Unit-5

RMI

Distributed Object Model

Remote Object - an object whose methods can be invoked from another Virtual Machine.

Remote Interface - a Java interface that declares the methods of a remote object.

Remote Method Invocation (RMI) - the action of invoking a method of a remote interface on a remote object.

Marshals (Write & Transmits)

UnMarshals (read)

Stubs: (Client Side) / an object act as an Gateway

Skeleton: (Server Side) / an object act as an Gateway

What is RMI ?

- The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java.

- The RMI allows an object to invoke methods on an object running in another JVM.
- The RMI provides remote communication between the applications using two objects *stub(Client Side)* and *skeleton(Server Side)*.

Q1. The Roles of Client and Server

OR

How Client Server Communicates in Distributed Systems OR Roles of Client & Server in Distributed Systems

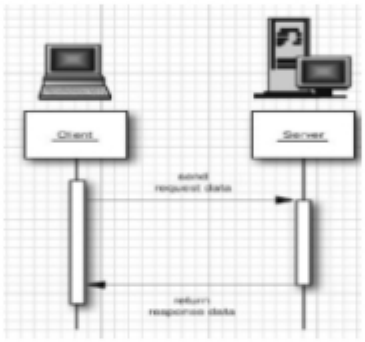


Figure .1. Transmitting objects between client and server

- In the Traditional Client / Server model, the client translates the request to an Intermediary transmission format and Sends the request data to the server.
- The server parses the request format, computers the respose and formats the respose for transmission to the client.
- The client then parses the response and displays it to the user.

- *Now, we have to design a transmission format and We have to write code for conversion between your data and the transmission format.*

- *Now, What we want is a mechanism by which the client programmer makes a regular method call, The problem is that the object that carries out the work is not located in the same virtual network.*

- *The solution is to install a proxy for the server object on the client.*

- *The client calls the proxy , making a regular method call. The client proxy contacts the server.*

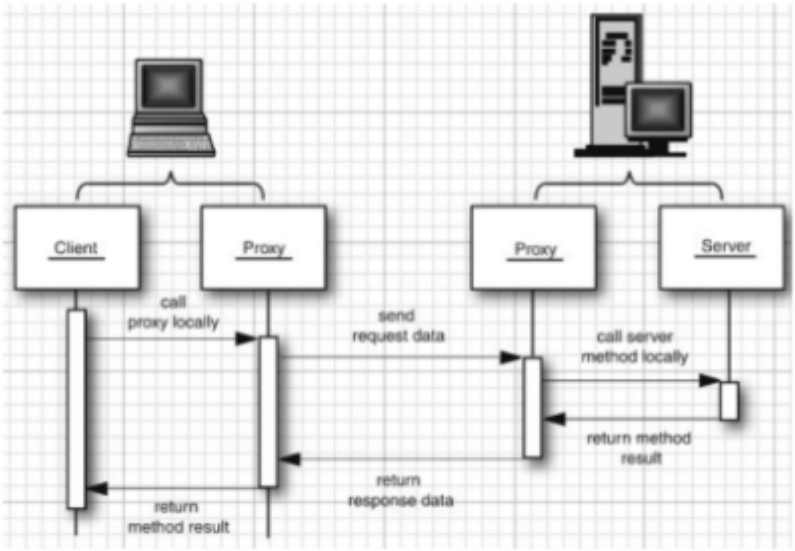


Figure .2. Remote method call with proxies

- *Similarly, the programmer of the server object doesn't want to fuss(show unnecessary) with client communication.*
- *The solution is to install a second proxy object on the server.*
- *The server proxy communicates with the client proxy, and it makes regular method calls to the Server object.*

Q1.1. How do proxies communicate with each other ?

a. *RMI*

Its supports method calls between distributed Java objects.

b. *CORBA (Common Object Request Broker Architecture)*

- *Its* supports method calls between objects of any programming language.
- CORBA uses the binary Internet Inter-ORB(object request brokers) Protocol, or IIOP(Internet Inter-ORB Protocol), to communicate between objects.

c. *SOAP(Simple Object Access Protocol)*

- web services architecture is a collection of protocols, sometimes collectively described as WS-*.
- It is also programming-language neutral.
- However, it uses XML-based communication formats.
- The format for transmitting objects is the Simple Object Access Protocol (SOAP).

- If you have an access to an object on a different machine, you can call methods of the remote object. Of course, the method parameters must somehow be shipped to the other machine, the server must be informed to execute the method & the return value must be shipped back.

- In RMI, the object whose methods makes the remote call is called the **client object.**

- The remote object is called the **server object**.
- The computer running the java code that calls the remote method is the **client** for that call.

- The computer hosting the object that processes the call is the **server** for that call.

Registration (binding)

A server can register its remote objects with a naming service – the rmiregistry. Once registered, each remote object has a unique URL.

Obtaining a remote object reference.

- *A client can use the naming service to Lookup a URL to obtain a remote object reference.*
  - *The application can pass and return remote object references as part of its operation.*

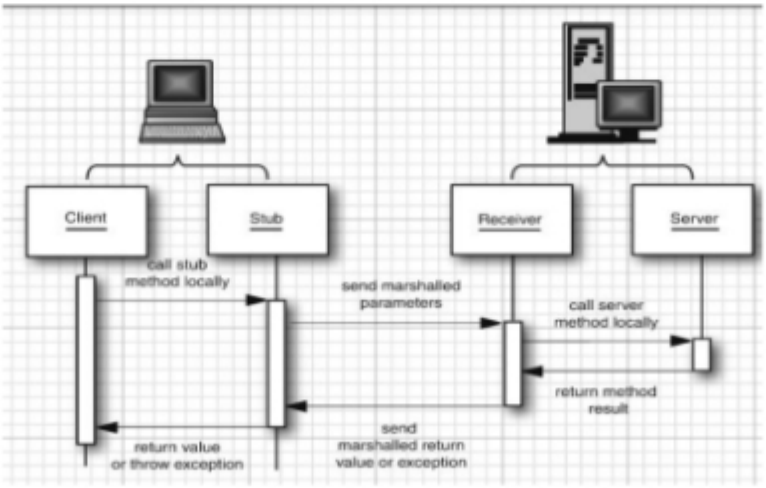
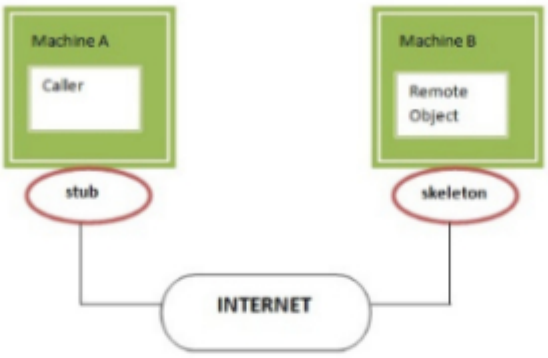


Figure .3. Parameter marshalling

Stubs & Parameter Marshalling :



Stubs: (Client Side) / an object act as an Gateway

- when client code invokes a remote method on a remote object, it actually calls an ordinary method on a proxy object called a *stub*. For example

```
Warehouse centralWarehouse = get stub object;
double price = centralWarehouse.getPrice("Blackwell Toaster");
```

- The stub resides on the client machine.The stub packages the parameters used in the remote method into a block of bytes. This packaging uses a device independent encoding for each parameter.
- The process of encoding the parameters is called *parameter marshalling*. The Purpose of parameter marshalling is to convert the parameters into a format suitable for transport from one machine to another.

Stub functions include:

- Initiating a call to the remote object.
- A description of the method to be called.
- The marshalled parameters.
- Unmarshals(read) the return value or exception from the server.
- Return value to the caller.

Skeleton

A proxy object on server side is called Skeleton. Skeleton function includes:

a) It unmarshals the parameters.
b) It locates the object to be called.
c) It calls the desired method.
d) It captures & marshals(writes and transmits) the return value or exception of the call.
e) Returns marshalled data back to the stub.

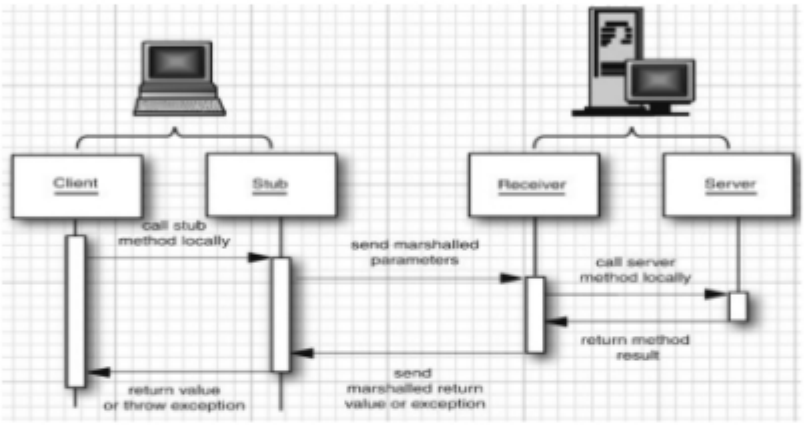


Figure 3. Parameter marshalling

3	• The RMI Programming Model

3.1. Interfaces and Implementations
3.2. The RMI Registry
3.3. Deploying the Program
3.4. Logging RMI Activity

Example:

- A remote object represents a warehouse.
- The client program asks the warehouse about the price of a product.

In the following sections, you will see how to implement and launch the server and client programs.

3.1. Interfaces and Implementations

The capabilities of remote objects are expressed in interfaces that are shared between the client and server.

Listing 3.1. warehouse1/Warehouse.java

```
import java.rmi.*;
 /**
 The remote interface for a simple warehouse.
 */

 public interface Warehouse extends Remote
 {
     double getPrice(String description) throws RemoteException;
 }
```

Listing 3.2. warehouse1/WarehouseImpl.java

The term "unicast" refers to the fact that the remote object is located by making a call to a single IP address and port.

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

/**
* This class is the implementation for the remote Warehouse interface.
*/
public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
{
      private Map<String, Double> prices;

      public WarehouseImpl() throws RemoteException
      {
       prices = new HashMap<>();
       prices.put("Blackwell Toaster", 24.95);
       prices.put("ZapXpress Microwave Oven", 49.95);
      }

       public double getPrice(String description) throws RemoteException
       {
             Double price = prices.get(description);
             return price == null ? 0 : price;
       }
}
```

3.2. The RMI Registry

To access a remote object that exists on the server, the client needs a local stub object.
How can the client request such a stub? The most common method is to call a remote method of another remote object and get a stub object as a return value.

a chicken-and-egg problem here:
The first remote object has to be located some other way. For that purpose, the JDK provides a bootstrap registry service.
To register a remote object, you need a RMI and Reference to the Implementation object.

Syntax:

URLs start with **rmi:** & contain an optional hostname, and port , and the name of the remote object that unique.

Eg:

rmi://regserver.mycompany.com:99/central_warehouse

By default:

host: localhost
port: 1099

Server tells registry at given location to associate or "bind" name with object.

Here is the code for registering a WarehouseImpl object with the RMI registry on the same server:

```
WarehouseImpl centralWarehouse = new WarehouseImpl();
Context namingContext = new InitialContext();
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

A client can enumerate(mention) all registered RMI objects by calling:

Enumeration: the action of mentioning a number of things one by one.

```
Enumeration<NameClassPair> e = namingContext.list("rmi://regserver.mycompany.com");
```

Displays the names of all registered objects:

```
while (e.hasMoreElements())
System.out.println(e.nextElement().getName());
```

A client gets a stub to access a remote object by specifying the server and the remote object name in the following way:

```
String url = "rmi://regserver.mycompany.com/central_warehouse";
Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);
```

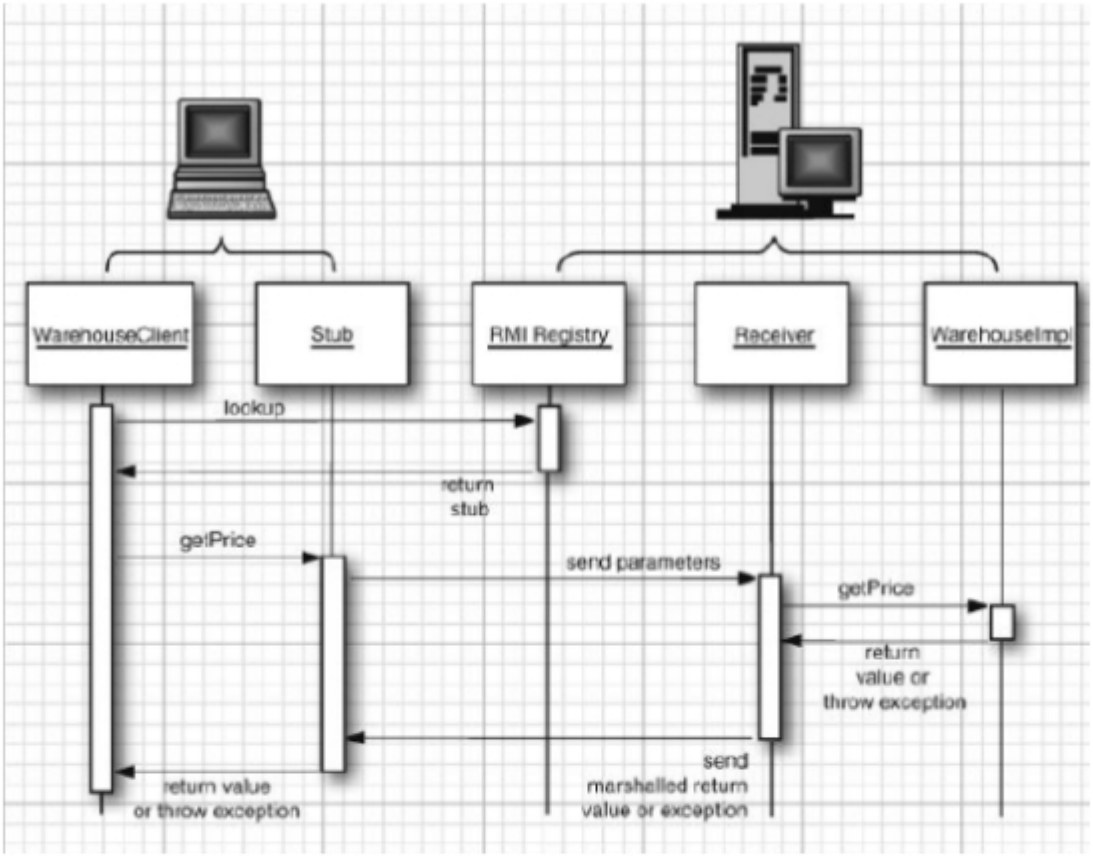


Figure .4. Calling the remote getPrice method

Listing .3. warehouse1/WarehouseServer.java

```
import javax.naming. * ;

/**
 * This server program instantiates a remote warehouse object, registers it with
the naming
 * service, and waits for clients to invoke methods.
*/
public class WarehouseServer{
    public static void main(String[]args)throws RemoteException,
    NamingException{
        System.out.println("Constructing server implementation...");
        WarehouseImpl centralWarehouse = new WarehouseImpl();

        System.out.println("Binding server implementation to registry...");
        Context namingContext = new InitialContext();
        namingContext.bind("rmi:central_warehouse", centralWarehouse);

        System.out.println("Waiting for invocations from clients...");
    }
}
```

Listing 11.4. warehouse1/WarehouseClient.java

```
import java.rmi.*;
import java.util.*;
import javax.naming.*;

/**
 * A client that invokes a remote method.
 */
public class WarehouseClient{
    public static void main(String[]args)throws NamingException,
    RemoteException{
        Context namingContext = new InitialContext();

        System.out.print("RMI registry bindings: ");
        Enumeration < NameClassPair > e =
namingContext.list("rmi://localhost/");

        while (e.hasMoreElements())
            System.out.println(e.nextElement().getName());

        String url = "rmi://localhost/central_warehouse";
        Warehouse centralWarehouse =
(Warehouse)namingContext.lookup(url);

        String descr = "Blackwell Toaster";
        double price = centralWarehouse.getPrice(descr);
        System.out.println(descr + ": " + price);
    }
}
```

11.3.3. Deploying the Program

Deploying an application that uses RMI can be tricky because so many things can go wrong—and the error messages you get when something goes wrong are so poor.

Make two separate directories to hold the classes for starting the server and client.

```
server/
        WarehouseServer.class
        Warehouse.class
        WarehouseImpl.class
client/
        WarehouseClient.class
        Warehouse.class
```

download/
Warehouse.class

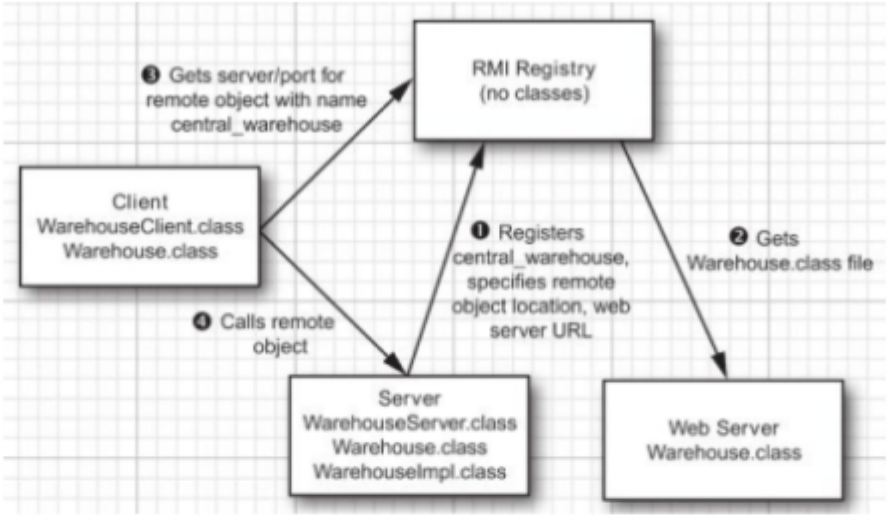


Figure 5. Server calls in the Warehouse application

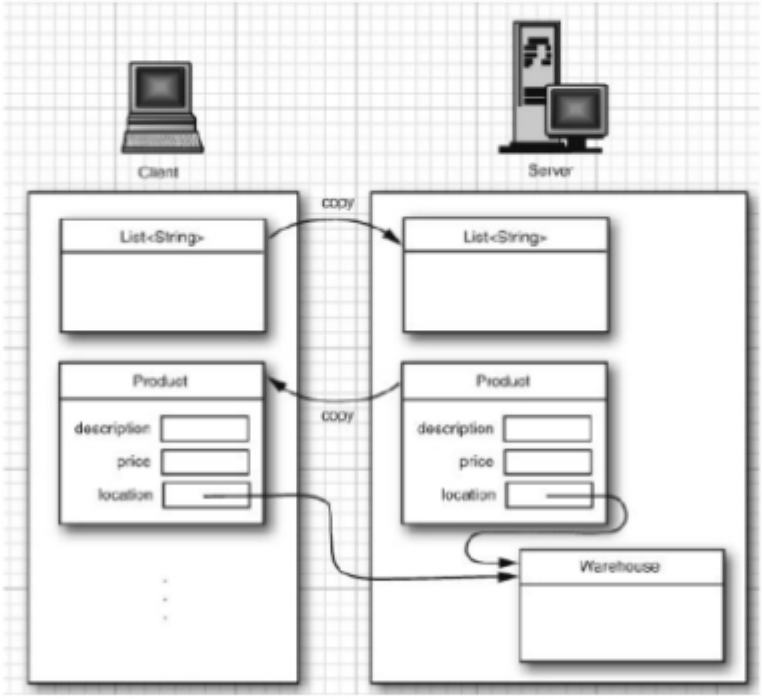


Figure 6. Testing an RMI application

.3.4. Logging RMI Activity

If you start the server with the option

```
-Djava.rmi.server.logCalls=true WarehouseServer
```

Then the server logs all remote method calls on its console.
For additional logging messages, you have to configure RMI loggers using the standard Java logging API.

Make a file logging.properties with the following content:

```
handlers=java.util.logging.ConsoleHandler.level=FINE

java.util.logging.ConsoleHandler.level=FINE
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
```

To track the class-loading activity, you can set

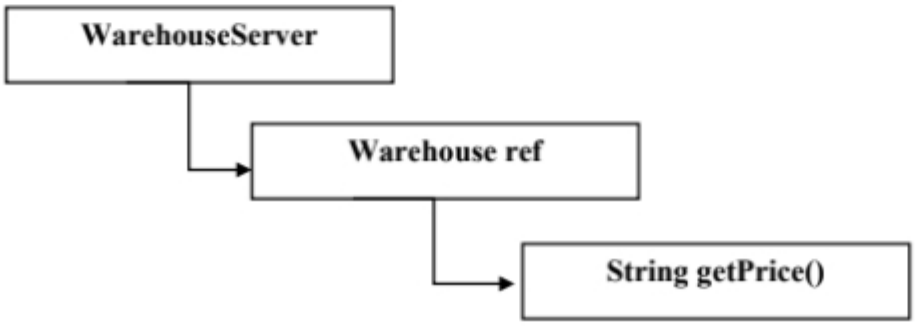
```
sun.rmi.loader.level=FINE
```

Table 11.1. RMI Loggers

Logger Name	Logged Activity
sun.rmi.server.call	Server-side remote calls
sun.rmi.server.ref	Server-side remote references
sun.rmi.client.call	Client-side remote calls
sun.rmi.client.ref	Client-side remote references
sun.rmi.dgc	Distributed garbage collection
sun.rmi.loader	RMIClassLoader
sun.rmi.transport.misc	Transport layer
sun.rmi.transport.tcp	TCP binding and connection
sun.rmi.transport.proxy	HTTP tunneling

5.4. Parameters and Return Values in Remote Methods

- At the start of a remote method invocation, the parameters need to be moved from the virtual machine of the client to the virtual machine of the server.
- After the invocation has completed, the return value needs to be transferred in the other direction.
- When a value is passed from one virtual machine to another, we will get two cases:
    1. passing remote objects
    2. passing nonremote objects.



5.4.1. Transferring Remote Objects

- It consist of a network address and a unique identifier for the remote object.
- It information is encapsulated in a stub object.
- It is method cal on remote reference is significantly slower and potentially less reliable than a method call on a local reference.

5.4.2. Transferring Nonremote Objects

There are two mechanisms for transferring values between virtual machines.

• Objects of classes that implement the Remote interface are transferred as remote references.

• Objects of classes that implement the Serializable interface but not the Remote interface are copied using serialization.

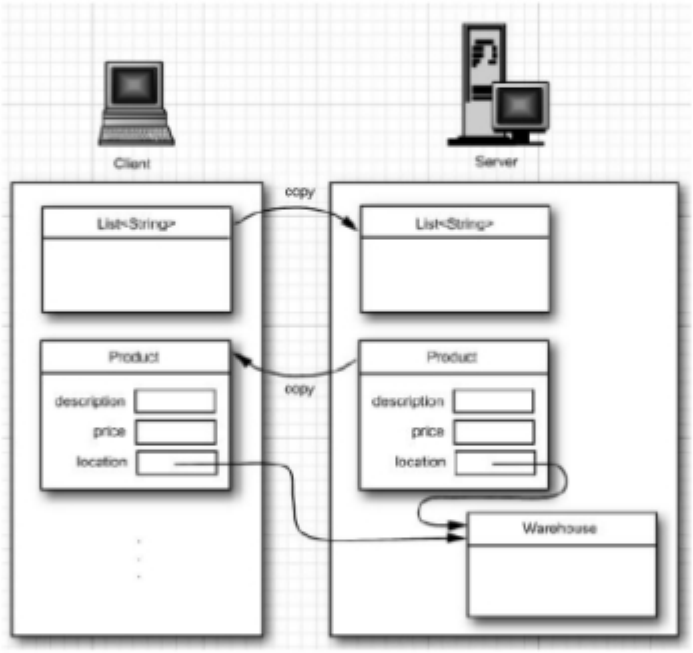


Figure .7. Copying local parameter and result objects

Listing 11.5. warehouse2/Warehouse.java

```
import java.util. * ;

 /**
 The remote interface for a simple warehouse.
  */
 public interface Warehouse extends Remote{
     double getPrice(String description)throws RemoteException;
     Product getProduct(List < String > keywords)throws RemoteException;
```

}

Referece:

https://www.javatpoint.com/RMI

http://www.ejbtutorial.com/java-rmi/new-easy-tutorial-for-java-rmi-using-eclipse