EEN1035 — OOP With Embedded Systems

Assignment: Java GUI-based Client/Server Sensor Aggregation

Date: 29th November 2024

Mahmood Albadaai

DCU ID: 20106386

<u>**Introduction:**</u>

This project focuses on creating a client-server application that enables real-time monitoring and communication between multiple devices and a central server. The main problem addressed is the need to efficiently transmit and visualize sensor data from various devices over a network.

By simulating a setup with three sensors, the Sound Level, CO2 Level, and Light Pollution Level the project demonstrates how devices can collect data and transmit it periodically to a server. The server, in turn, processes and displays the data using a graphical interface, providing aggregated and device-specific views.

<u>**Design:**</u>

This project is based on client-server architecture, designed to facilitate communication between multiple clients (representing sensor devices) and a central server. The primary goal is to simulate the collection, transmission, and visualization of sensor data in a structured and user-friendly way.

The client component represents a device equipped with three simulated sensors: Sound Level, CO2 Level, and Light Pollution Level. Users can adjust the sensor values manually through a graphical interface. Additionally, the client features options to apply noise adjustments to the data and periodically transmit the readings to the server every 5 seconds. This periodic transmission mimics real-world scenarios where data is sent in fixed time intervals, rather than continuously.

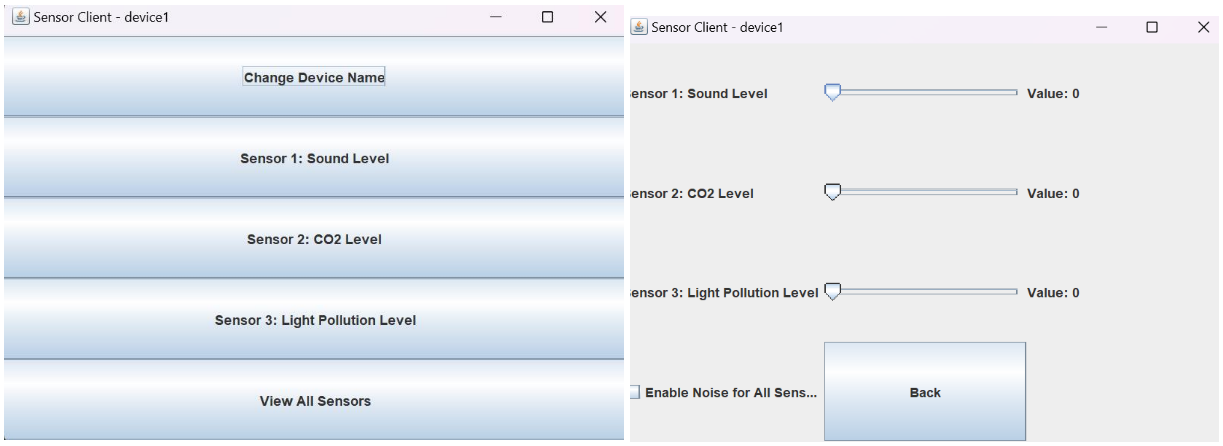


Figure 1: Client-Side GUI

The server acts as the central hub, receiving data from multiple clients and processing it for visualization. A graphical dashboard provides users with both device-specific and aggregated views of the sensor data. This enables the server to display minimum, maximum, and average values for each sensor type across all connected devices. To handle multiple client connections efficiently, the server uses multithreading, ensuring smooth operation even with concurrent communication.

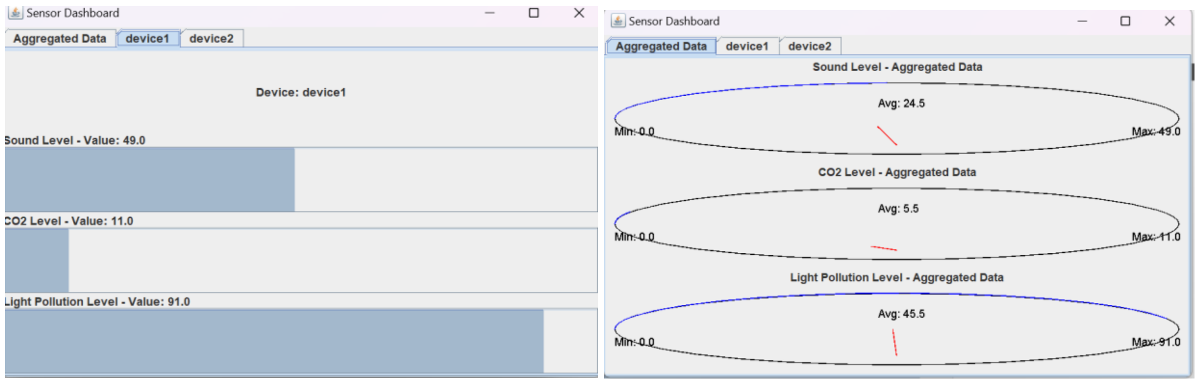


Figure 2: Server-Side GUI

The graphical user interface is a key aspect of the design, improving usability for both the client and server. On the client side, sliders simulate sensor values, and users can switch between views to adjust or monitor individual sensors. The server dashboard displays periodic updates, offering a clear visualization of the incoming data.

The system is designed to accommodate multiple clients while maintaining simplicity. The server enforces a maximum limit on connected devices to prevent overload and keep the project manageable. The implementation uses Java concepts, including socket programming for networking, multithreading for handling concurrent clients, object serialization for data exchange, and Swing for graphical user interfaces.

Implementation:

Server

The server serves as the core component of the system, designed to handle multiple client connections concurrently, process the incoming data, and visually display the sensor information using a GUI. Its implementation is structured into three main components: the ThreadedServer, the ThreadedConnectionHandler, and the SensorDashboard. Below is a detailed explanation of these components.

ThreadedServer

The ThreadedServer class manages the server's overall operation, from initializing the listening socket to handling client connections and integrating with the GUI.

1. Initialization and Listening for Connections

The server is initialized by creating a ServerSocket bound to a fixed port, 5050, allowing it to listen for client connections. The server operates in a loop, staying in a "listening" state to accept multiple connections. Each incoming connection is handled by creating a Socket object, representing the communication channel between the client and the server.

The server logs important connection details, such as the IP address and port number of the client. This mechanism keeps the server constantly available for new client connections.

2. Multithreading and Connection Handling

The server employs multithreading to manage client connections efficiently. When a client connects, the server creates an instance of ThreadedConnectionHandler and assigns the client's socket to it. This handler is executed as a separate thread, ensuring that each client's communication is independent and does not block the server from accepting new connections.

3. Integration with the GUI

The server is its integration with the SensorDashboard GUI. The server initializes a SensorDashboard instance upon startup, which acts as the visual interface for monitoring connected devices and their sensor data. The dashboard supports:

- Device Tabs**:** Each connected device has a dedicated tab displaying its sensor.
- Aggregated View: A consolidated display of all sensor data, showing minimum, maximum, and average values across devices.

The server provides synchronized access to the dashboard via the getDashboard() method, ensuring thread-safe updates from multiple ThreadedConnectionHandler instances. This integration bridges backend data processing with the frontend visualization.

SensorDashboard

The SensorDashboard is the GUI interface of the server, designed to provide a user-friendly and dynamic visualization of sensor data.

1. Device Management

The dashboard dynamically creates tabs for each connected device. Each tab, managed by the DevicePanel class, displays the device's sensor readings. The dashboard supports up to three devices simultaneously, ensuring manageable complexity. If a device reconnects with the same name, its tab is reused to maintain data continuity and avoid duplication.

2. Aggregated View

The AggregatedPanel consolidates data from all devices, calculating minimum, maximum, and average values for each sensor type. The visual representation of this data is managed by custom gauges, performed dynamically. These gauges provide an intuitive view of overall system performance, enabling quick identification of trends.

Client

The client is designed to act as a sensor device, responsible for generating and transmitting simulated sensor data to the server while providing a dynamic and interactive GUI for the user. It is a critical component in the client-server architecture, enabling the server to collect and visualize sensor data.

Initialization and Connection to the Server

The client begins by initializing a connection to the server using a specified IP address and port (5050). The connectToServer method establishes the connection via a Socket object and initializes ObjectOutputStream for data transmission. This ensures a reliable communication channel with the server. If the connection fails, the application gracefully exits, notifying the user of the failure.

Upon successfully connecting, the client sends an initial message to the server using the custom SensorMessage class, encapsulating the device name and placeholder sensor data. This step registers the device with the server, enabling the server to track incoming data from this specific client.

Graphical User Interface (GUI)

The client features a GUI implemented using Swing components. The setupMainWindow method initializes the main frame and organizes the interface into a card-based layout (CardLayout) for easy navigation between different views. The main interface consists of buttons that allow the user to:

- Change the device name.
- View and adjust individual sensor data (e.g., Sound Level, CO2 Level, Light Pollution Level).
- Access an aggregated view displaying all sensor readings.

The GUI is dynamically updated as the user interacts with various components.

Individual Sensor Views

For each sensor, a dedicated view allows the user to adjust simulated sensor values using a slider (JSlider). A checkbox enables adding random noise to the sensor values, mimicking real-world variability. The showSensorView method manages these views, allowing users to modify data for specific sensors. Real-time changes are displayed using a JLabel component, and the updated values are stored for subsequent transmission to the server. This modular design encapsulates sensor-specific functionality.

Aggregated Sensor View

An aggregated view consolidates all sensor values into a single interface. It features sliders and labels for each sensor, allowing users to monitor and adjust multiple sensors simultaneously. The inclusion of a "noise mode" toggle applies variability across all sensors. This view is managed by the showAllSensorsView method, which ensures consistent and synchronized updates for all sensors.

Periodic Data Transmission

This ensures that sensor data is sent to the server every five seconds. The startPeriodicDataSend method creates a dedicated thread that continuously transmits the latest sensor data to the server. This method uses the SensorMessage class to encapsulate the device name, sensor type, and value, ensuring consistent and structured communication. The use of a separate thread ensures that the GUI remains responsive while the data transmission occurs in the background.

Communication

The communication system is very important for this project, enabling interaction between the client and the server. It ensures that sensor data is transmitted reliably and processed correctly. This section outlines how communication is established, managed, and secured between the client and server components.

Communication Protocol

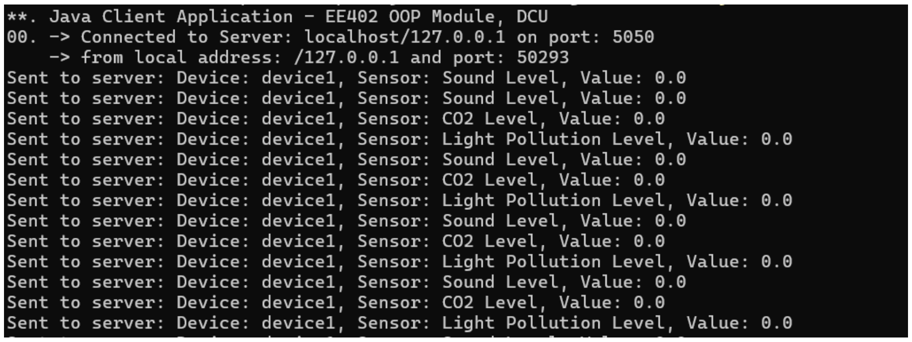
The client-server communication is based on a socket-based protocol, which ensures a persistent connection for transmitting serialized objects. This design is both lightweight and robust, making it suitable for this kind of application. The Socket class is used on both the client and server sides to facilitate this communication.

Each message exchanged between the client and server is encapsulated using the custom SensorMessage class. This class is marked as Serializable, enabling the conversion of objects into a stream of bytes for transmission over the network. This structured messaging system ensures that both ends interpret the data consistently.

Client-Side Communication

On the client side, the communication begins with the initialization of a socket connection to the server. The connectToServer method in the Client class establishes this connection using the server's IP address and port number. Once connected, an ObjectOutputStream is created to send serialized SensorMessage objects to the server.

Periodic data transmission is managed by the startPeriodicDataSend method, which runs in a separate thread. Every five seconds, the client packages the latest sensor data into SensorMessage objects and sends them to the server. This mechanism ensures that the server receives a steady stream of data, simulating continuous sensor updates. Each SensorMessage includes the device name, sensor type, and sensor value, maintaining a consistent data format.



```
**. Java Client Application - EE402 OOP Module, DCU
00. -> Connected to Server: localhost/127.0.0.1 on port: 5050
    -> from local address: /127.0.0.1 and port: 50293
Sent to server: Device: device1, Sensor: Sound Level, Value: 0.0
Sent to server: Device: device1, Sensor: Sound Level, Value: 0.0
Sent to server: Device: device1, Sensor: CO2 Level, Value: 0.0
Sent to server: Device: device1, Sensor: Light Pollution Level, Value: 0.0
Sent to server: Device: device1, Sensor: Sound Level, Value: 0.0
Sent to server: Device: device1, Sensor: CO2 Level, Value: 0.0
Sent to server: Device: device1, Sensor: Light Pollution Level, Value: 0.0
Sent to server: Device: device1, Sensor: Sound Level, Value: 0.0
Sent to server: Device: device1, Sensor: CO2 Level, Value: 0.0
Sent to server: Device: device1, Sensor: Light Pollution Level, Value: 0.0
Sent to server: Device: device1, Sensor: Sound Level, Value: 0.0
Sent to server: Device: device1, Sensor: CO2 Level, Value: 0.0
Sent to server: Device: device1, Sensor: Light Pollution Level, Value: 0.0
```

Figure 3: The Client Output Data

Server-Side Communication

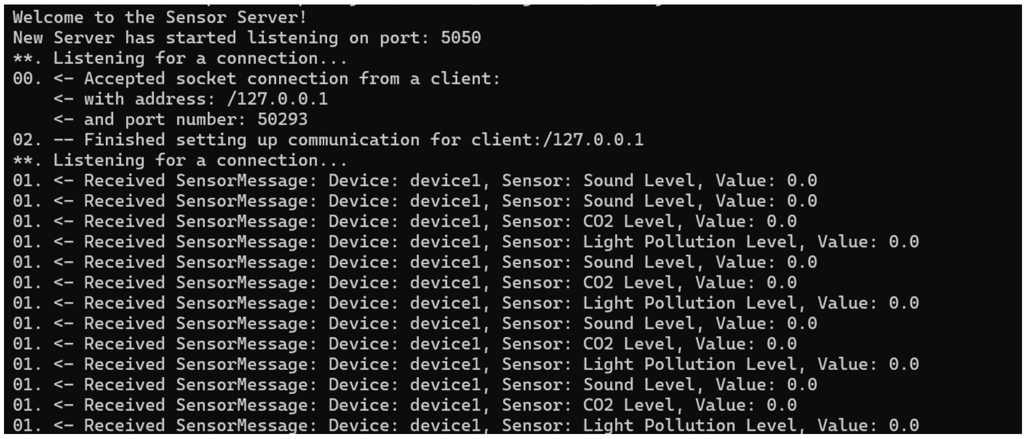
On the server side, communication is managed by the ThreadedServer and ThreadedConnectionHandler classes. The ThreadedServer listens to incoming client connections using a ServerSocket. When a client connects, a new ThreadedConnectionHandler instance is created to manage communication with that specific client. This multithreaded design allows the server to handle multiple clients simultaneously without blocking other connections.

Each ThreadedConnectionHandler operates independently, using an ObjectInputStream to receive serialized objects from the client. The readCommand method processes each received object, ensuring it is a valid SensorMessage. If the message is valid, the handler extracts the device name, sensor type, and sensor value and updates the SensorDashboard accordingly. If the message is invalid, the server logs an error and sends a descriptive error message back to the client.

Message Format and Structure

The SensorMessage class is the foundation of the communication system. It encapsulates:

- Device Name: Identifies the device sending the data.
- Sensor Type: Specifies the type of sensor (e.g., Sound Level, CO2 Level).
- Sensor Value: Contains the actual sensor reading.



```
Welcome to the Sensor Server!
New Server has started listening on port: 5050
**. Listening for a connection...
00. <- Accepted socket connection from a client:
    <- with address: /127.0.0.1
    <- and port number: 50293
02. -- Finished setting up communication for client:/127.0.0.1
**. Listening for a connection...
01. <- Received SensorMessage: Device: device1, Sensor: Sound Level, Value: 0.0
01. <- Received SensorMessage: Device: device1, Sensor: Sound Level, Value: 0.0
01. <- Received SensorMessage: Device: device1, Sensor: CO2 Level, Value: 0.0
01. <- Received SensorMessage: Device: device1, Sensor: Light Pollution Level, Value: 0.0
01. <- Received SensorMessage: Device: device1, Sensor: Sound Level, Value: 0.0
01. <- Received SensorMessage: Device: device1, Sensor: CO2 Level, Value: 0.0
01. <- Received SensorMessage: Device: device1, Sensor: Light Pollution Level, Value: 0.0
01. <- Received SensorMessage: Device: device1, Sensor: Sound Level, Value: 0.0
01. <- Received SensorMessage: Device: device1, Sensor: CO2 Level, Value: 0.0
01. <- Received SensorMessage: Device: device1, Sensor: Light Pollution Level, Value: 0.0
01. <- Received SensorMessage: Device: device1, Sensor: Sound Level, Value: 0.0
01. <- Received SensorMessage: Device: device1, Sensor: CO2 Level, Value: 0.0
01. <- Received SensorMessage: Device: device1, Sensor: Light Pollution Level, Value: 0.0
```

Figure 4: The Server Input Data

This consistent message structure simplifies data processing on the server. By serializing the SensorMessage objects, the project leverages Java's built-in mechanisms for efficient object transmission.

<u>Testing:</u>

The testing phase focused on validating functionality and reliability. Initial tests with a single client confirmed successful connections, periodic data transmission, and updates in the SensorDashboard. Sensor data for Sound Level, CO2 Level, and Light Pollution were accurately reflected in individual tabs and the aggregated view.

Tests with multiple clients demonstrated the server's ability to handle three simultaneous connections, update each device's data dynamically, and reflect consolidated sensor data in the aggregated view. When a fourth device attempted to connect, the server correctly refused the connection.

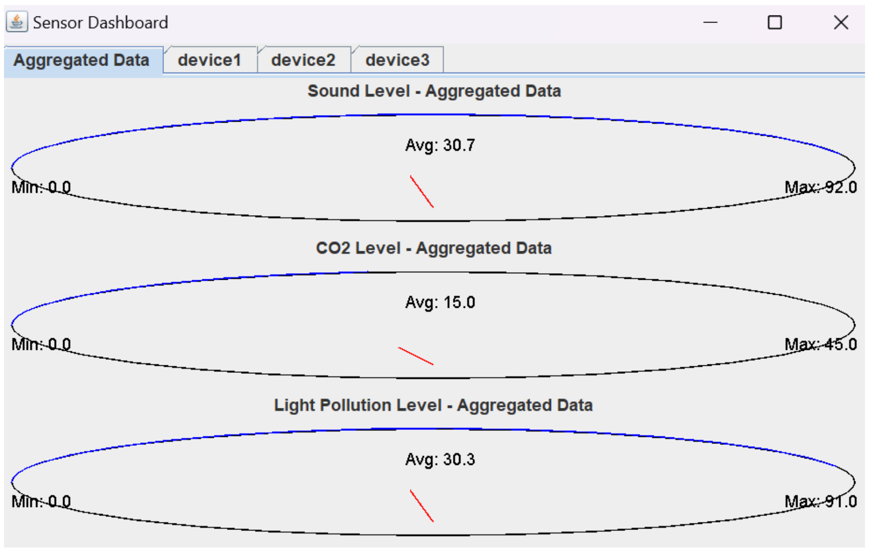


Figure 5: Show Case Of the ability For the Server to Handle 3 Devices

However, a limitation was observed when a client changed its device name mid-session. The server treated the renamed device as a new entity, creating a duplicate tab in the SensorDashboard.

The system's key limitations include its three-device connection cap, restricting scalability, and reliance on device names for identification, which causes issues when names are changed mid-session.

<u>Conclusion:</u>

This project offered valuable insights into client-server systems, multithreading, and real-time GUI development in Java. It successfully demonstrated robust communication between multiple clients and a server, with dynamic data visualization through the SensorDashboard. The use of the SensorMessage class simplified data exchange, and the system effectively managed simultaneous connections.

Key challenges included the three-device limit and the issue of duplicate entries when clients changed their names mid-session. Addressing these with dynamic scalability and unique identifiers would improve the system's robustness. Overall, the project provided practical experience in building a functional networked application, highlighting both strengths and areas for improvement.

Appendices:

Client.java

```java
package een1035;

import java.net.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.util.Random;

public class Client {
    private static int portNumber = 5050;
    private Socket socket = null;
    private ObjectOutputStream os = null;

    private String deviceName;

    private int soundLevelValue = 0;
    private int co2LevelValue = 0;
    private int lightPollutionValue = 0;

    private JFrame mainFrame;
    private JPanel cardPanel;
    private CardLayout cardLayout;

    private static final int TIME_WINDOW = 5; // Send data every 5 seconds

    // Constructor expects the IP address of the server and device name
    public Client(String serverIP, String deviceName) {
        this.deviceName = deviceName;
        if (!connectToServer(serverIP)) {
            System.out.println("XX. Failed to open socket connection to: " + serverIP);
            System.exit(1); // Exit if connection fails
        }

        // Initial message to the server
        sendToServer(new SensorMessage(deviceName, "Sound Level", 0));

        // Setup GUI
        setupMainWindow();

        // Start periodic data transmission
        startPeriodicDataSend();
    }

    private boolean connectToServer(String serverIP) {
        try {
            this.socket = new Socket(serverIP, portNumber);
            this.os = new ObjectOutputStream(this.socket.getOutputStream());
            System.out.println("00. -> Connected to Server: " + this.socket.getInetAddress()
                    + " on port: " + this.socket.getPort());
            System.out.println("    -> from local address: " + this.socket.getLocalAddress()
                    + " and port: " + this.socket.getLocalPort());
        } catch (Exception e) {
            System.out.println("XX. Failed to Connect to the Server at port: " + portNumber);
            System.out.println("    Exception: " + e.toString());
            return false;
        }
        return true;
    }

    private void setupMainWindow() {
        mainFrame = new JFrame("Sensor Client - " + deviceName);
        cardLayout = new CardLayout();
        cardPanel = new JPanel(cardLayout);

        // Main panel with buttons
        JPanel mainButtonPanel = new JPanel(new GridLayout(5, 1));

        JButton changeNameButton = new JButton("Change Device Name");
        changeNameButton.addActionListener(e -> {
            String newName = JOptionPane.showInputDialog(mainFrame, "Enter new device name:", deviceName);
            if (newName != null && !newName.trim().isEmpty()) {
                this.deviceName = newName.trim();
                mainFrame.setTitle("Sensor Client - " + deviceName);
                sendToServer(deviceName); // Inform the server of the name change
            }
        });

        JButton sensor1Button = new JButton("Sensor 1: Sound Level");
        sensor1Button.addActionListener(e -> showSensorView("Sound Level", 0, 100, value -> soundLevelValue = value));

        JButton sensor2Button = new JButton("Sensor 2: CO2 Level");
        sensor2Button.addActionListener(e -> showSensorView("CO2 Level", 0, 50, value -> co2LevelValue = value));

        JButton sensor3Button = new JButton("Sensor 3: Light Pollution Level");
        sensor3Button.addActionListener(e -> showSensorView("Light Pollution Level", 0, 100, value -> lightPollutionValue = value));

        JButton allSensorsButton = new JButton("View All Sensors");
        allSensorsButton.addActionListener(e -> showAllSensorsView());

        mainButtonPanel.add(changeNameButton);
        mainButtonPanel.add(sensor1Button);
        mainButtonPanel.add(sensor2Button);
        mainButtonPanel.add(sensor3Button);
        mainButtonPanel.add(allSensorsButton);

        cardPanel.add(mainButtonPanel, "Main");

        // Add cardPanel to mainFrame
        mainFrame.add(cardPanel);
        mainFrame.setSize(400, 300);
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainFrame.setVisible(true);
    }

    private void showSensorView(String sensorName, int min, int max, ValueUpdater valueUpdater) {
        JPanel sensorPanel = new JPanel(new GridLayout(4, 1));
        JLabel sensorLabel = new JLabel(sensorName);
        JSlider sensorSlider = new JSlider(min, max, (min + max) / 2);
        JLabel sensorValue = new JLabel("Value: " + sensorSlider.getValue());
        JCheckBox noiseCheckBox = new JCheckBox("Enable Noise");
        Random random = new Random();

        sensorSlider.addChangeListener(e -> {
            int baseValue = sensorSlider.getValue();
            int finalValue = baseValue;

            if (noiseCheckBox.isSelected()) {
                int noise = random.nextInt(5) - 2; // Random adjustment between -2 and 2
                finalValue = Math.max(min, Math.min(max, baseValue + noise));
            }

            sensorValue.setText("Value: " + finalValue);
            valueUpdater.updateValue(finalValue);
        });

        JButton backButton = new JButton("Back");
        backButton.addActionListener(e -> cardLayout.show(cardPanel, "Main"));

        sensorPanel.add(sensorLabel);
        sensorPanel.add(sensorSlider);
        sensorPanel.add(sensorValue);
        sensorPanel.add(noiseCheckBox);
        sensorPanel.add(backButton);

        cardPanel.add(sensorPanel, "SensorView");
        cardLayout.show(cardPanel, "SensorView");
    }

    private void showAllSensorsView() {
        JPanel allSensorsPanel = new JPanel(new GridLayout(4, 3));
        JCheckBox noiseModeCheckBox = new JCheckBox("Enable Noise for All Sensors");
        Random random = new Random();

        JLabel sensor1Label = new JLabel("Sensor 1: Sound Level");
        JSlider sensor1Slider = new JSlider(0, 100, soundLevelValue);
        JLabel sensor1Value = new JLabel("Value: " + soundLevelValue);

        sensor1Slider.addChangeListener(e -> updateSensorValue("Sound Level", sensor1Slider, sensor1Value, noiseModeCheckBox, random, value -> soundLevelValue = value));

        JLabel sensor2Label = new JLabel("Sensor 2: CO2 Level");
        JSlider sensor2Slider = new JSlider(0, 50, co2LevelValue);
        JLabel sensor2Value = new JLabel("Value: " + co2LevelValue);

        sensor2Slider.addChangeListener(e -> updateSensorValue("CO2 Level", sensor2Slider, sensor2Value, noiseModeCheckBox, random, value -> co2LevelValue = value));

        JLabel sensor3Label = new JLabel("Sensor 3: Light Pollution Level");
        JSlider sensor3Slider = new JSlider(0, 100, lightPollutionValue);
        JLabel sensor3Value = new JLabel("Value: " + lightPollutionValue);

        sensor3Slider.addChangeListener(e -> updateSensorValue("Light Pollution Level", sensor3Slider, sensor3Value, noiseModeCheckBox, random, value -> lightPollutionValue = value));

        allSensorsPanel.add(sensor1Label);
        allSensorsPanel.add(sensor1Slider);
        allSensorsPanel.add(sensor1Value);
        allSensorsPanel.add(sensor2Label);
        allSensorsPanel.add(sensor2Slider);
        allSensorsPanel.add(sensor2Value);
        allSensorsPanel.add(sensor3Label);
        allSensorsPanel.add(sensor3Slider);
        allSensorsPanel.add(sensor3Value);
        allSensorsPanel.add(noiseModeCheckBox);

        JButton backButton = new JButton("Back");
        backButton.addActionListener(e -> cardLayout.show(cardPanel, "Main"));

        allSensorsPanel.add(backButton);

        cardPanel.add(allSensorsPanel, "AllSensors");
        cardLayout.show(cardPanel, "AllSensors");
    }

    private void updateSensorValue(String sensorName, JSlider slider, JLabel label, JCheckBox noiseCheckBox, Random random, ValueUpdater valueUpdater) {
        int baseValue = slider.getValue();
        int finalValue = baseValue;

        if (noiseCheckBox.isSelected()) {
            int noise = random.nextInt(5) - 2;
            finalValue = Math.max(slider.getMinimum(), Math.min(slider.getMaximum(), baseValue + noise));
        }

        label.setText("Value: " + finalValue);
        valueUpdater.updateValue(finalValue);
    }

    private void sendToServer(Object message) {
        try {
            if (os != null) {
                os.writeObject(message);
                os.flush();
                System.out.println("Sent to server: " + message);
            }
        } catch (IOException e) {
            System.out.println("XX. Error sending data to server: " + e.getMessage());
        }
    }

    private void startPeriodicDataSend() {
        Thread transmissionThread = new Thread(() -> {
            while (true) {
                try {
                    sendToServer(new SensorMessage(deviceName, "Sound Level", soundLevelValue));
                    sendToServer(new SensorMessage(deviceName, "CO2 Level", co2LevelValue));
                    sendToServer(new SensorMessage(deviceName, "Light Pollution Level", lightPollutionValue));
                    Thread.sleep(TIME_WINDOW * 1000); // Wait for N seconds
                } catch (InterruptedException e) {
                    System.out.println("XX. Error in periodic data transmission: " + e.getMessage());
                }
            }
        });
        transmissionThread.start();
    }

    public static void main(String[] args) {
        System.out.println("**. Java Client Application - EE402 OOP Module, DCU");
        if (args.length == 2) {
            new Client(args[0], args[1]);
        } else {
            System.out.println("Error: you must provide the address of the server and device name");
            System.out.println("Usage is: java Client x.x.x.x <deviceName>");
        }
    }

    @FunctionalInterface
    interface ValueUpdater {
        void updateValue(int value);
    }
}
```

ThreadedServer.java

```java
package een1035;

import java.net.*;
import java.io.*;

public class ThreadedServer {
    private static int portNumber = 5050;
    private SensorDashboard dashboard;

    public static void main(String args[]) {
        ThreadedServer server = new ThreadedServer();
        server.startServer();
    }

    public ThreadedServer() {
        // Initialize the dashboard GUI
        dashboard = new SensorDashboard();
        System.out.println("Welcome to the Sensor Server!");
    }

    public void startServer() {
        boolean listening = true;
        ServerSocket serverSocket = null;

        // Set up the Server Socket
        try {
            serverSocket = new ServerSocket(portNumber);
            System.out.println("New Server has started listening on port: " + portNumber);
        } catch (IOException e) {
            System.out.println("Cannot listen on port: " + portNumber + ", Exception: " + e);
            System.exit(1);
        }

        // Server is now listening for connections
        while (listening) {
            Socket clientSocket = null;
            try {
                System.out.println("**. Listening for a connection...");
                clientSocket = serverSocket.accept();
                System.out.println("00. <- Accepted socket connection from a client: ");
                System.out.println("    <- with address: " + clientSocket.getInetAddress().toString());
                System.out.println("    <- and port number: " + clientSocket.getPort());
            } catch (IOException e) {
                System.out.println("XX. Accept failed: " + portNumber + e);
                listening = false; // Stop listening for further client requests
            }

            if (clientSocket != null) {
                // Start a new thread to handle the client connection
                ThreadedConnectionHandler con = new ThreadedConnectionHandler(clientSocket, this);
                con.start();
                System.out.println("02. -- Finished setting up communication for client:"
                        + clientSocket.getInetAddress().toString());
            }
        }

        // Server is no longer listening for client connections - time to shut down.
        try {
            System.out.println("04. -- Closing down the server socket gracefully.");
            if (serverSocket != null) serverSocket.close();
        } catch (IOException e) {
            System.err.println("XX. Could not close server socket. " + e.getMessage());
        }
    }

    // Method to get the dashboard (used by connection handlers)
    public synchronized SensorDashboard getDashboard() {
        return dashboard;
    }
}
```

ThreadedConnectionHandler.java

```java
package een1035;

import java.net.*;
import java.io.*;

public class ThreadedConnectionHandler extends Thread {
    private Socket clientSocket = null; // Client socket object
    private ObjectInputStream is = null; // Input stream
    private ObjectOutputStream os = null; // Output stream
    private ThreadedServer server; // Reference to the main server
    private String currentDeviceName; // Track the current device name

    // Constructor for the connection handler
    public ThreadedConnectionHandler(Socket clientSocket, ThreadedServer server) {
        this.clientSocket = clientSocket;
        this.server = server;
    }

    // Thread execution method
    public void run() {
        try {
            this.is = new ObjectInputStream(clientSocket.getInputStream());
            //this.os = new ObjectOutputStream(clientSocket.getOutputStream());

            while (this.readCommand()) {
                // Continuously process client commands
            }
        } catch (IOException e) {
            System.out.println("XX. There was a problem with the Input/Output Communication:");
            e.printStackTrace();
        } finally {
            closeSocket(); // Ensure socket is closed on exit
        }
    }

    // Read and process incoming commands from the client
    private boolean readCommand() {
        try {
            Object receivedObject = is.readObject();

            // Check if the received object is a SensorMessage
            if (receivedObject instanceof SensorMessage) {
                SensorMessage message = (SensorMessage) receivedObject;
                System.out.println("01. <- Received SensorMessage: " + message);

                // Handle device name changes
                if (currentDeviceName == null ||
!currentDeviceName.equals(message.getDeviceName())) {
                    currentDeviceName = message.getDeviceName();
                    if (server != null) {
                        server.getDashboard().addDevice(currentDeviceName);
                    }
                }

                // Update the server dashboard with the new sensor data
                if (server != null) {
                    server.getDashboard().updateSensorData(
                            message.getDeviceName(),
                            message.getSensorType(),
                            message.getSensorValue()
                    );
                }
            } else {
                // Handle invalid objects
                System.out.println("XX. Unsupported object received: " + receivedObject);
                sendError("Unsupported object type received.");
            }
        } catch (Exception e) {
            System.out.println("XX. Client disconnected or communication error: " + e.getMessage());
            return false; // Stop processing if there's an error
        }
        return true; // Continue processing commands
    }

    // Send a generic object back to the client
    private void send(Object o) {
        try {
            System.out.println("02. -> Sending (" + o + ") to the client.");
            this.os.writeObject(o);
            this.os.flush();
        } catch (Exception e) {
            System.out.println("XX. Exception occurred while sending: " + e.getMessage());
        }
    }

    // Send a pre-formatted error message to the client
    public void sendError(String message) {
        this.send("Error: " + message); // A String IS-A Object
    }

    // Close the client socket
    public void closeSocket() {
        try {
            if (os != null) os.close();
            if (is != null) is.close();
            if (clientSocket != null) clientSocket.close();
        } catch (Exception e) {
            System.out.println("XX. Exception occurred while closing socket: " + e.getMessage());
        }
    }
}
```

SensorMessage.java

```java
package een1035;

import java.io.Serializable;

public class SensorMessage implements Serializable {
    private static final long serialVersionUID = 1L;

    private String deviceName;
    private String sensorType;
    private double sensorValue;

    public SensorMessage(String deviceName, String sensorType, double sensorValue) {
        this.deviceName = deviceName;
        this.sensorType = sensorType;
        this.sensorValue = sensorValue;
    }

    public String getDeviceName() {
        return deviceName;
    }

    public String getSensorType() {
        return sensorType;
    }

    public double getSensorValue() {
        return sensorValue;
    }

    @Override
    public String toString() {
        return "Device: " + deviceName + ", Sensor: " + sensorType + ", Value: " + sensorValue;
    }
}
```

SensorDashboard.java

```java
package een1035;

import javax.swing.*;
import java.awt.*;

public class SensorDashboard extends JFrame {
    private static final int MAX_DEVICES = 3;
    private String[] sensorTypes = {"Sound Level", "CO2 Level", "Light Pollution Level"};
    private JTabbedPane tabbedPane;
    private AggregatedPanel aggregatedPanel;
    private DevicePanel[] devicePanels;
    private int deviceCount = 0;

    public SensorDashboard() {
        devicePanels = new DevicePanel[MAX_DEVICES];
        aggregatedPanel = new AggregatedPanel(sensorTypes);
        setupGUI();
    }

    private void setupGUI() {
        setTitle("Sensor Dashboard");
        setLayout(new BorderLayout());

        tabbedPane = new JTabbedPane();

        JLabel waitingLabel = new JLabel("Waiting for devices to connect...", SwingConstants.CENTER);
        JPanel welcomePanel = new JPanel(new BorderLayout());
        welcomePanel.add(waitingLabel, BorderLayout.CENTER);
        tabbedPane.add("Welcome", welcomePanel);

        tabbedPane.add("Aggregated Data", aggregatedPanel);

        tabbedPane.setSelectedIndex(0);

        add(tabbedPane, BorderLayout.CENTER);

        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public synchronized void addDevice(String deviceName) {
        if (tabbedPane.getTabCount() > 1 && tabbedPane.getTitleAt(0).equals("Welcome")) {
            tabbedPane.remove(0);
        }

        for (int i = 0; i < deviceCount; i++) {
            if (devicePanels[i] != null && devicePanels[i].getDeviceName().equals(deviceName)) {
                return;
            }
        }

        if (deviceCount < MAX_DEVICES) {
            DevicePanel devicePanel = new DevicePanel(deviceName, sensorTypes);
            devicePanels[deviceCount] = devicePanel;
            tabbedPane.add(deviceName, devicePanel);
            deviceCount++;
        } else {
            System.out.println("XX. Maximum device limit reached. Cannot add more devices.");
        }
    }

    public synchronized void updateSensorData(String deviceName, String sensorType, double sensorValue) {
        for (int i = 0; i < deviceCount; i++) {
            if (devicePanels[i] != null && devicePanels[i].getDeviceName().equals(deviceName)) {
                devicePanels[i].updateSensorData(sensorType, sensorValue);
                break;
            }
        }

        aggregatedPanel.updateAggregatedData(devicePanels, deviceCount);

        SwingUtilities.invokeLater(() -> tabbedPane.repaint());
    }
}
```

AggregatedPanel.java

```java
package een1035;

import javax.swing.*;
import java.awt.*;

public class AggregatedPanel extends JPanel {
    private String[] sensorTypes;
    private GaugeData[] gauges;

    public AggregatedPanel(String[] sensorTypes) {
        this.sensorTypes = sensorTypes;
        setupGUI();
    }

    private void setupGUI() {
        setLayout(new GridLayout(sensorTypes.length, 1));

        gauges = new GaugeData[sensorTypes.length];

        for (int i = 0; i < sensorTypes.length; i++) {
            JPanel panel = new JPanel(new BorderLayout());
            JLabel titleLabel = new JLabel(sensorTypes[i] + " - Aggregated Data",
SwingConstants.CENTER);

            gauges[i] = new GaugeData(); // Represents data and rendering for each gauge
            gauges[i].setPreferredSize(new Dimension(200, 200));

            panel.add(titleLabel, BorderLayout.NORTH);
            panel.add(gauges[i], BorderLayout.CENTER);

            add(panel);
        }
    }

    public void updateAggregatedData(DevicePanel[] devices, int deviceCount) {
        for (int i = 0; i < sensorTypes.length; i++) {
            double min = Double.MAX_VALUE;
            double max = Double.MIN_VALUE;
            double total = 0.0;
            int count = 0;

            for (int j = 0; j < deviceCount; j++) {
                Double value = devices[j].getSensorValue(sensorTypes[i]);
                if (value != null) {
                    min = Math.min(min, value);
                    max = Math.max(max, value);
                    total += value;
                    count++;
                }
            }

            double avg = count > 0 ? total / count : 0.0;

            if (gauges[i] != null) {
                gauges[i].updateValues(count > 0 ? min : 0.0, count > 0 ? max : 0.0, avg);
            }
        }

        SwingUtilities.invokeLater(this::repaint);
    }





    private class GaugeData extends Canvas {
        private double minValue = 0.0;
        private double maxValue = 0.0;
        private double avgValue = 0.0;

        public void updateValues(double minValue, double maxValue, double avgValue) {
            this.minValue = minValue;
            this.maxValue = maxValue;
            this.avgValue = avgValue;
            repaint();
        }

        @Override
        public void paint(Graphics g) {
            Graphics2D g2d = (Graphics2D) g;
            int width = getWidth();
            int height = getHeight();

            // Draw outer circle
            g2d.setColor(Color.BLACK);
            g2d.drawOval(10, 10, width - 20, height - 20);

            // Draw min and max arcs
            g2d.setColor(Color.BLUE);
            int minAngle = (int) (180 * (minValue / 100));
            int maxAngle = (int) (180 * (maxValue / 100));
            g2d.drawArc(10, 10, width - 20, height - 20, 180 - maxAngle, maxAngle - minAngle);

            // Draw avg needle
            g2d.setColor(Color.RED);
            int avgAngle = (int) (180 * (avgValue / 100));
            int centerX = width / 2;
            int centerY = height - 20;
            int needleLength = Math.min(width, height) / 2 - 20;

            int needleX = centerX + (int) (Math.cos(Math.toRadians(180 - avgAngle)) * needleLength);
            int needleY = centerY - (int) (Math.sin(Math.toRadians(180 - avgAngle)) * needleLength);

            g2d.drawLine(centerX, centerY, needleX, needleY);

            // Labels for min, max, and avg
            g2d.setColor(Color.BLACK);
            g2d.drawString("Min: " + String.format("%.1f", minValue), 10, height - 30);
            g2d.drawString("Max: " + String.format("%.1f", maxValue), width - 60, height - 30);
            g2d.drawString("Avg: " + String.format("%.1f", avgValue), centerX - 20, centerY - 40);
        }
    }
}
```

DevicePanel.java

```java
package een1035;

import javax.swing.*;
import java.awt.*;

public class DevicePanel extends JPanel {
    private String deviceName;
    private String[] sensorTypes;
    private JLabel[] sensorLabels;
    private JProgressBar[] gaugeBars;

    public DevicePanel(String deviceName, String[] sensorTypes) {
        this.deviceName = deviceName;
        this.sensorTypes = sensorTypes;

        setupGUI();
    }

    private void setupGUI() {
        setLayout(new GridLayout(sensorTypes.length + 1, 1));

        JLabel titleLabel = new JLabel("Device: " + deviceName, SwingConstants.CENTER);
        add(titleLabel);

        sensorLabels = new JLabel[sensorTypes.length];
        gaugeBars = new JProgressBar[sensorTypes.length];

        for (int i = 0; i < sensorTypes.length; i++) {
            JPanel panel = new JPanel(new BorderLayout());
            sensorLabels[i] = new JLabel(sensorTypes[i] + " - Value: 0.0");
            gaugeBars[i] = new JProgressBar(0, 100);
            gaugeBars[i].setValue(0);

            panel.add(sensorLabels[i], BorderLayout.NORTH);
            panel.add(gaugeBars[i], BorderLayout.CENTER);
            add(panel);
        }
    }

    public String getDeviceName() {
        return deviceName;
    }

    public void updateSensorData(String sensorType, double value) {
        for (int i = 0; i < sensorTypes.length; i++) {
            if (sensorTypes[i].equalsIgnoreCase(sensorType)) {
                sensorLabels[i].setText(sensorTypes[i] + " - Value: " + value);
                gaugeBars[i].setValue((int) value);
                break;
            }
        }
    }

    public Double getSensorValue(String sensorType) {
        for (int i = 0; i < sensorTypes.length; i++) {
            if (sensorTypes[i].equalsIgnoreCase(sensorType)) {
                return Double.valueOf(gaugeBars[i].getValue());
            }
        }
        return null;
    }
```