1. INTRODUCTION:

1.1 Introduction:

Remote Desktop is an application that serves users of it to capture other systems in order to access the services of those systems. As with advanced technology it is being common that in big organization a server or client system may be at remote place and they may be need to access the services of the Remote System. In this project we can control the desktop of all systems that are present under the control of our network. In this project if any desktop is under control then the user can do many things? Here if the user opens window then in the connected system(Client) window will be opened and if the user opens the notepad then the notepad will be opened in the connected system then if the user types any message in the opened notepad the same will be visible on the connected system notepad. The user can shut down the connected system if the software was properly installed in the network. Using Remote Frame Buffer (RFB) protocol and Virtual Network Computing (VNC) technique the end user is able to capture as well as perform various operations on the desktop to act as locally on the server desktop.

This project is to develop software, which allows accessing the remote system desktop with that it enables the user to control the remote system. The actual thing we are doing here to implement this task is to access the frame buffer of the remote system with this we are able to get the control of the remote system desktop and able to refresh the remote system frame buffer according to actions on the picture.

1.2 Architecture of Remote Desktop

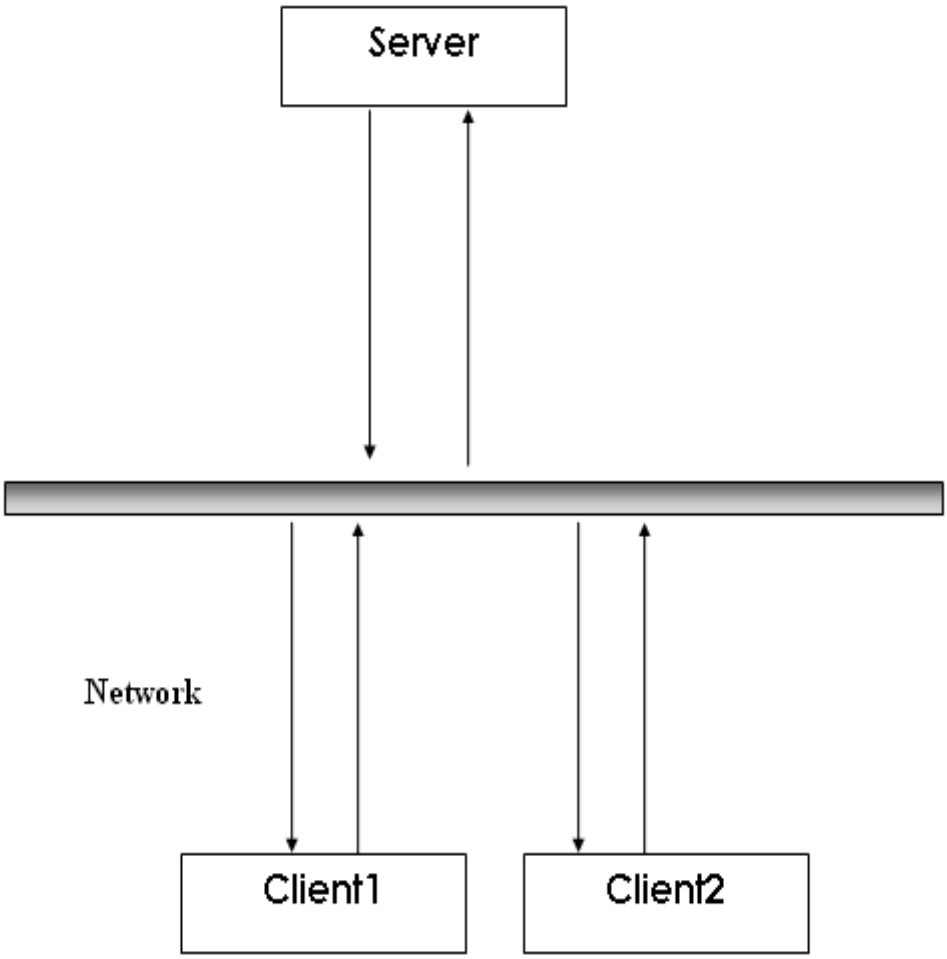

Figure 1.2 Architecture of Remote Desktop

1.3 Architecture of Remote Desktop Server

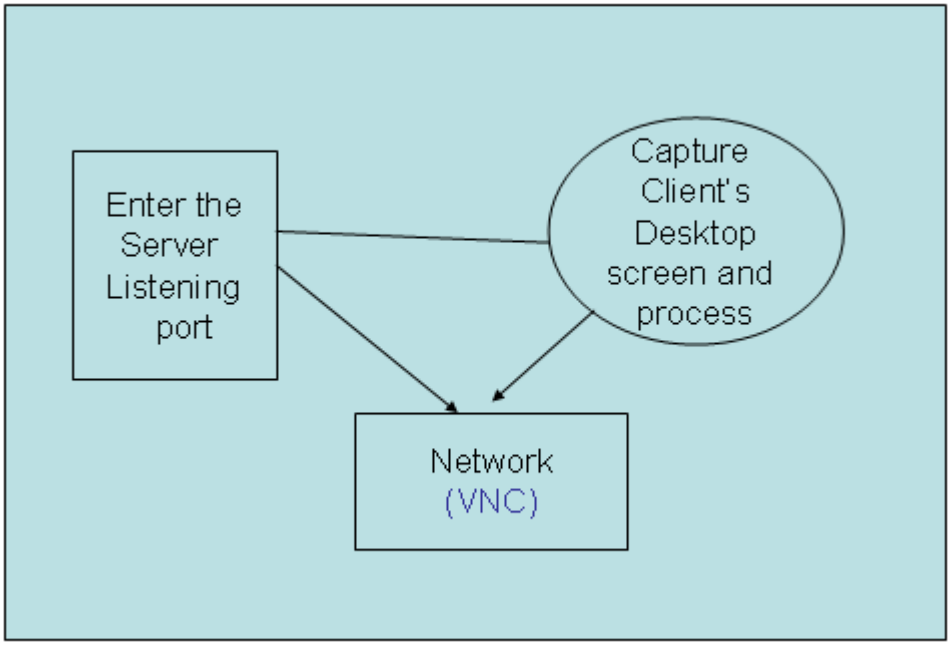

Figure 1.3 Architecture of Remote Desktop Server

The server will enter the listening port through which the clients can connect to server.

The server consists of RFB protocol for buffering the desktop of clients after connection.

When the client enters the server IP and port given by server then the client will connect ie the server will accept the client to connect. The live screen ie the desktop of the client will appear in the server desktop. The server can have full control over the client's desktop like save, delete, copy, shut down etc on the client's desktop. All these controls can be managed with the Mouse control program at server side.

1.4 Architecture of Remote Desktop client

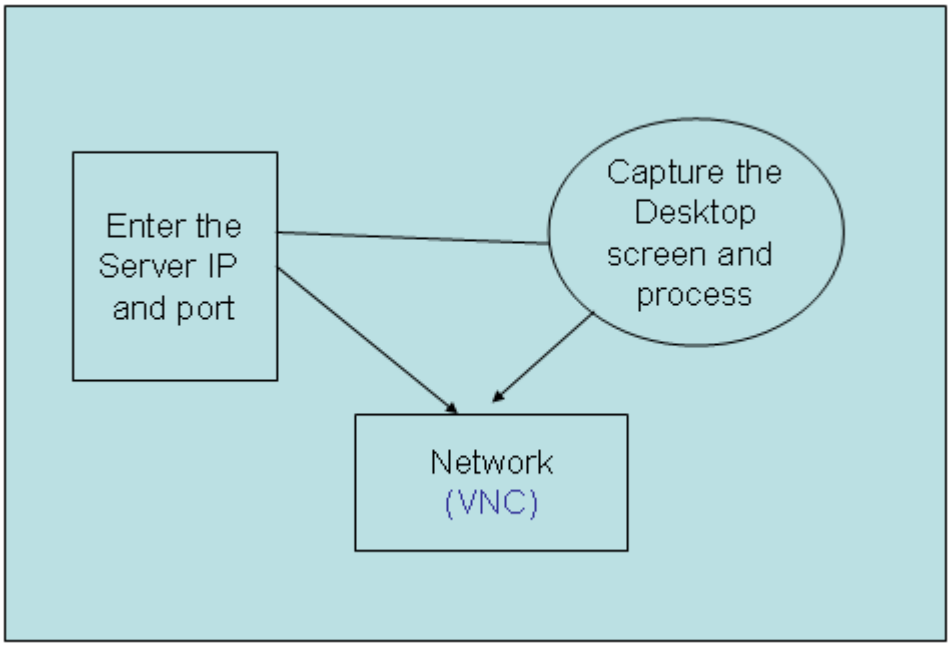

Figure 1.4 Architecture of Remote Desktop client

The client will enter server IP and the listening port through which the clients can connect to server. The client consists of RFB protocol for buffering the desktop of clients after connection. When the client enters the server IP and port given by server then the client will connect ie the server will accept the client to connect. The live screen ie the desktop of the client will appear in the server desktop. The client can observe full control over the client's desktop like save, delete, copy, shut down etc on client system by the server. All these controls can be managed with the Mouse control program at client side.

The client can reject/close the connection.

1.4 Scope and Objective:

The main aim of this project is to control and monitor the clients desktop remotely. Accessing more than two clients desktop simultaneously, providing complete control over the client's desktop.

2. SYSTEM STUDY AND ANALYSIS

EXISTING AND PROPOSED SYSTEMS:

2.1 Existing System:

In the existing system, accessing the Client's desktop and controlling the client's desktop is managed well but one of the draw back of the existing system is Accessing more than two clients. In the existing system accessing and controlling more than two clients is not possible. It won't be flexible well applied to many clients.

2.2 Proposed System:

The proposed system is flexible and simple. In the proposed system accessing more than two clients desktop and controlling is possible. Using RFB protocol and Virtual Network Computing accessing and monitoring of clients desktop made easy

2.3 REQUIREMENTS:

The following hardware and software is required.

2.3.1 Hardware Requirements

COMPONENTS	REQUIREMENTS
Hard Disk	At least 20GB
RAM	At least 512 MB

2.3.2 Software Requirements

- ✓ Windows 98/XP or later version
- ✓ J2SE 1.6
- ✓ J2SE 1.5(SWINGS)

2.4 MODULES

2.4.1Server:

This module contains all the connection, screen receiver, Mouse controls and keyboard control script for getting connected with the Client's system. Screen receiver program will receive the Client's desktop screen simultaneously using RFB.

Mouse Control program will provide the accessing control over the clients desktop. Keyboard control program will allow the server to control the client's desktop through the keyboard controls. Each time Server will enter the listening port for connection with the clients.

2.4.2 Client:

This module contains all the connection, screen Capturing, Mouse controls and keyboard control script for getting connected with the Server system. Screen capturing program will capture the Client's desktop screen simultaneously using RFB.

Mouse Control program will provide the accessing control over the clients desktop. Keyboard control program will allow the server to control the client's desktop through the keyboard controls. Client will enter the Server IP and listening port for connection with the server system

2.5 SYSTEM ANALYSIS

2.5.1 RFB:

RFB ("*remote frame buffer*") is a simple protocol for remote access to graphical user interfaces. Because it works at the frame buffer level it is applicable to all windowing systems and applications. RFB is protocol used in VNC (Virtual Network Computing).

The remote endpoint where the user sits (i.e. the display plus keyboard and/or pointer)is called the RFB client or viewer. The endpoint where changes to the frame buffer originate (i.e. the windowing system and applications) is known as the RFB server

RFB is truly a "thin client" protocol. The emphasis in the design of the RFB protocol is to make very few requirements of the client. In this way, clients can run on the widest range of hardware, and the task of implementing a client is made as simple aspossible.

The protocol also makes the client stateless. If a client disconnects from a given server and subsequently reconnects to that same server, the state of the user interface is preserved.Furthermore, a different client endpoint can be used to connect to the same RFB server. At the new endpoint, the user will see exactly the same graphical user interface as at the original endpoint. In effect, the interface to the user's applications becomes completely mobile. Wherever suitable network connectivity exists, the user can access their own personal applications, and the state of these applications is preserved between accesses from different locations. This provides the user with a familiar, uniform view of the computing infrastructure wherever they go.

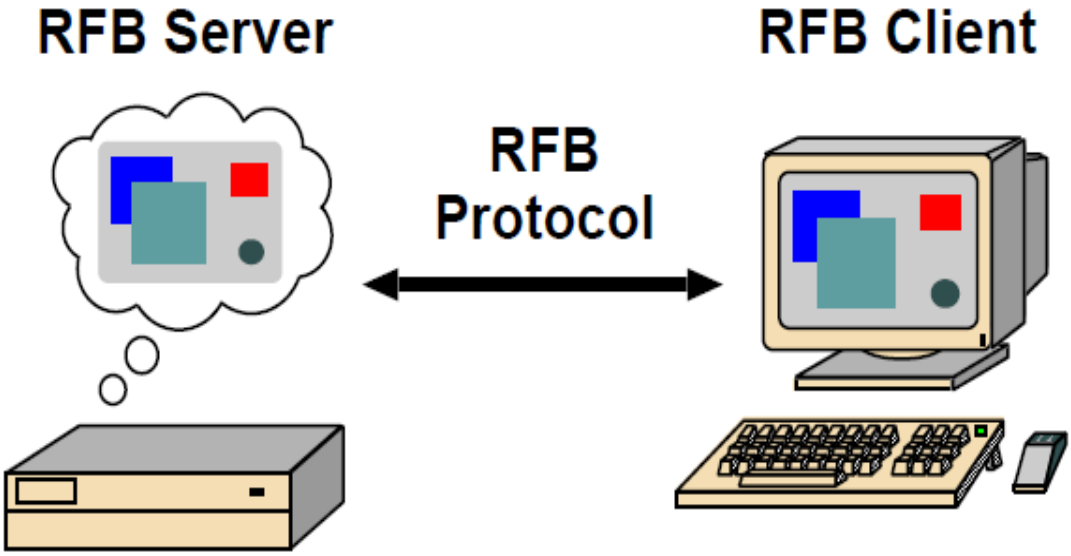

Figure 2.5.1 RFB

2.5.2Display Protocol

The display side of the protocol is based around a single graphics primitive: ***"put a rectangle of pixel data at a given x, y position".*** At first glance this might seem an inefficient way of drawing many user interface components. However, allowing various different encodings for the pixel data gives us a large degree of flexibility in how to trade off various parameters such as network bandwidth, client drawing speed and server processing speed.A sequence of these rectangles makes a *frame buffer update* (or simply *update*). An update represents a change from one valid frame buffer state to another, so in some ways is similar to a frame of video. The rectangles in an update are usually dis jointbut this is not necessarily the case.The update protocol is demand-driven by the client. That is, an update is only sent from the server to the client in response to an explicit request from the client. This gives the protocol an adaptive quality. The slower the client and the network are, the lower the rate of updates becomes. With typical applications, changes to the same area of the frame buffer tend to happen soon after one another. With a slow client and/or network, transient states of the frame buffer can be ignored, resulting in less network traffic and less drawing for the client.

2.5.3 Input Protocol

The input side of the protocol is based on a standard workstation model of a keyboard and multi-button pointing device. Input events are simply sent to the server by the client whenever the user presses a key or pointer button, or whenever the pointing device is moved. These input events can also be synthesized from other non-standard I/O devices. For example, a pen-based handwriting recognition engine might generate Keyboard events.

2.5.4 Representation of pixel data

Initial interaction between the RFB client and server involves a negotiation of the *format* and *encoding* with which pixel data will be sent. This negotiation has been designed to make the job of the client as easy as possible. The bottom line is that the server must always be able to supply pixel data in the form the client wants. However if the client is able to cope equally with several different formats or encodings, it may choose one which is easier for the server to produce.

Pixel *format* refers to the representation of individual colors by pixel values. The most common pixel formats are 24-bit or 16-bit “true color”, where bit-fields with in the pixel value translate directly to red, green and blue intensities, and 8-bit “colors map” where an arbitrary mapping can be used to translate from pixel values to the RGB intensities.Encodingrefers to how a rectangle of pixel data will be sent on the wire. Every rectangle of pixel data is prefixed by a header giving the X,Y position of the rectangle on the screen, the width and height of the rectangle, and an *encoding type* which specifies the encoding of the pixel data. The data itself then follows using the specified encoding.The encoding types defined at present are *Raw*, *CopyRect*, *RRE*, *Hextile*and *ZRLE*. Inpractice we normally use only the *ZRLE*, *Hextile*and *CopyRect*encodings since theythe best compression for typical desktops.

2.6 Virtual Network computing:

Virtual network computing is a remote display system that allows you to view a computer's desktop display from different locations. You can view the display not only on the computer it is running from, but also via Internet access and from a variety of different machines in different locations. Virtual network computing is a small and simple to use system that can be integrated into your own computer without the need for any installation.

Virtual network computing is a truly independent system. You can leave your desk and go to another computer, which can either be in the next room in your office or even in your home, and then reconnect to your original computer and finish your work. Virtual network computing is also shareable, which means that several people can view the desktop on their machines. It is excellent for systems testers who need to keep an eye on programs and operations as they are running on different platforms.

The protocol for virtual network computing is a very simple remote access to graphic user interface. The protocol allows the server to update from a remote frame buffer to the frame buffer on your display screen. Because the frame buffer is already in most computer display screens, many different types of computer are able to use this technology. Virtual network computing can work on Windows, Macintosh and X/UNIX The protocol used in virtual network computing has been designed so that it is very simple to use. There are virtually no requirements from the user. You can run a wide range of hardware with very little implementation.

To start with, the server from whom the information is being accessed will request authentication from the client, usually in the form of a password access code. The server and client then exchange information such as screen size, encoding schemes and pixel format. The client will then ask for an update for the whole screen and the session will begin. The session can be closed at anytime by the client without any adverse effects to the server.

3. SYSTEM DESIGN

Systems design is the process or art of defining the hardware and software architecture, components, modules, interfaces, and data for a computer system to satisfy specified requirements. One could see it as the application of systems theory to computing. Some overlap with the discipline of systems analysis appears inevitable.

Prior to the standardization of hardware and software in the 1990s which resulted in the ability to build modular systems, systems design had a more crucial and respected role in the data processing industry. The increasing importance of software running on generic platforms has enhanced the discipline of software engineering at the expense of systems design.

Object-oriented analysis and design methods are becoming the most widely used methods for system design. The UML has become the standard language used in Object-oriented analysis and design. It is widely used for modeling software systems and is increasingly used for designing non-software systems and organizations.

System designers are in between business analysts and technical experts. Generally System Design includes the following types:

3.1 LOGICAL DESIGN:

Logical design describes the format of inputs, outputs, and procedures that meets the user requirements.

The design covers the following:

- Reviews the current physical system.
- Prepares the output specification.
- Prepares the Input specifications.
- Prepares controls specifications.

3.2 PHYSICAL DESIGN:

This produces the working system by defining the design specifications that tell the programmers exactly what the system must do. The programmers write the necessary programs that accept inputs from the user, perform the necessary calculations, produce hardcopy of the report or display it on the screen

Design the Physical system.

- Specify the Input/Output media
- Design physical information flow through the system
- Plan system implementation
- Prepare a Conversion schedule and a target date
- Determine training procedure courses and timetable

3.3 SOFTWARE DESIGN:

The goal of software design is to produce a model or representation of an entity that will be built later. This project makes use of three main types of designs namely.

- Input Design
- Output Design
- Code Design

There is another important part in the design phase that is database design. It must be done at first.

3.4 INPUT DESIGN:

Input design is the process of converting a user-oriented description of the inputs to a computer-based system into a programmer oriented specification. Inaccurate input data is the most common cause of data processing error. If poor input design particularly where operators must enter data from source produced are of little value, the input design process was initiated in the study phase as part of the feasibility study. The most common source of data processing error is inaccurate input data. The objective of input design is to create an input layout that is easy to follow and does not include operator errors. One of the most common input devices is CRT terminal. The layout of the screen is designed and documented with the use of a display layout sheet or its equivalent. Their input media, such as key-to-tape and key-to-disk devices, optical readers and punched cards is also used and have particular design consideration.

3.5 OUTPUT DESIGN:

Output design has been an ongoing activity almost from the beginning of the project. In this phase outputs were identified and described generally in the project directive. Medium was the selected and sketches made of each output. Both printed outputs and CRT displays can include a title, column headings, detailed data and tools. They must be described in detail for programmers. Forms called print charts and display layout sheets are used to communicate this design detail. Especially forms can be designed for printed reports.

3.6 CODE DESIGN:

After designing the new system, the whole system is required to be converted into computer understanding language. Coding the new system into computer programming language does this. It is an important stage where the defined procedures are transformed into control specifications by the help of a computer language. This is also called the programming phase in which the programmer converts the program specifications into computer instructions, which we refer as programs. The programs coordinate the data movements and control the entire process in a system.

It is generally felt that the programs must be modular in nature. This helps in fast development, maintenance and future change, if required.

3.7 UML Diagrams:

UML is the international standard notation for object-oriented analysis and design. The Object Management Group defines it. The heart of object-oriented problem solving is the construction of a model. The model abstracts the essential details of the underlying problem from its usually complicated real world. Several modeling tools are wrapped under the heading of the Unified Modeling Language.

An Overview of UML

The UML is a language for

Visualizing

Specifying

Constructing

Documenting

These are the artifacts of a software-intensive system. The major elements of UML are

The UML's basic building blocks

The rules that dictate how those building blocks may be put together.

Some common mechanisms that apply throughout the UML.

3.7.1 BASIC BUILDING BLOCKS OF THE UML:

The vocabulary of UML encompasses three kinds of building blocks:

Things

Relationships

Diagram

Things in the UML:

They are the abstractions that are first-class citizens in a model. There are four kinds of things in the UML

Structural things

Behavioral things.

Grouping things.

Annotational things.

These things are the basic object oriented building blocks of the UML. They are used to write well-formed models.

Relationships in the UML:

There are four kinds of relationships in the UML:

Dependency

Association

Generalization

Realization

Diagrams in the UML:

Diagrams play a very important role in the UML. There are nine kind of modeling diagrams as follows:

Use Case Diagram

Class Diagram

Sequence Diagram

Collaboration Diagram

Object Diagram

Activity Diagram

State Chart Diagram

Component Diagram

Deployment Diagram

3.7.2 Class Diagram:

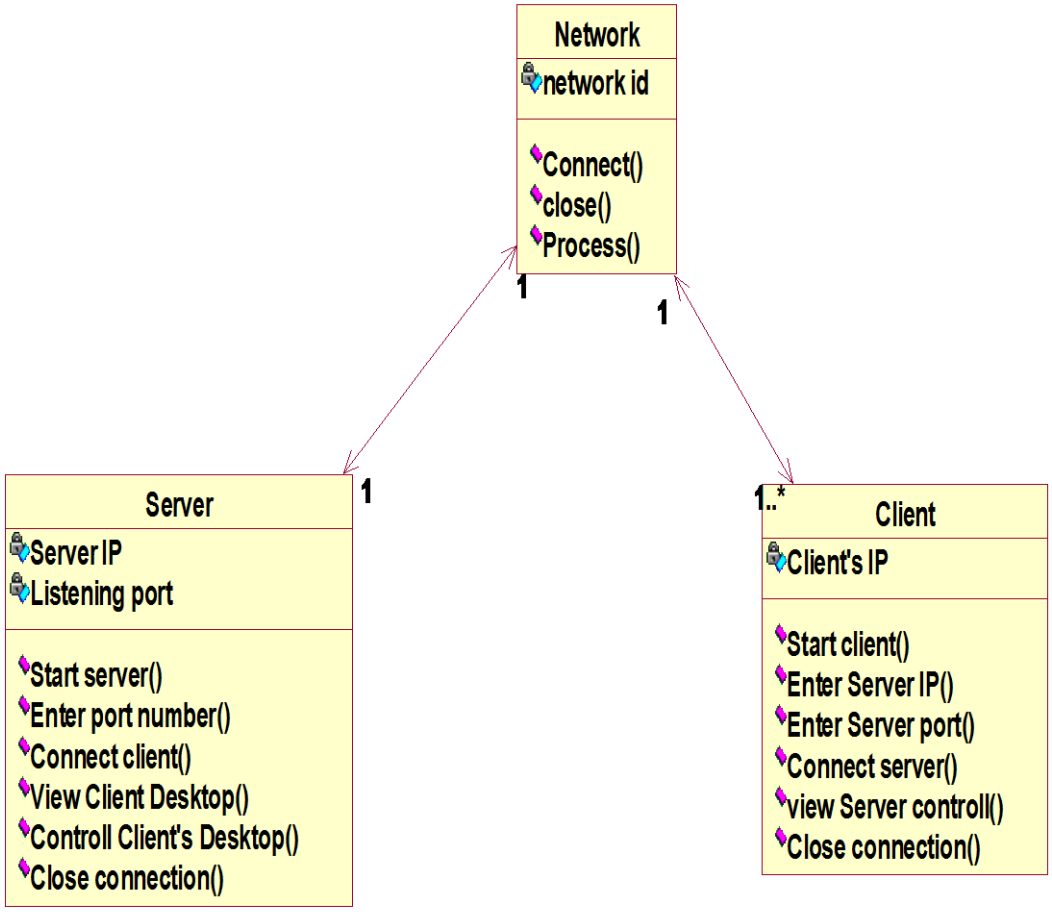

Fig.3.7.2 Class Diagram

3.7.3 Use Case Diagram:

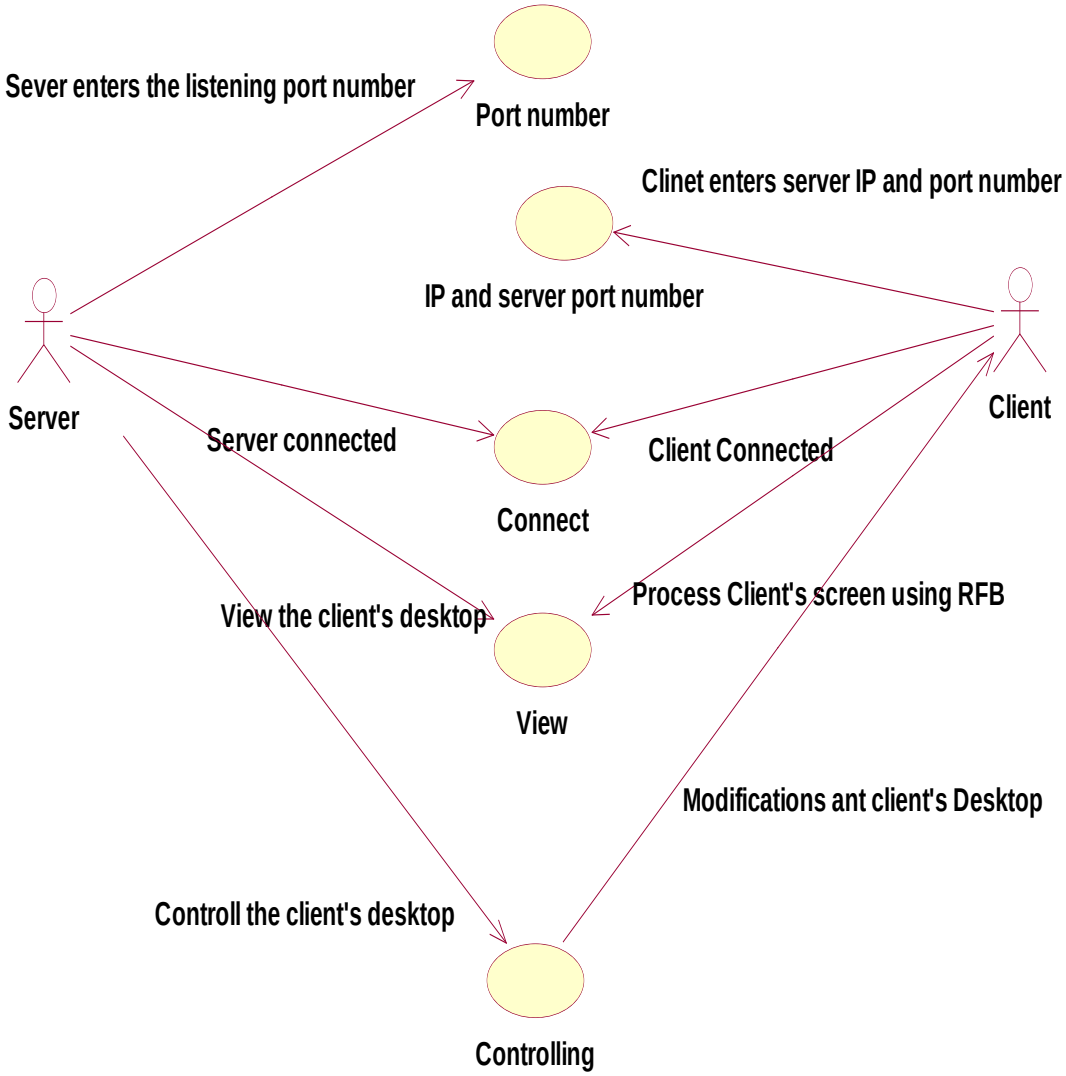

Fig.3.7.3Use Case Diagram of Remote Desktop

3.7.4 Sequence Diagram

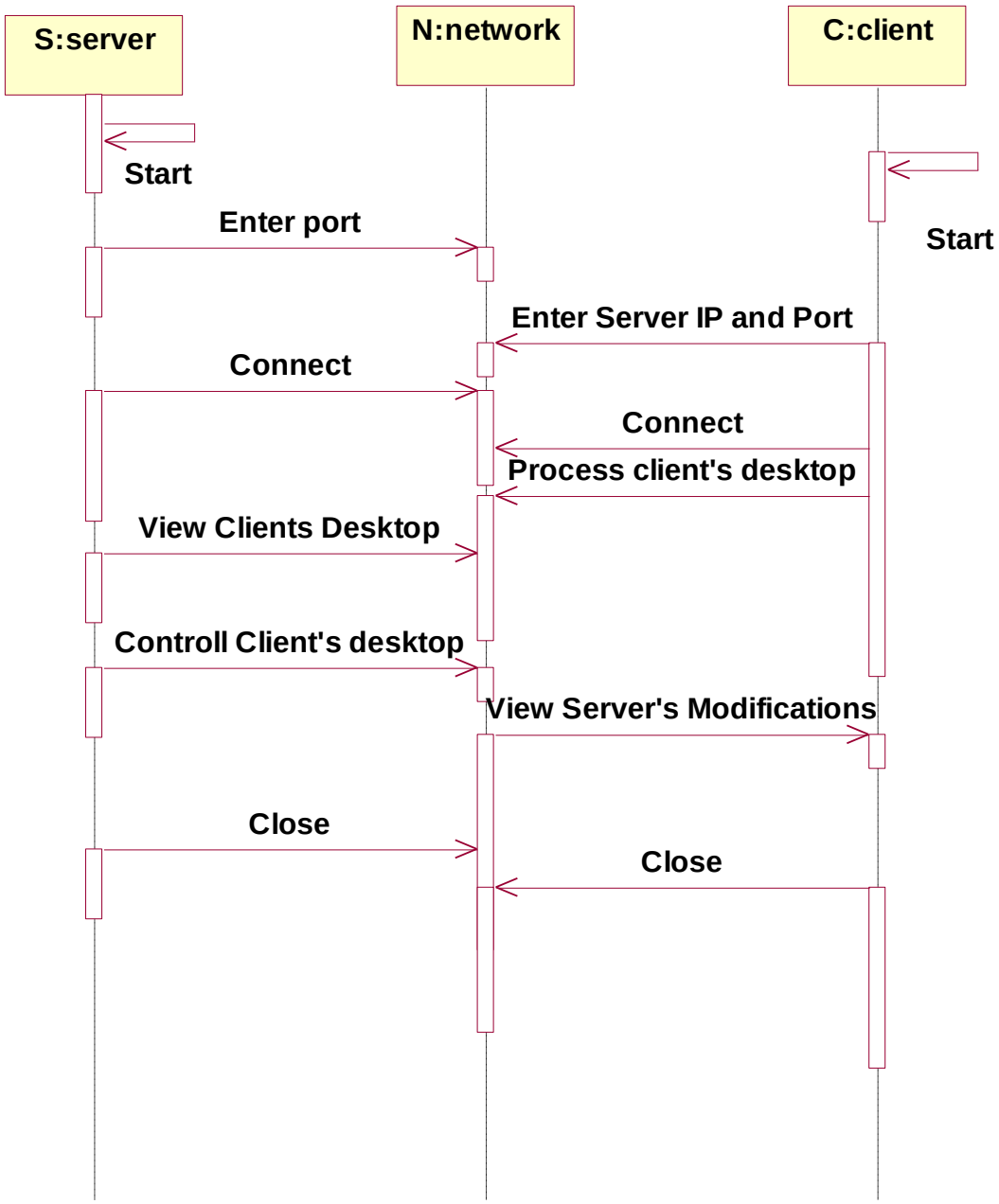

Fig.3.7.4Sequence Diagram of Remote desktop

3.7.5 Collaboration Diagram:

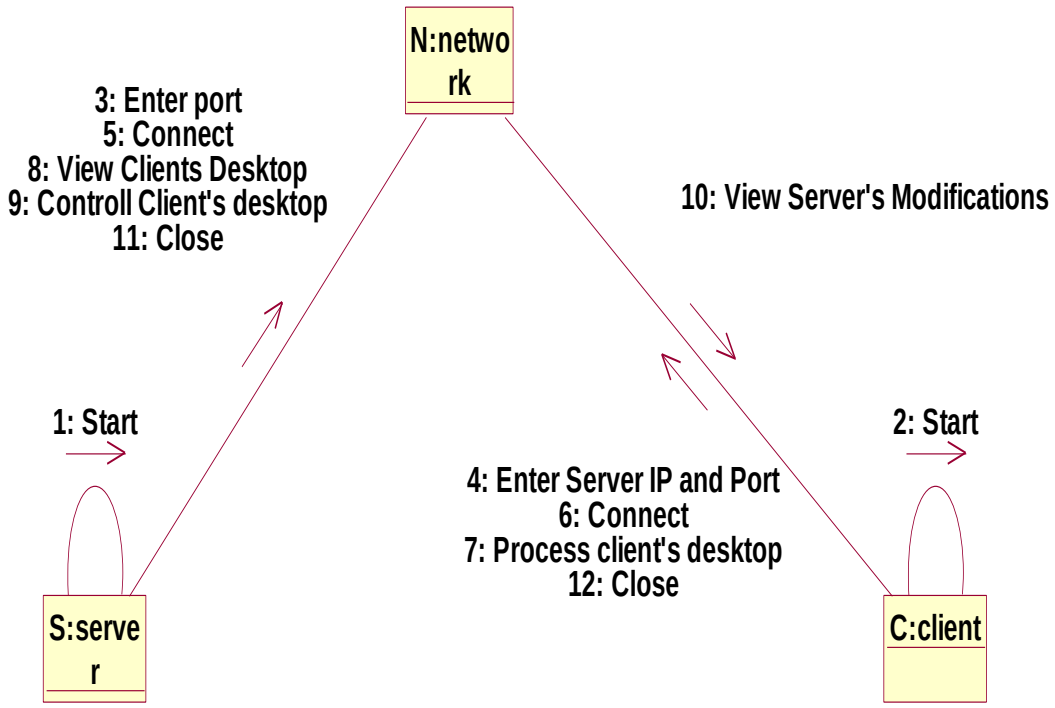

Fig. 3.7.5 Collaboration Diagram of remote desktop

3.7.6 Activity Diagram of server:

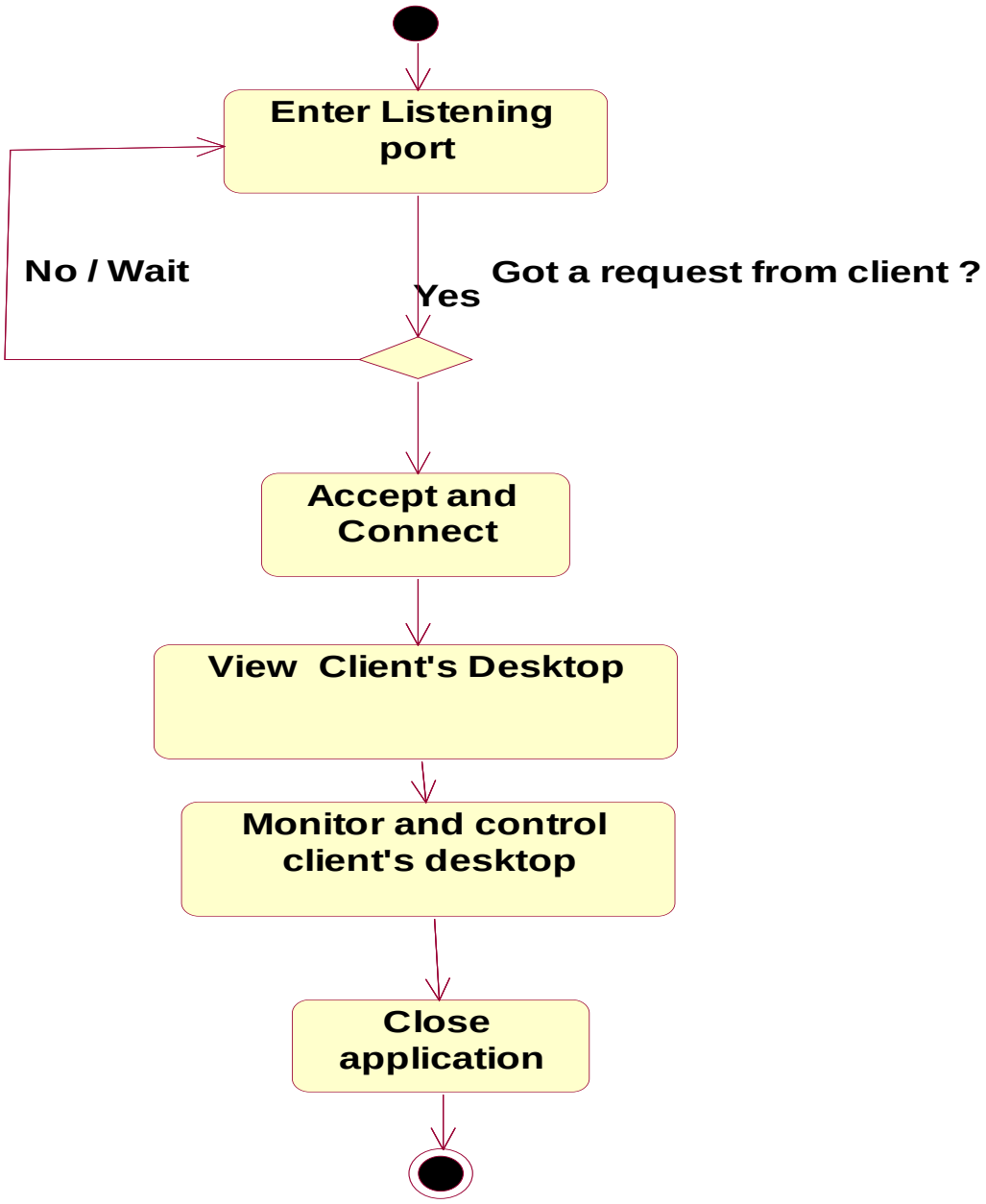

Fig. 3.7.6 Activity diagram for Server

3.7.7Activity Diagram of Client:

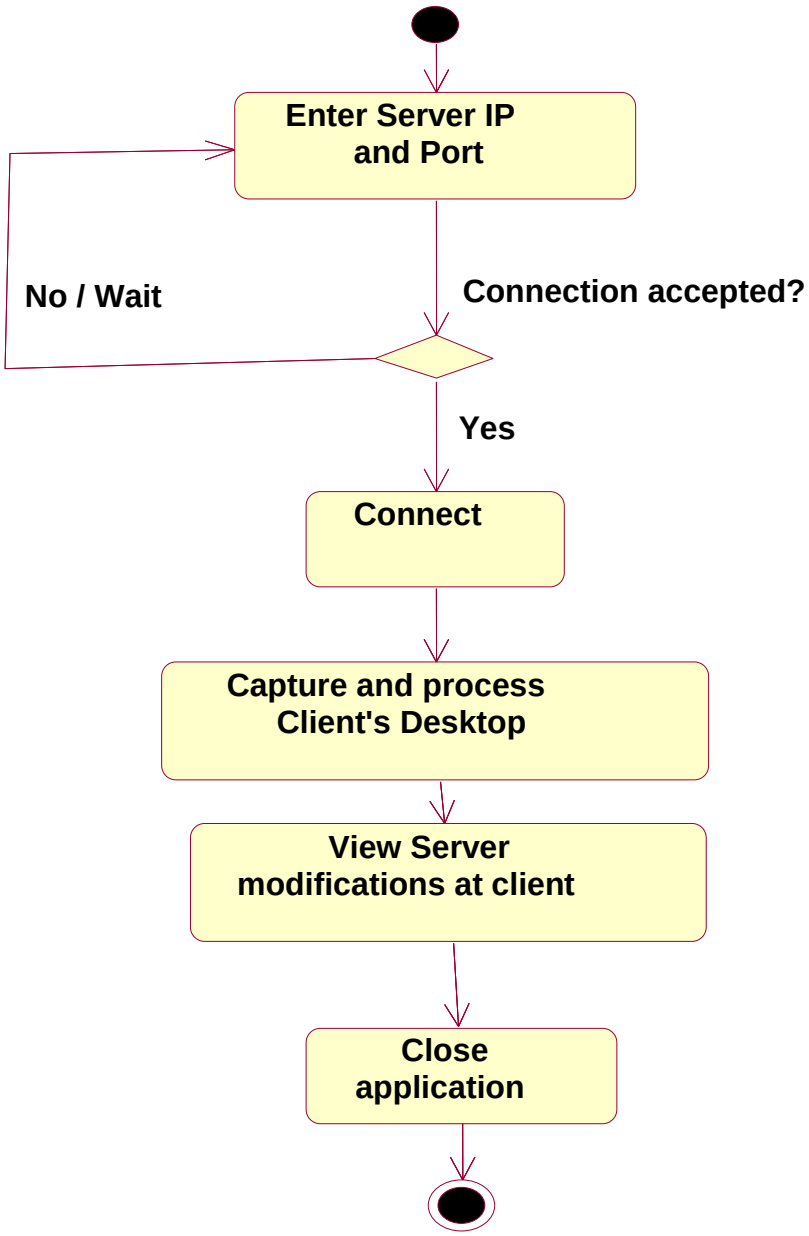

Fig.3.7.7 Activity diagram for Client

4.CODING

Server:

```
packageremoteserver;

importjava.awt.BorderLayout;
importjava.io.IOException;
importjava.net.ServerSocket;
importjava.net.Socket;
importjavax.swing.JDesktopPane;
importjavax.swing.JFrame;
importjavax.swing.JOptionPane;

/**
 * This is the entry class of the server
 */
public class ServerInitiator {
   //Main server frame
privateJFrame frame = new JFrame();
   //JDesktopPane represents the main container that will contain all
   //connected clients' screens
privateJDesktopPane desktop = new JDesktopPane();

public static void main(String args[]){
      String port = JOptionPane.showInputDialog("Please enter listening port");
newServerInitiator().initialize(Integer.parseInt(port));
   }

public void initialize(int port){

try {
ServerSocketsc = new ServerSocket(port);
         //Show Server GUI
drawGUI();
         //Listen to server port and accept clients connections
while(true){
            Socket client = sc.accept();
System.out.println("New client Connected to the server");
            //Per each client create a ClientHandler
newClientHandler(client,desktop);
         }
      } catch (IOException ex) {
ex.printStackTrace();
      }
   }
```

```
   /*
    * Draws the main server GUI
    */
public void drawGUI(){
frame.add(desktop,BorderLayout.CENTER);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          //Show the frame in a maximized state
frame.setExtendedState(frame.getExtendedState()|JFrame.MAXIMIZED_BOTH);
frame.setVisible(true);
   }
}



packageremoteserver;

importjava.awt.Rectangle;
importjava.awt.event.KeyEvent;
importjava.awt.event.KeyListener;
importjava.awt.event.MouseEvent;
importjava.awt.event.MouseListener;
importjava.awt.event.MouseMotionListener;
importjava.io.IOException;
importjava.io.PrintWriter;
importjava.net.Socket;
importjavax.swing.JPanel;

classClientCommandsSender implements KeyListener,
MouseMotionListener,MouseListener {

private Socket cSocket = null;
privateJPanelcPanel = null;
privatePrintWriter writer = null;
private Rectangle clientScreenDim = null;

ClientCommandsSender(Socket s, JPanel p, Rectangle r) {
cSocket = s;
cPanel = p;
clientScreenDim = r;
      //Associate event listners to the panel
cPanel.addKeyListener(this);
cPanel.addMouseListener(this);
cPanel.addMouseMotionListener(this);
try {
          //Prepare PrintWriter which will be used to send commands to
          //the client
writer = new PrintWriter(cSocket.getOutputStream());
      } catch (IOException ex) {
ex.printStackTrace();
```

```
        }

    }

    //Not implemeted yet
public void mouseDragged(MouseEvent e) {
    }

public void mouseMoved(MouseEvent e) {
doublexScale = clientScreenDim.getWidth()/cPanel.getWidth();
System.out.println("xScale: " + xScale);
doubleyScale = clientScreenDim.getHeight()/cPanel.getHeight();
System.out.println("yScale: " + yScale);
System.out.println("Mouse Moved");
writer.println(EnumCommands.MOVE_MOUSE.getAbbrev());
writer.println((int)(e.getX() * xScale));
writer.println((int)(e.getY() * yScale));
writer.flush();
    }

    //this is not implemented
public void mouseClicked(MouseEvent e) {
    }

public void mousePressed(MouseEvent e) {
System.out.println("Mouse Pressed");
writer.println(EnumCommands.PRESS_MOUSE.getAbbrev());
int button = e.getButton();
intxButton = 16;
if (button == 3) {
xButton = 4;
        }
writer.println(xButton);
writer.flush();
    }

public void mouseReleased(MouseEvent e) {
System.out.println("Mouse Released");
writer.println(EnumCommands.RELEASE_MOUSE.getAbbrev());
int button = e.getButton();
intxButton = 16;
if (button == 3) {
xButton = 4;
        }
writer.println(xButton);
writer.flush();
    }

    //not implemented
public void mouseEntered(MouseEvent e) {
```

```
    }

    //not implemented
public void mouseExited(MouseEvent e) {

    }

    //not implemented
public void keyTyped(KeyEvent e) {
    }

public void keyPressed(KeyEvent e) {
System.out.println("Key Pressed");
writer.println(EnumCommands.PRESS_KEY.getAbbrev());
writer.println(e.getKeyCode());
writer.flush();
    }

public void keyReleased(KeyEvent e) {
System.out.println("Mouse Released");
writer.println(EnumCommands.RELEASE_KEY.getAbbrev());
writer.println(e.getKeyCode());
writer.flush();}}


packageremoteserver;

importjava.awt.BorderLayout;
importjava.awt.Rectangle;
importjava.beans.PropertyVetoException;
importjava.io.IOException;
importjava.io.ObjectInputStream;
importjava.net.Socket;
importjavax.swing.JDesktopPane;
importjavax.swing.JInternalFrame;
importjavax.swing.JPanel;

classClientHandler extends Thread {

privateJDesktopPane desktop = null;
private Socket cSocket = null;
privateJInternalFrameinterFrame = new JInternalFrame("Client Screen",
true, true, true);
privateJPanelcPanel = new JPanel();

publicClientHandler(Socket cSocket, JDesktopPane desktop) {
this.cSocket = cSocket;
this.desktop = desktop;
start();
    }
```

```
   /*
    * Draw GUI per each connected client
    */
public void drawGUI(){
interFrame.setLayout(new BorderLayout());
interFrame.getContentPane().add(cPanel,BorderLayout.CENTER);
interFrame.setSize(100,100);
desktop.add(interFrame);
try {
          //Initially show the internal frame maximized
interFrame.setMaximum(true);
      } catch (PropertyVetoException ex) {
ex.printStackTrace();
      }
      //this allows to handleKeyListener events
cPanel.setFocusable(true);
interFrame.setVisible(true);
   }

public void run(){

      //used to represent client screen size
      Rectangle clientScreenDim = null;
      //Used to read screenshots and client screen dimension
ObjectInputStreamois = null;
      //start drawing GUI
drawGUI();

try{
          //Read client screen dimension
ois = new ObjectInputStream(cSocket.getInputStream());
clientScreenDim =(Rectangle) ois.readObject();
}catch(IOException ex){
ex.printStackTrace();
}catch(ClassNotFoundException ex){
ex.printStackTrace();
      }
      //Start recieveing screenshots
newClientScreenReciever(ois,cPanel);
      //Start sending events to the client
newClientCommandsSender(cSocket,cPanel,clientScreenDim);
   }

}
```

```
packageremoteserver;

importjava.awt.Graphics;
importjava.awt.Image;
importjava.io.IOException;
importjava.io.ObjectInputStream;
importjavax.swing.ImageIcon;
importjavax.swing.JPanel;

/**
 * ClientScreenReciever is responsible for recieving client screenshot and displaying
 * it in the server. Each connected client has a separate object of this class
 */
classClientScreenReciever extends Thread {

privateObjectInputStreamcObjectInputStream = null;
privateJPanelcPanel = null;
privatebooleancontinueLoop = true;

publicClientScreenReciever(ObjectInputStreamois, JPanel p) {
cObjectInputStream = ois;
cPanel = p;
       //start the thread and thus call the run method
start();
   }

public void run(){

try {

             //Read screenshots of the client then draw them
while(continueLoop){
                  //Recieve client screenshot and resize it to the current panel size
ImageIconimageIcon = (ImageIcon) cObjectInputStream.readObject();
System.out.println("New image recieved");
                  Image image = imageIcon.getImage();
image = image.getScaledInstance(cPanel.getWidth(),cPanel.getHeight()
,Image.SCALE_FAST);
                  //Draw the recieved screenshot
                  Graphics graphics = cPanel.getGraphics();
graphics.drawImage(image, 0, 0, cPanel.getWidth(),cPanel.getHeight(),cPanel);
               }
          } catch (IOException ex) {
ex.printStackTrace();
        } catch(ClassNotFoundException ex){
ex.printStackTrace();
        }
    }
}
```

```
packageremoteserver;

/**
 * Used to represent commands which can be sent by the server
 */
publicenumEnumCommands {
   PRESS_MOUSE(-1),
   RELEASE_MOUSE(-2),
   PRESS_KEY(-3),
   RELEASE_KEY(-4),
   MOVE_MOUSE(-5);

privateint abbrev;

EnumCommands(int abbrev){
this.abbrev = abbrev;
   }

publicintgetAbbrev(){
return abbrev;
   }
}
```

Client:

```
packageremoteclient;

importjava.awt.AWTException;
importjava.awt.Dimension;
importjava.awt.GraphicsDevice;
importjava.awt.GraphicsEnvironment;
importjava.awt.Rectangle;
importjava.awt.Robot;
importjava.awt.Toolkit;
importjava.awt.event.ActionEvent;
importjava.awt.event.ActionListener;
importjava.io.IOException;
importjava.net.Socket;
importjava.net.UnknownHostException;
importjavax.swing.JButton;
importjavax.swing.JFrame;
importjavax.swing.JOptionPane;

/**
 *
 * This class is responsible for connecting to the server
 * and starting ScreenSpyer and ServerDelegate classes
 */
public class ClientInitiator {

   Socket socket = null;

public static void main(String[] args){
      String ip = JOptionPane.showInputDialog("Please enter server IP");
      String port = JOptionPane.showInputDialog("Please enter server port");
newClientInitiator().initialize(ip, Integer.parseInt(port));
   }

public void initialize(String ip, int port ){

      Robot robot = null; //Used to capture the screen
      Rectangle rectangle = null; //Used to represent screen dimensions

try {
System.out.println("Connecting to server .....");
socket = new Socket(ip, port);
System.out.println("Connection Established.");

         //Get default screen device
GraphicsEnvironmentgEnv=GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevicegDev=gEnv.getDefaultScreenDevice();

         //Get screen dimensions
```

```
        Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
rectangle = new Rectangle(dim);

        //Prepare Robot object
robot = new Robot(gDev);

        //draw client gui
drawGUI();
        //ScreenSpyer sends screenshots of the client screen
newScreenSpyer(socket,robot,rectangle);
        //ServerDelegaterecievesserver commands and execute them
newServerDelegate(socket,robot);
      } catch (UnknownHostException ex) {
ex.printStackTrace();
      } catch (IOException ex) {
ex.printStackTrace();
      } catch (AWTException ex) {
ex.printStackTrace();
      }
   }

private void drawGUI() {
JFrame frame = new JFrame("Remote Admin");
JButton button= new JButton("Terminate");

frame.setBounds(100,100,150,150);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.add(button);
button.addActionListener( new ActionListener() {

public void actionPerformed(ActionEvent e) {
System.exit(0);
        }
      }
    );
frame.setVisible(true);
   }
}
```

```
packageremoteclient;

importjava.awt.Robot;
importjava.io.IOException;
importjava.net.Socket;
importjava.util.Scanner;

/*
 * Used to recieve server commands then execute them at the client side
 */
classServerDelegate extends Thread {

   Socket socket = null;
   Robot robot = null;
booleancontinueLoop = true;

publicServerDelegate(Socket socket, Robot robot) {
this.socket = socket;
this.robot = robot;
start(); //Start the thread and hence calling run method
   }

public void run(){
      Scanner scanner = null;
try {
         //prepare Scanner object
System.out.println("Preparing InputStream");
scanner = new Scanner(socket.getInputStream());

while(continueLoop){
            //recieve commands and respond accordingly
System.out.println("Waiting for command");
int command = scanner.nextInt();
System.out.println("New command: " + command);
switch(command){
case -1:
robot.mousePress(scanner.nextInt());
break;
case -2:
robot.mouseRelease(scanner.nextInt());
break;
case -3:
robot.keyPress(scanner.nextInt());
break;
case -4:
robot.keyRelease(scanner.nextInt());
break;
case -5:
robot.mouseMove(scanner.nextInt(), scanner.nextInt());
```

```
break;
                }
            }
        } catch (IOException ex) {
ex.printStackTrace();
        }}}



packageremoteclient;

importjava.awt.Rectangle;
importjava.awt.Robot;
importjava.awt.image.BufferedImage;
importjava.io.IOException;
importjava.io.ObjectOutputStream;
importjava.net.Socket;
importjavax.swing.ImageIcon;

/**
 * This class is responisble for sending sreenshotevery predefined duration
 */
classScreenSpyer extends Thread {

    Socket socket = null;
    Robot robot = null; // Used to capture screen
    Rectangle rectangle = null; //Used to represent screen dimensions
booleancontinueLoop = true; //Used to exit the program

publicScreenSpyer(Socket socket, Robot robot,Rectanglerect) {
this.socket = socket;
this.robot = robot;
rectangle = rect;
start();
    }

public void run(){
ObjectOutputStreamoos = null; //Used to write an object to the streem


try{
            //Prepare ObjectOutputStream
oos = new ObjectOutputStream(socket.getOutputStream());
            /*
             * Send screen size to the server in order to calculate correct mouse
             * location on the server's panel
             */
oos.writeObject(rectangle);
}catch(IOException ex){
ex.printStackTrace();
```

```
        }

while(continueLoop){
            //Capture screen
BufferedImage image = robot.createScreenCapture(rectangle);
            /* I have to wrap BufferedImage with ImageIcon because BufferedImage class
             * does not implement Serializable interface
             */
ImageIconimageIcon = new ImageIcon(image);

            //Send captured screen to the server
try {
System.out.println("before sending image");
oos.writeObject(imageIcon);
oos.reset(); //Clear ObjectOutputStream cache
System.out.println("New screenshot sent");
            } catch (IOException ex) {
ex.printStackTrace();
            }

            //wait for 100ms to reduce network traffic
try{
Thread.sleep(100);
}catch(InterruptedException e){
e.printStackTrace();
            }
        }

    }

}
```

```
packageremoteclient;

/**
 * Used to represent commands which can be sent by the server
 */
publicenumEnumCommands {
    PRESS_MOUSE(-1),
    RELEASE_MOUSE(-2),
    PRESS_KEY(-3),
    RELEASE_KEY(-4),
    MOVE_MOUSE(-5);

privateint abbrev;

EnumCommands(int abbrev){
this.abbrev = abbrev;
    }

publicintgetAbbrev(){
return abbrev;
    }
}
```

5. TESTING

S.NO	Input	Expected Behavior	Observed Behavior	Status P=Pass F=Fail
1.	Giving the correct port address and ip address	The connection to be established and capture screen.	Connection has been established and screen has been captured.	P
2.	Giving the incorrect port address and correct ip address.	The connection to be established and capture screen.	Connection has not been established and screen has not been captured.	F
3.	Giving the correct port address and incorrect ip address.	The connection to be established and to capture screen.	Connection has not been established and screen has not been captured.	F
4.	Connecting to two different clients using the same port address and ip address.	The connection to be established and two different screens are captured.	Connection has been established and two different screens are captured.	P
5.	Connecting to more than two clients using the same port address and ip address.	The connection to be established and more than two different screens are captured.	Connection has been established and more than two different screens are captured.	P

6.RESULTS

6.1 Screen Shots

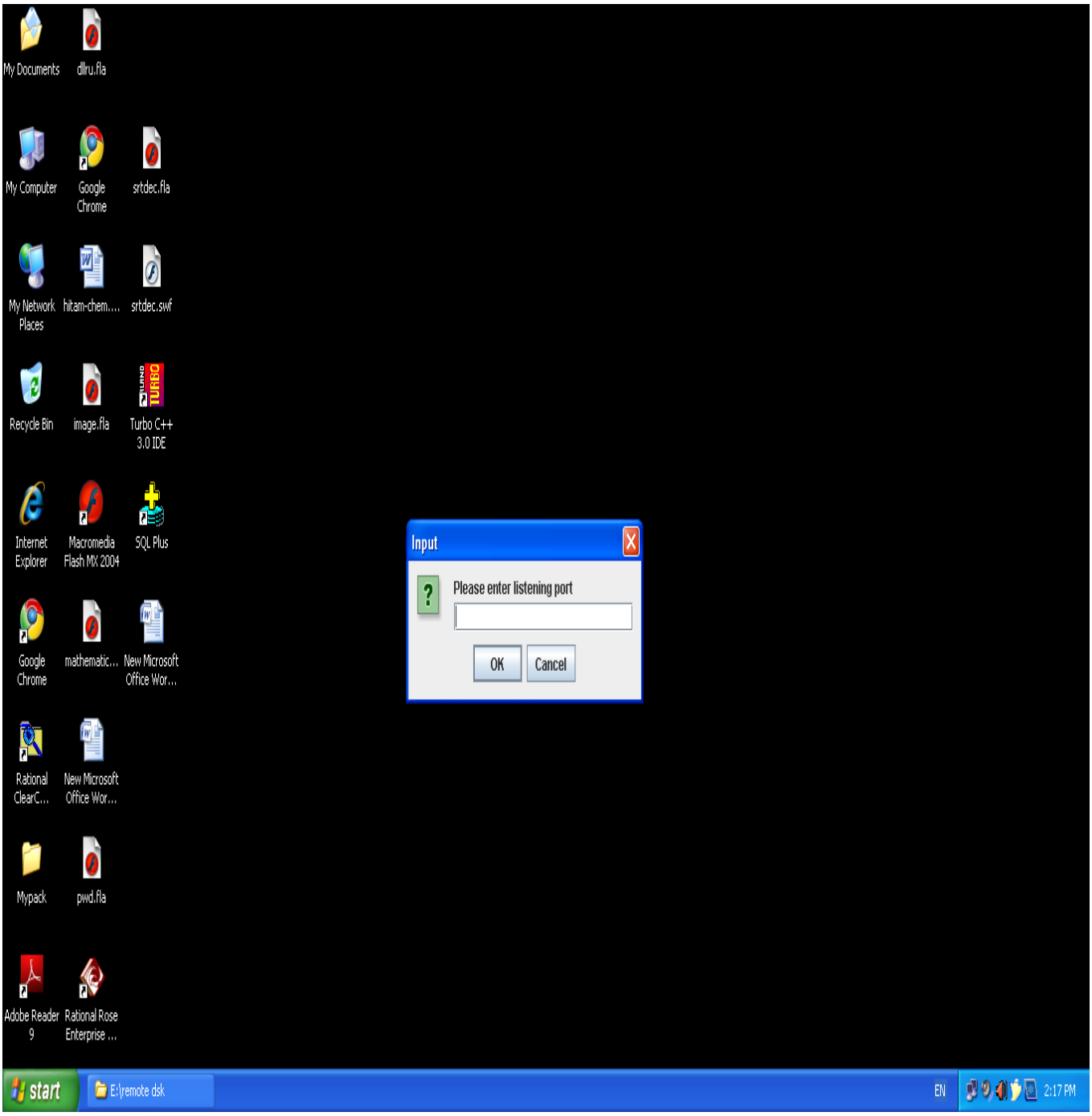

Figure 6.1.1 server listening port

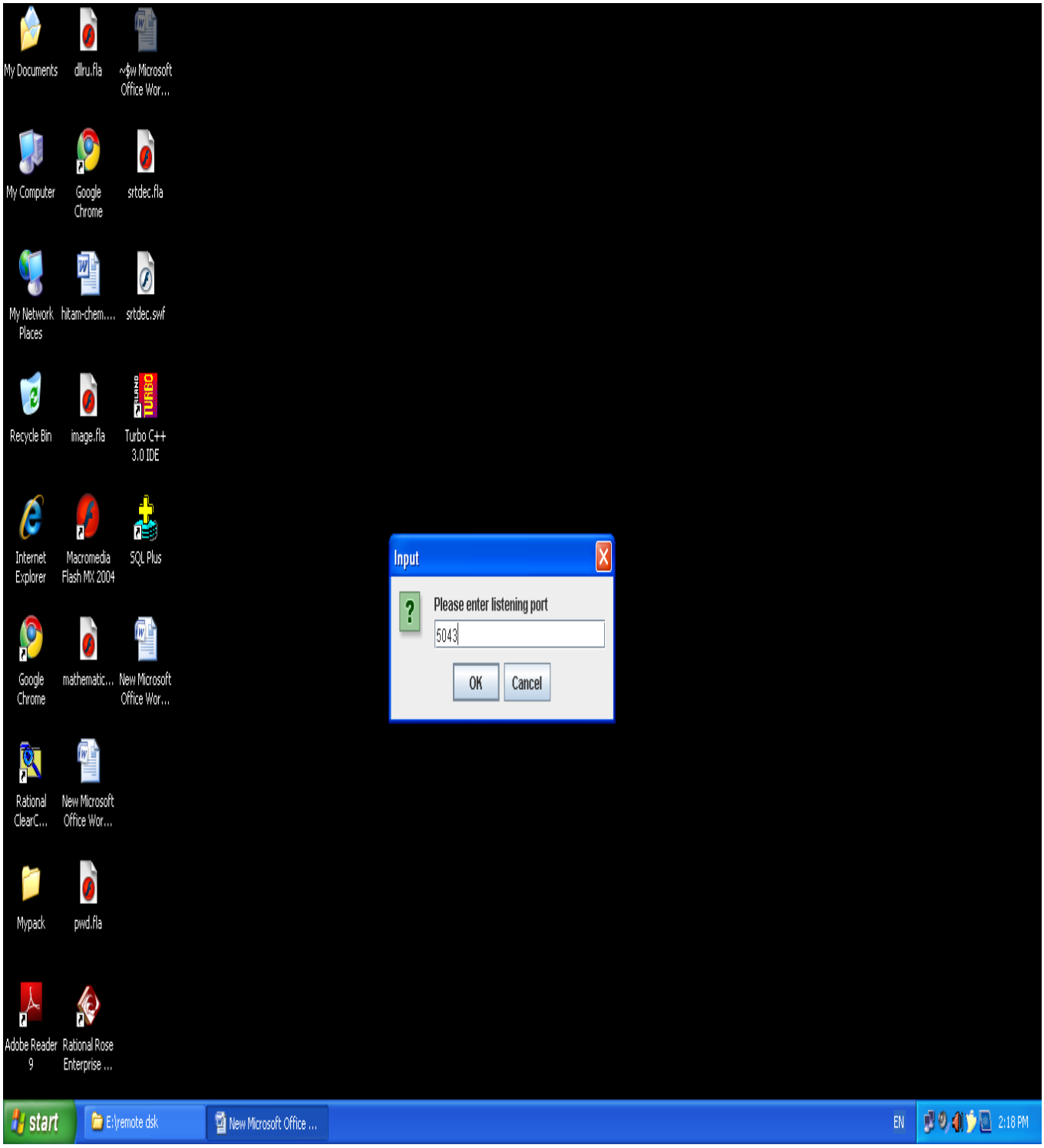

Figure 6.1.2 server screen entering listening port

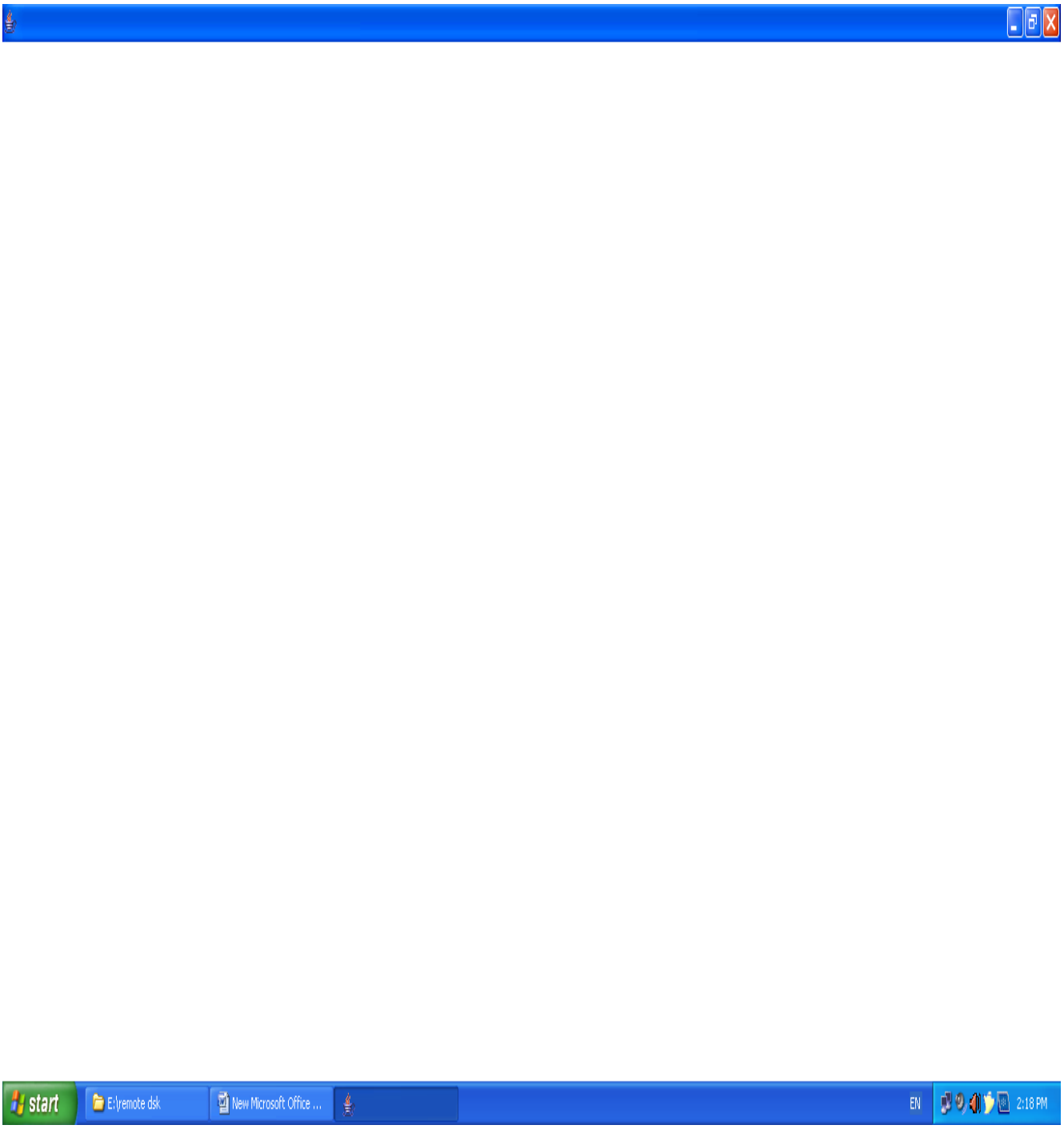

Figure 6.1.3 server screen

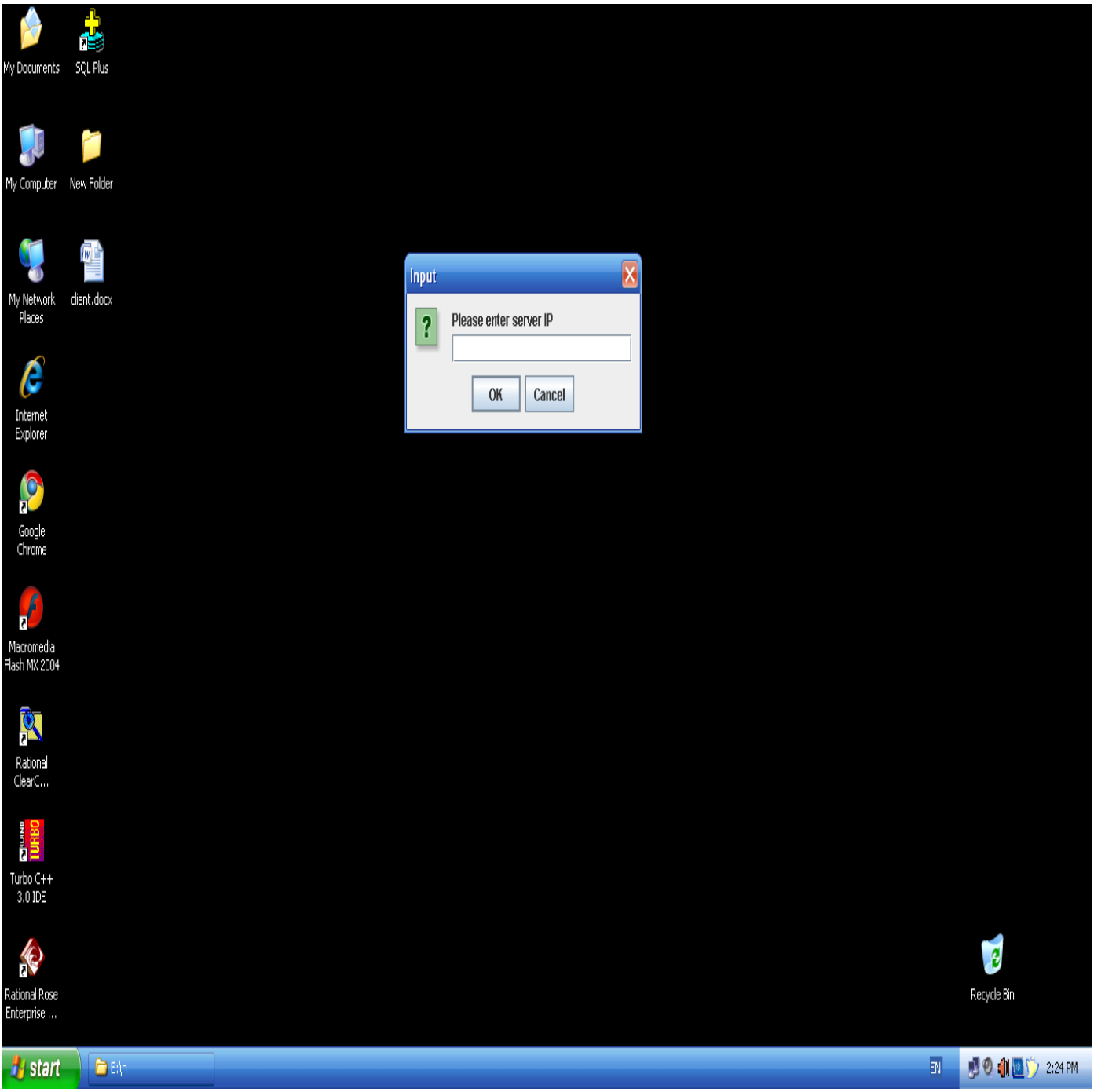

Figure 6.1.4 client screen entering server ip

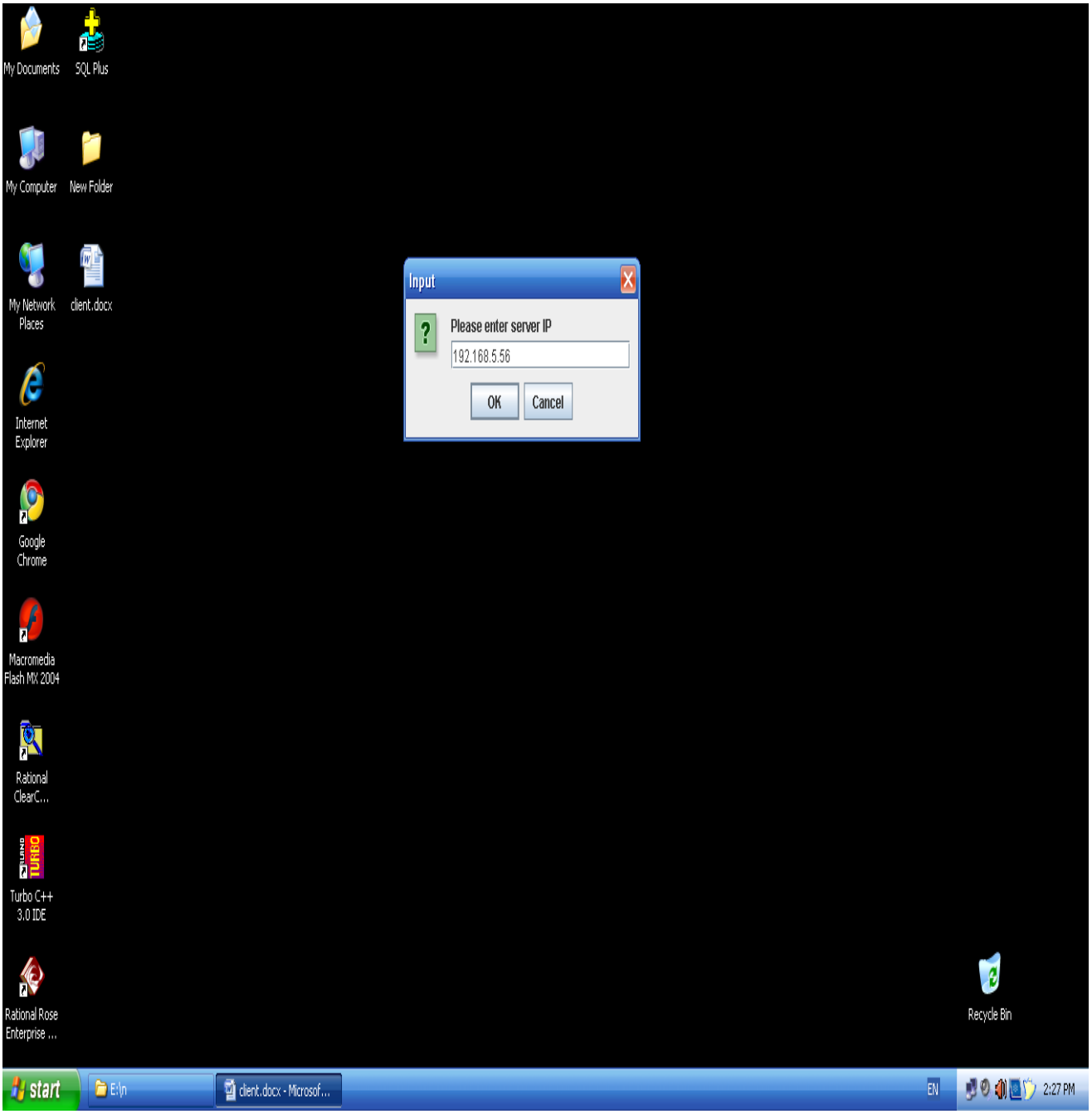

Figure 6.1.5 client screen entering server ip

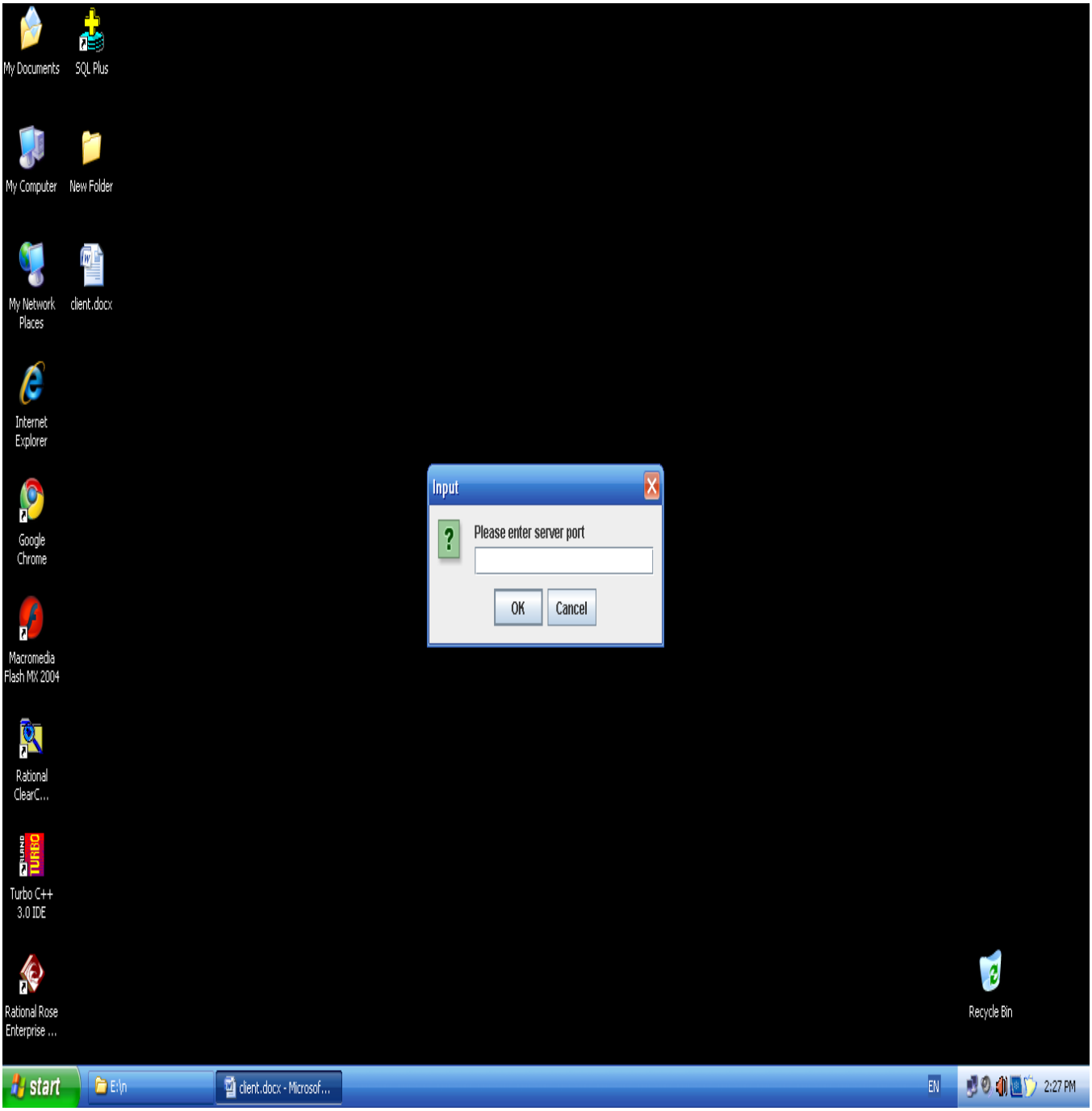

Figure 6.1.6 client screen entering server port

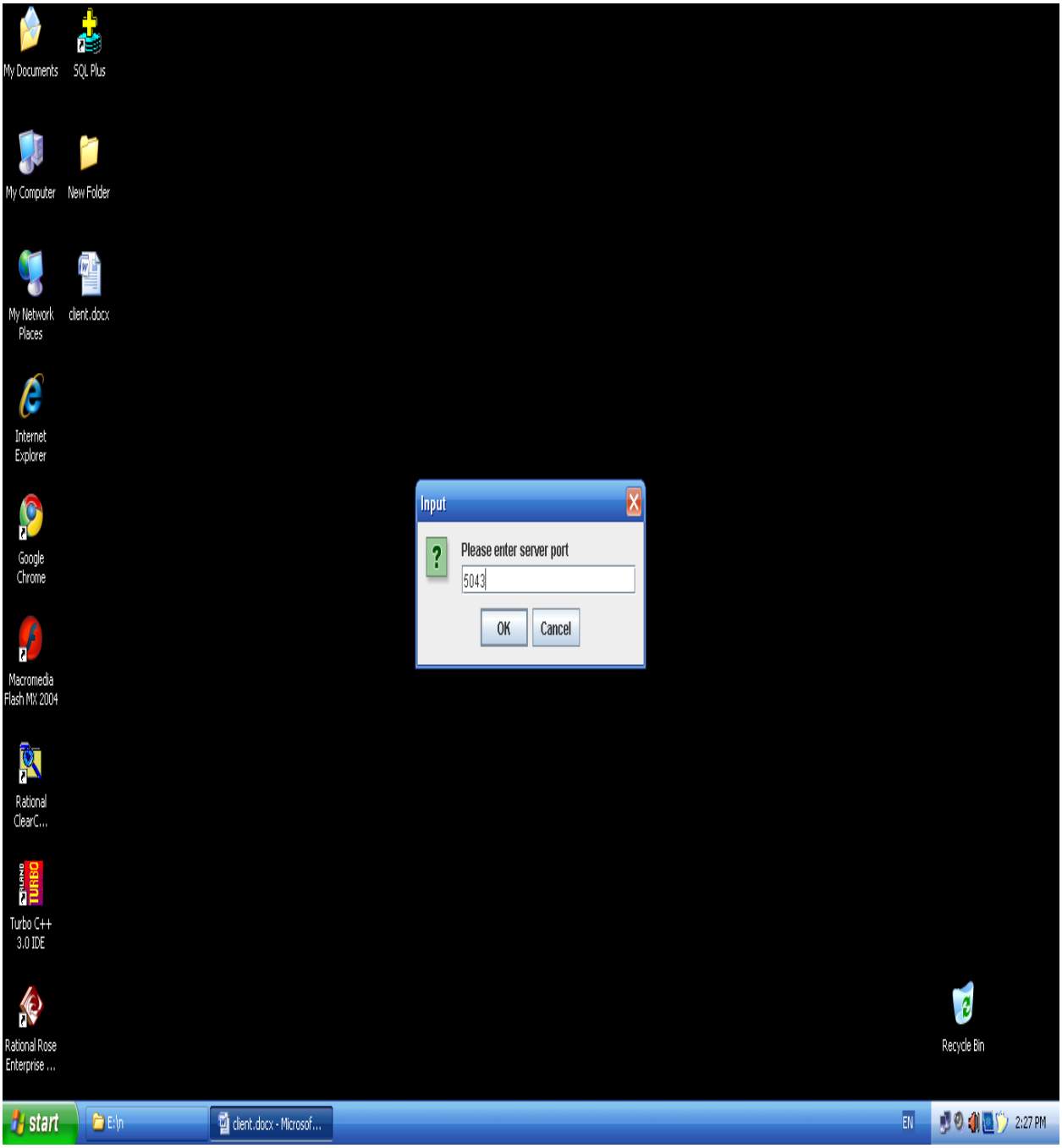

Figure 6.1.7 client screen entering server port

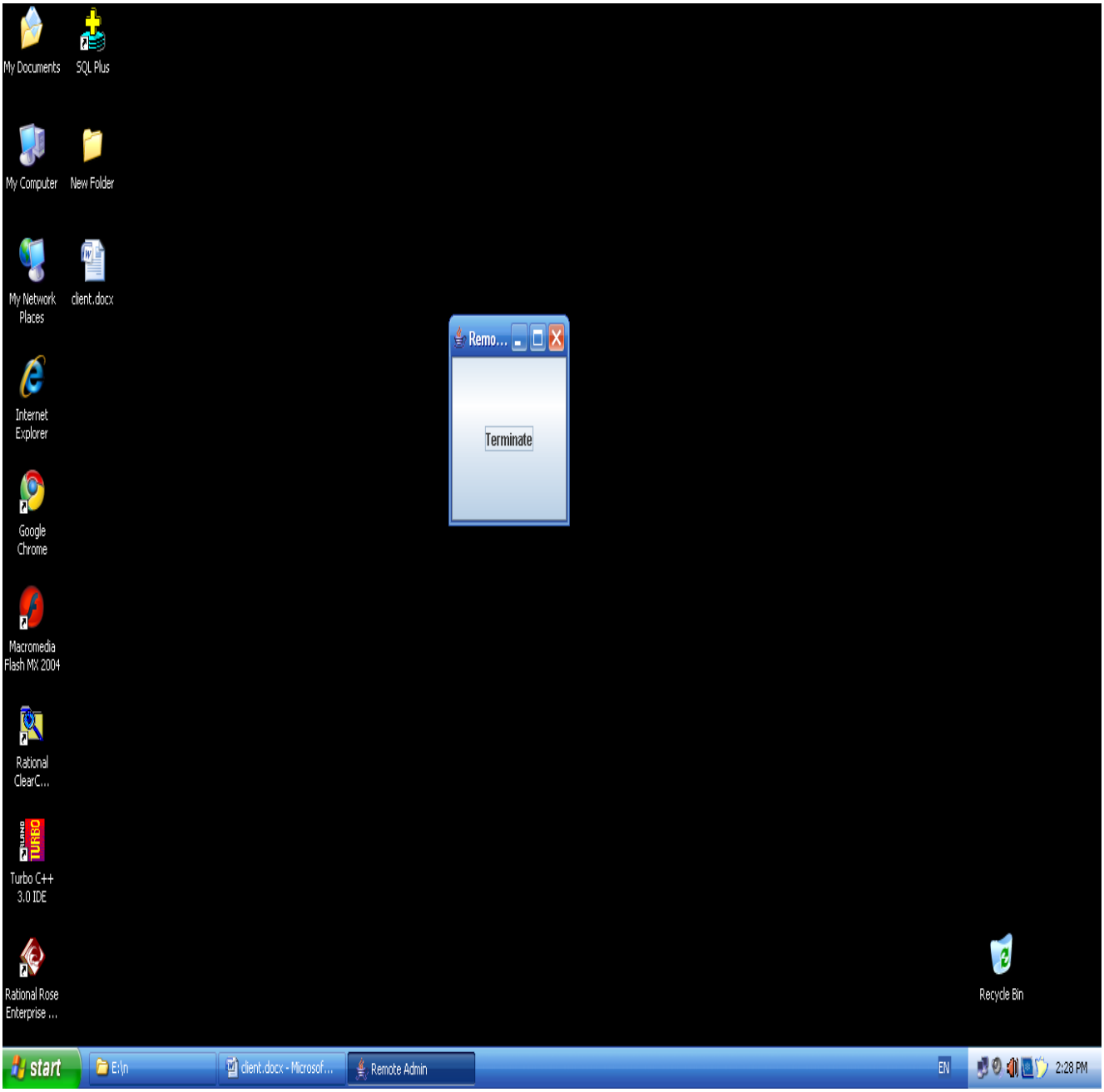

Figure 6.1.8 client screen after connection

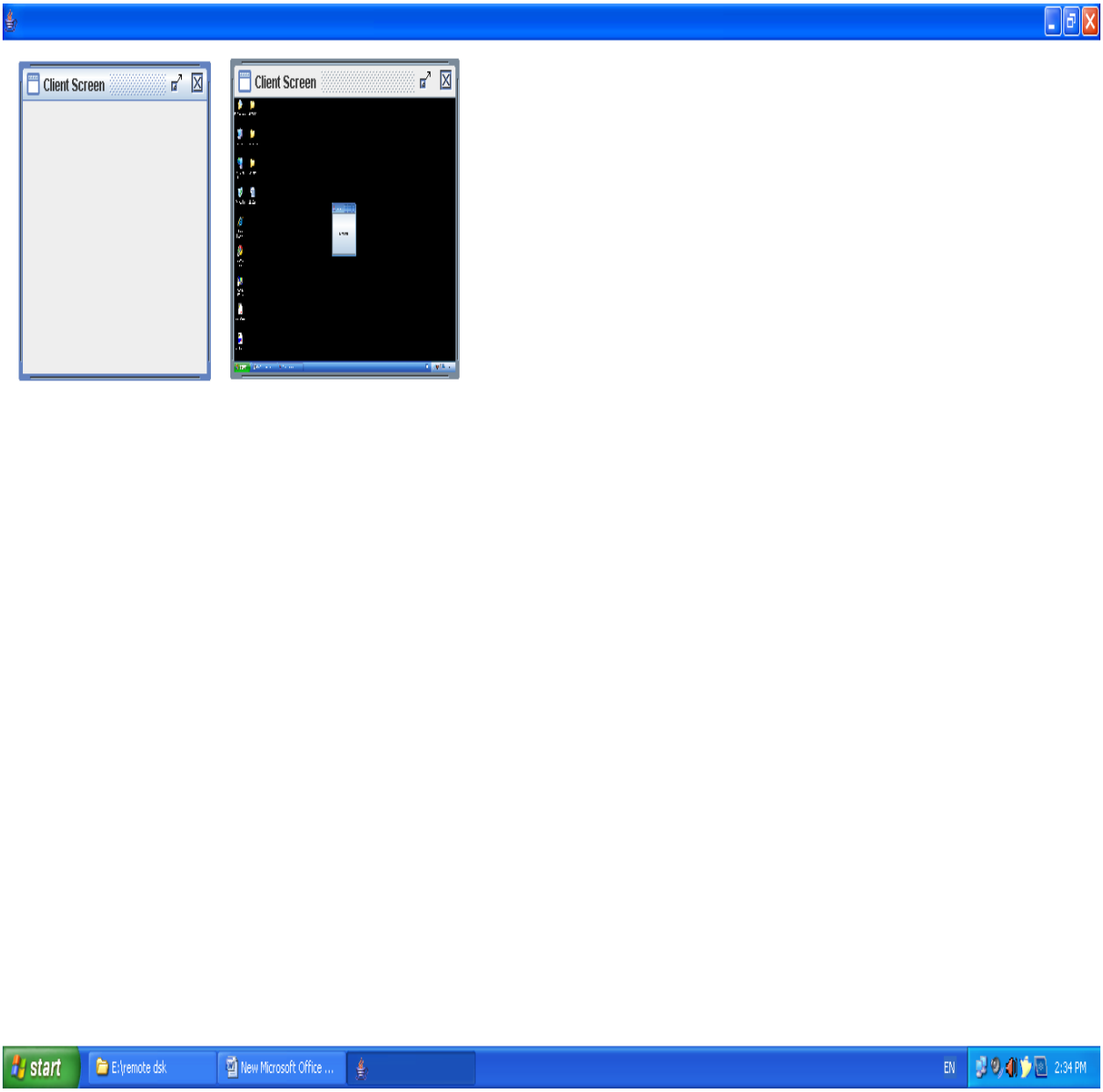

Figure 6.1.9 server screen with two clients connecting

7.FUTURE ENHANCEMENTS

The same project can be further implemented by including the copying of the data from any client and pasting it on any client or server desktop.

The server can control the client desktop even if it is in standby mode and can also have an automatic connection establishment once the system is turned on. The clients can also be allowed to access each other systems.

8. CONCLUSION

We have proposed the Remote Desktop for connecting to more than two clients at a time using the same port address and ip address .It not only allows to connect but to capture the desktops of the clients (which can be more than two).

The normal remote applications cannot be connected to more than two clients and also the screen capturing is not up to the mark. The different client windows cannot be contained in a single frame. Thus we developed it into a project which can be connected to many number of clients and the screen capturing of it is very effective. The different client windows can be contained in a single frame and we can select a window and maximize it.

9. REFERENCES

1. THE COMPLETE REFERENCE: JAVA J2SE 5 EDITION BY **HERBERT SCHILDT**

Pages(47-49),(233- 288),(330-333)

2.DEVELOPING JAVA SERVLETS BY **JAMES GOODWIL, 4th edition**

Pages(413-455)

3.SWING a beginners guide by HERBERT SCHILDT 2nd edition

Pages(10-433)

4.CORE JAVA BY **HORSTMAN** volume 2

Chapter 4 pages(244-255),chapter 7 pages(712-745)

WebPages:

http://www.microsoft.com/windowsxp/using/mobility/getstarted.

http://www.microsoft.com/windowsxp/downloads/tools/rdclientd.

http://en.wikipedia.org/wiki/**Remote_Desktop**_Protocol

http://www.**remote-desktop**-control.com

https://secure.logmein.com

http://www.r**desktop**.org