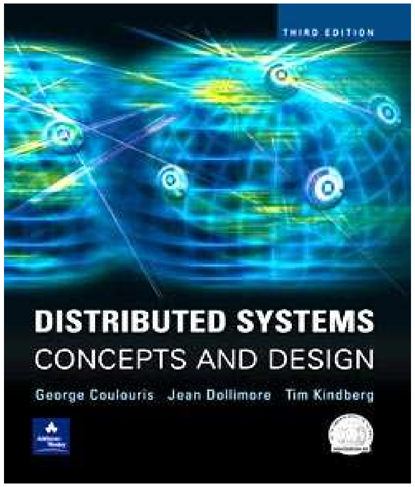
Case Study: Java RMI

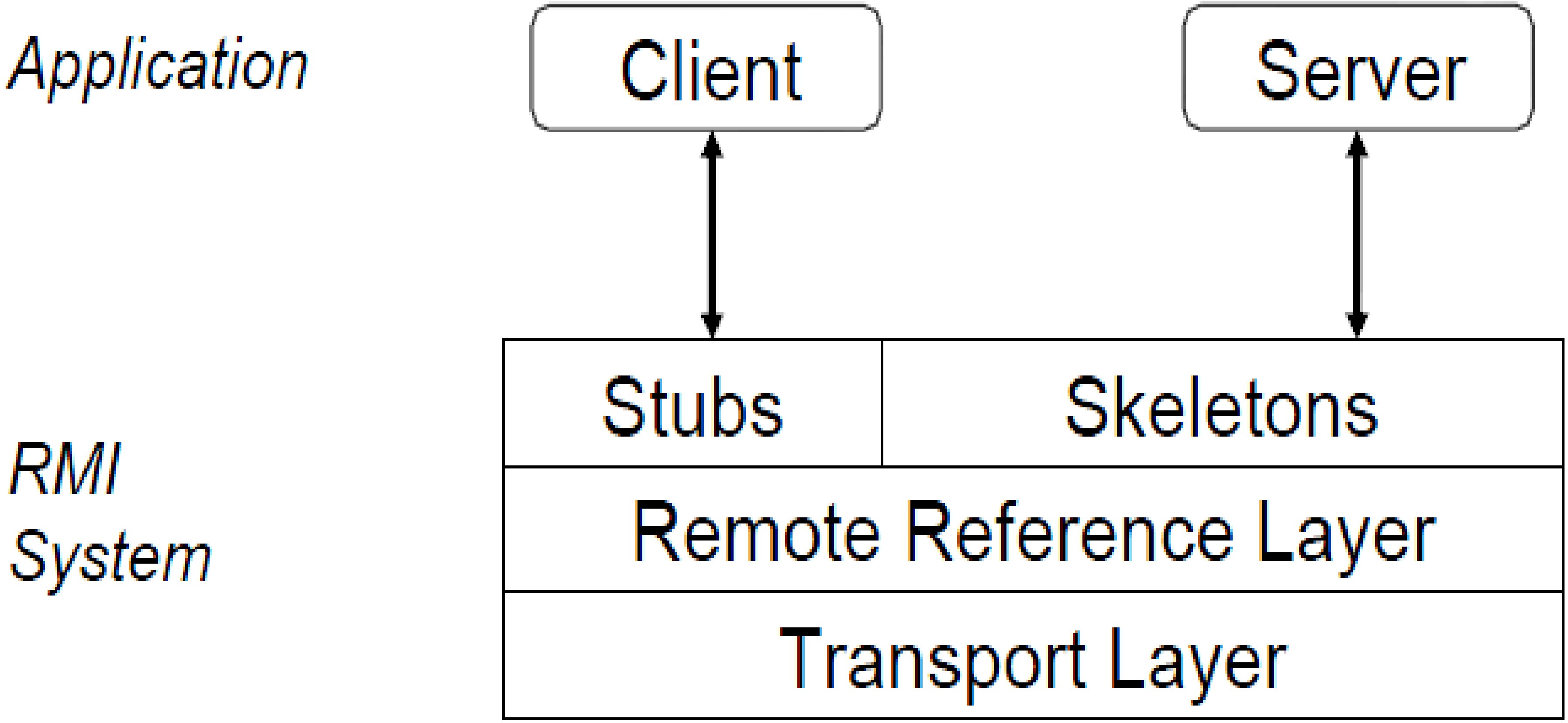


From **Coulouris, Dollimore and Kindberg**

Distributed Systems: Concepts and Design

Edition 3, © Addison-Wesley 2001

RMI Architecture



Application

Client

Server

RMI System

Stubs

Skeletons

Remote Reference Layer

Transport Layer

Java RMI

- Java RMI extends the Java object model to provide support for distributed objects in the Java language.
  - It allows objects to invoke methods on remote objects using the same syntax as for local invocations.
  - Type checking applies equally to remote invocations as to local ones.
  - The remote invocation is known because **RemoteExceptions** has been handled and the remote object is implemented using the **Remote** interface.
  - The semantics of parameter passing differ because invoker and target are remote from on another.

Java RMI

- Programming distributed applications in Java RMI is simple.
  - It is a single-language system.
  - The programmer of a remote object must consider its behavior in a concurrent environment.
- The files needed for creating a Java RMI application are:
  - A **remote interface** defines the remote interface provided by the service. Usually, it is a single line statement specifies the service function (**HelloInterface.java)**. (An interface is the skeleton for a public class.)

Java RMI

- The files needed for creating a Java RMI application are (continued):
    - A **remote object** implements the remote service. It contains a constructor and required functions. (**Hello.java)**
    - A **client** that invokes the remote method. **(HelloClient.java)**
    - The **server** offers the remote service, installs a security manager and contacts rmiregistry with an instance of the service under the name of the remote object. (**HelloServer.java**)

HelloInterface.java

```
import java.rmi.*;

public interface HelloInterface extends Remote {
  public String say(String msg) throws
    RemoteException;
}
```

Hello.java

```
import java.rmi.*;
import java.rmi.server.*;
public class Hello extends
                  UnicastRemoteObject implements HelloInterface {
  private String message;

  public Hello(String msg) throws RemoteException {
    message = msg;
  }
```

Hello.java (continued)

```
public String say(String m) throws RemoteException {
  // return input message - reversing input and suffixing
  // our standard message
  return new StringBuffer(m).reverse().toString() + "\n" +
  message;
 }
}
```

HelloClient.java

```
import java.rmi.*;

public class HelloClient  {
  public static void main(String args[]) {
  String path = "//localhost/Hello";
  try {
    if (args.length < 1) {
      System.out.println("usage: java HelloClient
   <host:port> <string> ... \n");
    } else path = "//" + args[0] + "/Hello";
```

HelloClient.java

```
    HelloInterface hello =
      (HelloInterface) Naming.lookup(path);
    for (int i = 0; i < args.length; ++i)
      System.out.println(hello.say(args[i]));
    } catch(Exception e) {
      System.out.println("HelloClient exception: " + e);
    }
  }
}
```

HelloServer.java

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

public class HelloServer {
  public static void main(String args[]) {
  // Create and install a security manager
  if (System.getSecurityManager() == null)
    System.setSecurityManager(new RMISecurityManager());
    try {
      Naming.rebind("Hello", new Hello("Hello, world!"));
      System.out.println("server is running...");
    }
```

HelloServer.java

```
     catch (Exception e) {
      System.out.println("Hello server failed:" + e.getMessage());
    }
  }
}
```

Java RMI

- **Compile the code**

  javac Hello.java HelloClient.java HelloInterface.java HelloServer.java

- **Generate stubs for the remote service** (make sure that your classpath contains your current directory)

  rmic Hello

- **Start the registry** (in a separate window or in the background)

  rmiregistry

  (be sure to kill this process when you're done)

Java RMI

- **Start the server** in one window or in the background with the security policy

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

  or without the security policy

  ```
  java HelloServer
  ```

- **Run the client** in another window

  ```
  java HelloClient testing
  ```

Java RMI Remote Object References

- An object must have the remote object reference of other object in order to do remote invocation of that object.
- Parameter and result passing
    - Remote object references may be passed as input arguments or returned as output arguments.
    - Parameters of a method in Java are input parameters.
    - Returned result of a method in Java is the single output parameter.
    - Objects are serialized to be passed as parameters.
    - When a remote object reference is returned, it can be used to invoke remote methods.
    - Local serializable objects are copied by value.

Java RMI Remote Object References

- Downloading of classes
  - Java is designed to allow classes to be downloaded from one virtual machine to another.
  - If the recipient of a remote object reference does not posses the proxy class, its code is downloaded automatically.
- RMIregistry
  - The RMIregistry is designed to allow is the binder for Java RMI.
  - It maintains a table mapping textual, URL-style names to references to remote objects.

Java RMI Remote Object References

- Server Program
  - The server consists of a main method and a servant class to implement each of its remote interface.
  - The main method of a server needs to create a security manager to enable Java security to apply the protection for an RMI server.
- Client Program
  - Any client program needs to get started by using a binder to look up a remote reference.
  - A client can set a security manager and then looks up a remote object reference.

Java RMI Callbacks

- **Callback** refers to server's action in notifying the client.
- **Callback Facility -** Instead of client polling the server, the server calls a method in the client when it is updated.
- Details
  - Client creates a remote object that implements an interface for the server to call.
  - The server provides an operation for clients to **register** their callbacks.
  - When an event occurs, the server calls the interested clients.

Java RMI Callback Issues

- Advantages of callback
  - more efficient than polling
  - more timely than polling
  - provides a way for the server to inquire about client status
- Disadvantages of callback
  - may leave server in inconsistent state if client crashes or exits without notifying server
  - requires server to make series of synchronous RMI's

Shared Whiteboard Example

- In the RMI and CORBA case studies, we use a shared whiteboard as an example
    - This is a distributed program that allows a group of users to share a common view of a drawing surface containing graphical objects, each of which has been drawn by one of the users.
- The server maintains the current state of a drawing and it provides operations for clients to:
    - Add a shape, retrieve a shape or retrieve all the shapes,
    - Retrieve its version number or the version number of a shape

Figure 5.11
Java Remote interfaces *Shape* and *ShapeList*

- Note the interfaces and arguments
- *GraphicalObject* is a class that implements *Serializable*.

Figure 5.11

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject  getAllState() throws RemoteException;
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

Figure 5.12
The *Naming* class of Java RMIregistry

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.13, line 3.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 15.15 line 1. A remote object reference is returned.

String [] list()

This method returns an array of Strings containing the names bound in the registry.

Figure 5.13
Java class *ShapeListServer* with *main* method

```
import java.rmi.*;
public class ShapeListServer{
      public static void main(String args[]){
            System.setSecurityManager(new RMISecurityManager());
             try{
                  ShapeList aShapeList = new ShapeListServant();                1
                        Naming.rebind("Shape List", aShapeList );               2
                  System.out.println("ShapeList server ready");
                  }catch(Exception e) {
                  System.out.println("ShapeList server main " + e.getMessage());}
      }
}
```

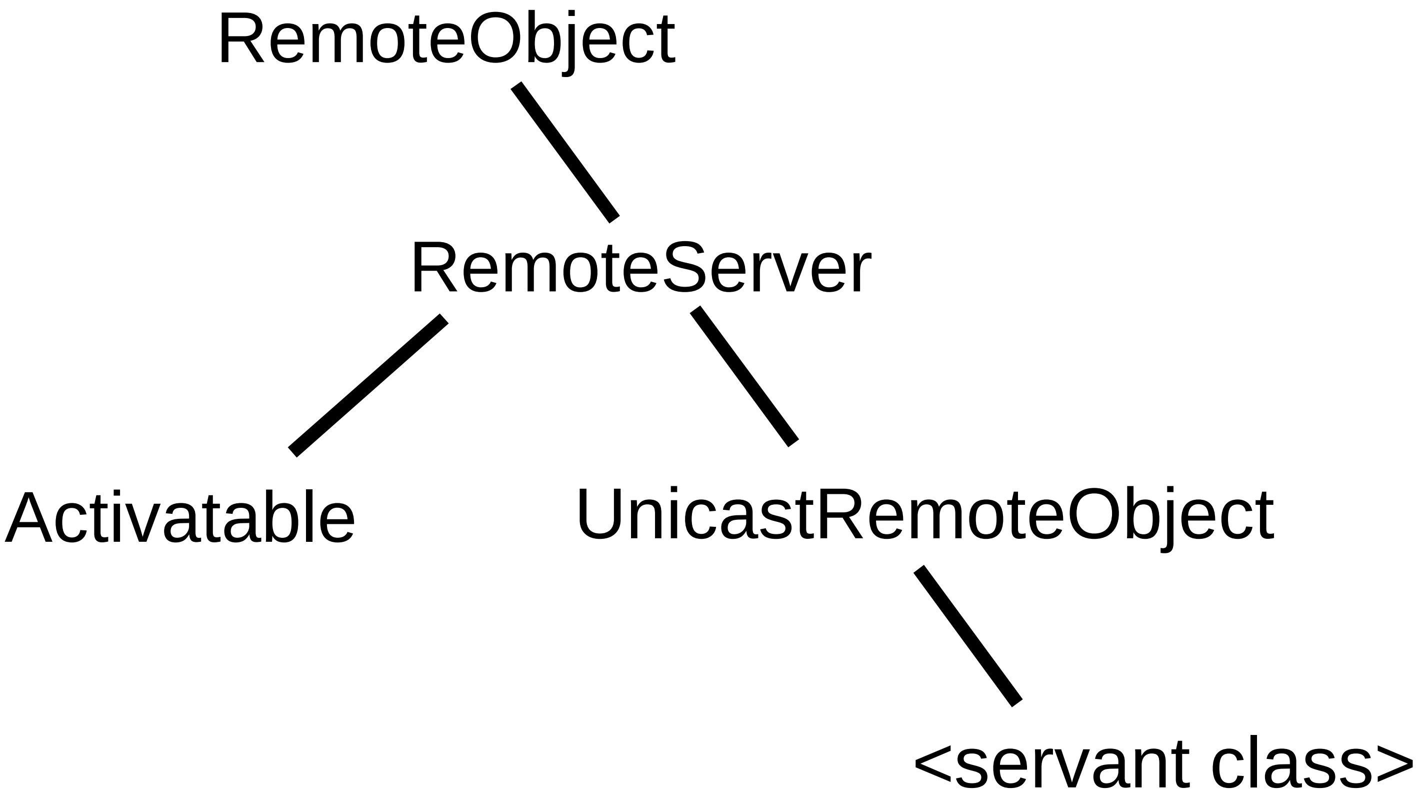
Figure 5.14
Java class *ShapeListServant* implements interface *ShapeList*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
       private Vector theList;                // contains the list of Shapes          1
       private int version;
      public ShapeListServant()throws RemoteException{...}
      public Shape newShape(GraphicalObject g) throws RemoteException {         2
            version++;
                  Shape s = new ShapeServant( g, version);                       3
                  theList.addElement(s);
                  return s;
      }
      public  Vector allShapes()throws RemoteException{...}
      public int getVersion() throws RemoteException { ... }
}
```

Figure 5.15
Java client of *ShapeList*

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
   public static void main(String args[]){
     System.setSecurityManager(new RMISecurityManager());
     ShapeList aShapeList = null;
     try{
           aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList") ;         1
           Vector sList = aShapeList.allShapes();                                  2
     } catch(RemoteException e) {System.out.println(e.getMessage());
     }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
   }
}
```

Figure 5.16
Classes supporting Java RMI



RemoteObject

RemoteServer

Activatable

UnicastRemoteObject

<servant class>

RMI Summary

- Each object has a (global) remote object reference and a remote interface that specifies which of its operations can be invoked remotely.
- Local method invocations provide exactly-once semantics; the best RMI can guarantee is at-most-once.
- Middleware components (proxies, skeletons and dispatchers) hide details of marshalling, message passing and object location from programmers.

Thank You