An Internet Connected Ventilation System for Residential Homes

Mobile Systems

Group:
SW802F19

Supervisor:
Michele Albano

November 13, 2019

Department of Computer Science
Aalborg University
http://cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
An Internet Connected Ventilation System for Residential Homes

Theme:
Mobile Systems

Project Period:
Spring 2019

Group:
SW802F19

Participants:
Anton Christensen
Casper M. Grosen
Henrik H. Sørensen
Jacob A. Svenningsen
Kim Larsen
Martin F. Karkov

Supervisor:
Michele Albano

Pages:
52

Date of Completion:
November 13, 2019

Number of Copies:
1

Abstract:

To reduce the energy consumption and noise pollution of residential ventilation systems we design and implement a system based on modular units that combines sensor data from the whole house with occupants' location to intelligently decide how to ventilate. We are provided with a hardware platform by Advanced Electronics Repair and create firmware for it, we design a centralised server for the units to communicate with and implement a mobile app that allows users to manage units in their house by setting a desired temperature and visualising historical data. We also design a set of tests to verify the correct working order of many different aspects of the system and the system as a whole. We investigate methods for indoor localisation and decide to use the signal strength of Wi-Fi signals and lateration to obtain information about which rooms occupants are in. Our finished system is functional, but remains untested in a real world setting. Additionally, it requires more work regarding security, user experience and localisation.

Anton Christensen
achri15@student.aau.dk

Casper M. Grosen
cgrose15@student.aau.dk

Henrik H. Sørensen
hsaren14@student.aau.dk

Jacob A. Svenningsen
jsvenn15@student.aau.dk

Kim Larsen
klars15@student.aau.dk

Martin F. Karkov
mkarko15@student.aau.dk

Preface

Source Code

The code written in this project is not public, as it could disclose confidential information regarding the devices commercialised by Advanced Electronics Repair.

Special Thanks

David Stien Pedersen, Advanced Electronics Repair

Acronyms

AERep Advanced Electronics Repair

CO_2 Carbon Dioxide

GPIO General Purpose Input Output

I^2C Inter-Integrated Circuit

IAQ Indoor Air Quality

IEQ Indoor Environmental Quality

IoT Internet of Things

JSON JavaScript Object Notation

MQTT Message Queuing Telemetry Transport

PIR Passive Infrared

PRL Pseudo-range Lateration

PWA Progressive Web App

RSS Received Signal Strength

RSSI Received Signal Strength Indication

TDOA Time Difference of Arrival

TRL True-range Lateration

UML Unified Modeling Language

UWE UML-based Web Engineering

WHO World Health Organization

Contents

Chapter 1

Introduction

Maintaining good air quality is important to our health, but it is not uncommon for indoor environments to have bad air quality which is often caused by a lack of ventilation. Opening doors and windows can be a good source of ventilation, but they tend to stay closed especially in cold or humid months. Particles which can be harmful to our health accumulate in the air and during these periods the levels of them are seldom brought down. Systems exist that can ensure constant ventilation when it is needed. Such a system can consume a lot of power and cause discomfort due to noise. In an attempt to mitigate this, we present a solution for tracking the location and number of people in individual rooms in order to focus ventilation where it is needed the most.

Initial Problem Statement

How can a ventilation system ensure good air quality by tracking the location of occupants?

We are interested in working with embedded systems and developing an Internet of Things (IoT) platform, which is why we accept an opportunity to work with David Stien Pedersen, the owner of the local business Advanced Electronics Repair (AERep). AERep supplies us with a hardware platform, ideas and insight.

In order to better understand the initial problem, we explore the problem domain, by investigating what good air quality entails and how ventilation systems operate. Additionally, we look into how the location of occupants in a residential home can be used to improve the way a ventilation system operates.

Chapter 2

Problem Analysis

In this chapter analyse what makes a good Indoor Air Quality (IAQ), which we use to determine specific factors to focus on when developing a ventilation system. We look into the different types of ventilation systems that exist and the techniques they use. Additionally, we discuss what a mobile system is and argue for why the system we develop, is a mobile system. Lastly, we investigate the advantages of detecting and tracking nearby users to preemptively ventilate to improve IAQ.

2.1 Indoor Climate

In this section we analyse what makes good Indoor Environmental Quality (IEQ) and give our own definition of IAQ.

IEQ is a general term which covers several aspects of an indoor environment. Toftum, Wargocki, and Clausen (2011) define it as a combination of the following four terms:

Air Quality Dust, gasses, steam and particulate matter in the air.

Thermal Conditions Thermal radiation, air temperature, air velocity and humidity.

Light Conditions Light intensity, colours, contrasts and reflections.

Sound Conditions Amplitude and frequency of sound waves.

While all four are important for how people experience an indoor environment, we are mainly interested in air quality and thermal conditions, which we combine into the term IAQ and cover in more detail below. Additionally, we are interested in the noise pollution that is produced through ventilation systems. While we have not been able to find a singular definition of what IAQ entails, several countries have created their own so-called Air Quality Index, which relate to outdoor air, but is based on similar conditions (U.S. Environmental Protection Agency, 2019).

2.1.1 Indoor Air Quality

According to the European Commission, there are multiple factors that play a role in IAQ (SCHER, 2007). They mention several toxic chemicals such as carbon monoxide, nitrogen dioxide, benzene, radon, tobacco smoke, but also temperature, humidity and particulate matter. Additionally, the World Health Organization (WHO) focuses on complex hydrocarbons, which have a tendency to

attach themselves to small particles that are then inhaled (WHO, 2010). Combining this„ with the previous definition, leads us to mainly consider a selection of gasses, particulate matter of a certain size, humidity and temperature as the main constituents in deciding the IAQ.

Even in small concentrations, most of the problematic gasses either pose a direct hazard to human health through a carcinogenic effect or cause sensory irritation (WHO, 2010). While not toxic by itself, high concentrations of Carbon Dioxide (CO_2) have been proven to significantly lower perceived air quality and concentration levels while increasing health risks (Seppänen, Fisk, and Mendell, 1999; Ole Fanger, 2006). Some studies find that continually ventilating to lower CO_2 concentrations decreases the associated health risks (Seppänen, Fisk, and Mendell, 1999). Additionally, studies have found that indoor concentration of CO_2 increase rapidly when people are present (Toftum, Wargocki, and Clausen, 2011; Turanjanin et al., 2014). Standards set by the Danish government suggests that for optimal indoor climate, the concentrations of CO_2 must be below 1000ppm, with 2000ppm being the upper limit of what is an acceptable level in working environments (Arbejdstilsynet, 2018).

Another factor to IAQ is humidity, which can be described as absolute humidity or humidity relative to temperature (Zehnder, 2014).

Absolute humidity The measure of water molecules by weight present in a given volume of air. Usually measured in g/m^3.

There is an upper limit to how much water can be contained in air, depending on the temperature and pressure of the air.

Relative humidity At a certain absolute humidity it saturates the air and additional water vapour will condense as dew. Since this saturation point depends on the temperature and pressure of the air, it makes sense to express humidity as the ratio of absolute humidity to the saturation point in the current air conditions. In the context of IAQ the pressure differences are often negligible, such that the temperature and absolute humidity is the main factor in relative humidity. Usually measured in percentages.

Relative humidity is a relevant component of human comfort levels, as a high relative humidity impedes our ability to evaporate sweat, which makes it difficult to cool our bodies. According to the Danish government the relative humidity levels in indoor climates should be between 25% and 60% (Arbejdstilsynet, 2018). A low level of humidity can cause dry skin and irritation, while a high level of humidity can foster mould growth, which can potentially be hazardous to human health (Heseltine and Rosen, 2009).

The temperature of indoor air also has a high impact on the perceived air quality. For office work, optimal working conditions are achieved when the air temperature is around 20-22 °C, with temperatures above and below impacting people in a negative way. The optimal working temperature depends on the clothing worn by a building's occupants. However, the temperature should never exceed 25 °C (Arbejdstilsynet, 2018).

Varying temperatures can also be found to be uncomfortable as this may feel like draught. Temperature should not vary by more than 4 °C during a workday. Mechanical ventilation systems should not cause people who sit nearby to experience air velocity from the system above 0.15 m/s either, as this would also cause them to experience the same feeling (Arbejdstilsynet, 2018).

We find several environmental factors that play a major role in indoor comfort levels. We define IAQ as the combination of temperature, humidity and CO_2 levels.

2.2 How to improve Indoor Air Quality

According to Ole Fanger (2006) an ever increasing focus on conservation of energy has led to modern buildings with poor circulation of fresh outdoor air. This has caused a rise in illnesses such as asthma especially among children. Improving the IAQ can decrease the symptoms and prevalence of such illnesses. In this section we explore different existing technologies, which can help to improve the IAQ of residential homes.

2.2.1 Behavioural Methods

To improve the IAQ without use of specialised mechanical equipment, several methods that involve modifying the behaviour of people can be utilised (Harvard Women's Health Watch, 2018). Teaching people to open their windows and cleaning regularly are both examples of such methods. Regular ventilation through open windows and doors can allow old stale air to leave the building and fresh air to enter. However, there is little to no control of the quality of the air entering, which can potentially be polluted by mould spores, nearby fires, exhausts or other polluting sources. Additionally, the temperature and humidity of the outdoor air might not be desirable indoors. Cleaning can improve IAQ by removing harmful particles from surfaces, which prevents them from polluting the air. A lot of harmful particles gather with dust, including radon, so removing dust can help remove dangerous particles. While plants can help to reduce the levels of CO_2 in the air, by converting it to oxygen, they also increases relative humidity. The benefit is often outweighed by the increase in humidity and potential mould the greenery can introduce (Gubb et al., 2018; Heseltine and Rosen, 2009).

2.2.2 Ventilation Strategies

Alternatives to relying on building occupants to improve the IAQ, are natural and mechanical ventilation. Natural ventilation is the simplest, as it refers to unaided airflow through a building. This can either be through open doors, windows and leaks in the building (U.S. Department of Energy, 2019a). The main drawback to this type of ventilation is the fact it often wastes a lot of the energy that has been used to bring the indoor air to a pleasant temperature and humidity. Being energy inefficient natural ventilation has become less prevalent in newer buildings.

An alternative to natural ventilation, is mechanical ventilation, which can be further classified into two kinds of systems: spot ventilation and whole house ventilation. The former relates to focusing ventilation in areas where the air quality is often worse, such as humid bathrooms and kitchens where cooking pollutes the air, with the latter being systems that uniformly ventilate the entire building through ventilation fans and duct systems (U.S. Department of Energy, 2019a). This is the primary alternative to natural ventilation and is what we will be focusing on. It improves the IAQ in the entire building, but can be costly because of the infrastructure requirements. Additionally, it might require spot ventilation in areas with more pollution sources.

Whole house ventilation systems can be further split into three main categories, exhaust, supply, and balanced (U.S. Department of Energy, 2019b).

Exhaust Ventilation Systems

Exhaust ventilation systems use active ventilators to lower the building's internal pressure. This lower pressure is then relieved by outside air flowing in through passive vents and leaks in the structure. This creates air diffusion that dilutes the air inside the building with outside air. Since exhaust ventilation systems are dependant on drawing outside air in through the structure, it can

lead to moisture damage if the indoor air temperature can not contain the humidity levels of the outdoor air, which means the excess humidity condenses in the leaks. Because of this, exhaust ventilation systems are mostly applicable in environments where the outdoor air is colder than the inside air. Furthermore, pollutants might enter the building, along with the outdoor air (U.S. Department of Energy, 2019b).

Supply Ventilation Systems

Supply ventilation systems work by pressurising the building. It accomplishes this with ducts, fan systems or a combination of the two, forcing outside air into the building. The pressure makes indoor air leak out of the building through leaks or vents in the building structure. Seeing as this is the opposite of exhaust ventilation systems, it also suffers from the same problem of differences in humidity between the indoor and outdoors, but in reverse. Therefore, supply ventilation systems are best used in areas where the indoor air is colder than the outdoor air, as this reduces the risk of moisture collecting in the building's structure. An advantage of supply ventilation systems over exhaust, is that the intake of air happens at specific points, which means filters and similar can be used to control the quality of the air that enters the building. Furthermore, this also allows the temperature of the intake air to be changed at the cost of increased energy (U.S. Department of Energy, 2019b).

Balanced Ventilation Systems

Balanced ventilation systems replace the indoor air with outdoor air, but maintains the air pressure of the building while doing so. This is achieved by supplying the same amount of air that is being exhausted combining the exhaust and supply ventilation systems. This also means that there are no restrictions on the difference in temperature between the inside and outside air. This is because any condensation occurs at specified points in a building's structure, reducing the risk of mould growth and other damage related to moisture. As with supply ventilation, filters can be applied to the inflow of air along with both heating and cooling to reach a desired temperature (U.S. Department of Energy, 2019b).

Balanced ventilation systems have a higher cost compared to both the Exhaust and Supply ventilation systems as they require at least two separate ducts and fan systems. However, balanced ventilation systems also allow for energy recovery since they allow more control of airflow. This can help to reduce draught caused by high temperature changes (U.S. Department of Energy, 2019b).

Energy recovery ventilation systems help to minimise energy loss. It accomplishes this with a heat exchanger that uses the exhaust air to either heat or cool down intake air. Additionally, some systems also seek to retain the humidity of the air (U.S. Department of Energy, 2019b). Energy recovery ventilation systems are applicable where the difference in temperature between outdoor and indoor is high, as this is where the energy recovery is at its highest. At lower temperature differences the operation and maintenance cost may exceed the energy recovery gains (U.S. Department of Energy, 2019b).

2.2.3 Ventilation System Control

Whether a ventilation system is balanced, exhaust or supply based, several paradigms exist for controlling the components in it. The simplest ventilation systems are controlled manually with a switch or by a timer, while others use more advanced sources of information to decide when to ventilate. In this section we look at a selection of sensors and information sources that existing technologies use for controlling ventilation systems (aereco, 2019; AerHaus, 2018; Nilan, 2014).

CO_2 sensors Used to detect the concentration of CO_2 in the air, which is used as a proxy for several other pollutants as it is released by burning hydrocarbons and exhaled by people.

Humidity sensors Used to detect the humidity in air, often installed in an exhaust duct. It is used to determine how intense the ventilation system should run to bring down the level of humidity to acceptable levels. Similarly to CO_2 humidity levels can also be used as a proxy for the amount of occupants in a room.

Passive Infrared (PIR) Used to prioritise ventilation in specific areas by detecting movement in them. Movement must be in direct line of sight of the sensor.

Temperature sensors Sensors that measure the temperature can be used for deciding on how much outside air to pull in or how much of the air should be recirculated in order to reach a target temperature. Air temperature can also be controlled by how quickly air is pushed through a heat exchanger in each direction controlling the heat recovery.

Additionally, some ventilation systems use historic data about humidity and CO_2 for predicting the presence of occupants in different rooms at different times. This allows them to ventilate preemptively.

2.2.4 Ventilating in Advance

As shown by Toftum, Wargocki, and Clausen (2011) and Turanjanin et al. (2014), CO_2 levels can increase rapidly in indoor locations with several people present which also affects their ability to concentrate. Conversely, CO_2 levels also decrease rather quickly when people are not present. Only using CO_2 levels as a way to control ventilation might not be the most optimal way, as the effect of multiple people breathing in a room has a delayed effect, especially on CO_2 sensors that are not located right next to where people are. Instead, detecting the presence of people could allow the system to start ventilation in anticipation of the rising CO_2 levels.

It is difficult to say whether the IAQ can be improved more by preemptive ventilation than traditional ventilation as we have not been able to find research on the subject. It is also outside the scope of this project to prove it. However, Wang et al. (2017) shows that detecting the presence of people via a video feed used in conjunction with CO_2 concentrations allows for a lower energy consumption of the system. Based on this, we argue that using information about occupants' location at the very least allows us to keep ventilation to a minimum which can reduce both noise and energy levels (Wang et al., 2017; Mirakhorli and B. Dong, 2016). Multiple methods exist for locating or tracking the occupants in a building, which is discussed section 3.2.1.

In this section we gave a brief overview of methods that exist for improving the IAQ and found that a balanced system often performs the best, especially if designed with recovering energy in mind. These types of systems are also more expensive than their counterparts. Additionally, we discussed which sources of information ventilation systems are often controlled by and found that CO_2, temperature and humidity sensors are common. Additionally, we discussed the possible advantages of using the presence of people as a controlling variable.

2.3 Mobile Services Related to Ventilation Systems

There are multiple ways of defining mobile services. In this section we explore a few and how they can improve a ventilation system.

Mobile systems can relate directly to the use of applications on mobile phones. When focusing on ventilation systems for residential homes, homeowners' mobile phones can be used to create smarter systems by taking advantage of some of the technical opportunities afforded by modern smart phones, such as locating the occupants with technologies like GPS. If a ventilation system knows the location of everyone who regularly spends time in the house, it can use this to schedule ventilation. An example could be ventilation being increased in anticipation of a family member returning home, so they return to a freshly ventilated house but avoid listening to a noisy system while they are present. Additionally, the ventilation system could allow users to access real-time and historical data from the sensors of the system in the same app that collects their location data. Furthermore, it opens up the possibility of letting users control their ventilation, for example by setting a preferred temperature.

A different definition of mobile systems, is systems that are designed to continue working in situations where connectivity is lost or resources are limited. Given limited resources on the low power chips used in small scale residential home ventilation systems, it makes sense to look at the system as a distributed one. The historical data could be saved somewhere with more permanent storage such as a server. Additionally, it makes sense to offload the logic that use sensor data to decide the level of ventilation to the same server. This is especially useful if the ventilation system is made up of several modular units, as it reduces the complexity of the system, if they follow a client-server pattern of uploading their data to a central location and receive commands in return. Such a system can also be designed such that the individual units fall back to simpler calculations using their own sensor data if they lose connection to their central server.

2.4 Cooperating with Advanced Electronics Repair

In order to motivate ourselves to create a functioning and full-stack system with multiple inter operating parts, we choose to cooperate with a local business, AERep. It is a small business owned by David Stien Pedersen, who also has an interest in starting a company related to selling smart ventilation systems. His idea for the system, which we call TeleVent, is based around using smaller modular units with sensors, inlet and outlet fans, that communicate through Wi-Fi, which reduces the amount of construction work and planning that would be needed for retrofitting a house with them.

In the initial plan for the product, several TeleVent units are placed on inside walls to move air between rooms and on outside walls to draw fresh air in and move stale air out. The idea is that the sensor readings from multiple units placed around a house can be used by a centralised controller to decide how all the units should ventilate.

A preliminary hardware platform, which uses an ESP8266 chip was already available before the beginning of the project, and is what we will be using as a base for our development. The platform comes equipped with a CO_2 sensor, two humidity and temperature sensors and two controllable fan outputs.

The plan for the product might change down the line to include multiple types of units with different responsibilities. Furthermore, the units we currently have access to might be updated with different sensing capabilities. Because of this, David expresses a desire to have a system which is able to handle different types of units and different versions of the same unit. Additionally, given changing firmware, it is important for the system to be updated remotely.

2.5 Problem Statement

In the previous sections we define what we mean by IAQ is and discuss some problems and solutions to improving it. Additionally, we find that many homes are so well insulated that the IAQ suffers. We also find that a ventilation system which uses the location of occupants in a home can be more efficient and introduce our cooperation with David from AERep.

Retrofitting homes with ventilation systems tends to be costly no matter the type, but the fewer ducts and modifications that need to be made, the cheaper it can be done, which is central to AERep's product. We propose creating a system made up of several parts: firmware for the hardware platform, a central server and a mobile app. The firmware is a mobile service, due to its resource constraints. Additionally, it must be able to maintain some basic functionality in case it loses the connection to the central server. Putting multiple units inside a normal sized home also makes it possible for the individual units to track the presence of people near them. We can use this information in the process of determining which units should ventilate and which should not. We believe that there is value in creating an IoT system, by allowing the users to monitor the real-time and historical data from their ventilation system on their mobile devices. Additionally, the user should be able to have an impact on how the ventilation is regulated, either by selecting a specific temperature or humidity, or allowing them to tell the system whether they are too cold or too warm. This leads us to the following problem statement.

Problem Statement

How can we design and develop software for an IoT ventilation system that regulates indoor climate using environmental factors and the locations of occupants?

2.5.1 Requirements

The discussion portrayed in the previous sections leads to the following list of requirements for our proposed system, which is made up of three distinct parts: the individual units and the firmware, mobile app and the applications running on the remote server.

Firmware

- Must communicate sensor data to a central server if connected to the internet
- Must be able to regulate IAQ using sensor data while disconnected from the internet
- Must be able to detect occupants in its vicinity
- Must have a feature that allows flashing the microchip remotely
- Should be easy to extend for new units with different capabilities

Server-side

- Must be able to receive data from units
- Must be able to store historical data

- Must feature an API that allows users access to their own historical data
- Must be able to handle different types of TeleVent units
- Must be able to regulate IAQ using sensor data from multiple units in the same house
- Must allow remote firmware updates of the units
- Should allow a preferred temperature to be set

App

- Must be able to give a graphical overview of historical data
- Must be able to support the creation of users
- Must be able to configure units
- Should be able to give users an overview of the units in their house
- Should allow users to set a preferred temperature
- Should be able to run on most phones currently on the market

Chapter 3

Solution

From our newly specified requirements, we describe how we design each part of our system and explain noteworthy elements of the implementation of each. We first give an overview of the system. Afterwards, we explain methods for localising unknown points with methods called lateration. Then we present the hardware we have been provided and our design for the firmware. This is followed by the design of each of our server-side applications. Finally we discuss our choices regarding app development.

3.1 Architecture Overview

Our proposed solution contains multiple components. This section will illustrate the relationship between each component and give a brief explanation of them. The list of the component families involved in the solution is as follows:

App A mobile application that allows for the creation of a home and see the IAQ of their house.

API An API which allows front-end applications like the app, to access the user's information.

Database Collection of documents containing all sensor data, houses, users and TeleVent units.

TeleVent units Uploads sensor data and receives fan control settings using the Message Queuing Telemetry Transport (MQTT) protocol.

Storage service In charge of converting data sent by the TeleVent units and upload it to the database

Controller A controller which adjusts the speed of the fans on each unit based on recently uploaded data.

MQTT Broker A server that sends published messages with a given topic, to all clients that are subscribed to the same topic

This proposed system design can be seen in figure 3.1.

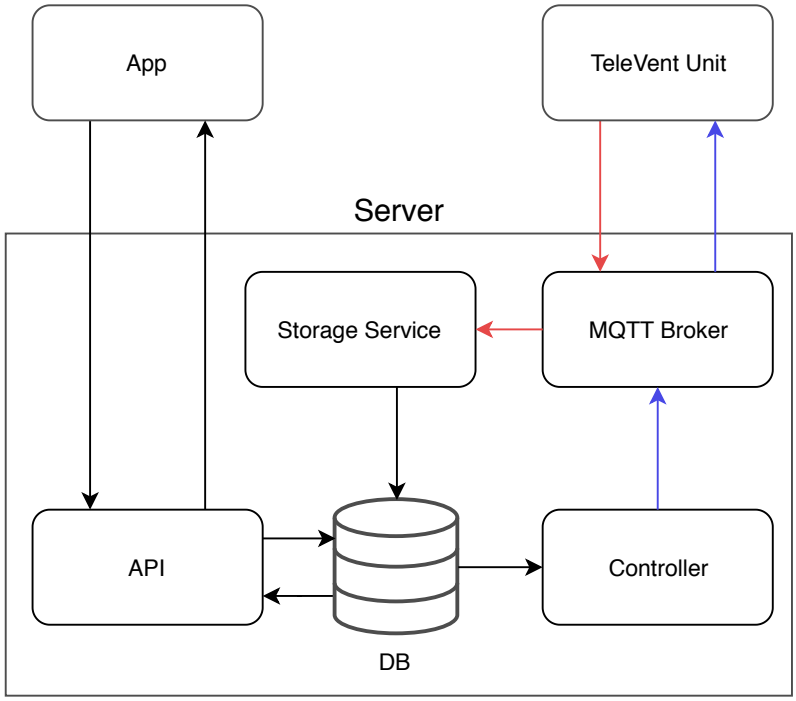

Figure 3.1: System Architecture, the blue arrows show data flow from the controller, and the red arrows shows sensor data being sent to the storage service

3.2 Detecting and Locating Occupants

As per our requirements, we need to be able to detect the presence of occupants in individual rooms such that our controller can use it when deciding how and where to ventilate. In this section we describe possible ways of locating occupants.

Multiple ways to detect occupants in indoor environments exist. In section 2.2.3 we mention that some ventilation systems use PIR sensors and in section 2.2.4 we talk about the results of a ventilation system that uses a video camera.

Passive Infrared Sensors

Using a PIR sensor is a simple way of detecting the presence of occupants in a room (Adafruit, 2019). It is arguably also too simple for our purposes, as it cannot be used to detect the number of occupants. This is because it only reacts to movement, which also means anyone sitting still for too long are undetectable by it. A PIR sensor needs direct line of sight which means that the placement of it is important.

Image Recognition with Cameras

A camera coupled with image recognition software can be used to detect the number of occupants even if they are not moving, which is an advantage over PIR sensors. It does, however, suffer the same problem of requiring line of sight. Additionally, cameras produce a lot of data. Handling the data and running a good image recognition algorithm on the images is generally so demanding that it becomes infeasible for most embedded platforms.

Wi-Fi Signals

Monitoring Wi-Fi signals from occupants mobile devices means that we can detect their presence and count the number of them. Using lateration as described in section 3.2.1 we can combine the results of measurements from multiple TeleVent units to pinpoint which room a Wi-Fi signal originates from. Compared to the two previous methods, the setup is simpler, as we can use a chip on the units to do measurements that, for the most part, are unaffected by line of sight. However, since several Wi-Fi devices such as desktop computers and other IoT devices are stationary and not carried by occupants, filtering out such devices is important. Additionally, we cannot guarantee that every occupant carries a Wi-Fi enabled device or even owns one.

We choose to use the Wi-Fi method for locating occupants, because it simplifies setup and requires fewer components.

3.2.1 Lateration

In this section we discuss two distinct methods, True-range Lateration (TRL) and Pseudo-range Lateration (PRL), for locating a point in an n-dimensional space using $n + 1$ other points with a known location. We refer to the point at an unknown location as the unknown node and the remaining points as reference nodes.

True-range Lateration

In general, if the exact distance from the unknown node to each reference node is known, we are able to determine the coordinates of the unknown node in n dimensions if we have access to $n + 1$ reference nodes. This is accomplished by creating an n-dimensional sphere around each reference node with a radius equal to the distance measured from that node to the unknown node. The point at which these spheres intersect is the location of the unknown node (Dargie and Poellabauer, 2011, p. 252-256). The coordinates of the point can be calculated by solving a system of equations, but in the real world the measured distances are likely to be inaccurate to a degree that means no intersection exists. As a consequence, there is no solution to the equations either. Instead, we introduce the following method to reason about the location of the unknown node.

In this example, the dimensionality is two and we have information from three reference nodes. Given two intersecting circles it is possible to calculate the point(s) at which they intersect (Bourke, 1997). As shown in figure 3.2 these points lie on the line, q that is perpendicular to $\overline{P_1P_2}$ and goes through the point Q which is located a units from P_1 and b units from P_2. Even when no intersections exist, the point Q can still be found as a and b are only dependant on the distance, $d = |P_2 - P_1|$ and the radii of the circles centred in P_1 and P_2. Intuitively a and b give an indication of the middle point between the two circles with respect to their differing radii.

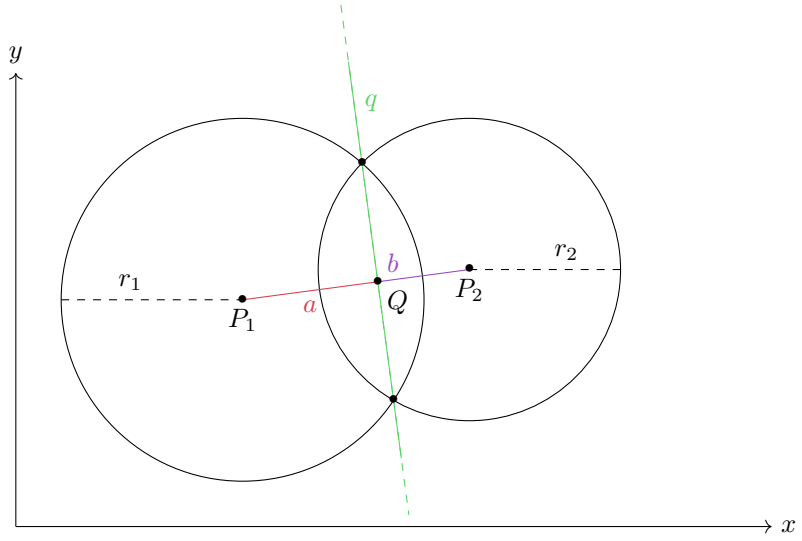

Figure 3.2: Finding intersections between two circles

a can be calculated as $a = \frac{d}{2} + \frac{r_1^2 - r_2^2}{2d}$ and Q_1 which lies on the line segment $\overline{P_1P_2}$ can then be found as $Q_1 = P_1 + \frac{a(P_2 - P_1)}{d}$. The line, q_1 perpendicular to $\overline{P_1P_2}$ going through Q_1 can then be found.

Repeating this process with a third circle centred in P_3 instead of the one centred in P_2, the intercept between the two lines q_1 and q_2 is the point we are looking for. As long as the point is located roughly within the triangle encompassed by the reference nodes, this method allows us to find its location. This can be seen in figure 3.3 with figure 3.4 showing that this still works, even if there is no exact intersection between the circles. Problems arise if any of the lines involved are perpendicular to the x-axis, as this requires the slope term of the equation describing the line to be infinite. This can be avoided by positioning the reference nodes in such a way that P_1 does not share its x-coordinate with P_2 and its y-coordinate with P_3.

Accurately measuring the distance between the unknown node and the reference nodes is challenging in the real world. One method of doing so is by having a synchronised clocks between the unknown node and the reference nodes. However, the velocity of radio waves means that the precision required in the clocks is extreme, if determining distances in the order of meters. Additionally, this puts requirements on the unknown nodes.

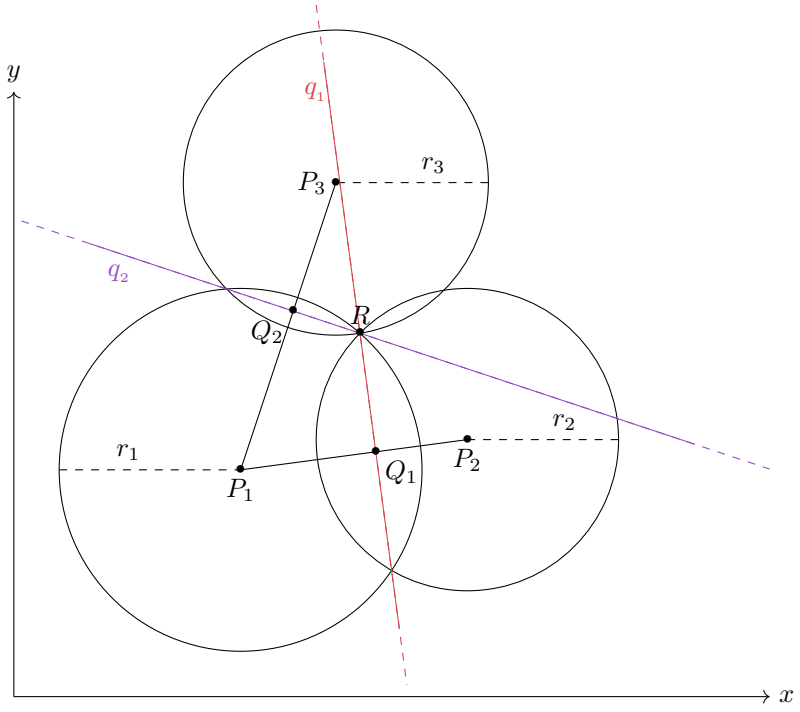

Figure 3.3: Finding intersection between three circles

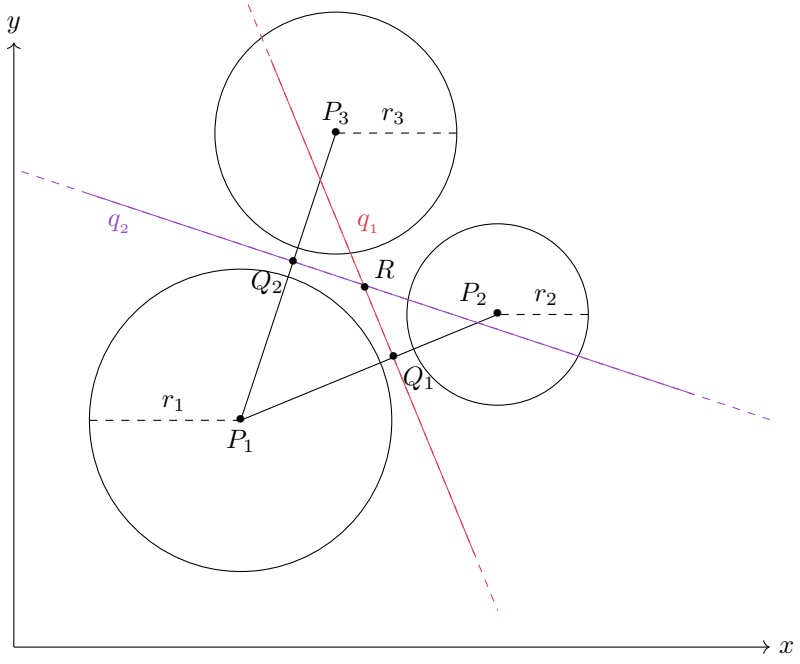

Figure 3.4: 2D lateration with slight errors in distance measurements

Pseudo-range Lateration

Maintaining a synchronised clock between all nodes in a system is challenging, bordering on impossible if the unknown nodes are devices of different make and model. A solution to this, is to only maintain a synchronised clock on the reference nodes and use the difference in time between the arrival of a message from two different reference nodes, the Time Difference of Arrival (TDOA) to determine the location of the unknown node that received transmissions from multiple different reference nodes. Each transmission from a reference node, P_1 can be paired with the transmission from any other reference node, P_2 to determine the difference in distance to the to reference nodes. For a pair of transmissions with difference in distance, $\Delta dist$, there are an infinite number of points for which it is true that the point is x units from P_1 and $x + \Delta dist$ units from P_2. This can be visualised as a hyperbola with its focal point located at the reference node that is closest to the unknown node as seen in figure 3.5.

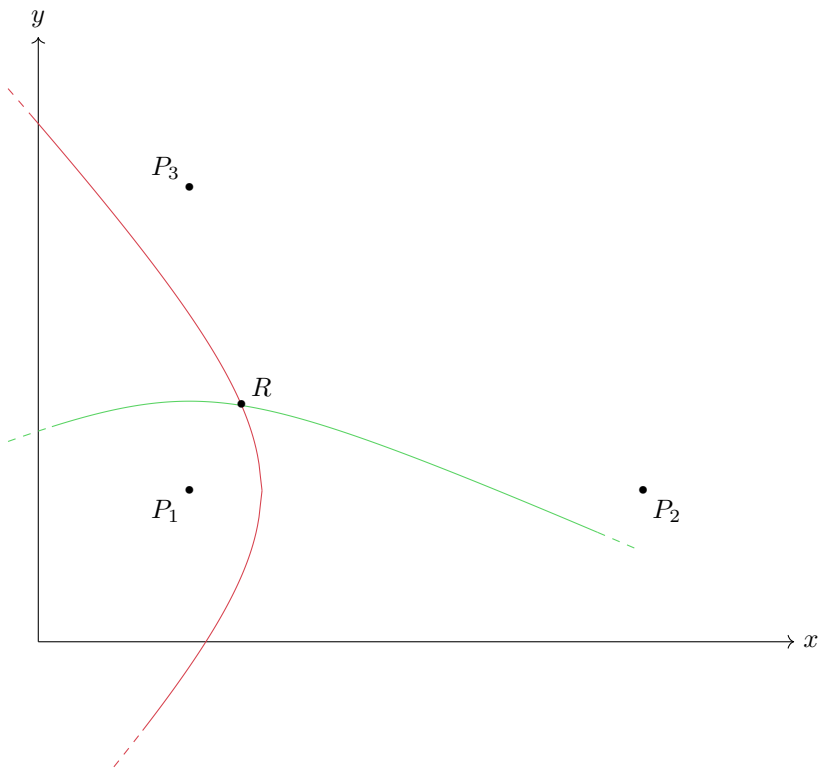

Figure 3.5: Two hyperbolas drawn from TDOA. The red visualises the TDOA between transmissions received from reference nodes P_1 and P_2. The green uses P_1 and P_3

3.2.2 Using Lateration to Locate Mobile Devices

In this section we discuss how the methods TRL and PRL can be used in our system to determine the location of occupants within a home. We use TeleVent units as reference nodes and occupant's devices as unknown nodes.

The degree of accuracy required for synchronised clocks between nodes makes the use of timing radio signals entirely unfeasible for use in indoor location. An alternative is generating high frequency sound waves whenever a transmission is sent. As the speed of sound is significantly lower than

that of light, this gives room for much more for error in the clocks, but with the downside of requiring additional hardware and difficulty penetrating walls. However, the precision required in the synchronisation is still a problem in our situation, as we are hesitant to make guarantees about the up-time of the ventilation units, which undergo firmware updates and lose internet connectivity.

Additionally, we ascribe significant value to being able to "passively" locate devices without requiring them to run specific software as this means we are unable to detect anyone without our app on their phone. This makes PRL of little use to us and leaves us with TRL. That means we need a way of accurately measuring distance between our units and mobile devices that does not rely on synchronised clocks.

The most readily available method is measuring the Received Signal Strength (RSS) of Wi-Fi transmissions from mobile devices as the signals are picked up by our units. Given a sufficiently precise algorithm for converting Received Signal Strength Indication (RSSI) into distance, finding the location of the Wi-Fi device in two dimensions requires that at least three of our units pick up the signal.

RSSI to Distance

According to Q. Dong and Dargie (2012) the inverse square law coupled with real world factors such as reflection on surfaces, refraction from passing through obstacles, etc. the RSSI can be expressed as a function of distance in meters, d and RSSI at 1 meter, p:

$$\text{RSSI} = -20 * log(d) + p$$

Solving for distance leads to

$$d = 10^{(p-\text{RSSI})/20} \tag{3.1}$$

We conduct a simple experiment to empirically test the accuracy at which equation 3.1 can convert RSSI to a distance in meters by using three prototypes running identical firmware. They are placed approximately 20 cm apart at the end of a 12 meter long table with three mobile phones spaced equally apart being placed at varying distances from the prototypes as shown in figure 3.6.

Each prototype is capable of measuring the RSSI of the phone directly in front of it, by sniffing Wi-Fi packages. Preliminary testing shows that the RSSI values fluctuate quite a bit, so all values we present have been smoothed using a Gaussian filter with $\sigma = 2$ and a kernel length of 11.

We measure the RSSI of each phone at different distances and plot these in a graph next to the results of equation 3.1 as seen in figure 3.7.

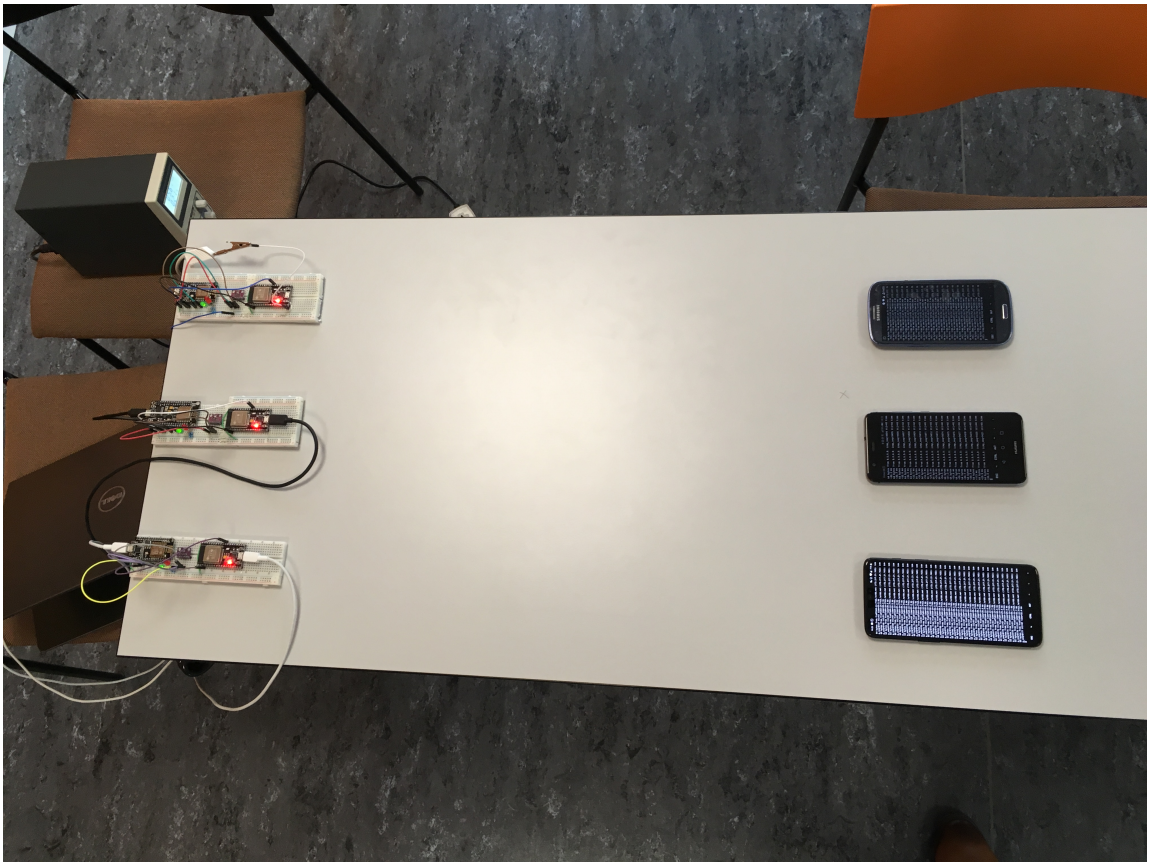

Figure 3.6: The setup used for testing RSSI to distance accuracy. Each prototype unit only picks up the RSSI from the phone directly in front of it

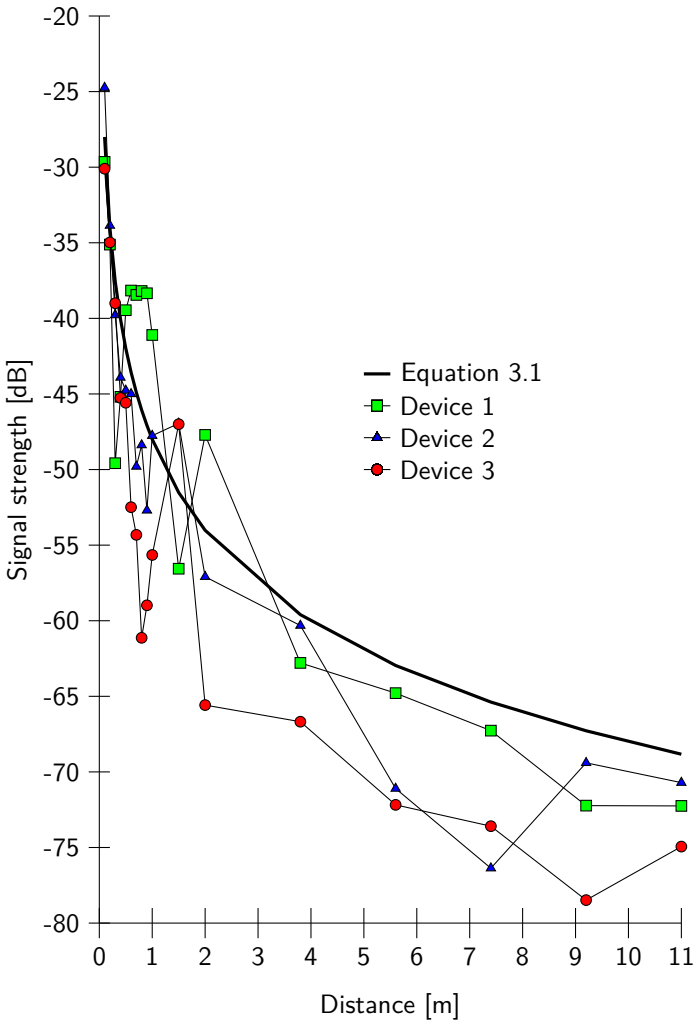

Figure 3.7: The RSSI measurements of three different mobile devices at increasing distances. Each value is an average of 10 measurements made at each distance

The results seem to show that there is a significant difference between different make and models of mobile devices in terms of signal strength. Additionally, it shows that while the accuracy of equation 3.1 is debatable, the trend is the same. Looking at the measurements, it is possible that there are problems with our test setup. For example, for each phone the measurements at 1.5 meters vary significantly from the surrounding ones, which is an indication that something affected the signal strength at this location. The table on which we performed the experiments might have been to blame, as large metal parts cover the underside of it. In order to achieve proper results, a better test setup with less interference would have to be used, but ultimately it is of little importance, as a model created this way will only be accurate in a similar setting.

Nevertheless, we test how accurately we can detect location within a 7 by 8 meter room by placing the prototypes around the perimeter of it and moving devices around in the room. Figure 3.8 shows the approximated location of the phone as a cross, and its actual location as a circle. The radii of the large circles represent the distances they measured to the device.

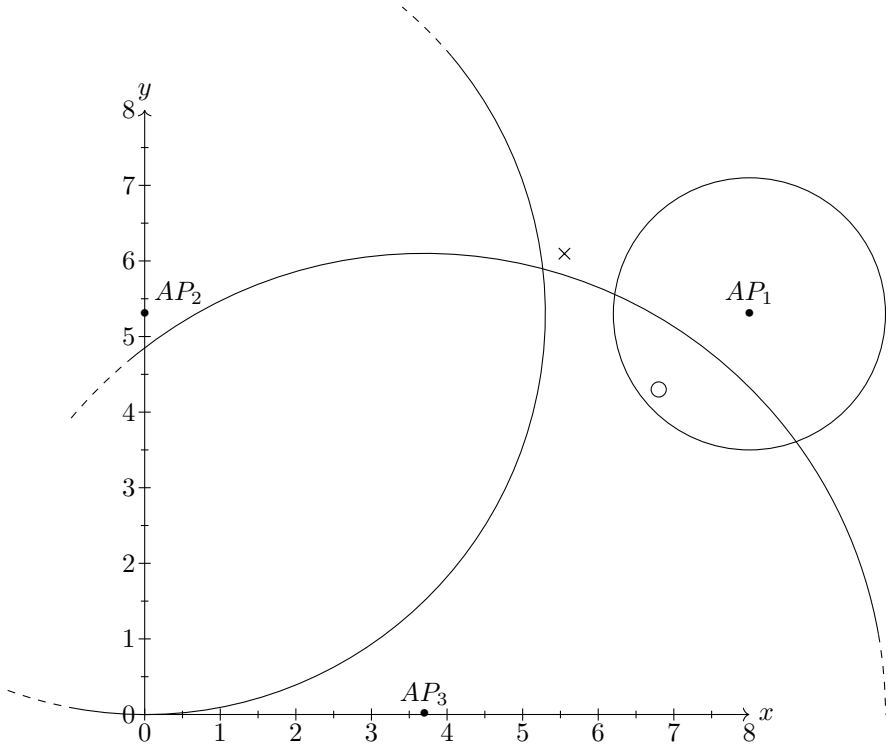

Figure 3.8: Experiment to test the accuracy of TRL using equation 3.1 to convert RSSI to distance in meters

The result is promising and should allow us to determine whether a person is inside a given room, especially if the TeleVent units are positioned on walls around the house. Our experiment does not explore the effects of walls or other massive objects on RSSI.

In this section we find that using Wi-Fi signals to detect the location and number of occupants is more feasible than using PIR sensors or a camera with image recognition. The experiments we performed show us that different mobile devices have quite different signal strength, but with TeleVent units located on several walls in a house, locating occupants is still feasible.

3.3 Firmware

In this section we propose our design for the firmware to be used on the hardware platform provided by AERep as described in section 2.4. We give an overview of the microcontroller and peripherals we use and decide on how to set up communication between the TeleVent units and our server-side applications.

The hardware platform, is centred around an ESP8266, a 80-160MHz, 32 bit microcontroller with Wi-Fi capabilities. Connected to the ESP8266, through the Inter-Integrated Circuit (I^2C) bus, are two sensors capable of measuring temperature and humidity, HDC1010s, and a CO_2 sensor, a CCS811. Additionally, a digital potentiometer, an MCP4451, is present and can be used to control the voltage of two outputs independently of each other. These outputs are used to control the speed of the fans, one for inlet fans and one for outlet fans. Up to four fans can be connected to

the platform. To read the speed of each fan, the fans have a tach[1] signal wire which have a direct connection to pins on the ESP8266.

If attached to an outer wall, the platform is meant to be oriented in such a way that the CO_2 sensor measures the indoor air. One set of fans is used to draw air in, the inlet fans, while the other is used to blow inside air out, the outlet fans. The temperature and humidity sensors are located so that one measures the inlet air and the other the outlet air.

There are two LEDs, one green and one orange, located on the platform as well. These are useful for telling which state the firmware is currently in, a helpful feature when developing and debugging the device. Due to many of the General Purpose Input Output (GPIO) pins on the ESP8266 serving multiple purposes making them difficult to work with without losing critical functionality, the LEDs are connected through a GPIO expander, a PCA9570. Figure 3.9 shows a diagram of the relevant sensors and chips we use on the platform.

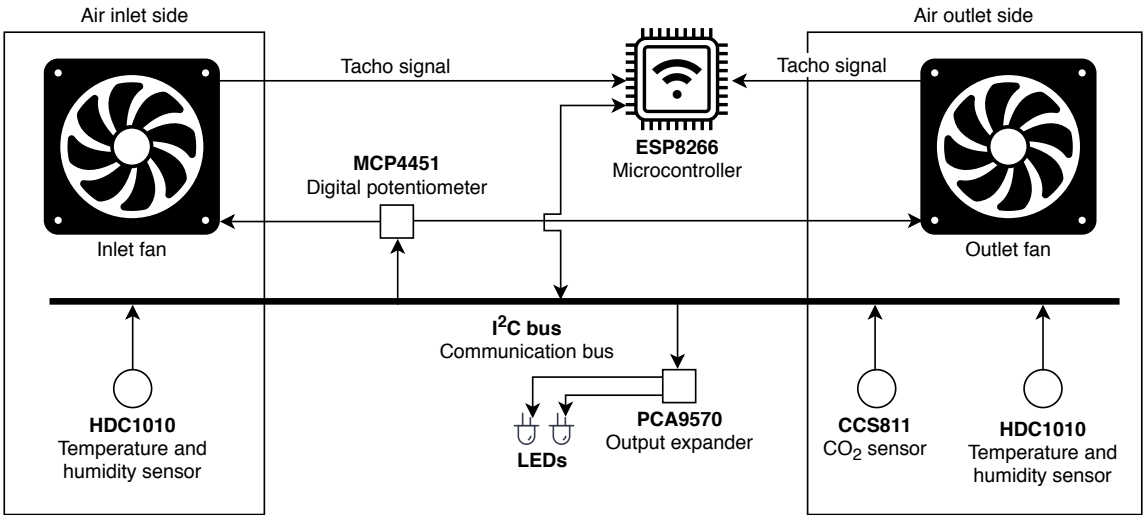

Figure 3.9: Overview of hardware components of the TeleVent units

As this hardware platform is subject to change in the future, it is important for us to develop our firmware in a way that encapsulates the hardware details, such that developing firmware for new platforms is as easy as possible. Specifying an interface that the firmware for all platforms can adhere to, is a way to do so. This interface must contain methods for reading data and sending it, while also supporting different behaviours depending on whether or not an internet connection is possible. This is because we want to be able to control the fans of a disconnected unit manually based on its own sensors' readings.

In order to meet the requirements as defined in section 2.5.1, the units must be able to read and send their sensor data at specified intervals and receive fan control settings. Given that the microchip has Wi-Fi capabilities, it makes sense to communicate the sensor readings and fan control through the internet using a network in the home in which the unit is located. This means that the units must know the SSID and password of the network before being able to connect to the internet. In order to send this information to the units, we utilise the fact that they are capable of acting as a Wi-Fi access point. Making a Wi-Fi-enabled device change its Wi-Fi network to a unit's, send some data and log back on the original network is a process that can be made automatic. Once the unit has access to the internet, it is able to communicate with a server. We can do this through

[1] A tach signal is a square wave signal that generally pulses every revolution of a motor

HTTP but since the units upload quite frequently, it makes sense to use a protocol with smaller headers and especially one that keeps a connection open to avoid frequent TCP handshakes. It is possible to create our own protocol, but using an existing and tested one is significantly easier and safer. We choose the protocol MQTT, which is often used to allow IoT devices to communicate with a low memory and bandwidth footprint. We give a more detailed description of MQTT in section 3.4.2.

Another requirement for the firmware is that it is possible for us to update the firmware (known as flashing) remotely. This can be a cost-saving measure compared to sending technicians out to houses in case a critical bug is found in the firmware.

The open-source project Homie fits the requirements we have very well (ROGER, 2019). It is a framework for IoT devices using ESP8266 chips that uses MQTT to send and receive data. It also uses the same configuration method that we propose. As Homie is open-source, it is possible for us to modify it, if we require additional functionality.

Based on the requirements specified in section 2.5.1 we decided on the use of MQTT and Homie for the firmware used on our hardware platform. Having talked about our ideas for the firmware, we next explain how we implement these ideas.

3.3.1 Implementation

In this section we describe how we implement the firmware for the ESP8266 chips used on the individual units. We discuss our use of the library Homie, how we handle running the same firmware on different types of units, our general design philosophy and the details of some of the peripheral devices we used, such as fans and sensors.

Implementation on ESP8266

Multiple software development kits exist for programming the ESP8266 we use, which allows us to choose between several programming languages, such as Python through MicroPython, Lua through NodeMCU or JavaScript through Espruino. It is common to use a language that provides direct access to memory management for developing on embedded platforms. This makes C an obvious candidate, but some object-oriented constructs such as classes and interfaces are beneficial when it comes to preparing support for future units and hardware revisions. We choose to use C++ through the Arduino framework and generally attempt to keep our firmware organised in classes and avoid global program state where possible. There are situations in which this breaks down though, for example, when we need to use interrupts, as these require us to use global variables rather than keep everything encapsulated in classes.

We use caution whenever we allocate memory, as we are limited to 80 KiB of total memory. With the sensors present on the TeleVent units available to us, we are using less than 50 bytes of memory per sensor reading. We are storing several of these at a time between each publish cycle however, so they do eventually add up. The number of readings between publishing is a set constant so we have exact control of our memory footprint. The RSSI and MAC addresses of nearby devices take up significantly more space, as it stored as JSON array, as described in section 3.3.1. If we allow our device sniffing sensor to store up to 20 devices between sending them, the array is about 500 bytes in size. Using a packet a sniffing tool, we look at how much traffic a TeleVent unit sends over the network. We find that a unit that does not look for occupants sends approximately 820 bytes to the server each time it publishes. By far the most of this, is TCP overhead and the topic strings used by MQTT. If we factor in the occupant data, we reach around 1450 bytes. If we publish to the server every fifth second, this results in around 25 MB of data per unit per day. We discuss

ways of lowering this in section 5.5.1.

Homie

As described in section 3.3 we use the library Homie, designed for IoT applications, as it fulfils most of the requirements we have related to firmware. There are some exceptions to these, which is why we modified Homie to better fit our needs.

Homie does not allow full customisation of the topic strings that each unit uses with MQTT. In order to properly manage multiple units in multiple houses, it makes sense to include the name or ID of a house in the topic string. To follow the hierarchical nature of a URI, the house needs to come before the unit name or ID. We solve this problem by modifying the required fields in the configuration files used by Homie to include a house ID and make sure that this is prepended to all topic strings used by MQTT.

A different problem with Homie, is the setup of new units, which requires logging on to a configuration pending units network, sending a configuration file and logging back on to an internet-connected network. While the app we develop can make this process easier, by automatically changing networks, it is helpful if the units themselves can configure each other. We make this possible by editing Homie, such that configured online units will scan for configuration pending units network every few minutes. If it detects a unit, it logs on to it and sends a configuration file similar to its own. To have better control over this process, we include an additional field into the configuration files, which describes whether a unit is allowed to configure other units. This field is always set to false when one unit configures another. A different way to handle this with more control could be for any unit that discovers a unit awaiting configuration to use MQTT to tell the server about it. The choice of whether to configure it, can then be delegated to the app through the API.

Different unit Types

As mentioned in section 3.3 multiple types of units may exist the future and we want our firmware to automatically detect the type of unit it is running on and act accordingly. In practice, we do so by looking at which units are available on the platform when we run the firmware. Ideally each unit type and hardware revision would come with a a voltage reference by way of two resistors making a voltage divider, which would be read by the ESP8266's analog to digital converter to determine the revision number. As this is not available on our hardware we scan the I^2C bus instead, to see which addresses have sensors on them. We concatenate these addresses and use the resulting string to tell which unit we are dealing with. Our units all follow the same interface which contain methods for taking measurements, sending them over MQTT, signalling using LEDs and running specific code in case the unit loses connection. Having Homie interact with the units through this interface alone, allows us to reuse most of our control logic.

Peripherals

Our TeleVent units are connected to multiple different types of peripherals such as sensors, fans and LEDs. For some of the sensors, libraries already exist for communicating with them and we use these where applicable. This is the case for both the CO_2 sensor, a CCS811, and the temperature and humidity sensor, a HDC1010. Other units such as the four-bit output expander, PCA9570, used to control the LEDs on a ventilation unit does not come with their own library, so we create our own following the specifications of the chips. These generally consist of methods that wrap some logic to flip a few bits in a variable containing the state of the chip and sending the result through the I^2C bus that most of our sensors use to communicate.

Controlling Fans

Our fans are controlled by a digital potentiometer, MCP4451, given a percentage value between 0% and 100% to impact the speed of the fan. We find that the relationship between this value and the actual speed of the fans is not linear. For example, setting the digital potentiometer to 50% does not equate to the fans stabilising at halfway between their minimum and maximum RPM To make it easier for our controller to reason about the fan speeds it dictates, we solve this problem in the firmware. We do this by continually adjusting the potentiometer value based on the current RPM to bring it close to the set percentage value.

By setting up an interrupt handler to increment a counter on the rising edge of a signal sent from the fans twice every full revolution we can count the revolutions of a fan. Using this counter along with the elapsed time since the counter was reset we can calculate the RPM.

Detecting Occupants

In order to detect the presence of occupants by the activity of their mobile phones we need to detect the wireless traffic from them. This is done by setting the Wi-Fi module on the ESP8266 chips to promiscuous mode, where all packets are processed rather than only the ones intended for the unit itself. Changing the mode of the Wi-Fi module means that we are disconnected from the network. This makes it impossible to send the data we detect back to the server as we obtain it. We could detect the packages on an interval, such that we go back and forth between Wi-Fi modes, but it often takes a while for the units to connect to the server. Instead we create the entire sniffing routine on a separate chip and communicate it back to the main chip. This essentially means that we can treat the sniffer chip as a sensor like any other in our system. We send the data as a JavaScript Object Notation (JSON) array containing the MAC address and RSSI of multiple different devices. Compared to the amount of data from the other sensors in the units this JSON array is very large. Therefore, we are interested in sorting out as many irrelevant MAC addresses as possible before the data is sent. As per the IEEE 802.11 protocol, each packet header contains information about the type of communication it is carrying, where it is heading and where it is coming from. Using this information means that we are able to filter out packets sent from access points, as these tell us nothing about the location of occupants. Further processing has to be done on the server-side to filter out devices that rarely move such as desktop computers and other IoT devices.

Handling Loss of Connection

In case a unit loses connection to the server or the Wi-Fi, it needs to be able to re-establish connection. Homie comes with functionality for automatically reconnecting but from experience we find that the units are sometimes unable to re-establish a connection. When this is the case, rebooting the units often allows them to establish a connection as soon as they are back up and running. When deployed, it is not feasible for users to restart the units if they stop working, so we implement software resets as a reaction to the unit being disconnected from either Wi-Fi or the MQTT server for a set amount of time. We already have code in place that runs when a unit is unable to communicate with the server, as we use this to control the ventilation based on local sensor readings.

Rarely, we also experience erroneous readings from some of the sensors, perhaps because the I^2C bus is put in a faulty state. If we see readings are above a certain threshold, such as temperatures above 100 řC we also perform a software reset.

3.4 Server-side Applications

As described in section 3.1, the server consists of several different applications with different responsibilities. In this section we describe the design choices behind each of them, starting with the database which the rest of the server applications depend on. We follow this with a description of the MQTT protocol, and why using it for the communication between the TeleVent units and the server makes sense for the project. We then describe the storage service, the purpose of which is to update the database with information from TeleVent units. After that we design an API which describes the interactions between users and the system. Finally the controller is described. It updates the units with new fan speeds based on recent data.

3.4.1 Database Design

The choice of database is rooted in ease of prototyping rather than in performance. In this section we describe why that is and how we structure our data in the database.

We have chosen to use MongoDB for our database. There are several reasons why MongoDB is a good fit for our project, but the main reason is that both the initial setup process and any subsequent changes to the design are made significantly easier by avoiding a schema. If we ever decide on sending sensor data in a completely different format or decide on saving it differently, we can write code that handles two different data formats simultaneous. Additionally, MongoDB has support for geospatial queries, which might become relevant in a future update that handles locations of users and events in a different way (Kay Kim and Andrew Aldridge, 2018).

Our database design can be seen in figure 3.10, where each box is a collection.

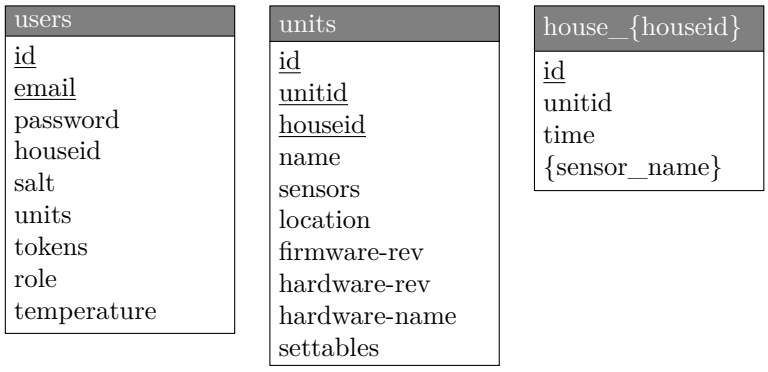

Figure 3.10: Database design. Fields in plural are arrays, underlined fields are indexed and brackets mean variables

The *users* and *units* collections contain information about each user and unit respectively. A user document is created when they sign up through the API, with the *units* array being added to when new unit documents with the same *houseid* are created.

Units and their fields are created and updated dynamically, as data arrives through MQTT. The fields *sensors* and *settables* contain arrays with the names of sensors and controllable devices respectively. An example of controllable device is a fan, which can be controlled by publishing to its topic through MQTT.

The final part of the design is multiple collections, one for each house, that contain all sensor reading from that house. A house collection is created when a unit in a house first publishes data. Each document in these collections contains the reading, the unit that made it and the time at which

it was made. Additionally, we take advantage of the schema-less database here, by dynamically naming a field the name of the sensor that took the reading. This makes it simpler to support new types of sensors in the future. If we are interested in the temperature in a given house, we can extract the data from the relevant collection where the sensor name fits.

3.4.2 MQTT

MQTT is a TCP protocol that uses a publisher-subscriber pattern and works on top of TCP-IP (Piper, 2013). MQTT was designed and developed by IBM in 1990 and was intended as a means of linking sensors in an oil pipeline with the only communication available being the use of satellites, requiring the bandwidth to be as low as possible (Yuan, 2017). This makes the MQTT protocol very useful for IoT devices. The protocol requires a server called a broker, which ensures that packets are sent from the publishers to all subscribers.

Each MQTT packet consists of a message and a topic that subscribers can subscribe to. The topic is a string which can be anything, but often used to describe the message being sent. If a device publishes its online status, then a topic could be "hardware-id/status/online". Messages for this topic, could be "true" and "false". Any subscriber can then subscribe to this topic, and each time any other client publishes something with this topic, then the subscriber will receive the message.

A client can also specify a last will, which will be published by the broker should the client disconnect unexpectedly. This comes in handy for a device if it needs to let people know its current online status. Such a device can tell the broker that its last will, is for the broker to publish "false" to the "hardware-id/status/online" topic.

As our system contains several ventilation system units in each house, we want to lower the bandwidth requirement as much as possible. Since MQTT has been designed with low bandwidth in mind, and it is already used by many IoT devices, we choose to use this protocol for communication between the devices and the server.

The topics that we use in our system are listed in table 3.1.

Topic	**Message and Outcome**
prefix/{unit_type}/{sensor}	Sensor output which gets uploaded to DB
prefix/{unit}/{fan_name}/set	Percentage, which adjusts unit's fan speed
prefix/$fw/version	Update firmware version, or upload new unit
prefix/$hw/version	Update hardware version string
prefix/{unit_type}/$properties	Update hardware name, sensors and settables
prefix/$online	Update online status. Also unit's last will
prefix/$implementation/ota/firmware/{md5}	Units verify {md5} and flashes itself

Table 3.1: Server-side client supported MQTT topics. The topic prefix is the following: `devices/{house_id}/{unit_id}`. $ are used for meta information about the unit and its firmware

3.4.3 Language Choice for Server-side Applications

We choose a dynamically typed language, for the same reason that we choose a schema-less database system, as described in section 3.4.1.

A statically typed language would mean we had to define a schema for the otherwise schema-less

database, which causes us to lose some of the database's capabilities. To use the full power of the database, we would also be required to use a driver framework, which require us to write the queries ourselves as text, meaning the static type checks will not benefit us much.

This is why we decide to write the back-end in the dynamic-typed language Python. Python has a MongoDB driver for the database. It also has the necessary frameworks to build an API for the front-end, and communicate with the TeleVent units we have through using the MQTT protocol.

3.4.4 Storage Service

The Storage Service is the application which handles incoming data from the TeleVent units and writes it to the database. It is necessary because MongoDB cannot subscribe to the topics the units publish to. Because we support different types of sensors, the Storage Service must be able to infer the type of each sensor value.

It must connect to the MQTT broker and subscribe to all relevant messages, see table 3.1, from the TeleVent units and whenever a message is received, create or update the relevant documents in the database. The storage service, when updating values in the database, should enforce consistency by, for example, ensuring a unit does not appear in two different houses.

3.4.5 API

The API is what allows other applications to access information and modify information on the server. Flask is a library that allows us to implement an API in Python. When an endpoint is accessed, flask calls the method that handles the request. The data extracted from the database is BSON formatted and can be represented as a dictionary, which is automatically handled by the MongoDB database driver. In order to provide an API, we need endpoints that allow users to sign up, add new units and update existing units such as changing the name of a TeleVent unit. The user also needs to be able to retrieve the data, uploaded by the units.

Administrators may need functionality that requires them to see and modify other units, inserting units and add these to any user they wish. They may also need to delete units if these units have failed. Other than this functionality, administrators also have the privilege to flash units remotely. We believe the endpoints listed in table 3.2 provide the necessary functionality.

Type	**Role**	**Address**	**Purpose**
POST	none	/login	Logs into user
POST	none	/users	Register new user and home
PATCH	user	/users	Change user email and password
DELETE	user	/users	Deletes a user and cleans up
POST	user	/logout	Logs user out
POST	user	/users/reset	Ask for resetting user password
POST	user	/users/reset/{reset_token}	Change user's password to new password
PUT	user	/house/temperature	Set target temperature on house
GET	user	/house/temperature	Get target temperature of house
GET	user	/units	Get all units in user's home
POST	user	/units	Adds a new unit to this user
GET	user	/units/{unit_id}	Gets information of unit
PATCH	user	/units/{unit_id}	Updates information of a unit
DELETE	user	/units/{unit_id}	Removes a unit from user
GET	user	/units/{unit_id}/sensors/{sensors}	Get data from {sensor}
GET	admin	/admin/units	Get all units
POST	admin	/admin/units	Adds new unit to a user
GET	admin	/admin/houses	Get all house IDs
PATCH	admin	/admin/users/{user_email}	Change role of {user_email}
GET	admin	/admin/units/{unit_id}	Gets information of unit
PATCH	admin	/admin/units/{unit_id}	Update this unit
DELETE	admin	/admin/units/{unit_id}	Removes this unit
POST	admin	/admin/units/flash	Update firmware of all units
POST	admin	/admin/houses/{house_id}/units/flash	Update firmware of all units in this house
POST	admin	/admin/units/{unit_id}/flash	Update firmware of this unit

Table 3.2: Overview of API endpoints

Some of these endpoints, such as `units/{unit_id}` only consider units owned the user making the request, not every single unit in the system, in order to prevent users from potentially accessing information they do not have access to. The complete specification can be seen in appendix B.

3.4.6 Controller

The last application needed is a controller that periodically fetches new data for each house from the database. The controller must process the fetched data and publish desired fan speeds to the TeleVent units. The more frequent that the units can publish data, and the more frequent the controller can calculate new fan settings for each unit, the more responsive the system will be. The units will also upload RSSI of nearby network units, so it can use lateration to determine where the residents are located.

Knowing the location of occupants relative to the units and having the values of the other sensors during the time we detected the presence of the occupants, we can attempt to decide the fan speeds

of all units in the house to improve the IAQ and sound comfort of the users.

For each sensor of a TeleVent unit, there is a range of values which we find acceptable and outside that range, the values become increasingly unacceptable. Each unit might also have access to outside air for which any of the properties like temperature or humidity etc. might be in the direction of our acceptable range, with respect to the indoor value. In the case where it is too hot inside, and the outside air is colder than the indoor air, we can tell the unit to pull in air from the outside. On the other hand, if the outdoor air is not in the direction of better values, but another room in the house is, we can tell the unit to push air out of the house to create a lower pressure in the room which should draw air in from adjacent rooms.

For each of the sensor types, we define a piecewise linear function, an example of which for relative humidity could be equation 3.2. The range of the functions are between -1 and 1. The function take a sensor value and the resulting numbers sign indicates the direction in which the function value should change to be more acceptable and the value is some indication of how "serious" it is that the sensor value is not in the acceptable range.

$$f_h(x) = \begin{cases} 1 & \text{if } x \leq 0 \\ -x + 1 & \text{if } 0 < x \leq 0.2 \\ -16x + 4 & \text{if } 0.2 < x \leq 0.25 \\ 0 & \text{if } 0.25 < x \leq 0.6 \\ -16x + 9.6 & \text{if } 0.6 < x \leq 0.65 \\ -x - 0.15 & \text{if } 0.65 < x \leq 0.85 \\ -1 & \text{if } 0.85 < x \end{cases} \tag{3.2}$$

Equation 3.2: An example of a piecewise function for humidity

for ease of entry the functions can be described as a list of endpoints of each of the function segments.

[(0.00, 1.00), (0.20, 0.80), (0.25, 0.00), (0.60, 0.00), (0.65, -0.80), (0.85, -1.00), (1.00, -1.00)]

This is just an example of a function. Deciding on good functions will take time and one set of functions might not suit all houses or all types of rooms. For example, in a bathroom you might expect the humidity and temperature to rise above the ordinary recommended levels for short periods of time while the room is in use. So instead of running the bathroom fan at 100% while the shower is in use, we could ventilate less under the assumption that levels will adjust as soon as the shower turns off.

For each of the piecewise functions we decide on a prioritisation constant, see Equation 3.3, which will indicate how strongly the difference between indoor and outdoor values will incentivise the controller to use the outdoor air as a source. This prioritisation constant also ends up being a weight which decides how to prioritise between the different sensor types. Say the CO_2 level outside is better for the IAQ but the outside temperature is not. These prioritisation constants will weight those options against another and make a decision on whether to pull in air or not. The prioritisation constant will also serve to dampen the effort of the fan as the difference between indoor and outdoor values become negligible. If the temperature level inside is bad, and the outside temperature is slightly better, the controller will not force the houses to use lots of energy, to change the indoor temperature to the slightly better outdoor temperature. Finding good difference values will also require some tuning.

$$F = f_c(c) \cdot \frac{c_{\text{outdoor}} - c}{\Delta c} + f_t(t) \cdot \frac{t_{\text{outdoor}} - t}{\Delta t} + f_h(h_{\text{rel}}) \cdot \frac{h_{\text{abs outdoor}} - h_{\text{abs}}}{\Delta h_{\text{abs}}} \tag{3.3}$$

Equation 3.3: c is the measured CO_2 value, t is temperature and h is humidity. The Δ values are the prioritisation constants

Equation 3.3 gives us a value F. If F is positive, that means that pulling in outside air is favourable for the IAQ and F will be the percentage speed indication given to the fans in the room. If F is negative on the other hand, then we are not interested in the outdoor air but will instead look at the air in other rooms of the house. To pull air from the other rooms we push the air in the current room outside, the fan speed for the fan pushing air out, is calculated in a similar manner to F. The outdoor values are replaced by the average value of all the other rooms in the house. Here again we will set the fan speed based on how large the resulting force value is.

We want to keep good IAQ while not emitting excessive amounts of noise, which can be distracting to people in the same room. Firstly we add a cap on the fan speed when any non-zero number of people are in the room. This cap is set per device by the user, as a higher noise level may be acceptable in some environments or scenarios, but not others. It might not matter as much that the fan is being noisy while people are taking an already noisy shower, compared to someone working quietly in the office. To anticipate a change of values like CO_2, which rise as a consequence of peoples presence in a room, we adjust the indoor value as a function of the amount of people in the room. For example, we add a fixed amount of CO_2, per person in a room, to the measured value. These functions will also require testing and adjusting to give good results.

In this section we described our method for deciding how to ventilate in a home based on sensor data from around a building. We also discuss how the information about the number of occupants and their location in the building can be used to augment this.

3.5 Choice of App Development Platform

From section 2.5.1 we have the requirements for the app. Any choice of platform must be able to fulfil these requirements.

One of the requirements is providing support for the majority of mobile phones currently in use. This can be achieved by covering iOS and Android (*Mobile Operating System Market Share Worldwide* 2019). We choose not to explicitly support iOS as there are entry barriers to develop for iOS, mainly the yearly developer licence fees. We make the decision to develop the app solely for Android, because it is:

- Free to develop for
- We own several Android devices between us as a group
- The Android platform has the highest market share of mobile operating systems (*Mobile Operating System Market Share Worldwide* 2019)

We still have intentions of providing support for iOS in future development. As such our choice of platform has to support the requirements and support both Android and iOS.

When developing a mobile application, there exist multiple possible options to choose from (Jobe, 2013). We look into three of them, in particular, a web application, a native application and using a mobile app framework. Additionally, we choose the development approach we will use for the mobile app.

3.5.1 Web Apps

Web applications are websites that run inside browsers, meaning they allow for cross-platform development. However, as they depend on the implementation of the browser, their speed is limited to the browser's speed.

Recently, a new initiative called Progressive Web App (PWA), has been pushed by several companies and organisations, one of them being Google (Google Developers, 2019). We explore the advantage and disadvantages of this new initiative.

A PWA is a website which looks and behaves like a native application. PWAs make use of the advancement in browsers, that allows websites access to system resources which makes the website feel like it was being run as a native app. PWA is a general term that can be used for apps that adheres to certain design criteria (Alex, 2015). PWA is developed like regular web apps, keeping properties such as being responsive and scale based on the phone screen's resolution.

Since PWA is developed as regular web apps, they are not suitable for app stores such as Google's Play Store, without use of some conversion to generate supported files like apk files. However, for newer versions of Chrome the browser will ask the user if they wish to add the app to their home-screen. If they grant the corresponding permissions, the application will be installed on the device and feel just like if the website was a native app (Alex, 2015; Google Developers, 2019).

While looking into PWA we found that creating an app following these guidelines was of great benefit, as this effectively allowed us to create an application that works on all devices equipped with a modern browser. However, we find that the functionality of scanning for access points over Wi-Fi is missing in Chrome, effectively meaning that we cannot use this required functionality on Android devices. We instead decide to look into creating a native app.

3.5.2 Native Apps

A native app is designed to work on a specific platform. This allows a native app to utilise the operating system directly. This allows for a higher degree of optimisation because of the closer interaction with the operating system. It also gives the app access to familiar UI elements from the operating system.

However, this means that the app must be developed for a specific platform. As one of our requirements is supporting most of the currently used mobile phones, we would have to develop an app for each platform. Since the market share is pretty much split into Android and iOS. This means we would have to maintain two apps.

The maintenance of multiple code-bases seem excessive so we look into mobile app frameworks that allow for cross-platform development for native apps.

3.5.3 Mobile App Frameworks

Mobile app frameworks allows for cross-platform development. They achieve this with a set of cross-platform abstractions, allowing for translating one code base to different target platforms. This allows us to create a native app that works on the two target platforms.

One of these frameworks is NativeScript, that supports both Android and iOS. NativeScript uses CSS styling and HTML-like syntax for its UI, and JavaScript for the app logic, that gets executed into native code.

We choose NativeScript as our application development framework as it is a cross-platform tool for mobile app development which fulfil our requirements for the app. Additionally, NativeScript

allows for object-oriented programming through TypeScript.

3.6 Mobile Application

In section 3.5 we choose to use NativeScript as the development platform. This section goes in depth with the design of the mobile app and the choices we make regarding its design. Additionally, it details the navigation of the app and its functionality.

3.6.1 Initial Design from Requirements

This section details the design in relation to the requirements and illustrate the planned navigation of the app.

As stated in the requirements described in section 2.5.1, the app should support creation of new users. Functionality for sign-in and sign-up processes must be present. Sign-up sends a request which creates a new user with the provided credentials, while log in attempts to log in using the provided credentials. If the credentials do not match, then the user is informed of the failed log in attempt. Additionally, in case the user forgets their password, the app should allow the user to reset their password by getting a password reset link sent to their email inbox.

Since the app needs to be a tool for the user to set up the ventilation system, the process of doing this should be easy for non-technical users can accomplish. This implies that the setup of the TeleVent units needs to be automated via the app.

Since we want to give our users an overview of the historical data, it is important to show it in a manner that is easy to comprehend. We choose to use a graph to visualise the data.

It is also necessary that each unit is recognisable to the user via the app, such that they can relate the data to a unit. We decide on a graph for every type of sensor, with the possibility of showing data from the last hour, day and week. The graph should be able to update in real time, and indicate the status of the indoor climate based on the context from the problem analysis. We find however, that users with lots of sensors, who wants an overview of the entire house, has to perform a tedious amount of actions. Specifically, it requires them to click every single unit to observe the graph of data produced by that specific unit, then compare this with all the others. Therefore, we decide on user-defined collections of units that can be shown in the same graph, allowing users to see the information they want at once. We refer to these as collections from now on. Lastly, the app should be able to set the desired temperature for the house.

From these initial design decisions we came up with an overview of the planned functions of the app and the scheme to navigate between them. The overview can be seen in figure 3.11. The figure uses UML-based Web Engineering (UWE) notation (KroiSS and Koch, 2008; Web Engineering Group, 2016) which is an extension for Unified Modeling Language (UML).

3.6.2 App Functionality

In this section, we briefly describe the functionality shown in figure 3.11 and the choices made with respect to the functionality. We also describe the separate application screens, which we from now on call pages.

When launching the app, it opens on the Log In page which asks the user to log in, instead of sign-up because of the disparity in usage between the two. The app caches the received token when they log in, so it should not be necessary to log in after restarting the app.

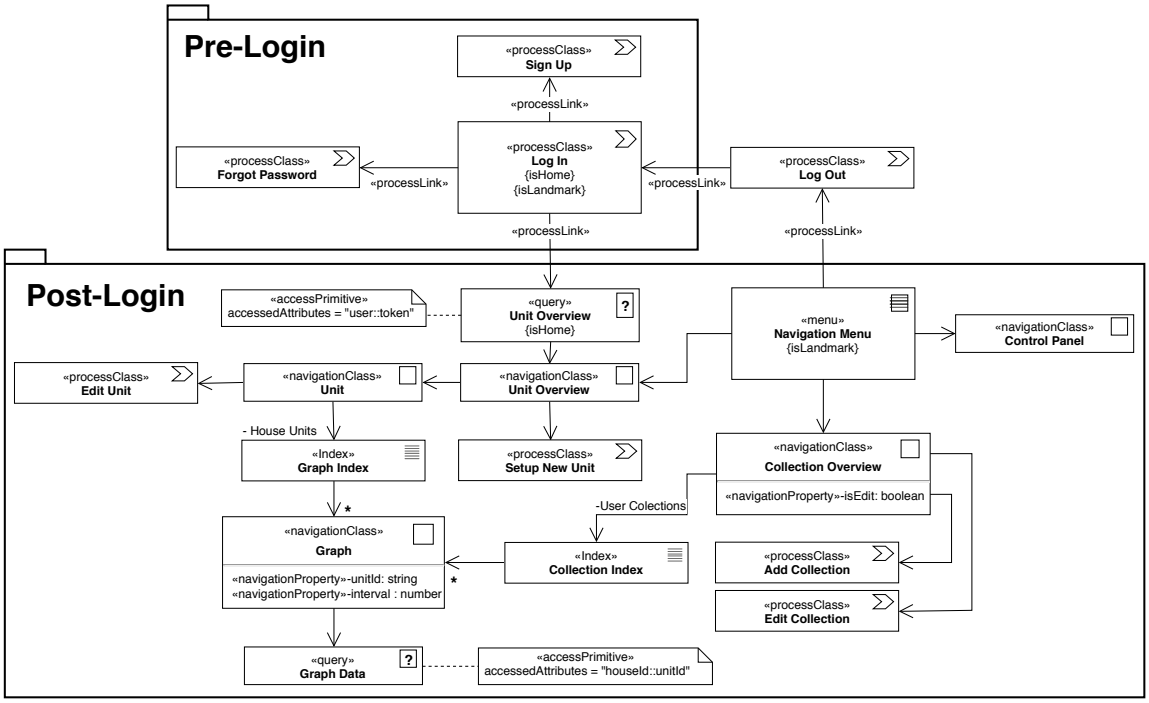

Figure 3.11: Navigation Model of the mobile app

If a user has not yet registered, they can do so by pressing a button on the Log In page that navigates to the Sign Up page. For creating a new user the only necessary fields are email and password. If a user has forgotten their password, they can navigate to the Forgot Password page and get a password-reset token by inputting their email. Once logged in, it should present the user with an overview of all their units, with the corresponding name and an indication of any errors for the unit. If the user has not given a name, we instead use the units MAC address. Pressing any of the units will open the specific Unit page where the user can see graphs, change unit name or desired temperature.

Navigating to the Graph page of any unit presents the user with graphs of the sensor data from that unit. This is to give the user an overview of the IAQ of the house. We split the Graph page into several tabs, each with their own graph. Each type of measurement explains something different. If both humidity and temperature were shown in the same coordinate system, it could confuse people as they use different units of measurement. The unit of measurement should be displayed on the vertical axis, to help the user read the graph correctly. There might be multiple sensors measuring the same property, for instance: two temperature sensors one measuring the indoor temperature, and one measuring the outdoor temperature. In such cases they should be shown the same coordinate system, as long as there is a distinction between them. Using colour coding for this could be an option, such as using blue for the outdoor temperature, and red for the indoor temperature.

As previously mentioned we include a way for the user to specify a grouping of units called collections. To give the user an overview of the existing collections, we have the Collection Overview page. This page lists all the existing collections and allows users to add and edit collections. The page also allows a user to view graphs, showing data from the units of a collection if the user taps one.

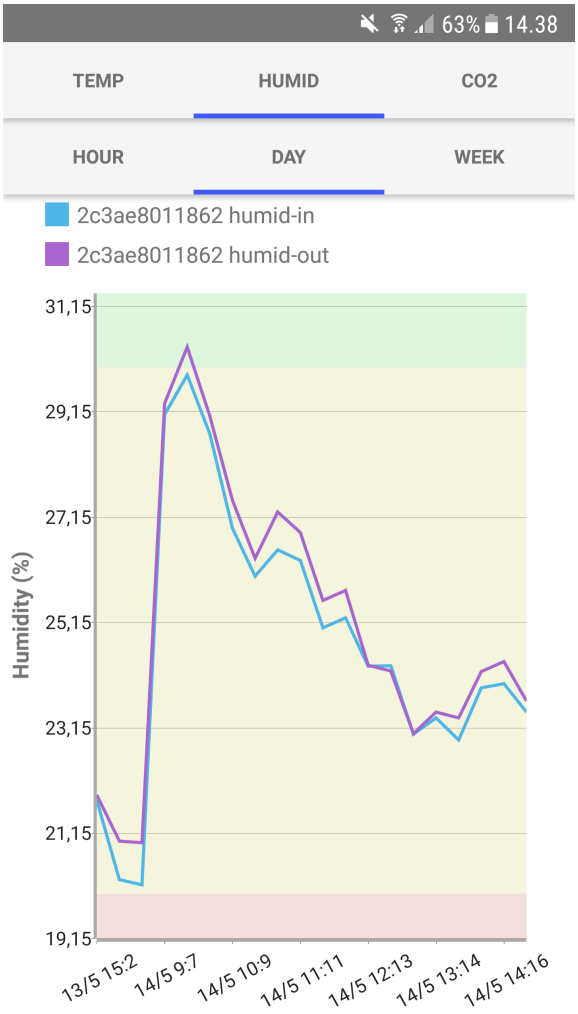

Figure 3.12: Example of a graph in the app

When visualising the data, we found the amount of data points to display could become too many. Since the units send data every five seconds, we get 120,960 data points within a week per unit per sensor. Including this amount of data in a graph is unnecessary and makes it very cluttered, so we limited the amount of data points in the graph. The amount of units is dynamic and some unit sensors can be shown in the same graph, like indoor and outdoor humidity. For an illustration of our sensor types, see figure 3.12 where the sensor types of our units are visible: temperature, humidity and CO_2 respectively. The chosen sensor type, humidity, has two sensors, each shown with the two legends, giving an indication of indoor and outdoor humidity.

The Collection page also allows users to add and edit collections. By tapping the *edit* or *add* collection buttons on the Collection Overview page, the user gets redirected to the Add/Edit Collection page. The page should have two initial states; add and edit. Add will load nothing while edit loads the previously saved options. In either case, the Collection page contains a list of all units the user owns and a way to mark a unit. Additionally, there should be a text-field to give the collection a name. Lastly, it contains the *save* button, that either adds the collection to the list of collections or commits the edit.

Should a user want to add a new unit, then the user should be able to do this from the Unit Overview page. By pressing the a *add* button, the app asks for the required permissions that allow the app to scan for access points, since units that are not configured set up access points. If the users phone finds such an access point, then the user can configure that unit as part of their home network. If they choose to configure it, the app disconnects from the current Wi-Fi network and connects to the access point of the unit to configure it. This causes the unit to connect to the user's own Wi-Fi and to the MQTT-server, allowing the unit to publish measured data, and receive commands from the controller. Afterwards, the app reconnects to the network the user's phone was previously connected to. The unit then shows up on the Unit page together with the others.

The Unit Overview page also allows the user to set a target temperature of the house. This allows the controller to try and adjust the temperature of the house towards the temperature the user wishes for.

In the end not all the planned functionality was added the the app, see section 5.1 for a discussion about what was not added and why.

In this chapter, we explained our solution for a ventilation system that improves IAQ by considering the factors and the location of detected devices through the use of lateration. We describe how we design and implement the firmware, the server-side applications and the app. Having developed the system, we next explain how we tested the major parts of it.

Chapter 4

Testing

We have to test our system to make sure it works according to specification. We first write a test plan that describes what parts we test and how, as well as what tools we use to test each part. We then explain in more detail how we test the hardware to check if all sensors and the behaviour of the fans work as we expect them to. We then explain how we test the API and what issues that uncovers and what problems having existing tests help to mitigate when things are modified in the API. Having tested our API we explain how we test the app. Finally, we explain how we plan to perform our system test and then we reflect upon our decisions and test results.

4.1 Master Test Plan

We want to test the three parts of our system individually in order to ensure that they behave according to their specification. In this section we describe which types of tests and tools we use to do so and why. The prototype mentioned in 3.2.2 is not tested, as it is a proof of concept without a specification. We do not use line-, method- or class-coverage as a measure of test quality, as they provide little insight in the quality of tests. Instead we analyse which parts are crucial to test and document how comprehensive the tests is for each critical part.

4.1.1 Definitions

The following is a list of definitions for concepts and types of tests that we use.

Bug An inconsistency between the specification and a part of the system.

Test case Is a single test with a single purpose.

Test Suite A collection of one or more test cases.

Unit test A small test which tests a small part of a program like a single execution of a single method.

System test Is like an acceptance test and integration test, where we test the whole system and the interaction of the different parts of the system.

Hardware test We test the hardware for bugs by verifying that each component provides sensible output when given correct input. The quality of the output is not important, we are interested

in whether or not the component is properly connected and functional. We assume that the hardware has been tested by the manufacturer and that its output is in accordance to the specification.

Firmware test There are large parts of the firmware code base that we cannot automatically test, as it relies on changes in the physical world. To test this, we load the firmware onto a TeleVent unit, artificially change the environment and send values to it via a test server and observe the result to see if the firmware behaves as expected. The parts of the firmware that performs logic operations which can be separated from the hardware specific code is also unit tested as it can be prone to mathematical errors.

API tests The API endpoints will be tested to validate their output as erroneous API endpoints can result in poor user experience, as the users can not read historical data if it can not be retrieved from the server.

Load test We will not perform any load testing because the amount of users we can serve is not important for this project as we merely want to show how a user can interact with the ventilation units.

App tests We perform App tests to verify that our app can be used with different kinds of Android mobile phones. We test multiple different phones, as the hardware configuration and operating system version varies a lot. We would like to test all of the most used devices but this requires access to each of them, so we stick to testing the different mobile phones we have access to within the group.

4.1.2 Tools Used for Testing

In this section we describe which tools we use for each type of testing

Google Test

Google Test is a unit testing framework for C++ and has support for both Continuous Integration and Deployment (google, 2019).

Postman

Postman is a tool that allows the testing of endpoints for a developed API (Postman, 2019). We use Postman as it allows us to create tests for each endpoint in our API. It also allows us to create pre-requests so we can check what data is already present in the system, login, clean up or similar before we send the request that we are actually testing. If the response is JSON, we can test if all the required fields are present, and test if the values are what we expect as well. The tool allows us to asynchronously test all created requests, which allows us to easily run all test whenever we modify the API.

Katalon

Katalon is a free testing tool that can be used for API, mobile and web testing. (Katalon LLC, 2018) We use Katalon for automated mobile testing, as it supports testing both iOS and Android. We connect a device and run the app through Katalon then we perform the actions from our designed test case and save that recording. Afterwards we can add statements to verify properties of UI

elements. The test cases can be bundled in a test suite, which we can then run on a connected device.

4.1.3 Test Phases

Phase 1: Initial Tests The initial system specification must be formed. During this phase we will write the test plan, hardware tests, firmware tests and the API tests. The exit criteria for this phase is that the initial test plan is complete, defined by all the above tests having been implemented.

Phase 2: App Tests Before this phase can begin the test plan and the app must be completed. During this phase app tests will be designed, implemented and run. The exit criteria for this test phase is a app test suite that can test the apps adherence to the specification.

Phase 3: Acceptance This phase will begin once all implementation is halted and only bug-fixing will occur. During this phase we will design the system test and perform it and we will evaluate our system. Exit criteria is the end of this project.

4.1.4 Resource Requirements

To perform our firmware tests, a computer with a C++ compiler such as gcc or clang installed, and a GNU-compatible make system is required (google, 2019). To perform the App test, a computer running Katalon and a connected Android device is required. To perform the API tests, a computer running Postman, is required. To perform the system test, an Android device, a TeleVent Unit and a computer capable of performing the afformentioned tests, are required. To perform the Hardware tests, a computer with a C++ compiler, and a TeleVent Unit is required.

4.2 Hardware Tests

We have found no free tool to perform hardware testing, so we create a program for the TeleVent units that will write the test results to a connected computer.

We need to test all of the hardware peripherals and the MCU listed in Figure 3.9. In the hardware test program we test each component independently and write the state of the component. This kind of test is useful while developing because if our firmware fails it can be hard to determine whether it is a bug in the firmware or in the hardware. It is also a useful tool to have when mass producing a product, as we can use the test for quality control before a product is sent from the factory.

4.2.1 Hardware Test Design

The following tests should be performed in a small chamber with control over temperature, humidity and CO_2 contents of the air. These tests are meant as a tool for better testing which could be perfomed by AERep. Lacking access to such we performed a second set of tests described in section 4.2.2.

HDC1010 Test Suite

To test the HDC1010 we have a test suite consisting of three test cases each with different temperatures and relative humidity. If any of the temperature readings vary by more than 1 řC, we reject

the value. If any of the relative humidity readings vary by more than 5 %, we reject the value. If any of the values are rejected, then the test case and test suite fails. The three test cases has the following temperatures and relative humidity:

First test case Relative humidity is 70% and temperature is -20 řC.

Second test case Relative humidity is 40% and temperature is 20 řC.

Third test case Relative humidity is 10% and temperature is 50 řC.

CCS811 Test Suite

To test the CCS811 sensor, we create a test suite consisting of four test cases, each with different levels of CO_2. A test case is failed if the reading differs with more than 100 ppm. If any of the test cases fail, then the test suite fails. The four test cases has the following CO_2 levels:

1. 400 ppm
2. 1000 ppm
3. 4500 ppm
4. 8000 ppm

LED Test Suite

We have two LEDs, one green and one orange, we create three test cases for each LED. We accept a test case if the time a LED should be turned on varies with less than 10 milliseconds. The test suite fails if any of the test cases fail. A LED test suite consist of the following three test cases:

1. Light on for 200 ms, off for 200 ms, repeat five times
2. Light on for 500 ms, off for 100 ms, repeat five times
3. Light on for 5000 ms, off for 1000 ms, repeat five times

Fan Test Suite

To test the two fans, we set the fan speed to every 100 RPM setting within its operating range and measure the speed three seconds later. If the measured speed diverges with more than 100 RPM from the set speed, then the test fails.

Wi-Fi Test Suite

To test the Wi-Fi module we connect to a known router and if the connection is successful, then we accept the module as functioning correctly.

4.2.2 Simplified Hardware Tests

We do not own the equipment required to perform the above tests, so we have formulated a set of test cases that we can perform with our equipment and in our environment. Since we cannot correctly test our hardware for bugs according to the specification in appendix A, we form the following test cases:

HDC1010 Test Suite

To test the two HDC1010 components we read the temperature and relative humidity from them and check that the values are within acceptable limits. The acceptable limits are between 20-26 řC, and the relative humidity is between 30-60 %.

CCS811 Test Suite

To test the CCS811 sensor we read a value from it and check that the value is within the acceptable range of 400-2000 ppm.

Fan Test Suite

To test the two fans we set a speed and then wait three seconds for the fans accelerate, then we calculate the RPM. We then check if the RPM is as expected within the margin of error of 100 RPM. The test personnel should listen and observe the fan during this test to verify that the fan actually rotates and changes speed.

LED Test Suite

To test the LEDs we need test personnel to observe that the LEDs light up as expected.

Wi-Fi Test Suite

To test the Wi-Fi module we connect to a known router, and if the connection is successful, we accept the module as functioning correctly.

Hardware Test Results

We found no bugs with the hardware tests. At an early point before the creation of hardware tests, we discarded several TeleVent units because we suspected faulty hardware. We eventually found a bug with with interrupts related to the tach signal in our firmware, but we could have avoided the discarding with hardware tests.

4.3 API Test Execution

Testing the API requires first creating a request for an endpoint that needs testing. For the API, many endpoints require authentication, so prior to sending the request, a pre-request is necessary in order to log in and get a token that can be used for the request requiring authentication.

The pre-requests allows us to assign environment variables with values received from the responses, allowing the actual request to succeed as expected. We can then write tests to check if the response code from the request is as expected, and if the received object matches the specification, specified in appendix B. Some API endpoints involves modifying the database, such as adding new units, modifying them or deleting them. In order to get the database back into the same state it was before the request, additional cleanup code is needed within the tests, using other endpoints while assuming the API behaves as expected.

An example of this is testing if it is possible to add a unit. First, it is necessary to make a pre-request that makes us log in and receive a verification token that can be used for the add device request. Then the request is sent, and we check the response to see if it matches the specification. Once the test has been executed, we need to send a request to delete this unit using the delete device endpoint. The story is similar for the endpoint testing if a unit can be deleted, as we first have to add a unit in a pre-request.

Many of the endpoints return JSON objects, as specified in appendix B. To test if all required fields are present and correctly formatted, Postman makes use of a JSON validation library that allows it to check if all required fields are present, and their types are correct. In case of modifying an existing entry in the database, we can also access the fields from the returned result of the modification request, but also the fields of the returned object from a GET request, in order to test if modifications are also saved correctly to the database.

A test to see if we get back the correct temperature by testing the GET method on the /house-/temperature endpoint, can be seen in listing 4.1.

```
1  //Pre-request Script
2  pm.sendRequest({
3      url: pm.environment.get("IP") + '/login',
4      method: 'POST',
5      header: {
6          'content-type': 'application/json'
7      },
8      body: {
9          mode: 'raw',
10         raw: JSON.stringify({ email: pm.environment.get("email"),
               ↪ password: pm.environment.get("password") })
11     }
12 }, function (err, res) {
13     pm.environment.set("token", res.json().token);
14
15     pm.sendRequest({
16         url: pm.environment.get("IP") + '/house/temperature',
17         method: 'PUT',
18         header: {
19             'content-type': 'application/json',
20             'Auth-Token': pm.environment.get('token')
21         },
22         body: {
23             mode: 'raw',
24             raw: JSON.stringify({ temperature: 25 })
25         }
26     });
27 });
28
29 //The test
30 pm.test("Temperature on house is correct", function () {
31     pm.expect(pm.response.json()).to.eql(25);
32 });
```

Listing 4.1: Pre-request and test on /house/temperature endpoint for GET method

The pre-request first logs in a user, as the endpoint requires authentication, then stores the token in the environment, as this is needed to make the request. Once logged in, the pre-request script makes another request which sets a temperature on the house. We assume this method works as intended as the only way we know that the GET method works correctly, is if the result we get back from the request, matches what is on the house. By setting the temperature in advance, we

know what the result should be. The test part runs after the actual GET request has been made, and this test, just checks if the response we get, is the same value we set the temperature to be in advance.

API Test Results

Each time the API has been extended with new functionality, additional test-cases have been written to test if the new functionality behaves as expected. If existing functionality changes, the already existing tests have been used to ensure the endpoints still behave as expected and do not introduce any new bugs. By regression testing our API we have discovered some bugs in the API when we made changes that has helped us identify the bugs and correct them. An example of this is when we changed the API to reflect changes in the database. Issues appeared when we moved the name and online fields of what was previously embedded unit documents in the units array on a user, to having them as fields on unit documents in the units collection. This caused these fields to be missing for certain endpoints and the tests helped catch these problems and fix them in advance.

4.4 App Test

Testing the mobile application we use the Katalon tool to create and run tests, but to run the tests on an Android device, the device must be physically connected to the computer performing the tests. The Android phones we have access to, are listed in table 4.1.

Model	**Android version**
Samsung Galaxy Ace	4.4
Samsung Galaxy S3	LineageOS 14.1 built on Android 7.1
Samsung Galaxy S6	7.0
Huawei P10 lite	8.0
Oneplus 6	9.0

Table 4.1: Android device for testing

We use the Samsung Galaxy Ace and the Samsung Galaxy S6 to create two test suites one for Android 5.0 and higher and one for Android 4.4 and older. The reason for this is that newer version of Android uses a UI element called `view.viewGroup` in place of the `view.view` used by older versions. Test cases are created by connecting a device that can run the app and then recording the actions. Katalon then saves the UI elements and the actions performed on the elements as a test case. After the test case is created we can manually add statements that verify certain properties.

Item	Object	Input	Output	Description
1 - Tap	Login-SingInButton	0		Log default user in
2 - Get Attribute	Devices-PrefTemp	"text"; 0	PrefTempStart	Assigns the preferred temperature to the 'PrefTempStart' variable
3 - Tap	Devices-TempUpButton	0		Tap the 'up' button to change the preferred temperature
4 - Get Attribute	Devices-PrefTemp	"text"; 0	PrefTempNow	Assigns the preferred temperature to the 'PrefTempNow' variable
5 - Verify Greater Than		PrefTempNow; PrefTempStart		Compares 'PrefTempStart' < 'PrefTempNow' to check that the setting has changed
6 - Tap	Devices-TempDownButton	0		Tap the Down button to change the preferred temperature setting
7 - Get Attribute	Devices-PrefTemp	"text"; 0	PrefTempNow	Assigns the preferred temperature to the 'PrefTempNow' variable
8 - Verify Equal		PrefTempNow; PrefTempStart		Compares 'PrefTempStart' == 'PrefTempNow' to check that the setting has changed

Figure 4.1: Test of changing preferred temperature shown in Katalon

Figure 4.1 shows a test case from Katalon, this test case is used to test if we can change the preferred

temperature. Katalon finds UI elements and perform actions on them as *tap*, *set text* and *tap and hold*, it can fetch data from the UI elements to verify their content. In the test shown in figure 4.1, we use UI elements to change the preferred temperature and fetch data from another UI element to verify that the preferred temperature has changed. All our test cases, find certain UI elements and perform some action on that element. If the action is not viable or if the element cannot be found, the test fails.

We test the following features of the app:

Sign up To test this feature we start the app and choose the sign up option, then we fill in the user name and password for the test user and tap the *sign up* button. This test is accepted if we get a message saying user exists. This is expected since we can not automate this test with a new user. We would like to make a similar test with a new user but we can not delete a user through the app.

Log in In this test we start at the log in screen of the app, where we enter the credentials for our test user, and tap the *sign in* button. If we get to the Unit Overview page this test is accepted.

Add unit In this test we start at the Unit Overview page

, here we tap the '+' button to add a new unit to our house. We get directed to a new screen showing us a list if the unconfigured TeleVent units. It can take a while before any units show up because we scan for Wi-Fi acces points named correctly. We choose the TeleVent unit at the top of the list, then we input the password for our Wi-Fi router and tap *next* until the unit is configured. When the unit is configured we should be redirected to the Unit Overview page. If the new unit shows up on the list of units once it starts publishing, then we accept the test

Delete unit We start this test on the Unit Overview page which has a list of TeleVent units. We tap the unit at the top of the list, which navigates us to the Unit page where we tap the *delete* button. This takes us back to the Unit Overview page, and if the unit has been removed from the list of units we accept this test.

Set temperature This test starts on the Unit Overview page. First we note the number in the preferred temperature setting, then we click the button to increment the temperature, and check if the preferred temperature setting is higher than when we started this test. We then decrement the preferred temperature down button and if the preferred temperature has changed back to the same value as the start of this test we accept.

View historical data We start this test on the Unit Overview page. We choose the TeleVent unit at the top of the list of units, this sends us to, which redirects us to the Unit page where we click the *graphs* button. When we get to the Graph page we have a graph and several tabs. We click each tab once, and between each tab change we note if the graph has changed. If we can get through all tabs and close the app we accept this test.

We create a test suite that tests all of the above features, this makes running the tests easier since we can start a single suite for a device and if it gets through all test cases without detecting bugs, all features work on that device type.

App Test Results

We ran the tests on all five phones. All the features works on most of the phones see table 4.2 for the results of the app tests.

Model	Sign up	Log in	Add unit	Delete unit	Set temperature	View data
Samsung Galaxy Ace	X	X	X	X	X	X
Samsung Galaxy S3	X	X	–	X	X	X
Samsung Galaxy S6	X	X	X	X	X	X
Huawei P10 lite	X	X	X	X	X	X
Oneplus 6	X	X	X	X	X	X

Table 4.2: App test results

As shown in table 4.2 there is only one phone that can not perform all features. The Samsung Galaxy S3 cannot configure a new unit. We believe it cannot detect Wi-Fi points through the app because of lacking support from the API on the phone, possibly due to it not running a standard version of Android, although this is purely speculation. The tests has confirmed that our app is compatible with a variety of configurations but not all, we expect that all features of the app will work with standard versions of android between 4.4 and 9.0, and that most of the features will be compatible with some nonstandard android versions.

4.5 System Test

We have tested the different system parts for bugs and bugs, but we need to test if bugs arise when all parts work together, therefore we design a system test. The tools required for completing a system test is stated in section 4.1.4. Our system test goes through all the requirements as specified in section 2.5.1.

Test Setup

Before we start the test we need to set up an environment so the TeleVent units can connect to a Wi-Fi network and communicate with the server. We have to install the app on an Android phone so we have some way of using the app, the phone should be connected to a router and have the newest version of the app from the development branch. Lastly we need a computer that has some tool to make API requests to the server, we choose Postman so we can use the request we created for the API test.

System test 1: User Creation

This test case is used to test whether the app and the server support the use and creation of users. To perform this test we create a user through the app and we use the application to log our new user in. After these step are done we use Postman to fetch information about the user before deleting it to clean up. We confirm that we have deleted the user by trying to fetch its data again and expecting an error code since it no longer exists. This test case is accepted if we can create a user, log in, fetch data and delete the user without any bugs.

System test 2: Add Units

This test case is used to test the ability to add and configure TeleVent units for a house. To perform this test we put the TeleVent unit into configuration mode. We then open the app and log in with the test user before tapping the button to add a new unit. The app then searches for Wi-Fi networks from TeleVent units and notifies us, when one is found. We tap the unit to start and specify the password to the Wi-Fi network we were on before looking for units to complete the configuration. If the mobile was not connected to a viable Wi-Fi network we have to choose a nearby router and input the password for that router. This test is successful if we are able to put the unit into configuration mode and then add and configure it the app. We can use Postman to test if the TeleVent unit has been set up correctly.

System test 3: Change Temperature Setting

This test case is used to test the user's ability to change the temperature setting for a house and if the TeleVent units can use this setting. To perform this test case we must ensure that at least two units is configured to the test user and is sending data to the server. The following test are made under the assumption that only the temperature affects how the units ventilate. This can be ensured by disabling the effects of the other sensors in the firmware and controller. We then log in as the test user and set the temperature setting. To test if a unit can use our new temperature setting we consider a single unit, the indoor temperature sensor needs be warmer than the temperature setting, and the outdoor temperature needs to be lower than the indoor sensor, then the unit should start pulling air in. We test if the controller can use sensor readings from multiple units by considering two or more units, the unit that should push air out, and the other units in the house. The outdoor temperature reading on the unit that should push air out needs to warmer than the indoor sensor reading and the indoor reading should be warmer than the temperature setting. The other units should have a lower indoor temperature reading than the unit that should push air out. The unit should speed up the outlet fan as a result. This test is successful if all steps can be completed.

System test 4: View Sensor Data

This test case is for testing if historical sensor data can be saved on the server and viewed on the app. This test requires some setup, we need to have at least one TeleVent unit running and sending sensor data to the server for at least one hour. After the server has collected data for at least one hour, we log in via the app and open graphs for the unit that has been transmitting data, if the graphs contain data for at least one hour this test case is accepted.

System test 5: Offline Test

This test case is to confirm that the TeleVent units can use senor data to control ventilation. Before we start this test case we must set the preferred temperature for the TeleVent unit to 27 degree, and turn off the test router to verify that the unit is indeed offline. We can use the tool Postman to try and fetch new sensor data from the unit. If there is no new data, then unit is offline. When the setup is complete we can place a heat source that is above 30 řC on the indoor temperature sensor. If the fan pulling air inside speeds up then the test case is accepted, otherwise it is rejected.

System test 6: Remote Installation of Firmware

This test case verifies that updating the firmware remotely through the server is possible. To perform this test we use the API to install a new version that turns the orange LED on when the unit is functioning normally instead of the green one. If the unit restarts and after it starts sending new sensor data while the orange LED is on, and the green LED is off, then the test is accepted, otherwise it is rejected.

Requirement	Sytem test 1	Sytem test 2	Sytem test 3	Sytem test 4	Sytem test 5	Sytem test 6
The firmware must communicate sensor data to a central server if connected to the internet			×	×		
The firmware must be able to regulate IAQ using sensor data while disconnected from the internet					×	
The firmware must be able to detect occupants in its vicinity						
The firmware must have a feature that allows flashing the microchip remotely						×
The firmware should be easy to extend for new units with different capabilities						
The server must be able to receive data from units				×		
The server must be able to store historical data				×		
The server must feature an API that allows users access to their own historical data				×		
The server must be able to handle different types of TeleVent units						
The server must be able to regulate IAQ using sensor data from multiple units in the same house			×			
The server must allow remote firmware updates of the units						×
The server should allow a preferred temperature to be set			×			
The app must be able to give a graphical overview of historical data				×		
The app must be able to support the creation of users	×					
The app must be able to configure units		×				
The app should be able to give users an overview of the units in their house		×		×		
The app should allow users to set a preferred temperature			×			
The app should be able to run on most phones currently on the market						

Table 4.3: Requirements test fulfilment

System Test Results

We have performed all our system tests, they were all successful. The purpose of this test is to check if we meet our requirements, we have created test cases for most but not all requirements as shown in table 4.3. We have not designed test cases for the following requirements:

The firmware must be able to detect occupants in its vicinity The TeleVent unit lacks the proper hardware to detect occupants, but the controller does support location data to ventilate. Hence we cannot test this.

The firmware should be easy to extend for new units with different capabilities This is a design requirement and is not measurable by a test, hence we have not created a test case for this.

The server must be able to handle different types of TeleVent units We have not created a test case for this requirement since we have created small prototypes during this project and they have all worked with the rest of the system.

The app should be able to run on most phones currently on the market We have not created a test case for this in the system test since the purpose of the App test is to test the types of phones we can reach with the app, see section 4.1.

The remaining requirements have been covered by the system test cases as seen in table. 4.3, we have performed all these test cases which has been accepted.

4.6 Test Reflections

We have completed Three types of testing to show some testing methods and give some idea of how we can ensure a good product quality. There exist many more ways to test that could be relevant like load testing and usability testing. Many of the bugs we have found has not been through testing tools but development techniques like code review and code inspections. This has helped us find bugs early, while giving more people a better understanding of how the system works

We have found few bugs with our tests. The amount bugs found with tests may be due to the code reviews done in the group or we might not have found all the bugs, since testing can only verify a single trace of actions. We have tested for stability by leaving the TeleVent units running for days without restarting or flashing them. This has some times resulted in various bugs like the I^2C bus error where we can no longer communicate with other devices on the bus, which is handled by restarting the microcontroller.

Chapter 5

Discussion

Having developed a system for regulating IAQ in homes, based on sensor data provided by all units in the house, we discuss what we could have done differently to provide a possibly better and more robust system than the one we currently have. First, we discuss what we did with the app and why we could have gone with a PWA instead of using a framework. Next, we discuss issues of security, and why they are important to consider. We discuss what advantages there are by ventilating when we detect people, rather than considering just the sensor data which can be used as proxies for detecting people. Here, we also discuss issues with the CO_2 sensor. We discuss problems with the large amount of data produced and how we could possibly handle it in the future. Additionally, we discuss methods for localisation and how using a whitelisting approach could improve the accuracy of detecting occupants. Lastly, we discuss the problem of using our own modified version of Homie.

5.1 App

As mentioned in section 3.5.1 we talked about the advantages of a PWA, but we could not create such an app as one of our requirements is for our app being able to configure the TeleVent units.

However, as described in section 3.3.1 we have modified Homie such that the units are capable of automatically configuring each other. This is because it is laborious to configure multiple units, even with the ability of the app to change the Wi-Fi access point. With that function, it would not be much trouble setting up any number of units, even without the Wi-Fi changing ability. In addition, it is very likely a technician will be the one configuring the units, and not the customer.

We also did not manage to implement the entire design as shown in Figure 3.11. We have considered redesigning the app in any case, when we realised that the residents of a home, likely would not be the ones to configure the units. Instead of a single app, we would like a system for the technicians responsible for installation and an app to monitor and set settings for the residents.

Because of the aforementioned, we have considered, given more time, to develop a new app as a PWA, as it is arguable easier to develop a web app than a mobile app. Especially since this gives us most of the same advantages as a mobile framework application, like easy support for multiple platforms and with no license fees for extending support to other platforms[1].

[1] Like iOS

5.2 Security

Security has not been taken into account at this stage. There are many considerations to make before giving the product to customers. Mainly, all MQTT communication is transmitted as plain-text and our broker does not require username, password or any other form of credentials to allow listening in. It is possible to configure MQTT to require authentication and encrypt all the data that is sent through it, which is one way to improve a significant security vulnerability. The same can be done with the database (mongodb, 2019).

5.3 Alternative Controlling Schemes

Since we have no way of testing how well our controller performs in a real world setting, it is difficult to say if we made the right choice with how we designed and implemented it. We tried to keep it as simple as possible, while still taking into consideration multiple parameters. There are more parameters we could use to possibly to improve the quality of our controller though. An obvious choice is to include a local weather forecast, as knowing how the weather is going to be in the near future can improve the reasoning about when to ventilate.

An entirely different way to solve the problem, would be to use machine learning to find patterns in historical data, such as when residents leave and enter their house or when they exercise or cook. This could help us make better decisions regarding ventilating in advance. As with any machine learning model, data sets of a certain size are required in order to train them. This could potentially be a problem in new houses where little historical data exists.

5.4 CO_2 Sensor

We had difficulties in getting consistent results from the CO_2 sensor. It is generally more complex than the temperature sensors and for accurate measurements, calibration is a necessity. To understand if the sensor should be replaced with another type or if we do not use it correctly, more testing is needed as well as a closer look at the documentation details. If the sensor is less precise than we expected, then we need to account for this inaccuracy in systems like the controller, for example by reducing the influence of the CO_2 value.

5.5 Large Amounts of Data

In this section we discuss problems related to the data we produce and how we store it in our final product and what can be done to improve it.

5.5.1 Bandwidth

As mentioned in section 3.3.1, each TeleVent unit transmits around 1450 bytes of data every five seconds or 25 MB of data per day. If a typical house contains eight units, this is approximately 6 GB a month and could potentially be a problem for users with data caps on their internet connections. Until now we have sent all data over MQTT as ASCII strings. This is a waste of bandwidth, as we could significantly reduce the number of bytes required for our values. For a sensor reading such as temperature, whose possible values could be fully contained in a single byte, we are using two

or three bytes to send it. If we do this for all sensor readings of the TeleVent units, not including the occupancy data, we could transmit all the data using 6 bytes rather than the 14 bytes we use now, but the real savings come from the MAC addresses and accompanying RSSI that is sent as part of the occupancy detection. Sending 20 devices using a JSON array takes up a total of 540 bytes, most of which is unnecessary delimiting characters. Each device can be represented by a 6 byte MAC address and 1 byte of RSSI, which means it could be transmitted using 140 bytes. This brings the total from 554 to 146 bytes.

The 554 bytes only makes up about a third of the total bandwidth usage though. A lot of the bandwidth requirement comes from the topic strings. The length of these can be significantly reduced. A topic string as used currently could look like *devices/jasirkflwobdwsbi/2c3ae8011862/HEXUnit/temp-in*. These 53 bytes can be reduced to 11 by removing *devices* and *HEXUnit* neither of which are strictly necessary as the information is accessible in other places and using Base64 to shorten both the house id, unit id and the sensor name to 5, 2 and 2 bytes respectively. This allows us to have a billion unique houses, each with 4096 unique devices in them. Combining all these would allow us to reduce the amount of data transmitted by each unit from 6 GB to 3.

5.5.2 Storage

As discussed in section 5.5.1 the units upload a lot of data, and as such, the database will eventually consume a large amount of space as well. It is difficult to reason about how much space a single sensor reading takes up on the database because compared to the data transferred over the internet, the topic strings are not stored, the database can be compressed, but the database has to store keys for each document. Instead we try to upload a day's worth of randomly generated sensor readings and RSSI values for ten unique devices from a total of eight TeleVent units. This amounts to a total of 829,440 documents which ends up consuming 24 MB of storage. Over the course of a year this equates to 8.6 GB per house, which is a non-negligible amount. It is important to note that the compression used by MongoDB, *snappy*, might mean that the storage requirements scales sub-linearly. It is also possible to use the more effective *zlib* algorithm (Kamsky, 2015). Additionally, it does not make sense to store the RSSI for more than a few minutes at a time, as it is not shown to users. Additionally, it is possible for us to cut down the number of stored data points, for data older than a month for example.

Adding an index is not feasible because of how write-intensive the collections are. Because of this the sheer number of unindexed documents means that the controller is slowed down, because it takes a long time to fetch the newest documents every time the controller or app asks for them. It is possible to divide each house collection into a collection for each sensor, which would make indexing more feasible. Another solution is to create documents that contain every sensor reading made between two moments in time. Having a document for each hour for example, would mean creating a new document once an hour and then storing the new sensor readings in an array in this document. This also makes indexing much more feasible as the creation of new documents becomes less frequent.

5.5.3 Is a User a House?

As our database is currently designed, a user and a house has the same role. This is slightly confusing and could lead to a worse user-experience. For example, it would make more sense for multiple people living in one house to have multiple accounts all connected to the house. This means they would not have to share an account with all that entails. Additionally, we do not support a user with multiple homes. Creating a distinction between houses and users would require several changes to the API and other systems.

An additional thing to consider is including the concept of a room. A house has multiple rooms, and each unit is associated with two rooms or one room and the outside. In the future there might be more than one device associated with a single room. This information could be taken into account by the controller in order to improve the IAQ.

5.6 Localisation

There are methods available for indoor localisation that we did not look into or test. We did investigate using Bluetooth® RSSI in a similar manner as to how we used Wi-Fi RSSI, but found that our hardware was not sufficient. Using Bluetooth would require that occupants have it turned on at all times. While not an unrealistic requirement, it would make us more reliant on how occupants configure their mobile devices, which is what made us avoid other common methods for indoor localisation. These are often designed for helping users find their own location within buildings, which makes it more acceptable to require user action. Since we were interested in detecting any occupants, including guests in the home, this is not something we were interested in.

5.6.1 Whitelisting Devices

As we mentioned in section 3.3.1, to detect only Wi-Fi enabled devices that can be used as a proxy for an occupant's position, we need a system for filtering out stationary devices such as desktop computers or other IoT devices. Due to time constraints, we did not manage to implement such a system, but our preliminary idea is based around whitelisting devices, if we see them in different locations in the house a set number of times within a certain period of time, as the RSSI can sporadically fluctuate. Additionally, taking devices off the whitelist after some time might be necessary, as a desktop computer could otherwise end up on it permanently if it was ever moved. Neither solution solves a potential problem that could occur if several guests are present and stay in one room, for example at a dinner party. None of these guests' mobile devices would be whitelisted if they stayed mostly stationary all night, but they would certainly have an impact on the IAQ.

A solution to this problem could be to use a blacklist instead. This way, stationary devices will be seen as mobile devices for some time, but after not moving for a certain amount of time, they are are eventually blacklisted.

5.7 Homie

After using Homie and having to change parts of it to reflect our needs, we ask ourselves if Homie even makes sense to use. If Homie gets updated, we either have to maintain our own fork of the project, or merge the upstream changes. This can become quite a big task that could otherwise be circumvented by writing the functionality of Homie which we need ourselves. This can lead to a significantly smaller code-base that is easier to maintain, as we only really use two features of Homie, these being the configuration functionality, and acting as a client using MQTT. As space is also a limiting factor, having a smaller firmware image, is also an advantage.

Chapter 6

Conclusion

As described in section 2.5, our problem analysis leads to the following problem statement.

Problem Statement

How can we design and develop software for an IoT ventilation system that regulates indoor climate using environmental factors and the locations of occupants?

In order to answer the question we explored the technologies that are often used in IoT devices, ventilation systems and mobile systems. Through cooperation with David Stien Pedersen from AERep we combined all three in an already existing hardware platform. Given that the hardware is subject to change in the future, meant creating firmware that would support new types of sensor configurations and units with different capabilities.

We incorporated the concept of mobile systems into the solution by creating a distributed system that communicates through a central server and by utilising the location of occupant's mobile devices. We used the strength of Wi-Fi signals from their devices and the concept of lateration to pinpoint their location within a home. Combining the whereabouts of occupants with sensor readings form different parts of the house allowed us to create a controller that decides which units should draw air from neighbouring rooms and the outside and which units should do the opposite. We hoped this would make it possible to optimise the ventilation such that noise pollution and energy consumption could be kept at a minimum while still keeping the IAQ at acceptable levels.

In order to present the data collected from the units to users, we also developed a mobile app that can show historical sensor data as graphs. The app can also be used to keep track of the units in a user's home, by naming them. Additionally, the app also allow for a simplified setup process of each individual unit, by utilising Android's API to automatically switch between Wi-Fi networks.

Unfortunately, it was not possible with the limited resources available to us, to verify whether our controller was actually successful in performing better than a similar system that did not take location or sensor data into account. But we created a set of tests that will be useful in verifying the correct working order of the system as a whole and each component individually. It is also questionable whether or not our focus on easing the setup of individual devices in the user-facing part of the system made much sense, as the system will almost certainly have to be set up by a professional technician. Instead we could have put more focus into providing a cohesive user experience across the app, and possibly also tested the usability of it.

In conclusion, our system met the requirements we defined for it, but more work can be put into it, especially regarding keeping data secure and improving the user experience of the app.

Bibliography

Toftum, Jørn, Pawel Wargocki, and Geo Clausen (2011). *Indeklima i skoler Status og konsekvenser*. URL: http://orbit.dtu.dk/files/6383686/prod21325495269795.toftum-2.pdf.

U.S. Environmental Protection Agency (2019). *International Air Quality*. URL: https://www.airnow.gov/index.cfm?action=airnow.international.

Scientific Committee on Health and Environmental Risks (2007). "Opinion on Risk Assessment on Indoor Air Quality". In: URL: http://ec.europa.eu/health/ph_risk/committees/04_scher/docs/scher_o_055.pdf.

WHO (2010). *WHO Guidelines for Indoor Air Quality: Selected Pollutants*. WHO Regional Office for Europe. ISBN: 9789289002134. URL: http://www.euro.who.int/__data/assets/pdf_file/0009/128169/e94535.pdf.

Seppänen, O. A., W. J. Fisk, and M. J. Mendell (1999). "Association of Ventilation Rates and CO_2 Concentrations with Health and Other Responses in Commercial and Institutional Buildings". In: *Indoor Air* 9.4, pp. 226–252. DOI: 10.1111/j.1600-0668.1999.00003.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1600-0668.1999.00003.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1600-0668.1999.00003.x.

Ole Fanger, P (2006). In: *Indoor Air* 16.5, pp. 328–334. DOI: 10.1111/j.1600-0668.2006.00437.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1600-0668.2006.00437.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1600-0668.2006.00437.x.

Turanjanin, Valentina et al. (2014). "Indoor CO2 measurements in Serbian schools and ventilation rate calculation". In: *Energy* 77, pp. 290–296. ISSN: 0360-5442. DOI: https://doi.org/10.1016/j.energy.2014.10.028. URL: http://www.sciencedirect.com/science/article/pii/S0360544214011773.

Arbejdstilsynet (2018). *Indeklima - Vejledning om de hyppigste årsager til indeklimagener samt mulige løsninger*. URL: https://amid.dk/regler/at-vejledninger/indeklima-a-1-2/.

Zehnder (2014). *Absolute vs. Relative Humidity Whats the Difference?* URL: https://zehnderamerica.com/absolute-vs-relative-humidity-whats-the-difference/.

Heseltine, Elisabeth and Jerome Rosen (2009). *WHO guidelines for indoor air quality: dampness and mould*. WHO Regional Office Europe.

Harvard Women's Health Watch (2018). *Easy ways you can improve indoor air quality*. URL: https://www.health.harvard.edu/staying-healthy/easy-ways-you-can-improve-indoor-air-quality.

Gubb, C. et al. (2018). "Can houseplants improve indoor air quality by removing CO2 and increasing relative humidity?" In: *Air Quality, Atmosphere & Health* 11.10, pp. 1191–1201. ISSN: 1873-9326. DOI: 10.1007/s11869-018-0618-9. URL: https://doi.org/10.1007/s11869-018-0618-9.

U.S. Department of Energy (2019a). *Ventilation*. URL: https://www.energy.gov/energysaver/weatherize/ventilation.

— (2019b). *Whole-House Ventilation*. URL: https://www.energy.gov/energysaver/weatherize/ventilation/whole-house-ventilation.

aereco (2019). *DX SYSTEM - HEAT RECOVERY, DEMAND CONTROLLED VENTILATION SYSTEM*. URL: https://www.aereco.com/wp-content/uploads/2019/01/FLY651GB_v8_display.pdf.

AerHaus (2018). *Renson HealthBox DCV system*. URL: https://aerhaus.com/products/dcv/renson-healthbox-dcv-system/.

Nilan (2014). *Ventilation & passive heat recovery*. URL: http://reader.livedition.dk/nilan/1460198822/html5/.

Wang, Fulin et al. (2017). "Predictive control of indoor environment using occupant number detected by video data and CO2 concentration". In: *Energy and Buildings* 145, pp. 155–162. ISSN: 0378-7788. DOI: https://doi.org/10.1016/j.enbuild.2017.04.014. URL: http://www.sciencedirect.com/science/article/pii/S0378778816312075.

Mirakhorli, Amin and Bing Dong (2016). "Occupancy behavior based model predictive control for building indoor climateA critical review". In: *Energy and Buildings* 129, pp. 499–513. ISSN: 0378-7788. DOI: https://doi.org/10.1016/j.enbuild.2016.07.036. URL: http://www.sciencedirect.com/science/article/pii/S0378778816306338.

Adafruit (2019). *How PIRs Work*. URL: https://learn.adafruit.com/pir-passive-infrared-proximity-motion-sensor/how-pirs-work.

Dargie, Waltenegus and Christian Poellabauer (2011). *Fundamentals of Wireless Sensor Networks: Theory and Practice*. DOI: 10.1002/9780470666388.

Bourke, Paul (1997). *Intersection of two circles*. URL: http://paulbourke.net/geometry/circlesphere/.

Dong, Qian and Waltenegus Dargie (2012). "Evaluation of the reliability of RSSI for indoor localization". In: *2012 International Conference on Wireless Communications in Underground and Confined Areas*. IEEE, pp. 1–6. URL: https://www.rn.inf.tu-dresden.de/dargie/papers/icwcuca.pdf.

ROGER, Marvin (2019). *Homie for ESP8266*. URL: https://github.com/homieiot/homie-esp8266.

Kay Kim and Andrew Aldridge (2018). *Geospatial Queries*. URL: https://docs.mongodb.com/manual/geospatial-queries/.

Piper, Andy (2013). *questions*. URL: https://github.com/mqtt/mqtt.github.io/wiki/questions.

Yuan, Michael (2017). *Getting to know MQTT*. URL: https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/.

Mobile Operating System Market Share Worldwide (2019). URL: http://gs.statcounter.com/os-market-share/mobile/worldwide.

Jobe, William (2013). "Native Apps Vs. Mobile Web Apps". In: *International Journal of Interactive Mobile Technologies (iJIM)* 7, pp. 27–32. DOI: 10.3991/ijim.v7i4.3226. URL: https://www.researchgate.net/publication/268153001_Native_Apps_Vs_Mobile_Web_Apps.

Google Developers (2019). *Progressive Web Apps*. URL: https://developers.google.com/web/progressive-web-apps/.

Alex (2015). *Progressive Web Apps: Escaping Tabs Without Losing Our Soul*. URL: https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/.

KroiSS, Christian and Nora Koch (2008). *The UWE Metamodel and Profile User Guide and Reference*. URL: http://uwe.pst.ifi.lmu.de/download/UWE-Metamodel-Reference.pdf.

Web Engineering Group (2016). *About UWE - An approach based on standards*. URL: http://uwe.pst.ifi.lmu.de/aboutUwe.html.

google (2019). *Google Test*. URL: https://github.com/google/googletest.

Postman (2019). *Postman Simplifies API Development*. URL: https://www.getpostman.com/.

Katalon LLC (2018). *Simplify API, Web, Mobile Automation Tests*. URL: https://www.katalon.com.

mongodb (2019). *Security*. URL: https://docs.mongodb.com/manual/security/.

Kamsky, Asya (2015). *New Compression Options in MongoDB 3.0*. URL: https://www.mongodb.com/blog/post/new-compression-options-mongodb-30.

Appendices

Appendix A

System Specification

A.1 Unit specification

A.1.1 Hardware specification

- must be able to measure temperature from -20 to 50 degrees C with an accuracy of ±0.5 degrees C
- must be able to measure relative humidity with an accuracy of ±2 percent.
- must be able to measure CO_2 in parts per million with an accuracy of ± 100ppm.
- must be able to control the speed of two 12v fans drawing a maximum of 250mA
- must have two buttons, one to reset the MCU and one for general purposes
- must have a green and a orange/red LED
- must be able to connect to Wi-Fi networks and send and receive data from servers outside of the WLAN
- must be able to act as Wi-Fi access point for at least one connected client
- must be able to measure distance to nearby Wi-Fi stations (mainly smartphones) to an accuracy of ±5 meters in a room without obstruction
- The MCU must be able to receive new firmware over Wi-Fi

A.1.2 Firmware specification

Led blinking patterns

- When the device is booting and not yet ready to operate, the orange and green LED will toggle synchronously between off and on every 500ms (±50ms).
- After booting, if the device hasn't been configured with a Wi-Fi SSID (and password if relevant), then the orange LED will be off and the green LED will toggle between off and on every 500ms(±50ms).
- After booting, if the device is configured

- If the device looses connection to the WLAN during normal operation then the green LED will be off and the orange LED will toggle between off and on every 500ms (±50ms).
- If the device looses connection to the MQTT server during normal operation but remains connected to the WLAN, then the green LED will be off and the orange LED will toggle between off and on every 1000ms (±50ms).
- When the device is downloading new firmware, the orange and green LED will toggle synchronously between off and on every 100ms (±50ms).

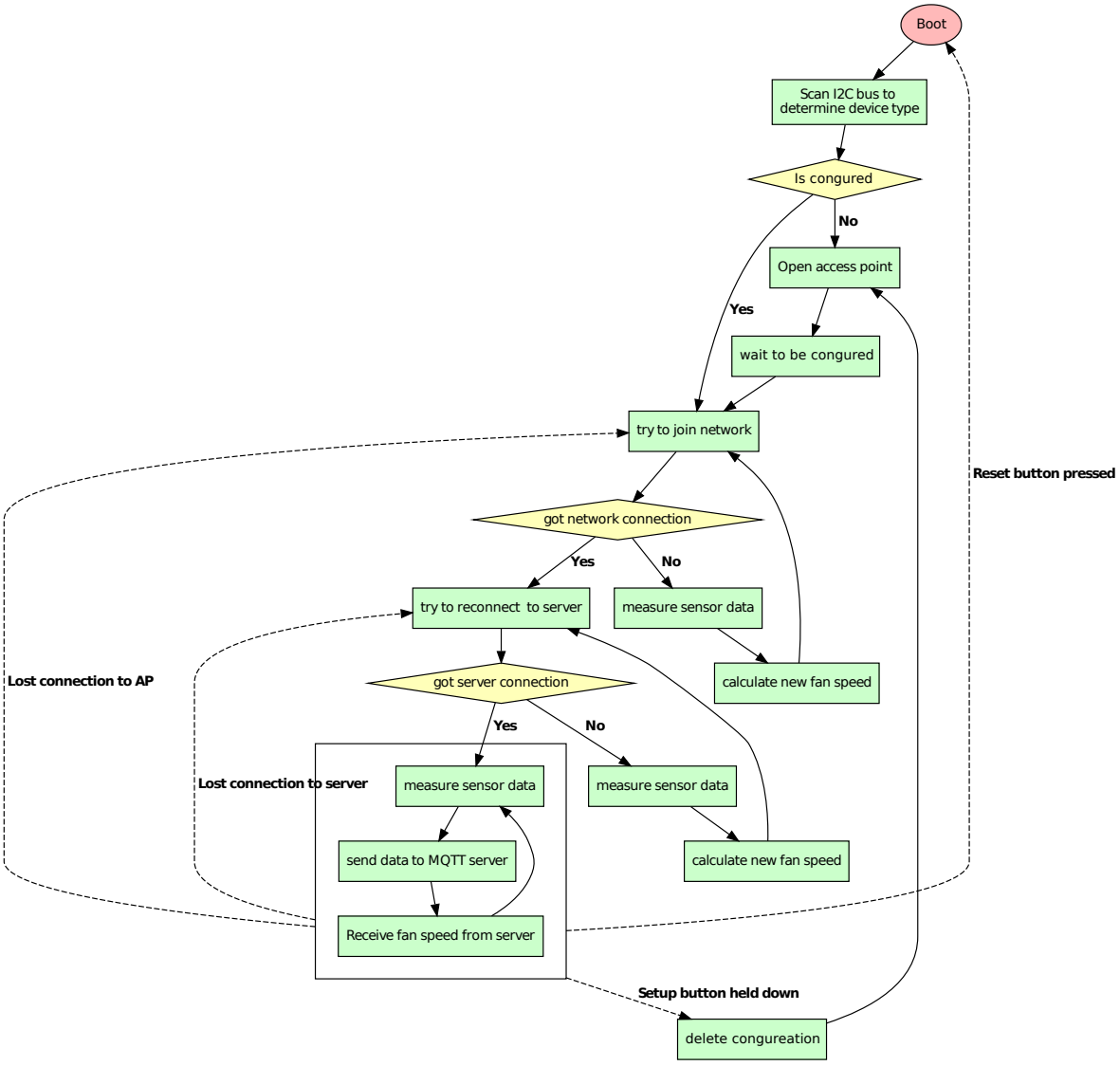

Figure A.1: TeleVent unit firmware program flow

Appendix B

API Specification

The following is a detailed explanation of each endpoint of the API. This entails what data it expects and what it returns.

B.1 General

All endpoints that expect JSON data in their body, requires it to be properly formatted. If it is not, a HTTP status code `400` is returned.

All endpoints that requires authentication needs the following header to be present. A token is provided when a user signs up, and when they log in.

- (**Required**) Auth-Token: (string)

If the header is missing or the token does not provide the proper level of authentication, status code `401` is returned.

Any endpoints that can modify existing objects responds with status code `304` if no modifications are made.

All successful requests return a status code `200`.

B.2 `/users`

This endpoint supports three HTTP verbs: POST, PATCH and DELETE

B.2.1 *POST*

Create a user.
Requires no headers.
Requires JSON body:

- (**Required**) email: (string)
- (**Required**) password: (string)

Example Usage and Response

To create a user with the email "john@example.com", and the password "smith", send the following:

```
{
  "email": "john@example.com",
  "password": "smith"
}
```

If the email is not currently in use, a response similar to the following is returned.

```
{
  "email": "john@example.com",
  "houseid": "cflxjagpjqtyraff",
  "token": "KxtKwzLLvpdWGYoABmIgGOnVTZVmZZeP"
}
```

If email is already in use, status code `409` is returned, along with a message describing what went wrong.

B.2.2 *PATCH*

Change email and password.
Requires authentication.
Requires JSON body:

- (Optional) email: (string)
- (Optional) password: (string)

Example Usage and Response

To change the email of an authenticated user send the following:

```
{
  "email": "smith@example.com"
}
```

If the email is not currently in use, a response similar to the following is returned.

```
{
  "email": "smith@example.com",
  "houseid": "cflxjagpjqtyraff",
  "token": "KxtKwzLLvpdWGYoABmIgGOnVTZVmZZeP"
}
```

If email is already in use, status code `409` is returned, along with a message describing what went wrong.

B.2.3 *DELETE*

Delete a user.
Requires authentication header.
Requires no body.

Example Usage and Response

This endpoint deletes the authenticated user.

B.3 /users/reset

This endpoint supports one verb: POST.

B.3.1 *POST*

Request password reset.
Requires no headers.
Requires JSON body:

- (**Required**) email: (string)

Example Usage and Response

To request a reset password token for the email "john@example.com" send the following:

```
1 {
2   "email": "john@example.com"
3 }
```

If a user with such an email exists, a reset password token is sent to it.

B.4 /users/reset/{reset_token}

This endpoint supports one verb, POST.

B.4.1 *POST*

Set password.
Requires no headers.
Requires JSON body:

- (**Required**) password: (string)

Example Usage and Response

To set the password of a user which has requested a password reset to "smith" send the following:

```
{
  "password": "smith"
}
```

If the reset token is valid, a response similar to the following is returned:

```
{
  "email": "john@example.com",
  "houseid": "cflxjagpjqtyraff",
  "token": "KxtKwzLLvpdWGYoABmIgGOnVTZVmZZeP"
}
```

If email is already in use, status code `409` is returned, along with a message describing what went wrong.

If the reset token is invalid status code `404` is returned.

B.5 /login

This endpoint supports one verb: POST.

B.5.1 *POST*

Log a user in.
Requires no headers.
Requires JSON body:

- (**Required**) email: (string)
- (**Required**) password: (string)

Example Usage and Response

To log in with the existing user with email "john@example" and password "smith" send the following:

```
{
  "email": "john@example.com",
  "password": "smith"
}
```

If the credentials are valid a response similar to the following is returned:

```
{
  "email": "john@example.com",
  "houseid": "cflxjagpjqtyraff",
  "token": "KxtKwzLLvpdWGYoABmIgGOnVTZVmZZeP"
}
```

If password and email do not much match any found in the database, status code `401` is returned, along with a message describing what went wrong.

If two requests are sent within five seconds of each other, status code `429` and a message describing what went wrong is returned.

B.6 /logout

This endpoint supports one verb: POST

B.6.1 *POST*

Log a user out.
Requires authentication header.
Requires no body.

Example Usage and Response

This endpoint logs the authenticated user out, by invalidating the token used in the header.

B.7 /house/temperature

This endpoints supports two verbs: GET and PUT.

B.7.1 *GET*

Get temperature.
Requires authentication header.
Requires no body.

Example Usage and Response

This endpoint returns the set temperature in the house owned by the authenticated user if it exists. Otherwise returns no value.

B.7.2 *PUT*

Set temperature.
Requires authentication header.
Requires JSON body:

- (**Required**) temperature: (string)

Example Usage and Response

To set the desired temperature in the house owned by the authenticated user to 25, send the following:

```
{
  "temperature": 25
}
```

B.8 /units

This endpoint supports two verbs: GET and POST

B.8.1 *GET*

Get units.
Requires authentication header.
Requires no body.

Example Usage and Response

This endpoints gets the units belonging to the house owned by the authenticated user. The response looks similar to the following:

```
[
  {
    "deviceid": "2c3ae8011862",
    "houseid": "cflxjagpjqtyraff",
    "name": null,
    "online": false,
    "sensors": [
      "temp-in",
      "temp-out",
      "humid-in",
      "humid-out",
      "co2"
    ],
    "location": {
      "latitude": 51.30
      "longitude": 0.70
    }
  },
  {
    "deviceid": "ecfabc1a3388",
```

```
21      "houseid": "cflxjagpjqtyraff",
22      "name": "kitchen",
23      "online": false,
24      "sensors": [
25        "temp-in",
26        "temp-out",
27        "humid-in",
28        "humid-out",
29        "co2"
30      ],
31      "location": null
32    }
33  ]
```

B.8.2 *POST*

Create a new unit.
Requires authentication header.
Requires JSON body:

- (**Required**) unitid: (string)
- (Optional) name: (string)
- (Optional) longitude: (number)
- (Optional) latitude: (number)

Example Usage and Response

To create a unit with the id "2c3ae842bfe7" send the following:

```
1  {
2    "unitid": "2c3ae842bfe7"
3  }
```

If no other unit exists with the specified id, a response similar to the following is returned.

```
1  {
2    "unitid": "2c3ae842bfe7",
3    "houseid": "cflxjagpjqtyraff",
4    "name": null,
5    "online": false,
6    "sensors": []
7    "location": null
8  }
```

If the unit id is already in use, status code `409` is returned, along with a message describing what went wrong.

B.9 /units/{unit_id}

This endpoint supports three verbs: GET, PATCH and DELETE.

B.9.1 *GET*

Get a unit.
Requires authentication header.
Requires no body.

Example Usage and Response

This endpoints gets a units belonging to the house owned by the authenticated user. If a unit with the specified id exists the response looks similar to the following:

```
{
  "unitid":"2c3ae842bfe7",
  "name": "kitchen",
  "online": true,
  "sensors": [
    "temp-in",
    "temp-out",
    "humid-in",
    "humid-out",
    "co2"
  ],
  "location": {
      "latitude": 51.30
      "longitude": 0.70
  }
}
```

If the device does not exist, the API returns code `404`.

B.9.2 *PATCH*

Change information on a unit.
Requires authentication.
Requires JSON body:

- (Optional) name: (string)
- (Optional) longitude: (number)
- (Optional) latitude: (number)

Example Usage and Response

To change the name of an existing unit send the following:

```
{
  "name": "living room"
}
```

If a modification is done, this results in a response similar to the following:

```
{
  "unitid":"2c3ae842bfe7",
  "name": "living room",
  "online": true,
  "sensors": [
    "temp-in",
    "temp-out",
    "humid-in",
    "humid-out",
    "co2"
  ],
  "location": {
      "latitude": 51.30
      "longitude": 0.70
  }
}
```

B.9.3 *DELETE*

Delete a unit.
Requires authentication header.
Requires no body.

Example Usage and Response

This endpoint deletes the specified device by removing it from the database.

B.10 `/units/{unit_id}/sensors/{sensor_id}`

This endpoint supports one verb: GET.

B.10.1 *GET*

Get sensor data.
Requires authentication header.
Requires no body.
Requires parameters:

- (Optional, default is current unix time) new: (int)

- (Optional, default is 24 hours prior to current unix time) old: (int)
- (Optional, default is 50) nresults: (int)

Example Usage and Response

This endpoint returns a number of sensor data from the specified sensor on the specified unit in a time interval. If a unit and sensor with the specified ids exist and have sent data in the specified interval, the response looks similar to the following:

```
[
  {
    "co2": 1202,
    "unitid": "2c3ae842bfe7",
    "time": "Mon, 22 Apr 2019 07:30:23 GMT"
  },
  {
    "co2": 1208,
    "unitid": "2c3ae842bfe7",
    "time": "Mon, 22 Apr 2019 07:29:23 GMT"
  },
  {
    "co2": 1215,
    "unitid": "2c3ae842bfe7",
    "time": "Mon, 22 Apr 2019 07:28:23 GMT"
  }
]
```

If the unit or sensor does not exist, the API returns code `404`.

B.11 `/admin/users/{user_email}`

This endpoint supports one verb: PATCH.

B.11.1 PATCH

Change role of a user.
Requires admin authentication header.
Requires JSON body:

- (**Required**) role: (string)

Example Usage and Response

To change the role of the user with the specified email to “admin” send the following:

```
{
```

```
2   "role": "admin"
3 }
```

Listing B.1: Sent data

If the user does not exist, status code `404` is returned.

B.12 /admin/houses

This endpoint supports one verb: GET.

B.12.1 *GET*

Get house ids.
Requires admin authentication header.
Requires no body.

Example Usage and Response

This endpoint retrieves all house ids in the database. The response looks similar to the following:

```
1 [
2     "cflxjagpjqtyraff",
3     "vftheldyjsqstksc"
4 ]
```

Listing B.2: Response data

B.13 /admin/units

This endpoint supports two verbs: GET and POST.

B.13.1 *GET*

Get units.
Requires admin authentication header.
Requires no body. Requires parameters:

- (**Optional** firmware-version: (string)

Example Usage and Response

This endpoint retrieves all units in the database. The firmware parameter can be used to only retrieve units with a specific firmware version. The response looks similar to the following:

```
[
  {
    "units": [
      {
        "unitid": "2c3ae842bfe7",
        "firmware-rev": "1.0.0",
        "hardware-rev": "2.0.0",
        "hardware-name": "HEXUnit",
        "sensors": [
          "temp-in",
          "temp-out",
          "humid-in",
          "humid-out",
          "co2"
        ],
        "settables": [
          "fan-in",
          "fan-out"
        ]
      },
      {
        "unitid": "ecfabc1a3388",
        "firmware-rev": "1.0.0",
        "hardware-rev": "2.0.0",
        "hardware-name": "HEXUnit",
        "sensors": [
          "temp-in",
          "temp-out",
          "humid-in",
          "humid-out",
          "co2"
        ],
        "settables": [
          "fan-in",
          "fan-out"
        ]
      }
    ],
    "houseid": "cflxjagpjqtyraff",
    "newest-firmware": "1.0.0"
  }
]
```

Listing B.3: Response data

B.13.2 *POST*

Create a device.
Requires admin authentication header.

Requires JSON body:

- (**Required**) unitid: (string)
- (**Required**) houseid: (string)
- (Optional) name: (string)
- (Optional) longitude: (string)
- (Optional) latitude: (string)

Example Usage and Response

To create a unit called "kitchen" with the id "2c3ae842bfe7" in house with id "cflxjagpjqtyraff", send the following:

```
{
  "unitid": "2c3ae842bfe7",
  "houseid": "cflxjagpjqtyraff",
  "name": "kitchen"
}
```

Listing B.4: Sent data

If no unit exists with the specified id, this results in a response similar to the following:

```
{
  "unitid": "2c3ae842bfe7",
  "houseid": "cflxjagpjqtyraff",
  "name": "kitchen",
  "online": false,
  "sensors": []
  "location": null
}
```

Listing B.5: Response data

If the unit id is already in use, status code `409` is returned, along with a message describing what went wrong.

B.14 `/admin/units/{unit_id}`

This endpoint supports three verbs, GET, PATCH and DELETE.

B.14.1 *GET*

Get a unit.
Requires admin authentication header.
Requires no body.

Example Usage and Response

This endpoints returns information on the specified unit. If a unit with the specified id exists the response looks similar to the following:

```
{
    "deviceid": "2c3ae8011862",
    "houseid": "cflxjagpjqtyraff",
    "online": false,
    "sensors": [
      "temp-in",
      "temp-out",
      "humid-in",
      "humid-out",
      "co2"
    ]
}
```

Listing B.6: Response data

If the device does not exist, the API returns code `404`.

B.14.2 *PATCH*

Change information on a unit.
Requires admin authentication header.
Requires JSON body:

- (Optional) houseid: (string)
- (Optional) firmware-revision: (string)
- (Optional) hardware-revision: (string)
- (Optional) latitude: (number)
- (Optional) longitude: (number)

Example Usage and Response

To change the location of an existing unit, send the following:

```
{
  "latitude": "56.636662",
  "longitude": "9.815581"
}
```

Listing B.7: Sent data

If a modification is done, the response looks similar to the following:

```
{
    "unitid":"2c3ae842bfe7",
    "firmware-rev": "1.0.0",
    "hardware-rev": "2.0.0",
    "hardware-name": "HEXUnit",
    "sensors": [
        "temp-in",
        "temp-out",
        "humid-in",
        "humid-out",
        "co2"
    ],
    "settables": [
        "fan-in",
        "fan-out"
    ],
    "location": {
        "latitude": "56.636662",
        "longitude": "9.815581"
    }
}
```

Listing B.8: Response data

If the device does not exist, the API returns code `404`.

B.14.3 *DELETE*

Delete a unit.
Requires admin authentication header.
Requires no body.

Example Usage and Response

This endpoint logs deletes the specified device if it exists, by removing it from the database.
If the device does not exist, the API returns code `404`.

B.15 `/admin/units/flash`

This endpoint supports one verb: POST.

B.15.1 *POST*

Flash all units.
Requires headers:

- (**Required**) Auth-Token: (string)

- (**Required**) Content-MD5: (string)

Requires body: binary file.

Example Usage and Response

To flash all units, send the binary firmware file and its MD5-hash in the body and its MD5-hash in the Content-MD5 header.

B.16 `/admin/houses/{house_id}/units/flash`

This endpoint supports one verb: POST.

B.16.1 *POST*

Flash all units in a house.
Requires headers:

- (**Required**) Auth-Token: (string)
- (**Required**) Content-MD5: (string)

Requires body: binary file.

Example Usage and Response

To flash all units in the specified house id, send the binary firmware file in the body and its MD5-hash in the Content-MD5 header.

If the specified house id does not exist, status code `404` is returned.

B.17 `/admin/units/{unit_id}/flash`

This endpoint supports one verb: POST.

B.17.1 *POST*

Flash unit.
Requires headers:

- (**Required**) Auth-Token: (string)
- (**Required**) Content-MD5: (string)

Requires body: binary file.

Example Usage and Response

To the specified unit, send the binary firmware file in the body and its MD5-hash in the Content-MD5 header.

If the specified unit does not exist, status code `404` is returned.