**REMOTE DESKTOP CONTROL PROTOCOL**

**-SKYLINK**

**A PROJECT REPORT**

Submitted by

CH.EN.U4CYS22041 Mokshagna Bhuvan

CH.EN.U4CYS22063 Sai Manohar

CH.EN.U4CYS22028 Aditya Sai

20CYS383 JAVA PROGRAMMING LAB

IV Semester (2023-2024)

B. Tech in Computer Science and Engineering (Cyber Security)



**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Contents** | **Page** |
| **1.** | **Abstract** | **3** |
| **2.** | **Introduction** | **4** |
| **3.** | **Project Description** | **5** |
| **4.** | **Methodology** | **6** |
| **5.** | **Schematic Diagram** | **8** |
| **6.** | **Implementation** | **9** |
| **7.** | **Source Code** | **10** |
| **8.** | **Results/Output** | **20** |
| **9.** | **Conclusion** | **23** |

**Abstract**

Skylink Remote Desktop Control Protocol is a sophisticated Java-based solution designed to enable seamless remote access and management of desktop environments. Leveraging Object-Oriented Programming (OOP) principles, Skylink offers a robust and reliable framework for both client and server components.

The client application of Skylink boasts a user-friendly graphical interface, providing users with an intuitive platform to initiate secure connections and engage in real-time interactions with remote desktops. This interface streamlines the process of accessing and controlling remote desktops, ensuring efficiency and ease of use for users across various technical proficiencies.

On the server side, Skylink excels in managing sessions effectively. By overseeing the intricacies of session establishment and maintenance, the server component guarantees smooth communication between client and server, thereby enhancing the overall user experience. Additionally, the server is equipped to handle various security protocols, ensuring the integrity and confidentiality of data transmitted during remote sessions.

One of the key principles that Skylink prioritizes is modularity and extensibility. This design philosophy allows for seamless integration with existing systems and the flexibility to adapt to evolving requirements. The modular architecture of Skylink facilitates easy customization and scalability, making it suitable for diverse use cases such as technical support, telecommuting, remote collaboration, and more.Overall, Skylink Remote Desktop Control Protocol stands out as a reliable and versatile platform for remote desktop control. With its focus on robustness, usability, and adaptability, Skylink provides a comprehensive solution for organizations and individuals seeking efficient remote access and management capabilities.

**Introduction**

Skylink represents a modern approach to Remote Desktop Protocols (RDPs), offering a comprehensive solution for secure remote access to graphical desktop environments. Rooted in Java, Skylink capitalizes on the platform's renowned strengths, such as platform independence, Object-Oriented Programming (OOP) features, and robust security measures.Java's platform independence is a cornerstone of Skylink's architecture. By adhering to the "Write Once, Run Anywhere" principle, Skylink ensures compatibility across a wide range of operating systems, including Windows, macOS, Linux, and more. This flexibility allows users to seamlessly access and manage remote desktops regardless of their underlying platform, enhancing productivity and accessibility for diverse user bases.Moreover, Skylink leverages the power of Object-Oriented Programming (OOP) to promote code organization and scalability. By encapsulating functionalities into modular components, Skylink enhances maintainability and extensibility, facilitating rapid development and integration of new features. This OOP approach not only streamlines development processes but also ensures that Skylink remains adaptable to evolving requirements and technological advancements.Additionally, Java's strong emphasis on security aligns perfectly with Skylink's commitment to providing a secure remote desktop access solution. Java's robust security features, such as sandboxing, bytecode verification, and cryptography libraries, help mitigate potential security threats, safeguarding sensitive data and protecting against unauthorized access. By integrating these security measures into its architecture, Skylink prioritizes the confidentiality, integrity, and availability of remote desktop connections, instilling confidence among users and organizations alike.

**Project Description**

Advanced Security: Skylink implements robust encryption, authentication, and secure channels to protect data integrity and prevent unauthorized access during remote desktop sessions.

Dynamic Screen Sharing: Users can seamlessly switch between viewing their desktop and sharing their screen with remote participants, facilitating collaboration, presentations, and training sessions.

Customization and Extensibility: Skylink offers customizable interface layouts, keyboard shortcuts, and supports plugins and third-party extensions, enabling users to tailor the system to their specific needs and integrate additional functionalities.

Performance Optimization: Through efficient resource management, data compression, and network optimization techniques, Skylink minimizes latency and maximizes throughput for a smooth and responsive remote desktop experience.

Monitoring and Management: Administrators have access to comprehensive monitoring tools to track session activity, manage user permissions, and troubleshoot issues in real-time, ensuring optimal performance and security across the network.

This summary captures the essence of Skylink's key features and aspects in a concise manner, suitable for inclusion in a one-page Word document.

**Methodology**

1. Requirement Analysis

Define functionalities: Screen capture/rendering, user input handling (mouse/keyboard), data transmission/reception.

Specify supported platforms (operating systems).

Establish security requirements (authentication, encryption).

Determine performance goals (latency, responsiveness).

2. System Design

Architecture: Client-server model with network communication protocol (e.g., TCP/IP).

Data Exchange: Define data formats for screen updates, user input, and control messages. Consider data compression for network efficiency.

Threading: Design thread management strategies for handling concurrent operations (screen capture, network communication, rendering).

3. Development

Client-Side Application

Implement screen capture using libraries or platform-specific APIs.

Compress captured data for efficient network transmission.

Render received screen updates smoothly on the client machine.

Handle user input (mouse/keyboard) and send it to the server.

Server-Side Application

Capture screen updates on the target machine.

Compress screen data before sending it to the client.

Process incoming user input from the client and interact with the remote desktop.

Handle incoming client connections and manage multiple clients (if applicable).

4. Integration and Testing

Integrate the client and server applications.

Conduct thorough testing:

Functional testing to ensure all features work as intended.

Security testing to identify and address vulnerabilities.

Performance testing to measure latency, responsiveness, and resource usage.

Refine the implementation based on test results.

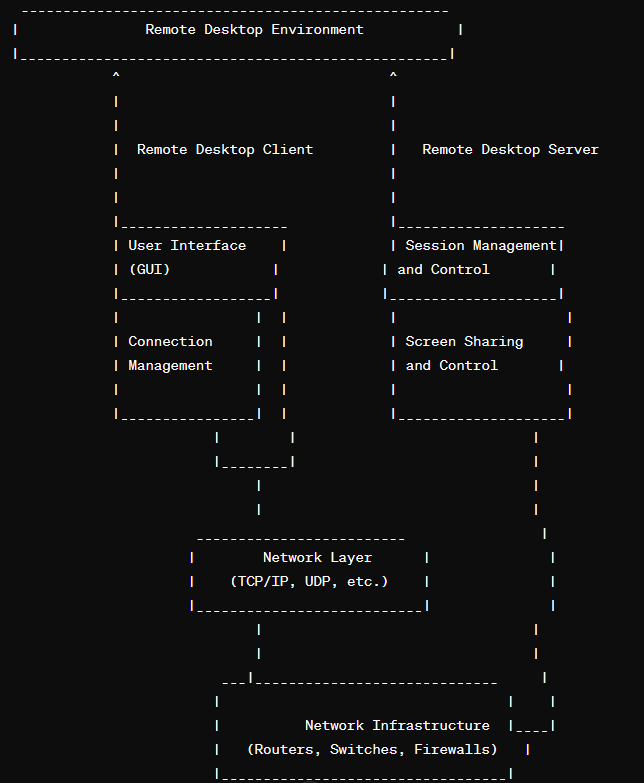
5. Deployment (Optional)

If Skylink is intended for broader use, define deployment strategies for both client and server applications.

Consider packaging mechanisms (e.g., installers) and distribution channels.

Remember: This is a high-level overview. You can expand on each step, providing more details specific to your chosen technologies and design choices.

**Schematic Diagram**



**Implementation**

Skylink utilizes Java libraries to establish a remote desktop connection between a client and server.

Client Application:

The client application, developed in Java using object-oriented principles, initiates a connection to the server by prompting the user for the server's IP address and port number. It utilizes the Robot class to capture screenshots of the client's desktop, which are then serialized into ImageIcon objects and sent to the server via an ObjectOutputStream. Concurrently, the client listens for user inputs such as mouse clicks, keyboard inputs, and mouse movements, translating them into corresponding commands and dispatching them to the server through a PrintWriter over the established socket connection. This enables real-time interaction with the client's desktop environment from the server side, facilitating remote desktop control.

Server Application:

The server application, also written in Java and following object-oriented design principles, listens for incoming client connections on a specified port using a ServerSocket. Upon connection, it creates an internal frame (JInternalFrame) for each client to display their screen and receives screenshots and screen dimensions via an ObjectInputStream. The server interprets commands sent by clients, such as mouse clicks, mouse movements, and keyboard inputs, utilizing the Robot class to execute them on the server-side. This bidirectional communication enables seamless remote desktop interaction, providing users with a responsive and intuitive means of controlling remote systems.

**Source Code**

Client side:

    import java.awt.\*;  // robot class

    import java.awt.event.\*;

    import javax.swing.\*;   //joptionpane

    import java.net.\*;  //socket

    import java.io.\*;   //stream

    import java.awt.image.\*;    // not used

    import javax.imageio.\*;     //

    import java.util.Scanner;

// main class of client

    public class ClientMain {

        Socket socket = null;

        public static void main(String[] args){

            //joptionpane is small window with a label

            //showInoputDialog accepts user input and returns as string

            String ip = JOptionPane.showInputDialog("Please enter server IP");

            String port = JOptionPane.showInputDialog("Please enter server port");

            new ClientMain().initialize(ip, Integer.parseInt(port));

        }

        public void initialize(String ip, int port ){

            Robot robot ; //Used to capture the screen

            Rectangle rectangle; //Used to represent screen dimensions

            try {

                System.out.println("Connecting to server ..");

                socket = new Socket(ip, port);

                System.out.println("Connection done.");

                //Get screen dimensions

                // dimension n rectangel class is present in awt package

                Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();

                rectangle = new Rectangle(dim);

                //Prepare Robot object

                robot = new Robot();

                //ScreenSender sends screenshots of the client screen

                new ScreenSender(socket,robot,rectangle);

                //ServerCmdExecution recieves server commands and execute them

                new ServerCmdExecution(socket,robot);

            }

            catch (Exception ex) {

                ///

            }

        }

    }

 //send sshots

    class ScreenSender extends Thread {

        Socket socket ;

        Robot robot; // Used to capture screen

        Rectangle rectangle; //Used to represent screen dimensions

        boolean continueLoop = true; //Used to exit the program

        public ScreenSender(Socket socket, Robot robot,Rectangle rect) {

            this.socket = socket;

            this.robot = robot;

            rectangle = rect;

            start();    //to start the thread

        }

        // overriding run() of Thread class

        public void run()

        {

            ObjectOutputStream oos = 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(Exception ex){

                ////

            }

        while(continueLoop){

                //Capture screen

                BufferedImage image = robot.createScreenCapture(rectangle);

            // we acnnot send BufferedImage in stream

            // so we converted it to imageicon

                ImageIcon imageIcon = 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 (Exception ex) {

                ////

                }

                //wait for 100ms to reduce network traffic

                try{

                    Thread.sleep(100);

                }

                catch(Exception e){

                    /////

                }

            }

        }

    }

class ServerCmdExecution extends Thread {

        Socket socket = null;

        Robot robot = null;

        boolean continueLoop = true;

        public ServerCmdExecution(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 (Exception e) {

                ///

            }

Server side:

import java.awt.\*;

import java.io.\*;

import java.net.\*;

import javax.swing.\*;

import java.awt.event.\*;

/\*\*

 \* This is the entry class of the server

 \*/

public class ServerMain {

    //Main server frame

    private JFrame frame = new JFrame("SERVER MAIN");

    //JDesktopPane represents the main container that will contain all

    //connected clients' screens

    private JDesktopPane desktop = new JDesktopPane();

    public static void main(String args[]){

        // showinputdialog will return user input as string

        String port = JOptionPane.showInputDialog("Please enter listening port");

        new ServerMain().initialize(Integer.parseInt(port));

    }

    public void initialize(int port){

        try {

            ServerSocket sc = 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 ClientCreation

                new ClientCreation(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);

    }

}

class ClientCreation extends Thread {

    JInternalFrame interFrame = new JInternalFrame("Client Screen",true, true, true);

     JPanel cPanel = new JPanel();

     Socket client;

     JDesktopPane desktop;

    ClientCreation(Socket client, JDesktopPane desktop) {

        this.client = client;

        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 (Exception e) {

           ///

        }

        //this allows to handle KeyListener 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

        ObjectInputStream ois = null;

        //start drawing GUI

        drawGUI();

        try{

            //Read client screen dimension

            ois = new ObjectInputStream(client.getInputStream());

            clientScreenDim =(Rectangle) ois.readObject();

        }catch(Exception e){

            ////

        }

        //Start recieveing screenshots

        new ClientScreenReciever(ois,cPanel);

        //Start sending events to the client

        new ClientCommandsSender(client,cPanel,clientScreenDim);

    }

}

class ClientScreenReciever extends Thread {

     ObjectInputStream ois;

     JPanel p;

     boolean continueLoop = true;

    public ClientScreenReciever(ObjectInputStream ois, JPanel p) {

        this.ois = ois;

        this.p = 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

                    ImageIcon imageIcon = (ImageIcon) ois.readObject();

                    System.out.println("New image recieved");

                    Image image = imageIcon.getImage();

                    image = image.getScaledInstance(p.getWidth(),p.getHeight()

                                                        ,Image.SCALE\_SMOOTH);

                    //Draw the recieved screenshot

                    Graphics graphics = p.getGraphics();

                    graphics.drawImage(image, 0, 0, p.getWidth(),p.getHeight(),p);

                }

            } catch (Exception e) {

               ////

          }

     }

}

class ClientCommandsSender implements KeyListener,

        MouseMotionListener,MouseListener {

     Socket s;

     JPanel p ;

     PrintWriter writer ;

     Rectangle r ;

    ClientCommandsSender(Socket s, JPanel p, Rectangle r) {

        this.s = s;

        this.p = p;

        this.r = r;

        //Associate event listners to the panel

        p.addKeyListener(this);

        p.addMouseListener(this);

        p.addMouseMotionListener(this);

        try {

             //Prepare PrintWriter which will be used to send commands to

             //the client

            writer = new PrintWriter(s.getOutputStream());

        } catch (Exception e) {

           /////

        }

    }

    //Not implemeted yet

    public void mouseDragged(MouseEvent e) {

    }

    public void mouseMoved(MouseEvent e) {

        // we find ratio between client screen n server screen size by dividing them

        double xScale = r.getWidth()/p.getWidth();

        System.out.println("xScale: " + xScale);

        double yScale = r.getHeight()/p.getHeight();

        System.out.println("yScale: " + yScale);

        System.out.println("Mouse Moved");

        writer.println(-5);

        writer.println((int)(e.getX() \* xScale));

        writer.println((int)(e.getY() \* yScale));

        writer.flush();

    }

    //this is not implemented

    public void mouseClicked(MouseEvent e) {

    }

    /\*  for getButton() methiod left mouse click = 1

                                    right mouse click = 3

            for robot class left mouse click = 16

                            right mouse click = 4

                            \*/

    public void mousePressed(MouseEvent e) {

        System.out.println("Mouse Pressed");

        writer.println(-1);

        int button = e.getButton();

        //first we assume left button is clicked

        int xButton = 16;

        if (button == 3) // if right button is clciked

        {

            xButton = 4;

        }

        // xbutton is value used to tell robot class which mouse button is pressed

        writer.println(xButton);

        writer.flush();

    }

    public void mouseReleased(MouseEvent e) {

        System.out.println("Mouse Released");

        writer.println(-2);

        int button = e.getButton();

        System.out.println(button);

        int xButton = 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(-3);

        writer.println(e.getKeyCode());

        writer.flush();

    }

    public void keyReleased(KeyEvent e) {

        System.out.println("Mouse Released");

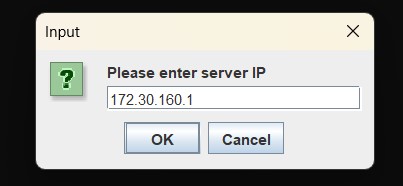
        writer.println (-4);

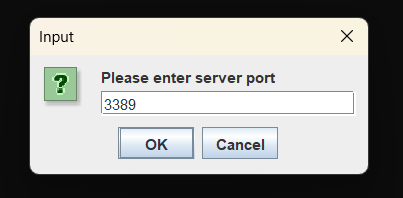
        writer.println(e.getKeyCode());

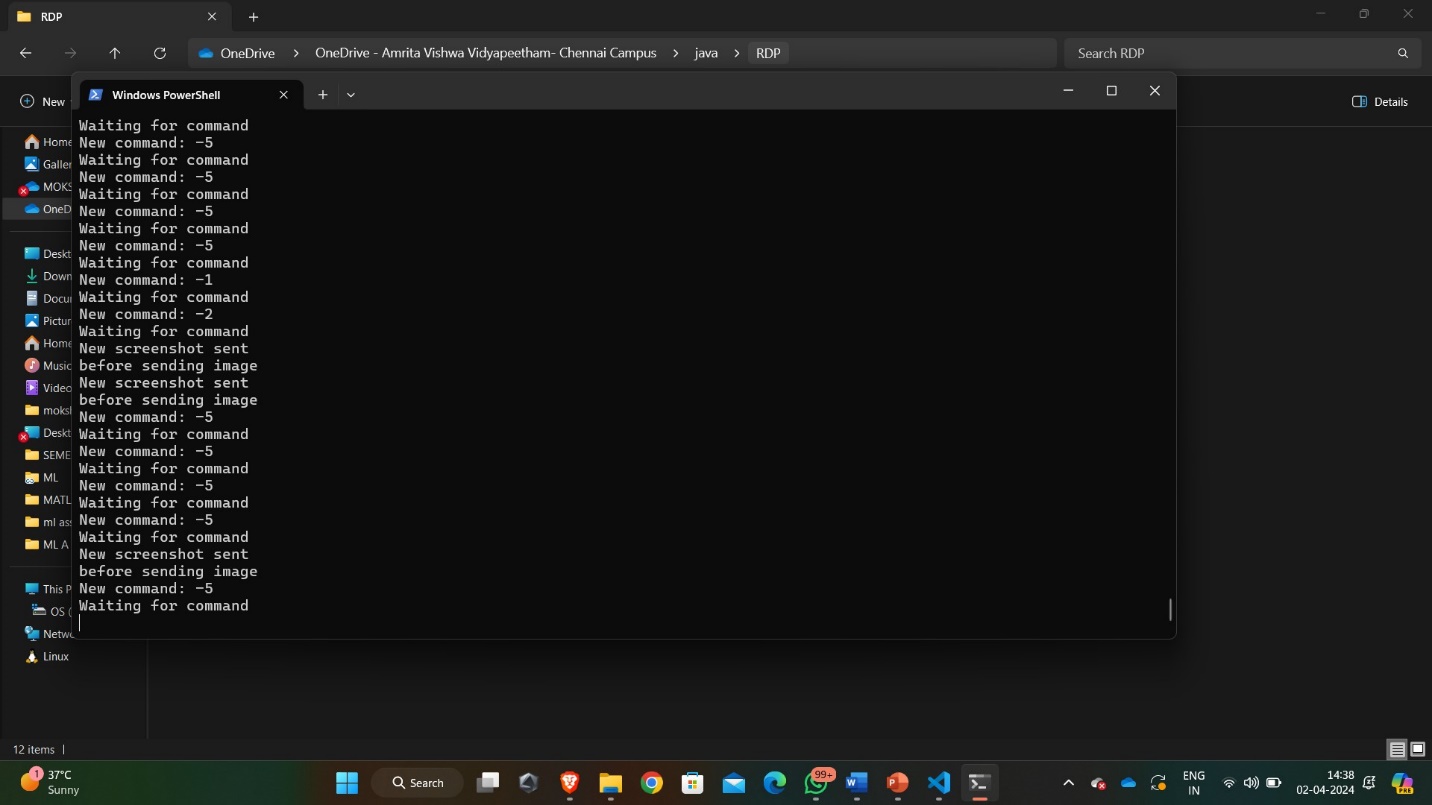
        writer.flush();

**Results/Output**

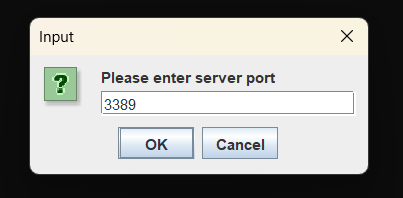
Client-Side:

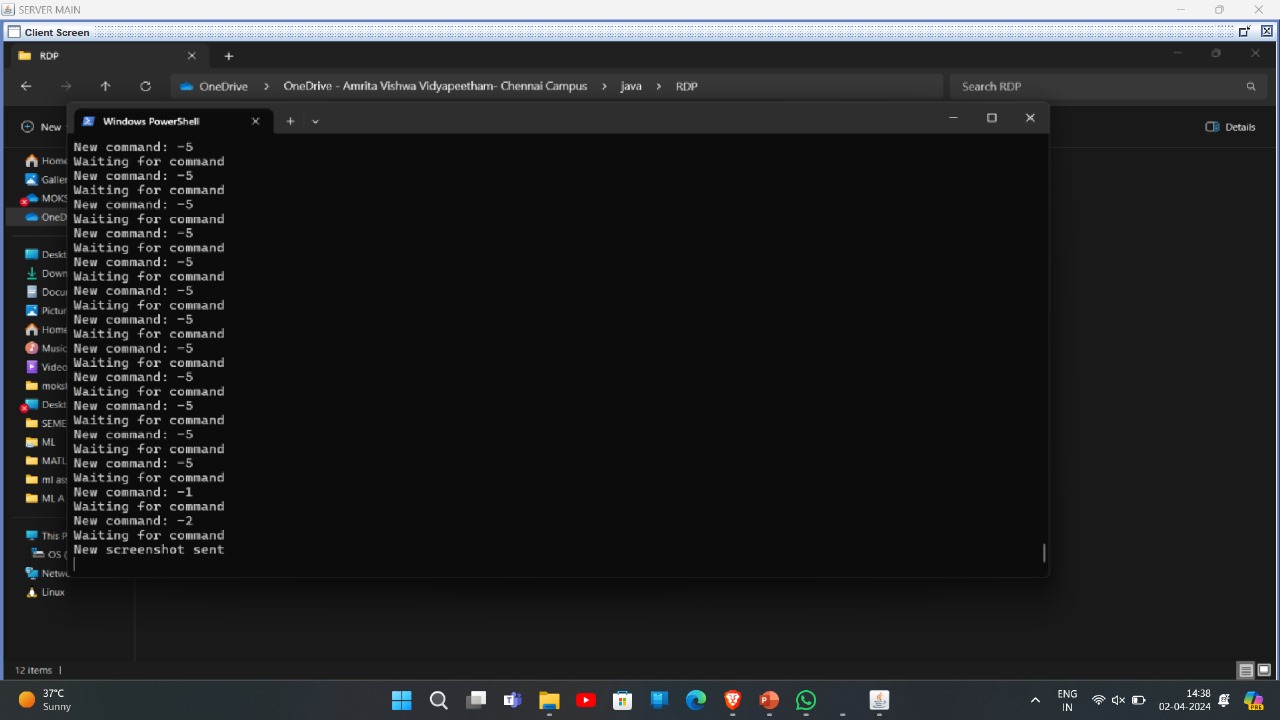






Server-side:





**Conclusion**

The development of Skylink, a Java-based RDP system, emphasizes OOP principles for seamless remote desktop access. Leveraging Java's platform independence and robustness, Skylink offers reliability for IT support and remote work scenarios. Careful consideration was given to requirements analysis, design, development, and testing, ensuring key features like secure connections and cross-platform compatibility. Skylink's intuitive client interface and robust server component enable secure session management and real-time interaction. By following a systematic approach, Skylink delivers a user-friendly, secure, and efficient remote desktop access solution, marking a significant milestone in remote desktop technology.

In conclusion, the Skylink Java project presents a robust and efficient solution for remote desktop control, employing a client-server model to facilitate seamless interaction between users and remote desktop environments. Through advanced features such as dynamic screen sharing, customization options, and comprehensive security measures, Skylink aims to deliver a user-friendly and secure platform for remote desktop access. With its emphasis on performance optimization, scalability, and integration with existing systems, Skylink offers a versatile solution suitable for various use cases, including IT support, remote work, and collaborative projects. Overall, Skylink stands as a reliable and adaptable tool, providing users with the means to access and manage desktop environments remotely with ease and efficiency.