ADVANCED JAVA SWEG2033

Advanced Programming

Code: SWEG2033

Chapter Six

Networking in Java

For More read Chapter 24 :Java™ How to Program, Seventh Edition

Introduction

- A network is a collection of computers and other devices that can send data to and receive data from each other.
- What a network program do?
  - Retrieve data
  - Send data – Once a connection between two machines is established, Java programs can send data across the connection just as easily as they can receive from it.
  - Peer-to-peer interaction
    - Games
    - Chat
    - File sharing
  - Servers
  - Searching the web
  - E-commerce

Introduction

- When a computer needs to communicate with another computer, it needs to know the other computer's address.
- An Internet Protocol (IP) address uniquely identifies the computer on the Internet.
- TCP and UDP
  - TCP enables two hosts to **establish a connection** and exchange streams of data.
  - TCP **guarantees delivery** of data and also guarantees that packets will be **delivered in the same order** in which they were sent.
  - UDP is a connectionless protocol.

Client/Server Computing

- Networking is tightly integrated in Java.
- Java API provides the classes for creating sockets to facilitate program communications over the Internet.
- **Sockets** are the endpoints of logical connections between two hosts and can be used to send and receive data.
- Java treats socket communications much as it treats I/O operations; thus programs can read from or write to sockets as easily as they can read from or write to files.
- Network programming usually involves a server and one or more clients.

Client/Server Computing

- The client sends requests to the server, and the server responds.
- The server can accept or deny the connection.
- The client begins by attempting to establish a connection to the server.
- Once a connection is established, the client and the server communicate through sockets.
- The server must be running when a client attempts to connect to the server.
- The server waits for a connection request from a client.

The Server Socket

- To establish a server, you need to create a server socket and attach it to a port, which is where the server listens for connections.
- Port is a software address of a computer on the network.
- The port identifies the TCP service on the socket.
- A socket is a communication path to a port.
- To communicate program over the network, give a way of addressing the port. How? Create a socket and attach it to the port.
- Port numbers range from 0 to 65536, but port numbers 0 to 1024 are reserved for privileged services.
- For instance, the email server runs on port 25, and the Web server usually runs on port 80.

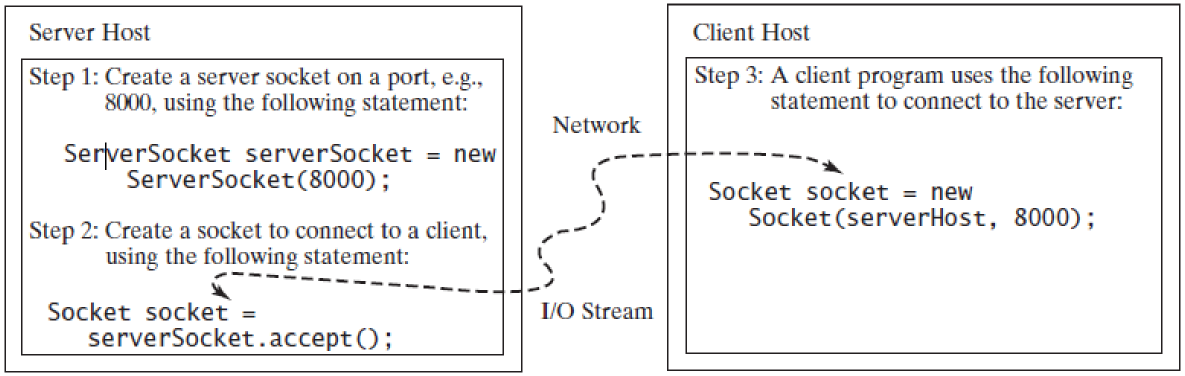
The Server Socket

- You can choose any port number that is not currently used by any other process.
- The following statement creates a server socket **serverSocket:**
  - ServerSocket serverSocket = **new ServerSocket(port);**
- After a server socket is created, the server can use the following statement to listen for connections:
  - Socket socket = serverSocket.accept();

The Client Socket

- The client issues the following statement to request a connection to a server:
  - Socket socket = new Socket(serverName, port);
    - ServerName is the server’s Internet host name or IP address.
- The following statement creates a socket at port 8000 on the client machine to connect to the host 130.254.204.36:
  - Socket socket = new Socket("130.254.204.36", 8000)

The Client Socket


Server Host
Step 1: Create a server socket on a port, e.g., 8000, using the following statement:
Step 2: Create a socket to connect to a client, using the following statement:
Network
I/O Stream
Client Host
Step 3: A client program uses the following statement to connect to the server:

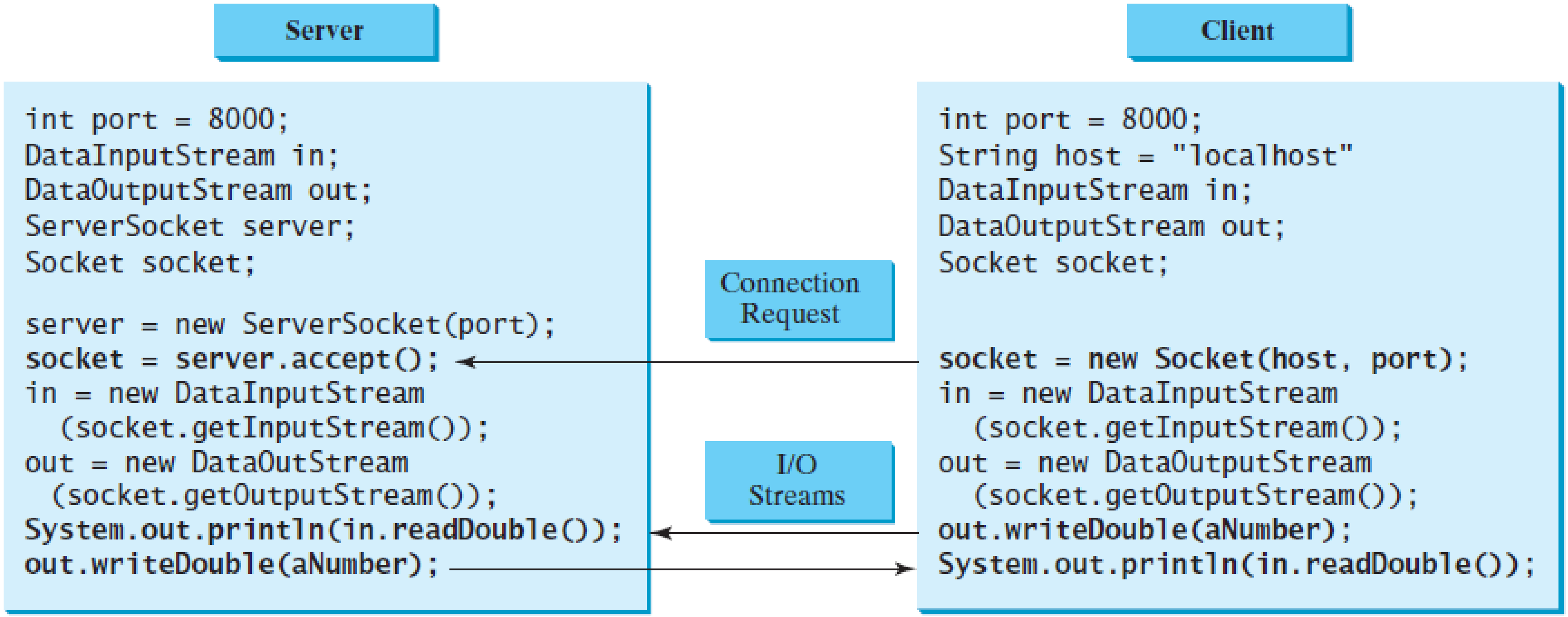
```
ServerSocket serverSocket = new
    ServerSocket(8000);
```

```
Socket socket =
   serverSocket.accept();
```

```
Socket socket = new
   Socket(serverHost, 8000);
```

The server creates a server socket and, once a connection to a client is established, connects to the client with a client socket.

Data Transmission Through Sockets



```
int port = 8000;
DataInputStream in;
DataOutputStream out;
ServerSocket server;
Socket socket;

server = new ServerSocket(port);
socket = server.accept();
in = new DataInputStream
  (socket.getInputStream());
out = new DataOutStream
  (socket.getOutputStream());
System.out.println(in.readDouble());
out.writeDouble(aNumber);
```

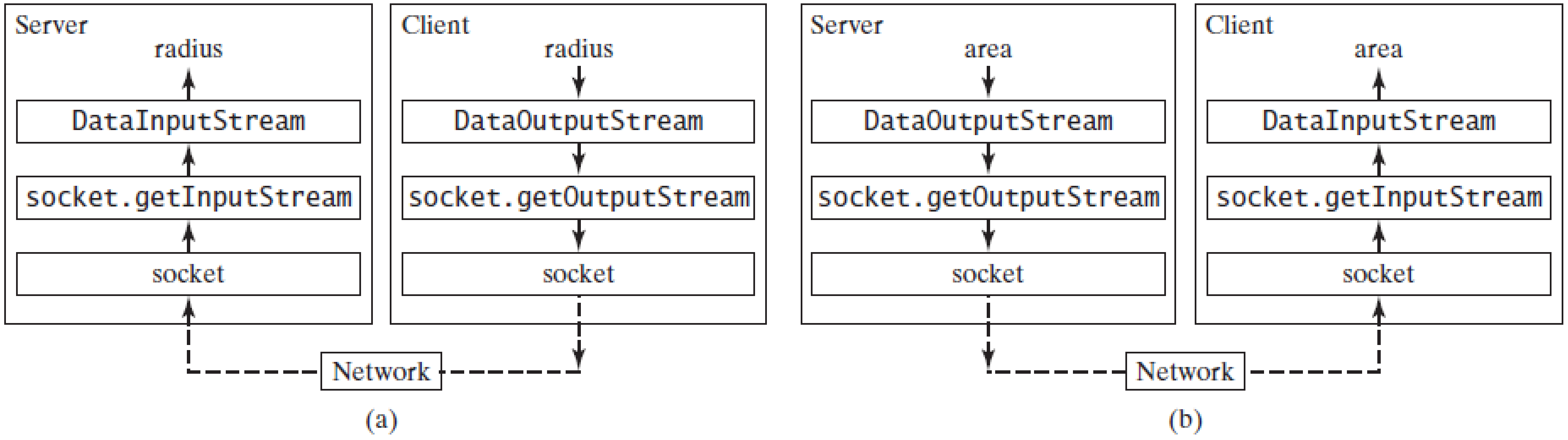
```
int port = 8000;
String host = "localhost"
DataInputStream in;
DataOutputStream out;
Socket socket;

socket = new Socket(host, port);
in = new DataInputStream
  (socket.getInputStream());
out = new DataOutputStream
  (socket.getOutputStream());
out.writeDouble(aNumber);
System.out.println(in.readDouble());
```

The server and client exchange data through I/O streams on top of the socket.

Data Transmission…

- Example: The client sends the radius to the server; the server computes the area and sends it to the client.



(a) The client sends the radius to the server.

(b) The server sends the area to the client.

Example: Sever

```
//Server.java
import java.io.*;
import java.net.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;
public class Server extends JFrame {
    // Text area for displaying contents
    private JTextArea jta = new JTextArea();
    public static void main(String[] args) {
        new Server();
    }
    public Server() {
        // Place text area on the frame
        setLayout(new BorderLayout());
        add(new JScrollPane(jta), BorderLayout.CENTER);
        setTitle("Server");
```

Example: Sever(cont’d)

```
        setSize(500, 300);
         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         setVisible(true); // It is necessary to show the frame here!
try {
            // Create a server socket
         ServerSocket serverSocket = new ServerSocket(8000);
         jta.append("Server started at " + new Date() + '\n');
         // Listen for a connection request
         Socket socket = serverSocket.accept();
         // Create data input and output streams
         DataInputStream inputFromClient = new DataInputStream(
          socket.getInputStream());
         DataOutputStream outputToClient = new DataOutputStream(
          socket.getOutputStream());
         while (true) {
         // Receive radius from the client
          double radius = inputFromClient.readDouble();
         // Compute area
         double area = radius * radius * Math.PI;
         // Send area back to the client
```

Example: Sever(cont’d)

```
          outputToClient.writeDouble(area);
           jta.append("Radius received from client: " + radius + '\n');
           jta.append("Area found: " + area + '\n');
       }
    }
       catch(IOException ex) {
                 System.err.println(ex);
    }
    }
   }
```

Example: Client

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Client extends JFrame {
 // Text field for receiving radius
 private JTextField jtf = new JTextField();
 // Text area to display contents
 private JTextArea jta = new JTextArea(); // IO streams
 private DataOutputStream toServer;
 private DataInputStream fromServer;
public static void main(String[] args) {
 new Client();
 }
```

Example: Client(cont'd)

```
public Client() {
// Panel p to hold the label and text field
 JPanel p = new JPanel();
 p.setLayout(new BorderLayout());
 p.add(new JLabel("Enter radius"), BorderLayout.WEST);
 p.add(jtf, BorderLayout.CENTER);
 jtf.setHorizontalAlignment(JTextField.LEFT);
 setLayout(new BorderLayout());
 add(p, BorderLayout.NORTH);
 add(new JScrollPane(jta), BorderLayout.CENTER);
 jtf.addActionListener(new TextFieldListener());
setTitle("Client");
 setSize(500, 300);
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 setVisible(true); // It is necessary to show the frame here!
```

Example: Client(cont'd)

```
try {
// Create a socket to connect to the server
Socket socket = new Socket("localhost",8000);
// Socket socket = new Socket("130.254.204.36", 8000);
// Socket socket = new Socket("drake.Armstrong.edu", 8000);
// Create an input stream to receive data from the server
fromServer = new DataInputStream(
  socket.getInputStream());
// Create an output stream to send data to the server
toServer =
 new DataOutputStream(socket.getOutputStream());
}
catch (IOException ex) {
jta.append(ex.toString() + '\n');
}
}
```

Example: Client(cont'd)

```
private class TextFieldListener implements ActionListener {
 public void actionPerformed(ActionEvent e) {
 try {
 // Get the radius from the text field
 double radius = Double.parseDouble(jtf.getText().trim());
 // Send the radius to the server
 toServer.writeDouble(radius);
 toServer.flush();
 // Get area from the server
 double area = fromServer.readDouble() ;
 // Display to the text area
 jta.append("Radius is " + radius + "\n");
 jta.append("Area received from the server is "
 + area + '\n');
 }
 catch (IOException ex) {
 System.err.println(ex);
 }
 }
 }
```

The InetAddress Class

- To know who is connecting to sever,. You can use the **InetAddress class.**
- InetAddress has three static methods
  - public static InetAddress getByName(String hostName)
  - public static InetAddress[] getAllByName(String hostName)
  - public static InetAddress getLocalHost( )

InetAddress (cont'd)

- Example - A program that prints the address of host

```
import java.net.*;
public class HostName {
  public static void main (String[] args) {
    try {
      InetAddress address = InetAddress.getByName("www.google.com");
      System.out.println(address);
    }
    catch (UnknownHostException ex) {
      System.out.println("Could not find www.google.com");

    }
 }
}
```

InetAddress (cont’d)

- Some computers have more than one Internet address.
- Given a hostname, InetAddress.getAllByName() returns an array that contains all the addresses corresponding to that name.
- Example

```
import java.net.*;
public class AllAddressOfGoogle {
    public static void main (String[] args) {
      try {
              InetAddress[] addresses =
              netAddress.getAllByName("www.google.com");
              for (int i = 0; i < addresses.length; i++) {
                   System.out.println(addresses[i]);
               }
         }
         catch (UnknownHostException ex) {
              System.out.println("Could not find www.google.com");
```

InetAddress (cont’d)

- Example

```
import java.net.*;
public class MyAddress {
 public static void main (String[] args) {
    try {
      InetAddress address = InetAddress.getLocalHost( );
      System.out.println(address);
    }
    catch (UnknownHostException ex) {
      System.out.println("Could not find this computer's address.");

    }
  }

}
```

Serving Multiple Clients

- Multiple clients are quite often connected to a single server at the same time.
- Typically, a server runs continuously on a server computer, and clients from all over the Internet can connect to it.
- You can use threads to handle the server's multiple clients simultaneously.
- Simply create a thread for each connection.
- Here is how the server handles the establishment of a connection:

```
while (true) {
    Socket socket = serverSocket.accept(); // Connect to a client
    Thread thread = new ThreadClass(socket);
    thread.start();
}
```

- The server socket can have many connections.
- Each iteration of the **while loop creates a new** connection.
- Whenever a connection is established, a new thread is created to handle communication between the server and the new client; and this allows multiple connections to run at the same time.

ADVANCED JAVA SWEG2033

Advanced Programming

Code: SWEG2033

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

- 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

- 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

- 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 (Remote Method Invocation)

- 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

- 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

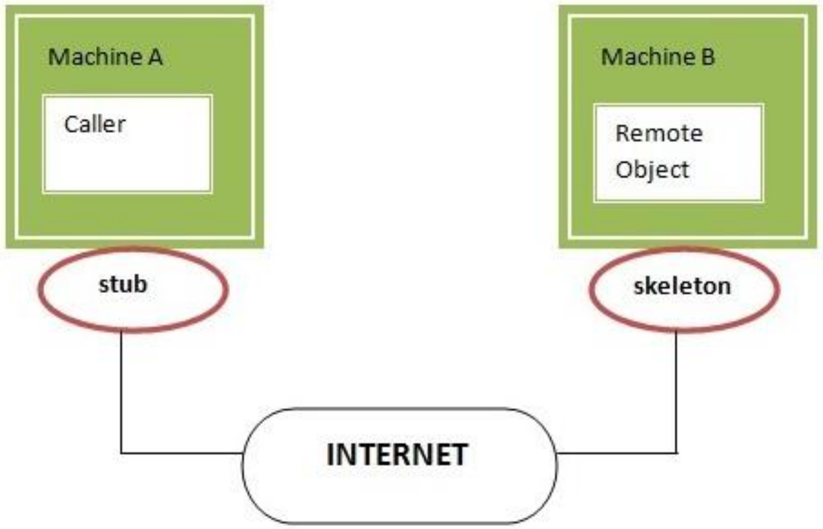
- RMI uses stub and skeleton object for communication with the remote object.
- A **remote object** is an object whose method can be invoked from another JVM
- Stub : The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

Stubs and Parameter Marshaling

Skeleton : The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.



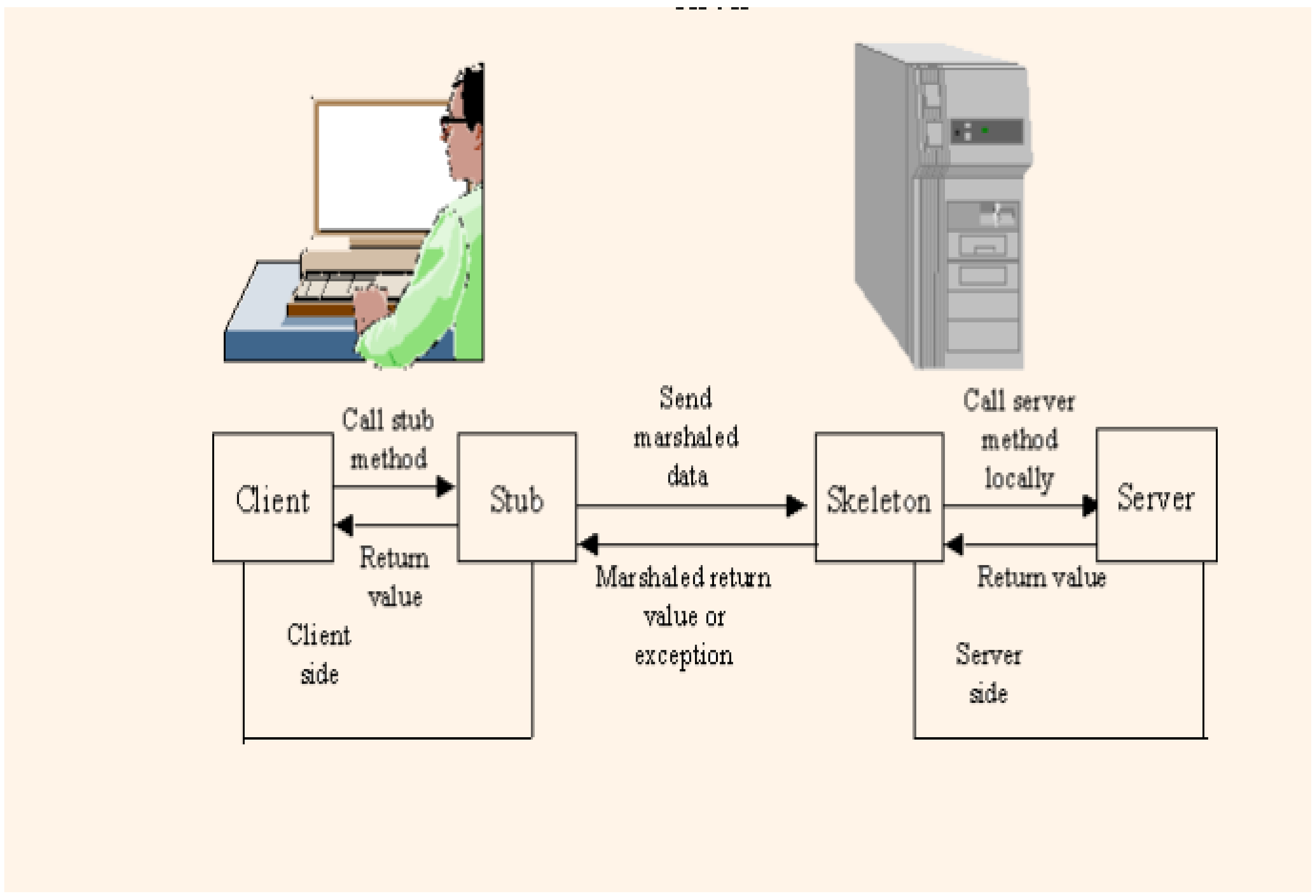
Stubs and Parameter Marshaling

- 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.

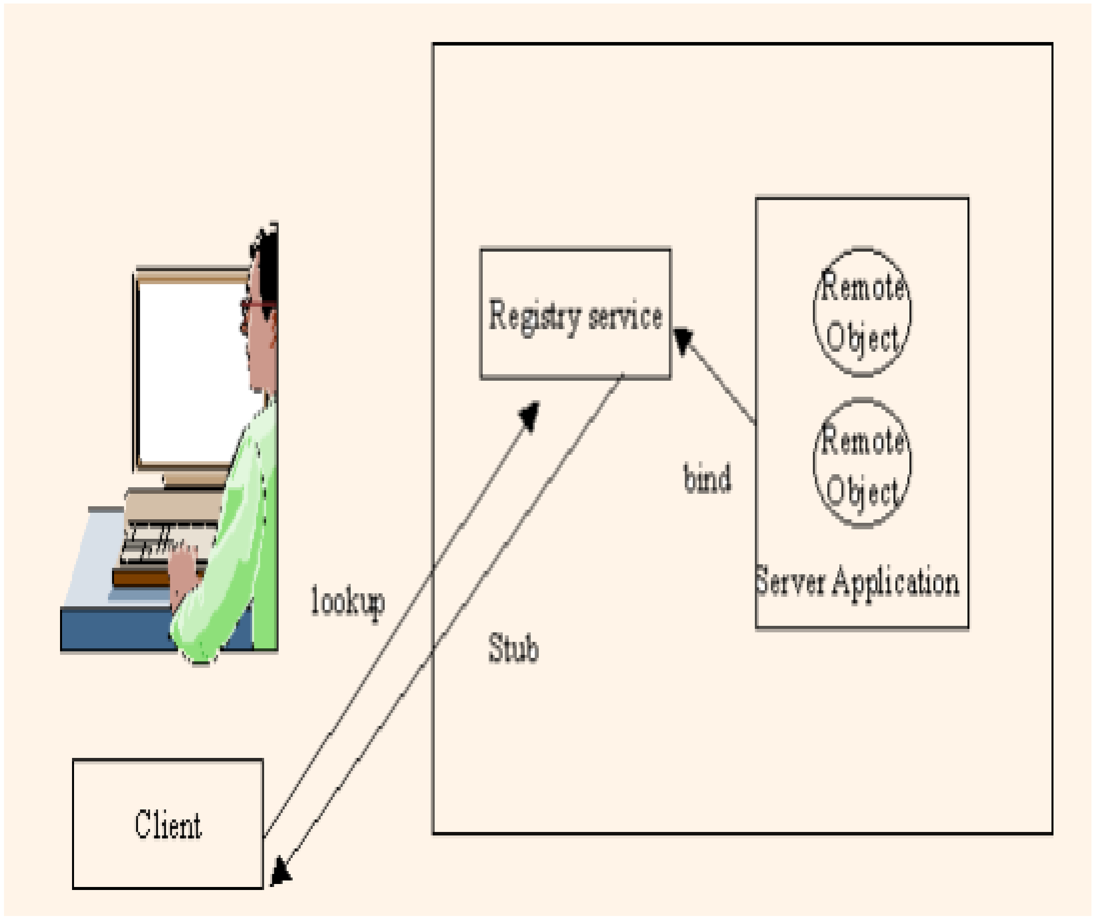
Stubs…


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

RMI Registry



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 sub-interface 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

For more information

https://www.javatpoint.com/RMI

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.