Chapter 13: Remote Method Invocation (RMI)

1) Introduction 13-2
2) RMI Architecture 13-3
3) The Remote Interface 13-4
4) The Remote Object 13-5
5) Writing the Server 13-6
6) The RMI Compiler 13-8
7) Writing the Client 13-9
8) Remote Method Arguments and Return Values 13-10
9) Dynamic Loading of Stub Classes 13-11
10) Remote RMI Client Example 13-12
11) Running the Remote RMI Client Example 13-20

Evaluation Copy /training/etc CARE TO LEARN Evaluation Copy

Introduction

Evaluation Copy

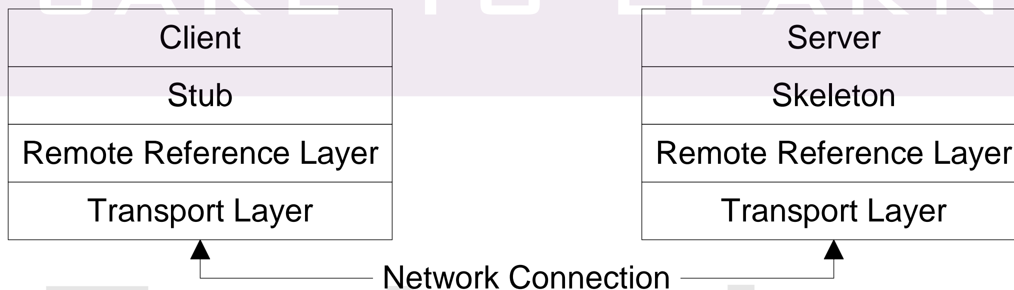
- The **R**emote **M**ethod **I**nvocation (**RMI**) model represents a **distributed object application.**
  - RMI allows an object inside a JVM (a client) to invoke a method on an object running on a remote JVM (a server) and have the results returned to the client.
    - Therefore, RMI implies a client and a server.
- The server application typically creates an object and makes it accessible remotely.
  - Therefore, the object is referred to as a remote object.
  - The server registers the object that is available to clients.
    - One of the ways this can be accomplished is through a naming facility provided as part of the JDK, which is called the `rmiregistry`.
    - The server uses the registry to bind an arbitrary name to a remote object.
- A client application receives a reference to the object on the server and then invokes methods on it.
  - The client looks up the name in the registry and obtains a reference to an object that is able to interface with the remote object.
    - The reference is referred to as a remote object reference.
  - Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object.

/training/etc CARE TO LEARN Evaluation Copy

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

RMI Architecture

- The interface that the client and server objects use to interact with each other is provided through stubs/skeleton, remote reference, and transport layers.
  - Stubs and skeletons are Java objects that act as proxies to the client and server, respectively.
    - All the network-related code is placed in the stub and skeleton, so that the client and server will not have to deal with the network and sockets in their code.
  - The remote reference layer handles the creation of and management of remote objects.
  - The transport layer is the protocol that sends remote object requests across the network.
- A simple diagram showing the above relationships is shown below.



Evaluation Copy /training/etc CARE TO LEARN Evaluation Copy

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

The Remote Interface

- The server's job is to accept requests from a client, perform some service, and then send the results back to the client.
  - The server must specify an interface that defines the methods available to clients as a service.
    - This **remote interface** defines the client view of the remote object.
- The remote interface is always written to extend the `java.rmi.Remote` interface.
  - `Remote` is a "marker" interface that identifies interfaces whose methods may be invoked from a non-local virtual machine.

CalendarTask.java

```
1.  package examples.rmi;
2.  import java.rmi.Remote;
3.  import java.rmi.RemoteException;
4.  import java.util.Calendar;
5.  public interface CalendarTask extends Remote {
6.      Calendar getDate() throws RemoteException;
7.  }
```

- In the example above, `getDate()` is a remote method of the remote interface `CalendarTask`.
  - All methods defined in the remote interface are required to state that they throw a `RemoteException`.
    - A `RemoteException` represents communication-related exceptions that may occur during the execution of a remote method call.

Evaluation Copy /training/etc CAN TO LEARN Evaluation Copy

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

The Remote Object

- An implementation of the `CalendarTask` interface is shown below.
  - The implementation is referred to as the remote object.
  - The implementation class extends `UnicastRemoteObject` to link into the RMI system.
    - This is not a requirement. A class that does not extend `UnicastRemoteObject` may use its `exportObject()` method to be linked into RMI.
  - When a class extends `UnicastRemoteObject`, it must provide a constructor declaring that it may throw a `RemoteException` object.
    - When this constructor calls super(), it activates code in `UnicastRemoteObject`, which performs the RMI linking and remote object initialization.

CalendarImpl.java

```
 1.  package examples.rmi;
 2.  import java.rmi.RemoteException;
 3.  import java.rmi.server.UnicastRemoteObject;
 4.  import java.util.Calendar;
 5.
 6.  public class CalendarImpl extends UnicastRemoteObject
 7.          implements CalendarTask {
 8.
 9.      private int counter = 1;
10.
11.      public CalendarImpl() throws RemoteException {}
12.
13.      public Calendar getDate() throws RemoteException{
14.          System.out.print("Method called on server:");
15.          System.out.println("counter = " + counter++);
16.          return Calendar.getInstance();
17.      }
18.  }
```

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

Writing the Server

Evaluation Copy

- The server creates the remote object, registers it under some arbitrary name, then waits for remote requests.
  - The `java.rmi.registry.LocateRegistry` class allows the RMI registry service (provided as part of the JVM) to be started within the code by calling its `createRegistry` method.
    - This could have also been achieved by typing the following at a command prompt: `rmiregistry`.
    - The default port for RMI is `1099`.
  - The `java.rmi.registry.Registry` class provides two methods for binding objects to the registry.
    - `Naming.bind("`*`ArbitraryName`*`", `*`remoteObj`*`);` throws an Exception if an object is already bound under the "*`ArbitrayName`*."
    - `Naming.rebind ("`*`ArbitraryName`*`", `*`remoteObj`*`);` binds the object under the "*`ArbitraryName`*" if it does not exist or overwrites the object that is bound.
- The example on the following page acts as a server that creates a `CalendarImpl` object and makes it available to clients by binding it under a name of "`TheCalendar`."

/training/etc CARE TO LEARN Evaluation Copy

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

Writing the Server

CalendarServer.java

```
 1.  package examples.rmi;
 2.  import java.rmi.Naming;
 3.  import java.rmi.registry.LocateRegistry;
 4.
 5.  public class CalendarServer {
 6.
 7.      public static void main(String args[]) {
 8.          System.out.println("Starting server...");
 9.          // Start RMI registry service and bind
10.          // object to the registry
11.          try {
12.              LocateRegistry.createRegistry(1099);
13.              Naming.rebind("TheCalendar",
14.                      new CalendarImpl());
15.          } catch (Exception e) {
16.              e.printStackTrace();
17.              System.exit(1);
18.          }
19.          System.out.println("Server ready");
20.      }
21.  }
```

- If both the client and the server are running Java SE 5 or higher, no additional work is needed on the server side.
  - Simply compile the `CalendarTask`, `CalendarImpl`, and `CalendarServer`, and the server can then be started.
  - The reason for this is the introduction in Java SE 5 of dynamic generation of stub classes.
    - Java SE 5 adds support for the dynamic generation of stub classes at runtime, eliminating the need to use the RMI stub compiler, `rmic`, to pre-generate stub classes for remote objects.
    - Note that `rmic` must still be used to pre-generate stub classes for remote objects that need to support clients running on earlier versions.

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

The RMI Compiler

- If RMI is being used with a version of Java prior to Java SE 5, a stub must be generated on the server-side and made available to the client.
  - The RMI compiler (`rmic`) is a tool used to create any necessary stubs and/or skeletons to support the remote object.
    - Skelton(s) have been optional since Java SE 1.2.
  - The `rmic` compiler is passed to the compiled version of the remote object and generates a stub class from it as shown below.

```
rmic -keep -d %CLASSES% examples.rmi.CalendarImpl
```

    - The "`-keep`" is optional and is being used so that, in addition to the generated `.class` files, the `.java` files will be retained so that they can be viewed if desired.
    - The result of running the `rmic` command above is a file named `CalendarImpl_Stub.class`.
    - Since the "`-keep`" option was used, there is also a file named `CalendarImpl_Stub.java`.
  - The `CalendarImpl_Stub` class generated is a client-side component and as such, must exist on the client's classpath in order for client code to successfully communicate with the server.
    - If the stub class is not available locally to the client, it must be loaded dynamically over the network.
    - Keep in mind that this is only necessary if a version of Java prior to Java SE 5 is being used.

Evaluation Copy /training/etc CARE TO LEARN Evaluation Copy

Writing the Client

- An RMI client is a program that accesses the services provided by a remote object.
  - The `java.rmi.registry.LocateRegistry` class allows the RMI registry service to be located by a client by its `getRegistry` method.
    - The `java.rmi.registry.Registry` class provides a `lookup` method that takes the "`ArbitraryName`" the remote object was bound to by the server.
- Once the client obtains a reference to a remote object, it invokes methods as if the object were local.

CalendarClient.java

```
 1. package examples.rmi;
 2.
 3. import java.rmi.registry.*;
 4. import java.util.Calendar;
 5.
 6. public class CalendarClient {
 7.
 8.     public static void main(String args[]) {
 9.         Calendar c = null;
10.         CalendarTask remoteObj;
11.         String host = "localhost";
12.         if(args.length == 1)
13.             host = args[0];
14.         try {
15.             Registry r =
16.               LocateRegistry.getRegistry(host, 1099);
17.             Object o = r.lookup("TheCalendar");
18.             remoteObj = (CalendarTask) o;
19.             c = remoteObj.getDate();
20.         } catch (Exception e) {
21.             e.printStackTrace();
22.         }
23.         System.out.printf("%tc", c);
24.     }
25. }
```

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

Remote Method Arguments and Return Values

- The arguments to a remote method must be `Serializable`.
  - They must be primitive types or objects that implement the `Serializable` interface.
  - The same restriction applies to return values.
- The RMI stub/skeleton layer decides how to send arguments and return values over the network.
  - If the object is `Serializable` but not `Remote`:
    - the object is serialized and streamed in byte format; and
    - the receiver de-serializes the bytes into a copy of the original object.
  - If the object is a `Remote` object:
    - a remote reference for the object is marshaled and sent to the remote process; and
    - this reference is received and converted into a stub for the original object.
  - If the argument or return value is not serializable, a `java.rmi.MarshalException` is thrown.
- The key difference between remote and non-remote objects is that `Remote` objects are sent by reference, while non-remote objects (and primitive types) are sent by copy.

Evaluation Copy /training/etc CARE TO LEARN Evaluation Copy

Dynamic Loading of Stub Classes

- If the client stub class is not available in the local `CLASSPATH`, it must be loaded dynamically over the network.
  - This is a typical scenario when the client and server are not running on the same machine.
  - We will illustrate downloading of stub classes via a web server.
- When the RMI run-time system marshals a remote object stub, it encodes a URL in the byte stream to tell the process on the other end of the stream where to look for the class file for the marshaled object.
  - This URL is obtained from a system property called `java.rmi.server.codebase`.
    - We will set this property on the command line to point to a directory within the web server's document base.
  - Note that in order for a Java runtime system to be able to load classes remotely, it has to have a security manager installed that will allow the remote load.
    - There is one provided by the `java.rmi.RMISecurityManager` class.
  - The final issue is that the default Java security policy does not allow all the networking operations required to load a class from a remote host.
    - An RMI client that needs to load classes remotely must have a policy file granting the necessary permissions.
    - The name of the policy file can be specified on the command line by setting the `java.security.policy` property.

Evaluation Copy /training/etc CARE TO LEARN Evaluation Copy

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

Remote RMI Client Example

- The RMI application shown below illustrates dynamic loading of stub classes.
  - It also shows an example of a `Remote` object used as a method argument.
  - Following the source code are detailed instructions on how to run the Client and Server.

- We begin with the remote interface.

Account.java

```
 1.  package examples.rmi;
 2.
 3.  import java.rmi.*;
 4.
 5.  public interface Account extends Remote {
 6.      public String getName() throws RemoteException;
 7.
 8.      public double getBalance()
 9.        throws RemoteException;
10.
11.      public void withdraw(double amt)
12.        throws RemoteException;
13.
14.      public void deposit(double amt)
15.        throws RemoteException;
16.
17.      public void transfer(double amt, Account src)
18.        throws RemoteException;
19.  }
```

- The implementation of the remote interface is shown on the next page.

Evaluation Copy /training/etc CARE TO LEARN Evaluation Copy

Remote RMI Client Example

AccountImpl.java

```
 1. package examples.rmi;
 2.
 3. import java.rmi.server.*;
 4. import java.rmi.*;
 5.
 6. public class AccountImpl extends UnicastRemoteObject
 7.   implements Account {
 8.     private double balance = 0.0;
 9.     private String name = "";
10.
11.     public AccountImpl(String aName)
12.       throws RemoteException {
13.         name = aName;
14.     }
15.
16.     public String getName() throws RemoteException {
17.         return name;
18.     }
19.
20.     public double getBalance()
21.       throws RemoteException {
22.         return balance;
23.     }
24.
25.     public void withdraw(double amt)
26.       throws RemoteException {
27.         if (amt > balance)
28.             throw new RemoteException();
29.         balance -= amt;
30.     }
31.
32.     public void deposit(double amt)
33.       throws RemoteException {
34.         balance += amt;
35.     }
36.
37.     public void transfer(double amt, Account src)
38.       throws RemoteException {
39.         src.withdraw(amt);
40.         this.deposit(amt);
41.     }
42. }
```

Evaluation Copy /training/etc CARE TO LEARN Evaluation Copy

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

Remote RMI Client Example

- The Server is shown below with the following features.
  - The compiling of the stub class for the client is done using `Runtime.exec()`.
    - The `exec` method allows the JVM to run an external process (in this case the rmi compiler - `rmic`).
  - The server creates and starts the registry service.

AccountServer.java

```
 1.  package examples.rmi;
 2.
 3.  import java.io.IOException;
 4.  import java.net.InetAddress;
 5.  import java.rmi.registry.*;
 6.
 7.  public class AccountServer {
 8.      private static String buildCommandLine(){
 9.          String jcp = "java.class.path";
10.          StringBuffer sb = new StringBuffer();
11.          sb.append('"');
12.          sb.append(System.getProperty(jcp));
13.          sb.append('"');
14.          String classpath = sb.toString();
15.          sb.setLength(0);
16.          sb.append("rmic -d ").append(classpath);
17.          sb.append(" -classpath ").append(classpath);
18.          sb.append(" examples.rmi.AccountImpl");
19.          System.out.println(sb.toString());
20.          return sb.toString();
21.      }
22.      public static void main(String args[]) {
23.          // execute rmic as an external process
24.          Process p = null;
25.          try {
26.              String command = buildCommandLine();
27.              p = Runtime.getRuntime().exec(command);
28.              p.waitFor(); // wait for completion
29.          } catch (Exception e1) {
30.              e1.printStackTrace();
31.          }
```

Remote RMI Client Example

AccountServer.java - continued

```
32.
33.         try {
34.             String key = "java.rmi.server.codebase";
35.             InetAddress server =
36.                 InetAddress.getLocalHost();
37.             String address = server.getHostAddress();
38.             String value =
39.                 "http://" + address + ":8080/";
40.             System.setProperty(key, value);
41.             //Start the StubServer
42.             Thread t = new StubServer();
43.             t.start();
44.             // Create registry service
45.             Registry reg =
46.                 LocateRegistry.createRegistry(1099);
47.             // Create some Accounts
48.             AccountImpl acct1 =
49.               new AccountImpl("Alan");
50.             AccountImpl acct2 =
51.               new AccountImpl("Dave");
52.
53.             // Register with the naming registry.
54.             reg.rebind("Alan", acct1);
55.             reg.rebind("Dave", acct2);
56.
57.             System.out.println("Accts registered");
58.         } catch (Exception e) {
59.             e.printStackTrace();
60.         }
61.     }
62. }
```

Evaluation Copy /training/etc CAREER LEARN Evaluation Copy

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

Remote RMI Client Example

- Although a web server such as Tomcat, WebLogic, or WebSphere could be used to host the stub class necessary for dynamic loading of the stub class, the file below is a simple web server based on the code from the Networking chapter of this course.

StubServer.java

```
 1.  package examples.rmi;
 2.
 3.  import java.io.*;
 4.  import java.net.*;
 5.
 6.  public class StubServer extends Thread {
 7.
 8.      static byte[] hdrNotFound =
 9.          "HTTP/1.0 404 Not Found\n\n".getBytes();
10.      static byte[] notFound =
11.          "<html>Resource Not Found</html>".getBytes();
12.      static byte[] hdrOK =
13.          "HTTP/1.0 200 OK\n\n".getBytes();
14.      static byte[] testResponse =
15.          "<html>Server operational</html>".getBytes();
16.
17.      public void run() {
18.          ServerSocket theServer = null;
19.          Socket clientSocket;
20.          // Attempt to start the server
21.          try {
22.              theServer = new ServerSocket(8080);
23.              while (true) {
24.                  clientSocket = theServer.accept();
25.                  handleClient(clientSocket);
26.              }
27.          } catch (IOException ioe) {
28.              ioe.printStackTrace();
29.              System.exit(1);
30.          }
31.      }
32.
```

▸ *Continued on following page*

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

Remote RMI Client Example

StubServer.java - *continued*

```
33.     private void handleClient(Socket cSocket) {
34.         OutputStream toClient = null;
35.         BufferedReader fromClient = null;
36.         try {
37.             // Get Input and Output
38.             fromClient = new BufferedReader(
39.                    new InputStreamReader(cSocket
40.                     .getInputStream()));
41.             toClient = cSocket.getOutputStream();
42.             // read from Client
43.             String theLine = fromClient.readLine();
44.             String request = theLine.split(" ")[1];
45.             System.out.println("StubServer Request:"
46.                                  + theLine);
47.             if (request.equals("/")) {
48.                 toClient.write(hdrOK);
49.                 toClient.write(testResponse);
50.             } else {
51.                 processStub(request, toClient);
52.             }
53.
54.             fromClient.close();
55.             toClient.close();
56.             cSocket.close();
57.         } catch (IOException ioe) {
58.             String msg = "Connection lost";
59.             System.out.println(msg);
60.         }
61.     }
```

▸ *Continued on following page*

Evaluation Copy /training/etc CARE TO LEARN Evaluation Copy

Remote RMI Client Example

StubServer.java - *continued*

```
62.     private void processStub(String req,
63.             OutputStream toClient){
64.         InputStream is =
65.             this.getClass().getResourceAsStream(req);
66.
67.         byte[] bufferedStub = null;
68.         try {
69.             if (is != null) {
70.                 int size = is.available();
71.                 bufferedStub = new byte[size];
72.                 is.read(bufferedStub);
73.                 is.close();
74.                 toClient.write(hdrOK);
75.                 toClient.write(bufferedStub);
76.             } else {
77.                 toClient.write(hdrNotFound);
78.                 toClient.write(notFound);
79.             }
80.         } catch (IOException e) {
81.             e.printStackTrace();
82.         }
83.     }
84. }
```

- Finally, the client code is shown below.

AccountClient.java

```
 1. package examples.rmi;
 2.
 3. import java.net.URL;
 4. import java.rmi.*;
 5. import java.rmi.registry.*;
 6.
 7. public class AccountClient {
 8.     public static void main(String args[]) {
 9.         String host = "localhost";
10.         if (args.length > 0) { host = args[0]; }
11.         try {
12.             String key = "java.security.policy";
13.             Class c = AccountClient.class;
14.             URL u =c.getResource("policy.client");
```

Evaluation Copy /training/etc CARE TO LEARN

Remote RMI Client Example

AccountClient.java - *continued*

```
15.             String value = u.getPath();
16.             System.setProperty(key, value);
17.             System.setSecurityManager(
18.                     new RMISecurityManager());
19.             Registry r =
20.               LocateRegistry.getRegistry(host, 1099);
21.             // Lookup Account objects
22.             Account act1 =
23.                 (Account) r.lookup("Alan");
24.             Account act2 =
25.                 (Account) r.lookup("Dave");
26.
27.             showBalance(act1);
28.             showBalance(act2);
29.
30.             // Make some deposits
31.             act1.deposit(200);
32.             act2.deposit(100);
33.
34.             // Show results
35.             System.out.println("Deposit 200 & 100");
36.             showBalance(act1);
37.             showBalance(act2);
38.
39.             // Do a transfer
40.             act2.transfer(10, act1);
41.
42.             // Show results
43.             System.out.println("Transfer 10");
44.             showBalance(act1);
45.             showBalance(act2);
46.         } catch (Exception e){ e.printStackTrace(); }
47.     }
48.
49.     public static void showBalance(Account acct)
50.       throws RemoteException {
51.         System.out.println("Balance for " +
52.                 acct.getName() + " is " +
53.                 acct.getBalance());
54.     }
55. }
```

Evaluation Copy /training/etc CAREER LEARN Evaluation Copy

Running the Remote RMI Client Example

Evaluation Copy

- Running the `AccountServer` will accomplish the following.
  - It generates the `AccountImpl_Stub` needed by the client so that it can be dynamically loaded by the client.
  - It starts the `StubServer` so that the `AccountImpl_Stub` is available via the `java.rmi.server.codebase` property.
  - It starts the rmi registry to bind the remote `Account` objects.

/training/etc CARE TO LEARN

- Running the `AccountClient` will accomplish the following.
  - It installs the `RMISecurityManager`.
  - It utilizes the policy file named `policy.client` to allow the JVM to load a class from a remote URL.
  - If the Stub class is not available on the client's classpath when the lookup is performed, the client will automatically retrieve the necessary stub from the servers codebase.

Evaluation Copy

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

Exercises

1. Build and test an implementation for a `MathServices` interface with remote methods as shown below.

```
public double sqroot (double value);
public double square(double value);
```

- Begin with `MathServices.java` in the `starters` directory.

Evaluation Copy
/training/etc
CARE TO LEARN
Evaluation Copy

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.

This Page Intentionally Left Blank

Evaluation Copy

/training/etc

CARE TO LEARN

Evaluation Copy

©2009 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.