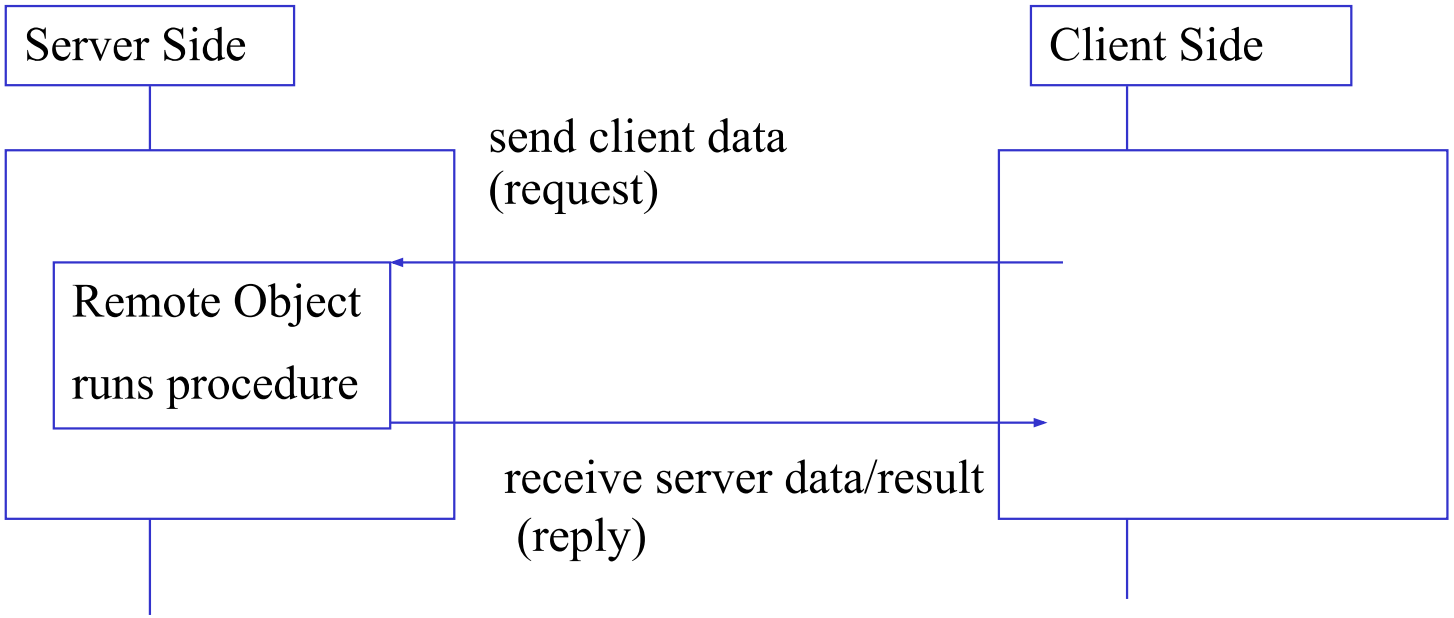
# 5. Remote Method Invocation (RMI)

## 5.1. In Search of the Simplest Communication Form

Main Idea: Working with an object on a remote machine is *made to look like calling a procedure on the remote site*, i.e.

- the application/client sends a message to the remote object
- the remote object receives message, does processing and sends back message with results - the server side;
- the client receives message and uses/prints result



Why do we need yet another mechanism for remote object communication?

We have sockets, and, if we find sockets tedious, there are intermediate representations, such as name/value pairs or XML.

They would do the job, yes.

In the traditional client/server model the request is formulated or translated into an intermediary language, e.g. name/value pairs, XML data. The server parses the requests, translates it into its language, computes reply, translates into intermediary languages, sends it to client, then client repeats the process.

BUT in case of *user defined structured data* these mechanisms

- require a significant coding effort;
- are not very intuitive and not part of the basic vocabulary/concepts of programming languages

As opposed to this RMI

- has a form that is familiar even to an entry level programmer;
- naturally extends the "everything is an object" principle to "objects can be everywhere";

# Goals of RMI

More specifically RMI **specification goals** are listed as follows (http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmi-title.doc.html)

- Support *seamless remote invocations* on objects in different JVMs.
- Support *callbacks* from servers to clients.
- *Integrate* the distributed object model into the Java programming language in a natural way while retaining most of the language's object semantics.
- Make differences between the distributed object model and the java object model apparent.
- Make writing distributed applications as *simple* as possible (certainly simpler than with sockets).
- Preserve the *safety* provided by the Java run-time environment.

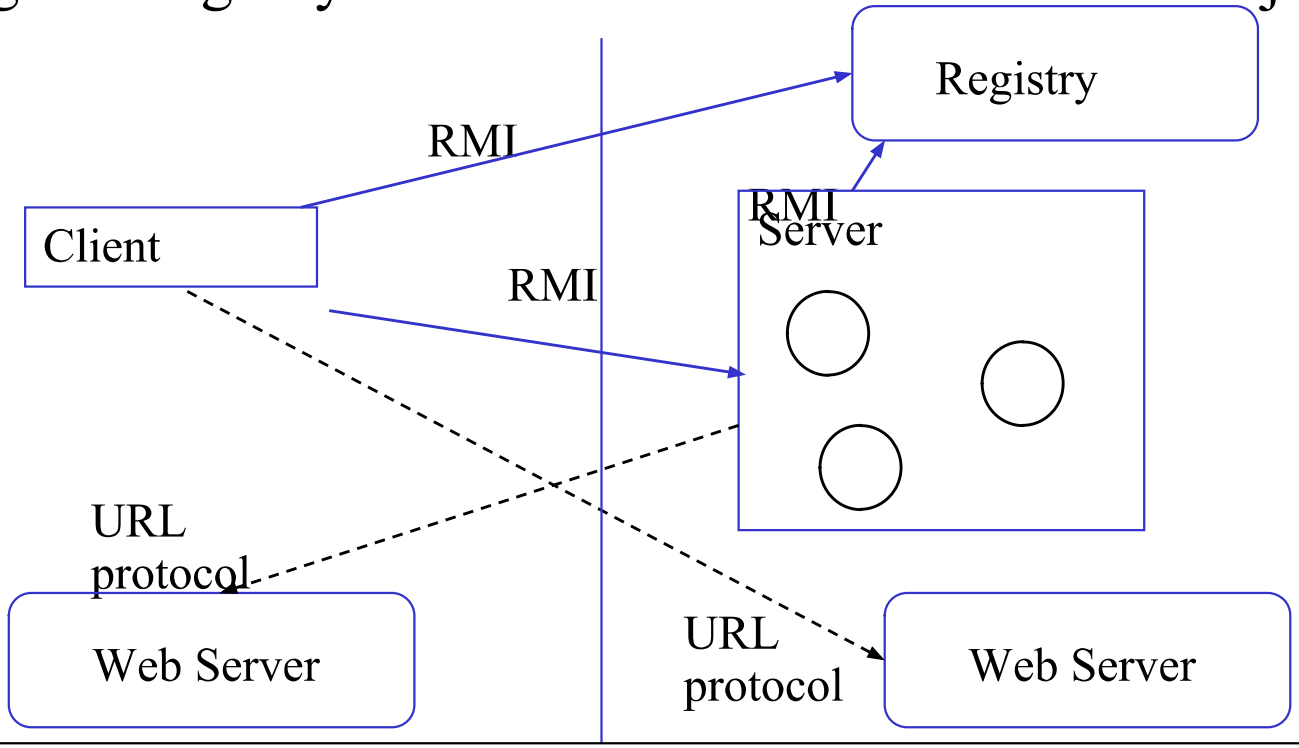
Flexibility and extensibility are provided by:

- *Distributed garbage collection*.
- Capability to *support multiple transports*
- Varying remote invocation mechanisms, such as *unicast* and *multicast*

# Distributed Application Tasks

- *Locate* Remote Objects: Application can obtain references to remote objects when
  a) objects are registered with the RMI naming facility, the **rmiregistry,** or
  b) application can pass and return remote object references as part of its normal operation
- *Communicate* with remote objects: handled by RMI; looks like standard Java method invocation to the programmer
- *Load class bytecodes* for objects that are passed as parameters or return values: Because RMI allows a caller to pass pure Java objects to remote object, RMI provides the necessary mechanisms for loading an object's code as well as transmitting its data

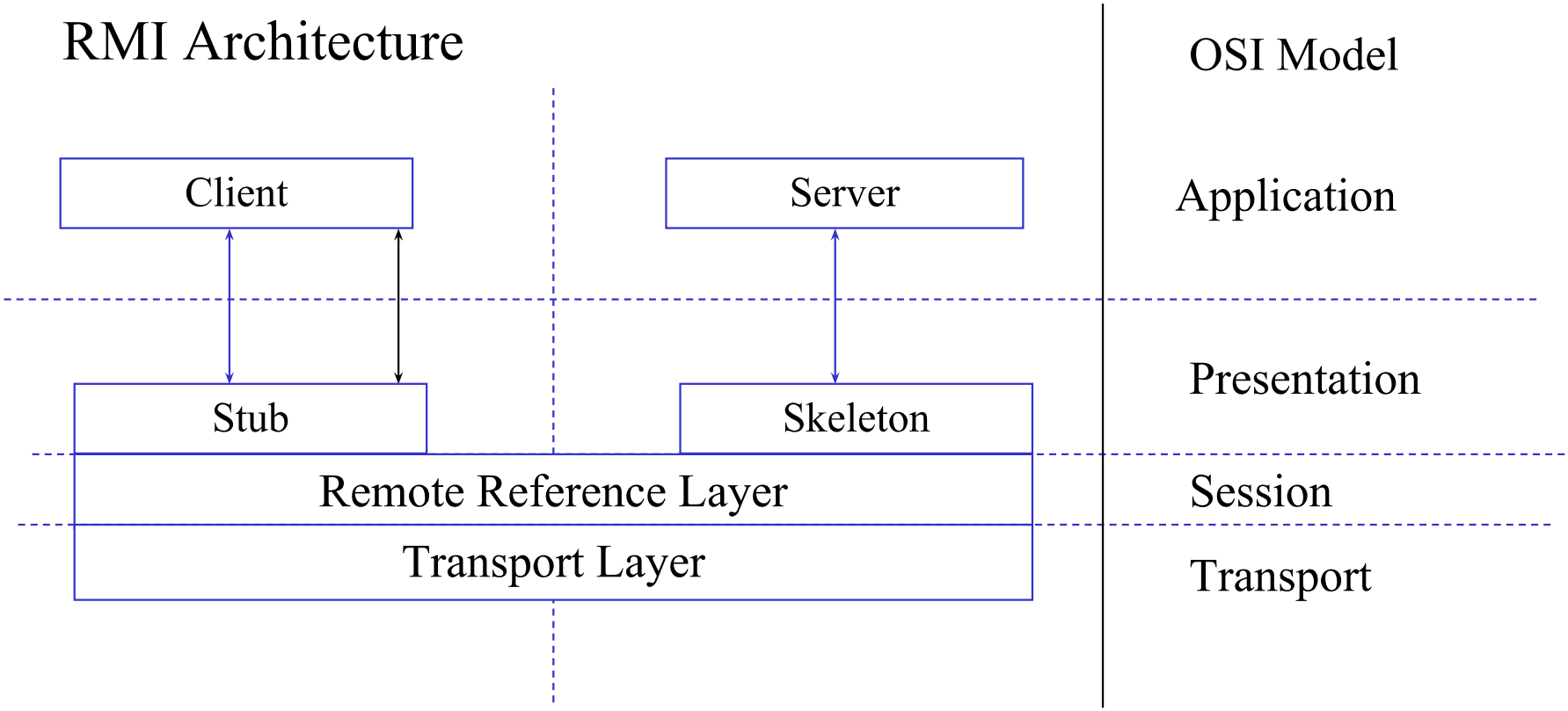
# Using the Registry to obtain References to Remote Object




- Server calls registry to associate name with a remote object.
- Client looks up remote object by name in server's registry.
- Client invokes method on object

Note: The RMI system uses an existing web server to load Java class bytecodes, from server to client and from client to server, when needed. RMI can load class bytecodes using any URL protocol (e.g. HTTP, FTP, file, etc.) that is supported by the Java system.

# RMI Architecture




The layers are independent. Each layer is defined by specific protocol and built using specific interface. Any layer can be replaced by an alternate implementation without affecting the others, e.g. the current transport layer in RMI is TCP based, but can be substituted by a UDP based transport layer

# RMI Architecture (continued)

Stub/Skeleton Layer: Interface of the application with the rest of the system (standard mechanism used in RPC system). This layer transmits the data to the remote reference layer via the abstraction of **marshal streams** *that use object serialization*.

- A **stub** is a surrogate for the remote object, its representative on the client side, that acts as a proxy for the remote object. It resides on the client side (although it is generated on the server side) and handles all the interaction with the remote object
- A **skeleton** handles the communication on the server side (not required in JDK1.2-only environments)

Remote Reference Layer: responsible for providing ability to support varying remote reference or invocation protocols independent of client stubs and server skeletons.

Examples: unicast provides point-to-point invocation, multicast to groups of objects; other protocols deal with replication strategies or persistent references to the remote object, such as enabling remote object activation (supported only by Java2)

# RMI Architecture (continued)

Transport Layer: Low level layer that ships the marshal streams between different address spaces. Responsible for managing connection:

- Setting up connections
- Listening to incoming calls
- Maintaining table of remote objects that reside in same address space

Remote object references are represented by an object identifier and end point. This representation is called a **live reference.** Given a live reference for a remote object the identifier looks up the targeted remote object and the end point sets up the connection to the address space the object resides

# Distributed Garbage Collection

In stand-alone application object that are no longer referenced by any client are automatically deleted.

RMI provides distributed garbage collector that automatically removes objects that are no longer referenced by any client.

RMI uses a reference counting garbage collection that *keeps track of all live references of a given object* on each JVM. When a live reference enters a JVM its count is incremented, when it becomes unreferenced, its count is decremented; when the count is 0 (no live reference) the object can be garbage collected. As long as there is a local reference to a remote object it cannot be garbage collected, since it can be passed to remote server or returned to a client.

# Parameter Marshalling and Unmarshalling

When a client code invokes a remote method on a remote object, it actually calls an ordinary/local Java method encapsulated in the stub.

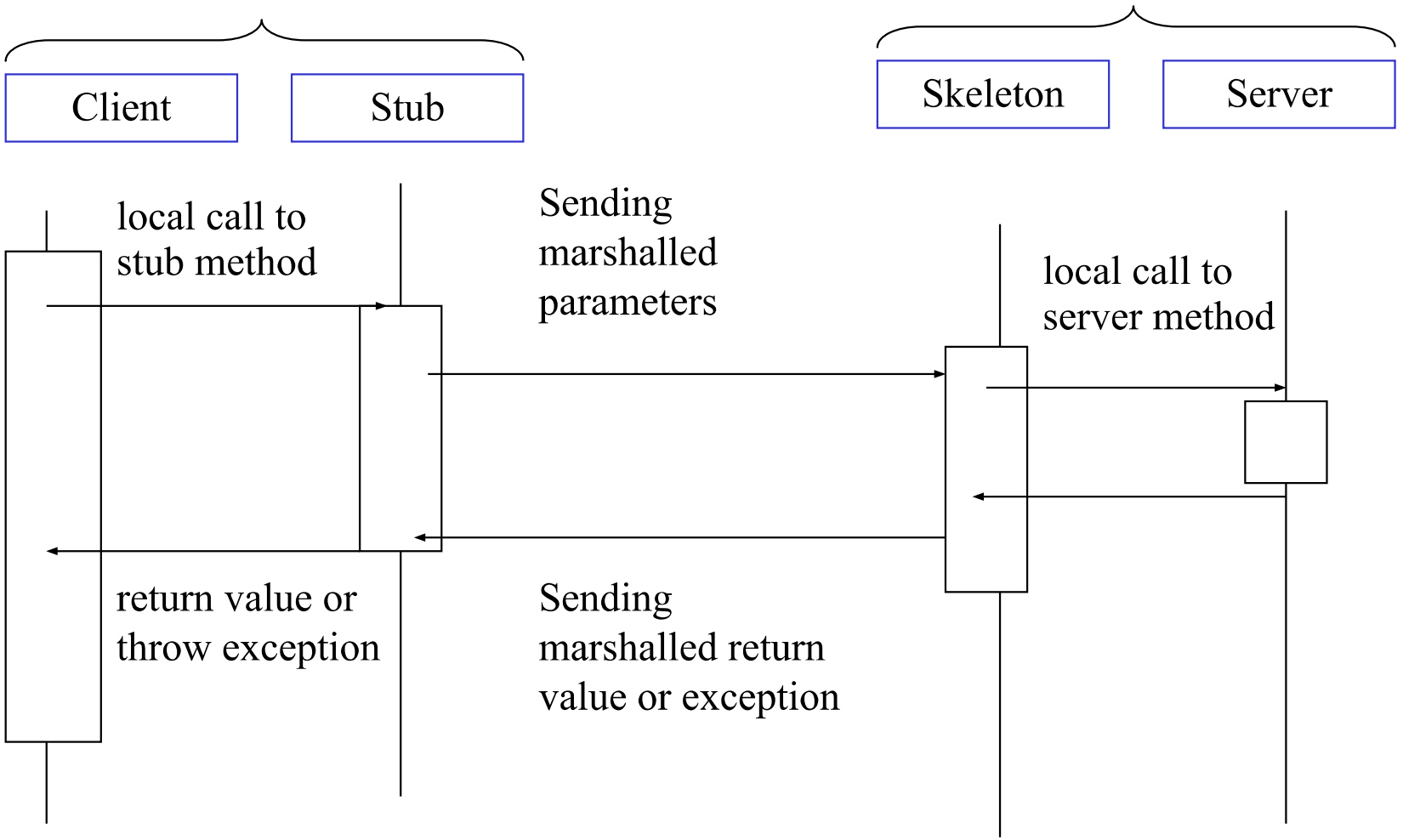
The stub encodes the parameters used in the remote method with a device-independent encoding and transforms them in a format suitable for transport. The process of encoding, writing to a stream and sending an object is referred as **parameter marshalling.**

Thus the stub method on the client side builds an information block that consists of:

- Identifier of remote object to be used;
- Name of the method to be called
- Marschalled parameters.

The reverse process of receiving, reading and decoding is called **parameter unmarshalling.**

# RMI in action



# RMI in action (continued)

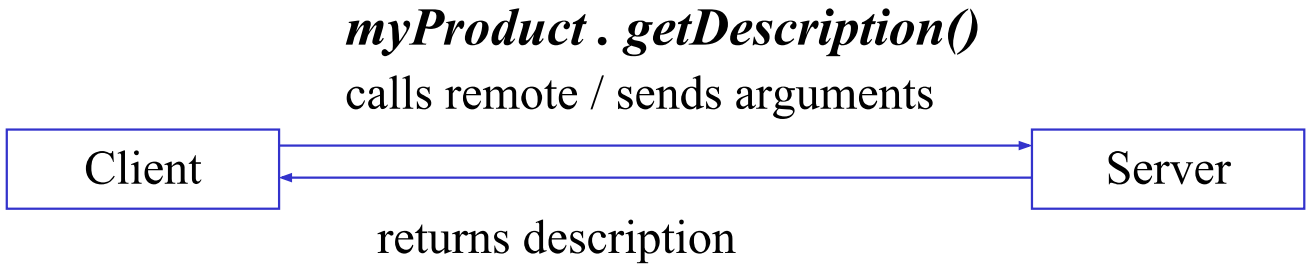
When a stub's method is invoked, it does the following:

- Initiates connection with the JVM on which remote object resides;
- Marshals the parameters
- Waits for result of method invocation
- Unmarshals the value or exception returned
- Returns value to caller

On the server side, the skeleton or receiver object

- Unmarshals the parameters of the remote method
- Locates object to be called
- Calls desired method on remote object implementation
- Captures and marshals return value or exception to the caller.

# 5.2 Anatomy of a simple RMI-based application
# Example 1: Product Info Client-Server




- Client program in class **ProductClient**
- Remote Interface **Product**
- **ProductImpl_Stub** automatically generated by the **rmic** compiler from the **ProductImpl** class

- Server program in class **ProductServer**
- Remote Interface **Product**
- Interface Implementation in class **ProductImpl**
- **ProductImpl_Stub** automatically generated by the **rmic** compiler from the **ProductImpl** class

(HoCo 2002, Ch. 5, Examples 5-1,2,3,5, p. 343-347)

# The Remote Interface

- The remote interface *must be* **public** (it cannot have package access, i.e it cannot be "friendly"). Otherwise the client cannot load a remote object that implements the remote interface
- The remote interface *must extend* the interface **java.rmi.Remote.**
- Each method in the remote interface *must declare* **java.rmi.RemoteException** in its **throw** clause in addition to any application-specific exceptions. This is necessary as remote methods are inherently less reliable than local ones.

# Example: remote interface Product

```
//The Product information remote interface
/**
*@(#) Product.java
*/
import java.rmi.*;

public interface Product extends Remote{
  public String  getDescription( ) throws RemoteException;
}
```

remote interface

must for all methods in remote interface

# Server Side: Implementing the Remote Interface: class ProductImpl

```
/**
*@(#) ProductImpl.java
*/
import java.rmi.*;
import java.rmi.server.*;

public class ProductImpl
  extends UnicastRemoteObject
  implements Product {

  //Constructor
   public ProductImpl ( String n)
     throws RemoteException {
       //super(); called by default
       name = n;
     }
```

implements remote interface

*must* **extend server** subclass **UnicastRemoteObject**, i.e. it is a *server class*

constructor must be *explicitly* defined, even if it is just the default, because *it must* **throw** the **RemoteException**

# Implementing the Remote Interface: class ProductImpl (continued)

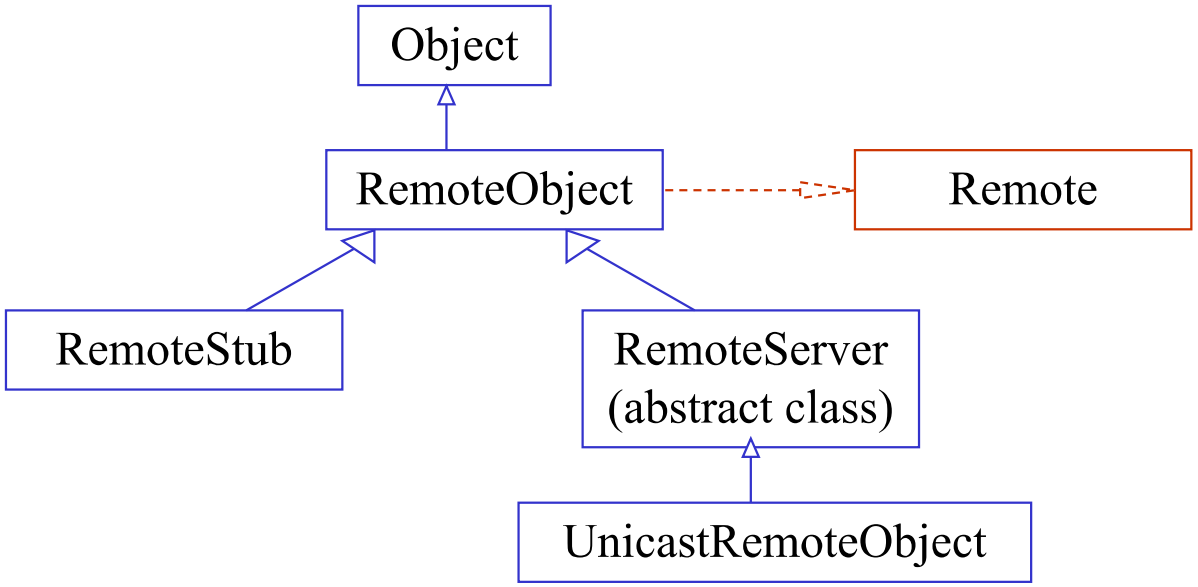
```
    //implementation of remote interface method
    public String getDescription( )
      throws RemoteException {
        return "I am an excellent " + name + ". from server
!";
      }

    private String name;//product name
  }
```

# **UnicastRemoteObject** - API Specification

**public class UnicastRemoteObject extends RemoteServer**

- Concrete class;
- defines a non-replicated remote object whose references are valid only while the server process is alive;
- provides support for point-to-point active object references (invocations, parameters, and results) using TCP streams.



# Server Side Security

In the main( ) method of the server program (here implemented in class **ProductServer) :**

- Create and install security manager to support RMI

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

**Note:** Server examples in the RMI tutorial documentation typically have a security manager installed. However, it is the client not the server that is more in need of a security manger, at it is the one downloading files. A security manger on the server side is a must only if the server is in its turn a client, or there is a special reason to put restrictions.

# The Server creates and registers remote objects

- Create one or more instances of the remote object (**ProductImpl**);
- An application can *bind, unbind, and rebind* registry object references only if it runs on the *same* host as the registry. This shields the registry from hostile clients attempting to change registry information. However, *any* client can lookup objects.
- Register at least one of the remote objects with the RMI remote object registry. The server registers the object with the bootstrap registry service and the client retrieves stubs to those objects. An object is registered by giving the registry its reference and a *name:*

**ProductImpl p = new ProductImpl("Laptop");**
**Naming. bind("//host/Laptop", p);**

In general:

**Naming. bind("//<*hostName*>/<*ObjRegistryName*>",**
**<*objReference*>);**

The RMI ***default port is 1099***

# The Server … (continued)

Optionally, one can also specify a port instead of the default through:

**Naming. bind("//host:3000/Laptop", p);**

In general:

**Naming. bind (**

**"//<*hostName:portnumber*> /<*ObjRegistryName*>",**
**<*objReference*>);**

A remote object can have methods that produce references to other objects. This allows setting it up so that the *client goes to the registry only once.* As it is notoriously difficult to keep names unique (and the names objects are registered with should be unique), one should register as few objects as possible. If one tries to register an object with a name that is already taken, one gets the **AlreadyBoundException**. To avoid this one can use **rebind()** instead of bind(): rebind() either adds a new entry or replaces the existing one.

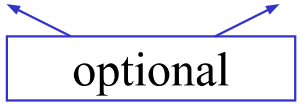
# The Client accesses remote objects

- The *client* gets access to a server object that implements the interface Product by creating a *variable of type interface Product*. The value that is looked up and returned by the server is stored in the variable of type interface.
  The Naming.lookup() method is called and the remote host and object name are specified:

  **Product a = (Product) Naming.lookup(" rmi: //host/Laptop");**

or

**Product a = (Product) Naming.lookup(“rmi: //host:3000/Laptop");**




- A method on the remote object is then called through
  **a . getDescription()**

***Note:*** *the form of the remote call is exactly the same as the form of a local call!!*

# Interface Objects Access Remote Objects

The client variable a, making the remote call

**a . getDescription()**

is an *object of type remote interface* (Product), *NOT* an object of class remote interface implementation (ProductImpl).

Why do we cast to the remote interface instead of to the class that implements it?

The simple answer is: because the client does not have the class that implements the interface.

On the other hand interfaces are *abstract entities* that define methods only, *not objects*. Thus whenever we have an object variable of some interface type, it *must be bound to a real object*. This is where the stub class comes to the rescue!! The *interface object refers to a stub object,* which is *an actual object* of the stub class, i.e. when calling the remote method the remote interface object refers to the stub object representing the remote interface implementation class. The client does not actually know the type of the remote object. It relies on the stub that is obtained by running a special compiler **rmic** on the remote interface implementation class (ProductImpl.class), and is somehow (by hand, or through another call) obtained from the server

# The Server program: class ProductServer

```
/**
*@(#) ProductServer.java
*/
import java.rmi.*;
import java.rmi.server.*;
class ProductServer{
  public static void main(String args[]) {
    try{
                System.setSecurityManager(new
  RMISecurityManager());
          System.out.println("Starting server...");
          System.out.println("Creating remote objects...");
          ProductImpl p = new ProductImpl("Laptop");
          ProductImpl q = new ProductImpl("Display");
          System.out.println("Binding remote objects to registry...");
          Naming.rebind("//host/Laptop", p);
          Naming.rebind("//host/Display", q);
          System.out.println("Waiting for client call...");
    }catch (Exception e){
          System.out.println("Server error:" + e);
          }
  }
}
```

# The Client program: class ProductClient

```
//**
*@(#) ProductClient.java
*/
import java.rmi.*;
import java.net.*;

public class ProductClient {
  public static void main(String args[]){
    try{
      Product a = (Product)Naming.lookup("//host/Laptop");
      Product b = (Product)Naming.lookup("//host/Display");
      System.out.println( a.getDescription( ) );
      System.out.println(b.getDescription());
    }catch (Exception e){
      System.out.println("Client error:" + e);
    }
    System.exit(0);
  }
}
```

Client should have security manager to protect

# 5.3. Location of RMI class files

When building an RMI application threr are basically five categories of files that are generated

- Remote interface classes
- Client implementation classes
- Server implementation classes , including the remote interface implementation class
- Stub classes
- Skeleton classes

To run the application these classes must be distributed as follows

| Client | Server |
| --- | --- |
| • Client implementation classes<br>• Remote interface classes<br><br><br><br>• Stub classes | • Server implementation classes<br>• Remote interface classes<br>• Remote interface implementation class<br>• Stub classes<br>• Skeleton classes |

# Server Side Classes and Objects

**interface <*RemoteInterface*> extends Remote**

**class <*RemoteInterfaceImpl* >**

**extends UnicastRemoteObject**

**implements RemoteInterface**

**//constructor**

**public <*RemoteInterfaceImpl*> throws RemoteException {…}**

**//implementation of *RemoteInterface* methods**

**public <*remoteMethod*> (…) throws RemoteException**

**class <*RemoteInterfaceServer*>**

**main()**

create & set **RMISecurityManager**

create server objects **<*objRemoteInterfaceImpl* >** of type **<*RemoteInterfaceImpl*>**

bind **<*objRemoteInterfaceImpl*>** through **Naming.rebind(" //<*host*>/<*objName* ", <*objRemoteInterfaceImpl*>)**

class *<RemoteInterfaceImpl>_Stub: automatically generated by rmic;*

class *<RemoteInterfaceImpl>_Skel: automatically generated by rmic; not needed in JDK1.2s*

# Client Side Classes and Objects

**interface** ***<RemoteInterface>*** **extends Remote**

**class** ***<RemoteInterfaceClient>***

**main()**

set **RMISecurityManager**

create client objects ***<objRemoteInterface>*** of type ***<RemoteInterface>***
(note difference to server !!) through

***<RemoteInterface>*** ***<objRemoteInterface>*** **=**

**(*<RemoteInterface>*) Naming.lookup**
**("*//<host>/<objName>* ")**

call remote method through

***<objRemoteInterface>* . *<remoteMethod>(…)***

class *RemoteInterfaceImpl_Stub: automatically generated by rmic;*

# Location of RMI class files - Notes

- The stub classes are executed only by the client. However, they are also needed by the server when it is exporting itself for remote access.

- Most RMI applications do not have a cleanly separated client and server functionality, i.e. any participating site has elements of the server as well as the client. Indeed , one of the great strength of RMI is that the server can make remote method calls to the client. It is entirely possible to create pure peer-to-peer RMI applications that have no distinction between the their endpoints. In these cases, the *site where a given remote object resides should include all class files* (thus being the "server" for this object), while the *site from which the object is remotely accessed includes only the interface and the stub.*

## 5.4. Testing the Distributed Application on a single machine

1. Create two separate directories for the client and the server and place the corresponding *.java files in them.
2. Compile the source files for the interface, implementation, client and server classes, e.g.

   **javac Product*.java**

```
C:\Courses\cs667\CompiledExamples\ProductRMI\ProductClient>javac
ProductClient.java
C:\Courses\cs667\CompiledExamples\ProductRMI\ProductClient>dir
Directory of C:\Courses\cs667\CompiledExamples\ProductRMI\ProductClient
09/25/2002  05:20 PM               219 Product.class
10/11/2000  07:05 PM               227 Product.java
09/25/2002  05:20 PM               946 ProductClient.class
09/25/2002  02:42 PM               578 ProductClient.java
```

```
C:\Courses\cs667\CompiledExamples\ProductRMI\ProductServer>javac
ProductServer.java
C:\Courses\cs667\CompiledExamples\ProductRMI\ProductServer>dir
Directory of C:\Courses\cs667\CompiledExamples\ProductRMI\ProductServer
09/25/2002  05:30 PM               219 Product.class
02/04/2002  12:00 AM               230 Product.java
09/25/2002  05:30 PM               641 ProductImpl.class
02/04/2002  12:00 AM               643 ProductImpl.java
09/25/2002  05:30 PM             1,096 ProductServer.class
02/04/2002  12:00 AM               776 ProductServer.java
```

## **rmic** Produces Stub and Skeletons

3. On server side: compile the <objRemoteInterfaceImpl> with the rmic compiler from the command line (in DOS window), e.g.

   **rmic ProductImpl**

   This produces the stub and skeleton for the implementation:

   **ProductImpl_Stub.class** and **ProductImpl_Skel.class**

   There is an option for Java 2 compilation

   **rmic –v1.2 ProductImpl**

   that produces only **ProductImpl_Stub.class** (skeleton is not needed any more)

   More generally: **rmic** ***<options>*** ***<ImplementationClassName>***

   (see documentation for rmic optional argument list)

```
C:\Courses\cs667\CompiledExamples\ProductRMI\ProductServer>rmic
ProductImpl
Directory of C:\Courses\cs667\CompiledExamples\ProductRMI\ProductServer
09/25/2002  05:30 PM               219 Product.class
02/04/2002  12:00 AM               230 Product.java
09/25/2002  05:30 PM               641 ProductImpl.class
02/04/2002  12:00 AM               643 ProductImpl.java
09/25/2002  05:35 PM             1,434 ProductImpl_Skel.class
09/25/2002  05:35 PM             2,876 ProductImpl_Stub.class
09/25/2002  05:30 PM             1,096 ProductServer.class
```

# Stub Classes Needed by Server *and* Client

4. To perform a first test on a single machine *without using automatic class downloading* place a copy of the stub class in the client directory. (If you did not place and compile a <RemoteInterface>.java file in the client directory, now is the time to place a copy of the <RemoteInterface>.class on the client side). If you are working with projects in an IDE and are starting the server and client from the project, you typically need to add the stubs and skeletons in the corresponding projects.

   Of course, when deploying a real application the stub and remote interface classes *will be always automatically downloaded*. The above is for testing purposes *only*.

Note: For the JDK1.1.1 **localhost** does not work with RMI and the name of the machine must be provided. Under 32 bit Windows you can find the machine name as follows: go to Control Panel, select "Network", select "Identification" tab and you will see the Computer Name.

# Start the RMI Registry

5. In the server directory start the RMI registry:

   **start rmiregistry** or **start rmiregistry 3000** for Windows

   **rmiregistry &** or **rmiregistry 3000&** for Unix

Make sure you have an active TCP/IP connection, i.e. you must be connected to your Internet service provider.

```
C:\Courses\cs667\CompiledExamples\ProductRMI\ProductServer>start rmiregistry
```

The registry is launched in separate window and bound to the specified port (default 1099) on machine executing the command. *No text/message* appears in the command line registry window:

If you attempt to start the server before the rmiregistry is running the the **java.rmi.ConnectException** is thrown.

# Run the Server

6. Start server

   **start java ProductServer** for Windows

   (w/o start runs in same window)

   **java ProductServer &** for Unix

Note: In Unix, if the process is not run in the background, it never finishes normally. The main function exits immediately after registration as expected. However, when an object of a class that extends UnicastRemoteObject is created, a separate thread is started that keeps the program alive indefinitely.

```
C:\Courses\667\CompiledExamples\ProductRMI\ProductServer> java ProductServer

Starting server...
Creating remote objects...
Binding remote objects to registry...
Waiting for client call...
```

# Run the Client

7. Start the client, e.g.

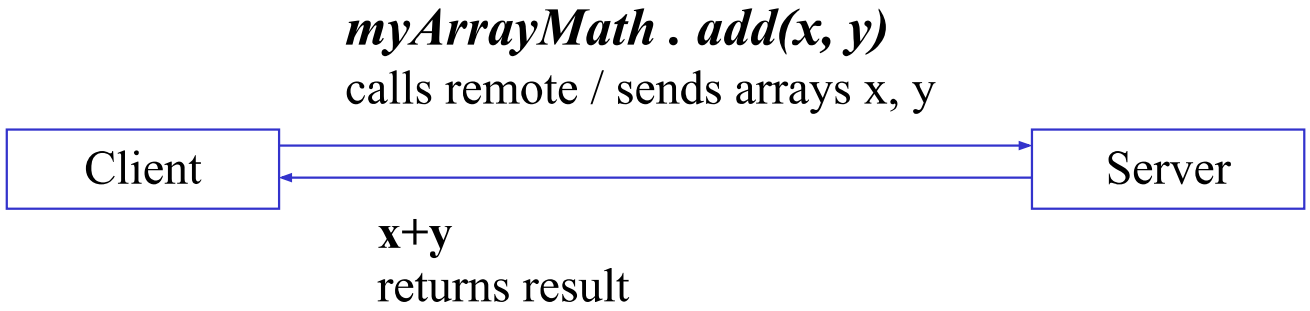
| | |
|---|---|
| **start java ProductClient** | for Windows |
| **java ProductClient** | for Unix |

```
C:\Courses\cs667\CompiledExamples\ProductRMI\ProductClient>java
ProductClient
I am an excellent Laptop. Buy me!
I am an excellent Display. Buy me!
```

# Naming Conventions for RMI classes

| | |
|---|---|
| no suffix | remote interface |
| e.g. Product | |
| **Impl** suffix | server class implementing interface |
| e.g. ProductImp | |
| **Server** suffix | server program that creates server objects |
| e.g. ProductServer | |
| **Client** suffix | client program that calls remote method |
| e.g. ProductClient | |
| **_Stub** suffix | stub class automatically generated by **rmic** |
| e.g. ProductImpl_Stub | program |
| **_Skel** suffix | skeleton class automatically generated by **rmic** e.g. |
| ProductImpl_Skel | (needed for JDK 1.1, but nor for 1.2) |

# Example 2: Array Math




- Client program **ArrayMathClient**
- Remote Interface **ArrayMath**
- **ArrayMathImpl_**Stub Stub class of **ArrayMathImpl**

- Server program **ArrayMathServer**
- Remote Interface **ArrayMath**
- Interface Implementation **ArrayMathImpl**

(Mah 2000, p. 115, modified)

# Example 2: Array Math Client-Server - interface ArrayMath

```
/**
*@(#) ArrayMath.java
*/
public interface ArrayMath
   extends java.rmi.Remote {
    int[] addArray(int a[], int b[]) throws java.rmi.RemoteException;
}
```

# Example 2: class ArrayMathImpl

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
/**
* @(#) ArrayMathImpl.java
*/
public class ArrayMathImpl
      extends UnicastRemoteObject
      implements ArrayMath{
      private String objectName;
      public ArrayMathImpl (String s) throws RemoteException{
      //super( ); called by default
      objectName = s;
      }
      public int[] addArray(int a[], int b[] ) throws RemoteException{
      int sum[] = new int[16];
      for(int i=0; i<sum.length; i++)
      sum[i] = a[i] + b[i];
      return sum;
      }
}
```

# Example 2: class ArrayMathServer

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.net.*;
/**
* @(#) ArrayMathServer.java
*/

class ArrayMathServer{
      public static void main (String argv[]) {
      System.setSecurityManager(new RMISecurityManager());
      try{
            ArrayMathImpl ams = new ArrayMathImpl("ArrayMathServer");
            Naming.rebind("//host/ArrayMathServer", ams);
            System.out.println("ArrayMathServer bound in registry");
      }catch (Exception e){
            System.out.println("ArrayMathServer error"+ e.getMessage());
            e.printStackTrace();
      }
      }
}
```

# Example 2: class ArrayMathClient

```
import java.rmi.*;
import java.net.*;
/**
*@(#) ArrayMathClient.java
*/
public class ArrayMathClient {
    public static void main(String argv[]){
    int x[]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
        int y[]={2,2,2,2,2,2,2,2,2,2, 2, 2, 2, 2, 2, 2};
    int result[]= new int[16];

    try{
        ArrayMath ams =
                (ArrayMath)Naming.lookup("//host/ArrayMathServer");
        result = ams.addArray(x,y);
    }catch(Exception e){
        System.out.println("ArrayMathClient: "+ e.getMessage());
        e.printStackTrace();
    }
```

# Example 2: class ArrayMathClient (continued)

```
   System.out.println("The two arrays are ");
    for(int j=0; j<x.length; j++)
            System.out.print(x[j] + "  ");
    System.out.println();
    for(int k=0; k<y.length; k++)
            System.out.print(y[k] + "  ");
    System.out.println();

    System.out.println("The sum is ");
    for(int i=0; i<result.length; i++)
            System.out.print(result[i] + "  ");
    System.out.println();

  }
}
```