Remote Method Invocation

Distributed Computing

- *Distributed Computing* involves the design and implementation of applications as a set of cooperating software entities (processes, threads, objects) that are distributed across a network of machines
- Advantages to Distributed Computing
  - ⇒ Performance
  - ⇒ Scalability
  - ⇒ Resource Sharing
  - ⇒ Fault Tolerance
- Difficulties in developing Distributed Computing systems
  - ⇒ Latency
  - ⇒ Synchronization
  - ⇒ Partial Failure

Client-Server Programming

- Client-Server Model
  - ⇨ Client - entity that makes a request for a service
  - ⇨ Server - entity that responds to a request and provides a service

- The predominant networking protocol in use today is the Internet Protocol (IP). The main API for writing client-server programs using IP is the Berkeley socket API.

Client-Server Programming

- Dealing with all of the details of the socket library calls can be tedious. (See, for example, Stevens' *Unix Network Programming*.)
- The java.net package provides classes to abstract away many of the details of socket-level programming, making it simple to write client-server applications

Client Example

```
import java.net.*;
import java.io.*;

/**
 * Client Program.
 * Connects to a server which converts text to uppercase.
 * Server responds on port 2345.
 * Server host specified on command line: java Client server_host
 */
public class Client {
  public static void main(String args[]) {
    Socket s;
    String host;
    int port = 2345;
    DataInputStream is;
    DataInputStream ui;
    PrintStream os;
    String theLine;
```


Client Example (Continued)

```
    host = args[0];

    try {
      s = new Socket(host, port);

      is = new DataInputStream(s.getInputStream());
      os = new PrintStream(s.getOutputStream());
      ui = new DataInputStream(System.in);
      System.out.println("Enter Data");
      while(true) {
        theLine = ui.readLine();
        if (theLine.equals("end"))
          break;
        os.println(theLine);
        System.out.println(is.readLine());
      }
```

Client Example (Continued)

```
      os.close();
      is.close();
      ui.close();
      s.close();
    }
    catch(UnknownHostException e) {
      System.out.println(”Can’t find " + host);
    }
    catch(SocketException e) {
      System.out.println("Could not connect to ” + host);
    }
    catch(IOException e) {
      System.out.println(e);
    }
  }
}
```


Server Example

```
import java.net.*;
import java.io.*;

/**
 * Server Program.
 * Converts incoming text to uppercase and sends converted
 * text back to client.
 * Accepts connection requests on port 2345.
 */
public class Server {
  public static void main(String args[]) {
    ServerSocket theServer;
    Socket con;
    PrintStream ps;
    DataInputStream dis;
    String input;
    int port = 2345;
    boolean flag = true;
```

Server Example (Continued)

```
    try {
      theServer = new ServerSocket(port);
      con = theServer.accept();
      dis = new DataInputStream(con.getInputStream());
      ps = new PrintStream (con.getOutputStream());
      while(flag == true) {
        input = dis.readLine();
        if ( input == null ) break;
        ps.println(uppers(input));
      }
      con.close();
      dis.close();
      ps.close();
      theServer.close();
    }
    catch(NullPointerException e){
      System.out.println("NPE" + e.getMessage());
    }
```


Server Example (Continued)

```
    catch(IOException e) {
      System.out.println(e);
    }
  }

  public static String uppers(String input) {
    char let;
    StringBuffer sb = new StringBuffer(input);
    for (int i = 0; i < sb.length(); i++) {
      let = sb.charAt(i);
      let = Character.toUpperCase(let);
      sb.setCharAt(i,let);
    }
    return sb.toString();
  }
}
```

Remote Procedures

- In the previous example, the client communicated to the server over a socket connection using a protocol known to both parties
- But both the client and server had to be aware of the socket level details
- Wouldn't it be nice if even these details were abstracted away and the request to the server looked like a local procedure call from the viewpoint of the client?
- That's the idea behind a Remote Procedure Call (RPC), a technology introduced in the late 1970's
- Two RPC specifications:
  - ⇒ SUN's Open Network Computing (ONC) RPC
  - ⇒ OSF's Distributed Computing Environment (DCE) RPC

Distributed Object Technology

- But RPC is not object-oriented. In the OO world, we'd like to have distributed objects and remote method calls.
- While there are many Distributed Object Technologies available today, three are widely available:
  - ⇒ RMI
  - ⇒ CORBA
  - ⇒ SOAP
- Remote Method Invocation (RMI)
  - ⇒ Developed by SUN
  - ⇒ Available as part of the core Java API
  - ⇒ Java-centric
  - ⇒ Object interfaces defined as Java interfaces
  - ⇒ Uses object serialization

Distributed Object Technology

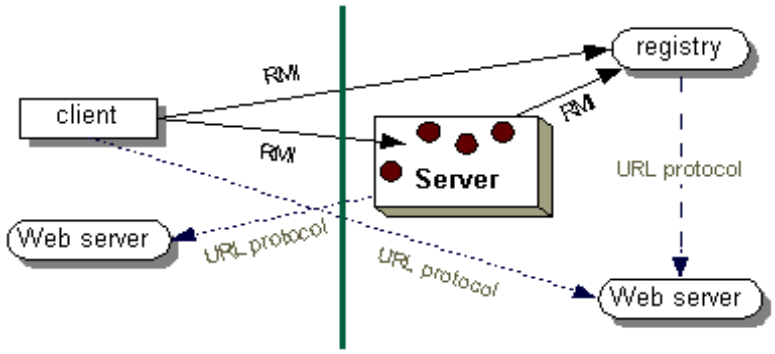
- Common Object Request Broker Architecture (CORBA)
  - ⇒ Developed by the Object Management Group (OMG)
  - ⇒ Language and platform independent
  - ⇒ Object interfaces defined in an Interface Definition Language (IDL)
  - ⇒ An Object Request Broker (ORB) facilitates the client-server request/response action
  - ⇒ ORBs communicate via a binary protocol called the Internet Inter-ORB Protocol (IIOP)
- SOAP
  - ⇒ Simple Object Access Protocol
  - ⇒ XML-Based
  - ⇒ Developed from an earlier spec called XML-RPC
  - ⇒ Standardized by the W3C
  - ⇒ Many implementations available

RMI

- Provides a distributed object capability for Java applications
- Allows a Java method to obtain a reference to a remote object and invoke methods of the remote object nearly as easily as if the remote object existed locally
- The remote object can be in another JVM on the same host or on different hosts across the network
- Uses object serialization to marshal and unmarshal method arguments
- Supports the dynamic downloading of required class files across the network

RMI Application

- RMI Application



RMI Stubs And Skeletons

- RMI uses stub and skeleton objects to provide the connection between the client and the remote object
- A *stub* is a *proxy* for a remote object which is responsible for forwarding method invocations from the client to the server where the actual remote object implementation resides
- A client's reference to a remote object, therefore, is actually a reference to a local stub. The client has a local copy of the stub object.
- A *skeleton* is a server-side object which contains a method that dispatches calls to the actual remote object implementation
- A remote object has an associated local skeleton object to dispatch remote calls to it

RMI Stubs And Skeletons

- Note: Java 2 (JDK1.2) does not require an explicit skeleton class. The skeleton object is automatically provided on the server side.
- A method can get a reference to a remote object
  - ⇒ by looking up the remote object in some directory service. RMI provides a simple directory service called the RMI registry for this purpose.
  - ⇒ by receiving the remote object reference as a method argument or return value

Developing An RMI Application

- An object becomes remote-enabled by implementing a remote interface, which has these characteristics:
  - ⇒ A remote interface extends the interface java.rmi.Remote
  - ⇒ Each method of the interface declares java.rmi.RemoteException in its throws clause, in addition to any application-specific exceptions
- Steps To Develop An RMI Application
  - ⇒ 1. Design and implement the components of your distributed application
    - ➡ Define the remote interface(s)
    - ➡ Implement the remote object(s)
    - ➡ Implement the client(s)
  - ⇒ 2. Compile sources and generate stubs (and skeletons)
  - ⇒ 3. Make required classes network accessible
  - ⇒ 4. Run the application

RMI Example 1

- The classic “Hello, World” Example using RMI!
- First, define the desired remote interface:

```
import java.rmi.*;

/**
 * Hello Interface.
 */
public interface IHello extends Remote {
  public String sayHello() throws RemoteException;
}
```

- A class that implements this remote interface can be used as a remote object. Clients can remotely invoke the sayHello() method which will return the string “Hello, World” to the client.

RMI Example 1 (Continued)

- Next, provide an implementation of the remote object
- We’ll implement the remote object as a server
- The remote object server implementation should:
  - ⇒ Declare the remote interfaces being implemented
  - ⇒ Define the constructor for the remote object
  - ⇒ Provide an implementation for each remote method in the remote interfaces
  - ⇒ Create and install a security manager
  - ⇒ Create one or more instances of a remote object
  - ⇒ Register at least one of the remote objects with the RMI remote object registry (or some other naming service), for bootstrapping purposes
- To make things simple, our remote object implementation will extend java.rmi.server.UnicastRemoteObject. This class provides for the “exporting” of a remote object by listening for incoming calls to the remote object on an anonymous port.

RMI Example 1 (Continued)

- Here's the server for our remote object:

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

// Hello  Server.
public class HelloServer extends UnicastRemoteObject
  implements IHello {

  private String name;

  public HelloServer(String name) throws RemoteException {
    super();
    this.name = name;
  }

  public String sayHello() {return "Hello, World!";}
```


RMI Example 1 (Continued)

```
  public static void main(String[] args) {
    // Install a security manager!
    System.setSecurityManager(new RMISecurityManager());

    try {
      // Create the remote object.
      HelloServer obj = new HelloServer("HelloServer");

      // Register the remote object as "HelloServer".
      Naming.rebind("rmi://serverhost/HelloServer", obj);
      System.out.println("HelloServer bound in registry!");
    }
    catch(Exception e) {
      System.out.println("HelloServer error: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

RMI Example 1 (Continued)

- Next, we need to write our client application:

```
import java.rmi.*;

// Hello Client.
public class HelloClient {

  public static void main(String[] args) {

    // Install a security manager!
    System.setSecurityManager(new RMISecurityManager());

    try {
      // Get a reference to the remote object.
      IHello server =
         (IHello)Naming.lookup("rmi://serverhost/HelloServer");
      System.out.println("Bound to: " + server);
```


RMI Example 1 (Continued)

```
      //Invoke the remote method.
      System.out.println(server.sayHello());
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

}
```

RMI Example 1 (Continued)

- Now we can compile the client and server code:

```
javac IHello.java
javac HelloServer.java
javac HelloClient.java
```

- We next use the rmic utility to generate the required stub and skeleton classes:

```
rmic HelloServer
```

- This generates the stub and skeleton classes:

```
HelloServer_Stub.class
HelloServer_Skel.class (Not needed in Java 2)
```


RMI Example 1 (Continued)

- Our next step would be to make the class files network accessible. For the moment, let's assume that all these class files are available locally to both the client and the server via their CLASSPATH. That way we do not have to worry about dynamic class downloading over the network. We'll see in the next example how to properly handle that situation.
- The files that the client must have in its CLASSPATH are:

```
IHello.class
HelloClient.class
HelloServer_Stub.class
```

- The files that the server must have in its CLASSPATH are:

```
IHello.class
HelloServer.class
HelloServer_Stub.class
HelloServer_Skel.class (Not needed in Java 2)
```

RMI Example 1 (Continued)

- If you run this example in Java 2, you need a security policy file that allows the downloading of class files
- Here is an example policy file that allows anything!

```
grant {
    permission java.security.AllPermission;
};
```

- Here's a policy file that allows the program to connect to or accept connections from any host on ports greater than 1024 and to connect to any host on port 80 (the default HTTP port):

```
grant {
    permission java.net.SocketPermission "*:1024-65535",
        "connect,accept";
    permission java.net.SocketPermission "*:80", "connect";
};
```


RMI Example 1 (Continued)

- Now, we are ready to run the application:
- On the server:
  - ⇨ Start the rmiregistry:

```
rmiregistry &
```

  - ⇨ Start the server:

```
java -Djava.security.policy=policy HelloServer
```

- On the client:
  - ⇨ Start the client:

```
java -Djava.security.policy=policy HelloClient
```

- Get this wonderful output on the client:

```
Hello, World!
```

RMI Example 2

- The server in this example implements a generic compute engine
- The idea is that a client has some CPU-intensive job to do, but does not have the horsepower to do it. So the client encapsulates the task to be done as an object and sends it over to a server to be executed. The compute engine on the server runs the job and returns the results to the client.
- The compute engine on the server is totally generic and can execute any kind of task requested by the client
- This example illustrates one of our Design Patterns. Which one??

RMI Example 2 (Continued)

- First, define the remote interface:

```
package compute;
import java.rmi.*;

/**
 * Generic Compute Interface.
 */
public interface Compute extends Remote {
  Object executeTask(Task t) throws RemoteException;
}
```

RMI Example 2 (Continued)

- Here's the generic task interface we'll need for this example:

```
package compute;
import java.io.Serializable;

/**
 * Task Interface.
 */
public interface Task extends Serializable {
  Object execute();
}
```


RMI Example 2 (Continued)

- Here's the remote server:

```
package engine;

import java.rmi.*;
import java.rmi.server.*;
import compute.*;

/**
 * Server that executes a task specified in a Task object.
 */
public class ComputeEngine extends UnicastRemoteObject
  implements Compute {

  public ComputeEngine() throws RemoteException {
    super();
  }
```

RMI Example 2 (Continued)

```
  public Object executeTask(Task t) {
    return t.execute();
  }

  public static void main(String[] args) {

    // Install a security manager!
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }

    // Create the remote object.
    // Register the remote object as "Compute".
    String name = "rmi://serverhost/Compute";
```


RMI Example 2 (Continued)

```
    try {
      Compute engine = new ComputeEngine();
      Naming.rebind(name, engine);
      System.out.println("ComputeEngine bound");
    }
    catch (Exception e) {
      System.err.println("ComputeEngine exception: " +
                         e.getMessage());
      e.printStackTrace();
    }
  }

}
```

RMI Example 2 (Continued)

- Here's a client which asks the server to compute the value of pi:

```
package client;

import java.rmi.*;
import java.math.*;
import compute.*;

/**
 * Client that asks the Generic Compute Server to compute pi.
 * The first command-line argument is the server hostname.
 * The second command-line argument is the number of required
 * digits after the decimal point for the computation.
 */
public class ComputePi {
```


RMI Example 2 (Continued)

```
  public static void main(String args[]) {

    // Install a security manager!
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }

    try {
      String name = "//" + args[0] + "/Compute";

      // Get a reference to the remote object from the registry.
      Compute comp = (Compute) Naming.lookup(name);

      // Create a Task object.
      Pi task = new Pi(Integer.parseInt(args[1]));
```

RMI Example 2 (Continued)

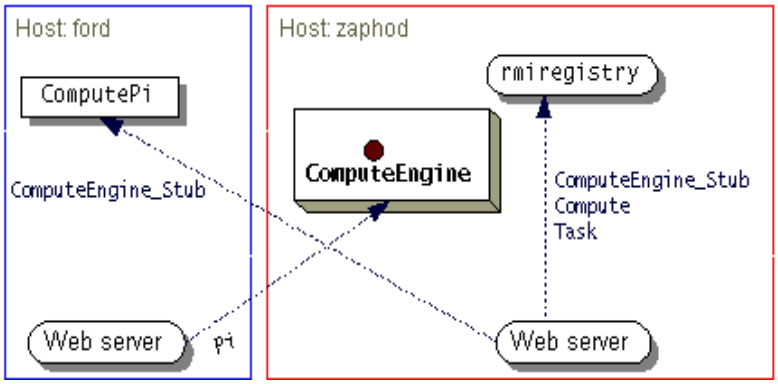
```
        // Ask the server to perform the computation.
        BigDecimal pi = (BigDecimal)(comp.executeTask(task));
        System.out.println(pi);
      }
      catch (Exception e) {
        System.err.println("ComputePi exception: " +
                           e.getMessage());
        e.printStackTrace();
      }
    }

}
```


RMI Example 2 (Continued)

- Here's part of the Pi class used in the client:

```
package client;
import compute.*;
import java.math.*;

public class Pi implements Task {

  private int digits;

  public Pi(int digits) {this.digits = digits;}

  public Object execute() {
    // Pi calculation code goes here!
  }

}
```

RMI Example 2 (Continued)



RMI Example 2 (Continued)

- Now we are ready to build the application. First, build a jar file of the interface classes:

```
cd /home/waldo/src
javac compute/Compute.java
javac compute/Task.java
jar cvf compute.jar compute/*.class
```

- Next, build the server classes:
  - ⇨ Let's say that, ann, the developer of the ComputeEngine class, has placed ComputeEngine.java in the /home/ann/src/engine directory and is deploying the class files for clients to use in a subdirectory of her public_html directory, /home/ann/public_html/classes. Let's assume that the compute.jar file is located in the directory /home/ann/public_html/classes. To compile the ComputeEngine class, your class path must include the compute.jar file and the source directory itself.

RMI Example 2 (Continued)

```
setenv CLASSPATH
  /home/ann/src:/home/ann/public_html/classes/compute.jar
cd /home/ann/src
javac engine/ComputeEngine.java
rmic -d . engine.ComputeEngine
mkdir /home/ann/public_html/classes/engine
cp engine/ComputeEngine*.class
  /home/ann/public_html/classes/engine
```

⇨ The -d option tells the rmic compiler to place the generated class files, ComputeEngine_Stub and ComputeEngine_Skel, in the directory /home/ann/src/engine. You also need to make the stubs and skeletons network accessible, so you must copy the stub and skeleton class to the public_html/classes area.

RMI Example 2 (Continued)

⇨ Since the ComputeEngine's stub implements the Compute interface, which refers to the Task interface, you need to make these two interface class files network accessible along with the stub. So, the final step is to unpack the compute.jar file in the directory /home/ann/public_html/classes to make the Compute and Task interfaces available for downloading.

```
cd /home/ann/public_html/classes
jar xvf compute.jar
```

RMI Example 2 (Continued)

- Finally, build the client classes:
  - ⇨ Let's assume that user jones has created the client code in the directory /home/jones/src/client and will deploy the Pi class (so that it can be downloaded to the compute engine) in the network-accessible directory /home/jones/public_html/classes (also available via some web servers as http://host/~jones/classes/). The two client-side classes are contained in the files Pi.java and ComputePi.java in the client subdirectory.
  - ⇨ In order to build the client code, you need the compute.jar file that contains the Compute and Task interfaces that the client uses. Let's say that the compute.jar file is located in /home/jones/public_html/classes. The client classes can be built as follows:

RMI Example 2 (Continued)

```
setenv CLASSPATH
    /home/jones/src:/home/jones/public_html/classes/compute.jar
cd /home/jones/src
javac client/ComputePi.java
javac -d /home/jones/public_html/classes client/Pi.java
```

- ⇨ Only the Pi class needs to be placed in the directory public_html/classes/client (the client directory is created by javac if it does not exist). That is because only the Pi class needs to be available for downloading to the compute engine's virtual machine.

RMI Example 2 (Continued)

- Start the RMI Registry:

```
unsetenv CLASSPATH
rmiregistry &
```

- Why do we make sure that rmiregistry has no CLASSPATH set when we start it??
  - ⇨ If the rmiregistry can find the stub classes in its CLASSPATH, it will not remember that the stub class can be loaded from the server's code base, as specified by the java.rmi.server.codebase property. Therefore, the rmiregistry will not tell the client the proper codebase when the client downloads the stub object from the rmiregistry. Consequently, the client will not be able to download the stub class.

RMI Example 2 (Continued)

- Start the server:

```
java -Djava.rmi.server.codebase=http://zaphod/~ann/classes/
     -Djava.rmi.server.hostname=zaphod.east.sun.com
     -Djava.security.policy=policyfile
     engine.ComputeEngine
```

  - ⇨ When you start the compute engine, you need to specify, using the java.rmi.server.codebase property, where the server's classes will be made available. In this example, the server-side classes to be made available for downloading are the ComputeEngine's stub and the Compute and Task interfaces, available in ann's public_html/classes directory.

RMI Example 2 (Continued)

- Start the client:

```
setenv CLASSAPTH
  /home/jones/src:/home/jones/public_html/classes/compute.jar
java -Djava.rmi.server.codebase=http://ford/~jones/classes/
     -Djava.security.policy=policyfile
     client.ComputePi zaphod.east.sun.com 20
```