Chapter 4

Java Socket based Distributed Computing

Client-Server Computing: Java provides two mechanisms for distributed computing: **Socket-based communication** and **Remote method invocation/RMI**. Socket-based communication is supported by the APIs in the package java.net while RMI communication is supported by the APIs in the package java.rmi.

Socket-based communication, in turn, could employ stream-based (using TCP) or packet-based (using UDP) communications. Stream-based communications are used in most areas of java programming and, thus, is the focus of this chapter.

Network programming usually involves a server and one or more clients. The client sends requests to the server, and the server responds. The client begins by attempting to establish a connection to the server. The server can accept or deny the connection. Once a connection is established, the client and the server communicate through sockets.

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. You should note that the server must be running when a client attempts to connect to the server. The server waits for a connection request from a client.

Socket-based Communication: There are two types of sockets: Server socket and Client socket. The server socket waits for requests from clients while client socket can be used to send and receive data. A server socket listens at a specific port. A port is a positive integer less than or equal to 65565. The port number is necessary to distinguish different server applications running on the same host. Ports 1 through 1023 are reserved for administrative purposes (e.g. 21 for FTP, 23 for Telnet, 80 for HTTP).

Server Sockets: 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. A server socket is an instance of the **ServerSocket** class and can be created by one of these constructors:

ServerSocket(int port)

ServerSocket (int port, int backlog)

where port is a port number at which the server will be listening for requests from clients and backlog is the maximum length of the queue of clients waiting to be processed (default is 50).

The following statement creates a server socket serverSocket:

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

Attempting to create a server socket on a port already in use would cause the **java.net.BindException**

Methods of Server Socket: Server sockets have two methods. These are Socket accept() and void close(). Socket accept waits for a connection request from clients. The thread that executes this method will be blocked until a request is received, at which time the method returns a client socket. Void close() instead stop the server from waiting for requests from clients.

Client Sockets: A client socket is an instance of the Socket class and can be obtained in two ways. The first way is on the server side as return value of the accept method(This is a socket to listen for connections of clients).

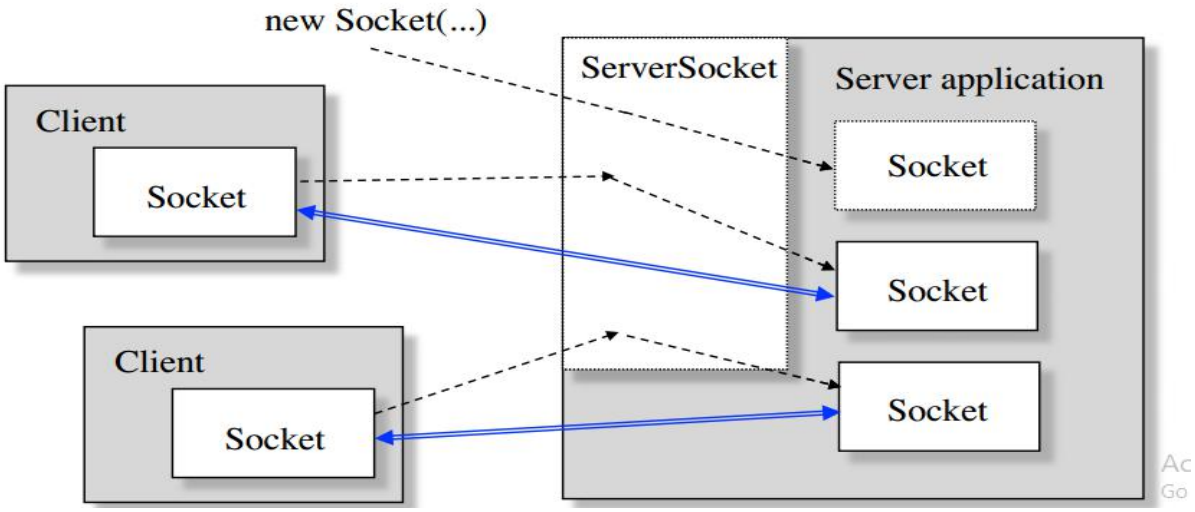
- Socket socket = serverSocket.accept();

This statement waits until a client connects to the server socket. The second way in which a client socket instance can be obtained is on the client side by using the Socket constructor. The client issues the statement **Socket (String host, int port)** to request a connection to a server. In this statement host refers the address of the host machine where the service is running and port is the port number on which the server is listening for clients' requests.

The statement Socket (String host, int port) in the above paragraph opens a socket so that the client program can communicate with the server. A program can use the host name **localhost** or the IP address **127.0.0.1** to refer to the machine on which a client is running. The socket constructor throws a j**ava.net.UnknownHostException** if the host cannot be found.

Methods of client socket: Client sockets have three important methods: getInputStream(), getOutputStream() and close(). The method getInputStream() returns an InputStream object for receiving data while the method getOutputStream returns an OutputStream object for sending data. Close, in fact, closes the socket connection.

Clients' communication with a server: The following diagram illustrates how a client starts communication with server and how the actual communication between clients and a server continues.



Creating Clients and Server: The **java.net** API contain ServerSocket and Socket classes using which servers and clients could be created. ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients. Server socket has two methods that we have seen above. A server socket need to be attached to a port, which is where the server listens for connections (from clients). The port identifies the TCP service on the socket. Example server socket:**ServerSocket** ss = new **ServerSocket(8080**). In this specific example, the server listens to clients on port 8080.

Once a server socket is set, a socket to connect to a client can be created using the statement

Socket socket = SS.accept();

The class Socket is used to issue a request a connection to a server. The server host and the port number in which the server listens to clients is used to create the client socket. Example client socket: **Socket socket = new Socket (Server Host, 8000);** Note that Server Host could be an IP address or domain name of the server.

While the server waits for client requests, it will get stream of the request using input stream objects. For example, let us see example how the server get streams of data from a client that send radius values.

Example 1: InputStream input = socket.getInputStream();

System.out.println("A client sent me:"+input.readDouble());

Example 2: DataInputStream input = new DataInputStream(socket.getInputStream()); System.out.println("A client sent me:"+ input.readInt());

The server will send stream of responses using output streams once it received client requests and processed the request.

Example 1: Assume the server computed area of a circle of which radius was received

OutputStream out = socket.getOutputStream();

Out.writeDouble(area_calculated);

Example 2:

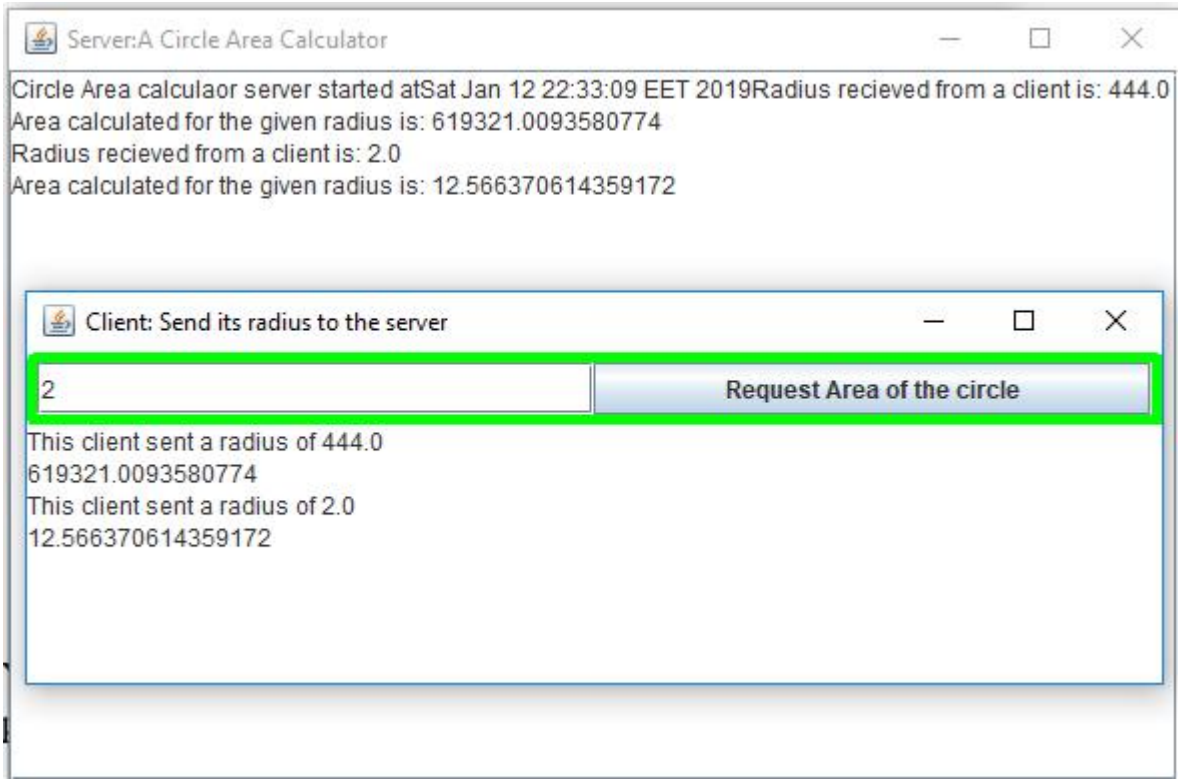
DataOutputStream out = new DataOutputStream (socket.getOutputStream());

Out.writeDouble(area_calculated);

Example Application: Assume that we have a server application that computes area of a circle using radius values from client applications. We will have one server window that has a text field to display the values of radius received from clients and to show area values of all client requests. A client can send different radius values and the server will compute the area and return the calculated values to the client. The client is supposed to enter the radius in the text

field and click the button “Request Area of the Circle” to ask the server for the computation of the area.

Here is the expected GUI of the application to be developed



You can read the full source code (one file for the client and one file for the server)of the application for this exercise in the next pages.

Server_Area_Calculator.java

```
import java.awt.BorderLayout;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class Server_Area_Calculator extends JFrame{
    /*declare the attributes of the class here*/
        private JTextArea info_display_area;
        private DataInputStream input_from_client;
```

```
    private DataOutputStream output_to_client;
    private ServerSocket server_socket;
    private Socket socket_data_exchange;
    Server_Area_Calculator(){
        info_display_area = new JTextArea();
        info_display_area.setEditable(false);
        this.setLayout(new BorderLayout());
        this.add(new JScrollPane(info_display_area),BorderLayout.CENTER);
        this.setTitle("Server:A Circle Area Calculator");
        this.setSize(600, 500);
        this.setLocation(80, 80);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);

        try {
            server_socket = new ServerSocket(8000);
            socket_data_exchange = server_socket.accept();
            info_display_area .append("server started at"+ new Date());
            input_from_client = new DataInputStream(socket_data_exchange.getInputStream());
            output_to_client = new DataOutputStream(socket_data_exchange.getOutputStream());
    while (true){
        double radius_from_client = input_from_client.readDouble();
        info_display_area.append("Radius recieved from a client is: "+radius_from_client+'\n');
         double area_of_circle = Math.PI * radius_from_client*radius_from_client;
          info_display_area.append("Area calculated for the given radius is: "+area_of_circle+'\n');
              output_to_client.writeDouble(area_of_circle);
           }

        } catch (IOException ex) {
            System.out.println("Error on the port on which the server is set to listen for clients: "+ex);
        }
    }
    public static void main (String [] args){
            new Server_Area_Calculator();
    }
}
```

Radius_Sender_Client.java

```
import java.awt.*;
import java.io.*;
import java.net.Socket;
import javax.swing.*;
import javax.swing.event.*;

public class Radius_Sender_Client extends JFrame{
   JLabel label_info;
   static JTextField txt_field_radius;
   static JButton request_area_btn ;
   static JTextArea txt_area_info_display;
  static  DataInputStream input_from_server;
   static DataOutputStream output_to_server;
```

```
   static Socket socket_exchange;

       Radius_Sender_Client(){
          label_info = new JLabel("Please insert radius. Only double valuesare accepted. Do not
include characters");
          txt_field_radius = new JTextField();
          txt_field_radius.getDocument().addDocumentListener(new
Check_appropriate_input_radius());
          request_area_btn = new JButton("Request Area of the circle");
          request_area_btn .setEnabled(false);
          request_area_btn.setCursor(new Cursor(Cursor.HAND_CURSOR));
           request_area_btn .addActionListener(new Talk_to_server_listener());
          JPanel p1 = new JPanel();
          p1.setLayout(new GridLayout(1,2));
          p1.add(txt_field_radius);
          p1.add(request_area_btn );
          p1.setBorder(BorderFactory.createLineBorder(Color.green, 5, true));
          txt_area_info_display = new JTextArea();
          txt_area_info_display.setEditable(false);

          JPanel p2 = new JPanel();
          p2.setLayout( new GridLayout(2,1));

          p2.add(label_info);
          p2.add(p1);
          this.setLayout(new BorderLayout());
          this.add(p1,BorderLayout.NORTH);
          this.add(txt_area_info_display,BorderLayout.CENTER);

       try {
          socket_exchange = new Socket ("localhost",8000);
          input_from_server = new DataInputStream(socket_exchange.getInputStream());
          output_to_server = new DataOutputStream(socket_exchange.getOutputStream());

       } catch (IOException ex) {
          Logger.getLogger(Radius_Sender_Client.class.getName()).log(Level.SEVERE, null, ex);
       }
          this.setTitle("Client: Send its radius to the server");
         this.setSize(600, 400);
         this.setLocation(100,180);
         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          this.setVisible(true);
       }

       public static void main(String [] args){
       new Radius_Sender_Client();
```

```
        }
}
class Talk_to_server_listener implements ActionListener {
   @Override
   public void actionPerformed(ActionEvent e) {
      double radius_entered = Double.parseDouble(Radius_Sender_Client.txt_field_radius.getText().trim());
      Radius_Sender_Client.txt_area_info_display.append("This client sent a radius of "+ radius_entered +'\n');
      try {
         Radius_Sender_Client.input_from_server = new DataInputStream (Radius_Sender_Client.socket_exchange.getInputStream());
         Radius_Sender_Client.output_to_server = new DataOutputStream(Radius_Sender_Client.socket_exchange.getOutputStream());

         Radius_Sender_Client.output_to_server.writeDouble(radius_entered);
         Radius_Sender_Client.output_to_server.flush();


Radius_Sender_Client.txt_area_info_display.append(Radius_Sender_Client.input_from_server.readDouble()+ "\n");
      } catch (IOException ex) {
         System.out.println("Error of stream: " + ex);
      }
     }
}

class Check_appropriate_input_radius implements DocumentListener {

   @Override
   public void insertUpdate(DocumentEvent e) {

      String i = Radius_Sender_Client.txt_field_radius.getText().trim();
      if (i.length()>0 ){
                Radius_Sender_Client.request_area_btn.setEnabled(true);
      }
   }

   @Override
   public void removeUpdate(DocumentEvent e) {
   }

   @Override
   public void changedUpdate(DocumentEvent e) {
   }
```

}

Exercise 1: This application works only for one client (where a client but can send radiuses repeatedly). Improve the code so that many clients can send their respective radius concurrently.

Exercise 2: Let you develop a group chat application in which messages of every participant of the group chat is broadcasted to every other participant of the group chat.