SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA 3rd semester
INTERNET OF THINGS (IoT) (PCA20D09J)

Lab Manual

Lab 1: Introduction to Eclipse IoT Project

Title: Lab 1: Introduction to Eclipse IoT Project

Aim: To understand the concept and significance of the Eclipse IoT Project.

Procedure:

1. Research the official Eclipse IoT website and documentation.
2. Define what the Eclipse IoT Project is and its overarching goals.
3. Explain its role in fostering open-source innovation and standardization within the IoT ecosystem.
4. Identify and describe the key benefits of using Eclipse IoT projects for developing IoT solutions (e.g., interoperability, community support, modularity).

Source Code: N/A (This lab focuses on theoretical understanding and research.)

Input: N/A

Expected Output: A comprehensive definition and explanation of the Eclipse IoT Project, including its mission, benefits, and general structure.

Lab 2: Overview of Eclipse IoT Projects

Title: Lab 2: Overview of Eclipse IoT Projects

Aim: To identify and summarize prominent projects within the Eclipse IoT ecosystem.

Procedure:

1. Explore the Eclipse IoT project list on their official website.
2. Select at least three distinct and significant Eclipse IoT projects (e.g., Mosquitto, Kura, Ditto, Hono, ioFog, etc.).
3. For each selected project, provide:
    - Its name and a brief description of its primary function.
    - The main technologies or protocols it utilizes.
    - Typical use cases or problems it aims to solve in the IoT domain.
4. Summarize the overall contribution of these projects to the broader IoT landscape.

Source Code: N/A (This lab focuses on theoretical understanding and research.)

Input: N/A

Expected Output: A list of at least three summarized Eclipse IoT projects, detailing their purpose, technologies, and use cases.

Lab 3: Smart Lighting

Title: Lab 3: Smart Lighting System

Aim: To conceptually design and simulate a basic smart lighting system, demonstrating automated control based on environmental conditions.

Procedure:

1. **System Design:**
   - Identify the core components: a light source (e.g., LED), a sensor (e.g., photoresistor for ambient light or PIR sensor for motion), and a microcontroller (e.g., Raspberry Pi, Arduino) acting as the controller.
   - Define the communication method between the sensor, controller, and light (e.g., direct GPIO connection, MQTT).
2. **Logic Definition:**
   - Establish a simple rule: if ambient light falls below a certain threshold, turn the light ON; otherwise, turn it OFF. (Alternatively: if motion is detected, turn light ON for a duration, then OFF).
3. **Conceptual Implementation (Pseudocode):** Write pseudocode that illustrates the control logic.

Source Code:

```
# Illustrative Pseudocode for Smart Lighting System (Python-like)

# --- Configuration ---
AMBIENT_LIGHT_THRESHOLD = 100  # Arbitrary value, lower means darker
LIGHT_PIN = 17                 # GPIO pin for the light
SENSOR_PIN = 4                 # GPIO pin for the ambient light sensor

# --- Initialize components (conceptual) ---
# setup_gpio(LIGHT_PIN, OUTPUT)
# setup_gpio(SENSOR_PIN, INPUT)

print("Smart Lighting System Initialized. Monitoring ambient light...")

# --- Main Loop ---
while True:
    # Read ambient light value from sensor (conceptual)
    # In a real system, this would involve ADC readings or sensor library
calls
    ambient_light_value = read_sensor(SENSOR_PIN)

    print(f"Current ambient light: {ambient_light_value}")

    if ambient_light_value < AMBIENT_LIGHT_THRESHOLD:
        # It's dark, turn the light on
        # write_gpio(LIGHT_PIN, HIGH)
        print("Ambient light is low. Turning light ON.")
    else:
        # It's bright enough, turn the light off
        # write_gpio(LIGHT_PIN, LOW)
        print("Ambient light is sufficient. Turning light OFF.")

    # Wait for a short period before checking again
    # sleep(5) # Simulate delay of 5 seconds
```

Input: Simulated ambient light values (e.g., `50` for dark, `200` for bright).

Expected Output:

```
Smart Lighting System Initialized. Monitoring ambient light...
Current ambient light: 50
Ambient light is low. Turning light ON.
Current ambient light: 200
Ambient light is sufficient. Turning light OFF.
... (repeats based on input)
```

Lab 4: Sketch the architecture of IoT

Title: Lab 4: IoT Architecture Sketch

Aim: To understand and illustrate the fundamental architectural layers of an IoT system.

Procedure:

1. **Research:** Study common IoT architectural models (e.g., 3-layer, 4-layer, 5-layer architectures).
2. **Identify Layers:** Identify the core layers present in most IoT systems. A common simplified model includes:
   - **Perception Layer (Device Layer):** Sensors, actuators, and smart objects that collect data from the environment or perform actions.
   - **Network Layer (Connectivity Layer):** Protocols and technologies responsible for data transmission from devices to the cloud/edge and vice-versa (e.g., Wi-Fi, Bluetooth, LoRaWAN, MQTT, CoAP, HTTP).
   - **Middleware/Processing Layer (Platform Layer):** Data aggregation, processing, storage, and management. This includes data filtering, analytics, and platform services.
   - **Application Layer:** User-facing applications and services that utilize the processed data to provide value (e.g., dashboards, mobile apps, business intelligence).
3. **Sketch/Describe:** Create a clear diagram or a detailed textual description outlining these layers and the flow of data between them. Emphasize the role of each layer.

Source Code: N/A (This lab focuses on conceptual understanding and diagramming.)

Input: N/A

Expected Output: A clear diagram (conceptual) and/or a detailed explanation of the typical IoT architectural layers, including the function of each layer and the data flow.

Lab 5: Demonstrate a smart object API gateway service reference implementation in IoT toolkit

Title: Lab 5: Smart Object API Gateway Service Reference Implementation

Aim: To conceptually demonstrate the role and function of a smart object API gateway in an IoT toolkit, focusing on how it exposes device functionality.

Procedure:

1. **Gateway Role Explanation:** Explain why an API gateway is essential in an IoT architecture (e.g., protocol translation, security, device abstraction, data aggregation, rate limiting). It acts as a single entry point for applications to interact with diverse IoT devices.
2. **Conceptual Flow:** Describe a typical data flow:
   - A smart object (e.g., a temperature sensor) sends data or exposes a function (e.g., `getTemperature()`).
   - The API Gateway receives this data/functionality from the device, potentially translating it from a device-specific protocol (e.g., CoAP, MQTT) to a standard web API (e.g., REST/HTTP).
   - An external application makes a standard HTTP request to the API Gateway.
   - The API Gateway forwards the request to the appropriate device, receives the response, and sends it back to the application.
3. **Illustrative API Structure:** Provide a simplified representation of how an API gateway might expose a smart object's functionality via a RESTful API.

Source Code:

```
// Illustrative Conceptual API Gateway Endpoint for a Smart Thermostat

// --- Device: Smart Thermostat (conceptual capabilities) ---
// - Reports temperature (e.g., via MQTT topic:
/devices/thermostat001/temperature)
// - Accepts commands to set target temperature (e.g., via MQTT topic:
/devices/thermostat001/set_temp)

// --- API Gateway Configuration (Conceptual) ---
// The API Gateway would map incoming HTTP requests to device-specific
// protocols and messages.

// Example API Endpoint for getting temperature:
{
  "path": "/api/v1/devices/{deviceId}/temperature",
  "method": "GET",
  "description": "Retrieves the current temperature from a specified
thermostat device.",
  "gateway_action": {
    "protocol_translation": "MQTT",
    "mqtt_topic_subscribe": "/devices/{deviceId}/temperature",
    "response_mapping": {
      "mqtt_payload": "$.temperature_value", // Extract 'temperature_value'
from MQTT payload
      "http_response_format": {
        "temperature": "{value}",
        "unit": "Celsius"
      }
    }
  }
}

// Example API Endpoint for setting target temperature:
{
  "path": "/api/v1/devices/{deviceId}/set_temperature",
  "method": "POST",
  "description": "Sets the target temperature for a specified thermostat
device.",
  "request_body_schema": {
    "type": "object",
    "properties": {
      "target_temperature": { "type": "number", "description": "Desired
temperature in Celsius" }
    },
    "required": ["target_temperature"]
  },
  "gateway_action": {
    "protocol_translation": "MQTT",
    "mqtt_topic_publish": "/devices/{deviceId}/set_temp",
    "mqtt_payload_mapping": {
      "from_http_body": "$.target_temperature", // Map HTTP request body to
MQTT payload
      "mqtt_payload_format": "{ \"target\": {value} }"
    },
    "http_response_status": 200
  }
}
```

Input:

Conceptual HTTP GET request: `GET /api/v1/devices/thermostat001/temperature`

Conceptual HTTP POST request: `POST /api/v1/devices/thermostat001/set_temperature` with body `{ "target_temperature": 22.5 }`

Expected Output:

For GET request: Conceptual JSON response like `{ "temperature": 25.0, "unit": "Celsius" }`

For POST request: Conceptual HTTP 200 OK status, indicating the command was sent to the device.

Lab 6: Write and explain working of an HTTP-to-CoAP semantic mapping proxy in IoT toolkit.

Title: Lab 6: HTTP-to-CoAP Semantic Mapping Proxy

Aim: To understand and conceptually outline the working of an HTTP-to-CoAP semantic mapping proxy, which facilitates communication between web applications and constrained IoT devices.

Procedure:

1. **Protocol Comparison:** Briefly explain the key differences between HTTP (Hypertext Transfer Protocol) and CoAP (Constrained Application Protocol), highlighting why a proxy is necessary (e.g., CoAP's UDP-based, lightweight nature for constrained devices vs. HTTP's TCP-based, heavier nature for general web).
2. **Proxy Functionality:** Describe how the proxy translates requests and responses:
   - **Request Translation (HTTP to CoAP):** How an incoming HTTP request (e.g., `GET /sensor/temp`) is mapped to a CoAP request (e.g., `GET coap://device.ip/temp`). This includes mapping HTTP methods (GET, POST, PUT, DELETE) to CoAP methods, and URI paths.
   - **Response Translation (CoAP to HTTP):** How a CoAP response from a device is converted back into an HTTP response for the client.
   - **Semantic Mapping:** How the proxy handles differences in data formats, resource naming, and other semantic aspects between the two protocols.
3. **Illustrative Logic (Pseudocode):** Provide pseudocode or a high-level description of the proxy's core translation logic.

Source Code:

```
# Illustrative Pseudocode for HTTP-to-CoAP Proxy Logic

# --- Configuration (Conceptual Mappings) ---
# Maps HTTP paths to CoAP resources
RESOURCE_MAP = {
    "/api/v1/temperature": "/sensors/temp",
    "/api/v1/light": "/actuators/light",
    "/api/v1/status": "/device/status"
}

# Maps HTTP methods to CoAP methods
METHOD_MAP = {
    "GET": "GET",
    "POST": "POST",
    "PUT": "PUT",
    "DELETE": "DELETE"
}

# --- Proxy Server (Conceptual) ---
def start_http_proxy_server():
    # Listen for incoming HTTP requests on a specific port
    # http_server.listen(8080)
    print("HTTP-to-CoAP Proxy Server started on port 8080...")
    while True:
        http_request = receive_http_request()
        process_http_request(http_request)
```

```
def process_http_request(http_request):
    http_method = http_request.method
    http_path = http_request.path
    http_body = http_request.body # if POST/PUT

    print(f"Received HTTP request: {http_method} {http_path}")

    # 1. Semantic Mapping: Map HTTP path to CoAP resource path
    coap_resource_path = RESOURCE_MAP.get(http_path)
    if not coap_resource_path:
        send_http_error_response(http_request, 404, "Resource Not Found")
        return

    # 2. Method Mapping: Map HTTP method to CoAP method
    coap_method = METHOD_MAP.get(http_method)
    if not coap_method:
        send_http_error_response(http_request, 405, "Method Not Allowed")
        return

    # 3. Construct CoAP Request (conceptual)
    # In a real system, this would involve a CoAP client library
    coap_request = {
        "method": coap_method,
        "uri": f"coap://constrained_device_ip{coap_resource_path}",
        "payload": http_body # Direct payload transfer for simplicity
    }

    print(f"Forwarding as CoAP request: {coap_request['method']}
{coap_request['uri']}")

    # 4. Send CoAP Request and Receive CoAP Response
    coap_response = send_coap_request(coap_request)

    # 5. Translate CoAP Response to HTTP Response
    http_response = translate_coap_to_http(coap_response)

    # 6. Send HTTP Response back to client
    send_http_response(http_request, http_response)

def translate_coap_to_http(coap_response):
    # Map CoAP status codes to HTTP status codes
    # Map CoAP payload to HTTP response body
    # For simplicity, assume direct payload transfer and 200 OK for success
    http_status = 200 if coap_response.status == "2.05 Content" else 500
    http_body = coap_response.payload
    print(f"Received CoAP response. Translating to HTTP status:
{http_status}")
    return {"status": http_status, "body": http_body, "headers": {"Content-
Type": "application/json"}}

# Helper functions (conceptual)
# def receive_http_request(): ...
# def send_http_error_response(req, status, msg): ...
# def send_coap_request(req): ...
# def send_http_response(req, res): ...

# Call to start the proxy (conceptual)
# start_http_proxy_server()
```

Input:

Conceptual HTTP GET request: `GET /api/v1/temperature`

Conceptual HTTP POST request: `POST /api/v1/light` with body `{ "state": "on" }`

Expected Output:

For GET request: The proxy logs the HTTP request, constructs and sends a CoAP GET request to the device (e.g., `coap://device.ip/sensors/temp`), receives a CoAP response, and translates it back to an HTTP response for the client.

For POST request: Similar process, translating to a CoAP POST request (e.g., `coap://device.ip/actuators/light`) with the appropriate payload.

Lab 7: Describe gateway as a service deployment in IoT toolkit.

Title: Lab 7: Gateway as a Service (GaaS) Deployment

Aim: To describe the concept of Gateway as a Service (GaaS) and its typical deployment model within an IoT toolkit or cloud platform.

Procedure:

1. **Define GaaS:** Explain what Gateway as a Service means. Emphasize that it's a cloud-based offering where the IoT platform provides and manages the gateway infrastructure, abstracting away the complexities of deployment, scaling, and maintenance from the user.
2. **Benefits of GaaS:** Discuss the advantages of using GaaS, such as:
   - **Scalability:** Automatically scales to handle varying loads of device connections and data.
   - **Reduced Operational Overhead:** Users don't need to provision, manage, or update gateway servers.
   - **Cost-Effectiveness:** Often a pay-as-you-go model.
   - **Security:** Cloud providers handle security updates and often provide built-in security features.
   - **Integration:** Seamless integration with other cloud services (e.g., databases, analytics, machine learning).
3. **Deployment Model:** Describe a typical GaaS deployment:
   - **Edge Gateways:** Physical devices deployed at the edge (e.g., Raspberry Pi, industrial gateways) that connect local devices and forward data to the cloud GaaS.
   - **Cloud GaaS:** The managed service in the cloud that receives data from edge gateways or directly from devices, performs protocol translation, authentication, authorization, and routes data to appropriate backend services.
   - **Device Connectivity:** How devices connect to the GaaS (e.g., MQTT, HTTP, CoAP, WebSocket).
   - **Backend Integration:** How the GaaS integrates with other cloud services (e.g., message brokers, data lakes, analytics engines, application logic).
4. **Use Cases:** Provide examples of scenarios where GaaS is particularly beneficial (e.g., large-scale deployments, rapid prototyping, remote device management).

Source Code: N/A (This lab focuses on conceptual understanding and architectural description.)

Input: N/A

Expected Output: A detailed description of Gateway as a Service, its benefits, and its typical deployment architecture within an IoT cloud platform, including the roles of edge and cloud components.

Lab 8: Explain application framework and embedded software agents for IoT toolkit

Title: Lab 8: Application Frameworks and Embedded Software Agents for IoT

Aim: To explain the roles and characteristics of IoT application frameworks and embedded software agents within the context of an IoT toolkit.

Procedure:

1. **IoT Application Frameworks:**
    - **Definition:** Define what an IoT application framework is (e.g., a set of libraries, tools, and conventions that simplify the development of IoT applications).
    - **Purpose:** Explain its purpose, such as providing common services (device management, data ingestion, analytics, visualization), abstracting underlying complexities, and enabling faster development.
    - **Examples:** Mention conceptual examples like cloud IoT platforms' SDKs, or open-source frameworks like Eclipse IoT projects (e.g., Eclipse Ditto for digital twins, Eclipse Hono for device connectivity).
2. **Embedded Software Agents:**
    - **Definition:** Explain what embedded software agents are in the context of IoT (e.g., lightweight software components or programs running directly on constrained IoT devices).
    - **Function:** Describe their primary functions, such as:
        - **Data Collection:** Reading sensor data.
        - **Local Processing:** Performing basic data filtering or aggregation at the edge.
        - **Device Control:** Actuating connected hardware.
        - **Communication:** Establishing and maintaining connections with gateways or cloud platforms using specific IoT protocols (e.g., MQTT client, CoAP client).
        - **Security:** Handling local authentication and encryption.
    - **Characteristics:** Highlight their typical characteristics (e.g., resource-constrained, real-time capabilities, low power consumption, robust against network intermittency).
3. **Interaction:** Describe how these two components interact: embedded agents collect and send data to the IoT platform (often facilitated by the application framework), and the application framework provides the tools and services for developers to build applications that consume this data and send commands back to the agents.

Source Code: N/A (This lab focuses on conceptual understanding.)

Input: N/A

Expected Output: Clear explanations of IoT application frameworks and embedded software agents, detailing their definitions, purposes, functions, and how they interact within an IoT solution.

Lab 9: Explain working of Raspberry Pi.

Title: Lab 9: Understanding Raspberry Pi

Aim: To understand the basic architecture, working principles, and relevance of Raspberry Pi as a versatile platform for IoT projects.

Procedure:

1. **Hardware Overview:**
   - Describe the key hardware components of a typical Raspberry Pi board (e.g., CPU, GPU, RAM, USB ports, HDMI port, Ethernet port, Wi-Fi/Bluetooth module).
   - **GPIO Pins:** Emphasize the importance of the General Purpose Input/Output (GPIO) pins, explaining their role in connecting to external sensors, actuators, and other electronic components.
2. **Operating System:**
   - Explain that Raspberry Pi runs a Linux-based operating system, typically Raspberry Pi OS (formerly Raspbian).
   - Discuss how the OS provides a familiar computing environment, enabling users to install software, write scripts (e.g., Python), and manage hardware.
3. **Working Principle:**
   - Describe the boot process and how the OS loads.
   - Explain how programs interact with the hardware, particularly through the GPIO pins (e.g., reading sensor values, controlling LEDs).
   - Discuss its capabilities for network connectivity (Wi-Fi, Ethernet) for sending and receiving data.
4. **IoT Relevance:** Discuss why Raspberry Pi is a popular and powerful choice for various IoT projects, citing its low cost, small form factor, versatility, strong community support, and ability to run full-fledged operating systems and programming languages.

Source Code: N/A (This lab focuses on conceptual understanding and hardware explanation.)

Input: N/A

Expected Output: A clear and concise explanation of Raspberry Pi's hardware, software (OS), working principles, and its suitability for IoT applications.

Lab 10: Connect Raspberry Pi with your existing system components

Title: Lab 10: Connecting Raspberry Pi to System Components

Aim: To conceptually demonstrate how to connect a Raspberry Pi to common system components (e.g., sensors, actuators) and interact with them programmatically.

Procedure:

1. **Physical Connections (Conceptual):**
   - **LED:** Describe how to connect a simple LED to a GPIO pin (with a resistor).
   - **Button/Switch:** Describe how to connect a push button to a GPIO pin for input.
   - **Sensor (e.g., DHT11 Temperature/Humidity):** Explain conceptually how a sensor might be wired (e.g., power, ground, data pin).
2. **Network Configuration (Conceptual):** Briefly explain the steps to connect the Raspberry Pi to a Wi-Fi network or via Ethernet, which is crucial for sending data to other system components or cloud services.
3. **Software Interaction (Illustrative Python Pseudocode):** Provide pseudocode that demonstrates how a Python script on the Raspberry Pi would:
   - Initialize GPIO pins.
   - Read input from a button or sensor.
   - Control an output device like an LED.
   - (Optional, conceptual) Send data over a network (e.g., via MQTT or HTTP).

Source Code:

```
# Illustrative Python Pseudocode for Raspberry Pi Component Interaction

# --- Import conceptual GPIO library ---
# import RPi.GPIO as GPIO
# import time
# import requests # For conceptual network communication

# --- GPIO Pin Definitions ---
LED_PIN = 17      # GPIO pin connected to an LED
BUTTON_PIN = 4    # GPIO pin connected to a push button
SENSOR_DATA_PIN = 23 # GPIO pin for a conceptual sensor (e.g., DHT11 data)

# --- Setup GPIO (conceptual) ---
# GPIO.setmode(GPIO.BCM) # Use Broadcom pin-numbering scheme
# GPIO.setup(LED_PIN, GPIO.OUT)
# GPIO.setup(BUTTON_PIN, GPIO.IN, pull_up_down=GPIO.PUD_UP) # Button
connected to GND

print("Raspberry Pi Component Interaction Demo Initialized.")

# --- Main Loop for Interaction ---
while True:
    # 1. Read Button Input
    # if GPIO.input(BUTTON_PIN) == GPIO.LOW: # Button is pressed (pulled
down)
    if conceptual_button_is_pressed(): # Conceptual function
        print("Button pressed! Toggling LED...")
        # GPIO.output(LED_PIN, not GPIO.input(LED_PIN)) # Toggle LED state
        conceptual_toggle_led(LED_PIN)
        # time.sleep(0.5) # Debounce delay

    # 2. Read Sensor Data (conceptual)
    # In a real scenario, this would involve a sensor library
    temperature, humidity = conceptual_read_sensor(SENSOR_DATA_PIN)
    if temperature is not None and humidity is not None:
        print(f"Sensor Data: Temp={temperature}°C, Humidity={humidity}%")

        # 3. Conceptual Network Communication (e.g., send data to a server)
        # try:
        #     response = requests.post("http://your-iot-platform.com/data",
        #                              json={"temp": temperature, "humidity":
humidity})
        #     if response.status_code == 200:
        #         print("Data sent successfully to IoT platform.")
        #     else:
        #         print(f"Failed to send data: {response.status_code}")
        # except requests.exceptions.RequestException as e:
        #     print(f"Network error: {e}")

    # time.sleep(1) # Wait for 1 second before next iteration

# --- Conceptual Helper Functions ---
def conceptual_button_is_pressed():
    # Simulate button press
    return input("Press 'b' to simulate button press, 'q' to quit: ").lower()
== 'b'

def conceptual_toggle_led(pin):
    print(f"LED on pin {pin} toggled.")

def conceptual_read_sensor(pin):
    # Simulate sensor readings
    import random
    return random.uniform(20, 30), random.uniform(50, 70) # Temp, Humidity

# Run the conceptual loop
# while True:
#     if conceptual_button_is_pressed():
#         conceptual_toggle_led(LED_PIN)
#     temp, hum = conceptual_read_sensor(SENSOR_DATA_PIN)
#     if temp:
#         print(f"Simulated sensor data: Temp={temp:.1f}°C,
Humidity={hum:.1f}%")
#     time.sleep(2)
```

Input:

Simulated button presses.

Conceptual sensor readings (e.g., temperature, humidity values).

Expected Output:

Messages indicating LED toggling when the button is "pressed".

Display of simulated sensor data (temperature, humidity).

Conceptual messages about data being sent to an IoT platform.

Lab 11: Give overview of Zetta.

Title: Lab 11: Overview of Zetta

Aim: To provide an overview of Zetta, an open-source, API-first IoT platform built on Node.js.

Procedure:

1. **Define Zetta:** Explain what Zetta is. Highlight its nature as an open-source platform for building IoT applications, emphasizing its "API-first" approach and its foundation in Node.js.
2. **Key Features:** Describe Zetta's core features:
    - **Hypermedia API:** How it exposes devices and their capabilities as RESTful APIs, making them easily accessible from web applications.
    - **Reactive Programming:** Its use of reactive streams (via RxJS) to handle asynchronous data flows from devices.
    - **Device Abstraction:** How it abstracts away the complexities of different device protocols and hardware, allowing developers to interact with devices consistently.
    - **Cloud/Edge Agnostic:** Its ability to run on various platforms, from Raspberry Pi (edge) to cloud servers.
    - **Siren Hypermedia:** Its use of Siren, a hypermedia specification, to provide discoverable APIs.
3. **Architecture (High-level):** Briefly explain its high-level architecture, typically involving:
    - **Zetta Server:** The core Node.js application that manages devices, exposes APIs, and handles data streams.
    - **Drivers:** Software modules that allow Zetta to communicate with specific hardware devices (e.g., GPIO, sensors, actuators).
    - **Scouts:** Components that discover and provision devices.
    - **Apps:** Applications built on top of Zetta using its APIs.
4. **Use Cases:** Mention common use cases for Zetta, such as home automation, smart environments, and rapid prototyping of IoT solutions.

Source Code: N/A (This lab focuses on conceptual understanding and platform overview.)

Input: N/A

Expected Output: A comprehensive overview of Zetta, including its definition, key features, high-level architecture, and typical use cases.

Lab 12: Home Automation - Level 0

Title: Lab 12: Home Automation - Level 0 (Basic On/Off Control)

Aim: To conceptually implement a very basic home automation system for simple on/off control of a single device (e.g., a light).

Procedure:

1. **Scenario:** Define a straightforward scenario: controlling a single light bulb (or any simple appliance) remotely or via a local switch.
2. **Components (Conceptual):**
   - **Controlled Device:** A light bulb connected to a relay or smart plug.
   - **Controller:** A microcontroller (e.g., Raspberry Pi, ESP32) that can send commands.
   - **Input Mechanism:** A conceptual button, a simple web interface, or a command sent via a message broker (e.g., MQTT).
3. **Logic:** Describe the simple logic: when a command "ON" is received, turn the light on; when "OFF" is received, turn it off.
4. **Illustrative Code (Pseudocode):** Provide pseudocode for the controller's logic.

Source Code:

```
# Illustrative Python Pseudocode for Home Automation - Level 0 (Light
Control)

# --- Configuration ---
LIGHT_CONTROL_PIN = 17 # GPIO pin connected to a relay for the light
MQTT_BROKER_HOST = "your_mqtt_broker.com"
MQTT_TOPIC_COMMAND = "home/livingroom/light/command"
MQTT_TOPIC_STATUS = "home/livingroom/light/status"

# --- Initialize (conceptual) ---
# setup_gpio(LIGHT_CONTROL_PIN, OUTPUT)
# connect_mqtt_client(MQTT_BROKER_HOST)
# subscribe_mqtt_topic(MQTT_TOPIC_COMMAND, on_message_received)

print("Home Automation Level 0: Light Control System Initialized.")
print(f"Listening for commands on MQTT topic: {MQTT_TOPIC_COMMAND}")

# --- Event Handler for MQTT Messages ---
def on_message_received(topic, message):
    print(f"Received command: {message} on topic: {topic}")
    if topic == MQTT_TOPIC_COMMAND:
        if message.lower() == "on":
            # write_gpio(LIGHT_CONTROL_PIN, HIGH) # Turn light on
            conceptual_control_light("ON")
            # publish_mqtt_message(MQTT_TOPIC_STATUS, "ON")
        elif message.lower() == "off":
            # write_gpio(LIGHT_CONTROL_PIN, LOW) # Turn light off
            conceptual_control_light("OFF")
            # publish_mqtt_message(MQTT_TOPIC_STATUS, "OFF")
        else:
            print("Invalid command. Use 'on' or 'off'.")

# --- Conceptual Light Control Function ---
def conceptual_control_light(state):
    print(f"Light is now: {state}")
```

```
# --- Main Loop (conceptual) ---
# This loop would keep the MQTT client running and processing messages
# while True:
#     process_mqtt_messages()
#     time.sleep(0.1)

# Simulate receiving commands
print("\nSimulating commands (type 'on' or 'off'):")
while True:
    cmd = input("> ").strip().lower()
    if cmd in ["on", "off"]:
        on_message_received(MQTT_TOPIC_COMMAND, cmd)
    elif cmd == "exit":
        break
    else:
        print("Unknown command.")
```

Input: Conceptual commands like "on" or "off" sent to the system (e.g., via a simulated MQTT message).

Expected Output:

```
Home Automation Level 0: Light Control System Initialized.
Listening for commands on MQTT topic: home/livingroom/light/command

Simulating commands (type 'on' or 'off'):
> on
Received command: on on topic: home/livingroom/light/command
Light is now: ON
> off
Received command: off on topic: home/livingroom/light/command
Light is now: OFF
> invalid
Received command: invalid on topic: home/livingroom/light/command
Invalid command. Use 'on' or 'off'.
```

Lab 13: Home Automation - Level 4

Title: Lab 13: Home Automation - Level 4 (Advanced Scenario with Multiple Devices and Rules)

Aim: To conceptually design an advanced home automation system involving multiple devices, sensors, and complex rules, demonstrating a higher level of integration and intelligence.

Procedure:

1. **Scenario:** Define a complex home automation scenario. Example:
   - Automated climate control based on temperature, humidity, and occupancy.
   - Smart lighting that adjusts brightness and color based on time of day, ambient light, and user presence.
   - Security features like door/window monitoring and motion detection with alerts.
   - Integration with a central hub or cloud platform for unified control and data analytics.
2. **Components (Conceptual):**
   - **Sensors:** Temperature, humidity, motion, door/window contact sensors.
   - **Actuators:** Smart thermostat, smart light bulbs (dimmable, color-changing), smart plugs/relays for other appliances.
   - **Controller/Hub:** A more powerful device (e.g., Raspberry Pi, dedicated home automation hub) or a cloud-based IoT platform.
   - **User Interface:** Mobile app or web dashboard for monitoring and manual override.
3. **Logic:** Describe intricate rules and interdependencies. Examples:
   - If temperature > 25°C AND occupancy detected, turn AC ON.
   - If it's evening (after 6 PM) AND ambient light is low AND motion detected in living room, turn living room lights to warm white, 50% brightness.
   - If door/window sensor opens when "Away" mode is active, trigger an alarm and send a notification.
   - Schedule-based actions (e.g., lights dim at bedtime).
4. **Illustrative Code (High-level Pseudocode):** Provide high-level pseudocode for the central automation logic, demonstrating how various sensor inputs trigger complex actions across multiple devices.

Source Code:

```
# Illustrative High-Level Pseudocode for Home Automation - Level 4

# --- Conceptual Device States and Sensor Readings ---
class DeviceState:
    def __init__(self):
        self.temperature = 22.0
        self.humidity = 55
        self.living_room_motion = False
        self.bedroom_motion = False
        self.front_door_open = False
        self.ambient_light_level = 300 # Lux
        self.time_of_day = "day" # "day", "evening", "night"
        self.home_mode = "home" # "home", "away", "sleep"

        self.ac_state = "OFF"
        self.living_room_light_state = {"power": "OFF", "brightness": 0,
"color": "white"}
        self.bedroom_light_state = {"power": "OFF", "brightness": 0, "color":
"white"}
        self.alarm_state = "INACTIVE"

# --- Conceptual Functions to Interact with Devices ---
def set_ac_state(state):
    print(f"AC set to: {state}")
    device_states.ac_state = state

def set_light_state(room, power, brightness=None, color=None):
    print(f"{room} light set: Power={power}, Brightness={brightness},
Color={color}")
    if room == "living_room":
        device_states.living_room_light_state["power"] = power
        if brightness is not None:
device_states.living_room_light_state["brightness"] = brightness
        if color is not None: device_states.living_room_light_state["color"]
= color
    elif room == "bedroom":
        device_states.bedroom_light_state["power"] = power
        if brightness is not None:
device_states.bedroom_light_state["brightness"] = brightness
        if color is not None: device_states.bedroom_light_state["color"] =
color

def trigger_alarm():
    print("!!! ALARM TRIGGERED !!! Sending notifications...")
    device_states.alarm_state = "ACTIVE"

def send_notification(message):
    print(f"Notification: {message}")

# --- Main Automation Logic ---
def run_automation_rules():
    global device_states # Access the global state object

    print(f"\n--- Running Automation Rules (Mode: {device_states.home_mode},
Time: {device_states.time_of_day}) ---")

    # Rule 1: Climate Control
    if device_states.temperature > 25.0 and device_states.home_mode ==
"home":
        if device_states.ac_state != "ON":
            set_ac_state("ON")
    elif device_states.temperature < 22.0 and device_states.home_mode ==
"home":
        if device_states.ac_state != "OFF":
            set_ac_state("OFF") # Or turn on heater if available

    # Rule 2: Smart Lighting - Living Room
    if device_states.time_of_day == "evening" and
device_states.ambient_light_level < 150:
        if device_states.living_room_motion:
            if device_states.living_room_light_state["power"] == "OFF":
                set_light_state("living_room", "ON", brightness=70,
color="warm_white")
        else:
            if device_states.living_room_light_state["power"] == "ON":
                set_light_state("living_room", "OFF")

    # Rule 3: Smart Lighting - Bedroom (for sleep mode)
    if device_states.home_mode == "sleep" and device_states.time_of_day ==
"night":
        if device_states.bedroom_light_state["power"] == "ON":
            set_light_state("bedroom", "OFF")
    # Rule 4: Security - Front Door
    if device_states.front_door_open and device_states.home_mode == "away":
        if device_states.alarm_state == "INACTIVE":
            trigger_alarm()
            send_notification("Front door opened while in AWAY mode!")

    # Add more complex rules here...

# --- Simulate Sensor Updates and Time Changes ---
device_states = DeviceState()

# Scenario 1: Evening, dark, motion in living room
device_states.time_of_day = "evening"
device_states.ambient_light_level = 100
device_states.living_room_motion = True
run_automation_rules()

# Scenario 2: Hot day, home mode
device_states.temperature = 26.5
device_states.home_mode = "home"
device_states.living_room_motion = False # Motion stopped
run_automation_rules()

# Scenario 3: Away mode, front door opens
device_states.home_mode = "away"
device_states.front_door_open = True
run_automation_rules()

# Scenario 4: Back home, door closed
device_states.home_mode = "home"
device_states.front_door_open = False
run_automation_rules()
```

Input: Simulated sensor data (temperature, humidity, motion, door status, ambient light) and changes in `home_mode` or `time_of_day`.

Expected Output:

```
--- Running Automation Rules (Mode: evening, Time: evening) ---
Living room light set: Power=ON, Brightness=70, Color=warm_white

--- Running Automation Rules (Mode: home, Time: evening) ---
AC set to: ON

--- Running Automation Rules (Mode: away, Time: evening) ---
!!! ALARM TRIGGERED !!! Sending notifications...
Notification: Front door opened while in AWAY mode!

--- Running Automation Rules (Mode: home, Time: evening) ---
(No new actions if conditions for previous actions are no longer met or
actions already taken)
```

The output will show the conceptual state changes of various devices (AC, lights, alarm) and notifications based on the simulated inputs and defined rules.

Lab 14: Smart Irrigation System

Title: Lab 14: Smart Irrigation System

Aim: To design a conceptual smart irrigation system that optimizes water usage based on soil moisture levels and potentially weather forecasts.

Procedure:

1. **Scenario:** Automate garden watering to prevent over-watering or under-watering, conserving water resources.
2. **Components (Conceptual):**
   - **Sensor:** Soil moisture sensor.
   - **Actuator:** Water pump or solenoid valve.
   - **Controller:** Microcontroller (e.g., ESP32, Raspberry Pi).
   - **(Optional) Data Source:** External weather API for rain forecasts.
3. **Logic:** Describe the control logic:
   - Continuously read soil moisture data.
   - If soil moisture falls below a predefined threshold, check for rain forecast (if integrated).
   - If soil is dry AND no rain is expected (or rain is not integrated), activate the water pump for a specific duration.
   - If soil is wet or rain is expected, keep the pump off.
4. **Illustrative Code (Pseudocode):** Provide pseudocode for the irrigation control logic.

Source Code:

```
# Illustrative Python Pseudocode for Smart Irrigation System

# --- Configuration ---
SOIL_MOISTURE_DRY_THRESHOLD = 300 # Lower value means drier (conceptual ADC
reading)
WATER_PUMP_PIN = 27              # GPIO pin for the water pump relay
WATERING_DURATION_SECONDS = 60   # How long to water when activated

# --- Initialize (conceptual) ---
# setup_gpio(WATER_PUMP_PIN, OUTPUT)
# write_gpio(WATER_PUMP_PIN, LOW) # Ensure pump is off initially

print("Smart Irrigation System Initialized. Monitoring soil moisture...")

# --- Main Loop ---
def run_irrigation_system():
    while True:
        # 1. Read Soil Moisture (conceptual)
        # In a real system, this would involve ADC readings from a sensor
        soil_moisture_value = conceptual_read_soil_moisture_sensor()
        print(f"Current Soil Moisture: {soil_moisture_value}")

        # 2. Check Rain Forecast (conceptual - optional)
        # is_rain_expected = conceptual_check_weather_forecast()
        # if is_rain_expected:
        #     print("Rain is expected. Skipping watering cycle.")
        #     time.sleep(3600) # Wait for an hour
        #     continue
        # 3. Apply Irrigation Logic
        if soil_moisture_value < SOIL_MOISTURE_DRY_THRESHOLD:
            print("Soil is dry. Activating water pump...")
            # write_gpio(WATER_PUMP_PIN, HIGH) # Turn pump ON
            conceptual_control_pump("ON")
            # time.sleep(WATERING_DURATION_SECONDS) # Water for specified
duration
            conceptual_wait_for_duration(WATERING_DURATION_SECONDS)
            print("Watering complete. Deactivating water pump.")
            # write_gpio(WATER_PUMP_PIN, LOW) # Turn pump OFF
            conceptual_control_pump("OFF")
        else:
            print("Soil moisture is sufficient. No watering needed.")

        # Wait before next check
        # time.sleep(300) # Check every 5 minutes

# --- Conceptual Helper Functions ---
def conceptual_read_soil_moisture_sensor():
    # Simulate sensor readings
    import random
    # Simulate drying over time, then getting wet after watering
    if not hasattr(conceptual_read_soil_moisture_sensor, "current_moisture"):
        conceptual_read_soil_moisture_sensor.current_moisture = 500 # Start
moist

    if conceptual_read_soil_moisture_sensor.current_moisture >
SOIL_MOISTURE_DRY_THRESHOLD:
        conceptual_read_soil_moisture_sensor.current_moisture -=
random.randint(10, 50) # Dry out
    else:
        # If it was dry and pump was on, simulate getting wet
        if conceptual_control_pump.state == "ON":
            conceptual_read_soil_moisture_sensor.current_moisture = 600 # Get
wet
            conceptual_control_pump.state = "OFF" # Reset pump state after
watering
        else:
            conceptual_read_soil_moisture_sensor.current_moisture -=
random.randint(10, 50) # Continue drying

    if conceptual_read_soil_moisture_sensor.current_moisture < 0:
        conceptual_read_soil_moisture_sensor.current_moisture = 0
    return conceptual_read_soil_moisture_sensor.current_moisture

def conceptual_control_pump(state):
    conceptual_control_pump.state = state
    print(f"Water pump is now: {state}")

def conceptual_wait_for_duration(seconds):
    print(f"Simulating watering for {seconds} seconds...")
    # In a real system, this would be time.sleep(seconds)

# To run the simulation:
# run_irrigation_system() # This would run indefinitely
# For a limited simulation:
for _ in range(5): # Run 5 cycles
    run_irrigation_system()
    print("\n--- Next Cycle ---")
```

Input: Simulated soil moisture values (e.g., `500` for wet, `100` for dry).

Expected Output:

```
Smart Irrigation System Initialized. Monitoring soil moisture...
```

```
Current Soil Moisture: 500
Soil moisture is sufficient. No watering needed.

--- Next Cycle ---
Current Soil Moisture: 450
Soil moisture is sufficient. No watering needed.

--- Next Cycle ---
Current Soil Moisture: 300
Soil moisture is sufficient. No watering needed.

--- Next Cycle ---
Current Soil Moisture: 200
Soil is dry. Activating water pump...
Water pump is now: ON
Simulating watering for 60 seconds...
Watering complete. Deactivating water pump.
Water pump is now: OFF

--- Next Cycle ---
Current Soil Moisture: 600
Soil moisture is sufficient. No watering needed.
```

The output will show the system checking soil moisture and activating/deactivating the pump based on the conceptual logic and simulated sensor data.

Lab 15: Weather Reporting Systems

Title: Lab 15: Weather Reporting System

Aim: To design a conceptual IoT-based weather reporting system that collects and displays local environmental data.

Procedure:

1. **Scenario:** Build a system to monitor and report local weather conditions (e.g., temperature, humidity, atmospheric pressure).
2. **Components (Conceptual):**
   - **Sensors:** Temperature sensor (e.g., DHT11/DHT22), humidity sensor, barometric pressure sensor (e.g., BMP180/BME280).
   - **Controller:** Microcontroller (e.g., ESP8266, Raspberry Pi).
   - **Connectivity:** Wi-Fi module to send data.
   - **Data Destination:** A conceptual cloud-based IoT platform (e.g., ThingSpeak, Firebase, or a simple web server) for data storage and visualization.
3. **Data Flow:** Describe the process: sensors collect data -> controller reads data -> controller sends data over Wi-Fi -> data is received and stored on the cloud platform -> data is visualized on a dashboard.
4. **Illustrative Code (Pseudocode):** Provide pseudocode for reading sensor data and sending it to a conceptual cloud service.

Source Code:

```
# Illustrative Python Pseudocode for Weather Reporting System

# --- Configuration ---
# WIFI_SSID = "YourWiFiNetwork"
# WIFI_PASSWORD = "YourWiFiPassword"
# IOT_PLATFORM_API_ENDPOINT = "http://your.iot.platform.com/api/weather"
# API_KEY = "YOUR_API_KEY_HERE"

# --- Initialize (conceptual) ---
# connect_wifi(WIFI_SSID, WIFI_PASSWORD)
print("Weather Reporting System Initialized. Connecting to network...")
# if not is_wifi_connected():
#     print("Failed to connect to WiFi. Exiting.")
#     exit()
print("Network connected. Starting sensor readings...")

# --- Main Loop ---
def run_weather_reporter():
    while True:
        # 1. Read Sensor Data (conceptual)
        # In a real system, this would involve specific sensor libraries
        temperature = conceptual_read_temperature_sensor()
        humidity = conceptual_read_humidity_sensor()
        pressure = conceptual_read_pressure_sensor()

        if None not in (temperature, humidity, pressure):
            print(f"Readings: Temp={temperature:.1f}°C,
Humidity={humidity:.1f}%, Pressure={pressure:.1f}hPa")

            # 2. Prepare Data Payload
            weather_data = {
                "timestamp": conceptual_get_current_timestamp(),
                "temperature": temperature,
                "humidity": humidity,
                "pressure": pressure,
                "api_key": "YOUR_API_KEY_HERE" # Conceptual API key
            }

            # 3. Send Data to IoT Platform (conceptual HTTP POST)
            # try:
            #     response = requests.post(IOT_PLATFORM_API_ENDPOINT,
json=weather_data)
            #     if response.status_code == 200:
            #         print("Weather data sent successfully to IoT
platform.")
            #     else:
            #         print(f"Failed to send data: {response.status_code},
{response.text}")
            # except requests.exceptions.RequestException as e:
            #     print(f"Network error sending data: {e}")
            conceptual_send_data_to_platform(weather_data)
        else:
            print("Failed to read all sensor data. Retrying...")

        # Wait before next reading
        # time.sleep(300) # Read every 5 minutes

# --- Conceptual Helper Functions ---
def conceptual_read_temperature_sensor():
    import random
    return random.uniform(15.0, 35.0)

def conceptual_read_humidity_sensor():
    import random
    return random.uniform(40.0, 90.0)

def conceptual_read_pressure_sensor():
    import random
    return random.uniform(980.0, 1030.0)

def conceptual_get_current_timestamp():
    import datetime
    return datetime.datetime.now().isoformat()

def conceptual_send_data_to_platform(data):
    print(f"Simulating sending data to IoT platform: {data}")
    # In a real system, this would be an actual HTTP POST or MQTT publish

# To run the simulation:
# run_weather_reporter() # This would run indefinitely
# For a limited simulation:
for _ in range(3): # Run 3 cycles
    run_weather_reporter()
    print("\n--- Next Reading Cycle ---")
```

Input: Real-time sensor readings (simulated in the code).

Expected Output:

```
Weather Reporting System Initialized. Connecting to network...
Network connected. Starting sensor readings...
Readings: Temp=25.5°C, Humidity=65.2%, Pressure=1005.3hPa
Simulating sending data to IoT platform: {'timestamp': '...', 'temperature':
25.5, 'humidity': 65.2, 'pressure': 1005.3, 'api_key': 'YOUR_API_KEY_HERE'}

--- Next Reading Cycle ---
```

```
Readings: Temp=27.1°C, Humidity=70.8%, Pressure=1008.1hPa
Simulating sending data to IoT platform: {'timestamp': '...', 'temperature':
27.1, 'humidity': 70.8, 'pressure': 1008.1, 'api_key': 'YOUR_API_KEY_HERE'}

--- Next Reading Cycle ---
Readings: Temp=24.9°C, Humidity=60.1%, Pressure=1002.5hPa
Simulating sending data to IoT platform: {'timestamp': '...', 'temperature':
24.9, 'humidity': 60.1, 'pressure': 1002.5, 'api_key': 'YOUR_API_KEY_HERE'}
```

The output will show simulated sensor readings and messages indicating the conceptual sending of this data to an IoT platform.

Lab 16: Air Pollution Monitoring System

- **Title:** Lab 16: Air Pollution Monitoring System
- **Aim:** To design a conceptual IoT-based air pollution monitoring system that measures air quality parameters and potentially triggers alerts.
- **Procedure:**
    1. **Scenario:** Monitor ambient air quality by measuring levels of particulate matter (PM2.5, PM10) and potentially specific gases (e.g., CO, CO2, NO2).
    2. **Components (Conceptual):**
        - **Sensors:** Particulate matter sensor (e.g., PMS5003), gas sensors (e.g., MQ-series for CO, NO2, or a dedicated CO2 sensor).
        - **Controller:** Microcontroller (e.g., ESP32, Raspberry Pi).
        - **Connectivity:** Wi-Fi module for data transmission.
        - **Data Destination:** A conceptual cloud-based IoT platform for data storage, analysis, and visualization.
        - **Alert Mechanism:** Conceptual email/SMS notification service or a local indicator (e.g., LED).
    3. **Data Flow:** Describe the process: sensors collect air quality data -> controller reads data -> controller sends data over Wi-Fi -> data is stored/analyzed on the cloud platform -> if thresholds are exceeded, an alert is triggered.
    4. **Illustrative Code (Pseudocode):** Provide pseudocode for reading sensor data, checking thresholds, and triggering alerts.
- **Source Code:**

```
# Illustrative Python Pseudocode for Air Pollution Monitoring System

# --- Configuration ---
# WIFI_SSID = "YourWiFiNetwork"
# WIFI_PASSWORD = "YourWiFiPassword"
# IOT_PLATFORM_API_ENDPOINT = "http://your.iot.platform.com/api/air_quality"
# ALERT_EMAIL_SERVICE_ENDPOINT = "http://your.alert.service.com/send_email"

PM25_ALERT_THRESHOLD = 50  # micrograms/m3 (e.g., WHO guideline for 24-hour mean)
CO_ALERT_THRESHOLD = 10    # ppm

# --- Initialize (conceptual) ---
# connect_wifi(WIFI_SSID, WIFI_PASSWORD)
print("Air Pollution Monitoring System Initialized. Connecting to network...")
# if not is_wifi_connected():
#     print("Failed to connect to WiFi. Exiting.")
#     exit()
print("Network connected. Starting air quality monitoring...")

# --- Main Loop ---
def run_air_pollution_monitor():
    while True:
        # 1. Read Sensor Data (conceptual)
        pm25 = conceptual_read_pm25_sensor()
        co_level = conceptual_read_co_sensor()
        # Add other gas readings as needed (e.g., pm10, no2, co2)

        if None not in (pm25, co_level):
            print(f"Air Quality Readings: PM2.5={pm25:.1f} µg/m³, CO={co_level:.1f} ppm")

            # 2. Prepare Data Payload
            air_quality_data = {
                "timestamp": conceptual_get_current_timestamp(),
                "pm25": pm25,
                "co_level": co_level
            }

            # 3. Send Data to IoT Platform (conceptual)
            conceptual_send_data_to_platform(air_quality_data)

            # 4. Check for Alerts
            if pm25 > PM25_ALERT_THRESHOLD:
                alert_message = f"HIGH PM2.5 ALERT! Current: {pm25:.1f} µg/m³"
                print(alert_message)
                conceptual_send_alert(alert_message, "email", "recipient@example.com")

            if co_level > CO_ALERT_THRESHOLD:
                alert_message = f"HIGH CO LEVEL ALERT! Current: {co_level:.1f} ppm"
                print(alert_message)
                conceptual_send_alert(alert_message, "sms", "+1234567890")

        else:
            print("Failed to read all air quality sensor data. Retrying...")

        # Wait before next reading
        # time.sleep(60) # Read every minute

# --- Conceptual Helper Functions ---
def conceptual_read_pm25_sensor():
    import random
    return random.uniform(10.0, 70.0) # Simulate varying PM2.5 levels

def conceptual_read_co_sensor():
    import random
    return random.uniform(1.0, 15.0) # Simulate varying CO levels

def conceptual_get_current_timestamp():
    import datetime
    return datetime.datetime.now().isoformat()

def conceptual_send_data_to_platform(data):
    print(f"Simulating sending air quality data to IoT platform: {data}")
    # In a real system, this would be an actual HTTP POST or MQTT publish

def conceptual_send_alert(message, alert_type, destination):
    print(f"Simulating sending {alert_type} alert to {destination}: '{message}'")
    # In a real system, this would call an email/SMS API

# To run the simulation:
# run_air_pollution_monitor() # This would run indefinitely
# For a limited simulation:
for _ in range(5): # Run 5 cycles
    run_air_pollution_monitor()
    print("\n--- Next Monitoring Cycle ---")
```

- **Input:** Real-time air quality sensor readings (simulated in the code).
- **Expected Output:**

```
Air Pollution Monitoring System Initialized. Connecting to network...
Network connected. Starting air quality monitoring...
Air Quality Readings: PM2.5=35.2 µg/m³, CO=4.5 ppm
Simulating sending air quality data to IoT platform: {'timestamp': '...',
'pm25': 35.2, 'co_level': 4.5}

--- Next Monitoring Cycle ---
Air Quality Readings: PM2.5=62.1 µg/m³, CO=8.9 ppm
Simulating sending air quality data to IoT platform: {'timestamp': '...',
'pm25': 62.1, 'co_level': 8.9}
HIGH PM2.5 ALERT! Current: 62.1 µg/m³
Simulating sending email alert to recipient@example.com: 'HIGH PM2.5
ALERT! Current: 62.1 µg/m³'

--- Next Monitoring Cycle ---
Air Quality Readings: PM2.5=48.0 µg/m³, CO=11.2 ppm
Simulating sending air quality data to IoT platform: {'timestamp': '...',
'pm25': 48.0, 'co_level': 11.2}
HIGH CO LEVEL ALERT! Current: 11.2 ppm
Simulating sending sms alert to +1234567890: 'HIGH CO LEVEL ALERT!
Current: 11.2 ppm'

--- Next Monitoring Cycle ---
... (continues based on simulated data)
```

The output will show simulated air quality readings, conceptual data transmission, and alerts triggered when thresholds are exceeded.