SYSC3010 Computer Systems Development Project
FANS Detailed Design Report

Group L1-G8

[1]

Grant Achuzia, 101222695
Javeria Sohail, 101197163
Matteo Golin, 101220709
Saja Fawagreh, 101217326
TA: Sean Kirkby

Created: March 11th, 2024
Modified: March 11, 2024

Contents

1 Problem Statement

1.1 Functional Requirements

1. TODO

2 Design Overview

2.1 System Overview Design

2.2 Communication Protocols

2.2.1 I2C Communication

Sender	Receiver	Message	Data Format	Protocol

Table 1: Messages for I2C communication in FANS.

2.2.2 Local Area Network Communication

Sender	Receiver	Message	Data Format	Protocol

Table 2: Messages for local area network communication in FANS.

2.2.3 Cloud Database Communication

Sender	Receiver	Message	Data Format	Protocol

Table 3: Messages for cloud database communication in FANS.

2.2.4 User Notification Communication

Sender	Receiver	Message	Data Format	Protocol

Table 4: Messages for user notification communication in FANS.

2.3 Message Sequence Diagrams

2.3.1 Trigger Emergency Use Case

2.3.2 Add New Contact Information Use Case

2.3.3 Change Emergency Threshold Use Case

2.3.4 Emergency Response Use Case

2.4 Database Table Schema

3 Software Design

Each node in the FANS system has its software design outlined below. Simple system nodes with an algorithmic design have their control flow described using state machine diagrams as an aid. Class diagrams are included for nodes that use an object-oriented approach for their design.

3.1 Sensor Data Collection System

The sensor collection system will be programmed using the Python programming language for simplicity and its large collection of libraries.

The structure of the program will be that of two continuous polling loops, running as separate processes to utilize the CPU fully and get around the Global Interpreter Lock (GIL) of Python that prevents maximizing traditional thread performance.

The first polling loop will be responsible for collecting sensor data continuously and placing it on a queue for the other loop. It will collect data from all sensors, and that is its sole responsibility.

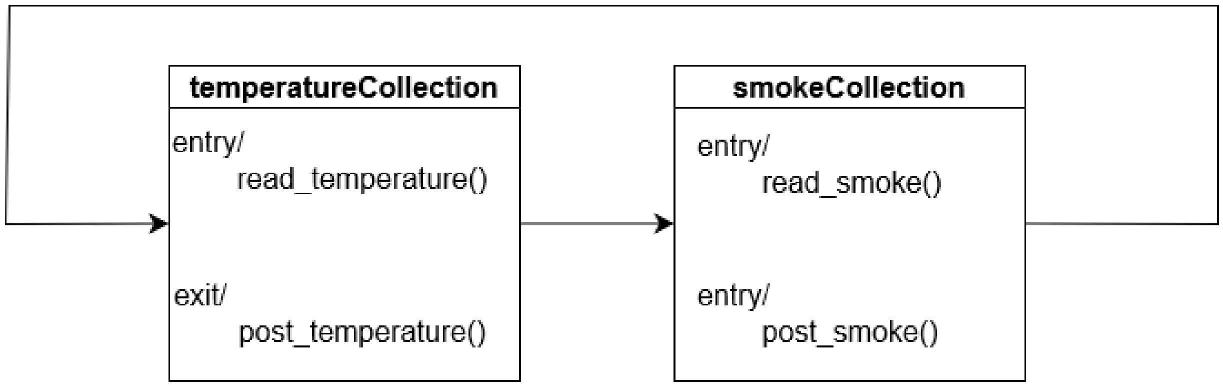


Figure 1: The polling loop for sensor data collection.

The second polling loop will be responsible for reading the sensor data from the shared queue and writing it to the Firebase database. It will also be responsible for performing logical checks on this data to determine whether or not there is an active fire emergency. It will update the emergency flag in the database whenever a sensor data measurement is above the specified threshold, and also communicate the emergency over UDP to the notification system and alarm system. Once all sensor data has been posted to the database, this loop will check for configuration updates in the sensor data thresholds from the Firebase database and apply them to the next iteration.

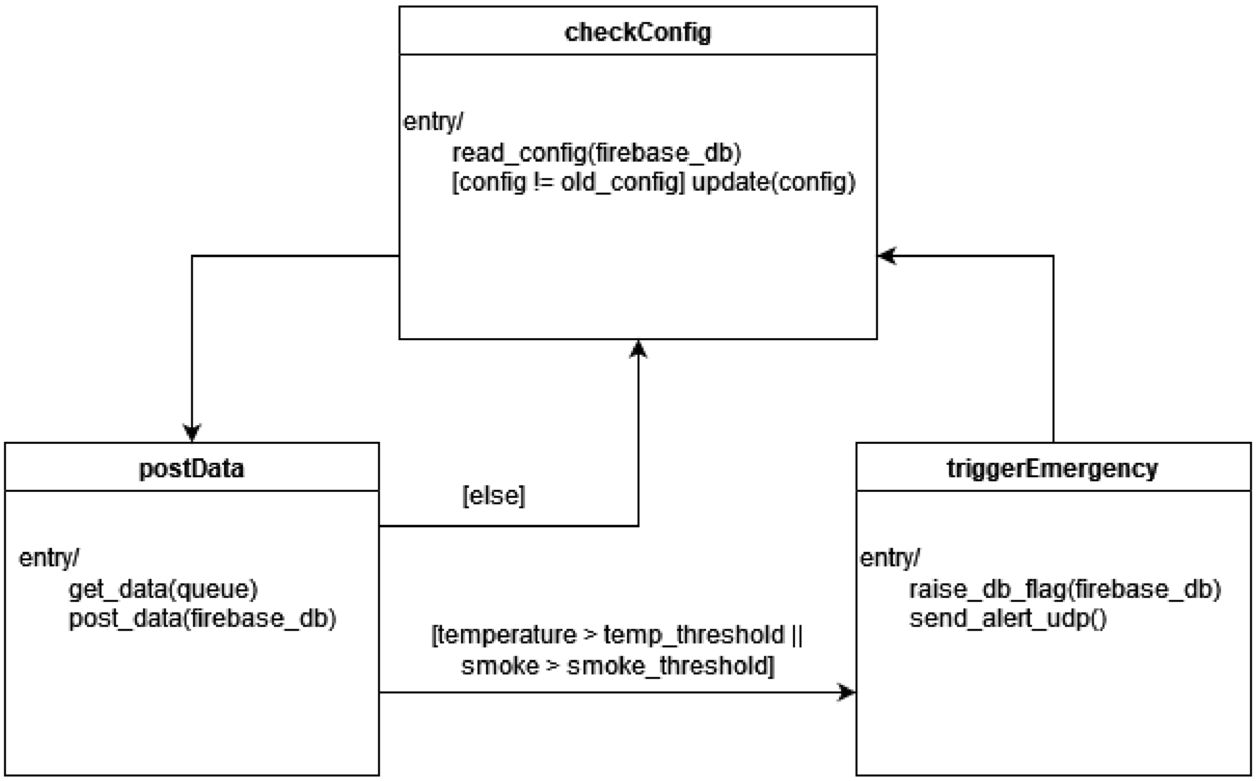


Figure 2: State machine diagram for the second process of the sensor data collection system.

3.2 Alarm System

The alarm system will be programmed using the Python programming language, again for its simplicity. The alarm system will be composed of one continuous loop, which waits to receive incoming UDP messages. When a UDP message signifying an emergency is received, the alarm system will trigger both an alarm buzzer and flashing LED lights for an infinite duration. The alarm response will be interrupted only when the system receives another UDP message signifying that the emergency has ended. This behaviour is similar to a state machine, so the software will be created using the state design pattern.

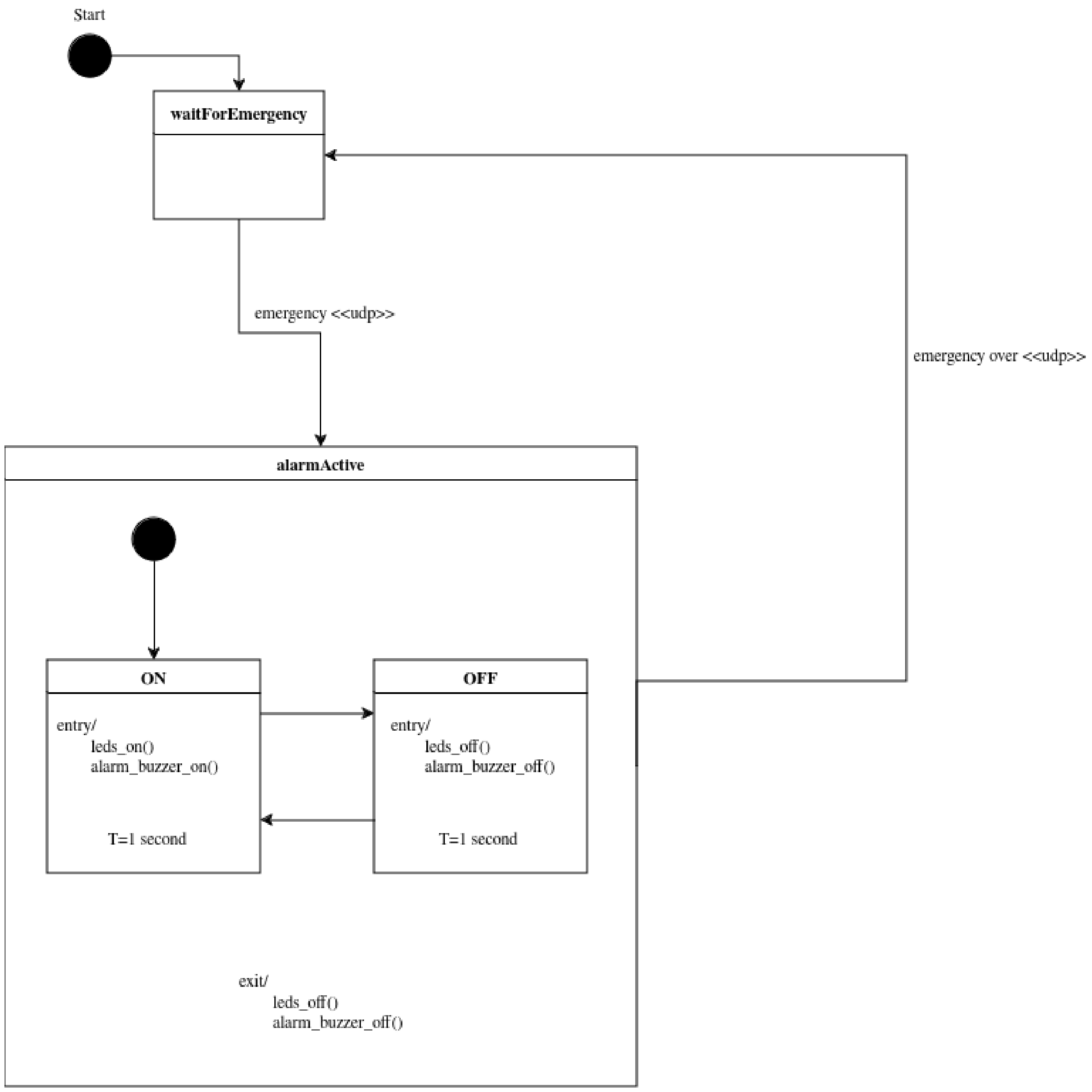


Figure 3: State machine diagram for the alarm system node.

3.3 Notification System

The notification system will be written in the Python programming language, also due to its simplicity. The notification system will listen for a UDP message signifying an emergency, and then send out email and SMS notifications to all users in the Firebase database. Once a UDP message signifying the end of the emergency is received, the system will send a followup email to all users that the emergency has been resolved.

The notification system will keep a local cache of user contact information as a backup for failing internet connectivity. It will periodically update its local cache with any changes in the upstream

Firebase database.

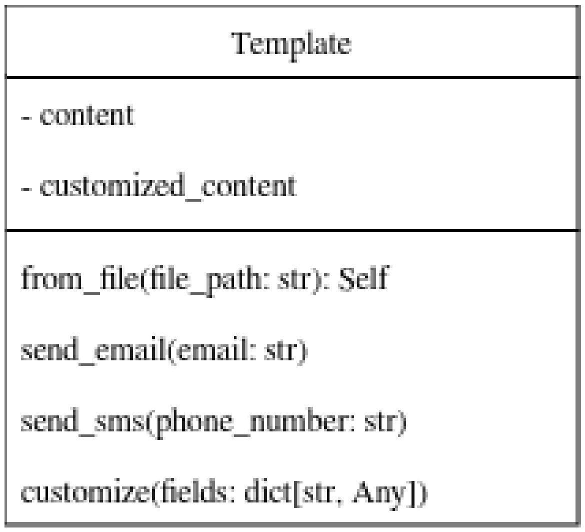
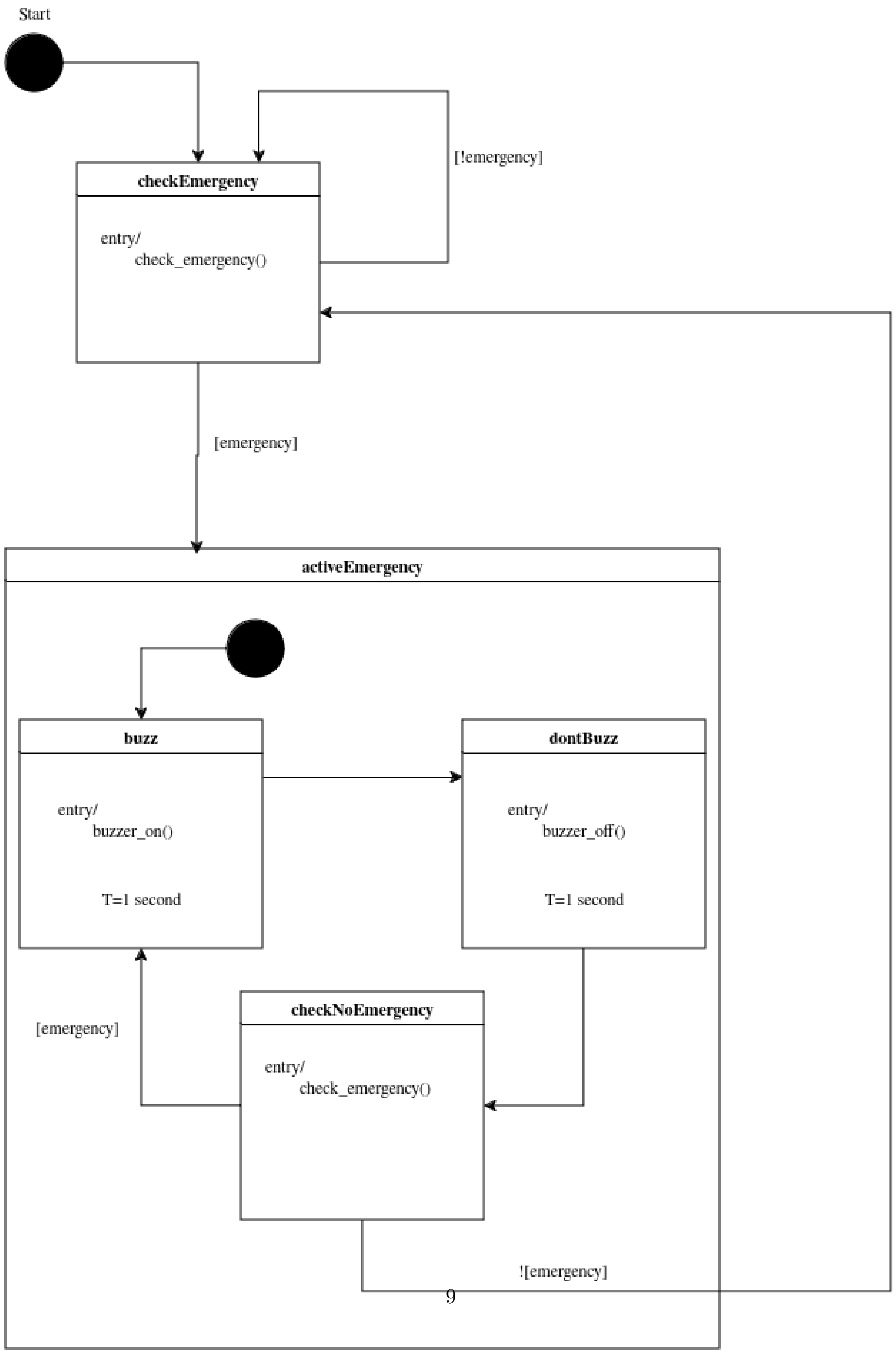

Template
- content
- customized_content
from_file(file_path: str): Self
send_email(email: str)
send_sms(phone_number: str)
customize(fields: dict[str, Any])

Figure 4: Class diagram for the notification system node.

The notification system will also have locally stored email and SMS templates for notifications. These will be loaded as "Templates", which provide an interface for easy customization and sending of notifications.

3.4 Haptic Alarm System

The haptic alarm wearable also has a very simple software design, and will be written in the Python programming language. The program will continuously poll the Firebase database for changes in the emergency flag. Once the emergency flag has been raised, the haptic alarm will begin to buzz on and off. During this time, it will continue checking for changes in the emergency flag on the Firebase database. Once the emergency flag is lowered, the haptic alarm will stop buzzing, signifying the end of the emergency.


Start
checkEmergency
entry/ check_emergency()
[!emergency]
[emergency]
activeEmergency
buzz
entry/ buzzer_on()
T=1 second
dontBuzz
entry/ buzzer_off()
T=1 second
checkNoEmergency
entry/ check_emergency()
[emergency]
![emergency]

4 Hardware Design

4.1 MQ2 Smoke Sensor

4.2 LCD Display

4.3 AudioHat Module

4.4 Piezoelectric Buzzer

5 GUI Design

5.1 Table of Users/Roles

6 Test Plans

Testing is an important part of any system's design. It ensures that project iterations follow the specification, behave appropriately and are less likely to include bugs.

6.1 End-to-end Communication Demo

The end-to-end communication demo should demonstrate all of the major forms of communication present in the FANS system. In order to achieve this level of demonstration, the following communication examples will be showcased:

- Sensor data upload to Firebase over internet.
- Email notifications to affected users over internet (email protocol).
- Haptic alarm buzzing in accordance with emergency flag in Firebase over internet.
- Display of sensor data on the user interface over internet.
- Emergency signal from sensor data collection system to the alarm system on local network.

It should be noted that the test for the sensor data upload over internet also demonstrates I2C communication between the Pi 4 and the SenseHat board.

First is the uploading of sensor data to the Firebase database over the internet. For this communication sequence, the sensor data collection system will be run on one of the Raspberry Pis. It will read temperature data from the SenseHat and post this data to the Firebase database using Pyrebase. The demonstration can be verified by watching the Firebase database console, since the data will be updated in real-time.

Second will be the email notifications to affected users over the internet using email protocol. To achieve this demonstration, the email notification system will be run on one of the Raspberry Pis. Using Pytest, the program will first send an email from the FANS email back to itself. Then, the program will use `imaplib` to login to the FANS email account and verify that the latest message in its inbox contains the same email message that was sent in the test. If the messages match, Pytest will report a success. This test is fully automated.

Next is the haptic alarm buzzing. For this test, the haptic alarm program will be run on one of the Raspberry Pi's to emulate the Pi Pico (as it has not arrived at the time of writing). When the program is run, it will poll the Firebase database using `pyrebase` to check if the emergency flag has been raised. The flag can either be raised by hand (the tester modifies the flag using the Firebase console) or it can be set by running the sensor data collection system with a low emergency reporting threshold for temperature. In either case, when the flag is raised, the output of the haptic alarm can be verified by observing the console messages. Console printing will be used in place of the buzzer actuator while we wait for components. A console message of ”buzz” would indicate the buzzer buzzing, and no console messages indicate no emergency while the alarm is polling the database.

To test the display of sensor data collection on the user interface, the GUI will be run on one of the Raspberry Pis using Flask. When a user clicks the ”refresh” button, the Flask API endpoint responsible for returning temperature data will be hit. The endpoint will use `pyrebase` to request the latest temperature data from Firebase and then will respond with JSON. The GUI's JavaScript logic will simply print this JSON as text content of an HTML paragraph tag to show that the connection is present.

Finally, to demonstrate UDP communication over the local area network, both the sensor data collection system and the alarm system will be run on two separate Pis. Both programs will first upload their public IP to the Firebase database under a 'devices' table, and then read the other device's IP. The sensor data collection system should have a sufficiently low emergency threshold to trigger an emergency at room temperature (only for testing purposes). Once the sensor data collection system detects an emergency, it will send a UDP message containing the numerical data '0' (the agreed upon encoding for signifying an emergency) over the LAN addressed to the alarm system. The alarm system will receive the emergency notification, and instead of activating an actuator it will print "Received 0" to the console. This can be manually verified.

6.2 Unit Test Demo

The purpose of the unit test demo is to demonstrate that the individual components of our system behave as expected and do not contain logical or physical errors. This will sanity check our hardware components and our software logic.

6.2.1 Hardware

The hardware components that will need to be tested are:

- Smoke sensor
- Temperature sensor
- Piezoelectric buzzer
- LCD screen

Smoke Detector

The smoke detector will be difficult to test due to the nature of what it measures. It will not be possible to light a fire within the school or near campus because that will set of existing fire suppression systems and may be dangerous. Testing the smoke detector in smokey environments can be done by:

- Running the data collection from the smoke detector in a smoking zone on campus.
- Running the data collection from the smoke detector off campus and lighting a fire nearby.

For both of these tests, it should be verified that smoke levels are detected. This will be a one-time sanity test of the sensor. The smoke detector can also be tested in smokeless environments by collecting data from the sensor indoors on campus and ensuring that no smoke is detected.

Temperature Sensor

To test the temperature sensor, we can collect data from the sensor in the lab setting and ensure that measurements are within the acceptable range for room temperature.

Piezoelectric Buzzer

To test the piezoelectric buzzer, we can test turning on and off the buzzer and verifying that it produces noise. Additionally, we can set the buzzer to produce different frequencies and verify that they are set correctly by checking with a instrument tuning app.

LCD Screen

Finally, to test the LCD screen, several test messages can be displayed on screen and visually inspected for correctness as a sanity check.

6.2.2 Database Integrity

To test cloud database integrity, we can perform several test and security checks.

Insecure Access

To test the security of the cloud database, we can attempt to perform read and write operations using an incorrect API key and ensure that the requests are denied.

Stress Test

To stress test the cloud database, several instances of the sensor data collection program can be run on multiple Raspberry Pis, all of which will upload sensor data to the database. The database should:

- Keep up with all of the simultaneous requests.
- Avoid any overwriting/race conditions with concurrent write operations.

6.2.3 Software

6.3 Final Demo

7 Project Update

7.1 Project Milestones

7.2 Schedule of Activities

References

[1] P. Matoušek, *Thick smoke on black background*. [Online]. Available: `https://www.freeimages.com/photo/thick-smoke-on-black-background-1633270` (visited on 02/11/2024).