CM2305 - Group 8 - Project Overview - Comprehensive Draft v1

Project Overview: Smart Doorbell System with Facial Recognition

This project revolves around creating a **smart doorbell system** with facial recognition, motion detection, door automation, and a mobile app for control and interaction. The system integrates various hardware components (e.g., sensors, cameras, motors) with a Raspberry Pi and Arduino, and allows users to interact with it through an intuitive **Android app**.

The focus is on **accessibility**, ensuring that the system is easy to use for people with disabilities. The hardware will be prototyped using **cardboard** and other basic materials, ensuring flexibility during testing and assembly, while maintaining a professional presentation for the final product.

Table of Contents

---

1. System Architecture & Communication Flow

The core of the system relies on **modular communication** between the hardware components (Raspberry Pi, sensors, motors) and the Android app. Here's how everything ties together:

Key Components:

1. **Raspberry Pi 4 Model B**
   - **Central Hub**: Controls all sensors, cameras, and actuators.
   - **Communication**: Uses **MQTT** protocol for real-time data transfer to and from the mobile app.
   - **Hardware Connectivity**: Interfaces with the sensors and camera through GPIO pins and the **GrovePi Plus**.
2. **Arduino (for Servo Motor Control)**
   - **Servo Motor**: Controls door opening/closing.
   - The Raspberry Pi communicates with Arduino for motor control via PWM signals.
3. **Sensors & Actuators:**
   - **PIR Motion Sensor**: Detects motion near the door.
   - **Ultrasonic Ranger**: Measures proximity to identify when a person is near.
   - **Temperature and Light Sensors**: Adjust the system's behavior based on environmental factors.
   - **Servo Motor**: Opens and closes the door based on commands from the app.
4. **Mobile App (Android)**
   - Provides a user interface for interacting with the door system.
   - **MQTT** communication with the Raspberry Pi allows the user to receive real-time updates and control the door remotely.

Communication Flow:

- **Raspberry Pi to App**: The Raspberry Pi subscribes to MQTT topics like `motion/detected`, `face/recognized`, and `door/status`. When an event occurs, such as detecting motion or recognizing a face, the Raspberry Pi sends a message to the app.
- **App to Raspberry Pi**: The mobile app sends commands like `door/open` or `door/close` via MQTT, which the Raspberry Pi then executes by triggering the servo motor.

---

2. Sensor Integration & Environmental Handling

The sensors in the system play a crucial role in detecting and reacting to the presence of users, as well as adjusting the system based on environmental changes.

Sensors and Their Roles:

1. **PIR Motion Sensor**:
   - Detects motion and sends a HIGH signal to the Raspberry Pi when someone approaches the door.
   - Positioned near the doorframe, this sensor will be used to trigger facial recognition when someone is detected.
2. **Ultrasonic Ranger**:
   - Measures distance to determine how close a person is to the door.
   - The data from this sensor will be used to trigger the facial recognition system when a person is within range.
3. **Temperature & Light Sensors**:
   - These sensors will help adjust the system's behavior based on the **ambient environment**.
   - The **light sensor** will determine if it's daytime or nighttime, influencing camera settings (e.g., switching to infrared at night).
   - The **temperature sensor** will ensure optimal sensor performance and can also trigger notifications if extreme temperatures are detected.

Environmental Adjustments:

- **Day/Night Mode**: If the light sensor detects that it's dark, the system will activate night mode on the camera (e.g., infrared lighting for facial recognition).
- **Motion Sensitivity**: The **temperature sensor** may adjust the motion sensitivity based on external weather conditions. For instance, cold weather might make the motion sensor less sensitive to avoid false positives caused by wind.

---

3. Mobile App Development & Interaction

The mobile app will serve as the primary interface for controlling the smart door system. It will be designed for **accessibility**, allowing easy navigation and clear feedback.

Core Features:

1. **Real-Time Notifications**:
   - Alerts when someone is at the door or when motion is detected.
   - **Push Notifications**: Using **Firebase Cloud Messaging (FCM)**, the app will notify the user of events like motion detection or facial recognition results.
2. **Facial Recognition**:
   - When a person approaches, the app will either scan their face (via the Raspberry Pi camera) or display the results from the Raspberry Pi's recognition system.
   - The app will compare the recognized face with the **database** to verify the user.
3. **Door Control**:
   - Buttons in the app will allow the user to **open or close** the door remotely.
   - **Manual Override**: In case of failure (e.g., facial recognition malfunction), the app will allow manual door control.
4. **MQTT Integration**:
   - The app communicates with the Raspberry Pi using **MQTT** for real-time communication. The app subscribes to topics like `motion/detected` or `face/recognized` to receive updates.
   - Commands to open or close the door are published via MQTT topics like `door/open` or `door/close`.

---

4. Door Automation & Actuation

The door will be controlled using a **servo motor** that can rotate to specific angles, allowing it to open and close based on commands from the mobile app or facial recognition.

Servo Motor Control:

1. **Hardware Setup**:
   - The servo motor is connected to the **Arduino** using a PWM signal.
   - The Raspberry Pi will communicate with the Arduino to send the **open/close** commands for the door.
2. **Manual Override**:
   - A **physical LED button** will be used for manual control in case the system fails. When pressed, it will trigger the door to open or close.

Cardboard Setup:

- The **servo motor** will be connected to a **cardboard door** prototype. You can create a basic model of the door using cardboard and attach the servo motor to simulate real-world usage. This setup will allow for testing before finalizing the design.

---

5. Testing & Validation

Unit Testing:

- Test each component individually (e.g., PIR sensor, servo motor, camera) to ensure that they work as expected.

System Integration Testing:

- Test the **end-to-end flow** where motion detection triggers facial recognition, and the mobile app controls the door via MQTT.

User Testing:

- Test the system with real users, particularly focusing on accessibility for individuals with disabilities. Gather feedback on usability and make improvements where needed.

---

6. Smart Home Integration & Future Considerations

Smart Home Integration:

- **Google Home, Alexa, Apple HomeKit**: The system can integrate with these platforms to allow voice control and automation.
- Using **IFTTT** or **Node-RED**, the system can be set up to respond to voice commands like “Hey Google, open the door.”

Future Enhancements:

- **AI Improvements**: Deep learning for more accurate facial recognition.
- **Multiple User Profiles**: Store and recognize multiple faces for different users.
- **Cloud Integration**: Allow remote control and monitoring via a cloud-based platform.

---

7. Deployment & Maintenance

Deployment Process

The deployment process involves getting your **smart door system** up and running in a real-world environment. This involves setting up both the **hardware** (Raspberry Pi, sensors, servo motor, etc.) and the **software** (mobile app, Raspberry Pi code, cloud services).

Raspberry Pi Deployment:

1. **Setting Up the Raspberry Pi**:
   - Install the **Raspberry Pi OS** on your **SD card**.
   - Set up the **MQTT broker** on the Raspberry Pi.
   - Connect the necessary sensors to the **GPIO pins**.
2. **Software Configuration**:
   - Install necessary libraries like **RPi.GPIO**, **OpenCV**, etc.
   - Set up serial communication between the Raspberry Pi and Arduino for servo control.
3. **Real-World Testing**:
   - Test sensors, camera, and servo motor performance in the field.

Mobile App Deployment:

1. **App Testing**:
   - Test the app on multiple devices and check compatibility.
   - Ensure the app can communicate via MQTT with the Raspberry Pi.
2. **Publishing**:
   - Publish the app on **Google Play** or distribute it via internal channels.

Cloud & Smart Home Integration:

1. **Set Up Cloud Infrastructure** (if applicable).
2. **Test Smart Home Voice Commands** (e.g., Google Assistant, Alexa).

Maintenance

To ensure long-term functionality, maintenance tasks include regular software updates, system monitoring, and hardware checks.

---

8. Prototyping with Cardboard & Presentation

Cardboard Usage:

- The cardboard will be used to prototype the **door**, **sensor mounts**, and **camera position**.
- Use **cardboard boxes** to simulate the door frame, positioning the **servo motor** to open/close the door effectively.
- For **aesthetic presentation**, use clean, well-cut cardboard to showcase the different components in a way that visually communicates the final design.

Conclusion: Visualizing the Final Product

At the end of the prototyping phase, the system will consist of the **Raspberry Pi**, **Arduino** (with servo motor), sensors, and camera mounted in a cardboard frame that simulates the final door setup. The **mobile app** will be user-friendly, with a **simple interface** for notifications, facial recognition, and door control.

The **cardboard mockups** will allow for testing the system's functionality in real-world conditions, ensuring that when the final components are added, the system will work seamlessly.

Core Technologies and Tools

1. MQTT (Message Queuing Telemetry Transport)

MQTT is a lightweight messaging protocol that is perfect for real-time communication in IoT systems. It allows devices to communicate with each other efficiently by publishing and subscribing to topics.

- **How MQTT Works**:
  - **Broker**: The server that facilitates communication by receiving and sending messages.
  - **Clients**: Devices (like the Raspberry Pi and mobile app) that subscribe to topics (to receive messages) and publish messages to topics (to send data).
- **Example Use**:
  - The **Raspberry Pi** acts as both an MQTT **publisher** and **subscriber**. It sends real-time updates like `motion/detected` or `door/status` to the mobile app.
  - The **Android app** will **subscribe** to these topics and receive the updates. It will also **publish** commands like `door/open` or `door/close` to the Raspberry Pi to control the door.
- **Example Code (Python for Raspberry Pi)**:

```
import paho.mqtt.client as mqtt

def on_connect(client, userdata, flags, rc):
    print("Connected with result code " + str(rc))
    client.subscribe("motion/detected")

def on_message(client, userdata, msg):
    print(f"Received message: {msg.payload.decode()}")
    # Handle the message (e.g., trigger facial recognition)

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect("mqtt-broker-address", 1883, 60)
client.loop_forever()
```

2. Raspberry Pi & Arduino Integration

- **Raspberry Pi**: A **single-board computer** that serves as the **central hub** for data processing, sensor control, and communication with the mobile app.
  - **GPIO Pins** on the Raspberry Pi will interface with sensors (like PIR motion sensor) and actuators (like the servo motor).
  - **Camera Module**: The Raspberry Pi Camera will be used for **facial recognition**.
- **Arduino**: A microcontroller used for **servo motor control**. The Raspberry Pi will send **commands** to Arduino, instructing it to move the servo motor to open or close the door.

    - **Communication** between Raspberry Pi and Arduino is done through **serial communication**.
- **Example Communication**:
    - Raspberry Pi sends a command to Arduino to open or close the door.

3. Android Development (Java/Kotlin)

- **Primary Language**: **Java** will be used to develop the mobile app, as it is widely known and easier for many developers to work with. However, **Kotlin** is also an option if the team prefers it, since Kotlin provides a more concise and modern syntax.
- **Android UI Development**:
    - **XML Layouts** will be used for UI design, but **Jetpack Compose** is another option if you prefer a **declarative UI** approach.
    - Use **Retrofit** or **OkHttp** for **networking** to communicate with the Raspberry Pi via MQTT.
    - **Firebase Cloud Messaging (FCM)** will handle push notifications.
    - **Google ML Kit** or **TensorFlow Lite** can be used for **face detection and recognition**.
- **Example Code** (Java for Door Control Button):

```
MqttAndroidClient mqttClient = new MqttAndroidClient(context, "tcp://mqttbroker.com", "ClientID");
mqttClient.connect();
mqttClient.publish("door/open", "open".getBytes(), 1, false);
```

4. Firebase Cloud Messaging (FCM)

FCM is used to send push notifications in real-time, which is ideal for alerting the user when something happens (e.g., motion detected, face recognized).

- **Example Use**: The Raspberry Pi detects motion and sends a message via MQTT to the mobile app. The app then sends a **FCM push notification** to the user to let them know someone is at the door.
- **Example Code (Android)**:

```
FirebaseMessaging.getInstance().subscribeToTopic("motion_alerts")
    .addOnCompleteListener(task -> {
        String msg = task.isSuccessful() ? "Subscribed!" : "Subscription failed";
        Log.d(TAG, msg);
    });
```

5. TensorFlow Lite / OpenCV for Facial Recognition

- **OpenCV**: A powerful **image processing library** that can be used for **face detection**. It allows the Raspberry Pi to recognize faces via the camera.
- **TensorFlow Lite**: A **lightweight version** of **TensorFlow** designed to run directly on mobile devices. It's perfect for running machine learning models for tasks like **facial recognition**.
- **Example Code (Python with OpenCV)**:

```
import cv2
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')

def detect_faces(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, 1.1, 4)
    return faces
```

- On the Android side, **ML Kit** can be used for **real-time face detection** using the front-facing camera.

6. Servo Motor Control (Arduino)

The **servo motor** will be used to open and close the door based on commands from the Raspberry Pi or Android app.

- **Arduino** will be used to control the servo motor through PWM (Pulse Width Modulation) signals.
- **Example Code (Arduino)**:

```
#include <Servo.h>
Servo myServo;

void setup() {
    myServo.attach(9); // Servo connected to pin 9
```

```
}

void loop() {
    myServo.write(0); // Open door
    delay(2000); // Wait 2 seconds
    myServo.write(90); // Close door
    delay(2000);
}
```

Project File Structure and Folder Layout

Here's how the **project file structure** could look for organizing the code:

```
smart-door-system/
├── android-app/
│   ├── src/
│   │   ├── main/
│   │   │   ├── java/
│   │   │   │   └── com/
│   │   │   │       └── smartdoor/
│   │   │   │           ├── ui/                # UI elements (activities, fragments)
│   │   │   │           ├── network/           # Network code (MQTT, FCM)
│   │   │   │           ├── services/          # Background services (MQTT service)
│   │   │   │           ├── utils/             # Helper functions (face recognition)
│   │   │   │           └── data/              # Data models (user, door status)
│   │   │   ├── res/
│   │   │   │   ├── layout/                    # UI layouts (XML)
│   │   │   │   ├── values/                    # Strings, colors, themes
│   │   │   │   └── drawable/                  # Images, icons
├── raspberry-pi/
│   ├── src/
│   │   ├── main.py                              # Main Raspberry Pi code
│   │   ├── mqtt_handler.py                      # MQTT subscription and publishing
│   │   ├── sensor_integration.py                # Code for connecting sensors
│   │   ├── face_recognition.py                  # Code for facial recognition (OpenCV)
│   │   ├── servo_control.py                     # Code for controlling the servo motor
│   ├── requirements.txt                         # Python dependencies (e.g., paho-mqtt, opencv)
├── arduino/
│   ├── servo_control.ino                        # Arduino code for controlling servo motor
└── README.md                                     # Project overview, setup instructions
```

How Things Connect

1. **Raspberry Pi to Sensors**: The **Raspberry Pi** will read data from **sensors** (e.g., PIR motion sensor, ultrasonic ranger) using its **GPIO** pins or through the **GrovePi+** extension.
   - For instance, the **PIR sensor** will detect motion, triggering the facial recognition system or sending a signal to the mobile app via MQTT.
2. **Mobile App to Raspberry Pi**: The **Android app** will subscribe to MQTT topics to get real-time updates and send **commands** to control the door (e.g., open or close).
   - The app uses **Retrofit** or **OkHttp** for network communication and handles **notifications** via **FCM**.
3. **Arduino to Raspberry Pi**: The **Raspberry Pi** will communicate with the **Arduino** (via serial communication) to control the **servo motor** that opens and closes the door. The Raspberry Pi will send signals like `open` or `close` to the Arduino, and the Arduino will then move the motor accordingly.

Summary

We've gone over the core **technologies** like **MQTT**, **Raspberry Pi**, **Arduino**, **Android development**, **TensorFlow Lite/OpenCV**, **FCM**, and **servo motor control**. Each of these technologies plays a critical role in creating a fully functioning **smart door system**.

We've also outlined the **file structure** of the project to ensure that everything is organized for easier development. The communication between the Raspberry Pi, mobile app, and Arduino will be based on MQTT for real-time data sharing, while the mobile app will interface with

the Raspberry Pi using both MQTT and Firebase Cloud Messaging.

Let's dive into **Part 1: System Architecture & Communication Flow** in detail. In this section, we'll thoroughly explain how the core components of your system fit together, focusing on the communication between the **Raspberry Pi**, **Arduino**, and **Android app**. We'll also discuss the **hardware setup** and the flow of data between each part, ensuring that the system is clear and flexible for any changes that might arise later.

Part 1: System Architecture & Communication Flow

1.1 Overview of the System Architecture

The system consists of the following primary components:

1. **Raspberry Pi 4**:
   - The **central hub** that processes sensor data, controls the door mechanism, and communicates with both the **Arduino** (for motor control) and the **Android app** (for user interaction).
   - Connects to sensors and actuators via GPIO pins and interfaces with external devices like the **Raspberry Pi Camera**.
2. **Arduino**:
   - A microcontroller responsible for controlling the **servo motor** that physically opens and closes the door. The Arduino receives signals from the Raspberry Pi and executes corresponding motor commands.
3. **Android App**:
   - Provides the **user interface (UI)** for the user to interact with the door system.
   - Uses **real-time communication** to control the door and receive notifications when motion is detected, or the facial recognition system identifies a visitor.

1.2 Communication Flow Between Components

Let's break down the communication between these components, using **MQTT** (Message Queuing Telemetry Transport) for real-time data transmission.

1.2.1 Communication Between Raspberry Pi and Android App (via MQTT)

- The **Raspberry Pi** acts as the **MQTT broker** or **client**, which publishes sensor data (e.g., motion detected) and door status to specific **topics**.
- The **Android app** subscribes to those topics and receives real-time updates from the Raspberry Pi. The app also **publishes commands** (e.g., `door/open`, `door/close`) back to the Raspberry Pi.
- **Flow of Communication**:
  - **Raspberry Pi to App**:
    - The Raspberry Pi continuously monitors the sensors (e.g., PIR motion sensor, ultrasonic ranger) and publishes relevant data to MQTT topics such as:
      - `motion/detected`: Sent when motion is detected.
      - `face/recognized`: Sent when a recognized face is detected by the facial recognition system.
      - `door/status`: Sent when the door is opened or closed.
    - The app **subscribes** to these topics to receive updates in real-time.
  - **App to Raspberry Pi**:
    - The Android app sends **commands** to the Raspberry Pi via MQTT to control the door (e.g., `door/open` to open the door).
    - The app can also send a **manual override** command if the automated system fails.
- **Example Communication (Android)**:
  - The app connects to the MQTT broker running on the Raspberry Pi and subscribes to topics like `motion/detected` and `door/status`. It also publishes messages like `door/open` when the user interacts with the app to control the door.

```
MqttAndroidClient mqttClient = new MqttAndroidClient(context, "tcp://mqttbroker.com", "ClientID");
mqttClient.connect();
mqttClient.publish("door/open", "open".getBytes(), 1, false);  // Open door command
```

- **Example Communication (Raspberry Pi)**:
  - The Raspberry Pi acts as an MQTT publisher, sending data to the app in real time, such as the status of the door and motion detection.

```
import paho.mqtt.client as mqtt

def on_connect(client, userdata, flags, rc):
    print("Connected with result code " + str(rc))
    client.subscribe("door/status")
```

```
def on_message(client, userdata, msg):
    print(f"Received message: {msg.payload.decode()}")  # Handle the message

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect("mqtt-broker-address", 1883, 60)
client.loop_forever()
```

1.2.2 Communication Between Raspberry Pi and Arduino

The **Raspberry Pi** communicates with the **Arduino** to control the **servo motor**, which is responsible for opening and closing the door. The Raspberry Pi sends commands (e.g., `open`, `close`) via **serial communication** (over USB or UART).

- **Flow of Communication**:
  - **Raspberry Pi to Arduino**: The Raspberry Pi sends commands like `open door` or `close door` to the Arduino via the serial connection.
  - The Arduino reads these commands and uses the **PWM signal** to control the servo motor, rotating it to open or close the door.
- **Example Communication (Raspberry Pi to Arduino)**:
  - The Raspberry Pi sends data to the Arduino over **serial communication**, instructing it to rotate the servo motor to specific angles.

```
import serial

# Open serial connection to Arduino (connected via USB or UART)
arduino = serial.Serial('/dev/ttyUSB0', 9600)

# Send command to open door
arduino.write(b'open')
```

  - On the **Arduino side**, the Arduino listens to the serial input and moves the servo motor accordingly:

```
#include <Servo.h>
Servo myServo;

void setup() {
    myServo.attach(9); // Servo connected to pin 9
    Serial.begin(9600); // Start serial communication
}

void loop() {
    if (Serial.available()) {
        char command = Serial.read();
        if (command == 'open') {
            myServo.write(0);  // Open door
        } else if (command == 'close') {
            myServo.write(90); // Close door
        }
    }
}
```

1.2.3 Handling Environmental and Sensor Data

The Raspberry Pi is responsible for monitoring the **environmental conditions** (e.g., motion, temperature, light) through sensors and triggering relevant actions (e.g., starting the facial recognition process, adjusting the camera settings based on lighting).

- **Flow of Data**:
  - The **PIR Motion Sensor** detects motion and sends a signal to the Raspberry Pi, which publishes a message to MQTT to notify the app.
  - The **Raspberry Pi Camera** detects a person's face using **OpenCV** or **TensorFlow Lite**, and if the person's face matches a registered profile, it sends a message to the app indicating that the person is authorized.
  - The **temperature sensor** or **light sensor** adjusts the system's behavior based on external factors. For example, the system may switch to **infrared mode** if the light level is low.
- **Example (Raspberry Pi)**:
  - The Raspberry Pi listens to sensor data and triggers events accordingly.

```
import RPi.GPIO as GPIO
import time

PIR_PIN = 17
GPIO.setmode(GPIO.BCM)
GPIO.setup(PIR_PIN, GPIO.IN)

while True:
    if GPIO.input(PIR_PIN):
        print("Motion detected!")
        # Trigger facial recognition or door control
    time.sleep(1)
```

- The system sends messages to the **Android app** via MQTT when motion is detected or when a face is recognized.

```
client.publish("motion/detected", "Motion detected at the door!")
```

1.3 Hardware Setup and Connections

Let's visualize the **physical setup** and connections between the different components.

- **Raspberry Pi**:
  - The **Raspberry Pi Camera** is mounted near the door to capture images of visitors for facial recognition.
  - The **PIR Motion Sensor** is positioned at the door to detect when someone approaches.
  - The **Ultrasonic Ranger** may be placed near the door to measure proximity.
- **Arduino**:
  - The **servo motor** is connected to the **Arduino** and used to open/close the door. The Arduino is connected to the **Raspberry Pi** via **USB** or **UART** to receive control commands.
- **Android App**:
  - The app communicates via **MQTT** to the **Raspberry Pi**, subscribing to topics like `motion/detected` and `face/recognized`, and publishing commands to control the door (e.g., `door/open`).

1.4 Security and Privacy Considerations

To prevent unauthorized access and ensure the security of data transmissions, the system implements the following security measures:

1. **MQTT Authentication & Encryption**:

   - The MQTT broker will require **username-password authentication** to prevent unauthorized access.
   - TLS encryption (SSL certificates) will be enabled to protect data in transit.

   Example MQTT Secure Connection (Python, Raspberry Pi):

   ```
   client.username_pw_set("user", "password")  # Set authentication
   client.tls_set("path_to_certificate.pem")   # Enable TLS encryption
   client.connect("mqtt-broker-address", 8883, 60)  # Secure connection port
   ```

2. **Facial Recognition Data Protection**:

   - **Local processing**: All facial recognition is performed **locally on the Raspberry Pi**, avoiding cloud-based storage to prevent privacy risks.
   - **Secure Face Data Storage**: If storing face embeddings for recognition, they will be encrypted using **AES-256** encryption.

   Example (Encrypting Face Data with AES-256 in Python):

   ```
   from Crypto.Cipher import AES
   import base64

   key = b'32_byte_secret_key_for_encryption'
   cipher = AES.new(key, AES.MODE_EAX)

   def encrypt_face_data(face_encoding):
       nonce = cipher.nonce
   ```

```
    encrypted_data, tag = cipher.encrypt_and_digest(face_encoding.tobytes())
    return base64.b64encode(nonce + encrypted_data).decode("utf-8")
```

3. **Access Control and Logging**:
   - **Failed attempts** to unlock the door will be logged, and an alert will be sent to the user.
   - Logs of all system interactions (door opened/closed, face recognized, motion detected) will be securely stored and periodically reviewed.

Summary of Part 1: System Architecture & Communication Flow

In this part, we outlined the system architecture, showing how each core component (Raspberry Pi, Arduino, and Android app) fits together. The communication flow is based on the **MQTT protocol**, enabling real-time communication between the **Raspberry Pi** and **Android app**. The **Raspberry Pi** handles the sensor data, camera control, and commands for the **Arduino** to actuate the servo motor.

The system will be flexible, with modular components that can easily be adapted or expanded as the project evolves.

Let's move on to **Part 2: Sensor Integration & Environmental Handling**. In this section, we'll focus on how the different sensors are integrated with the Raspberry Pi, how environmental factors (like lighting and weather conditions) can affect sensor readings, and how the system will handle these variables to make sure it operates reliably.

We will explore how to:

- **Integrate sensors** (motion, light, temperature, etc.) with the Raspberry Pi.
- **Handle environmental conditions** (like day/night cycles, temperature variations) to ensure optimal performance.
- **Calibrate sensors** for accurate readings and adapt to changing conditions.
- Use the sensor data to trigger appropriate actions, such as facial recognition or door control.

Part 2: Sensor Integration & Environmental Handling

2.1 Integrating Sensors with the Raspberry Pi

Your system will rely on several sensors to detect motion, measure distance, and monitor the environment. These sensors will be integrated with the **Raspberry Pi** via **GPIO pins** or the **GrovePi+** board (which simplifies the connection process).

Types of Sensors:

1. **PIR Motion Sensor**:
   - **Purpose**: Detects motion in the vicinity of the door, usually triggered by human movement.
   - **Connection**: Connected to the **GPIO** pins of the Raspberry Pi, or via the **GrovePi+** if you prefer a plug-and-play setup.
   - **Signal**: The sensor outputs a **HIGH signal** when motion is detected, and a **LOW signal** when no motion is detected.
   - **Code Example (Raspberry Pi)**:

```
import RPi.GPIO as GPIO
import time

PIR_PIN = 17
GPIO.setmode(GPIO.BCM)
GPIO.setup(PIR_PIN, GPIO.IN)

while True:
    if GPIO.input(PIR_PIN):
        print("Motion detected!")
        # Trigger facial recognition or send MQTT message
    time.sleep(1)
```

2. **Ultrasonic Ranger (Distance Sensor)**:
   - **Purpose**: Measures the distance to an object (such as a person standing in front of the door). This can help determine when a person is close enough to trigger facial recognition.
   - **Connection**: Typically connected to the Raspberry Pi through GPIO pins or the **GrovePi+** board.
   - **Signal**: The sensor sends out an ultrasonic pulse and calculates the time it takes for the pulse to return. The time is then converted into distance using the speed of sound.
   - **Code Example (Raspberry Pi)**:

```
import RPi.GPIO as GPIO
import time

TRIG = 23
ECHO = 24
GPIO.setmode(GPIO.BCM)
GPIO.setup(TRIG, GPIO.OUT)
GPIO.setup(ECHO, GPIO.IN)

while True:
    GPIO.output(TRIG, GPIO.LOW)
    time.sleep(0.2)
    GPIO.output(TRIG, GPIO.HIGH)
    time.sleep(0.00001)
    GPIO.output(TRIG, GPIO.LOW)

    pulse_start = time.time()
    while GPIO.input(ECHO) == GPIO.LOW:
        pulse_start = time.time()
    pulse_end = time.time()
    while GPIO.input(ECHO) == GPIO.HIGH:
        pulse_end = time.time()

    pulse_duration = pulse_end - pulse_start
    distance = pulse_duration * 17150  # Speed of sound in cm/s
    print(f"Distance: {distance} cm")
    time.sleep(1)
```

3. **Temperature and Humidity Sensor (e.g., DHT20)**:
   - **Purpose**: Measures environmental conditions like temperature and humidity. These readings can be used to ensure optimal sensor performance and can help trigger specific actions when conditions deviate from normal (e.g., too hot or too humid).
   - **Connection**: Connects to the Raspberry Pi via **I2C** or **GPIO**.
   - **Signal**: Outputs digital readings of temperature (in °C) and humidity (in %).
   - **Code Example (Raspberry Pi)**:

```
import Adafruit_DHT

DHT_SENSOR = Adafruit_DHT.DHT22
DHT_PIN = 4  # GPIO pin to which sensor is connected

while True:
    humidity, temperature = Adafruit_DHT.read(DHT_SENSOR, DHT_PIN)
    if humidity is not None and temperature is not None:
        print(f"Temp={temperature}C  Humidity={humidity}%")
    else:
        print("Failed to retrieve data from sensor.")
    time.sleep(2)
```

4. **Light Sensor (LDR - Light Dependent Resistor)**:
   - **Purpose**: Measures ambient light levels, which can affect the facial recognition accuracy. For instance, the system can adjust the camera's sensitivity or switch to infrared mode when light levels are low.
   - **Connection**: Connected to the **GPIO** pins or **GrovePi+**.
   - **Signal**: Outputs an **analog signal** that represents the intensity of light, typically converted to a voltage.
   - **Code Example (Raspberry Pi)**:

```
import RPi.GPIO as GPIO
import time

LDR_PIN = 18  # GPIO pin to which LDR is connected

GPIO.setmode(GPIO.BCM)
GPIO.setup(LDR_PIN, GPIO.IN)

while True:
    light_status = GPIO.input(LDR_PIN)
    if light_status == GPIO.HIGH:
        print("Bright light detected")
    else:
        print("Low light detected")
    time.sleep(1)
```

2.2 Environmental Handling

The system will need to adapt to changing **environmental conditions**, such as light levels, temperature, or the presence of external factors like rain or wind. To ensure the system is both **robust** and **reliable**, we will implement **adaptive behaviors** based on sensor data.

2.2.1 Handling Day/Night Adjustments (Lighting Conditions)

- **Problem**: Light levels affect the camera's ability to perform **facial recognition**. In bright daylight, faces may be overexposed, and in darkness, the system may fail to detect faces altogether.
- **Solution**: The **light sensor** (LDR) will help determine if it is **day** or **night**, allowing the system to adjust its behavior.
  - During the day (when light intensity is above a certain threshold), the system will use normal camera settings for facial recognition.
  - During the night (when light levels are low), the system will either use **infrared lighting** or switch the **Raspberry Pi Camera** to **night vision mode**.
- **Code Example (Adaptive Camera Settings)**:

```
# Check light intensity
if light_intensity > threshold_daytime:
    print("Daytime detected - using normal camera settings")
    # Use regular camera settings
else:
    print("Nighttime detected - using infrared mode")
    # Activate infrared camera mode or external IR light
```

2.2.2 Temperature & Humidity Handling

- **Problem**: Extreme temperature or humidity levels could impact sensor readings (e.g., ultrasonic distance sensor), or even affect the overall system's performance.
- **Solution**: The **temperature and humidity sensor** will provide real-time data to the Raspberry Pi. If the temperature or humidity goes beyond a **safe threshold**, the system can alert the user or adjust its behavior.
  - If the system detects **high humidity**, it might trigger an alert to the user or make adjustments (e.g., slowing down the servo motor to prevent issues).
  - Similarly, the system may adjust camera settings based on temperature, ensuring accurate facial recognition in different environmental conditions.
- **Code Example (Handling Temperature Extremes)**:

```
if temperature > threshold_high_temp:
    print("Warning: High temperature detected!")
    # Take actions (e.g., reduce motor speed, alert user)
elif temperature < threshold_low_temp:
    print("Warning: Low temperature detected!")
    # Take actions (e.g., reduce camera sensitivity)
```

2.2.3 Weather Sensitivity (e.g., Rain or Wind)

- **Problem**: Adverse weather conditions such as **rain** or **strong wind** could affect the operation of sensors, such as the ultrasonic ranger, which relies on sound waves that can be disturbed by weather.
- **Solution**: Although we don't have direct sensors for weather, we can use **environmental data** (e.g., temperature, humidity) or external **weather APIs** to adjust system behavior when conditions are unfavorable.
  - For instance, if **rain** is detected (based on high humidity levels), the system might temporarily disable the ultrasonic ranger to prevent false readings or errors.
- **Code Example (Weather Adjustments)**:

```
if humidity > threshold_rainy:
    print("High humidity detected - potential rain")
    # Adjust system behavior (e.g., stop ultrasonic sensor, alert user)
```

2.3 Calibration and Sensor Tuning

To ensure accurate sensor readings and reliable system behavior, **calibration** is essential.

2.3.1 PIR Motion Sensor Calibration

- **Challenge**: The PIR sensor may be too sensitive, leading to false positives, or not sensitive enough, missing important motion detections.
- **Solution**: Use the **potentiometer** on the PIR sensor to adjust its **sensitivity** so it only triggers when motion is within the desired range.

2.3.2 Ultrasonic Ranger Calibration

- **Challenge**: Ultrasonic sensors can be affected by environmental conditions such as temperature, humidity, and air pressure, leading to inaccurate distance measurements.
- **Solution**: Calibrate the sensor regularly by testing its accuracy at different distances and adjusting the calculation formula based on environmental readings (e.g., temperature).

2.3.3 Light Sensor Calibration

- **Challenge**: The light sensor's threshold for day/night detection may need to be adjusted based on the local environment.
- **Solution**: Test the light sensor under various lighting conditions and fine-tune the threshold to ensure the system adapts correctly to daylight and nighttime conditions.

2.3.4 **Handling Facial Recognition Edge Cases**

While facial recognition is a key feature, certain **real-world challenges** can impact accuracy:

1. **False Positives & False Negatives**:

   - **False Positive**: A stranger is incorrectly recognized as an authorized person.
   - **False Negative**: A valid user is denied access.

   Solution: Implement a Two-Step Authentication Process:

   - If **confidence score < 90%**, request a **manual override via the app** (e.g., PIN entry).
   - Adaptive thresholding based on **lighting conditions**: lower threshold at night.

   Example (Confidence Score Check in Python, OpenCV Face Recognition):

   ```
   if confidence_score < 0.90:
       send_mqtt_message("request_manual_override")
   else:
       unlock_door()
   ```

2. **Lighting Challenges**:
   - Use **infrared camera mode** for night-time recognition.
   - Dynamic brightness adjustments based on **light sensor readings**.
3. **Face Obstructions** (e.g., hats, masks, glasses):
   - Use a second layer of detection based on **eye & nose features**.
   - Notify users if a partially covered face is detected and request manual verification.

Summary of Part 2: Sensor Integration & Environmental Handling

In this part, we discussed how to integrate the various sensors (motion, light, temperature, ultrasonic) with the Raspberry Pi, and how to handle environmental conditions like day/night cycles, humidity, and temperature extremes. We also covered how to calibrate sensors for accurate readings and ensure that the system adapts to changing conditions.

Let's move on to **Part 3: Mobile App Development & Interaction**. In this section, we will focus on how the **Android app** interacts with the system, handles real-time communication with the **Raspberry Pi**, integrates **facial recognition**, and provides the user interface (UI) for controlling the door. We will also look at the overall design and how to ensure the app is user-friendly, especially considering accessibility features.

We'll break down the core aspects of the app, starting with real-time communication, then moving on to facial recognition, app structure, and UI considerations.

Part 3: Mobile App Development & Interaction

3.1 Mobile App Overview

The **Android app** will serve as the **user interface (UI)** for interacting with the **smart door system**. Users will be able to:

- **View real-time notifications** when motion is detected or a visitor is at the door.
- **Respond to visitors** by accepting or denying entry.
- **Open/close the door** remotely.
- **Monitor environmental conditions** (e.g., light, temperature, humidity).
- **Integrate facial recognition** for user authentication and access control.

The app will communicate with the **Raspberry Pi** over **MQTT** for real-time interaction, and it will be designed to handle facial recognition for security and accessibility.

We will use **Java** (or **Kotlin** based on preference) for Android development and integrate with libraries like **Google ML Kit** or **TensorFlow Lite** for face detection/recognition.

3.2 Core Features of the App

Let's break down the key features of the Android app:

3.2.1 Real-Time Communication (via MQTT)

The app will communicate with the Raspberry Pi through **MQTT**, a lightweight messaging protocol suitable for IoT systems. This allows the app to receive real-time updates about the system's status, such as when a visitor is at the door or when motion is detected.

- **App subscribes to topics**: The app will subscribe to specific topics, such as:
  - `motion/detected` (motion sensor triggers).
  - `face/recognized` (facial recognition results).
  - `door/status` (whether the door is open or closed).
- **App publishes commands**: The app will also **publish messages** to MQTT topics to control the door:
  - `door/open` (command to open the door).
  - `door/close` (command to close the door).
- **Real-time updates**: When motion is detected or a face is recognized, the Raspberry Pi will send messages via MQTT, which the app will receive and display to the user.
- **Example MQTT Setup (Android)**:

```
// Setup MQTT client to connect to Raspberry Pi's broker
MqttAndroidClient mqttClient = new MqttAndroidClient(context, "tcp://mqttbroker.com", "ClientID");
mqttClient.connect();

// Subscribe to relevant topics for real-time updates
mqttClient.subscribe("motion/detected", 1);
mqttClient.subscribe("face/recognized", 1);
mqttClient.subscribe("door/status", 1);

// Publish a command to open the door
mqttClient.publish("door/open", "open".getBytes(), 1, false);
```

3.2.2 Facial Recognition

Facial recognition will play a key role in **user authentication** and **visitor verification**. Depending on the design, facial recognition can be performed either on the **mobile app** (using the phone's front-facing camera) or on the **Raspberry Pi** (using the Raspberry Pi Camera).

- **Mobile App**: For security, users may want to verify their identity using **facial recognition** on their phone before they can interact with the system (e.g., opening the door).
- **Raspberry Pi**: The Pi will capture images of visitors at the door, run facial recognition using **OpenCV** or **TensorFlow Lite**, and then send the results (recognized face or unknown face) to the app.
- **Integration with Google ML Kit**: Google ML Kit provides powerful pre-trained models for **face detection**, which can be easily integrated into Android apps.
  - **Face Detection**: Identifies faces in images captured by the phone's camera.
  - **Face Recognition**: Matches detected faces with registered users.
  - **Example Code for Face Detection (Android)**:

```
InputImage image = InputImage.fromBitmap(bitmap, 0);
FaceDetection.getClient().process(image)
    .addOnSuccessListener(faces -> {
        for (Face face : faces) {
            // Process detected face
            Rect bounds = face.getBoundingBox();
            // Draw rectangle or perform recognition
        }
    })
    .addOnFailureListener(e -> {
        // Handle failure
    });
```

3.2.3 Notifications

The app will use **Firebase Cloud Messaging (FCM)** to send push notifications to the user when important events happen, such as motion detection or visitor arrival.

- **Notifications**: The app will notify users in real-time when a visitor approaches the door or when the system detects something important.
- **Example Use of FCM**:
  - When the Raspberry Pi detects motion via the PIR sensor, it will send a message via MQTT. The app will receive the message and send a **push notification** to the user's phone.
  - **Firebase Messaging Setup**:

```
FirebaseMessaging.getInstance().subscribeToTopic("motion_alerts")
    .addOnCompleteListener(task -> {
        String msg = task.isSuccessful() ? "Subscribed!" : "Subscription failed";
        Log.d(TAG, msg);
    });
```

3.2.4 Door Control

Users will be able to control the door via the mobile app. There will be options to **open**, **close**, or **manually override** the system.

- **Manual Control**: In case of failure (e.g., facial recognition malfunction), users will be able to manually control the door from the app.
- **Code Example (Android - Door Control)**:

```
// Open the door when user presses "Open Door"
mqttClient.publish("door/open", "open".getBytes(), 1, false);

// Close the door when user presses "Close Door"
mqttClient.publish("door/close", "close".getBytes(), 1, false);
```

Manual Override (Enhanced Security & Accessibility)

1. **App-Based Manual Override**:

   - If facial recognition **fails twice**, users can input a **pre-set PIN** in the app to manually unlock the door.
   - **Biometric authentication (fingerprint or face unlock)** in the app can also serve as a backup.

   Example (Android Biometric Authentication for Door Unlocking, Java):

```
BiometricPrompt biometricPrompt = new BiometricPrompt(this, executor,
    new BiometricPrompt.AuthenticationCallback() {
        @Override
        public void onAuthenticationSucceeded(BiometricPrompt.AuthenticationResult result) {
            mqttClient.publish("door/open", "open".getBytes(), 1, false);
        }
    });
```

2. **Physical Backup Button (For Accessibility)**:

- A **physical push-button** will allow users with mobility impairments to open the door without using the app.
- Button LED **flashes different colors** to indicate door status.

Example (Physical Button Code for Raspberry Pi, Python):

```python
import RPi.GPIO as GPIO

BUTTON_PIN = 18
GPIO.setmode(GPIO.BCM)
GPIO.setup(BUTTON_PIN, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

def manual_override(channel):
    unlock_door()
    print("Manual override activated")

GPIO.add_event_detect(BUTTON_PIN, GPIO.RISING, callback=manual_override, bouncetime=300)
```

3.2.5 User Interface (UI)

The UI will be **simple** and **intuitive**, designed with accessibility in mind. Users should be able to interact with the app with minimal effort. Key elements include:

- **Main Screen**:
  - **Motion Detection**: A notification or alert appears when motion is detected.
  - **Visitor Info**: If facial recognition is implemented, display the name or status (recognized or not recognized) of the visitor at the door.
- **Door Control**:
  - **Open/Close Buttons**: Buttons to manually control the door.
  - **Override Button**: An emergency manual override button in case of system failure.
- **Settings**:
  - Allow users to configure preferences such as **notification settings**, **sensor thresholds**, and **user profiles** for facial recognition.
- **Accessibility Features**:
  - Large buttons and text for better readability.
  - Voice assistance and simple language for users with limited tech literacy.

3.3 App Structure & File Organization

Here's a potential **file structure** for the Android app, which separates concerns and keeps the code organized:

```
smart-door-app/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── smartdoor/
│   │   │           ├── ui/                # UI layer (Activities, Fragments)
│   │   │           │   ├── home/          # Main screen UI
│   │   │           │   ├── control/       # Door control (open/close)
│   │   │           │   ├── notifications/ # Push notifications
│   │   │           ├── mqtt/              # MQTT client and service
│   │   │           │   ├── MQTTService.java  # MQTT connection and message handling
│   │   │           ├── face/              # Face recognition (using Google ML Kit)
│   │   │           ├── data/              # Models and data handling
│   │   │           ├── utils/             # Utility functions (e.g., data encryption, camera utils)
│   │   ├── res/
│   │   │   ├── layout/                    # XML layouts (UI elements)
│   │   │   ├── values/                    # Strings, colors, styles
├── build.gradle                             # Dependencies and configurations
└── README.md                                # Project overview and setup instructions
```

Key Components:

1. `ui/`:
   - Contains all UI components such as Activities or Fragments, as well as **ViewModels** for managing UI-related data.

2. `mqtt/`:
   - The MQTT client and service responsible for communication with the Raspberry Pi.
3. `face/`:
   - Code responsible for integrating **Google ML Kit** or other face recognition tools.
4. `data/`:
   - Data models for handling information like door status, user profiles, etc.
5. `utils/`:
   - Helper methods for encryption, image handling, and other utility functions.

Summary of Part 3: Mobile App Development & Interaction

In this part, we outlined how the Android app will interact with the **Raspberry Pi** and **Arduino**, focusing on **real-time communication via MQTT**, **facial recognition**, and **door control**. We discussed the key features of the app, such as receiving notifications, responding to visitors, and controlling the door.

We also covered the **UI design** considerations, keeping accessibility in mind, and the **overall file structure** for organizing the app code effectively.

Let's move on to **Part 4: Door Automation and Actuation**. In this section, we will focus on how the door will be automatically controlled using the **servo motor**, how the **Arduino** will interface with the **Raspberry Pi** for motor control, and how the system will handle both automated and **manual overrides**.

We'll cover the setup of the **servo motor**, how the motor will be controlled via the **Arduino**, and how manual control (via app or button) will work if the automated system fails.

Part 4: Door Automation and Actuation

4.1 Controlling the Door with Servo Motors

The **servo motor** will be used to automate the process of opening and closing the door. Servos are ideal for this task because they can rotate to a specific angle, which makes them perfect for controlling a door's movement.

4.1.1 Hardware Setup

1. **Servo Motor**:
   - The **servo motor** is responsible for opening and closing the door.
   - It will be connected to the **Arduino**, which will control the motor's position based on commands received from the **Raspberry Pi**.
   - The servo is typically controlled by **Pulse Width Modulation (PWM)** signals, which define the position of the servo shaft (e.g., 0 degrees to open the door, 90 degrees to close it).
2. **Arduino Setup**:
   - The **Arduino** will be connected to the **Raspberry Pi** through **USB** or **UART (serial communication)**.
   - The **Raspberry Pi** will send **commands** to the Arduino (such as `open` or `close`), and the Arduino will control the servo accordingly.
3. **Power Considerations**:
   - Ensure that the servo motor is properly powered, as it may require a separate **power supply** to function correctly, especially for larger motors. The Raspberry Pi and Arduino will provide signal control, but the servo might require additional current.

4.1.2 Controlling the Servo with Arduino

The **Arduino** will be responsible for receiving commands from the **Raspberry Pi** (via serial communication) and actuating the servo motor.

- **Raspberry Pi** will send commands such as `open` or `close` via serial communication to the Arduino.
- The **Arduino** will interpret these commands and adjust the position of the **servo motor** accordingly.

Example Code for Controlling the Servo (Arduino)

```
#include <Servo.h>

Servo myServo;  // Create a servo object to control the motor

int openPosition = 0;   // Angle for the door to be fully open
```

```
int closedPosition = 90; // Angle for the door to be fully closed

void setup() {
    myServo.attach(9);  // Attach the servo to pin 9 on Arduino
    Serial.begin(9600);  // Begin serial communication at 9600 baud rate
}

void loop() {
    if (Serial.available()) {
        char command = Serial.read();  // Read the incoming byte

        if (command == 'o') {  // Command to open the door
            myServo.write(openPosition);  // Open the door
        }
        else if (command == 'c') {  // Command to close the door
            myServo.write(closedPosition);  // Close the door
        }
    }
}
```

4.1.3 Communication Between Raspberry Pi and Arduino

The **Raspberry Pi** will send commands like `o` (open) or `c` (close) to the **Arduino** to control the door. These commands will be sent via **serial communication** (either USB or UART).

- **Raspberry Pi to Arduino**: The Raspberry Pi sends the command (e.g., "open door") via the serial port to the Arduino.
- **Arduino to Servo**: Based on the command received, the Arduino moves the servo motor to the corresponding position (open or closed).

Example Code for Sending Commands from Raspberry Pi to Arduino (Python)

```
import serial
import time

# Open serial connection to Arduino (connected via USB or UART)
arduino = serial.Serial('/dev/ttyUSB0', 9600)  # Replace with the actual port

# Send command to open door
arduino.write(b'o')  # Send 'o' to Arduino for opening the door
time.sleep(2)  # Wait for 2 seconds

# Send command to close door
arduino.write(b'c')  # Send 'c' to Arduino for closing the door
```

4.2 Manual Override

While automation is the goal, there should always be a way to manually control the door in case of a failure (e.g., if the Raspberry Pi or Arduino malfunctions, or if the facial recognition system fails).

Manual control can be implemented in two ways:

1. **Manual Override via the Mobile App**: The app will have buttons to **open** or **close** the door manually. When the user presses these buttons, the app will send commands to the Raspberry Pi, which will forward them to the Arduino.
2. **Physical Button**: You can also add a **physical override button** (e.g., an LED button) connected to the **Raspberry Pi** or **Arduino**. When pressed, this button will send a signal to open or close the door, bypassing the automated system.

4.2.1 Manual Override via the App

In the event that the automated system fails, the app will allow the user to manually control the door.

- **App sends a command** to the Raspberry Pi to open or close the door.
- The **Raspberry Pi** forwards the command to the **Arduino**, which moves the servo motor accordingly.

Example MQTT Command (Manual Override):

```
// When the user presses "Open Door" in the app
mqttClient.publish("door/open", "open".getBytes(), 1, false);
```

```
// When the user presses "Close Door" in the app
mqttClient.publish("door/close", "close".getBytes(), 1, false);
```

4.2.2 Manual Override via Physical Button

You can add an **LED button** connected to the **Raspberry Pi** or **Arduino** for a **physical override**. This button will allow the user to control the door manually.

- **When the button is pressed**, it will send a signal to the Raspberry Pi or Arduino to trigger the servo motor to open or close the door.

Example Code for Physical Button (Raspberry Pi):

```
import RPi.GPIO as GPIO
import time

BUTTON_PIN = 18  # GPIO pin for the LED button
GPIO.setmode(GPIO.BCM)
GPIO.setup(BUTTON_PIN, GPIO.IN)

while True:
    if GPIO.input(BUTTON_PIN) == GPIO.HIGH:  # Button is pressed
        print("Manual Override: Opening Door")
        # Send command to open the door
        arduino.write(b'o')  # Open door
    else:
        print("Button released")
    time.sleep(1)
```

4.3 Safety Considerations

When automating the door, it's essential to implement **safety measures** to ensure the system remains safe to use, even in the event of failure.

4.3.1 Limit Switches:

- **Limit switches** will be placed at the fully open and fully closed positions of the door to prevent the servo motor from moving beyond its intended range.
- These switches will **cut power to the motor** when the door is fully opened or closed, preventing damage.

4.3.2 Fail-Safe Mechanism:

- If the system encounters a failure (e.g., Raspberry Pi or MQTT communication fails), the system should still allow the door to be manually controlled via the physical button or the app.
- **Alert the user** through the app when the system detects a failure or abnormal behavior (e.g., motor failure, sensor malfunction).

4.3.3 Motor Overload Protection:

- If the servo motor encounters **resistance** (e.g., the door gets stuck), it may fail to open or close the door. You can program the system to detect this and **stop the motor** before damage occurs.
- Implement **error handling** in the Raspberry Pi to monitor the servo's response and alert the user if the motor fails to move.

4.4 Integration with the Mobile App

Once the servo motor control is set up, the mobile app will send commands to the **Raspberry Pi** (via **MQTT**) to control the door's opening and closing.

- **Open Door**: When the user clicks the "Open Door" button in the app, the app will send a message to the Raspberry Pi to trigger the servo motor to open the door.
- **Close Door**: Similarly, when the "Close Door" button is pressed, the app will send a message to close the door.

App-Controlled Door Commands:

```
// Open the door when the user presses "Open Door"
mqttClient.publish("door/open", "open".getBytes(), 1, false);

// Close the door when the user presses "Close Door"
mqttClient.publish("door/close", "close".getBytes(), 1, false);
```

Power Management & Hypothetical Battery Backup Consideration

While the system currently runs on **direct power**, future iterations could incorporate **battery backup** to ensure functionality during power failures. A **12V lithium-ion battery system** or **uninterruptible power supply (UPS)** could be used to sustain the Raspberry Pi and door control mechanism in case of an outage.

Potential Benefits of a Battery Backup System

1. **Uninterrupted Operation:**
   - Allows the door to function **for a limited time** (e.g., **6–12 hours**) during a power outage.
   - Ensures that **facial recognition and manual overrides** still work even if the main power is lost.
2. **Low-Battery Alerts to App:**
   - The Raspberry Pi could monitor its power source and send an **MQTT alert** when switching to battery mode or when power is critically low.

Example (Detecting Power Failure & Sending Alert, Python, Raspberry Pi)

```
import os
import time

def check_power_status():
    power_state = os.system("vcgencmd get_throttled")  # Checks power issues on Raspberry Pi
    if power_state != 0:
        send_mqtt_message("power_failure_detected")
        print("Warning: System running on backup battery!")

while True:
    check_power_status()
    time.sleep(60)  # Check power every 60 seconds
```

3. **Energy-Saving Mode:**
   - When running on battery, the system could enter **"low-power mode"**, turning off **non-essential features** (e.g., reducing camera frame rate, disabling real-time notifications).
   - The door could **default to manual override mode** to conserve power.

Future Implementation Plan

- If integrated, the **battery system** could use a **relay switch** to **automatically shift between power sources**.
- The **MQTT app interface** could display **battery status**, allowing users to monitor backup levels.

Summary of Part 4: Door Automation and Actuation

In this part, we covered how the **servo motor** is used to automate the opening and closing of the door. We also discussed how the **Raspberry Pi** communicates with the **Arduino** to control the motor and how **manual overrides** (via the mobile app or physical button) will allow the user to bypass the automated system if needed.

We also highlighted **safety considerations** like **limit switches** to prevent over-rotation and **fail-safe mechanisms** to ensure the door can still be operated manually in case of system failure.

Let's move on to **Part 5: Testing & Validation**. In this section, we will focus on testing the entire system to ensure that it works as expected. We will cover the process of **unit testing**, **integration testing**, **stress testing**, **user testing**, and **performance testing**. Each of these testing types ensures that different parts of the system function properly, both individually and together.

We'll also explore how to handle edge cases, simulate real-world scenarios, and collect feedback to make sure the system is robust, reliable, and user-friendly.

Part 5: Testing & Validation

5.1 Unit Testing

Unit testing is the process of testing individual components of the system to ensure they work as expected. Since your system includes both **hardware** (sensors, motors, etc.) and **software** (Android app, Raspberry Pi code), you'll need to write unit tests for both.

5.1.1 Testing Raspberry Pi Code

For the Raspberry Pi, you can use **PyTest** to test the Python code for hardware interactions and sensor readings.

1. **Testing Sensors**: Test each sensor individually to ensure they are reading data correctly.

   - **PIR Motion Sensor**: Verify that the sensor correctly detects motion and triggers the appropriate action (e.g., sending an MQTT message).
   - **Ultrasonic Ranger**: Test the sensor to ensure it accurately measures distances and triggers actions based on proximity.
   - **Temperature and Humidity Sensor**: Verify that the sensor reads the correct temperature and humidity values, and triggers an action if the values go beyond safe thresholds.

   Example of Unit Test for PIR Motion Sensor (Python):

   ```
   import pytest
   from sensor_module import PIRMotionSensor  # Assuming you have this module

   def test_motion_detection():
       sensor = PIRMotionSensor(pin=17)
       assert sensor.detect_motion() == True  # Simulate motion and check if the sensor detects it
   ```

2. **Testing Servo Motor Control**: Test that the servo motor responds correctly to commands from the Raspberry Pi.

   - **Test servo movement**: Ensure that sending an `open` or `close` command moves the motor to the correct position (e.g., 0° for open and 90° for closed).

   Example of Unit Test for Servo Motor Control:

   ```
   def test_servo_motor():
       motor = ServoMotor(pin=18)
       assert motor.move_to_position(0) == True  # Check if motor moves to 0 degrees
       assert motor.move_to_position(90) == True  # Check if motor moves to 90 degrees
   ```

5.1.2 Testing Android App Code

For the Android app, you can use **JUnit** for unit tests and **Espresso** for UI tests.

1. **Unit Tests**:

   - **Test MQTT Client**: Test that the app correctly subscribes to MQTT topics and handles messages from the Raspberry Pi.
   - **Test Face Recognition**: Test that the app correctly captures faces and compares them against registered users.

   Example Unit Test for MQTT Client (Android):

   ```
   @Test
   public void testMqttClientConnection() {
       MqttClient mqttClient = new MqttClient();
       assertTrue(mqttClient.isConnected());  // Check if the MQTT client successfully connects
   }
   ```

2. **UI Tests**:

   - **Test App Interaction**: Test that the UI elements (buttons, notifications) work correctly and trigger the appropriate actions.
   - **Test Push Notifications**: Test that push notifications are received when motion is detected or when a visitor arrives.

   Example UI Test with Espresso:

```
@Test
public void testOpenDoorButton() {
    onView(withId(R.id.open_door_button)).perform(click());  // Simulate button click
    onView(withId(R.id.door_status)).check(matches(withText("Door is open")));  // Verify that the status
updates
}
```

5.2 Integration Testing

Integration testing ensures that different parts of the system (software and hardware) work together as expected. You will test the flow from sensor data collection to door actuation and verify that the app communicates correctly with the Raspberry Pi.

5.2.1 Testing Raspberry Pi & Sensors Integration

1. **Test Motion Detection and Door Opening**:
   - Simulate motion being detected by the PIR sensor. When motion is detected, the system should trigger the Raspberry Pi camera for facial recognition and send an MQTT message to the app.
2. **Test Ultrasonic Sensor and Door Control**:
   - When someone approaches the door, the ultrasonic sensor should detect the person's distance and trigger the camera. If a valid face is recognized, the Raspberry Pi should send a command to open the door.

5.2.2 Testing Mobile App & Raspberry Pi Communication

1. **Test MQTT Communication**:
   - Ensure that the mobile app can successfully subscribe to MQTT topics like `motion/detected` and `door/status`, and correctly receive real-time updates when motion is detected or when the door's status changes.
2. **Test App Commands for Door Control**:
   - When the user presses the "Open Door" button in the app, it should send the correct message (`door/open`) to the Raspberry Pi, which will then trigger the servo motor.

5.3 Stress Testing

Stress testing involves testing the system under extreme or unexpected conditions to ensure it can handle high traffic, device failures, or environmental challenges.

5.3.1 Simulating High Traffic

- **Simulate multiple visitors** or **motion events** in quick succession. Ensure that the system can handle multiple MQTT messages, facial recognition requests, and servo control commands without lag or failure.

5.3.2 Test System Behavior in Adverse Environmental Conditions

- **Test the system under different lighting conditions**, such as bright daylight and low-light situations (e.g., nighttime).
- Simulate **temperature extremes** to see how the system behaves (e.g., sensor readings under extreme temperatures or high humidity).

5.3.3 Test Motor Overload

- Simulate **resistance** (e.g., a heavy door) to see if the motor will handle the load and prevent system failure.

High-Traffic Stress Testing

1. **Simulating Concurrent Access**:

   - Simulate **multiple people arriving at the door at once** by running parallel MQTT messages.
   - Ensure system doesn't **lag or crash under heavy load**.

   Example (Simulating High Traffic, Python):

```
import threading

def simulate_visitor():
```

```
        send_mqtt_message("motion/detected")
        send_mqtt_message("face/recognized")

    for _ in range(10):  # Simulating 10 people at once
        threading.Thread(target=simulate_visitor).start()
```

5.4 User Testing & Feedback

User testing ensures the system is user-friendly, accessible, and meets the needs of the target audience, especially those with disabilities.

5.4.1 UI/UX Testing

- Have users interact with the app, focusing on its **accessibility** (e.g., font size, button size, voice commands).
- Test with users who have **limited mobility**, **visual impairments**, or other disabilities to ensure the app is easy to use.

5.4.2 Test Facial Recognition Usability

- Ensure the **facial recognition** system works across different users with varying facial features (e.g., glasses, hats, different lighting conditions).
- Gather feedback to improve accuracy and response time.

5.4.3 Test Manual Override

- Ensure the **manual override** feature works smoothly, both through the app and using a physical button. This is especially important if the system is inoperable or malfunctioning.

5.5 Performance Testing

Performance testing evaluates the system's responsiveness and stability under typical and peak conditions.

5.5.1 Response Time

- Measure the time it takes for the system to respond to commands. For example, when motion is detected, how long does it take for the system to activate the camera and send a notification to the app?
- When the user presses the "Open Door" button, how quickly does the system respond by triggering the servo motor?

5.5.2 System Resource Usage

- Monitor the **Raspberry Pi's** CPU and memory usage during extended operation to ensure that it doesn't become overloaded or unresponsive when handling sensor data, facial recognition, and motor control.

Summary of Part 5: Testing & Validation

In this part, we covered the different types of testing required to ensure the system works as expected:

- **Unit Testing**: Focused on testing individual components, such as sensors and the mobile app's MQTT functionality.
- **Integration Testing**: Verified that the Raspberry Pi, sensors, and mobile app work together seamlessly.
- **Stress Testing**: Ensured that the system can handle high traffic, environmental factors, and motor overloads without failure.
- **User Testing**: Tested the system with real users to ensure that the app is user-friendly and accessible, especially for people with disabilities.
- **Performance Testing**: Focused on ensuring fast response times and efficient resource usage.

Let's dive into **Part 6: Smart Home Integration & Future Considerations**. This section will focus on how to extend your **smart door system** to integrate with existing **smart home platforms**, such as **Google Home**, **Amazon Alexa**, and **Apple HomeKit**. We will also explore **future enhancements** that can be added to the system, ensuring scalability, future-proofing, and flexibility for adding new features as the project evolves.

Part 6: Smart Home Integration & Future Considerations

6.1 Integrating with Smart Home Ecosystems

Smart home integration is key to making the system more powerful and allowing users to interact with the door in multiple ways, not just through the app. By integrating with popular ecosystems like **Google Home**, **Amazon Alexa**, and **Apple HomeKit**, the door system can be controlled via voice commands, other smart devices, or centralized hubs.

6.1.1 Google Home Integration

To integrate with **Google Home**, you'll need to create a **Google Assistant Smart Home Action**. This allows users to control their smart door system via voice commands using Google Assistant.

- **MQTT to Google Assistant Bridge**: You can use services like **IFTTT** or **Node-RED** to bridge your MQTT messages to Google Assistant. This means that the door system can communicate via MQTT while still allowing Google Home to control it.
- **Voice Commands**: Users can say things like:
  - "Hey Google, open the door."
  - "Hey Google, is the door open?"
  - "Hey Google, close the door."
- **Google Assistant SDK**: The SDK allows you to build custom actions and integrate them with Google Assistant. This will enable the smart door system to be recognized as a device within Google Home.

Steps for Integration:

1. Set up an **MQTT broker** on the Raspberry Pi.
2. Create a **Google Smart Home Action** in the Google Actions Console.
3. Use **IFTTT** or **Node-RED** to create a flow that translates MQTT messages to Google Assistant commands.

6.1.2 Amazon Alexa Integration

To integrate with **Amazon Alexa**, you will need to create a **Smart Home Skill**. This skill allows users to control the door via voice commands using Alexa-enabled devices.

- **Alexa Smart Home Skill**: The skill will be built using **Alexa Skills Kit (ASK)** and a backend API to communicate with the Raspberry Pi. It will allow voice commands like:
  - "Alexa, open the door."
  - "Alexa, close the door."
  - "Alexa, is the door open?"
- **Alexa to MQTT Bridge**: You can use **Node-RED** to set up the integration, or if you have a backend API, you can build a custom bridge to forward the MQTT messages to Alexa.

Steps for Integration:

1. Create an **Alexa Smart Home Skill** using the **Alexa Skills Kit**.
2. Use **AWS Lambda** or another cloud service to handle requests and communicate with the Raspberry Pi.
3. Use **Node-RED** or a custom MQTT integration to relay commands to the Raspberry Pi.

6.1.3 Apple HomeKit Integration

For **Apple HomeKit** integration, the easiest way is to use **Homebridge**, an open-source platform that allows non-HomeKit devices to be exposed as HomeKit-compatible devices.

- **Homebridge Setup**: Homebridge acts as a bridge between your Raspberry Pi and Apple HomeKit, allowing users to control their smart door via the **Home app** on iOS or Siri.
- **Voice Commands**: Once integrated, users can say:
  - "Hey Siri, open the door."
  - "Hey Siri, close the door."
  - "Hey Siri, is the door open?"
- **Homebridge Plugin**: You will need to write or use an existing Homebridge plugin to connect the **MQTT broker** to HomeKit.

Steps for Integration:

1. Set up **Homebridge** on the Raspberry Pi.
2. Install a **Homebridge MQTT plugin** to connect MQTT topics to HomeKit.
3. Use the **Home app** on iOS to add your device and control it via Siri.

6.2 Future Considerations & Scalability

As your system evolves, you might want to add more features or devices. The key to scalability is designing the system to be **modular** and **expandable**. This way, adding new features doesn't disrupt existing functionality.

6.2.1 Adding More Smart Devices

As you move beyond the door, consider integrating other **smart devices** into your system. These could include:

- **Smart lights**: Integrate with the system to automatically turn on lights when someone arrives or when the door is opened.
- **Smart locks**: Use the same system to control **smart locks** on the door. When the door opens, the lock could automatically engage or disengage.
- **Cameras**: Add additional cameras or integrate with existing smart security cameras to monitor the area in front of the door.

To integrate these devices, you can use **IoT platforms** like **Thingsboard** or **Node-RED**, which provide the flexibility to add new devices and manage them within a unified ecosystem.

6.2.2 User Profiles and Access Levels

As your system grows, you may want to implement **multiple user profiles**. This can allow different individuals to use the system based on their specific permissions, especially useful in households or multi-person offices.

- **Facial Recognition Profiles**: Each registered user can have their own facial profile in the system. The door could then recognize them and grant access automatically.
- **Access Levels**: Create different access levels for different users. For example:
  - **Admin**: Full control over all settings (adding/removing users, modifying system settings).
  - **User**: Basic control, such as opening/closing the door and using the manual override.
  - **Guest**: Limited access, such as temporary facial recognition or password access.

6.2.3 Cloud Integration

While the system can run locally (on the Raspberry Pi), integrating **cloud-based services** allows for:

- **Remote Control**: Users can access and control the door remotely via an app or web interface, even when they're not at home.
- **Data Storage**: Collect data on when the door was opened/closed, who accessed it, and environmental conditions. This data can be stored securely in the cloud and accessed via the app for analytics.

Popular cloud platforms for IoT include:

- **AWS IoT**: Provides a complete IoT solution for device management, data collection, and analytics.
- **Google Cloud IoT**: Offers secure device connections, real-time analytics, and machine learning integration.
- **Azure IoT**: Similar to AWS and Google Cloud, with strong integration with Microsoft products.

6.2.4 Deep Learning and AI

With **deep learning** and **AI**, you can further enhance the capabilities of the system:

- **Advanced Facial Recognition**: Use deep learning models to improve **accuracy** in recognizing faces, even in poor lighting or with people wearing accessories (e.g., glasses, hats).
- **Smart Detection**: Use AI-based image classification to distinguish between different objects, such as recognizing whether the person at the door is a **visitor** or an **intruder**.

For example, you could use pre-trained **TensorFlow Lite** models or **OpenCV** for advanced image recognition tasks.

6.2.5 Automated Events and Actions

Once you've integrated the system into a larger smart home ecosystem, you can automate actions based on certain events. For example:

- **Automated Door Locking**: When the door is closed, automatically lock it after a set period of time.
- **Visitor Notifications**: Send a custom notification to the app when a **specific person** (e.g., a family member or friend) arrives.
- **Lighting Control**: Turn on the porch light when motion is detected or when the door is opened.

6.3 Potential Future Features

As the project progresses, you can explore these potential features:

- **Voice Control**: Once integrated with **Google Home** or **Alexa**, add more voice control features. For example, the system could announce who is at the door using a **text-to-speech** API, or even ask the user if they want to open the door.
- **Remote Access**: Allow the user to unlock the door remotely via a secure app, especially useful for delivery services or service personnel.
- **Weather Integration**: Integrate with weather APIs to adjust the system based on **weather conditions** (e.g., disable the ultrasonic ranger during heavy rain).

Integration with Smart Locks

- The system will integrate with **electronic smart locks** (e.g., August, Yale) via **Zigbee or Z-Wave**.
- If facial recognition succeeds, the **app will send a command** to unlock the smart lock.

  Example (Unlocking a Zigbee Smart Lock via Python API):

```
import zigbee_api

def unlock_smart_lock():
    zigbee_api.send_command("unlock")
```

Summary of Part 6: Smart Home Integration & Future Considerations

In this part, we explored how to integrate the **smart door system** with existing **smart home platforms** like **Google Home**, **Amazon Alexa**, and **Apple HomeKit**. We also discussed **future considerations** for scalability, such as adding more smart devices, implementing **multiple user profiles**, integrating with the **cloud**, and leveraging **deep learning** for facial recognition and smart detection.

We also looked at ways to automate tasks, improve system security, and add advanced features that enhance the user experience.

Let's move on to **Part 7: Deployment & Maintenance**. This part will focus on the process of **deploying** your system, getting it ready for real-world use, and ensuring it runs smoothly in the long term through proper **maintenance**. We will also discuss how to manage updates, handle potential issues, and prepare for future expansions or feature enhancements.

Part 7: Deployment & Maintenance

7.1 Deployment Process

The deployment process involves getting your **smart door system** up and running in a real-world environment. This involves setting up both the **hardware** (Raspberry Pi, sensors, servo motor, etc.) and the **software** (mobile app, Raspberry Pi code, cloud services).

7.1.1 Raspberry Pi Deployment

1. **Setting Up the Raspberry Pi**:
   - Install the **Raspberry Pi OS** on your **SD card** (the operating system that will run the Raspberry Pi).
   - Set up the **MQTT broker** on the Raspberry Pi using **Mosquitto** or **HiveMQ**. This will handle the communication between the Raspberry Pi and the mobile app.
   - Connect the necessary sensors (e.g., PIR motion sensor, ultrasonic ranger, camera) to the **GPIO pins** of the Raspberry Pi. Make sure each sensor is properly configured and tested before deploying.
2. **Raspberry Pi Software Configuration**:
   - Install the necessary libraries and dependencies, such as **Python libraries** for sensor communication (**RPi.GPIO**, **OpenCV**, etc.).
   - Set up **serial communication** between the Raspberry Pi and Arduino for controlling the servo motor.
   - Write the code that integrates the sensors, controls the motor, and communicates with the mobile app via MQTT.
3. **Testing the Raspberry Pi in the Real Environment**:
   - Once the Raspberry Pi is configured, **test the system in its physical location** (in front of the door).
   - Verify that the sensors are accurately detecting motion and distance and that the camera is capturing clear images for facial recognition.
   - Ensure that the door opens and closes reliably using the servo motor. Test the system in different lighting conditions (day/night) to ensure facial recognition works correctly.

7.1.2 Mobile App Deployment

1. **Building the Android App**:
   - Use **Android Studio** to build and compile the app.
   - Test the app on multiple devices to ensure compatibility (screen sizes, OS versions, etc.).
   - **Connect the app to the Raspberry Pi's MQTT broker**, and ensure the app subscribes to relevant topics like `motion/detected`, `door/status`, etc.
2. **App Testing**:
   - Test the mobile app's functionality, including:
     - Real-time notifications when motion is detected.
     - Receiving the camera feed when a person approaches the door.
     - Controlling the door (open/close) through the app.
     - Facial recognition and manual override.
3. **Deploy the App**:
   - Once the app is tested, you can publish it to the **Google Play Store** or distribute it through **internal testing channels** for further user feedback.
   - Ensure the app is optimized for both performance and user experience, focusing on intuitive controls and accessibility.

7.1.3 Cloud & Smart Home Integration

1. **Setting Up Cloud Services**:
   - If you've integrated cloud-based services (e.g., **AWS IoT**, **Google Cloud IoT**, or **Azure IoT**), make sure the cloud infrastructure is set up correctly to handle device management, real-time data, and remote access.
   - Ensure data is securely stored in the cloud (if applicable) and that users can remotely access their system without issues.
2. **Smart Home Integration**:
   - If the system is integrated with **Google Home**, **Alexa**, or **Apple HomeKit**, make sure to test voice commands and automation features after deployment.
   - Verify that commands like "Open the door" or "Close the door" work reliably through these platforms.

7.2 Maintenance

Maintaining the system is critical to ensure it continues to function smoothly over time. Regular maintenance includes updates to the software, system monitoring, hardware checks, and improvements based on user feedback.

7.2.1 Software Maintenance

1. **Mobile App Updates**:
   - Regularly update the **Android app** to address **bug fixes**, **performance improvements**, and **new features**. Ensure that the app remains compatible with the latest Android versions and devices.
   - Push updates to users through the **Google Play Store** or **internal testing channels**.
2. **Raspberry Pi Software Updates**:
   - Keep the **Raspberry Pi OS** and libraries up to date to ensure security and performance. You can use **cron jobs** or other scheduling tools to periodically check for updates.
   - Ensure that the **MQTT broker** and other services (e.g., camera, sensor modules) are functioning properly after each update.
3. **Cloud Service Maintenance**:
   - If you're using cloud services for remote access or data storage, regularly monitor the **cloud infrastructure** for performance and security.
   - **Scaling**: As the number of users grows, you may need to scale your cloud infrastructure to handle more devices or data.

7.2.2 Hardware Maintenance

1. **Sensor Calibration**:
   - **Re-calibrate** sensors periodically (e.g., PIR sensor, ultrasonic ranger) to ensure accurate readings.
   - Perform periodic **hardware checks** to ensure that sensors, motors, and the camera are in good working condition.
2. **Servo Motor Maintenance**:
   - Check the **servo motor** periodically to ensure it's operating smoothly. If the door feels resistant or the motor struggles to open/close the door, it may need maintenance or replacement.
   - Ensure that **limit switches** are correctly positioned and functioning to stop the motor at the proper positions (open/close).
3. **Camera Maintenance**:
   - Periodically **clean the camera lens** to ensure clear image capture.
   - If using **night vision** or infrared lighting, ensure that those components are not obstructed or malfunctioning.

7.2.3 Monitoring System Performance

1. **System Performance Metrics**:
   - Monitor system performance, such as response times (e.g., from motion detection to door opening) and overall app performance. Use tools like **Firebase Analytics** or **Google Analytics** for tracking user interactions and app performance.
2. **Error Logs**:
   - Implement **logging** on the Raspberry Pi to track errors, such as sensor malfunctions or communication issues with the mobile app.
   - Regularly review these logs to address any recurring issues.
3. **User Feedback**:
   - Collect user feedback via the mobile app (e.g., through surveys or feature requests).
   - Based on the feedback, implement new features or improvements. This can include adding more user profiles, improving facial recognition accuracy, or implementing new integrations (e.g., smart locks, cameras).

7.3 Handling System Failures and Troubleshooting

1. **System Failure Recovery**:
   - Ensure there are backup plans in place if the system fails. For example, if the **Raspberry Pi** stops working, users should still be able to open or close the door manually via the app or physical button.
2. **Troubleshooting**:
   - Provide a **troubleshooting guide** within the app to help users resolve common issues, such as:
     - Motion sensor not detecting movement.
     - Door not responding to app commands.
     - Facial recognition not working in low light.
   - In the event of hardware failure (e.g., servo motor malfunction), provide **guidelines** on how users can replace faulty components or contact support.

Handling System Failures

1. **Automatic Recovery from Crashes**:

   - A **watchdog script** will automatically restart the Raspberry Pi if it becomes unresponsive.

   Example (Watchdog Script, Raspberry Pi, Python):

```
import os

def restart_if_crashed():
    if os.system("ping -c 1 mqtt-broker-address") != 0:
        os.system("sudo reboot")

restart_if_crashed()
```

Summary of Part 7: Deployment & Maintenance

In this part, we discussed the process of **deploying** your system and getting it up and running in a real-world environment. This includes setting up the **Raspberry Pi**, deploying the **mobile app**, and integrating with **cloud services** and **smart home ecosystems**.

We also covered the importance of **maintenance**, including software updates, hardware checks, and system performance monitoring. Additionally, we discussed how to handle **system failures**, perform **troubleshooting**, and ensure that the system continues to function smoothly over time.

Part 8: Prototyping with Cardboard & Presentation

8.1 Overview of Cardboard Prototyping

Using **cardboard** for prototyping is an efficient way to quickly assemble and test the smart door system's hardware components before finalizing the design with more durable materials. It is especially useful during the early stages of development when flexibility and quick changes are required.

The **cardboard prototype** provides a low-cost, quick-to-build platform for testing various elements of the system, including the **servo motor**, **sensors**, **camera positioning**, and **user interface** (such as buttons or mounts). It allows for easy adjustments to design and functionality based on real-world testing.

8.2 Materials & Tools

To prototype your smart door system with cardboard, you'll need the following materials and tools:

- **Cardboard Sheets**: Use sturdy cardboard for the door frame and mounts. Consider using corrugated cardboard for additional durability.
- **Scissors/Utility Knife**: To cut the cardboard into appropriate shapes for the door, mounts, and components.
- **Glue/Tape**: For assembling the parts of the prototype and securing components in place.
- **Servo Motor and Arduino**: These will be the core components for opening and closing the door.
- **Sensors (PIR, Ultrasonic Ranger, etc.)**: To test motion detection and proximity sensing.
- **Camera**: To simulate facial recognition functionality.
- **LED Button (optional)**: For the manual override button.

8.3 Assembly of the Cardboard Prototype

1. **Cardboard Door Frame**:
   - Cut cardboard sheets to create a basic door frame. This frame will hold the servo motor and allow it to open and close.
   - Use **flaps** of cardboard for added structural strength if necessary.
2. **Servo Motor Mounting**:
   - Create a mount for the **servo motor** that holds it in place while it rotates the door. Use tape or glue to secure the motor on the cardboard frame.
   - Ensure that the **servo motor shaft** is aligned with the edge of the door so that it can rotate and simulate opening/closing the door.
3. **Sensor Placement**:
   - Mount the **PIR motion sensor** and **ultrasonic ranger** on the cardboard frame to test motion detection and proximity.
   - Ensure the **PIR sensor** is positioned at a height where it can accurately detect motion, typically around 1.5 to 2 meters from the ground.
   - Position the **ultrasonic sensor** so it can detect objects approaching the door.
4. **Camera Mount**:
   - Create a simple stand or mount for the **Raspberry Pi Camera** to be positioned at the door. This will help test facial recognition and camera angles.
5. **Manual Override Button**:
   - If you're testing a **physical button** for manual control, place it on the cardboard setup. This will allow you to test manual overrides when the system fails or for user interaction.

8.4 Testing & Iteration

Once the cardboard prototype is assembled, conduct real-world tests:

- **Motion Detection**: Test the PIR motion sensor to ensure that it detects movement accurately.
- **Proximity Detection**: Use the ultrasonic sensor to simulate different distances and ensure the system can accurately trigger actions when someone approaches.
- **Servo Operation**: Ensure the servo motor moves the door smoothly and responds to open/close commands from the app or Raspberry Pi.
- **Camera Positioning**: Test the camera placement to ensure it captures clear images for facial recognition.
- **User Feedback**: Use the cardboard prototype to demonstrate the system to others and gather feedback on the design, usability, and ease of interaction.

8.5 Aesthetic Presentation

Even though the prototype is made from cardboard, presenting it neatly can give stakeholders a better understanding of the final product. Ensure the cardboard is cut cleanly, and consider:

- **Neatly cutting** the cardboard for a polished look.
- **Labeling components** (e.g., sensors, motors) to make it easier for others to understand the setup.
- **Organizing wiring** and making the design visually clean for demonstrations.

You can even use markers or paint to add color or simulate the final design, giving it a more professional look for presentations.

8.6 Transition to Final Build

Once you've gathered feedback from the cardboard prototype and made necessary adjustments to your design, you'll be ready to transition to more durable materials for the final product.

- **Material Considerations**: After testing with cardboard, consider switching to materials like **acrylic**, **wood**, or **3D printed parts** for the final product, depending on your project requirements and budget.

- **Final Testing**: Once the final materials are selected and the system is assembled, repeat testing to ensure everything works as expected before full deployment.

8.7 Summary of Cardboard Prototyping

Cardboard prototyping is a cost-effective and flexible method to rapidly assemble and test the key components of your smart door system. It provides invaluable insights into the physical design, sensor placement, and overall system functionality, helping you refine the design before transitioning to more permanent materials. By using cardboard for testing and presentation, you ensure flexibility and adaptability throughout the early stages of your project.
