Faculty of Science of the University of Porto
Department of Computer Science

Embedded Systems (CC4040)

2023/24

Embedded Systems Project

Work by:

Breno Marrão (202308171)
Daniel Franco (202006746)
Gonçalo Santos (202005006)
Sofia Avelino (202008485)

Professor: Sérgio Armindo Lopes Crisóstomo

Contents

1 Introduction

In an era characterized by rapid technological advancement, the use of technology for the purpose of security enhancement has become increasingly widespread. This is a reality across various contexts, including residential settings.

In this project, our focus lies in developing a system specialized in residential security: a doorbell camera. Through the use of motion detection technology and wireless data transmission to smartphones, this system aims to provide real-time monitoring, instant notifications, and convenient remote access to recorded footage in order to seamlessly elevate home-security.

1.1 Objectives

The goal of this project is to create a doorbell camera that could be installed at the door of a house. The camera system should be capable of:

- Activating upon doorbell press or motion detection.
- Recording images in "stop-motion" format, consisting of several photos instead of a continuous video to conserve storage space.
- Stopping the recording after 5 minutes of no motion detection.
- Notifying the user on their smartphone upon detection.
- Allowing users to view recorded "stop-motion" images and delete them or request a live video stream, via the smartphone app.
- Indicating that the camera is recording through the activation of an incorporated red light.

1.2 Requirements

Functional

- Presence Detection:
  - Detection of presence at the door must occur within 3 seconds.
- Recording Initiation:
  - The camera should start recording within 3 seconds of either motion detection or doorbell press.
- Video Stream Request:
  - Requesting a video stream from the smartphone app should have a latency of less than 10 seconds.
- Minimum Presence Detection Range:
  - Presence must be detected at a minimum distance of 50 cm from the camera.

Non-functional

- Reliability:

  - The system should ensure consistent detection and recording functionality.

- Usability:
  - The smartphone application should have an intuitive interface for users to access live streams, recorded images, and camera controls.

- Power Efficiency:
  - Power consumption should be optimized in order to prolong battery life and reduce energy costs.

2 Actor Model

The actor model for the Doorbell Camera project involves several interacting actors responsible for different tasks within the system. Here's an overview:

- **Sensor Actor (Arduino)**:
  - Receives input from the motion sensor and doorbell button.
  - Notifies the Central Actor (Raspberry Pi) upon detecting motion or doorbell press.
  - Receive information from the raspberry to turn the light on or off.

- **Camera Agent (Raspberry Pi)**:
  - Captures images or streams video from the camera.
  - Controls the recording process based on commands from the Central Actor.
  - Manages the storage of recorded data.

- **Database Agent (PostgreSQL on Raspberry Pi)**:
  - Manages the database operations, including storing and retrieving data.
  - Responds to requests from the Central Actor or User Agent for accessing stored information.

- **User Agent (Android App)**:
  - Communicates with the Central Actor to request live video streaming or access recorded data.
  - Displays notifications to the user regarding doorbell presses or motion detection events.
  - Allows users to interact with the system, such as viewing recorded footage or controlling camera settings.

In this actor model, each agent represents a distinct component of the system responsible for specific tasks. The Central Actor (Raspberry Pi) acts as a coordinator, receiving input from the Sensor Agent, issuing commands to the Camera Agent and Database Agent, and responding to requests from the User Agent. This modular approach facilitates system scalability, flexibility, and ease of maintenance.

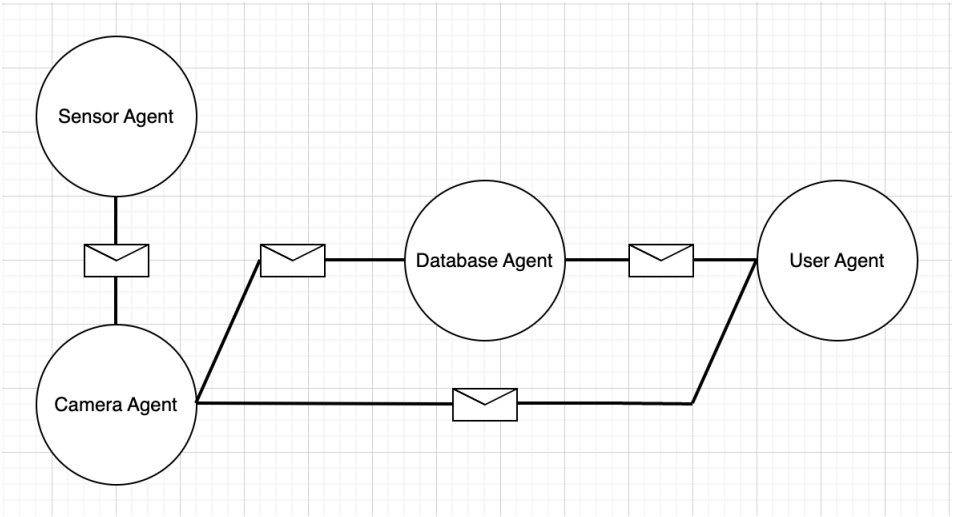

Figure 1: Actor Model

3 Hardware Prototype

The system used in this project is made up of 8 primary hardware components, which can be seen in figures 2-8. The Wi-Fi Shield (fig.3), the movement sensor (fig.4), the button (fig.5) and the the red LED (fig.6) are all connected to the Arduino(fig.2) through a breadboard.

In terms of physical connections, the **RPi** is connected to the camera through a cable.

In terms of wireless connections, the Arduino is connected to the **RPi** through Wi-Fi, as is the Android phone to the **RPi**.

Figure 2: **Arduino UNO R3**

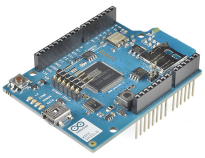

Figure 3: **Wi-Fi Shield**

Figure 4: **Movement Sensor Mini PIR**

Figure 5: **Tactile Button**

Figure 6: **Red LED**

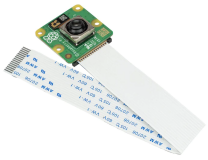

Figure 7: **Raspberry Pi Camera Module 2**

Figure 8: **Raspberry Pi**

Figure 9: **Android Phone**

4 Filled Requirement Form

Here is the requirement form

Name	Doorbell Camera
Purpose	To make a camera on a doorbell that can record and livestream image when someone is at the door.
Inputs	User request, movement, doorbell
Outputs	Footage of the camera on the doorbell
Performance	Camera start recording within 3 seconds of movement or doorbell , requesting stream from app should have latency < 10seconds
Manufacturing Cost	€111.65
Power	5.775W
Physical size/ weight	106grams

Figure 10: Filled Requirement Form

5 System architecture and specification

The system can be described in terms of it's logical components through block diagrams (fig.11 - hardware architecture diagram - & fig.12 - software architecture diagram).

5.1 Hardware architecture

The hardware architecture diagram illustrating this system contains 4 components:

1. **Arduino + Movement Sensor + Button + Wi-Fi Shield + Red LED**
2. **Raspberry Pi**
3. **Camera**
4. **Android**

The communication between the **Raspberry Pi** and the **Arduino** is bidirectional and is done through a Wi-Fi channel. Similarly, **Raspberry Pi** and the **Android** also communicate bi-directionally through a Wi-Fi channel, acting as a client-server interface.

The communication between the **Raspberry Pi** and the **Camera** is done through a cable, where the Raspberry Pi sends commands to the camera and receives information (videos) back.

5.2 Software Architecture

The software architecture of this project consists of independent modules running on different devices, ensuring a clear separation between software and hardware. The software components are designed to be interoperable of the implementation details of the physical devices.

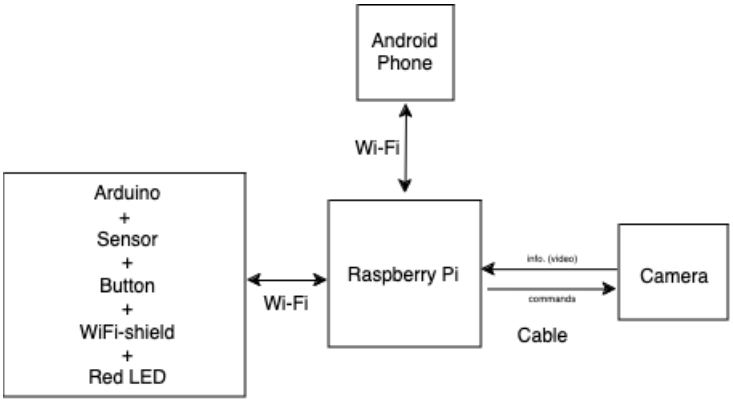

Figure 11: Hardware architecture diagram

5.2.1 Software Components

- **Sensor Agent (Arduino)**:
  - Responsible for detecting motion and button press events.
  - Triggers the Camera Agent upon motion detection or button press.
  - Communicates with the Raspberry Pi to convey sensor data , event notifications and to receive orders to turn the red light on or off.
- **Camera Agent (Raspberry Pi)**:
  - Controls the camera module to capture images in "stop-motion" format upon triggering by the Sensor Agent.
  - Manages image storage and stops recording after 5 minutes of inactivity.
  - Interfaces with the Database Agent to store image data and with the User Agent to provide live video streaming upon request.
- **Database Agent (PostgreSQL on Raspberry Pi)**:
  - Handles data storage and retrieval, particularly for recorded "stop-motions".
  - Ensures data integrity and security.
  - Facilitates communication between the Raspberry Pi and the User Agent for data access and management.
- **User Agent (Android App):**
  - Provides a user interface for interacting with the doorbell camera system.
  - Receives notifications on the smartphone upon doorbell press or motion detection.
  - Allows users to request live video streaming, view recorded "stop-motions," and delete stored images.
  - Communicates with the Raspberry Pi and the Database Agent to fetch data and control camera operations.

5.2.2 Integration and Communication

- The Sensor Agent (Arduino) communicates with the Raspberry Pi to trigger camera recording , send event notifications and to receive orders from the camera agent to turn the red led on or off.

- The Camera Agent (Raspberry Pi) interfaces with the Camera module to capture images and stores them in the PostgreSQL Database.
- The Database Agent (PostgreSQL) manages data storage and retrieval, ensuring seamless access to recorded images.
- The User Agent (Android App) interacts with the Raspberry Pi and the Database Agent to fetch live video streams and manage recorded images.

This hardware-independent software architecture allows for easy maintenance, and interoperability between components of the doorbell camera system.

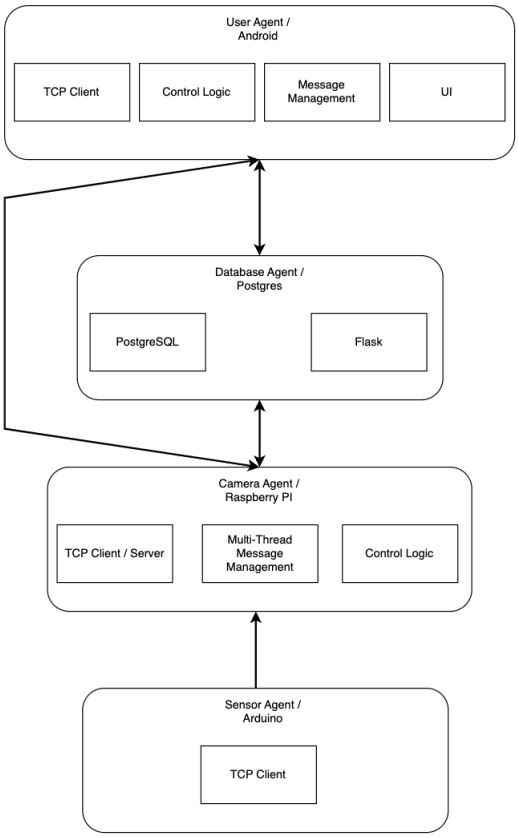

Figure 12: Software architecture diagram

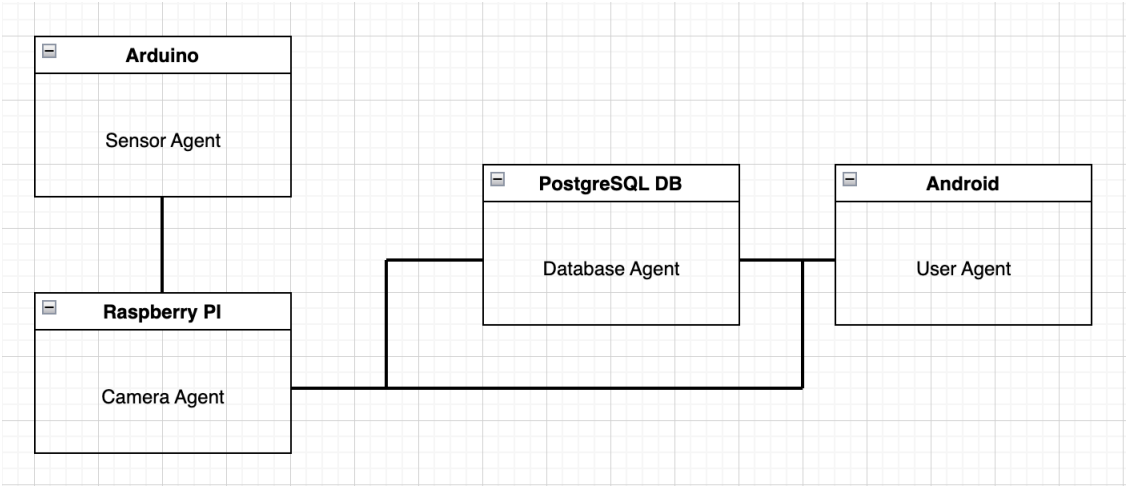

Figure 13: Deployment Schema

5.3 Finite State Machines

In this project, finite state machines were designed for all agents , here are the non-trivial ones.

- User Agent:

  In this system, there are three states:

  1. **Idle**: This state signifies that the system is inactive and not engaged in any specific task.
  2. **Stream**: This state occurs when the user signals that he wants to see the live footage from the camera.
  3. **dataBaseQuery**: This state occurs when the user signals that he wants to either get or delete footage from the database and leaves this state when there is a signal from the api with the answer to the request

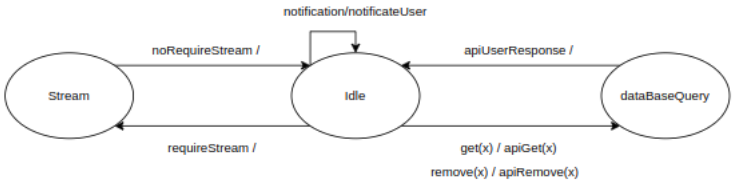

Figure 14: User Agent Finite State Machine

- Database Agent:

  In this finite-state machine, there are three states:

  1. **Idle**: This state signifies that the system is inactive and not engaged in any specific task.
  2. **getOrRemove**: This state occurs when the database agent receives a signal indicating to either get or remove something, and leaves this state when there is a response from the database.
  3. **put**: This state occurs when the database agent receives a signal that the camera agent wants to store footage, and leaves this state when there is a response from the database.

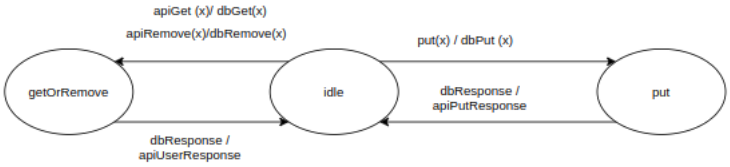

Figure 15: Database Agent Finite State Machine

- Sensorization actor :

  In this finite-state machine, there are two states:

  1. **Idle**: This state is engineered to intercept signals denoting movement detection or doorbell activation, promptly transmitting them to the camera control agent for further processing. Additionally, it is configured to await a "turnRedLight On" signal from the camera control agent.
  2. **redLightOn**: This state is activated upon receiving a signal through the camera control agent instructing the system to activate the red light, indicating an ongoing recording process. It remains in this state until a subsequent signal is received, prompting the system to deactivate the red light.

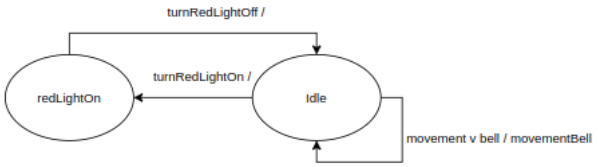

Figure 16: Sensorization actor Finite State Machine

- Camera Agent :

  In this finite-state machine, there are three states:

  1. **Idle**: This state signifies that the system is inactive and not engaged in any specific task.
  2. **Recording**: This state is triggered upon receiving a signal while in the idle state, indicating either movement detection or doorbell activation. Upon activation, the system turn the camera on, activates stop-motion mode, starts a countdown timer, and switches on the red light to indicate recording. Departure from this state occurs under two conditions: when the countdown timer reaches a value greater than or equal to 300, signifying that five minutes have elapsed since initialization, or when the user requests a stream.
  3. **Streaming**: This state indicates the system's active streaming of live camera footage to the user. Departure from this state occurs upon the user terminating the stream. If the count equals -1, indicating no ongoing recording, the system transitions to the idle state. Conversely, if recording is underway, the system shifts to the recording state to end the transmission.

  In the current implementation of this finite state machine , it is assumed that each increment in the count variable signifies the passage of one second.

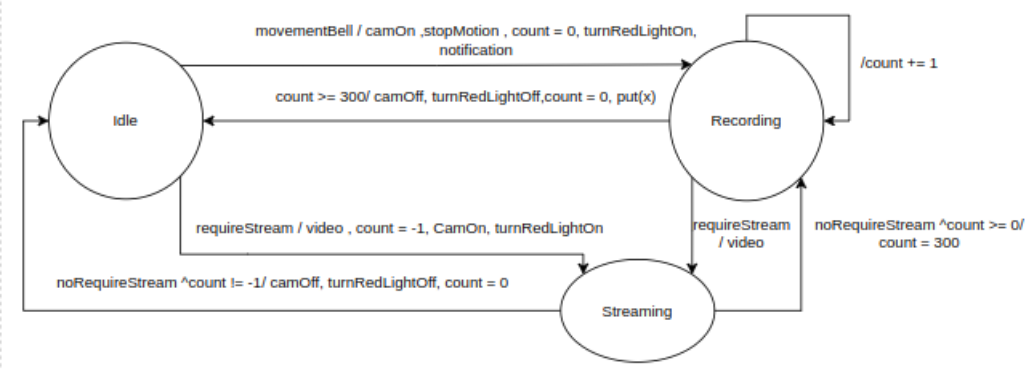

Figure 17: Camera Agent Finite State Machine

5.4 Functional Flow

To exemplify the functional flow of the system, we created a flow diagram comprised of different choices and states.

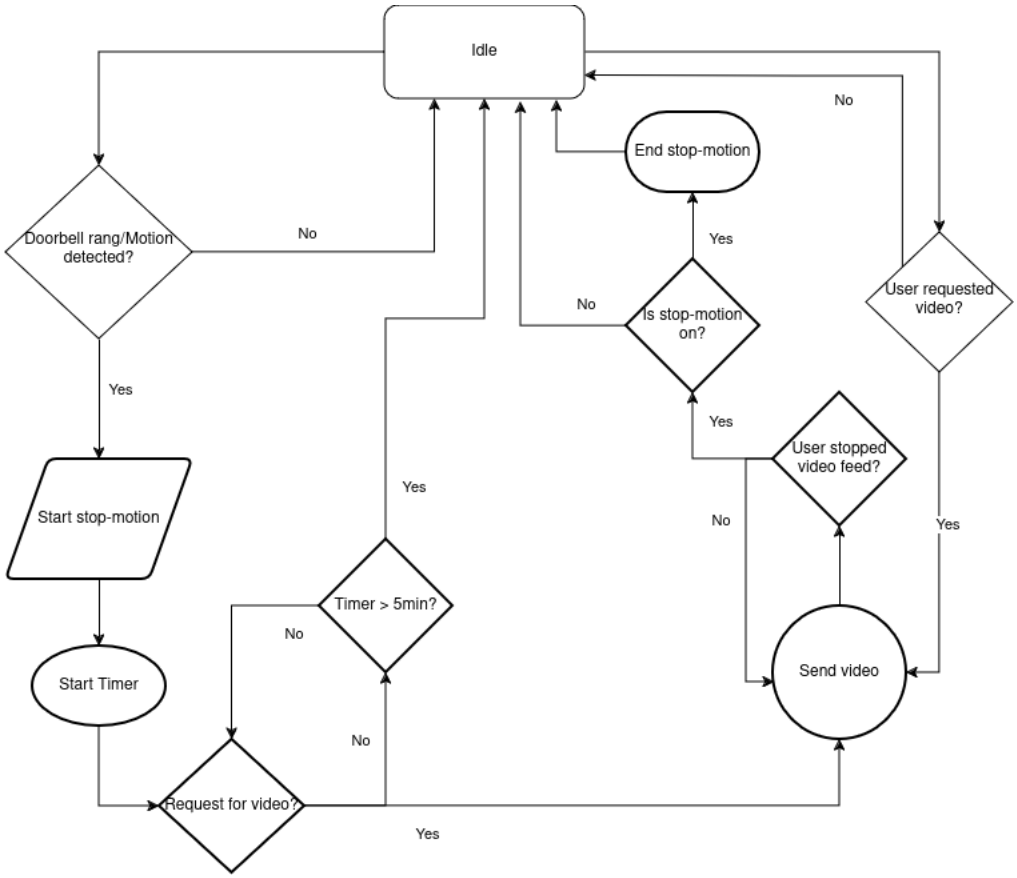

Figure 18: Doorbell Flowchart

First, we created an idle state, where the system will wait on some form of input, given either by the sensors or the user. The system stays in this loop while it waits.

- Choices at ***Idle*** state:
  1. If the user requested video feed, send video
  2. If there was motion detected or the doorbell rang, Start the stop-motion and the timer

- Choices at ***Request for video*** state:
  1. If no video was requested yet and the timer is **greater** than 5 minutes, terminate session
  2. If no video was requested yet and the timer is **smaller** than 5 minutes, return to the loop on ***Request for video*** state
  3. If video was requested ignore the timer (Set it to 0 and stop) and ***Send video***
- Choices at ***Send video*** state:
  1. Stay in the loop until user asks to stop the video feed
  2. If the user asked to stop the video feed and the stop-motion was **on**, end stop-motion and return to ***Idle*** state
  3. If the user asked to stop the video feed and the stop-motion was **off**, simply return to the ***Idle*** state

6 Planning

ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Make list of equipments to request	1	1	1	1	100%
Develop diagrams and first report	1	3	1	3	90%
Connect Arduino to sensor and test	4	2			0%
Implement RPi socket connections and traffic handling	4	2			0%
Test ML Model with camera	4	1			0%
Build mobile app	4	3			0%
Test Arduino + RPi	6	1			0%
Test RPi + App	7	1			0%
Test complete architecture	8	2			0%
Test video and stop-motion streams	8	2			0%
Final test	10	1			0%

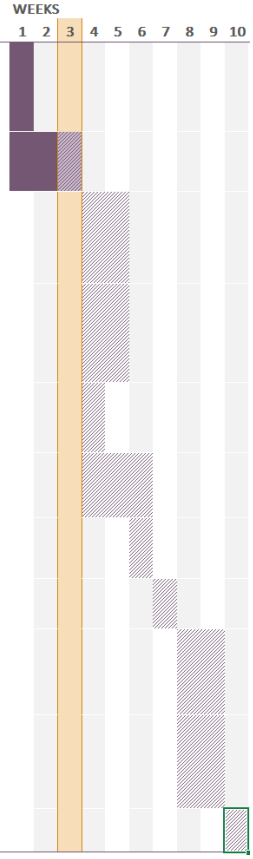

Figure 19: Gantt diagram

We brainstormed some possible breakdowns of the project into tasks, and we settled on the one exemplified by the Gantt chart below. We incorporated several tests and set aside time to connect the different electronic parts, and this resulted in a 10-week chart.

Also, given there are several tasks scheduled for the same time frame and we considered that each of us would be working in parallel, some tasks can only be started when the required dependencies are met.

This Gantt chart was made using Microsoft Excel software, so it can easily be updated to reflect any changes in scheduling. The original file for the chart can be accessed through the GitHub repository for the project.

7 Implementation

Here we describe how we implemented each component in our system , the code can be seen in https://github.com/bmarrao/TPSE

7.1 Arduino

The implementation of the Arduino component of the project can be divided into two main parts: the hardware component and the software component. This division allowed us to address each aspect independently, ensuring thorough development and integration of both components for a cohesive solution.

Unfortunately, we were unable to obtain a motion sensor. After discussing this with our teacher, we decided that the best option would be to discard this element and work solely with the tactile button for activating the doorbell camera.

Since one of the objectives was to have the camera stop recording after 5 minutes of motion detection, we adapted this to stop recording a set time after the button was pressed. For demonstration purposes, we set the time to 30 seconds, but this can be easily adjusted for real-life usage.

7.1.1 Hardware

To implement the hardware, we utilized an external breadboard to connect peripheral components such as the red LED and the tactile button to the Arduino.

Here are the steps we followed for this implementation:

1. **WiFi Shield**: We began by inserting the WiFi shield into the Arduino board.
2. **Tactile Button**: Next, we connected a tactile button to the breadboard. Using male-to-male jumper wires, one side of the button was connected to pin 2 of the WiFi shield, and the other side was connected to the ground (GND) pin.
3. **Red LED Light**: For the red LED light, we connected it with a 1000Ω resistor on the breadboard. Using male-to-male jumper wires, one leg of the LED was connected to pin 9 of the WiFi shield, while the other leg was connected to the ground. Pin 9 was chosen to avoid conflicts with the WiFi shield, as it utilizes certain larger pins for communication purposes.

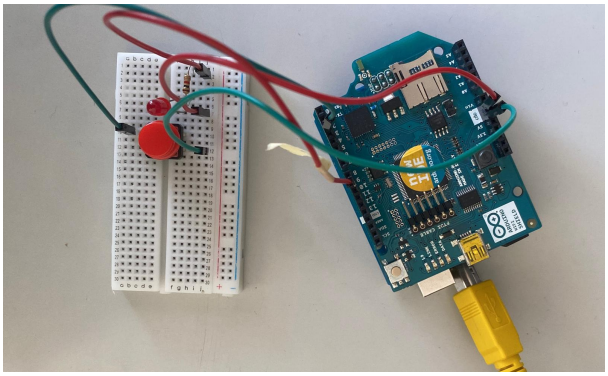

Figure 20: Arduino Hardware

7.1.2 Software

For the software component, we utilized the Arduino IDE for coding and compilation. The program is structured into two primary sections: setup and loop.

- **Setup**: We employed the WiFi.h library to establish connectivity to the WiFi network and configure a server. This server is designed to listen for incoming communication from the Raspberry Pi.
- **Loop**: Within the loop, we manage two distinct scenarios that require action:
  - **Button Clicked**: Upon detecting a button press, a designated function is invoked. This function initiates a GET request to the Raspberry Pi, utilizing the route 'movementBell' to indicate that the button press event has occurred.
  - **Communication Received from Raspberry Pi**: When the Arduino receives communication from the Raspberry Pi, it triggers another function to process the incoming message. If the message instructs 'redLightOn', the code activates the red LED. Conversely, if the message instructs 'redLightOff', the code turns off the red LED.

This setup ensures responsive interaction between the Arduino and Raspberry Pi, enabling effective control and coordination based on external inputs and communication signals.

7.2 Raspberry Pi

The implementation of the Raspberry Pi component of this project can be divided into two main sections: developing the Raspberry Pi API and establishing the connection with the camera.

During this implementation, we encountered several challenges, primarily due to inadequate equipment.

Initially, the selected Raspberry Pi lacked the necessary port for the camera module, prompting us to switch to another Raspberry Pi model. This model, however, lacked Wi-Fi capability, which we compensated for by using a USB antenna for connectivity.

Additionally, the Raspberry Pi camera did not function, necessitating the use of a USB-connected webcam. This also posed a problem due to the limited number of USB ports on the Raspberry Pi.

Despite these challenges, we aimed to achieve the best possible results in the implementation of this section.

7.2.1 Raspberry Pi API

The development of the Raspberry Pi API can be divided into three main sections:

- Database Integration
- Creation of application endpoints
- Execution

Environment Setup and Database Connection

Environment variables like database connection details (HOST, DATABASE, USERNAME, PASSWORD, PORT) and MAIN_DIRECTORY for storing video files are loaded from a **.env** file.

The **create_db_connection()** function establishes a connection to the PostgreSQL database using the credentials from the environment variables.

Listing 1: Connection establishment to PostgreSQL database

```
import os
import psycipg2
from flask import Flask, jsonify, send_file
from dotenv import load_dotenv
from ip import get_current_ip

load_dotenv()
MAX_ATTEMPTS = 5

api = Flask(__name__)
HOST = os.getenv('HOST')
DATABASE = os.getenv('DATABASE')
USERNAME = os.getenv('USERNAME')
PASSWORD = os.getenv('PASSWORD')
MAIN_DIRECTORY = os.getenv('MAIN_DIRECTORY')

def create_db_connection():
    return psycopg2.connect(
        host = HOST,
        database = DATABASE,
        user = USERNAME,
        password = PASSWORD,
        port = PORT
    )
```

Routes/Endpoints

- **Test API Connection**:
  - **Endpoint**: /api
  - **Method**: GET
  - **Description**: This endpoint serves as a health check for the API. When a GET request is made to this endpoint, it returns a JSON response indicating that the API is operational. This is useful for quickly verifying that the API server is running and responsive.

  - **Response**: A JSON object with a message confirming the API is working, structured as “message”: “API is working”. The HTTP status code for a successful request is 200.

- **Warn raspberry that bell has been rung**
  - **Endpoint**: /api/bell
  - **Method**: GET
  - **Description**: This endpoint serves as a way for the arduino to warn the raspberry that the bell has been rung and that it needs to start recording. When a GET request is made to this endpoint, it returns a JSON response indicating that it has received the warning.
  - **Response**: A JSON object with a message confirming the API is working, structured as return ‘message’: ‘Started recording for 30 seconds’. The HTTP status code for a successful request is 200.

- **Get Video URL by Timestamp**:
  - **Endpoint**: /api/video_url/<timestamp>
  - **Method**: GET
  - **Description**: This endpoint retrieves the URL of a video based on a provided timestamp. The timestamp should be in the format ‘YYYY-MM-DD HH:MM’. Internally, the timestamp is formatted to remove characters such as ‘-’, ‘:’, and ‘ ’ to create a video ID. The API then queries the database for the corresponding URL of the video.
  - **Response**: If the video is found, the API returns a JSON object with the URL (“url”: “¡video_url¿”) and a 200 status code. If the video is not found, it returns “error”: “Video not found” with a 404 status code. In case of any errors during the database query, it returns “error”: “Failed to retrieve video URL” with a 500 status code.

- **Get All Video Timestamps**:
  - **Endpoint**: /api/video_timestamps/
  - **Method**: GET
  - **Description**: This endpoint fetches all video timestamps from the database. It queries the database to retrieve the created_at timestamp for each video record. This can be useful for applications that need to display a list of available video timestamps to users.
  - **Response**: If timestamps are found, the API returns a JSON object containing a list of timestamps (“timestamps”: [“<timestamp1>”, “<timestamp2>”, ...]) with a 200 status code. If no videos are found, it returns “error”: “No videos found” with a 404 status code. In case of any errors during the database query, it returns “error”: “Failed to retrieve timestamps” with a 500 status code.

- **Stream Video by Timestamp**:
  - **Endpoint**: /api/video_stream/<timestamp>
  - **Method**: GET
  - **Description**: This endpoint streams a video file based on the provided timestamp. Similar to the URL retrieval, the timestamp is formatted to create a video ID, which is then used to query the database for the video’s URL. If the video URL is found, the server constructs the file path and streams the video file to the client.

  - **Response**: If the video file exists in the specified directory, it is sent with a 200 status code. If the file is not found in the directory, the API returns "error": "Video file not found in the specified directory" with a 404 status code. If the video is not found in the database, it returns "error": "Video not found" with a 404 status code. In case of any errors during the process, it returns "error": "Failed to retrieve video" with a 500 status code.

- **Delete Video by ID**:
  - **Endpoint**: /api/video_delete/<int:video_id>
  - **Method**: GET
  - **Description**: This endpoint deletes a video record and its corresponding file from the server based on the video ID. The API first queries the database to find the video URL, constructs the file path, and then deletes both the database record and the file from the directory.
  - **Response**: If the video is successfully deleted, the API returns "message": "Video deleted successfully" with a 200 status code. If the video record is not found in the database, it returns "error": "Video not found" with a 404 status code. If the video file is not found in the directory, it returns "error": "Video file not found in the specified directory" with a 404 status code. In case of any errors during the deletion process, it returns "error": "Failed to delete video" with a 500 status code.

Listing 2: API endpoint to retrieve video URL

```
@api.route('/api/video_url/<timestamp>', methods=['GET'])
def get_url(timestamp):
    connection = create_db_connection()
    cursor = connection.cursor()
    attempts = 0
    # replace '_', ':' and ' ' by '' (timestamp format: 'YYYY-MM-DD HH:MM:SS')
    video_id = timestamp.replace('-', '').replace(':', '').replace(' ', '')

    while attempts < MAX_ATTEMPTS:
        try:
            cursor.execute(
                "SELECT url FROM videos WHERE video_id = %s;",
                (video_id,)
            )
            url = cursor.fetchone()

            if url:
                return jsonify({'url': url[0]}), 200
            else:
                return jsonify({'error': 'Video not found'}), 404
        except Exception as e:
            attempts += 1
            print(e)
        finally:
            cursor.close()
            connection.close()

    # 500 Internal Server Error
    return jsonify({'error': 'Failed to retrieve video URL'}), 500
```

Execution

The API will run on the local network, accessible at the IP address determined by the get_current_ip() function on port 8888.

Listing 3: Function to retrieve IP address from machine/Raspberry Pi

```
def get_current_ip():
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.settimeout(0.5). # set timeout to 0.5 seconds

    try:
        # connect to a public Google DNS server
        s.connect(('8.8.8.8', 53))
        ip = s.getsockname()[0]
    except Exception:
        ip = '127.0.0.1' # fallback to localhost
    finally:
        s.close()

    return ip
```

7.2.2 Camera control

During the implementation of the connection of the Raspberry Pi with the camera, we realised that our Raspberry Pi model supports only one thread at a time, which significantly hampered our implementation. Since there were no other models available to us, this limitation made it challenging to integrate the Raspberry Pi API, which manages interactions with the database, mobile application, and Arduino components, while concurrently connecting to the camera using OpenCV. Ideally, the recording should run in a separate thread to allow users to request information from the Raspberry Pi even during recording.

To address this issue, we had to request camera recordings within the same thread used by the Raspberry Pi API. Consequently, while the camera is recording, the API remains idle and cannot perform any other tasks. As a result, users are unable to interact with the Raspberry Pi or request data during this period. While this solution falls short of our ideal scenario, it was the best workaround feasible with the hardware available to us.

To achieve this every time the api received a specific get request from the arduino, it called a function , record_video, this function starts the camera with the 'opencv' library and records the video ,then stores the video in the memory, and a new timestamp in the database .

Regarding the livestream, no workaround was possible since this requires two threads: one to connect to the camera for streaming and another to concurrently support the API for sending the stream to the mobile app.

However, we developed a Python script to simulate the behavior of a multi-threaded Raspberry Pi:

ADD HERE CODE/IMPLEMENTATION EXPLANATION FOR LIVESTREAM

7.3 Flutter Application

We developed a Flutter application with a main home page that integrates a bottom navigation bar and includes three different pages: ConnectPage, SearchPage, and LiveStreamPage.

The **HomePage** widget is a stateful widget that serves as the main container for the application. It includes a bottom navigation bar created using the *GNav* widget from the google_nav_bar package. The bottom navigation bar allows users to switch between three different pages: **ConnectPage**, **SearchPage**, and **LiveStreamPage**. The *currentPageIndex* variable keeps track of the currently selected page, and the *navigateBottomBar* method updates this index to switch pages. The content of the Scaffold's body is dynamically updated to show the selected page based on *currentPageIndex*.

7.3.1 ConnectPage

The **ConnectPage** widget is designed for users to connect to a server. It includes text fields for entering an IP address and port number. The **ConnectPage** interacts with a **ConnectionManager** to manage the connection state. Upon pressing the connect button, the application tests the connection by sending a request to the provided IP address and port. If the connection is successful, the app updates the **ConnectionManager** with connection details and starts a timer to display the connection duration. If already connected, the page displays the connection details and provides a button to disconnect.

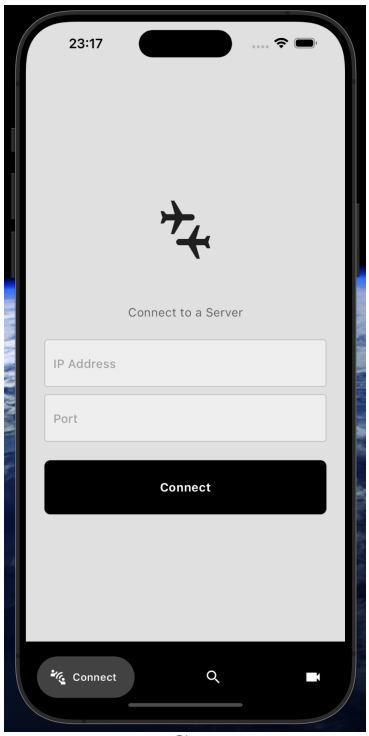

Figure 21: Connect Page

7.3.2 SearchPage

The **SearchPage** widget allows users to search for historical video data based on timestamps. It fetches available timestamps from the server and displays them in a searchable dropdown menu. When a timestamp is selected, it sends a request to retrieve the corresponding video data and navigates to a video player screen to display the video. The **SearchPage** also uses the **ConnectionManager** to manage server connections and handle HTTP requests to fetch the timestamps and video data.

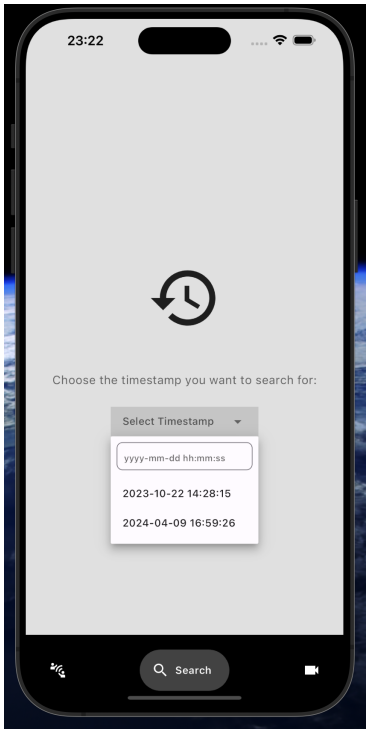

Figure 22: Search Page

7.3.3 LiveStreamPage

The **LiveStreamPage** widget provides functionality to request live stream data. It includes a button that, when pressed, triggers a method to request live stream data for the current timestamp. The request process is simulated by printing the current timestamp to the console. The widget displays a loading indicator while the request is being processed and shows an error message if the request fails. The **LiveStreamPage** is more of a placeholder for future implementation of live streaming functionality.

8 Results

8.1 Arduino

The outcomes with the Arduino were positive. Despite not having the movement sensor, we addressed this issue in consultation with the professor and opted to utilize only the tactile button. However, integrating the sensor would have been straightforward: connecting it to the Arduino on the hardware side and adjusting the software to monitor inputs from either the button or the sensor to trigger messages sent to the Raspberry Pi.

Despite this constraint, our Arduino successfully sends Wi-Fi messages upon button presses and responds to incoming signals to control the light, demonstrating its functionality in communication and responsiveness.

8.2 Raspberry Pi

Despite the threading limitations of the Raspberry Pi, we achieved several key outcomes. We successfully integrated the camera to initiate recordings based on Arduino signals, storing these as stop-motion videos on the Raspberry Pi. We also created a comprehensive database and an API to manage and retrieve these recordings, making them accessible to Android devices.

Although a direct workaround for livestreaming was not possible, our Python script simulation of a multi-threaded environment enabled us to request and view a livestream from the camera on the application, demonstrating the potential functionality in an ideal hardware scenario.

8.3 Flutter

For the user interface component, we successfully developed a mobile application that interfaces with the Raspberry Pi. This application enables users to request the timestamps of videos stored on the Raspberry Pi and select specific videos for viewing by choosing a valid timestamp. Upon request, the selected videos are successfully transmitted from the Raspberry Pi to the mobile app, allowing users to view them on their phones.

Additionally, the application provides an option for users to initiate a livestream from the camera, enabling real-time viewing of the recording. Users can start and end the livestream at their convenience, enhancing the application's functionality and user experience.

8.4 Overview

The initial objectives and the final outcomes can be summarized as follows:

Our initial objective was to create a doorbell camera system that would activate upon a doorbell press or motion detection. While we successfully implemented the activation upon doorbell press using a tactile button, we were unable to include motion detection due to the unavailability of the sensor. Despite this, the system still meets the primary goal of activation upon user interaction.

We aimed to record images in a "stop-motion" format to conserve storage space. This was successfully implemented, and the Raspberry Pi efficiently stores these recordings.

An objective was to stop recording after five minutes of no motion detection. Since motion detection was not integrated, this specific feature was not applicable. However, recordings can be managed manually as needed.

NOTIFICATION SYSTEM!!!!

The application allows users to view recorded "stop-motion" images, fulfilling the objective of providing access to recorded footage through a mobile interface.

The request for livestreams was not possible to implement due to lack of access to the needed hardware, but it was successfully simulated through a Python script.

Lastly, the system indicates that the camera is recording through the activation of an incorporated red light, meeting our objective to provide a visual indication of recording status.

9 Conclusion

In conclusion, this project successfully achieved most of its initial objectives, despite facing several challenges related to hardware limitations. We managed to integrate the Raspberry Pi with a webcam and develop a mobile application that provides users with access to recorded stop-motion videos. The absence of a motion sensor was mitigated by focusing on tactile button functionality, which proved effective in triggering recordings.

Despite the inability to reconcile the connection to the camera and the use of the Raspberry Pi API to request a livestream through the mobile app, we were still able to simulate this functionality with a Python script that acted as a Raspberry Pi simulator.

The development of a robust API and a user-friendly mobile interface enhanced the system's

overall functionality and user experience.

While some objectives related to motion detection and livestreaming were not met, the project demonstrated a comprehensive integration of hardware and software components, resulting in a functional doorbell camera system.

Future work could focus on incorporating motion detection and further optimizing hardware compatibility to fully realize the initial project vision.