EEN1035 — OOP With Embedded Systems Assignment: Java GUI-based Client/Server Sensor Aggregation

Name: Mohammed AL shuaili

ID: 20106181

Date: 26/11/2024

Introduction:

This assignment involves creating a Java-based client/server application to simulate and manage environmental sensor data. The application will consist of multiple client devices, represented by GUI applications, sending simulated sensor data to a central server. The server, also GUI-based, will aggregate, analyze, and display the data using graphical representations, such as dials and charts. The simulation includes three chosen sensor types (temperature, humidity, sound levels), with clients providing updates every five seconds.

The objective is to implement this system while following specific design constraints and functional requirements, including multithreading, object serialization for communication, and data aggregation. Additional goals include implementing innovative features, maintaining user-friendly interfaces, and demonstrating testing with multiple simulated client connections.

Design:

The client interface is designed with three sliders and corresponding textboxes, each representing a sensor type (e.g., temperature, sound level, and humidity level) displayed as a percentage out of 100. A checkbox is included to simulate real-life signals; when selected, it adds random noise to the recorded sensor data before sending it to the server.

Additionally, the interface features two buttons: one for connecting to the server and another for editing the device (client) name. When the server is running and the client successfully connects, the "Connect" button changes to green to indicate successful communication. The client then sends the three sensor values as objects to the server every five seconds. The application requires two command-line arguments before execution: one for the server's IP address and another for the client's name. Figures 1 and 2 illustrate the interface's design.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Figure 1 Client GUI after it's connected

Figure 2 Client interface before it's connected

The server GUI features a list displaying all connected clients along with an entry for the overall averages of all connected devices. It includes a button, similar to the client-side, for starting the server. When the server is actively listening for incoming connections, the button turns green. Selecting a client from the list displays the client’s name and shows graphical gauges for the three sensor readings, each representing the average values of that client.

A screenshot of a computer

Description automatically generated Additionally, the interface has three checkboxes, one for each sensor type. When a checkbox is selected, a plot of the last 10 readings for that sensor is displayed, with each sensor represented by a different color. The "average" option in the client list displays the overall average of all connected clients’ sensor values, as well as the average readings for each sensor, which are used for plotting. See Figures 3 and 4 for a visual representation.

Figure 3 Server GUI Showing Client List and Default State (No Data Plotted)

A screenshot of a computer

Description automatically generated

Figure 4 Server GUI Displaying Selected Client's Sensor Averages and Sensor Data Plots with Active Connections

Implementation:

* Client

The client implementation provides a GUI application for simulating environmental sensors and facilitating communication with the server. Built using Java Swing, the client interface is designed to be user-friendly, functional, and adaptable.

Key features of the client implementation include:

1. Graphical User Interface (GUI):
   * Sliders and textboxes to represent three sensor types: temperature, sound level, and humidity. Users can adjust the sliders, which automatically update the corresponding textbox, ensuring synchronization between components.
   * A checkbox is provided for real-life simulation, introducing random noise to the sensor readings to simulate real-world variations. This checkbox enables dynamic toggling of the noise features.
   * Two buttons are included: one to establish a connection with the server ("Connect") and another to dynamically edit the device name. The interface provides real-time feedback to users, such as changing the "Connect" button to green and updating its label to "Connected" when communication is successful.
2. Dynamic Sensor Value Input:
   * The textboxes allow direct input for sensor values as an alternative to adjusting sliders. The implementation ensures only numeric values between 0 and 100 are accepted, as the sensor readings are expressed as percentage levels.
   * If users input values exceeding this range, the program automatically resets the values to the nearest valid limit (0 or 100). This prevents erroneous data from being sent to the server, maintaining the integrity of the transmitted information.
3. Threaded Communication:
   * To ensure continuous and asynchronous updates, the client runs a background thread that sends sensor data to the server every five seconds (5000 ms). This design ensures the GUI remains responsive and user-friendly while the client communicates with the server in real time.
4. Device Name Customization:
   * Users can edit the device name dynamically through a pop-up dialog box, enabling better differentiation of clients in multi-client setups. The new name is immediately updated in the GUI and included in subsequent data transmissions.

**Communication**

The client communicates with the server using a socket-based connection. On pressing the "Connect" button, the client establishes a connection to the server at the specified IP address and port. It uses ObjectOutputStream to serialize and send SensorObject instances to the server, encapsulating the device name, sensor readings, and connection status.

The "Connect" button also allows users to disconnect from the server at any time. If the button is pressed while already connected, it resets the connection and updates its label and colour, indicating the disconnection status. Users can reconnect at any point, ensuring flexibility and adaptability.

* Server

The server, implemented in the ThreadedServer class, serves as the central hub for receiving, processing, and visualising sensor data from multiple clients. It uses multithreaded architecture, where each client connection is handled by a separate thread (ThreadedConnectionHandler). The server's GUI provides real-time monitoring and control capabilities, including toggling sensor plots, viewing client data, and displaying averages.

Key features of the serverimplementation include:

1. Dynamic Client Management:
   * The server maintains a list of connected clients, dynamically updating the GUI to reflect incoming and disconnected clients.
   * Each client has a unique identifier (ID) based on its port number, enabling the server to associate received data with the correct client.
2. Data Visualisation:
   * The server uses a custom canvas (MyCanvas) to display sensor data as graphical plots. Users can select a specific client from the list or choose "Average" to view the average sensor values across all connected clients.
   * Toggle switches allow users to enable or disable plots for temperature, sound, and humidity, providing a flexible and customizable visualisation experience.
3. Control Features:
   * The server can be started and stopped dynamically using the "Run the Server" button. When running, the button turns green and updates its label to "Running".
4. Multithreading:
   * Each client connection is assigned a dedicated thread via the ThreadedConnectionHandler class. This ensures that the server can handle multiple clients simultaneously without blocking operations.

**Connection Handler**

The ThreadedConnectionHandler class is responsible for managing the communication between the server and each individual client. It receives serialized SensorObject instances, processes the data, and updates the server's data structures and GUI accordingly.

Key features of the connection handler implementation include:

1. Sensor Data Processing:
   * The connection handler receives SensorObject instances from clients. Each object contains the client's name, sensor values, and connection status.
   * Sensor data is stored in a stack associated with the client’s unique ID. This stack allows the server to maintain a history of up to 10 recent data points for each client, enabling moving averages and historical analysis.
2. Dynamic Stack Management:
   * The findOrCreateStack method locates the stack corresponding to a client’s ID or creates a new one if no such stack exists (new client). This ensures that the server can handle new and existing clients and store all values associated with them.
   * When a client disconnects, its stack is removed, and the server GUI updates to reflect the change (the name is set to “Disconnected”).
3. Real-Time GUI Updates:
   * When the connection handler receives a SensorObject, it checks if the data corresponds to the currently selected client in the server's GUI. If it does, the plot is updated immediately to reflect the new data. Otherwise, the data is stored in the appropriate stack for later retrieval or processing.
4. Graceful Disconnection:
   * When a client disconnects (indicated by its status in the SensorObject), the connection handler closes the socket and removes the client’s data from the server’s data structures. If the disconnected client was selected in the GUI, the plot resets to reflect the disconnection.

**Communication**

Communication between the server and clients is facilitated through Java sockets. The server listens for incoming connections on a specified port (5050) and spawns a new ThreadedConnectionHandler for each connection. The handler uses an ObjectInputStream to receive serialized SensorObject instances from the client.

The SensorObject structure ensures that all transmitted data is well-organized and easy to process. Each SensorObject contains an ID set by the server to uniquely identify the client.

**Achievements**

* Real-Time Data Handling:
  + The server processes and visualises incoming sensor data in real time, ensuring that users have up-to-date information at all times.
* Scalable Architecture:
  + The multithreaded design allows the server to handle multiple clients simultaneously, with independent data stacks for each client.
* Disconnection Handling:
  + Clients can disconnect gracefully, with their data removed from the server’s GUI and data structures.

**Novel Features**

* Dynamic Client Visualisation:
  + Users can view data from individual clients or system-wide averages. This flexibility provides insights into both client-specific and overall performance.
* Historical Data Management:
  + The use of stacks for sensor data enables the server to maintain a history of the 10 most recent data points for each client. This supports moving averages and trend analysis.
* Interactive Sensor Selection:
  + Toggle switches allow users to customize the displayed plots by selecting which sensors (temperature, sound, humidity) to visualise.
* Dynamic Server Control:
  + The server can be started and stopped dynamically, with clear visual feedback indicating its status.
* Real-Time Input Validation:
  + Users can input sensor values directly into textboxes, but only numeric values between 0 and 100 are accepted. If users attempt to input invalid values (e.g., negative numbers, non-numeric characters, or values exceeding 100), the program resets these to the nearest valid range.
* Dynamic Connection Management:
  + The "Connect" button provides users with the ability to establish or terminate the connection with the server at any time. Disconnection is handled gracefully, with a final data packet sent to inform the server of the client’s status.
* Device Name Editing:
  + The ability to dynamically update the device name enhances usability in multi-client scenarios. The feature ensures that the server always receives the latest name in transmitted data packets.

Testing**:**

Describe any tests that you performed. Does it work correctly with multiple simultaneous clients? What are the limitations? Under what conditions does it fail?

Conclusions:

This assignment provided valuable experience in building a real-time client-server system, strengthening skills in network communication, multithreading, and GUI design. It was rewarding to implement features such as dynamic visualisation, synchronized sensor inputs, and error handling, which made the system interactive and practical. Key challenges included managing thread synchronization, handling client disconnections gracefully, and maintaining real-time responsiveness, all of them required careful debugging and design adjustments.

The modular structure of the system, with separate responsibilities for the client, server, and connection handler, worked well and facilitated development. However, the visualisation could be enhanced with more advanced graphs or charts, and the system could be scaled to support a larger number of clients more efficiently. Overall, this project was an excellent opportunity to apply theoretical concepts to a practical scenario, providing insights that will be valuable for future development tasks.

Appendices. Include the source code as an appendix (in the same form as Assignment 1-- i.e., label file names).