager or one that you have defined yourself.
    - A security manager needs to be running so that it can guarantee that the classes loaded do not perform "sensitive" operations.
  - – If no security manager is specified, no class loading for RMI classes, local or otherwise, is allowed.

JAVA RMI

» Making Code Available

- Make classes available via a web server (or your classpath):
  - E.g. copy them into your public html directory.
- Alternatively, you could have compiled your files directly into your public html directory:
  - *javac -d ~des/public_html City*.java*
  - *rmic -d ~des/public_html CityServerImpl*
- The files generated by rmic (in this case) are:
  - CityServerImpl_Stub.class
  - CityServerImpl_Skel.class

JAVA RMI

» Poylmorphic Distributed Computing

- Ability to recognise (at runtime) the actual implementation type of a particular interface.
- We will use example of a remote object that is used to compute arbitrary tasks:
  - Client sends task object to compute server.
  - Compute server runs task and returns result.
  - RMI loads task code dynamically in server.
- This example shows polymorphism on the server - it will also work on the client e.g.:
  - Server returns a particular interface implementation.

JAVA RMI

» The Task

– Simple interface that defines an arbitrary task to compute:

```
public interface Task extends Serializable
{
    Object run();
}
```

JAVA RMI

» Define a Remote Interface

```
import java.rmi.*;

public interface Compute extends Remote
{
    Object runTask(Task t)
            throws RemoteException;
}
```

JAVA RMI

» Notes on the Compute Interface

- A task may create a *Remote* object on the server and return a reference to that object:
  - The *Remote* object will be garbage collected when the returned reference is dropped (assuming no one else is handed a copy of the reference).
- A task may create a *Serializable* object and return a copy of that object:
  - The original object will be locally garbage collected when the Task ends.
- If the task creates an object that is neither a marshalling exception will be thrown.

JAVA RMI

» Implementation

– As in the previous example, for the peer-to-peer compute server implementation:

- Extend the *UnicastRemoteObject* class.
- Implement methods of remote interface.
- Create and install a security manager.
- Create remote object and bind in a name facility.

– On the client side:

- Create tasks to be executed.
- Lookup the compute service by name.
- Send tasks to compute service and print results.

JAVA RMI

» The Compute Server

```
import java.rmi.*;
import java.rmi.server.*;
public class ComputeServer extends
   UnicastRemoteObject implements Compute
{
      public ComputeServer()
                  throws RemoteException {}
      public Object runTask(Task t)    {
                  return t.run();
      }
      // …
```

JAVA RMI

» The main Method

```
public static void main(String args[])
{
      System.setSecurityManager(
                  new RMISecurityManager());
      try  {
         ComputeServer cs = new ComputeServer();
         Naming.rebind("Computer", cs);
      }   catch (Exception e)   {   // Exception Handling  }
}
```

JAVA RMI

» Task to Compute PI

```
public class Pi implements Task
{
    private int places;
    public Pi (int places)   {
            this.places = places;
    }
    public Object run()   {
            // Compute Pi
            return result;
    }
}
```

JAVA RMI

» Task to Compute a FFT

```
public class FFT implements Task
{
    public FFT (args …)   {
            // set FFT args …
    }
    public Object run()   {
            // Compute the FFT
            return result;
    }
}
```

JAVA RMI

» The Client

```
Compute comp = (Compute) Naming.Lookup(
     “//www.it.nuigalway.ie/Computer);

Pi pi = new Pi(100);
FFT fft = new FFT(args…);

Object piResult = comp.runTask(pi);
Object fftResult = comp.runTask(fft);

// Print Results ...
```

JAVA RMI

» Conclusion

– RMI is flexible and allows us to:

- Pass objects (both *Remote* and *Serializable*) by exact type rather than declared type
- Download code to introduce extended functionality in both client and server
- However…it is Java only and it has been superseded by REST and SOAP as the de-facto standards for communicating with remote services
- But…RMI is still worth learning to help understand concepts around distributed objects and distributed systems architecture