Department of Computer Science & Engineering
The University of Texas at Arlington

Detailed Design Specification
CSE 4317: Senior Design II
Fall 2024

Smart Crib IoT Team
Smart Crib

Don Dang
Luis Carrillo
Zait Martinez
Pranav Pujar

Revision History

Revision	Date	Author(s)	Description
0.1	10.12.2024	DD	document creation
0.2	10.14.2024	DD, LC	complete draft
0.3	10.15.2024	DD	updated logo
1.1	10.15.2024	DD, ZM, PP	imported subsystem images
1.1	10.15.2024	DD, LC	revised layer subsystems
1.0	10.18.2024	DD, LC	official release

CONTENTS

List of Figures

1 INTRODUCTION

The Smart Crib application is an IoT-based smart home management system designed to provide users with control over various connected home devices, including smart lights, door locks, and fans. The application integrates with an ESP32 and Raspberry Pi microcontroller-based setup and communicates with a server to manage and monitor the status of these devices in real-time. The product aims to enhance the convenience for home environments by offering intuitive controls via a mobile interface.

2 SYSTEM OVERVIEW

The Smart Crib application architecture is designed with a layered approach, ensuring efficient communication between the mobile interface, the server, and the IoT devices. The system's data flow diagram, as outlined in the architectural specification, details three key layers: the Application Layer, the Server Layer, and the Device Layer. The Application Layer is the front-end interface developed using React Native. It provides users with a seamless and intuitive way to interact with their connected devices, such as adjusting light brightness, managing fan speed, and controlling door locks. The client sends user commands as HTTP requests to the server and displays device statuses based on the server's responses.

The Server Layer acts as an intermediary between the client and the devices. It is built using a Python-based HTTP server that listens for and processes requests from the client layer. The server updates device states accordingly and sends relevant information back to the client. It is designed to handle multiple requests efficiently using a multi-threaded approach, ensuring that device operations and data consistency are maintained. Lastly, the Device Layer comprises the IoT devices controlled by an ESP32 microcontroller. These devices include smart lights, fans, and door locks, which receive instructions from the server and periodically report their status. The microcontroller translates the server's instructions into physical actions while keeping the system synchronized by sending back status updates. This multi-layered architecture ensures a responsive and reliable smart home experience for users.

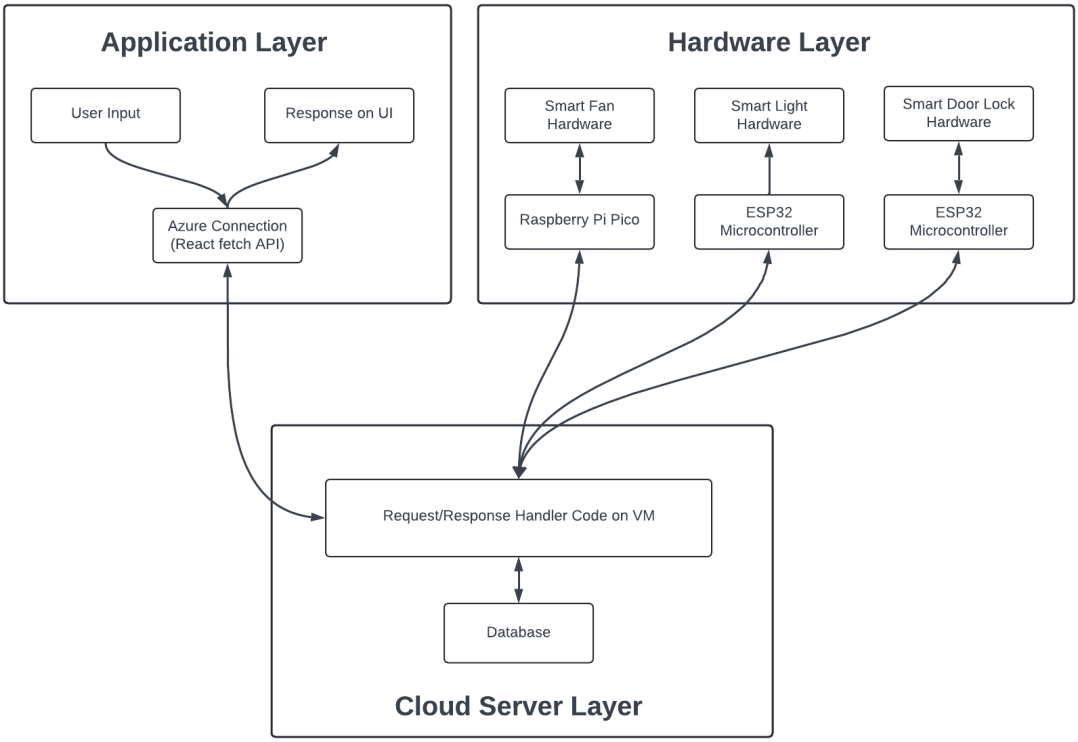

Figure 1: System Architecture

3 Application Layer Subsystems

The Application Layer of the Smart Crib system is responsible for providing users with an interface to interact with their smart home devices, such as smart lights, door locks, and fans. This layer is purely software-based and does not rely on specific hardware components. It is implemented using JavaScript and React Native, which is a popular framework for building cross-platform mobile applications. This approach allows the app to be deployed on both iOS and Android devices, ensuring a broad user base without needing to manage separate codebases.

The application utilizes several key frameworks and dependencies to enhance its functionality. React Native serves as the primary framework for developing the mobile interface, providing components and APIs for building user interfaces and handling mobile-specific functionalities like navigation and gesture handling. The navigation system is created using React Navigation, which facilitates a tab-based structure that allows users to switch easily between different device controls, such as Lights, Door lock, and Fan. Axios, an HTTP client, is used for making API requests from the application to the server, ensuring smooth and efficient data exchange. Additionally, Ionicons is used to add icons in the tab navigation, enhancing the visual appeal and providing users with intuitive navigation cues.

In terms of compatibility, the application is designed to work on both iOS and Android mobile operating systems, providing flexibility and accessibility for various types of mobile devices. The UI design focuses on simplicity and responsiveness, with a tab-based navigation system that lets users switch between device controls seamlessly. The app displays current device statuses and allows users to control various devices with minimal input, providing a user-friendly and efficient interface.

The Application Layer communicates with the server layer using HTTP protocols. It sends requests via Axios, and the server responds with updates based on user interactions, enabling real-time control of smart home devices. The network communication is optimized to ensure low latency, offering users a responsive and seamless experience. This layered approach ensures that the Application Layer integrates effectively with the rest of the system, providing a consistent and reliable experience across different mobile platforms.

3.1 Application Layer Hardware

The Application Layer of the Smart Crib system is primarily software-based and does not involve any specific hardware components directly. This layer operates as a mobile application running on users' smartphones or tablets, which serves as the primary interface for controlling and monitoring smart home devices. The devices that can run the application include iOS and Android smartphones, tablets, and other mobile devices capable of supporting React Native applications. The hardware requirements for these devices are minimal, as the application is designed to be lightweight, ensuring compatibility across a wide range of devices.

3.2 Application Layer Operating System

The Application Layer of the SmartCrib system operates on mobile devices running either iOS or Android operating systems. The application is developed using React Native, a cross-platform framework that allows the same codebase to be used for both operating systems, ensuring consistent functionality and user experience across devices.

On iOS, the application is compatible with devices running iOS 12.0 or later, while on Android, it supports versions from Android 5.0 (Lollipop) onwards. React Native's integration with native modules allows the application to utilize device capabilities such as network communication and sensors efficiently on both platforms.

No additional operating systems are required specifically for this layer beyond the mobile OS versions mentioned. The application is designed to be lightweight and compatible across a wide range of mobile devices to maximize accessibility and ensure broad user adoption.

3.3 Application Layer Software Dependencies

The Application Layer of the Smart Crib system relies on several software dependencies to enable its functionality and ensure a smooth user experience. The primary framework used is React Native, which facilitates the development of a cross-platform mobile application compatible with both iOS and Android devices. Within React Native, additional libraries and packages are utilized to enhance the application's capabilities.

To manage navigation between screens and components, the application uses React Navigation, which provides a seamless and intuitive experience for users by enabling tab navigation. This allows users to switch between different device controls like lights, door locks, and fans effortlessly. For network communication, Axios is used to handle HTTP requests between the mobile application and the backend server. This dependency allows the app to send user commands, such as adjusting brightness, and to receive updates on device statuses.

Expo is another crucial dependency, simplifying the development, testing, and deployment process across multiple platforms. It ensures compatibility with different devices, enabling quick feature testing

and deployment. To enhance the visual appeal of the application, React Native Vector Icons is utilized for adding icons that represent various controls and navigation elements, providing a user-friendly interface. Finally, Prop Types is used for type-checking props passed to components, ensuring data consistency and stability within the application.

These dependencies collectively support the functionality, user interface, and network communication of the Smart Crib mobile application, ensuring that it operates efficiently and seamlessly across various mobile platforms.

3.4 Light Control Subsystem

The Smart Crib application is the Smart Lights Control Subsystem, which is a combination of both hardware and software components designed to provide users with remote control over the lighting system within their smart home. This subsystem allows users to adjust the brightness of lights via the mobile application, ensuring convenience and energy efficiency.

On the software side, the Smart Lights Control Subsystem is managed through the mobile application interface, developed using React Native. The mobile app sends user inputs as HTTP POST requests to the backend server, which then relays the commands to the ESP32. The server is responsible for processing these requests, managing the state of the lights, and ensuring real-time feedback is communicated back to the user.

This subsystem provides seamless integration between the user interface, the server, and the smart light hardware, allowing for efficient and responsive control over the lighting in a smart home environment.

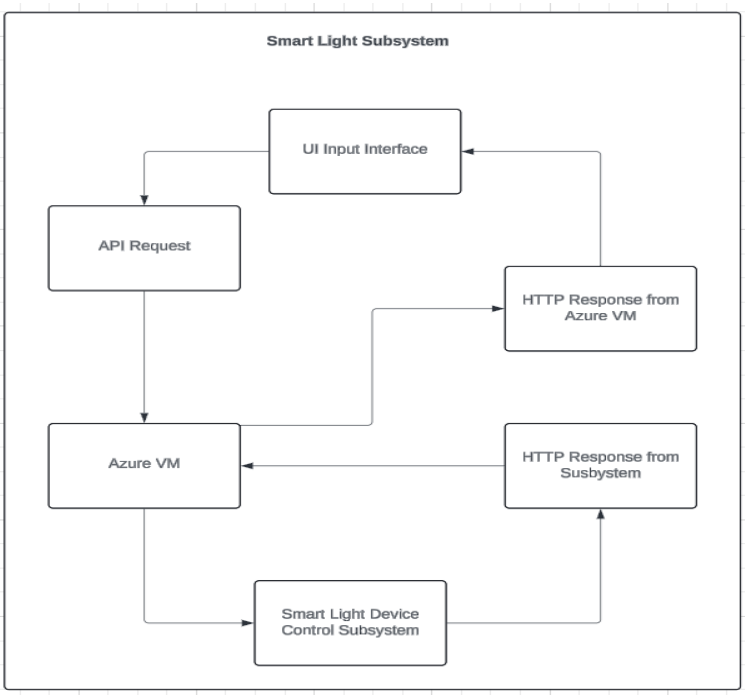

Figure 2: Smart Light Subsystem Architecture Diagram

3.4.1 Light Control Subsystem Hardware

The hardware component of this subsystem consists of smart light modules connected to an ESP32 microcontroller. The ESP32 is responsible for receiving commands from the server and adjusting the light's brightness levels accordingly. The microcontroller translates the digital signals sent by the application into physical actions, such as dimming or brightening the lights.

3.4.2 Light Control Subsystem Operating System

The Smart Lights Control Subsystem within the Smart Crib application relies on the ESP32 microcontroller, which runs on a real-time operating system (RTOS) called FreeRTOS. FreeRTOS is an open-source operating system specifically designed for embedded devices like the ESP32, providing low-latency and reliable task management crucial for IoT applications.

FreeRTOS allows the ESP32 to efficiently manage multiple tasks simultaneously, such as receiving commands from the server, adjusting the brightness of the lights, and sending status updates back to the server. The lightweight and efficient nature of FreeRTOS makes it an ideal choice for the resource-constrained environment of the ESP32, ensuring quick response times and seamless control over the lighting system.

3.4.3 Light Control Subsystem Software Dependencies

The Smart Lights Control Subsystem within the Smart Crib application relies on several software dependencies to manage the functionality of the ESP32 microcontroller and ensure seamless communication and control over the lighting system.

Key libraries used include the Adafruit NeoPixel library, which is essential for controlling LED strips and individual lights connected to the ESP32. This library provides functions for adjusting brightness levels, setting colors, and creating lighting effects. The WiFi library is also utilized, enabling the ESP32 to connect to the network and communicate with the Smart Crib server. For handling HTTP requests and responses, the HTTPClient library is implemented, allowing the ESP32 to receive commands from the server and send back status updates effectively.

Additionally, ArduinoJson is used to parse JSON data sent between the server and the ESP32, ensuring accurate interpretation and processing of commands related to light adjustments. These software libraries are integrated within the ESP32's development environment, programmed using the Arduino IDE, which provides a comprehensive platform for coding, testing, and debugging the firmware controlling the smart lights. These dependencies are essential for the subsystem to function efficiently, ensuring that the lights respond promptly to user commands via the mobile interface.

3.4.4 Light Control Subsystem Programming Languages

The Smart Lights Control Subsystem in the Smart Crib application is primarily programmed using C/C++, which is the standard for developing firmware on microcontroller platforms like the ESP32. This language choice is due to its efficiency and low-level hardware access, making it ideal for controlling components such as LEDs and managing network communication directly through the microcontroller's interfaces.

C/C++ is utilized to integrate essential libraries such as Adafruit NeoPixel for LED control and WiFi for network connectivity, ensuring the ESP32 can interact with the Smart Crib server. The code is developed and compiled using the Arduino IDE, which supports C/C++ and provides a user-friendly environment for building and deploying firmware onto the microcontroller. This combination of languages and de-

velopment tools allows precise control over the lighting hardware, enabling real-time response to user inputs from the Smart Crib mobile app.

3.4.5 Light Control Subsystem Data Structures

In the Smart Lights Control Subsystem of the Smart Crib application, data is transmitted from the ESP32 microcontroller to the server using JSON packets, which provide a lightweight and structured format for communication. Each packet contains an identifier (id) for the microcontroller, a device field indicating the type (where 1 represents the light component), and a brightness value ranging from 0 (off) to 100 (maximum brightness) that reflects the light's current setting. Additionally, a timestamp is included to log when the packet is sent, aiding in tracking and debugging. This structured approach ensures consistent and efficient communication between the light device and the server, enabling accurate control and monitoring within the Smart Crib system.

3.4.6 Light Control Subsystem Data Processing

The Light Control Subsystem in the Smart Crib application employs a straightforward data processing algorithm to adjust light brightness levels based on user inputs received through the mobile application. The algorithm first validates the brightness value, ensuring it falls within the acceptable range of 0 to 100. Once validated, the system sends the brightness value, along with the device identifier, to the ESP32 microcontroller via a JSON packet. The microcontroller processes this data and adjusts the PWM (Pulse Width Modulation) signal accordingly, altering the light intensity in real-time. The system uses a PID (Proportional-Integral-Derivative) control mechanism to smooth out transitions and prevent abrupt changes in brightness, ensuring a comfortable user experience. This algorithm not only allows precise control of lighting but also ensures energy efficiency by adjusting the brightness based on user preferences and predefined settings.

3.5 Door lock Control Subsystem

The Door lock Control Subsystem utilizes an ESP32 microcontroller connected to an electronic lock mechanism. The electronic lock operates using a solenoid or a motorized actuator to engage or disengage the lock based on the user's input. This hardware setup is compact and designed to fit within standard door lock assemblies, allowing for easy installation and integration with existing home setups.

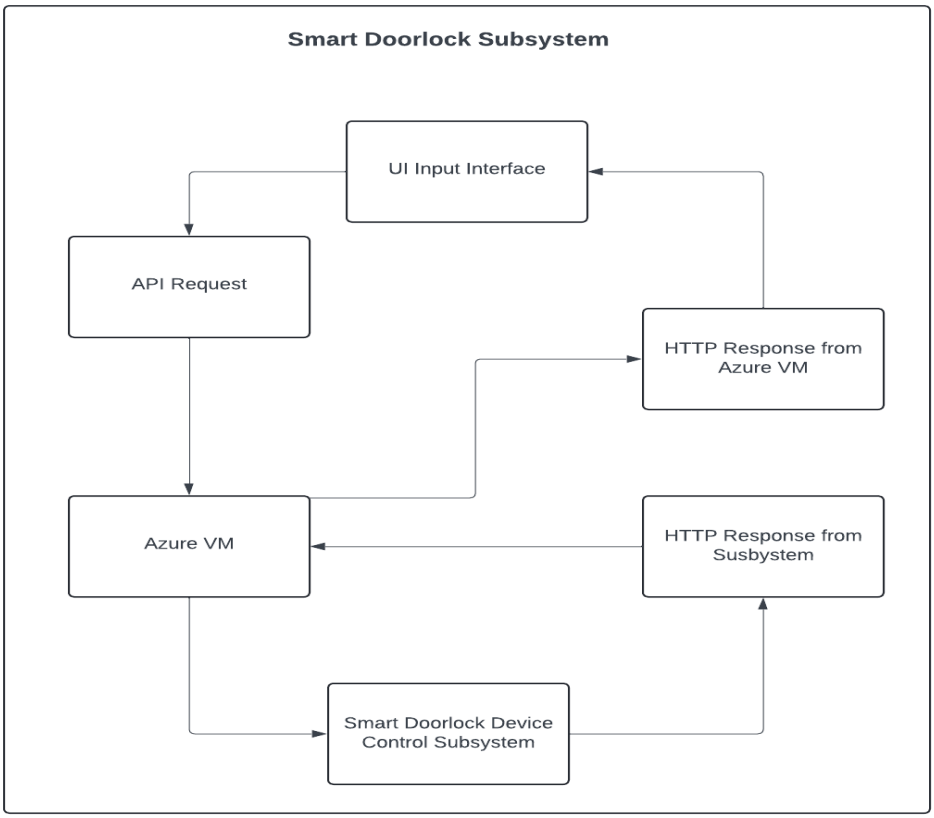

Figure 3: Smart Doorlock Subsystem Architecture Diagram

3.5.1 Door Lock Subsystem Hardware

The Door lock Control Subsystem utilizes an ESP32 microcontroller connected to an electronic lock mechanism. The electronic lock operates using a solenoid or a motorized actuator to engage or disengage the lock based on the user's input. This hardware setup is compact and designed to fit within standard door lock assemblies, allowing for easy installation and integration with existing home setups.

3.5.2 Door Lock Subsystem Operating System

The subsystem does not have a separate operating system as it relies on the real-time capabilities of the ESP32 microcontroller. The microcontroller runs its firmware, developed using the Arduino framework, which facilitates direct hardware control and communication with the server.

3.5.3 Door Lock Subsystem Software Dependencies

The Door lock Control Subsystem software depends on several libraries, including the WiFi.h library for establishing a wireless connection with the home network, HTTPClient.h for communicating with the Smart Crib server, and ArduinoJson.h for parsing and creating JSON packets to manage data exchange efficiently. Additionally, the Servo.h library is used for controlling the motorized actuator that locks or unlocks the door.

3.5.4 Door lock Subsystem Programming Languages

The subsystem is programmed using C/C++, specifically leveraging the Arduino framework. This choice allows for real-time, low-level hardware interaction essential for the control of the electronic lock com-

ponents and communication modules of the ESP32 microcontroller.

3.5.5 Door lock Subsystem Data Structures

The Door lock Control Subsystem uses a simple JSON data structure for communication between the server and the microcontroller. The structure includes fields for id, device, and door lock status (0 for unlocked and 1 for locked). This packet is transmitted from the server to the ESP32, which interprets the command and adjusts the lock mechanism accordingly.

3.5.6 Door lock Subsystem Data Processing

The data processing algorithm for the Door lock Control Subsystem checks incoming JSON packets for the door lock field value, which determines whether the door should be locked or unlocked. The ESP32 microcontroller receives this information and activates the actuator accordingly. To prevent unauthorized access or incorrect operations, the system validates the command using pre-configured security protocols, ensuring that the door lock status can only be modified by authenticated user requests. If the lock state changes, the system sends a status update back to the server, maintaining synchronization with the mobile application.

3.6 Smartfan Control Subsystem

The Smart Fan Subsystem is designed as an IoT-controlled piece of hardware that manages the speed and operation of a connected fan. It operates autonomously or through user commands received from the Smart Crib application. At its core, the subsystem uses an ESP32 microcontroller, which serves as the central processing unit for controlling fan speed and monitoring environmental conditions like temperature and humidity through sensors.

The subsystem communicates with the Smart Crib server via Wi-Fi, sending sensor data and receiving user commands to adjust the fan speed using Pulse Width Modulation (PWM) techniques. The microcontroller runs firmware built with the Arduino framework, ensuring real-time responsiveness and seamless integration with the overall system. This setup allows the Smart Fan Subsystem to act as an intelligent component within the Smart Crib ecosystem, providing efficient and customizable air circulation based on environmental feedback or user preferences.

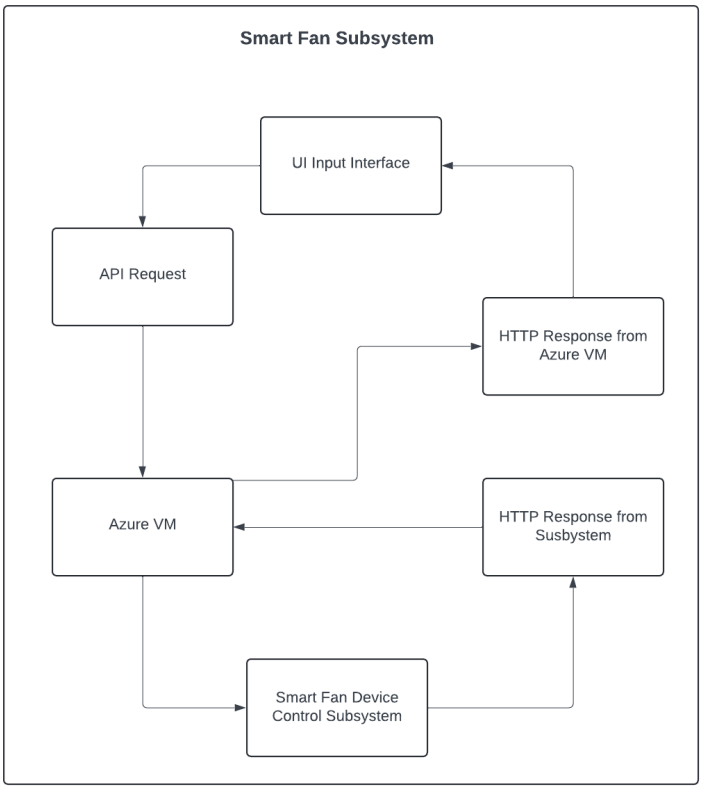

Figure 4: Smart Fan Device Subsystem Diagram

3.6.1 Smart Fan Subsystem Hardware

The Smart Fan Subsystem uses an ESP32 microcontroller connected to a variable-speed fan via a relay module or a Pulse Width Modulation (PWM) motor driver. The ESP32 adjusts the fan speed based on user input or environmental conditions. Sensors, such as DHT11 or DHT22, are connected to monitor temperature and humidity levels, providing the system with real-time environmental data.

3.6.2 Smart Fan Subsystem Operating System

The Smart Fan Subsystem runs on the ESP32 microcontroller without a dedicated operating system. It operates in a real-time environment using firmware developed with the Arduino framework, allowing direct control over the fan speed and sensors.

3.6.3 Smart Fan Subsystem Software Dependencies

The subsystem's software includes several libraries: WiFi.h for managing network connectivity, HTTPClient.h for server communication, and ArduinoJson.h for handling JSON data exchanged between the ESP32 and the Smart Crib server. Additionally, DHT.h is used to interface with temperature and humidity sensors, while PWM.h enables PWM signal control for managing fan speed.

3.6.4 Smart Fan Subsystem Programming Languages

The Smart Fan Subsystem is programmed using C/C++, leveraging the Arduino framework for precise control over the fan and sensors, as well as efficient communication with the Smart Crib server.

3.6.5 Smart Fan Subsystem Data Structures

The subsystem uses JSON packets for data transmission. These packets include fields such as id, device, fanSpeed, temperature, and humidity. The ESP32 parses these JSON packets to adjust the fan speed or send sensor readings to the server, ensuring seamless communication and synchronization with the overall system.

3.6.6 Smart Fan Subsystem Data Processing

The Smart Fan Subsystem employs a control algorithm that adjusts fan speed based on user input or sensor readings. When operating manually, the ESP32 processes the JSON packet to set the fan speed using PWM signals. In automatic mode, the fan speed is dynamically adjusted using a proportional control algorithm based on temperature and humidity readings. The subsystem ensures that changes in fan speed are reported back to the server to maintain synchronization with the mobile application.

4 Server Layer Subsystems

The Cloud Server Layer encompasses two primary subsystems: the virtual machine (VM) service and the server software. The VM service facilitates the provisioning of essential computing resources, an operational environment, and internet connectivity, enabling seamless data transmission through specified network ports. This connectivity is critical as the entire system depends on robust internet communication protocols, which are effectively managed and orchestrated by the server software.

4.1 Server Layer Hardware

In the Smart Crib system architecture, the server layer is hosted on an Azure Virtual Machine (VM), which provides the necessary infrastructure for handling user commands and coordinating communication between the client and IoT devices. The Azure VM is configured with Ubuntu OS, offering a stable and secure environment for hosting the Python-based HTTP server. This setup leverages the flexibility and scalability of cloud infrastructure, allowing for efficient scaling and management as the system grows.

4.2 Server Layer Operating System

The Cloud Server Layer operates on an Ubuntu-based Virtual Machine (VM), offering a stable and secure environment for running our application code. Ubuntu, a widely-used Linux distribution, provides the necessary libraries, package management, and flexibility needed for efficient software deployment. Its lightweight and efficient nature ensures minimal resource consumption while maintaining the reliability required for continuous server operations. This operating system supports the installation of essential components such as network management tools, security patches, and software dependencies, ensuring that the Smart Crib system remains responsive and secure throughout its operation.

4.3 Server Layer Software Dependencies

The Cloud Server Layer relies on several software dependencies to ensure smooth operation and communication between components. It utilizes Python 3 as the primary programming language, offering extensive libraries and modules to support efficient server management. The server is built on an HTTP server framework, which enables it to handle incoming and outgoing HTTP requests effectively. This setup is essential for processing commands and transmitting data between the client application and IoT devices. Additionally, various Python libraries are used to manage network connections, handle JSON data, and maintain server uptime and reliability, ensuring that the Smart Crib system remains robust and responsive.

4.4 Request Handler Subsystem

The Request Handling Subsystem for the Cloud Server Layer ensures that the server remains responsive and capable of handling multiple incoming HTTP requests simultaneously. It operates by continuously listening on a designated port. When an HTTP request is received, the subsystem spawns a new thread dedicated to processing that request. This threading approach keeps the port open and available for additional requests, preventing any blocking or delays in handling new connections. Each HTTP request typically contains a JSON file with essential information, such as the origin of the request (whether it's from the application or a connected device) and updates related to device states or relevant data. This setup allows the server to efficiently manage and process incoming data, maintaining seamless communication between the cloud infrastructure and the connected devices or application.

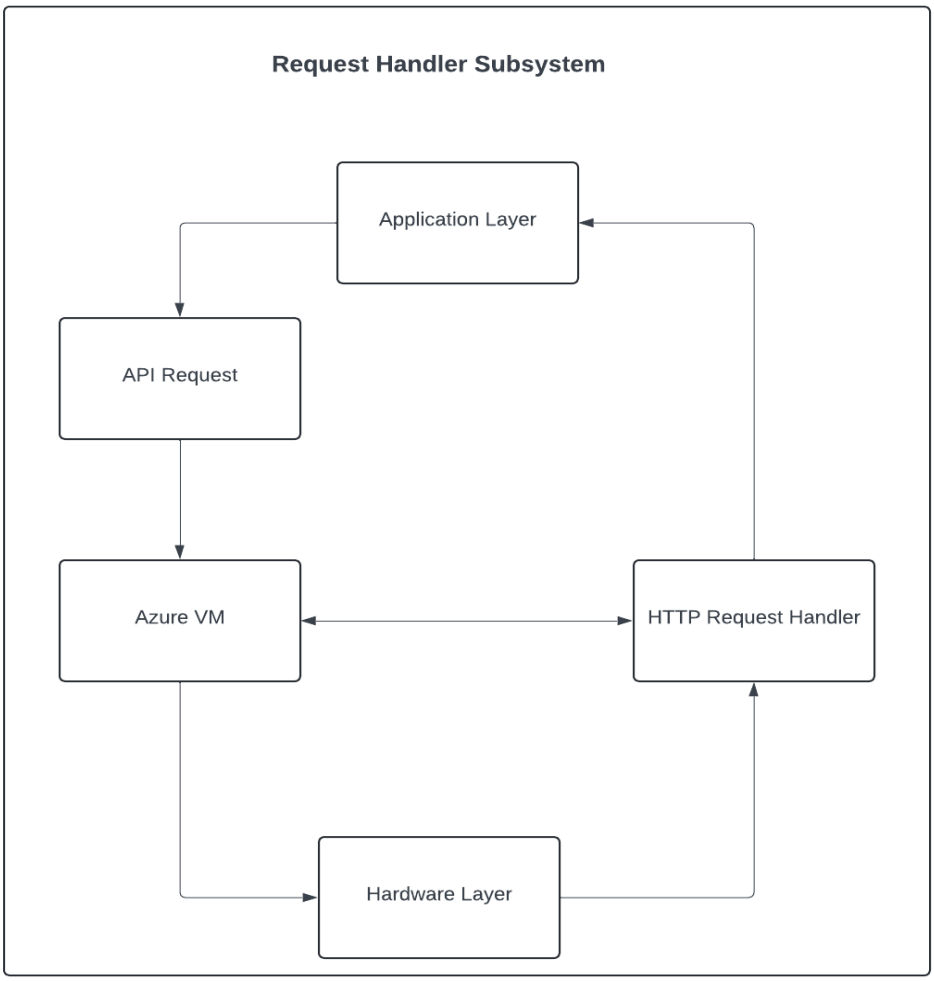

Figure 5: Request Handler Subsystem Diagram

4.4.1 Request Handler Subsystem Hardware

The Request Handling Subsystem runs entirely on a virtual machine (VM) hosted on Microsoft Azure. As such, the hardware components are virtualized and managed by Azure's infrastructure, providing computational resources, storage, and network connectivity necessary to handle HTTP requests and manage threads efficiently. The VM is configured to maintain high availability and scalability, ensuring it can handle increasing loads as the number of connected devices and requests grows.

4.4.2 Request Handler Subsystem Operating System

The operating system used for the Request Handling Subsystem is Ubuntu. Ubuntu, a widely-used Linux distribution, provides a stable and secure environment for running the server software. It is chosen for its compatibility with Python and its extensive support for networking services, which are essential for managing incoming HTTP requests and threading operations.

4.4.3 Request Handler Subsystem Software Dependencies

The software dependencies for the Request Handling Subsystem include Python3, which serves as the primary programming language for implementing the server and handling HTTP requests. The sub-

system utilizes Python's built-in `http.server` library to create an HTTP server capable of listening on the designated port and managing incoming requests. To handle multiple requests efficiently, the Python threading library is employed, allowing the server to spawn new threads for each request, thus ensuring non-blocking communication and processing. Additionally, Python's JSON library is used for parsing incoming JSON files that contain information such as device updates and request origins, and for structuring the JSON responses sent back to the client or devices. These dependencies are crucial for maintaining the system's functionality, efficiency, and reliability in managing real-time data processing and communication between the server, application, and connected devices.

4.4.4 Request Handler Subsystem Programming Languages

The Request Handling Subsystem is primarily implemented using Python. Python is chosen for its simplicity, flexibility, and robust support for web and network-related libraries, which are essential for handling HTTP requests and managing threading effectively.

4.4.5 Request Handler Subsystem Data Structures

The primary data structure used in the Request Handling Subsystem involves JSON objects. When a request is received, the server parses the incoming JSON data to extract information such as the source of the request (whether it is from the application or a connected device), the target device, and the relevant updates (e.g., brightness level or lock status). The JSON format is used for its lightweight nature and ease of serialization, allowing efficient transmission of structured data between the client, server, and IoT devices.

4.4.6 Request Handler Subsystem Data Processing

The Request Handling Subsystem uses a multi-threading algorithm to manage incoming HTTP requests. When a request is received, a new thread is spawned, allowing the server to process the request independently while keeping the port open for other incoming connections. This prevents bottlenecks and ensures that multiple devices or applications can interact with the server simultaneously without delay. Additionally, the subsystem processes JSON data by extracting key information (such as device type and state updates) and updating the server's state accordingly. This method ensures the system remains responsive, and the updates are processed in real-time, providing users with seamless control over their connected devices.

5 Device Layer Subsystems

Our device layer subsystem consists of three main components: the microcontroller, the hardware components, and the supporting software. The microcontroller plays a critical role in enabling independent device functionality, operating autonomously from other systems. It interfaces with the hardware components, which allow us to interact with the physical environment. The software leverages these components to execute the specific tasks required by the system.

5.1 Device Layer Hardware

The Device Layer hardware comprises two primary microcontrollers: the ESP32 and the Raspberry Pi Pico. The ESP32 is responsible for controlling the smart light and door lock devices, leveraging its built-in Wi-Fi capabilities for seamless communication with the cloud server. The Raspberry Pi Pico is used for the smart fan subsystem, chosen for its efficient processing and GPIO capabilities. These microcontrollers interface with various sensors and actuators to manage device states and respond to server commands. Each microcontroller is equipped with hardware components like relays, sensors, and motors that enable physical interaction with the environment.

5.2 Device Layer Operating System

The Device Layer does not use a traditional operating system, as both the ESP32 and Raspberry Pi Pico operate in a bare-metal configuration. This lightweight setup is ideal for IoT applications, enabling direct control over hardware resources without the overhead of an operating system.

5.3 Device Layer Software Dependencies

The Device Layer software relies on several libraries tailored to the microcontrollers. For the ESP32, the Arduino framework provides a development environment with libraries for managing Wi-Fi communication and interfacing with sensors and actuators. The Raspberry Pi Pico uses the MicroPython environment, allowing for easy coding and deployment on the device. Key libraries include networking libraries for communication protocols and sensor libraries for reading and transmitting device status.

5.4 Smart Light Subsystem

The Smart Light Subsystem is designed to provide dynamic lighting control within the Smart Crib system, integrating hardware components such as an LED light module and a motion sensor with an ESP32 microcontroller. The subsystem's primary function is to adjust the brightness and color of the LED light based on user input received from the server, as well as to detect motion and trigger light changes independently. The ESP32 microcontroller acts as the central controller, communicating with the server to receive updates, and directly interfacing with the LED module and motion sensor for real-time adjustments.

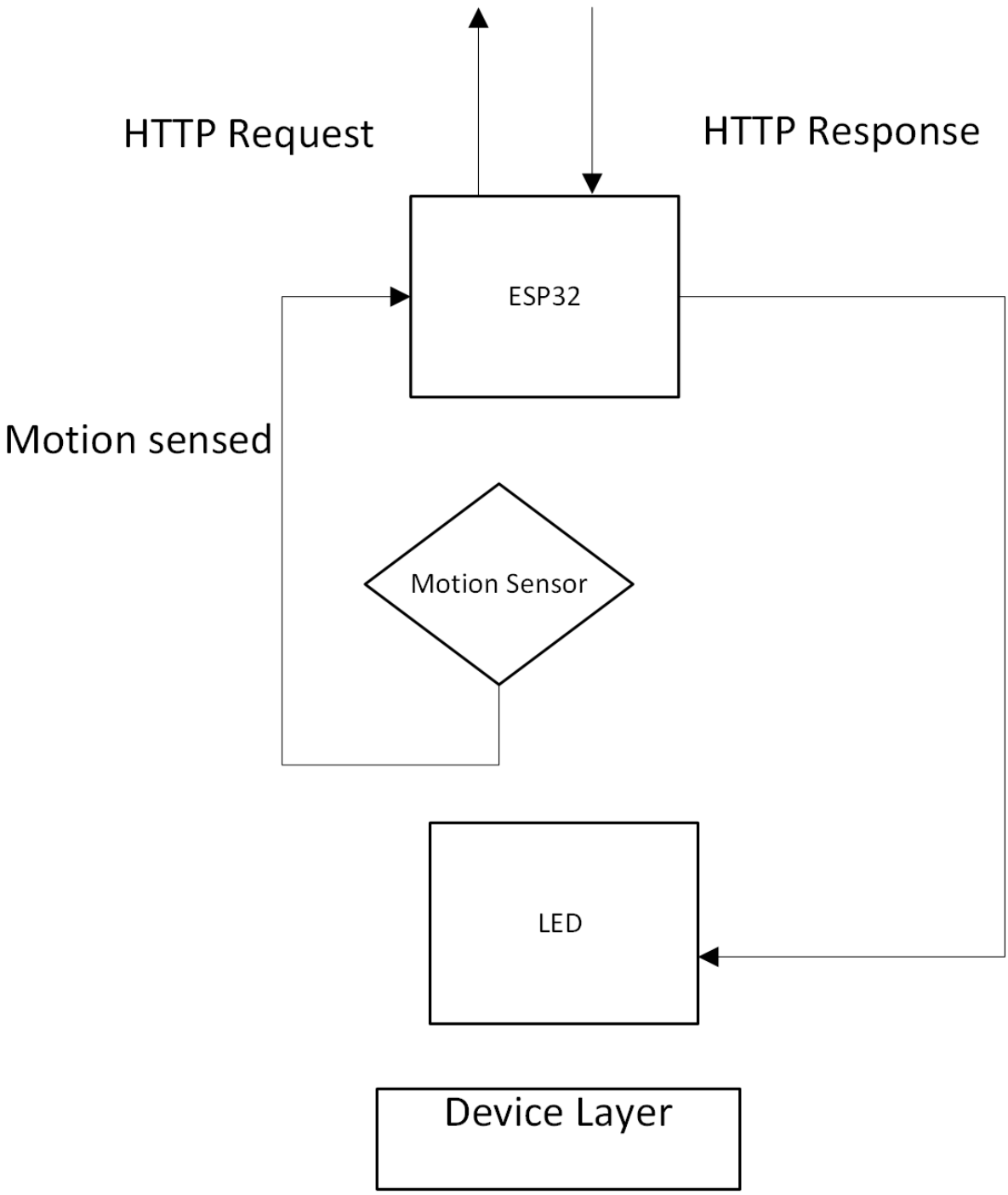

Figure 6: Smart Light Device Subsystem Diagram

5.4.1 Subsystem Hardware

The hardware for the Smart Light Subsystem includes an ESP32 microcontroller powered by a 5V source, an LED light module capable of adjustable brightness and color, and a motion sensor module for detecting movement. The ESP32 manages the communication and functionality of these components, using its GPIO pins to control the LED module via a signal wire and receive inputs from the motion sensor. The

LED module is equipped with integrated circuitry to simplify brightness and color adjustments based on signals sent by the ESP32, while the motion sensor provides detection signals that trigger actions.

5.4.2 Subsystem Operating System

The Smart Light Subsystem does not rely on a traditional operating system. Instead, it operates on a bare-metal configuration using the ESP32 microcontroller, which executes instructions directly without the overhead of a complex OS. This lightweight setup ensures minimal latency and quick response times for controlling the lighting system and processing motion sensor data.

5.4.3 Subsystem Software Dependencies

The software dependencies for the Smart Light Subsystem include the Arduino IDE, which provides a development environment for programming the ESP32 microcontroller. Additionally, several libraries are used: the Adafruit LED Light module library for controlling the LED brightness and color, the ESP32 board package for Arduino IDE compatibility, and the Arduino JSON library for parsing and structuring JSON data. The HTTP client library is also utilized to manage server communications, enabling the ESP32 to send and receive JSON-formatted data through HTTP requests.

5.4.4 Subsystem Programming Languages

The Smart Light Subsystem is programmed primarily using the C language through the Arduino IDE. This programming language allows for low-level access to the microcontroller's hardware features, providing precise control over the LED light module and motion sensor.

5.4.5 Subsystem Data Structures

The Smart Light Subsystem uses JSON objects as the primary data structure for communication with the server. Each request sent to the server contains a JSON object with information such as device identification, current hardware state, and user-specified updates (e.g., brightness levels). The JSON format allows for efficient data serialization and easy parsing by both the server and the microcontroller, ensuring effective communication and control over the smart lighting system.

5.4.6 Subsystem Data Processing

The Smart Light Subsystem processes data using two main strategies. First, it sends HTTP requests to the server every second, each containing a JSON file identifying the device and its state. The server responds with updated lighting parameters, which the ESP32 then compares with the current state to determine if an adjustment is necessary. The subsystem also features independent processing: when motion is detected by the sensor, the microcontroller triggers the LED light for a preset duration. This local control does not depend on server interaction, ensuring that basic lighting functions remain operational even if the network connection is disrupted.

5.5 Smart Lock Device Subsystem

The Smart Lock Device Subsystem is a hardware-based solution designed to manage the physical locking and unlocking of the Smart Crib door. At its core, the subsystem utilizes an ESP32 microcontroller, which serves as the control unit for the locking mechanism. The microcontroller coordinates with several components: a linear actuator for the physical lock movement, relay switches to manage the actuator's activation, a 12V rechargeable battery for power, and 5V step-down modules to regulate power to smaller components. Additionally, a keypad and an unlock button provide user input options directly on the device. The software is programmed in C using the Arduino IDE, leveraging its compatibility with various libraries, such as those for the keypad module, which facilitates easier development and integration. The subsystem employs multithreading to handle multiple tasks simultaneously, including processing server updates that provide app-based control and monitoring keypad inputs for user PIN

verification. The microcontroller sends HTTP requests to the server to sync the lock's state, ensuring real-time updates based on user actions through the mobile app. If the correct PIN is entered via the keypad, the device initiates an unlock sequence. Key software dependencies include the ESP32 board support package, the Arduino JSON library for parsing data, the HTTP client for server communication, and the keypad library. This architecture allows for seamless integration of hardware and software components to provide both app-based and physical control of the door lock.

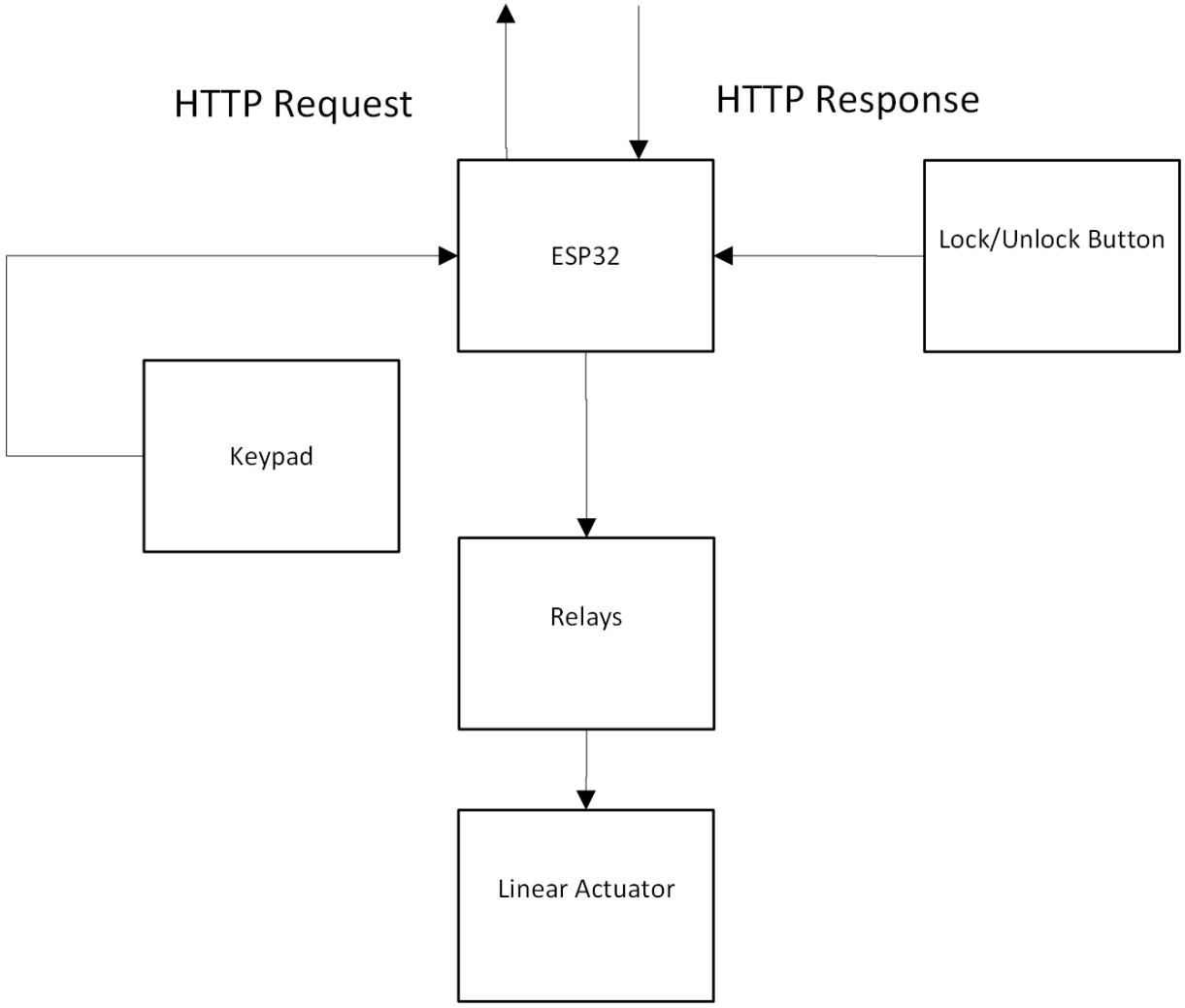

Figure 7: Smart Lock Device Subsystem Diagram

5.5.1 Smart Lock Subsystem Hardware

The hardware includes an ESP32 microcontroller, a linear actuator for the locking mechanism, relay switches for controlling the actuator, a 12V rechargeable battery, 5V step-down modules for power regulation, a keypad for external input, and an unlock button for internal control.

5.5.2 Smart Lock Subsystem Operating System

The ESP32 microcontroller operates on its integrated firmware, compatible with the Arduino platform.

5.5.3 Smart Lock Subsystem Software Dependencies

Dependencies include the ESP32 board package, the Arduino JSON library for data processing, an HTTP client for server communication, and the keypad library for reading user inputs.

5.5.4 Smart Lock Subsystem Programming Languages

The programming language used for the subsystem is C, leveraging the Arduino IDE for development.

5.5.5 Smart Lock Subsystem Data Structures

The data transmitted between the ESP32 and the server is structured as JSON packets containing device IDs, state updates, and command confirmations. The packets are parsed using the Arduino JSON library.

5.5.6 Smart Lock Subsystem Data Processing

The subsystem employs algorithms for multithreading to manage simultaneous tasks like monitoring keypad inputs and processing server updates. It also implements input validation algorithms for secure PIN verification and state synchronization routines to align physical lock states with server commands.

5.6 Smart Fan Subsystem

The Smart Fan Subsystem is a hardware-based solution designed to control the operation and speed of a fan within the Smart Crib environment. At its core, the subsystem utilizes a Raspberry Pi Pico microcontroller, which serves as the control unit for the fan's operation. The microcontroller coordinates with several components: a motor driver to control the fan motor speed, temperature sensors to monitor the environment, and a relay switch to manage the fan's power state. The fan's speed is adjusted based on temperature readings, ensuring that the environment is maintained at optimal comfort levels. The software is programmed in C using the Arduino IDE, taking advantage of its compatibility with various libraries, such as those for temperature sensing and motor control, which facilitate efficient development and integration. The subsystem employs multithreading to handle concurrent tasks, such as reading temperature values, adjusting fan speed, and communicating with the server for remote control. The microcontroller sends periodic updates to the server, reporting the fan's current state and temperature readings, and receives commands from the app for manual speed adjustments. Key software dependencies include the Raspberry Pi Pico board package, temperature sensor libraries for reading environmental data, and motor control libraries for managing fan speed. This architecture allows for an integrated approach, combining sensor data and remote commands to provide both automatic and manual control of the fan.

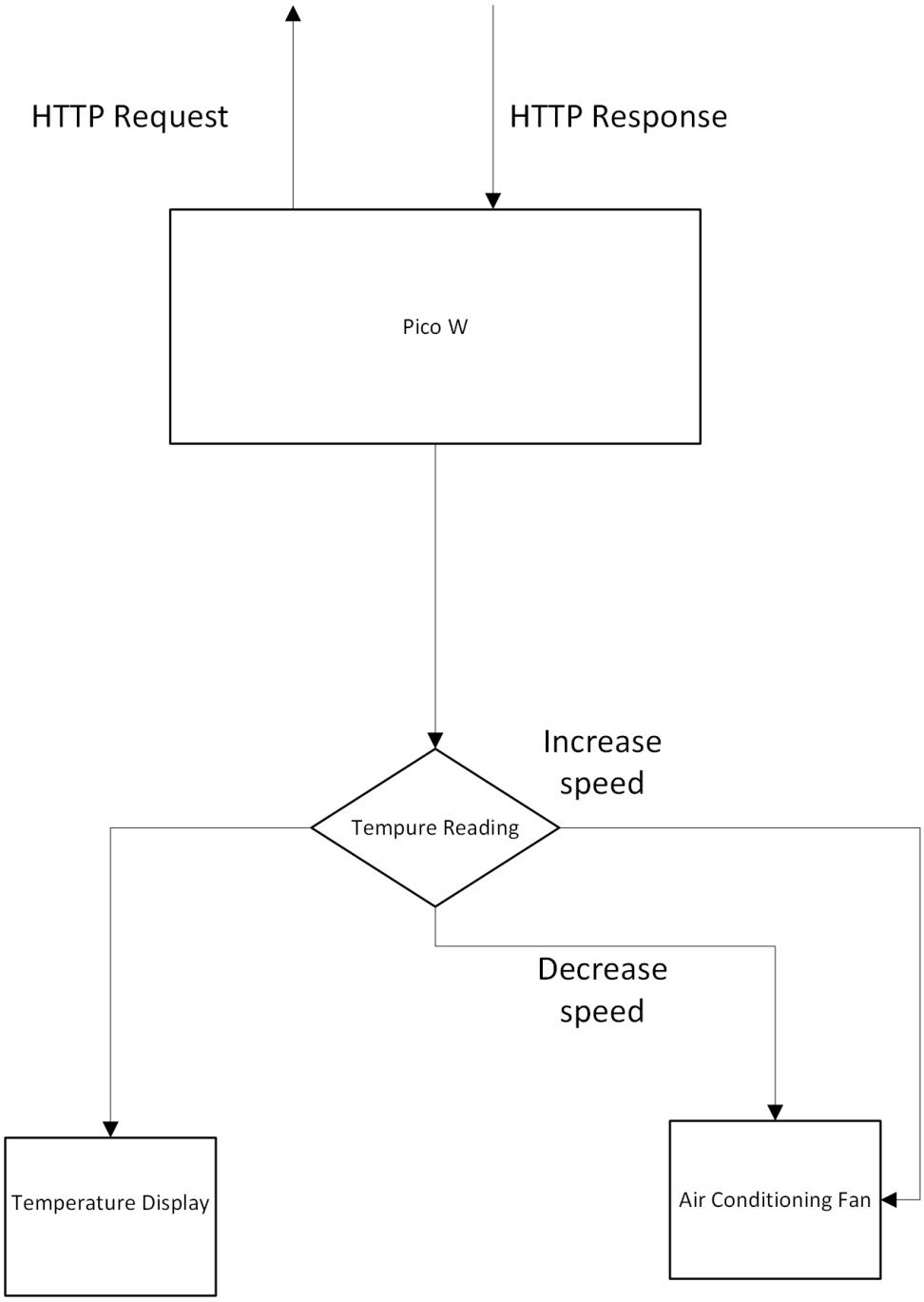

Figure 8: Smart Fan Device Subsystem Diagram

5.6.1 Smart Fan Subsystem Hardware

The hardware includes a Raspberry Pi Pico microcontroller, a motor driver for controlling fan speed, temperature sensors for monitoring environmental conditions, and a relay switch for managing the fan's power state.

5.6.2 Smart Fan Subsystem Operating System

The Raspberry Pi Pico operates using its integrated firmware, compatible with the Arduino platform for development.

5.6.3 Smart Fan Subsystem Software Dependencies

Dependencies include the Raspberry Pi Pico board package, libraries for temperature sensing, motor control libraries, and an HTTP client library for server communication.

5.6.4 Smart Fan Subsystem Programming Languages

The programming language used for the subsystem is C, leveraging the Arduino IDE for development.

5.6.5 Smart Fan Subsystem Data Structures

Data transmitted between the Raspberry Pi Pico and the server is structured as JSON packets containing temperature readings, fan speed, and device status. The JSON library is used for assembling and parsing these packets.

5.6.6 Smart Fan Subsystem Data Processing

The subsystem uses algorithms for multithreading to manage simultaneous tasks, such as monitoring temperature and controlling fan speed. It also implements a PID (Proportional-Integral-Derivative) control algorithm for precise fan speed adjustments based on temperature variations, ensuring efficient and responsive environmental management.

6 Appendix A

Appendix A: Smart Light Circuit Schematic

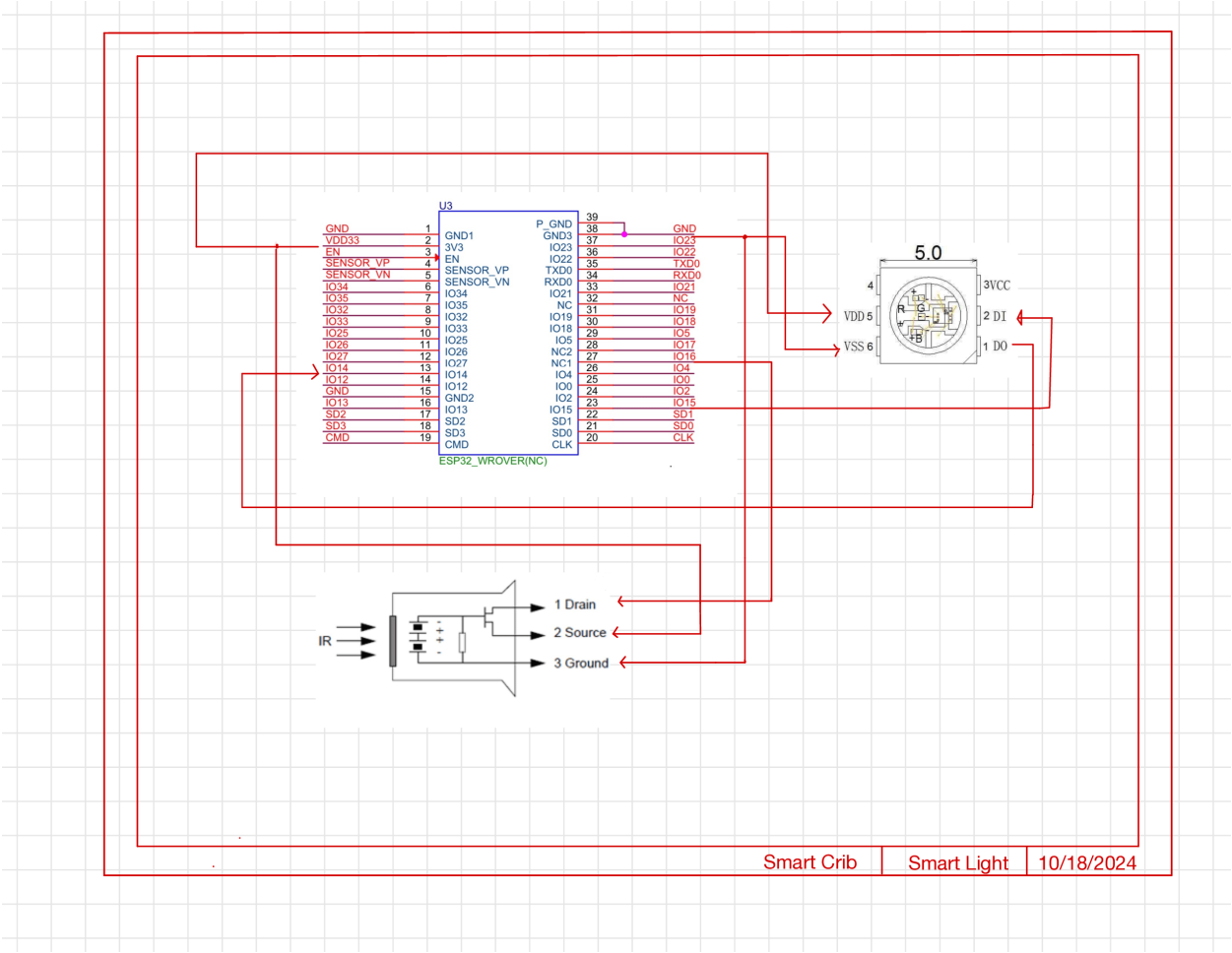

Figure 9: Circuit Schematic for the Smart Light Subsystem

APPENDIX B: SMART LOCK CIRCUIT SCHEMATIC

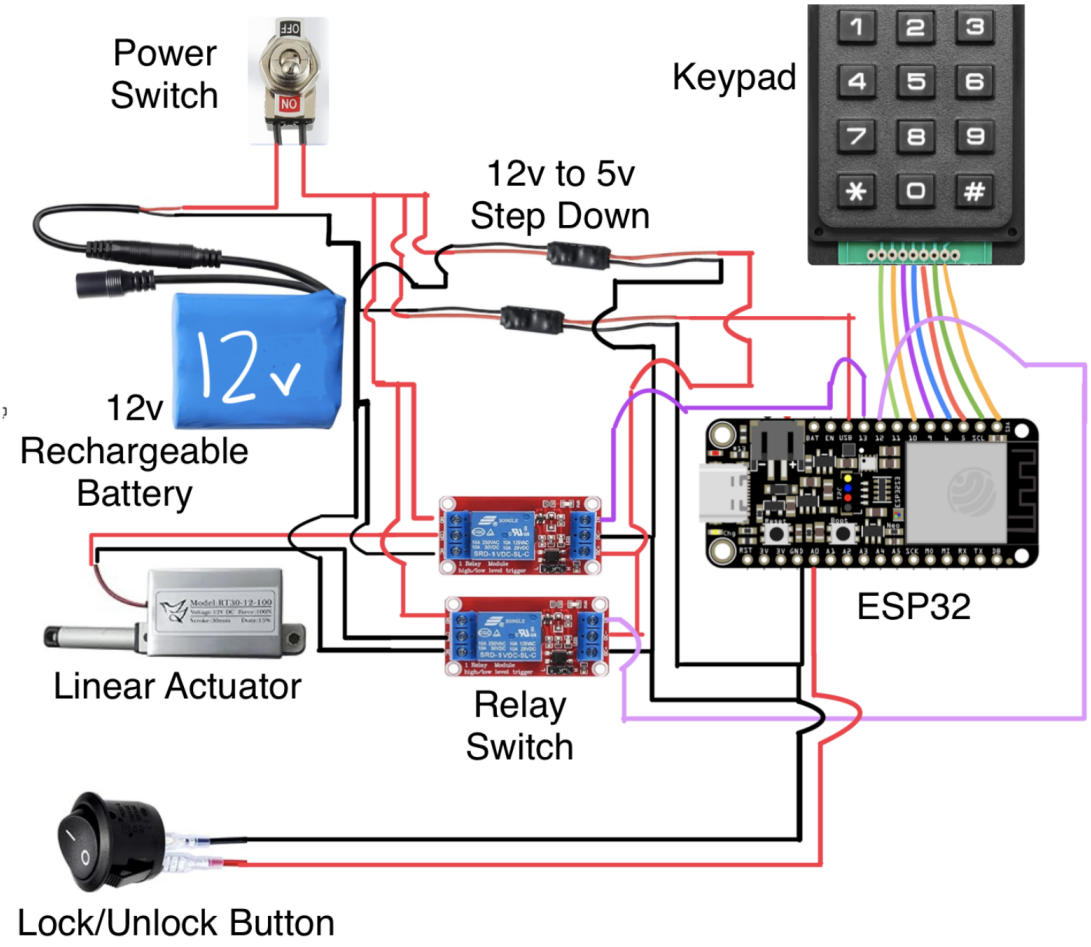

Figure 10: Circuit Schematic for the Smart Lock Subsystem

APPENDIX C: SMART FAN SUBSYSTEM CIRCUIT SCHEMATIC

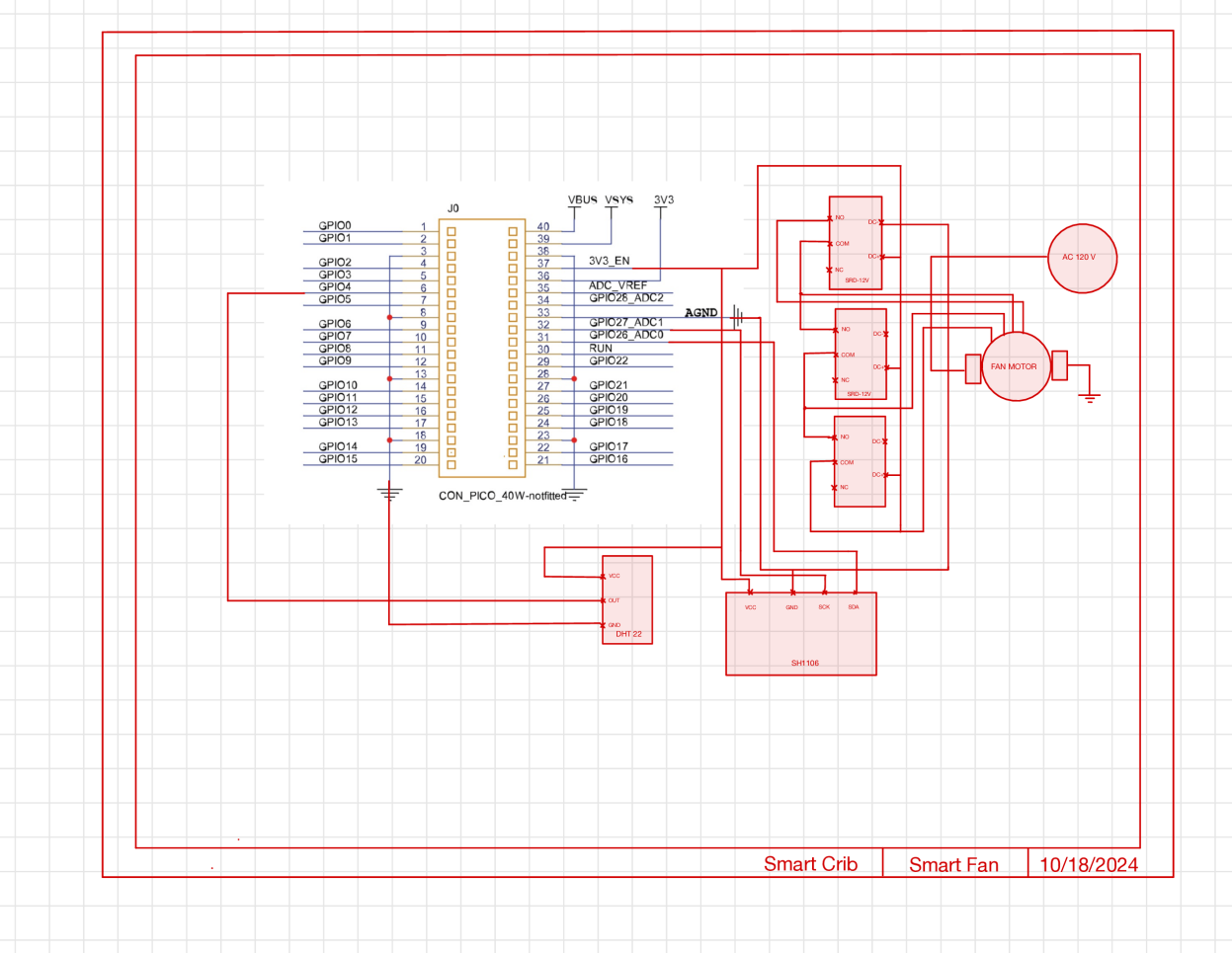

Figure 11: Circuit Schematic for the Smart Fan Subsystem

REFERENCES

[1] Arshdeep Bahga and Vijay Madisetti. *Internet of Things: A Hands-On Approach*. VPT, Atlanta, GA, 1st edition, 2014.

[2] Francesco De Pellegrini Daniele Miorandi, Sabrina Sicari and Imrich Chlamtac. *Internet of Things: Vision, Applications and Research Challenges*. Wiley, Hoboken, NJ, 1st edition, 2014.

[3] Samuel Greengard. *The Internet of Things*. MIT Press, Cambridge, MA, 1st edition, 2015.

[4] John Lee and Sanjay S. Kumar. *The Internet of Things in the Industrial Sector: Applications and Impacts*. Wiley, Hoboken, NJ, 1st edition, 2018.

[5] David L. Brock Sanjay E. Sarma and John R. Williams. *The Internet of Things: Roadmap to a Connected World*. Taylor & Francis, Boca Raton, FL, 1st edition, 2016.

[1] [3] [4] [2] [5]