# Unit-4 Distributed Objects and Remote Invocation

**Syllabus:** Distributed Objects and Remote Invocation: Introduction, Communication between Distributed Objects- Object Model, Distributed Object Modal, Design Issues for RMI, Implementation of RMI, Distributed Garbage Collection; Remote Procedure Call, Events and Notifications, Case Study: JAVA RMI

## Topic 01: INTRODUCTION

- Objects that can receive remote method invocations are called remote objects and they implement a remote interface.
- Programming models for distributed applications are:
  - Remote Procedure Call (RPC)
    - Client calls a procedure implemented and executing on a remote computer
    - Call as if it was a local procedure
  - Remote Method Invocation (RMI)
    - Local object invokes methods of an object residing on a remote computer
    - Invocation as if it was a local method call
  - Event-based Distributed Programming
    - Objects receive asynchronous notifications of events happening on remote computers/processes
- Middleware
  - Software that provides a programming model above the basic building blocks of processes and message passing is called middleware.
  - The middleware layer uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocation and events.

(Figure 1)

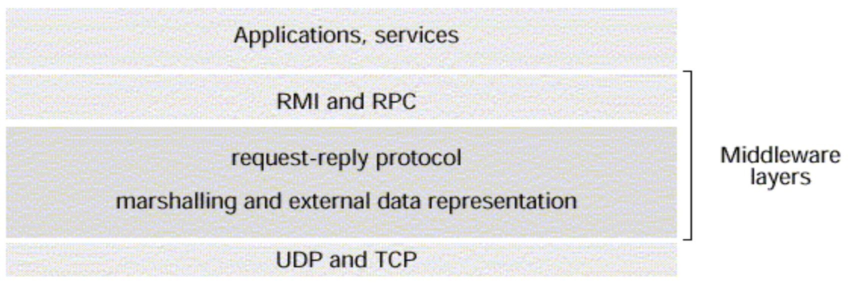


**Figure 1. Middleware layers**

- **Transparency Features of Middleware**
  - Location transparency:
    - In RMI and RPCs, the client calls a procedeure/method without knowledge of the location of invoked method/procedure.
  - Transport protocol transparency:
    - E.g., request/reply protocol used to implement RPC can use either UDP or TCP.
  - Transparency of computer hardware
    - They hide differences due to hardware architectures, such as byte ordering.
  - Transparency of operating system
    - It provides independency of the underlying operating system.
  - Transparency of programming language used
    - E.g., by use of programming language independent Interface Definition Languages (IDL), such as CORBA IDL.

```
// In file Person.idl
struct Person {
string name;
string place;
long year;
} ;
interface PersonList {
readonly attribute string listname;
void addPerson(in Person p) ;
void getPerson(in string name, out Person p);
long number();
};
```

**Figure 2. CORBA IDL example**

- Interfaces for RMI and RPC
  - An explicit interface is defined for each module.

- An Interface hides all implementation details.
- Accesses the variables in a module can only occur through methods specified in interface.
- Interface in distributed system
  - No direct access to remote variables is possible
    - Using message passing mechanism to transmit data objects and variables
      - Request-reply protocols
      - Local parameter passing mechanisms (by value, by reference) is not applicable to remote invocations
    - Specify input, output as attribute to parameters
      - Input: transmitted with request message
      - Output: transmitted with reply message
  - Pointers are not valid in remote address spaces
    - Cannot be passed as argument along interface
    - RPC and RMI interfaces are often seen as a client/server system
  - Service interface (in client server model)
    - Specification of procedures and methods offered by a server
  - Remote interface (in RMI model)
    - Specification of methods of an object that can be invoked by objects in other processes
- Interface Definition Languages (IDL)
  - Impossible to specify direct access to variables in remote classes
  - Hence, access only through specified interface
  - Desirable to have language-independent IDL that compiles into access methods in application programming language

- Example: CORBA IDL

(Figure 2)

Topic No 2: **Remote Procedure Call (RPC)**

- A remote procedure call (RPC) is similar to a remote method invocation (RMI).
- A client program calls a procedure in another program running in a server process.
- RPC, like RMI, may be implemented to have one of the choices of invocation semantics - at-least-once, at-most-once are generally chosen.
- RPC is generally implemented over a request-reply protocol.
- The software that support RPC is shown in Figure 3.

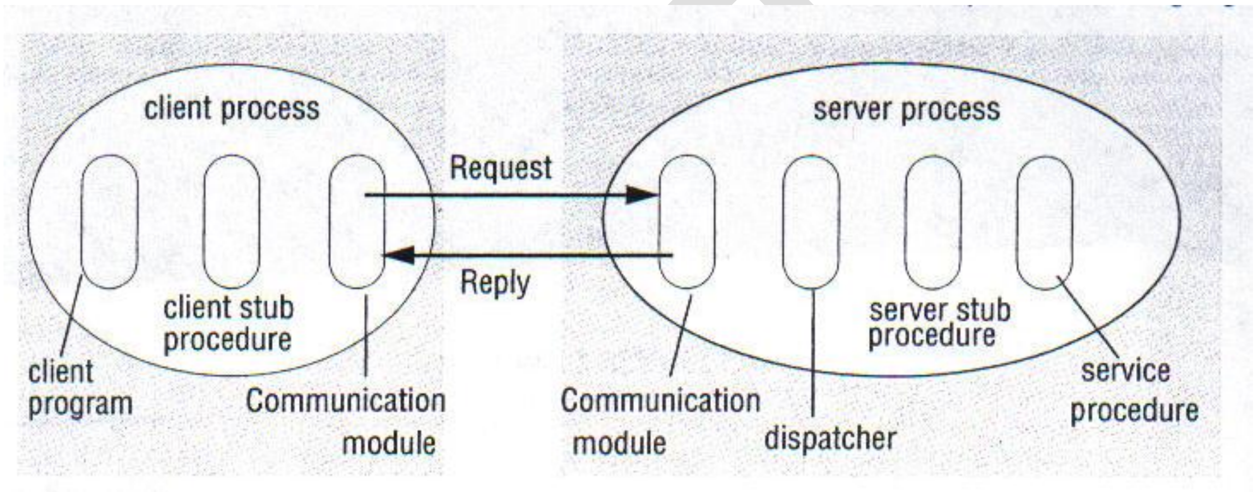


**Figure 3. Role of client and server stub procedures in RPC in the context of a procedural language**

- RPC only addresses procedure calls.
- RPC is not concerned with objects and object references.
- A client that accesses a server includes one stub procedure for each procedure in the service interface.
- A client stub procedure is similar to a proxy method of RMI (discussed later).
- A server stub procedure is similar to a skeleton method of RMI (discussed later).

**RPC Example 1: Local Program**

* A first program (hello.c) to test rpc. Use of this program is for

* testing the logic of the rpc programs.

*#include <stdio.h>*

```
int
main (void) {
    static char * result;
    static char msg[256];
    printf("getting ready to return value\n");
    strcpy(msg, "Hello world");
    result= msg;
    printf("Returning %s\n", result);
    return (0);
} /
```

**Protocol Definition Program**

- The name of this program is hello.x
- The number at the end is version number and should be updated each time the service is updated to make sure the active old copies is not responding to the client program.

```
program HELLO {
  version ONE{
  string PRINT_HELLO() = 1;
  } = 1 ;
} = 0x2000059;
```

**Client Program**

- Now we are ready to use rpcgen (command for generating the required programs).
- Note that so far we have only hello.c and hello.x
- After running "rpcgen -a -C hello.x" the directory contain following files:

```
-rw-rw-r--  1 aabhari aabhari    131 Oct  5 12:15 hello.c
-rw-rw-r--  1 aabhari aabhari    688 Oct  5 12:19 hello.h
-rw-rw-r--  1 aabhari aabhari     90 Oct  5 12:18 hello.x
-rw-rw-r--  1 aabhari aabhari    776 Oct  5 12:19 hello_client.c
-rw-rw-r--  1 aabhari aabhari    548 Oct  5 12:19 hello_clnt.c
```

```
-rw-rw-r--  1 aabhari aabhari   316 Oct  5 12:19 hello_server.c
-rw-rw-r--  1 aabhari aabhari  2076 Oct  5 12:19 hello_svc.c
```

- The two templates that we should modify for this example are hello_client.c and hello_server.c.

**Template of hello_client Program**

* This is sample code generated by rpcgen.

* These are only templates and you can use them as a guideline for developing your own functions.

```
#include "hello.h"
void hello_1(char *host)
{
    CLIENT *clnt;
    char * *result_1;
    char *print_hello_1_arg;
#ifndef DEBUG
    clnt = clnt_create (host, HELLO, ONE, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif  /* DEBUG */
    result_1 = print_hello_1((void*)&print_hello_1_arg, clnt);
    if (result_1 == (char **) NULL) {
        clnt_perror (clnt, "call failed");
    }
#ifndef DEBUG
 clnt_destroy (clnt);
#endif  /* DEBUG */
}
```

```
Int main (int argc, char *argv[])

{

    char *host;

    if (argc < 2) {

        printf ("usage: %s server_host\n", argv[0]);

        exit (1);

    }

    host = argv[1];

    hello_1 (host);

exit (0);

}
```

**hello_client Program**

- We have to modified hello_client template program.
- The modifications for our first example are very simple.
- Next show the modified program of hello_client that needs only few lines.
- * This is sample code generated by rpcgen.
- * These are only templates and you can use them as a guideline for developing your own functions.

```
#include "hello.h"

#include <stdlib.h>

#include <stdio.h>

void

hello_1(char *host)

{

    CLIENT *clnt;

    char * *result_1;

    char *print_hello_1_arg;
#ifndef DEBUG
clnt = clnt_create (host, HELLO, ONE, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif  /* DEBUG */
    result_1 = print_hello_1((void*)&print_hello_1_arg, clnt);
    if (result_1 == (char **) NULL)
        clnt_perror (clnt, "call failed");
    else printf(" from server: %s\n",*result_1);
#ifndef DEBUG
    clnt_destroy (clnt);
#endif   /* DEBUG */
}
int
main (int argc, char *argv[])
{
    char *host;
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    hello_1 (host);
exit (0);
} //end clinet_server.c
```

**Template of hello-server Program**

* This is sample code generated by rpcgen.

* These are only templates and you can use them as a guideline for developing your own functions.

```
#include "hello.h"

char **

print_hello_1_svc(void *argp, struct svc_req *rqstp)

{

    static char * result;

    /* insert server code here */

    return &result;

}
```

**hello-server Program**

* This is sample code generated by rpcgen.

* These are only templates and you can use them as a guideline for developing your own functions.

```
#include "hello.h"

char **

print_hello_1_svc(void *argp, struct svc_req *rqstp)

{

    static char * result;

    static char msg[256];

    printf("getting ready to return value\n");

    strcpy(msg, "Hello world");

    result= msg;

    printf("Returning\n");

    return &result;
```

}

**Making Client and Server Program**

- To compile the client

```
leda% gcc   hello_client.c    hello_clnt.c    -o client  -lnsl
```

- To compile the server

```
leda% gcc   hello_server.c    hello_svc.c   -o server  -lnsl
```

- To run the server use

```
leda% ./server
```

- To run the client use

```
elara% ./client leda
```

**RPC**

- rpcgen facilitates the generation of client and server stubs from the IDL program.
- It even generates client and server template programs.
- The option -a is passed to rpcgen and also all the generation of all support files including the make files.
- The -a option causes the rpcgen to halt with warnings if template files with default names exist.
- Consider turning a factorial program into a client-server program using RPC.

**RPC Example 2**

```
/* A program to calculate factorial numbers. */
#include <stdio.h>
void main(void){
    long int f_numb, calc_fact(int);
    int     number;
    printf("Factorial Calculator\n") ;
    printf("Enter a positive integer value");
    scanf("%d", &number);
    if (number < 0)
      printf("Positive value only!\n");
    else if ((f_numb = calc_fact(number)) > 0)
      printf("%d! = %d\n", number, f_numb);
    else
      printf("Sorry %d! is out of my range!\n", number);
  }
/* Calculate the factorial number and return the result or return 0 if
   the value is out of range. */
   long int calc_fact(int n) {
    long int  total = 1, last = 0;
    int     idx;
    for (idx = n; id<= 1; --idx)    {
      total *=idx;
      if (total < last)             /* Have we gone out of range? */
        return (0);
      last = total;
    }
  return (total);
}
/* The protocol definition file for the factorial program.
   the file name is fact.x */


program FACTORIAL {
   version ONE{
   long int CALC_FAC(int) = 1;
   } = 1 ;
} = 0x2000049;
```

- We may now use rpcgen with the option flags -a -C to generate header file, the client and the server stub files and in addition, the client and server template programs.

- The content of the fact_client.c program is as shown below.

```
/* This is sample code generated by rpcgen.

   These are only templates and you can use them as aguideline for developing your own functions. */

#include "fact.h"

void factorial_1(char *host)

{

   CLIENT *clnt;

    long *result_1

    int calc_fac_1_arg;

     #ifndef DEBUG

     clnt = clnt_create(host, FACTORIAL, ONE, "netpath");

     if (clnt == (CLIENT *) NULL) {

     clnt_pcreateerror(host);

      exit(1);

}

#endif  /* DEBUG */

      result_1 = calc_fac_1(&calc_fac_1_arg, clnt);

      if (result_1 == (long *) NULL) {

       clnt_perror(clnt, "call failed");

       }

       #ifndef DEBUG

       clnt_destroy(clnt);

        #endif  /* DEBUG */

}

main(int argc, char *argv[])

{

   char *host;
```

```
  if (arg < 2) {
  printf("usage: %s server_host\n", argv[0]);
  exit(1);
}
  host = argv[1];
   factorial_1(host);
}
```

- The template code for client needs to be modified to conform our original program.

```
/* This is sample code generated by rpcgen.
   These are only templates and you can use them
   as aguideline for developing your own functions. */
#include "fact.h"
#include  <unistd.h>      /* added because we will call exit*/
long int factorial_1(int calc_fac_1_arg, char *host)
{
  CLIENT *clnt;
   long *result_1;
#ifndef  DEBUG
        clnt = clnt_create(host, FACTORIAL, ONE, "netpath");
if (clnt == (CLIENT *) NULL) {
    clnt_pcreateerror(host);
     exit(1);
}
     #endif         /* DEBUG */
     result_1 = calc_fac_1(&calc_fac_1_arg, clnt);
     if (result_1 == (long *) NULL) {
      clnt_perror(clnt, "call failed");
      }
      #ifndef DEBUG
              clnt_destroy(clnt);
      #endif          /* DEBUG */
          return *result_1;
}
main(int argc, char *argv[])
{
  char *host;
/* Add own declarations here */
   long int  f_numb;
   int       number;

   if (arg < 2) {
   printf("usage: %s server_host\n", argv[0]);
   exit(1);
}
   host = argv[1];
/* This is the code from the previous main in program fact.c */
    printf("Factorial Calculation\n");
    printf("Enter a positive integer value");
    scanf("%d", &number);
    if  (number < 0)
       printf("Positive values only\n");
    else if ((f_numb = factorial_1(number, host)) >0 )
           printf("%d! = %d\n", number, f_numb);
     else
         printf("Sorry %d! is out of my range!\n", number);
```

```
}
```

- Here is the fact_server.c template generated by rpcgen

```
/* This is sample code generated by rpcgen.
   These are only templates and you can use them
   as aguideline for developing your own functions. */
   #include "fact.h"
    long int * calc_fac_1_srv(int *argp, struct svc_req *rqstp)
   {
       static long result;
     /* insert server code here */
          return(&result);
    }
```

- Here is the fact_server.c template with modification code

```
/* This is sample code generated by rpcgen.
   These are only templates and you can use them
   as aguideline for developing your own functions.  */
#include "fact.h"
    long int * calc_fac_1_srv(int *argp, struct svc_req *rqstp)
   {
       static long int result;
     /*  insert server code here */
       long int total = 1, last = 0;
       int idx;

for(idx = *argp; idx – 1; --idx)
          total *= idx;
           if (total <= last)     /* Have we gone out of range? */
```

```
            {
              result = 0;
              return (&result);
             }
           last = total;
         }
            result = total;
            return (&result);
      }
```

- Here is a modified makefile.fact
- # This is a template make file generated by rpcgen

```
#parameters
#added CC = gcc to use gcc
CC = gcc
CLIENT = fact_client
SERVER = fact_server
SOURCES_CLNT.c =
SOURCES_CLNT.h =
SOURCES_SVC.c =
SOURCES_SVC.h =
SOURCES.x = fact.x
TARGETS_SVC.c = fact_svc.c fact_server.c
TARGETS_CLNT.c = fact_clnt.c fact_client.c
TARGETS = fact.h fact_clnt.c fact_svc.c fact_client.c fact_server.c
OBJECT_CLNT = $(SOURCES_CLNT.c:%.c=%.o) $(TARGETS_CLNT.c:%.c=%.o)
 OBJECT_SVC = $(SOURCES_SVC.c:%.c=%.o) $(TARGETS_SVC.c:%.c=%.o)
```

```
# Compiler flags
CFLAGS += -g
LDLIBS += -lnsl
# added –C flag to PRCGENFLAGS or add –lm to LDLIBS
RPCGENFLAGS = -C
#Targets
all : $(CLIENT) $(SERVER)
$(TARGETS) : $(SOURCES.x)
        rpcgen $(RPCGENFLAGS) $(SOURCES.x)
$(OBJECTS_CLNT) : $(SOURCES_CLNT.c) $ (SOURCES_CLNT.h) \
         $(TARGETS_CLNT.c)
$(OBJECTS_SVC) : $(SOURCES_SVC.c) $ (SOURCES_SVC.h) \
         $(TARGETS_SVC.c)
$(CLIENT) : $(OBJECTS_CLNT)
        $ (LINK.c) –o $(CLIENT) $(OBJECTS_CLNT) $(LDLIBS)
$(SERVER) : $(OBJECTS_SVC)
        $ (LINK.c) –o $(SERVER) $(OBJECTS_SVC) $(LDLIBS)
Clean:
       $(RM) core $(TARGETS) $(OBJECTS_CLNT) $(OBJECTS_SVC) \
       $(CLIENT) $(SERVER)
```

**Strength and Weaknesses of RPC**

- RPC is not well suited for adhoc query processing. Consider the use of RPC to make SQL requests on servers that return arbitrary size of rows and number of tuples.
- It is not suited for transaction processing without special modification.
- A separate special mode of quering is proposed – Remote Data Access (RDA).
- RDA is specially suited for DBMS.
- In a general client_server environment both RPC and RDA are needed.

Topic No 3: **Java RMI**

- Java RMI extends the Java object model to provide support for distributed objects in the Java language.
- It allows object to invoke methods on remote objects using the same syntax as for local invocation.
- It is a single language system – remote interfaces are defined in the Java language.
- An object making a remote invocation needs to be able to handle RemoteExceptions that are thrown in the event of communication subsystem failures
- A remote object must implement Remote interface.
- The next example is a simple Hello world program that is implemented in Java RMI

**Java RMI- Example 1**

- In this example we do the followings:
  - Define the remote interface
  - Implement the server
  - Implement the client
  - Compile the source files
  - Start the Java RMI registry, server, and client

*Hello.java - a remote interface*

```
import java.rmi.Remote;

import java.rmi.RemoteException;

public interface Hello extends Remote {

   String sayHello() throws RemoteException;

}

// Implement the server

import java.rmi.registry.LocateRegistry;

import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {
   public Server() {}
   public String sayHello() {
           return "Hello, world!";
   }
public static void main(String args[]) {

           try {
              Server obj = new Server();
              Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
              // Bind the remote object's stub in the registry
              Registry registry = LocateRegistry.getRegistry();
              registry.bind("Hello", stub);
              System.err.println("Server ready");
           } catch (Exception e) {
              System.err.println("Server exception: " + e.toString());
              e.printStackTrace();
           }
   }
}
// Implement the client
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Client {
   private Client() {}
public static void main(String args[]) {
           String host = (args.length < 1) ? null : args[0];
      try {
          Registry registry = LocateRegistry.getRegistry(host);
```

```
        Hello stub = (Hello) registry.lookup("Hello");
        String response = stub.sayHello();
        System.out.println("response: " + response);
      } catch (Exception e) {
        System.err.println("Client exception: " + e.toString());
        e.printStackTrace();
      }
    }
}
```

- The source files for this example can be compiled as follows:
  - javac  Hello.java Server.java Client.java
  - Start the Java RMI registry :
  - rmiregistry
  - Start the Server: java Server
  - Start the Client: java Client

**Java RMI Example 2**

- Shared whiteboard
  - » It allows a group of users to share a common view of a drawing surface containing graphical objects.
  - » The server maintains the current state of a drawingClients can poll the server about the latest shape of a drawing.
  - » The server attaches version numbers to new arriving shapes.

- Remote Interface
  - Remote interfaces are defined by extending an interface called Remote provided in the java.rmi package.
  - Figure 4 shows an example of two remote interface called Shape and ShapeList.

```
import java.rmi.*;

import java.util.Vector;

public interface Shape extends Remote {

     int getVersion() throws RemoteException;

    GraphicalObject getAllState() throws RemoteException;          1

}

public interface ShapeList extends Remote {

    Shape newShape(GraphicalObject g) throws RemoteException;  2

   Vector allShapes() throws RemoteException;

   int getVersion() throws RemoteException;

}
```

**Figure 4. Java Remote interfaces Shape and ShapeList**

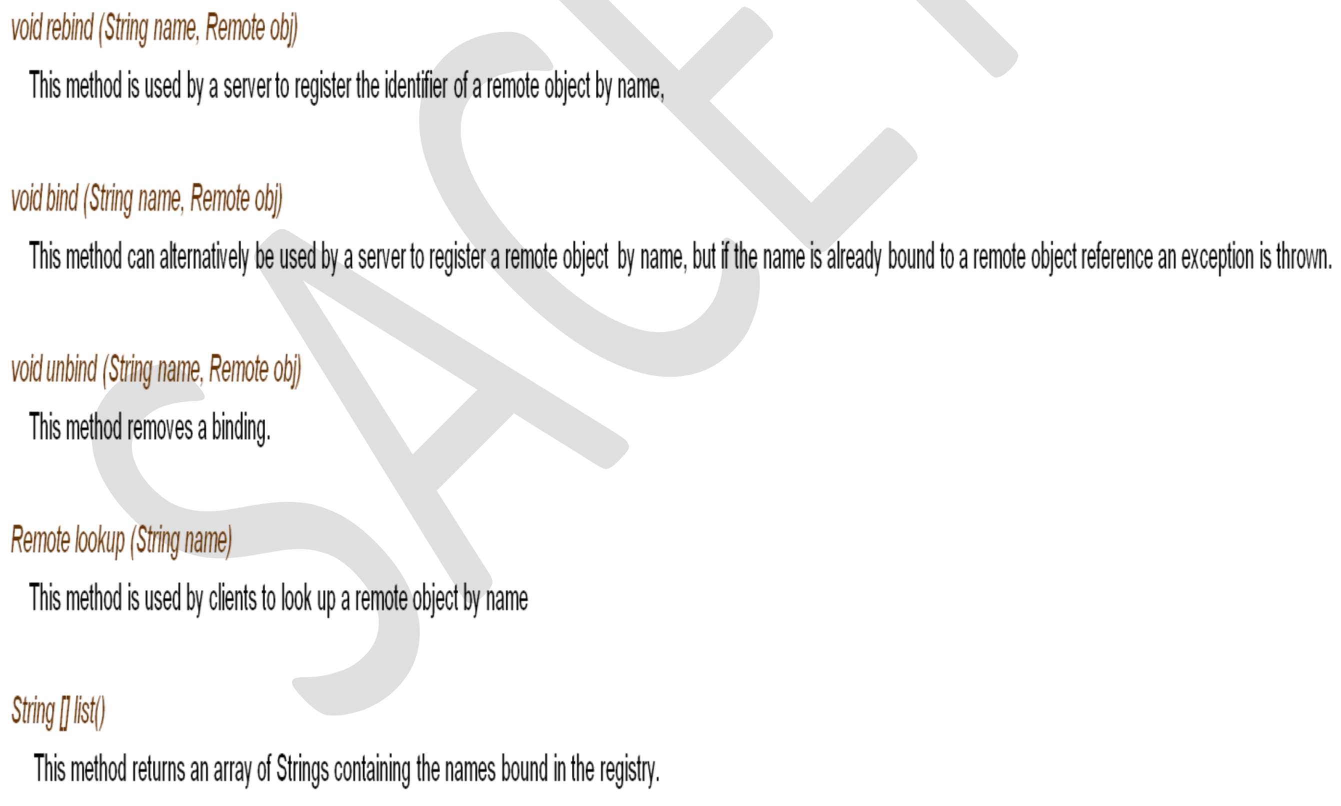
- In this example, GraphicalObject is a class that holds the state of a graphical object.
- GraphicalObject must implement the Serializable interface.
- Ordinary and remote objects can appear as input and output arguments.

- Parameter and result passing
  - Passing remote objects
    - The result is passed by (object) reference.
    - In line2, the return value of the method newShape is defined as shape - a remote interface.
    - When a remote object reference is received, an RMI can be issued on the object refered to by this reference.
  - Passing non-remote objects
    - The result is passed by value.
    - A new object is created locally, with the state differing from the original object.
- Downloading of Classes
  - Classes can be transferred from one Java VM to another.
  - Parameters are passed by value or by reference.

- If the recipient does not yet possess the class of an object passed by value, its code is downloaded automatically.
- If the recipient of a remote object reference does not yet possess the class for a proxy, its code is downloaded automatically.

- RMIregistry
  - The RMIregistry is the binder for Java RMI.
  - The RMIregistry runs on every server that hosts remote objects.
  - It maps local object names of the form *//computerName:port/objName* to object references.
  - This service is not a global service.
  - Clients need to query a particular host to get reference.

(Figure 5)



*void rebind (String name, Remote obj)*

This method is used by a server to register the identifier of a remote object by name,

*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

*String [] list()*

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

**Figure 5. The Naming class of Java RMIregistry**

```
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 6. Java class ShapeListServer with main method**

- In Figure 6:
  - Security Manager
    - implements various security policies for client accesses
  - Main method
    - 1: create instance of ShapeListServant
    - 2: binds name "ShapeList" to newly created instance in RMIRegistry
- *ShapeListServant* implements *ShapeList*
  - Figure 7 gives an outline of the class ShapeListServant.
    - 1: UnicastRemoteObject - objects that live only as long as creating process
    - 2: factory method - client can request creation of a new object

```
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 7. Java class ShapeListServant implements interface ShapeList**

- Client program
  - A simplified client for the ShapeList sever is illustrated in Figure 8.
    - polling loop:
      - 1: look up remote reference
      - 2: invoke allShapes() in remote object

```
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 8. Java client of ShapeList**

- Figure 9 shows the inheritance structure of the classes supporting Java RMI servers.

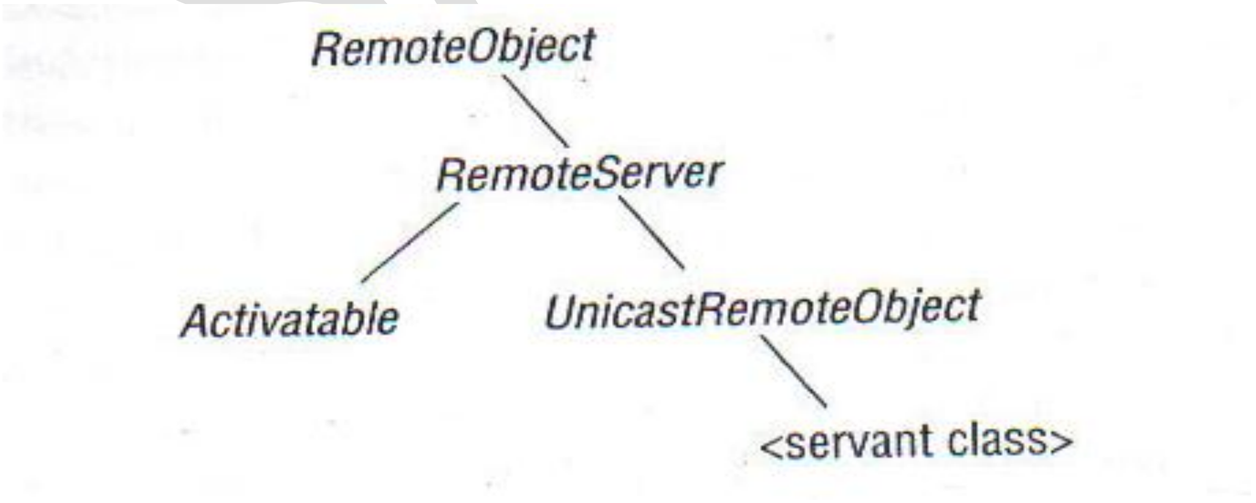


**Figure 9. Classes supporting Java RMI**