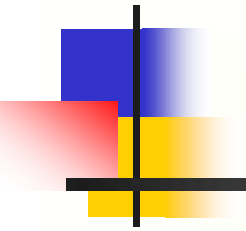


Advanced Programming

Computer Science Program

By Biruk

Chapter Seven

Remote Method Invocation

- Objective:
  - Understand the role of distributed objects in application development
  - Write Java programs that communicate through distributed object with remote objects

Introduction (cont'd)

- Before the mid-80s, computers were
  - very expensive (hundred of thousands or even millions of dollars)
  - very slow (a few thousand instructions per second)
  - not connected among themselves
- After the mid-80s: two major developments
  - cheap and powerful microprocessor-based computers appeared
  - computer networks
    - LANs at speeds ranging from 10 to 1000 Mbps
    - WANs at speed ranging from 64 Kbps to gigabits/sec

Introduction (cont'd)

- Consequence
  - feasibility of using a large network of computers to work for the same application; this is in contrast to the old centralized systems where there was a single computer with its peripherals.
- *Definition:*
  - *A distributed system is a collection of independent computers that appears to its users as a single coherent system.*
  - This definition has two aspects:
    1. *hardware: autonomous machines*
    2. *software: a single system view for the users*

Introduction (cont'd)

- Why Distributed?
  - Resource and Data Sharing
    - printers, databases, multimedia servers, ...
  - Availability, Reliability
    - the loss of some instances can be hidden
  - Scalability, Extensibility
    - the system grows with demand (e.g., extra servers)
  - Performance
    - huge power (CPU, memory, ...) available

RMI

- RMI is Java's distributed object solution.
- RMI allows us to leverage on the processing power of another computer. This is called *distributed computing.*
- RMI attempts to make communication over the network as transparent as possible.

RMI (cont'd)

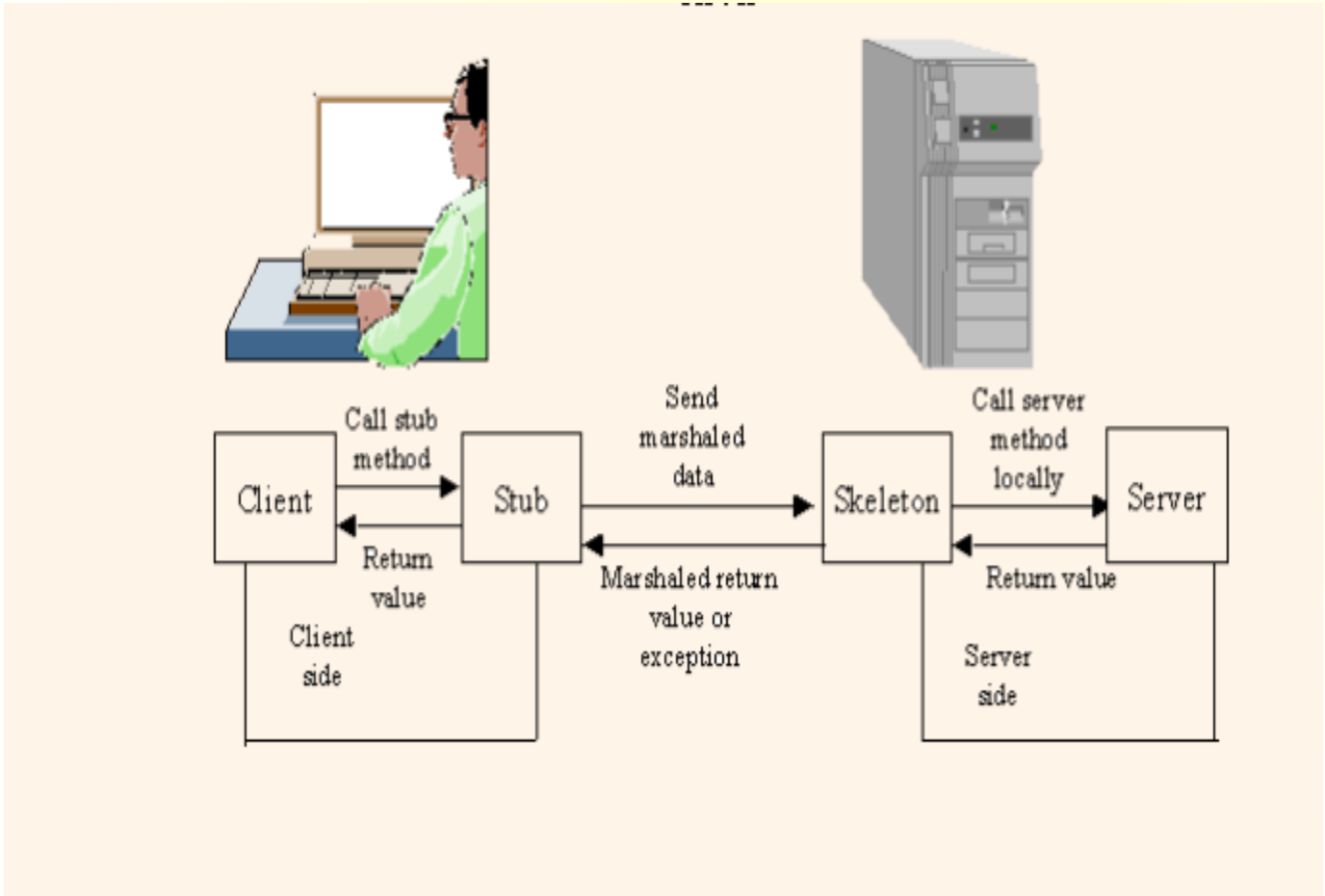
- An RMI application is often composed of two separate programs, a server and a client.
- The object whose method makes the remote call is called the **client object**. The remote object is called the **server object**.
- The computer running the Java code that calls the remote method is the client for that call, and the computer hosting the object that processes the call is the server for that call.

Stubs and Parameter Marshaling

- When client code wants to invoke a remote method on a remote object, it actually calls an ordinary method on a proxy object called a **stub**.
- The stub resides on the client machine, not on the server.
- A stub acts as a client local representative for the remote object.
- The client invokes a method on the local stub.
- 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**.
- Purpose: to convert the parameters into a format suitable for transport from one virtual machine to another.

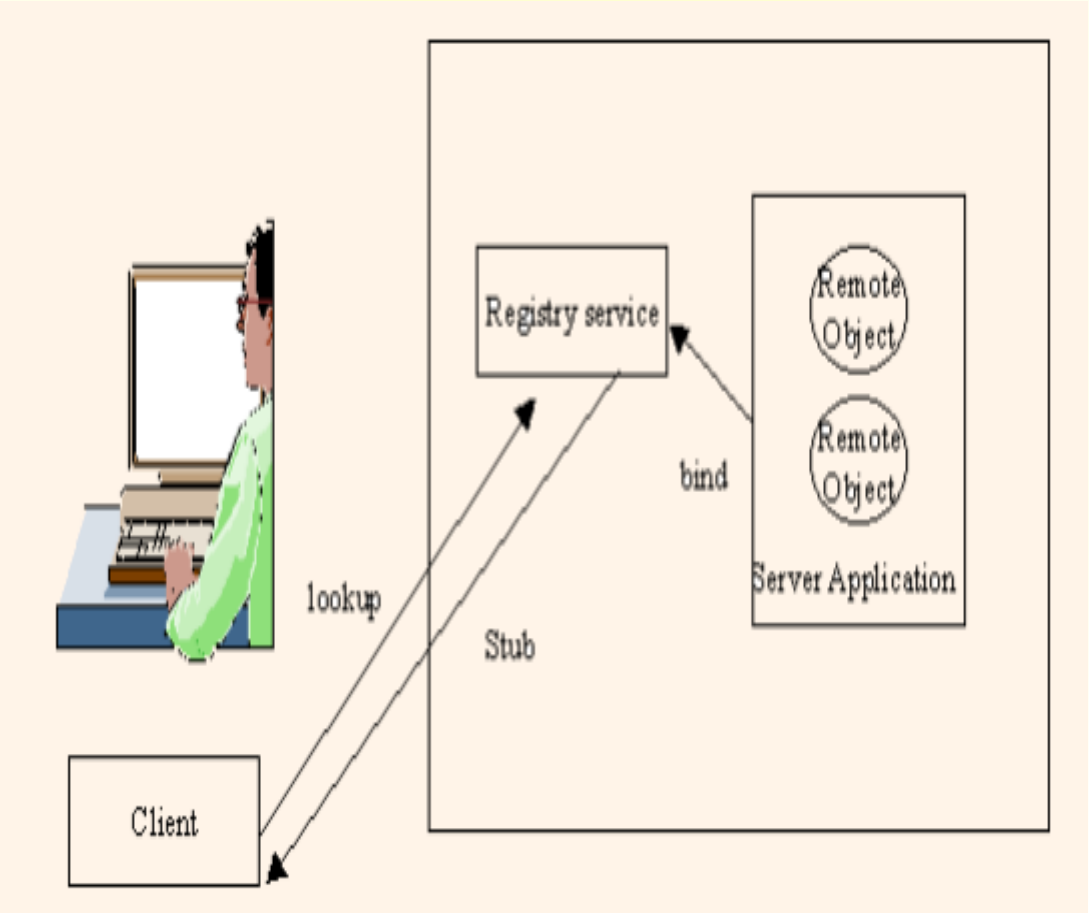
Stubs…

- The stub method on the client builds an information block that consists of:
  - An identifier of the remote object to be used;
  - A description of the method to be called; and
  - The marshalled parameters.
- The stub then sends this information to the server.
- On the server side, a receiver object performs the following actions for every remote method call:
  - It unmarshals the parameters.
  - It locates the object to be called.
  - It calls the desired method.
  - It captures and marshals the return value or exception of the call.
  - It sends a package consisting of the marshalled return data back to the stub on the client.


Client
Call stub method
Return value
Stub
Send marshaled data
Marshaled return value or exception
Skeleton
Call server method locally
Return value
Server
Client side
Server side

RMI Registry

- Is a simple name server provided for storing references to remote objects.
- The server is responsible for registering remote objects with the registry service.
- The registry keeps track of the addresses of remote objects.
- A remote objects handle can be stored using the methods of java.rmi.Naming class.
- A client can obtain a reference to a remote object by looking up in the server registry.


Registry service
Remote Object
Remote Object
Server Application
bind
lookup
Stub
Client

The Server Side

- To create a new remote object, first define an interface that extends the java.rmi.Remote interface.
- Purpose of Remote: to tag remote objects so that they can be identified as such.
- Your subinterface of Remote determines which methods of the remote object clients may call.
- A remote object may have many public methods, but only those declared in a remote interface can be invoked remotely.

```
//Hello Interface
import java.rmi.*;

public interface Hello extends Remote
{
   public String hello() throws RemoteException;
}
```

- **The next step** is to define a class that implements this remote interface.
- This class should extend java.rmi.server.UnicastRemoteObject.
  - public class HelloImpl extends UnicastRemoteObject implements Hello
- UnicastRemoteObject provides a number of methods that make remote method invocation work.
- In particular, it marshals and unmarshals remote references to the object.

```
//Hello Implementation
 import java.rmi.*;
import java .rmi.server.*;
 public class HelloImpl extends UnicastRemoteObject
   implements Hello
 {
  public HelloImpl() throws RemoteException
  {
  }

  public String hello() throws RemoteException
  {
    return "Hello World!";
  }
}
```

- **Next,** we need to write a server that makes the Hello remote object available to the world. All it has is a main( ) method.
- It constructs a new HelloImpl object and binds that object to the name “h" using the Naming class to talk to the local registry.
- A registry keeps track of the available objects on an RMI server and the names by which they can be requested.

```
//Hello Server
 import java.rmi.*;
import java.rmi.server.*;
 public class HelloServer
{
 public static void main(String args [])
 {
   try
   {
     Naming.bind("h",new HelloImpl());
     System.out.println("hello server is running");
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }
}
```

Client Side

- Before a regular Java object can call a method, it needs a reference to the object whose method it's going to call.
- Before a client object can call a remote method, it needs a remote reference to the object whose method it's going to call.
- A program retrieves this remote reference from a registry on the server where the remote object runs.
- It queries the registry by calling the registry's lookup( ) method.
- The exact naming scheme depends on the registry; the java.rmi.Naming class provides a URL-based scheme for locating objects.

```
//Hello Client
 import java.rmi.*;
 public class HelloClient
{
 public static void main(String args [])
 {
   try {
      Hello h1=(Hello)Naming.lookup("rmi://localhost/h");
      System.out.println(h1.hello());
   }
   catch(Exception e)
   {
     e.printStackTrace();
   }
 }
}
```

- How to run?
  I. Compile all using javac *.java
     I. Give path c:\jdk1.6.0_06\bin
  II. start rmiregistry
  III. start java HelloServer
  IV. java HelloClient

Reading Assignment

- Read the difference between socket and RMI.
- RMI is built on top of sockets, It translates method calls and return values and sends those through sockets.
- Socket programming - you have to handle exactly which sockets are being used, you specify TCP or UDP, you handle all the formatting of messages travelling between client and server. However, if you have an existing program that talks over sockets that you want to interface to, it doesn't matter what language it's written in, as long as message formats match.

- RMI - hides much of the network specific code, you don't have to worry about specific ports used (but you can if you want), RMI handles the formatting of messages between client and server. However, this option is really only for communication between Java programs. (You *could* interface Java RMI programs with programs written in other languages, but there are probably easier ways to go about it...)

- RMI: remote method invocation. Strictly JAVA stuff. outrside of java, known as RPC remote procedure call.
  Sockets is just a way to send data on a port to a different
  host, DATA not METHOD. it's up to you then to define your own protocol.

- You're talking apples and oranges here. Sockets deals with the low-level workings of establishing and maintaining connection between points in a network, as far as the nature of a Java program as one running inside a virtual machine allows.

- RMI on the other hand, is just an application of Sockets for transmitting messages between two Java programs according to a set of rules and conditions. What these rules and conditions do is they provide you with a structure. Such that you don't have to worry about the underlying workings of the connection and can focus on higher-level design requirements. Think of it like this, Without sockets RMI wouldn't exist. But RMI is just a small application of Sockets.