Project Status Report of Group 5

Week 3: Machine Learning Development and System Integration

Project: Lock-Hub: IoT-Based Centralized Control System Development

Members:
Escaña, Jylie Anne R.
Lagunilla, Genesis L.
Llabores, Rodrigo III D.

Enhanced Signal Processing

1.

1.1 Advanced Filtering Techniques Based on Fourier Analysis

Fourier-based analysis techniques were conceptually explored to enhance the interpretation of DHT22 sensor data. By decomposing time-series readings into their frequency components, this method can help identify underlying patterns or eliminate high-frequency noise. While temperature and signals are typically stable, frequency-domain analysis offers a systematic way to detect and suppress irregular fluctuations caused by environmental interference or sensor anomalies.

1.2 Real-Time Frequency Domain Processing

The ESP32 microcontroller has the capability to support real-time transformations such as Fast Fourier Transform (FFT) for small data sets. While not fully implemented in the current iteration, the software has been structured to support modular inclusion of FFT-based routines for identifying noise patterns in temperature/humidity readings or filtering out sensor glitches.

1.3 Implementation of Digital Filters Derived from Laplace Models

Digital approximations of Laplace-domain transfer functions were evaluated to simulate low-pass filter behavior directly in the microcontroller. Simple IIR (Infinite Impulse Response) filters have been proposed to suppress rapid, non-physical changes in sensor data. These can be incorporated to smooth DHT22 readings and make prediction logic (like the linear regression method used) more robust.

1.4 Performance Comparison Between Time and Frequency Domain Approaches

In test simulations, time-domain methods such as exponential smoothing and linear regression (already implemented) showed more consistent behavior under slowly changing environmental conditions.

Frequency-domain methods offer deeper insight into periodic or erratic data but are computationally heavier. For embedded systems like the ESP32, time-domain methods remain more efficient given resource constraints.

1.5 Optimization for Embedded Execution

All processing techniques were reviewed for compatibility with embedded execution. The current implementation uses lightweight regression-based forecasting, with potential for integrating digital filters. Optimizations such as reducing floating-point operations and limiting buffer sizes have been applied to ensure smooth, real-time performance on the ESP32.

Web Application Development

1. **Complete Backend Implementation with Database Integration**

   The backend was developed using the Arduino IDE, where the ESP32 microcontroller is programmed to handle sensor readings and relay commands. While full database integration is not yet implemented, the system is capable of transmitting temperature data and room occupancy status through Wi-Fi for real-time use.

2. **Frontend Development with Responsive Design**

   The frontend web interface was developed using the Processing software, designed to be visually intuitive and responsive. The interface supports real-time updates from the ESP32 and allows user commands to be sent back to control hardware components such as relays and locks. The design adapts well to various screen sizes for both desktop and mobile devices.

3. **Real-Time Data Visualization Components**

   Live temperature data from the DHT22 sensor is displayed through both the LCD and the web interface. Future improvements include adding graphical trend visualizations for better user interpretation of temperature changes in the monitored room.

4. **System Control Interface Implementation**

   The web interface supports system control commands such as marking rooms as occupied or unoccupied and toggling light or lock states. These commands are transmitted to the ESP32 over a local Wi-Fi connection and executed in real time, allowing users to actively manage the environment.

5. **API Documentation and Testing**

   Communication between the web interface and the ESP32 is handled through simple TCP-based commands. Preliminary API functions have been tested successfully, including commands like ROOM:101:OCCUPIED and LIGHT:101:ON. Full documentation of the command structure is being prepared to support future integration with mobile apps or other interfaces.

System Integration Progress

1. **Sensor-to-Actuator Subsystem Timeline**

   Step 1 – System Power On
   - ESP32 initializes relays, DHT22 sensor, and LCD display
   - Default states are set (relays OFF, LCD idle screen shown)

   Step 2 – Sensor Reads Data
   - ESP32 reads temperature from DHT22 every 5 seconds
   - Linear regression is applied to predict temperature trend
   - Data is checked for errors (sensor failure)

   Step 3 – Actuator Decision
   - If ROOM is marked OCCUPIED or command is received
   - Relays activate solenoid lock and bulb (based on control logic)

   Step 4 – Manual Reset Option
   - If reset button is pressed, relays turn OFF
   - LCD resets to default “ROOM 101” message

   Step 5 – Continuous Monitoring
   - Loop continues: sensor reads, logic evaluates, relays react accordingly

2. **Data Pipeline: Sensors → Processing → Web App Timeline**

   Step 1 – Start TCP Server
   - ESP32 connects to Wi-Fi and starts listening for client (Processing software)

   Step 2 – Sensor Data is Acquired
   - ESP32 reads and processes temperature from DHT22
   - Prediction is computed using numerical method

   Step 3 – Data Sent to Web App
   - ESP32 sends response strings (TEMP:27.4 or RESET:1:01) via TCP
   - Processing app receives and displays on-screen

Step 4 – Web App Sends Commands
- User interacts with web interface (click “Light ON”)
- Processing sends command to ESP32 (LIGHT:101:ON)

Step 5 – Actuators Respond
- ESP32 receives command and controls relay accordingly
- System updates are sent back for visual confirmation

Step 6 – Continuous Operation
- Communication loop continues for real-time control and monitoring

3. Performance Optimization for Real-Time Operation

To ensure responsiveness, key system tasks are handled with timing checks ensuring that sensor readings, button inputs, and command parsing happen efficiently. The use of lightweight string commands over TCP also contributes to faster communication with minimal overhead.

4. Error Handling and Recovery Mechanisms

Basic error-handling routines are in place, particularly for the DHT22 sensor, where invalid readings are detected using isnan() checks. In such cases, the system displays a sensor error message and avoids acting on unreliable data. Button debounce logic also prevents false triggers from mechanical noise.

5. Documentation of Integration Challenges and Solutions

During development, several integration challenges were encountered. For instance, initial sensor reading errors were solved by ensuring correct timing between reads. Relay misfires were addressed by standardizing logic LOW as “active” and HIGH as “inactive” states. The absence of real-time mobile notifications was noted as a limitation, pending future integration of additional communication modules or services. Overall, documentation has been maintained to track how sensor values trigger actuator responses and how these events are synchronized with the user interface.

Testing Framework

- **Unit tests for all major components:**

  Each hardware component (ESP32, DHT22, relays, push buttons, LCD) was tested individually. For example, the DHT22 sensor was tested using Serial Monitor outputs to confirm accurate temperature readings. Relay response was verified by manually toggling them via simple digitalWrite() commands in Arduino.

- **Integration test cases:**

  Integration testing involved simulating real scenarios, such as activating the

ROOM 101 relay through the Processing web interface and checking if the actuator responded. Commands like LIGHT:101:ON were sent from the app and validated in both hardware and serial output.

- **Performance benchmarking methodology:**
  We measured the system's reaction time from sensor read → processing → actuator output. For instance, temperature readings were benchmarked for stability and response rate (every 5 seconds), and predicted values were compared against real-time data using linear regression.

- **Automated testing implementation:**
  Full automation was limited due to the physical nature of the hardware. However, we scripted multiple command simulations from the Processing app (e.g., sending automated ROOM/OCCUPIED commands in sequence) to evaluate consistent response and error-free operation.

- **Testing documentation:**
  All test results, error logs, and observations were recorded manually. The documentation includes screenshots from the Processing app, Serial Monitor outputs, and step-by-step results from both hardware and software tests, compiled for review and debugging reference.