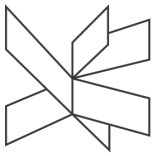

VIA University College

Large Object Detection Next to Waste Containers using IoT

PROJECT REPORT

Eimantas Sipalis 254018
Romans Kotikovs 252550

supervised by
Ib Havn

43749 characters (not including spaces)

Software Engineering, 7th semester
May 24, 2020

Document versions:

Version	Change	Date
0.1.0	Initial project report structure that follows ICT specific guidelines	2020/02/25
0.2.0	Start of versioning	2019/04/15

Contents

List of Figures

Listings

Abstract

The purpose of this project is to research and document findings of simplifying garbage collection from underground trash cans. This is a research paper rather than a development process of a “product to market”.

What are the main technical choices?

What are the results?

Used technologies: Microcontroller - ATMega2560, Ultrasonic Ranging Module HC - SR04, Servomotor, LoRaWAN, C, FreeRTOS, C# .NET Core 3.1.

Glossary

Here a list of terms used in this report can be found. It's also specified how these terms are emphasized in the text.

Terminology

Terms listed here are *italicized* when used in the text. They have the following meanings:

- *Use case* - a written description of how the task will be performed.

Acronyms, initialisms and abbreviations

The following terms are written in the same capitalization of the letters in the text as they are here.

- HAL - Hardware abstraction layer
- IoT - Internet of Things
- API - Application Programming Interface
- JSON - JavaScript Object Notation
- TCP - Transmission Control Protocol

1 Introduction

This introduction is based on the project description found in appendix **??**.

This project is conducted within the waste management industry. The stakeholder of this project is Runik Solutions that operates with Horsens commune who is the customer. It is a common problem for waste collectors, not only in Horsens, that people are leaving large and heavy objects such as household furniture, fridges, washing machines, etc. right next to the underground waste bins. This disrupts the usual waste collection workflow. Underground waste bins are one of the most advanced and used waste collection methods in Europe. That is why it is important to keep the waste collector workflow without impediments.

2 Analysis

This section in the report serves the purpose of outlining and explaining the project expectations.

2.1 Requirements

During the inception and elaboration phases, main requirements were established - functional requirements are expressed as use cases and non-functional requirements are stated in a list below. The use cases are extracted from a list of user stories.

2.1.1 User stories

1. As a kommune's dashboard system, it shall be notified about objects placed exceeding 0.4m width and 0.3m height on the trashcan's platform (coloured in red). The scan should be performed once a day. Items placed on top of the container do not need to be detected.
2. As a kommune's dashboard system, it shall be notified about previously notified objects that are no longer there.
3. As a kommune's dashboard system, it shall be notified about possibly malfunctioning (e.g. out of battery) devices.
4. As a system administrator, I should be able to register new devices and their corresponding address.
5. As a system administrator, I should be able to configure the time of the day when the scan should be performed.

2.1.2 Non-functional requirements

1. The notification should include the address, deviceEUI, timestamp and estimated object width (if possible).
2. The battery should last at least 2 years.
3. The project equipment should not interfere with the worker's usual way of cleaning out the underground containers.
4. The project should not damage the trash can's equipment (for example, drilling a hole through the trash can container's side is not allowed).
5. The delay of notifying the kommune's dashboard server after the large object detection scanning process should not exceed a 60 minute mark.
6. If the kommune's dashboard system is not responding to the notifications, the notifications should be resent at a later time until the kommune's dashboard system responds.

2.1.3 Use cases

Figure 1 shows the use case diagram of the system which displays the relationship between the actors and the use cases. It offers a high-level view which can be used as a basis for making sure that the system satisfies the requirements.

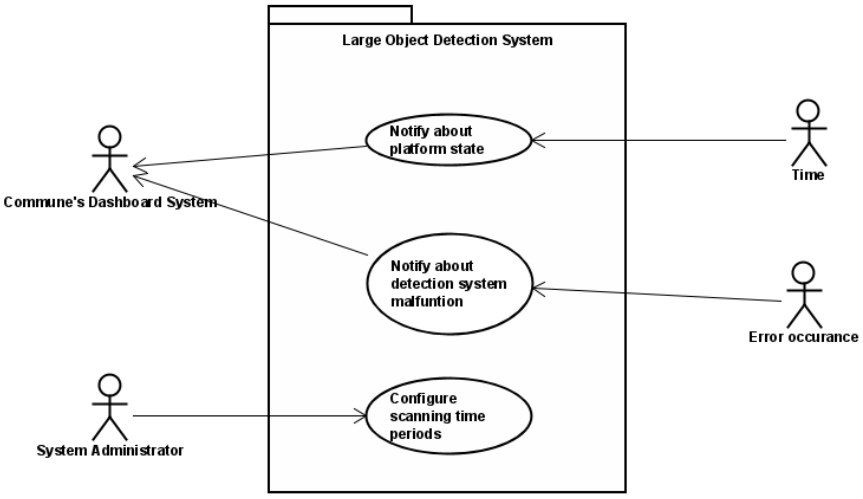

Figure 1: Use case diagram

Actor description

- **Kommune's dashboard system**
  An internal system used by the Horsens Kommune.
- **System administrator**
  An external person who manages the system.
- **Time**
  Triggers the scanning procedure.
- **Error occurrence**
  Triggers system malfunction and procedure of notifying about it.

2.2 Use case descriptions

The use cases are described in more detail in this section. First it is stated from which user stories the use case is formed from, then a brief use case description is given and a well detailed user description is shown in a corresponding figure.

2.2.1 Use case description: Notify about platform's state

Derived from user stories: 1, 2

Brief use case description: A person puts a large object on the underground bin platform. The system detects the object and notifies the Kommune's dashboard system where the object has been detected. See figure 2 for a detailed use case description.

ITEM	VALUE
UseCase	Notify about platform state
Summary	System detects the object and notifies the commune dashboard system where the object has been detected.
Actor	Commune's Dashboard System, Time
Precondition	No precondition.
Postcondition	Commune's Dashboard is notified about platform state.
Base Sequence	1. Object is placed on the platform. 2. Time triggers the scanning procedure. 3. System scans the platform. 4. System notifies about platform's state.
Branch Sequence	If Previously detected object is removed from the platform after next scan, send updated information to Commune's Dashboard.
Exception Sequence	3.1 System fails to scan the platform. 3.2 System notifies Commune's Dashboard about failed scan.
Sub UseCase	
Note	

Figure 2: Use case description for notifying about platform state

For the purpose of clarifying the use case scenario, an activity diagram is made which can be seen in figure 3

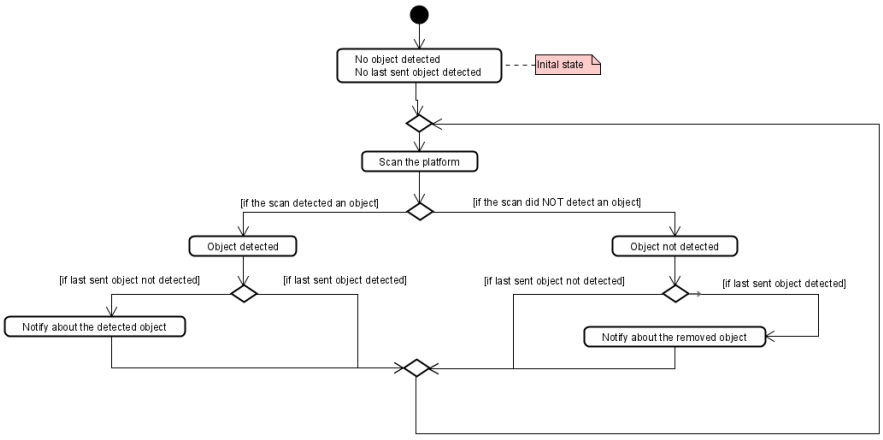

Figure 3: Use case description for notifying about platform's state

2.2.2 Use case description Notify about detection system malfunction

Derived from user story: 3

Brief use case description: If detection system is malfunctioning, e.g low battery, Kommune's Dashboard needs to be notified. See figure 4 for a detailed use case description.

ITEM	VALUE
UseCase	Notify about detection system malfunction
Summary	If detection system is malfunctioning, e.g low battery, Commune's Dashboard needs to be notified.
Actor	Commune's Dashboard System
Precondition	Detection system malfunctioning, e.g low battery.
Postcondition	Commune's Dashboard is notified about malfunction.
Base Sequence	1. Send daily heartbeat message. 2. Daily heartbeat message received. 3. System is working without malfunction.
Branch Sequence	
Exception Sequence	2.1 Heartbeat message not received. 2.2 Notify Dashboard about malfuntion.
Sub UseCase	
Note	

Figure 4: Use case description for notifying about system malfunction

2.2.3 Use case description Configure scanning time periods

Derived from user story: 4

As the trash collection time can change, scanning time period should too. Nodes should be remotely configurable by the system administrator.

2.3 Hardware and software investigation

This section server the purpose of describing the research process about the sensors and related software that was conducted to find out if the project requirements can be satisfied with today's technologies.

From the requirements the base functionality of the project can be stated - detect any large objects placed on the platform around the underground trash container. Figure 5 highlights the platform in red.

So any size-wise large (not necessarily heavy) object placed on the platform highlighted in red must be detected. There's no point in discussing the IT infras-

tructure of the project until a clear path to the solution of detecting a large object problem is described through the analysis process (proving that such functionality is possible). This analysis part will mostly revolve around the hardware - the sensor types and the sensor layout. While choosing the hardware it's important to consider the power consumption as the goal is for the system to operate on a battery.

Figure 5: Trash can container with platform highlighted in red

The trash cans come in batches of 4 (as far as it's been observed), however not all layouts are the same. Some are laid out in a line, others are in a square shape and sometimes they are placed on varying height level. Figures 6 and 7 indicate that.

Figure 6: Trash cans inline on different height levels

Figure 7: Trash cans in a square shape

Two main problems are raised from this analysis.

1. Sensor platform focus - Should the system optimise (reuse sensors) for a certain layout or focus on an individual trash can platform?
2. What sensors should be used?

No matter which sensor will be used in the final product it doesn't affect the sensor platform focus problem in any significant way. So the advantages, disadvantages and problems of focusing the sensors for a single platform or optimising sensors for multiple platforms will be described first.

2.3.1 Sensor platform focus

In this section the analysis of the sensor platform focus problem will be described. In the end, one approach will be chosen, with the reasons stated.

Optimising sensors for multiple platforms

The sensors can be placed in a way that allows the sensor to scan multiple platforms instead of just one. This is shown in figure 8.

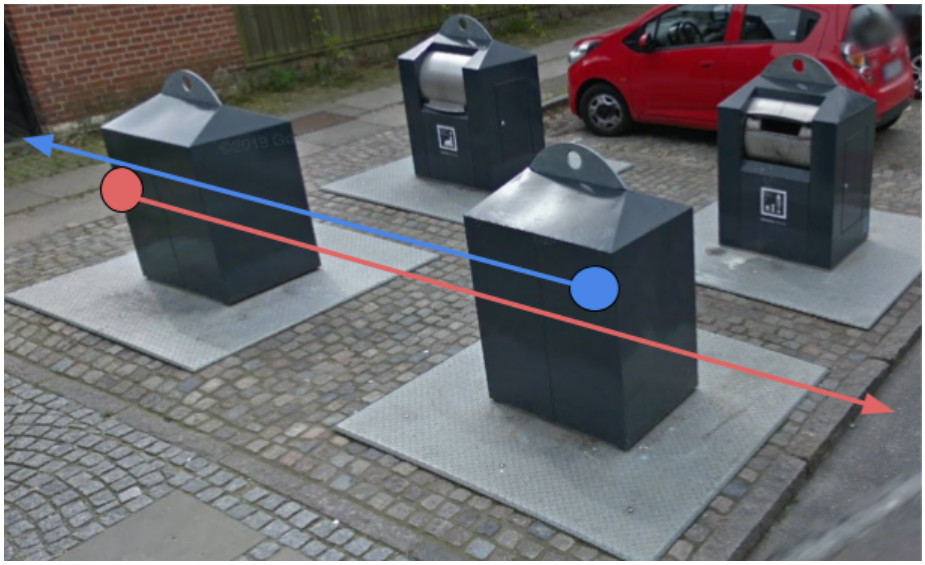

Figure 8: Sensors optimised for two trashcans

In the figure the sensors are indicated by the circles and they are facing the direction of the arrows. The sensors would measure the distance between the large object and the sensor and based on both of their measurements calculate the object's size. In the image above a single sensor set would be able to detect the large object of 2 platforms of one side. If the trash cans are in an inline layout a single set of sensors would be able to detect the object on 4 different platforms. See figure 9 for an example.

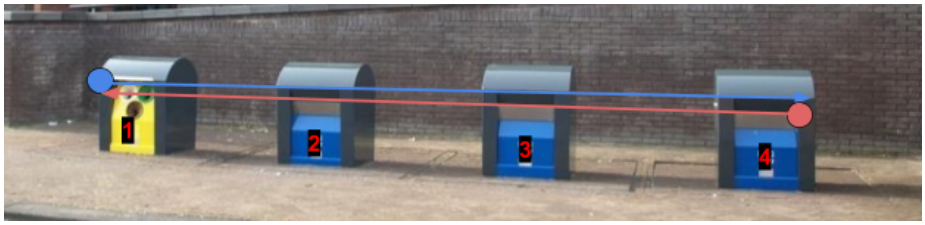

Figure 9: Sensors optimised for four trashcans

However this works only when there's only one object placed. If there's an object placed on platform 1 and another object on platform 3 (marked in figure 9) the system will think that the object is very large (which may or may not be true). Another issue arises in the case where an object is placed in between the platforms - objects placed in between the platforms should not be detected. While it's possible to calculate that the object is not on either of the platforms the object would disallow detecting other objects that are on the platforms as

indicated by figure 10. In the figure the large object is indicated by a red square. The large object is blocking sensors' lines of sight to see the other platforms. So the optimised sensor layout doesn't work greatly with multiple objects or objects in between the platforms, which is not such a common scenario, so this is not a huge disadvantage.

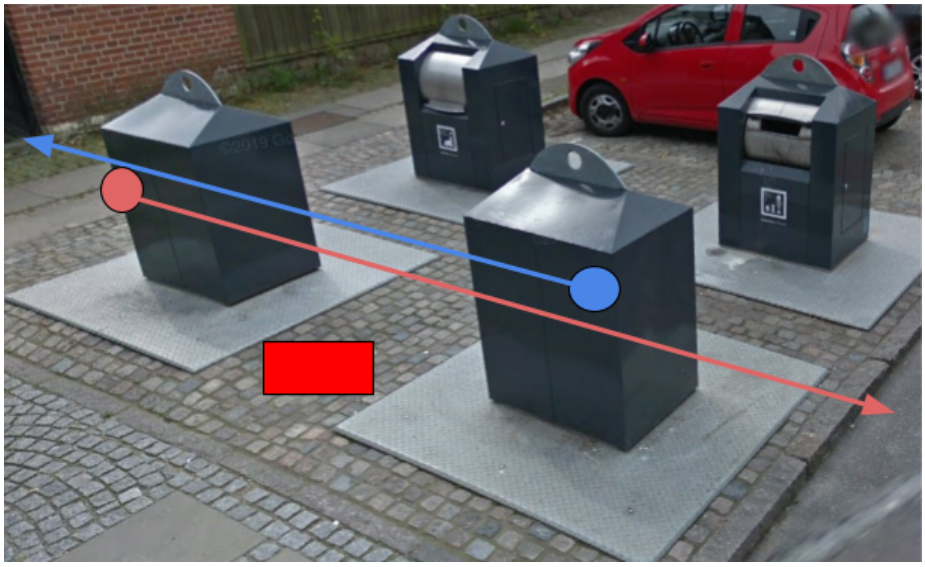

Figure 10: Object placed in between the platforms

However a big problem of this solution is scalability. There are a few different layouts and most importantly not all layouts have trash cans on the same height level. This would cause each setup to be different - sensors have to point at different angles and sometimes it would be even impossible to set up if the incline or decline is too high. While having one sensor set for multiple platforms is cost and power efficient, the project becomes increasingly more difficult because of all the new variables introduced by infinitely many different layouts.

Another disadvantage of optimised sensor placement is that if one of the sensors fails that's detecting for multiple platforms the detection system's functionality will be impacted for the whole trash can batch.

Focusing on an individual platform

All problems of optimised sensor layout can be solved by focusing on an individual platform:

1. The different layouts problem does not exist anymore because the focus is placed on a single platform so the layout of the platforms does not matter. Also from what was observed - an individual platform is always leveled out and not tilted. Even if there are any platforms that are tilted the platform is always going to be perpendicular to the trash can itself as shown in figure

11. This makes the solution way more scalable as it does not not depend on the batch layout. All the setups can be the same as the sensors can be placed on the trash cans which are always perpendicular to the platform or they can be placed on the platform itself as the platform is always in a straight line (even if the platform is tilted it is still going to be straight and not bent).

2. Multiple objects on different platforms or an object in between the platforms - is not a problem anymore since focus is placed on a single platform and the sensors are going to check only its boundaries.

3. Malfunctioning sensor - if one of the sensors fails it is going to disrupt only those platform's object detection and not the whole batch's.

Figure 11: Object placed in between the platforms

In conclusion, while using a single sensor for multiple platforms might cut down the hardware costs and increase power efficiency, it causes a lot of problems which would have to be solved individually for each different layout, which makes it extremely hard to scale the solution. Focusing on a single unit makes scalability easy - as the solution will be applicable for all different layouts. It is also likely that the savings in hardware costs would be offset by the amount of work hours needed to be done when setting up / maintaining the system because of all the different layouts. So focusing on a single platform is better for the project goals.

Note that here only the sensor layout is described, it is still possible to have the sensors focus on a single platform and have one central node that would handle the communication for the whole batch.

2.3.2 Sensor type

Since the problem for the sensor platform focus is solved, placing focus on choosing the optimal sensor type is a good idea. The following sensor types are going to be considered:

1. Ultrasonic sensor
2. LIDAR
3. Camera
4. Light array sensor

Sensor types that are the least likely to fit the project goals are described first, so they are eliminated as a possible consideration early.

Light array sensor

Light array sensors use 2 sensors (emitter and receiver) and an array of beams instead of a single beam - anything that comes in between the emitter and the receiver will be detected. This improves the detection rate by a lot compared to an ultrasonic sensor, especially for oddly shaped objects. However, its coverage area is still typically not wide enough to cover the entire side of the trashcan platform and its power consumption is way higher compared to a typical ultrasonic sensor - the Bulletin 45PVA–1LEB4–F4 has a detection width of 375mm with max power consumption of 155mA (Rockwell Automation, 2006). With these specifications the sensor already does not seem like a great choice because of the high power consumption, but more importantly because of the way it functions (requiring an emitter and a receiver) it forces the sensors to be placed in non optimal positions as indicated in figure 12.

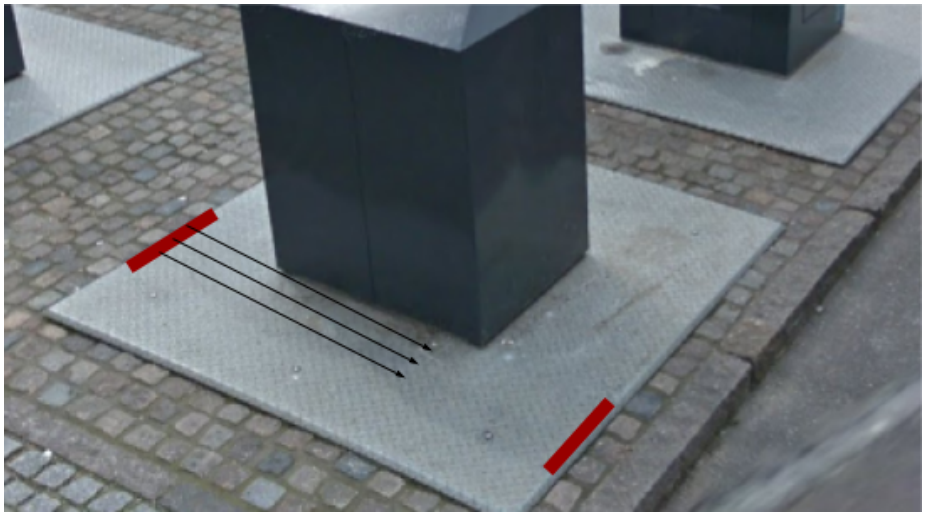

Figure 12: Light array sensor positioning

In the figure the sensors are marked in dark red. This positioning leaves the sensors very vulnerable to mechanical failures. Also it would not be easy to attach them there as they need to be elevated off the ground. In conclusion, looking at different light array sensors seems to indicate that light array sensors are more appropriate for more industrial projects like counting the number of produced objects in a production factory, where you also have access to a continuous power supply (Rockwell Automation, 2020) (Sick Sensor Intelligence, 2020).

Camera

Camera functions quite differently than other considered sensor types. The camera will be used to take images that will be analysed using computer vision to detect the object and their dimensions. However, it is not in the scope of the project to develop a computer vision solution from scratch, so a computer vision library must be used - OpenCV is one such library. From research, it seems that a good microcontroller choice would be ESP32 as it has good community support and a lot of modules available. There are quite a few projects online that are using ESP32, a camera module and computer vision to detect objects (Ayuso, 2018) (Ayuso, 2018).

However, these projects use ESP32 and the camera module to take pictures and then send them to a more powerful processing unit which does the work of computer vision. Although the mentioned projects are doing live recognition and this project needs to process only one or two images per day so the computational needs are way lower. Unfortunately even if it was possible to reduce the computational needs to a point where a battery powered ESP32 could last at least 2 years, people have not had success running OpenCV on it (albzn, 2019).

Another consideration is that the project members do not have a lot of expe-

rience working with computer vision. All in all, a camera module combined with computer vision is not an optimal choice as the computer vision libraries are not power efficient enough to run on the limited resources of the project and it's not in the scope of the project and the project members are not experienced in the computer vision area to develop or modify a computer vision library.

Lidar & Ultrasonic sensor

Lidar is a surveying method that measures the distance to a target by illuminating the target with laser light and measuring the reflected light with a sensor. Differences in laser return times and wavelengths can then be used to make digital 3-D representations of the target (Wikipedia contributors, 2020). Lidars tend to be expensive and usually sold in huge bundles of hundreds of thousands to big industries like car manufacturers. However, recently there has been development of smaller and cheaper Lidar modules that suit the needs of IoT (small module, low power usage):

1. tinyLiDAR - (MicroElectronicDesign, Inc., 2020)
2. LIDAR-Lite v3HP - (SparkFun Electronics, 2020)

LIDAR-Lite v3HP has a range of 40 meters and has field of view of around a $\frac{1}{2}$ degree - for distances lower than 1 meter the laser spread is the size of the lens and for higher distances the following formula can be used to calculate the beam diameter at a certain distance (SparkFun Electronics, 2018): Beam diameter = Distance / 100 (in whatever units the distance was measured). So even at the distance of 2 or 3 meters the beam diameter is very low - 2 or 3 centimeters. So LIDAR-Lite v3HP is not an optimal choice for the project needs.

However tinyLIDAR has a field of view of 25 degrees and the measurement distance from approximately 3 centimeters to 2 meters and a power consumption that's comparable to an ultrasonic sensor (24 mA average current during measurement, compared to 15 mA working current of the HC-SR04 ultrasonic sensor). Even with those specifications it's still difficult to find sensor placements that would be able to cover the entire platform and not use an absurd number of sensors. As an example the sensor is placed on top of the container pointing downwards to the platform as indicated in figure 13.

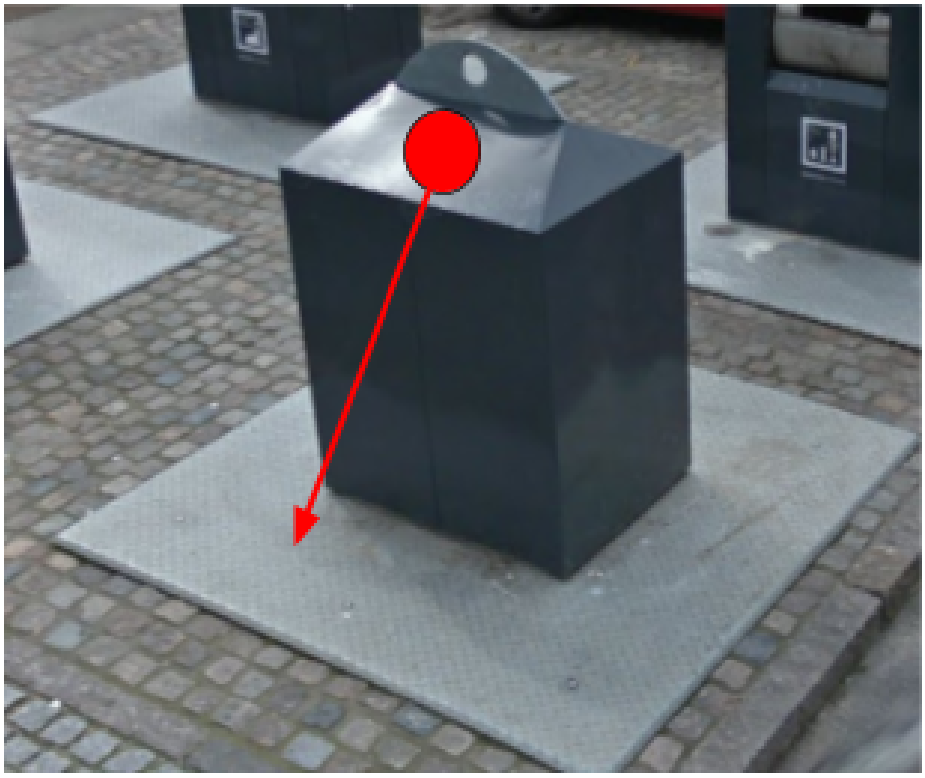

Figure 13: Single lidar positioning

Using a lidar resolution at distance calculator (georgehelser, 2018) the calculations that determine the maximum possible beam width at a certain distance can be performed (distance used is 0.5 meter as that's the distance between the sensor and a 0.3 meter height object placed on the platform) and shown in table 1

Scan field in degrees	**Distance in meters**	**Number of spots**	**Beam width in meters**
25	0.5	2	0.443
25	0.5	3	0.329
25	0.5	4	0.291
25	0.5	5	0.273
25	0.5	6	0.262

Table 1: Lidar resolution calculations

Even with the lowest possible number of spots chosen (2) the beam width is still only 44.3 centimeters at a 50 centimeter distance, since the platform is 160 centimeters wide, 4 of these sensors would be needed to cover one side. However if a servo motor is used for rotating the sensor more options become available. Ideally the servo motor with the sensor would be placed inside the container and the container sides would be drilled with holes. This way just one set of a servo motor and a sensor could do a full 360 degree spin and observe the whole platform. Unfortunately, drilling holes in the container sides is not allowed, so other options have to be considered. The second best option project members could think of was to place two separate motor and sensor sets on two opposite corners of the container. This allows one motor and sensor set to cover two sides of the platform. Figure 14 illustrates the sensor and motor set placed on the corner of the container and rotating and taking constant measurements to detect any objects. The motor will do a 180 degree turn to cover two sides of the platform. The same sensor and motor set is placed on the opposite corner of the container covering the other 2 sides of the platform.

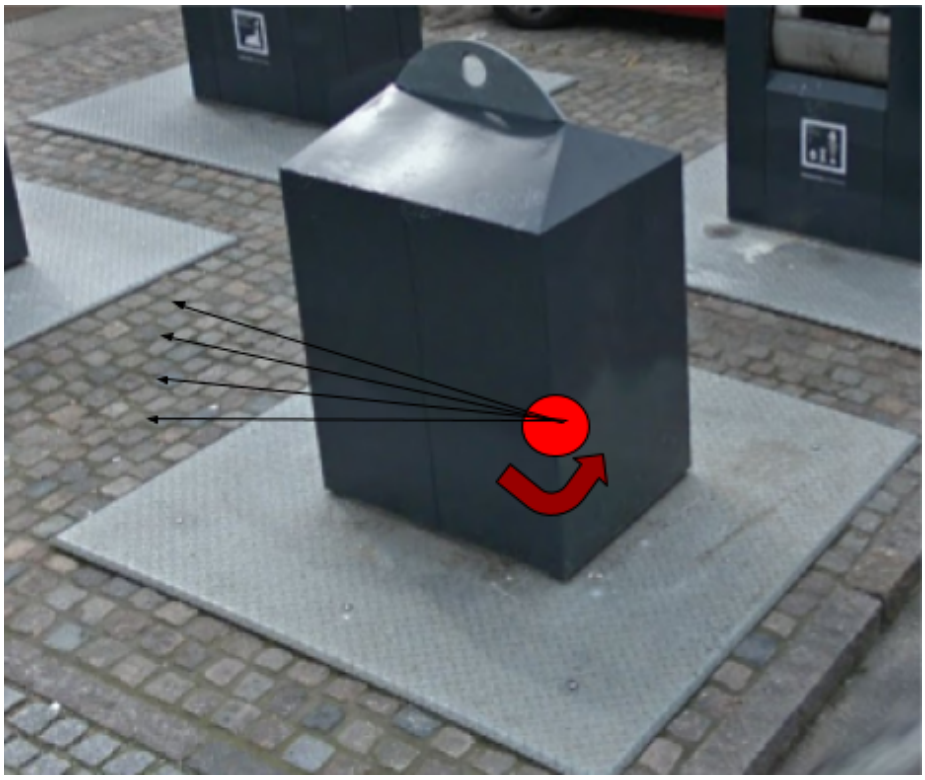

Figure 14: Rotating sensor

However, tinyLiDAR is not the only sensor that would work in this config-

uration, an ultrasonic sensor (specifically HC-SR04) would work really well too. On some level they can be compared as equal just with different specifications. tinyLIDAR has a higher sampling rate, bigger field of view, but it also has a higher power consumption. However they are also using different technologies - the tinyLIDAR uses infrared lasers and the HC-SR04 uses an ultrasonic sound - this difference affects how the sensors are impacted by the environment and how the emitted beams are reflected from objects. For now these differences will be ignored as this is a research project and if it turns out later that it would have made more sense to use the other sensor or perhaps a combination of both, the configuration can be updated easily as these sensors function similarly. Since the project team has access to the HC-SR04 ultrasonic sensor (tinyLIDAR has to be ordered and waited for to be shipped) and more experience working with ultrasonic sensors, the ultrasonic sensor will be the final choice.

2.4 Delimitations

Unless it has been stated otherwise, functionality is delimited due to time constraints.

The use cases for the system administrator actor were delimited. System administrator is responsible for configuring the detection system parameters, mainly the scanning times. However the need of such functionality is necessary only in production. As the purpose of this project is to find out if such detection system can be achieved, the configuration is left as a nice to have functionality.

In section 2.3 it was decided that the detection system will be focusing on an individual platform instead of trying to optimise a single sensor set for multiple platforms. On the other hand, it's possible to optimise the communication between the detection system nodes and the server. Since trash cans are placed in batches the system can be optimised by having a single "communication" node for the whole batch and a "detection" node for each trash can. The "detection" nodes would scan the platforms and send their results to the "communication" node through some low range communication protocol, after the "communication" node has collected the results of all the trash cans it will combine them into a single data set and send it out using the long-range LoRaWAN communication protocol. This way the long-range communication would be happening only once for the whole batch instead of once for every trash can in the batch. However, this kind of optimisation adds a lot of research to the project (like finding out which low-range communication protocol would be optimal for this project) and as the purpose of this project is to find out if minimum viable prototype is possible - this optimisation is out of the scope for the project.

2.5 Domain model

From the analysis above the following domain model entities are identified:

1. Trashcan's platform

2. Detection system
3. Kommune dashboard server

These entities are then modeled into a domain model diagram. Domain model diagram is shown in figure 15.

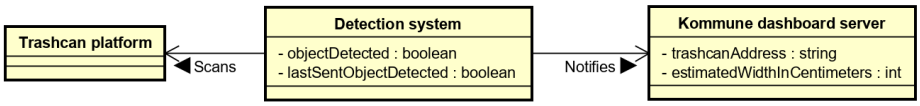

Figure 15: Domain model diagram

The domain model indicates that the detection system scans the trash can's platform for large objects and notifies kommune's dashboard server about any detected objects (or notify about the objects that were removed that were previously alerted about) with the trash can's address and an estimated object's width, measured in centimeters.

2.6 Technology choices

As the analysis of what the system has to do has been completed and a sensor set that would be able to detect objects placed on the platform has been found, now a list of main technologies will be listed in this section with a short description and an explanation of why this technology was chosen.

1. LoRaWAN - chosen as a network protocol for communication between the detection system attached to the trash cans and the server.
    - Low power - the system is running on a battery and has to last a long time.
    - Long range - there are trash cans everywhere so long range is a necessity.
    - The project team has some experience working with LoRaWAN.
2. ATMega2560 microcontroller - chosen as the microcontroller controlling the sensors of the detection system and communicating with the server.
    - Able to interface with the chosen sensor set.
    - Has a deep sleep mode for power efficiency.
    - There's a LoRa module available for this microcontroller.
    - The project team has experience working with this microcontroller.
3. C and FreeRTOS - chosen as the software stack for the microcontroller.
    - C is a general purpose, procedural computer programming language.

- FreeRTOS is a real-time operating system for microcontrollers.
- Both technologies have huge communities and are a very popular and well tested choice in the embedded world.
- Team members have experience working with both technologies and their combination.

4. C# .NET Core - chosen as the software stack for the application server
    - .NET Core is a general purpose cross-platform programming framework.
    - Teams members have experience working with it and it fits the goals of the application server well.

Perhaps these technology choices are not the most efficient ones, but as this is a research project and these technologies fit the project requirements they are the ones that will be used. If the product turns out to be a success they can be replaced with more efficient options.

3 Design

The following section describes the system design used to fulfill the functional and non functional requirements mentioned in the analysis. It is split up into the following sections:

- Section 3.1 describes the high level view of the system. It showcases how different parts of the system interact with each other.
- Section 3.2 describes the design of the detection system that's attached to the trash can container and is responsible for detecting large objects.
- Section 3.3 describes the communication between the different parts of the system.
- Section 3.4 describes the system designs of the application server and the simulated kommune's dashboard server.

3.1 Overall system design

For a high level view of the system and to showcase how different parts of the system interact with each other a block diagram was made, which can be seen in figure 16.

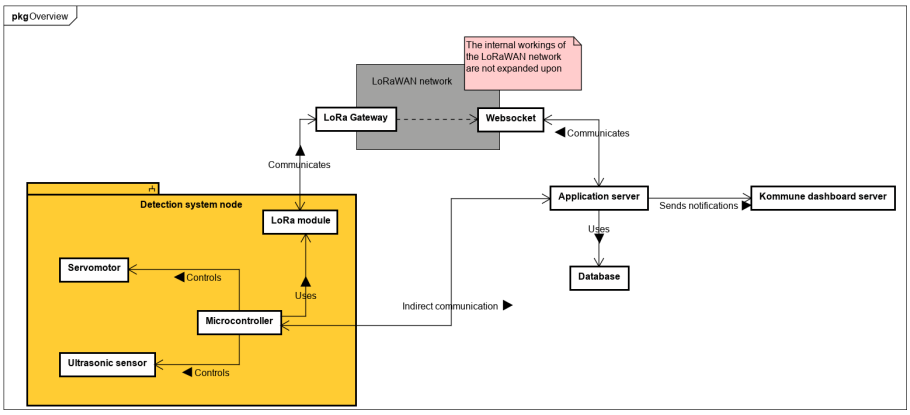

Figure 16: Overview block diagram

The system can be imagined as made of three different parts:

On the left side of the block diagram a "detection system node" subsystem can be seen, it represents the part of the system that's attached to each trash can and is responsible for scanning the trash can's platform for large objects and sending a notification about it. It consists of a microcontroller, a LoRa module and 2 sensor sets. The sensor set consists of a servomotor combined with an ultrasonic sensor. The microcontroller controls the sensor sets to scan the platform for large objects. Then it uses the LoRa module to send out a message containing the scan results to the LoRaWAN. An overview of the LoRaWAN architecture can be seen in figure 17. The detection system node would represent an end node in the mentioned figure. The end node communicates with a gateway, which forwards requests to the network server. The network server is configured to forward these messages to an application server that is controlled by the end node owner. In this project's case the messages from the network server are exposed through a websocket.

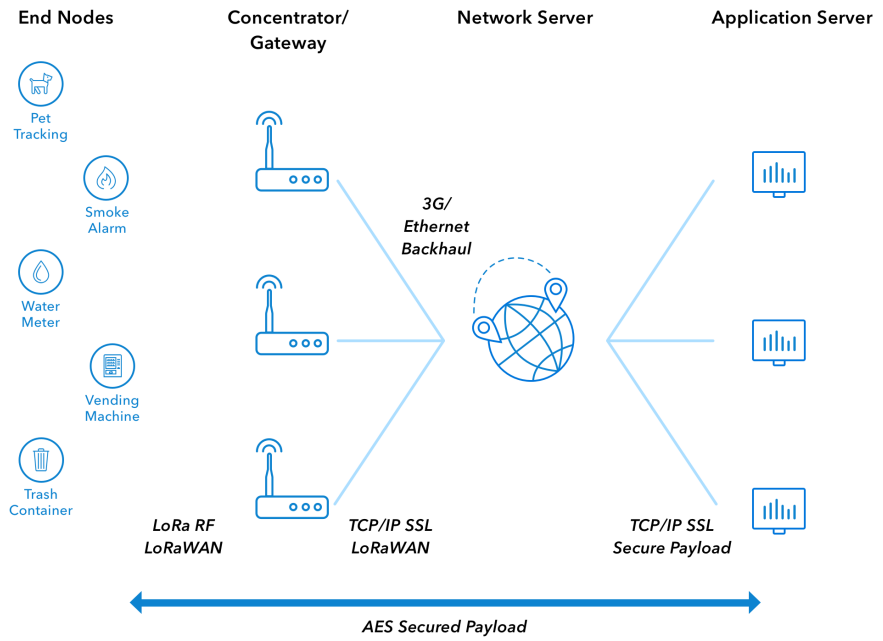

Figure 17: LoRaWAN architecture overview

An application server is basically just an application that will be running all the time and its main responsibility is to listen to the websocket and forward these messages to the kommune's dashboard server. It also exposes a few API endpoints for registering and configuring detection system nodes - to send these configurations to the detection system nodes it has to write to the websocket. For these requirements a database is necessary, which is used by the application server.

The last part would be a kommune dashboard server. Under normal circumstances, there would be a collaboration with the kommune's dashboard server's development team and the matter of what kind of communication channels to choose for transferring the messages would be disscused. However, no contact could be established with the Horsens kommune's contact person overseeing this project (most likely due to the COVID19 pandemic). As a consequence a simple kommune's dashboard simulation server will be implemented to represent the real server.

The application server can be considered as just a communication / translation layer between the detection system nodes and the kommune dashboard server. That's why the detection system node and the kommune dashboard server designs are done first and the application server design will be based on these designs.

3.2 Detection system node design

This section zooms in on the detection system node subsystem. First, the hardware choices will be described briefly. Following that, the subsystem design is shown and described, including chosen software technology choice explanations and what impact on the design they have.

3.2.1 Hardware

Atmega2560 microcontroller with VIA shield attached is used for the system. VIA shield provides a set of common peripherals - 8 switches, 8 LEDs, 7 segment display, temperature sensor, etc... Most of these peripherals are not necessary for this project, but it also provides simple connections for the servo motors and the LoRa module, which are used in this project. Parallax standard servo and HC-SR04 ultrasonic sensor are used as the sensor set hardware choices as they have simple interfaces, are simple to use, are very popular choices in the hobbyist community, are well tested and were readily available for the project members. The parallax standard servo is controlled using Pulse Width Modulation (PWM), so only a simple PWM pin is necessary. HC-SR04 is interfaced using 2 pins - Echo pin and Trigger pin. To interface the sensor in a sensible fashion the echo pin of the sensor must be connected to a pin of the microcontroller that allows interrupts (the reason is explained later in this section).

No circuit diagram was made as the connections between the microcontroller and the peripherals are pretty simple. The connections are as follows:

- Servo motors are connected to the connectors on the VIA shield that were made for controlling PWM devices.
- LoRa module is connected to LoRa module adapter that adapts the circuitry from Mikro Click Host module connector pins on the VIA shield to LoRa module pins.
- Ultrasonic sensors don't have dedicated pin connectors on the VIA shield, however VIA shield exposes PORTK pins of the microcontroller. Fortunately, PORTK pins allow interrupts, so the echo pin of the HC-SR04 can be connected to it.

3.2.2 Software

A class diagram for the system design is shown in figure 18. The system will use the C programming language, FreeRTOS - a real time operating system library and the LoRaWAN network stack. The following sections focus on describing the technologies with their impact on the design and some other decisions that impacted the design.

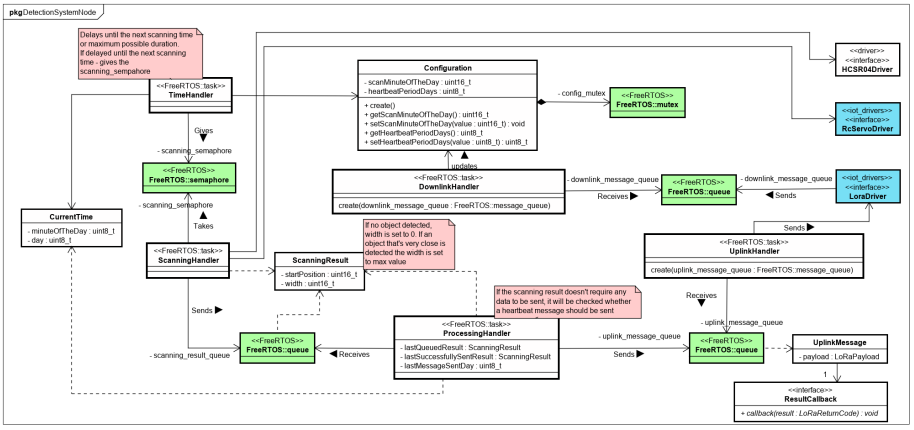

Figure 18: Detection system node class diagram

The C programming language

This subsystem is an embedded system, which makes C programming language a great choice for this system, as C is basically the de facto standard programming language for embedded applications. Atmel also provides an AVR compiler for compiling C code to AVR binary code, that's another plus for choosing C as Atmega2560 microcontroller is used in this subsystem. Even though C is not an object oriented programming language, some concepts from OOP paradigm can be borrowed and this is reflected in the design of the system.

FreeRTOS

The application has a lot of different tasks that need to be executed based on events or time - for this a real time operating system is used, in this case FreeRTOS. It allows the tasks to be executed based on events, synchronize the tasks and communicate between the tasks - make the application behave in a predictable manner. FreeRTOS offers a lot of concepts / interfaces for these purposes. The mentioned interfaces that are used in the design of detection system node subsystem are listed and their functionality and usage are described briefly:

- **Task** - FreeRTOS documentation itself describes a task in the following way. *A real time application that uses an RTOS can be structured as a set of independent tasks. Each task executes within its own context with no coincidental dependency on other tasks within the system or the RTOS scheduler itself. Only one task within the application can be executing at any point in time and the real time RTOS scheduler is responsible for deciding which task this should be. The RTOS scheduler may therefore repeatedly start and stop each task (swap each task in and out) as the application executes.* (FreeRTOS, 2020b). The system is designed in such a way that the whole program execution happens from FreeRTOS tasks. This allows the system to be built with separation of concerns in mind and schedule it easily in a predictable manner.

- **Semaphore** - semaphores are used in the subsystem for synchronization between the tasks. A task can pause its execution to wait for a semaphore (basically wait for a signal). Once the semaphore is given (the signal is given), the task execution continues.
- **Mutex** - mutexes are used for mutual exclusion, i.e. it can be used to protect a resource (e.g. a global field in the application) - make sure that the field is accessed only by one task at a time. Before using some resource that's protected by a mutex, the task should "take" the mutex, access the resource and "give back" the mutex. If the mutex is already taken the task execution will be paused until the mutex is given back by another task that had taken it.
- **Queue** - queues are used for communication between the tasks e.g. task A puts items in the queue and task B receives items from the queue. Queue can also be used for synchronization and communication at the same time e.g. if the queue is empty and task B sends out a command to receive an item from the queue (with an option specified to wait for items), task B execution will be paused until an item is put to the queue.

Drivers

The application is also using drivers to abstract the hardware usage, so no hardware registers are manipulated directly. As a simple example, if the LEDs are connected to PORTA of the microcontroller to change the state of a single LED instead of manipulating PORTA value directly, an LED driver is made that exposes a method for updating the LED states. The PORTA value manipulation is encapsulated inside the method implementation. In the case the LEDs are connected to a different port, only the method implementation needs to change and the methods calls can remain the same.

The microcontroller needs to control 3 different peripherals: an RC servomotor, LoRa module and an ultrasonic sensor. Fortunately, the VIA shield provides drivers (ihavn, 2020) for controlling the RC servomotors and the LoRa module. However, the ultrasonic sensor drivers have to be designed and implemented by the project members. The sensor can be used to take a distance measurement by following these steps (Elecfreaks, 2020):

1. Trigger pin must be set to high for at least 10 microseconds to initiate the measurement.
2. Then the ultrasonic sensor generates ultrasonic waves and sets the echo pin to HIGH.
3. The pin is set back to LOW once the reflected ultrasonic waves reach the sensor.
4. The time the echo pin was set to HIGH can be converted to distance using the following formula - distance = (HIGH level time * sound velocity) / 2

These instructions lead to a sequence diagram shown in figure 19.

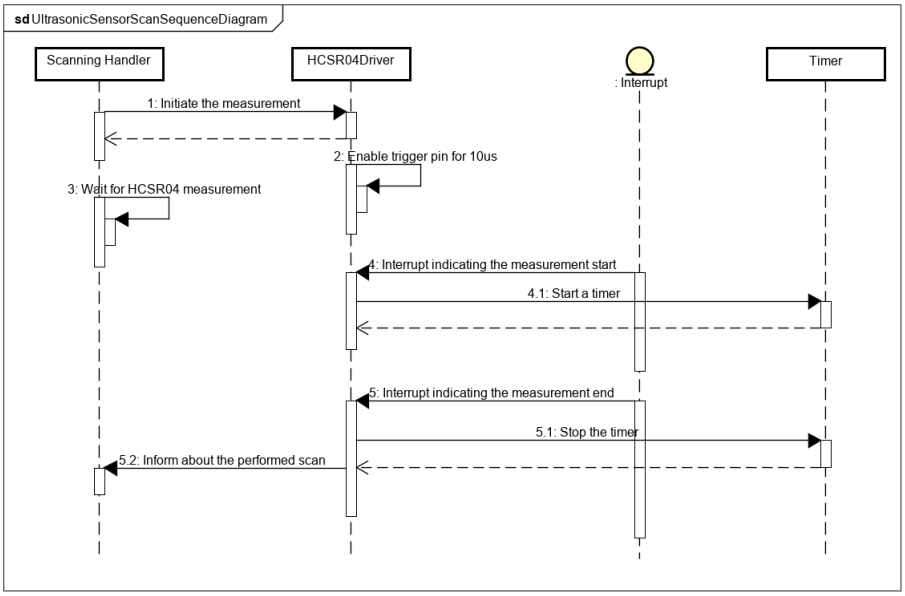

Figure 19: Ultrasonic sensor sequence diagram

The diagram actually illustrates not just how the HCSR04 driver works, but how it is used by the ScanningHandler task to perform a measurement. The measurement is initiated by calling a method in the HCSR04 driver from the ScanningHandler task. The driver will set the trigger pin to HIGH for 10 microseconds and then the ScanningHandler task execution will be paused until it gets informed about the measurement results. Once the ultrasonic sensor generates the soundwaves an echo pin will be set to HIGH, which will trigger the interrupt the driver is listening for. The driver will start a timer. Once the soundwaves are reflected back to the sensor, the echo pin is set to LOW, once again triggering the interrupt and now the driver stops the timer and informs the caller (ScanningHandler) about the elapsed time. The timer is used to measure how much time has passed. The only question that comes up when designing the driver's class diagram is how to inform the ScanningHandler task without having a direct dependency on the ScanningHandler from the driver. It is best if the driver has no such dependencies. For this purpose a callback function is used. With this decided, the driver's class diagram is made which can be seen in figure 20. The sensor can be initialized by calling the create method. The measurement can be initiated by providing the sensor number so the driver knows which sensor to use (as in the project there are 2 ultrasonic sensors) and a callback function, so it has a way of informing the caller about the completed measurement. When the driver informs about the measurement completion it provides the number of timer ticks passed and not the actual distance, as the callback will be called from

an interrupt service routine this keeps the interrupt execution as short as possible. Also it allows the driver to be used by different microcontrollers that have different timer / counter clock speeds (crystal frequencies). However, the driver interface also provides a utility function for converting from the timer ticks to distance by providing the timer speed (amount of timer ticks per second).

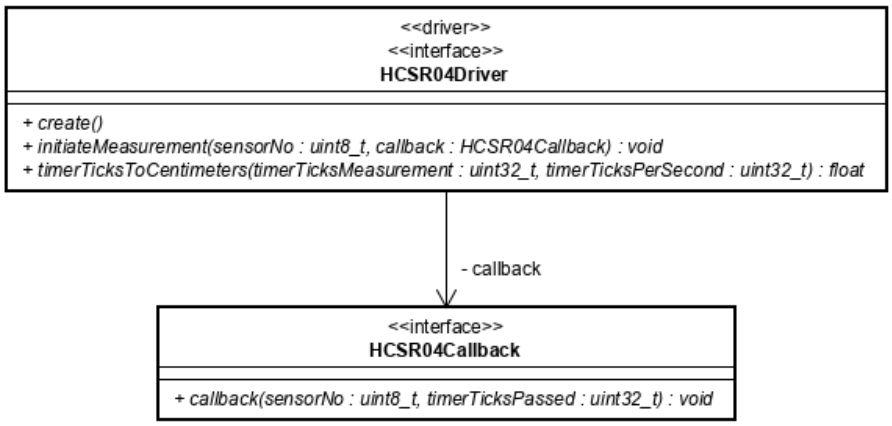

Figure 20: Ultrasonic sensor driver class diagram

Heartbeat messages & Acknowledgements

Since large objects are not placed on the trashcan platforms too often it was decided to send notifications only when the platform state changes (when an object is either placed or removed), with this decision there's no need to power up the LoRa module just to send the same state. However, there's also the requirement to report possibly malfunctioning devices - since messages are sent only for platform state changes, it is possible that the device won't send any messages for a very long time or even forever. To keep track if the device still works a heartbeat message must be sent periodically. As a consequence of this decision, the delivery of each message must be successful - for this message acknowledgements are used. If no acknowledgements were used, the new platform states might not get reported and the trash can platforms will not get cleaned up for a long time. The devices might also be registered as malfunctioning falsely - when the heartbeat message does not reach the gateway successfully.

Subsystem class diagram explained

Finally, after explaining the concepts that were used in the design of the detection system node application and how they impacted the design, the class diagram itself is explained. The TimeHandler task can be considered as the starting point of the execution, it starts up the scanning process. The execution flow is expressed in a list of numbered steps:

1. TimeHandler task checks the current minuteOfTheDay and pauses its own execution until the scanMinuteOfTheDay, that's set in the Configuration.
2. Once it wakes up it updates minuteOfTheDay and day fields to the correct values and gives a scanning semaphore and again puts itself back to sleep until the next scanMinuteOfTheDay (the value might have been changed by some other task since the last time it went to sleep, so it's not necessarily a 24 hour sleep).
3. Once the scanningSemaphore is given that wakes up the ScanningHandler task, which uses the RcServoDriver and HCSR04Driver to perform the scan (this process is explained in more detail later in this section) and then puts the scanning results (of type ScanningResult) into the scanning_result_queue.
4. The ProcessingHandler task is waiting for items in the scanning_result_queue, once an item (scanning result) arrives it checks if a notification should be sent:
   - If the new scanning result indicates a different result than the *lastSuccessfullySentResult* does, a message containing the scanning result is queued into the uplink_message_queue.
   - Else
     - If the device has not sent a message in a long time (IF lastMessageSentDay+ Configuration.heartbeatPeriodDays> CurrentTime.day) a heartbeat message is queued into the uplink_message_queue.
     - If the device has sent a message recently, no uplink message is queued up.

   If a message is supposed to be sent, it converts the *ScanningResult* into a *LoRaPayload* (which is provided by the VIA shield drivers) and queues it together with a callback function, which takes LoRaReturnCode as a parameter, indicating the sending process result. If the return code indicates a successful result, *lastSuccessfullySentResult* is set to *lastQueuedResult*, which holds the *ScanningResult* that was queued up last.
5. The UplinkHandler task is woken up if any messages were put into the uplink_message_queue, if so it uses the LoraDriver to send out a message and calls the provided callback function to report the sending result.

DownlinkHandler task is responsible for handling the received downlink messages. The task is sleeping until an item is put into the downlink_message_queue. Items are put in the queue by the LoraDriver when they are received by the Lora module from the gateway. The only downlink messages in the system are for updating the configuration. Thus, the DownlinkHandler updates the Configuration of the node whenever a downlink message is received. Alternatively, instead of DownlinkHandler updating the Configuration directly, a pending_configurations queue and ConfigurationHandler task could be created, and

whenever the DownlinkHandler receives a downlink configuration message it would put it in the queue for the ConfigurationHandler to handle. Configuration values are encapsulated by getters and setters - and these methods do not only deal with the field values, but they also ensure mutual exclusion by using a mutex.

An alternative design for the execution flow that starts from TimeHandler is to place all the execution in a single task as the execution is synchronous. This would remove the overhead of having multiple tasks and queues. However, the current design is much more flexible to changes - for example, if later the customer comes up with a requirement to send more uplink messages, the messages just need to be queued into the uplink_message_queue and no existing code needs to be modified. Also the overhead of the FreeRTOS kernel is minimal (FreeRTOS, 2020a).

The scanning process described in step 3 is explained in more detail in a sequence diagram shown in figure 21. The process starts out with waiting for the scanning semaphore. Then the scanning handler asks HCSR04Driver to perform a scan (this step was shown in more detail in figure 19). Then the execution enters a loop where a motor is rotated a few degrees and the task pauses its execution for some period of time to wait until the motor completes the rotation as there's no way to interact with the motor to know when it completed the rotation (the amount of time needed to wait will be found out in the implementation phase). Then the ultrasonic sensor is used to perform a distance measurement. The looping happens until the sensor set has scanned the whole platform side (270 degree rotation from the sensor). The reason there's a distance measurement performed before the loop is because the motor is in the starting position that still needs to be measured. It would be possible to put the scanning in the loop and the first cycle rotates the motor to a 0 degree position (instead of 0 + x, like now), but that would waste some time when waiting for motor rotation completion, even though it would be instantaneous as no rotation is actually necessary.

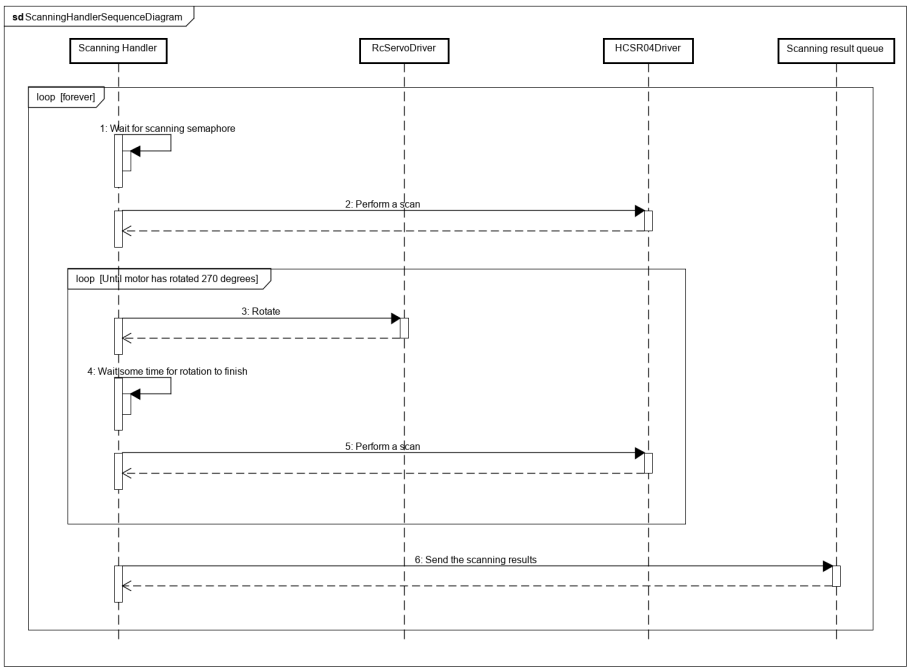

Figure 21: Scanning handler sequence diagram

When each distance measurement is executed, the motor position and the measured distance are saved. After the loop execution, all the measurements are analyzed and it's determined if there's an object placed and what is its estimated width. The analysis of the measured distances to determine the object's width is not that simple though as the distance from the sensor set to the edge of the trash can's platform keeps changing as the sensor set rotates as shown in figure 22. The outer highlighted rectangle represents the platform and the inner rectangle represents the trash can container. On the corner of the trash can container the sensor set is attached which takes distance measurements that are indicated by dashed lines. It can be seen that line *a* is shorter than line *b*, so it's not enough to check the measured distance against a constant, but it has to be checked against dynamic thresholds. One approach to determine these thresholds would be to surround the trash can platform completely with some objects (e.g. paper boxes) and then run the subsystem in a "setup" mode which would perform the distance measurements and save them. These saved measurements would be used as the distance thresholds as they indicate how far the platform's edges are from the sensor at a certain rotation angle. However, this approach has quite a few disadvantages:

- The setup process is quite tedious, requiring to surround the platform.
- It requires to implement the "setup" mode.

- Occasionally, there might distance measurement inaccuracies during the "setup" mode.

Because of these disadvantages another approach was looked into. The platform was looked at from a geometrical point of view. Distance *a* is perpendicular to the edge of the platform, by rotating the sensor set *x* degrees the distance to the edge is b now. Distance *a* value can be found by performing (160-60/2) operation as the trash can container is centered on the platform. Distances *a* and *b* with the edge of the platform make a triangle - this can be used to find distance *b* as distance *a* is known and 2 angles of the triangle are also known (x°and 90°). This can be used to find the distances of all the dashed lines by using the respective perpendicular lines that form a triangle with the respective edge of the platform.

After the analysis process, the results of the analysis (*estimatedWidth* and *startingPosition* - both are set to 0 if no large object was detected) are put into the scanning result queue.

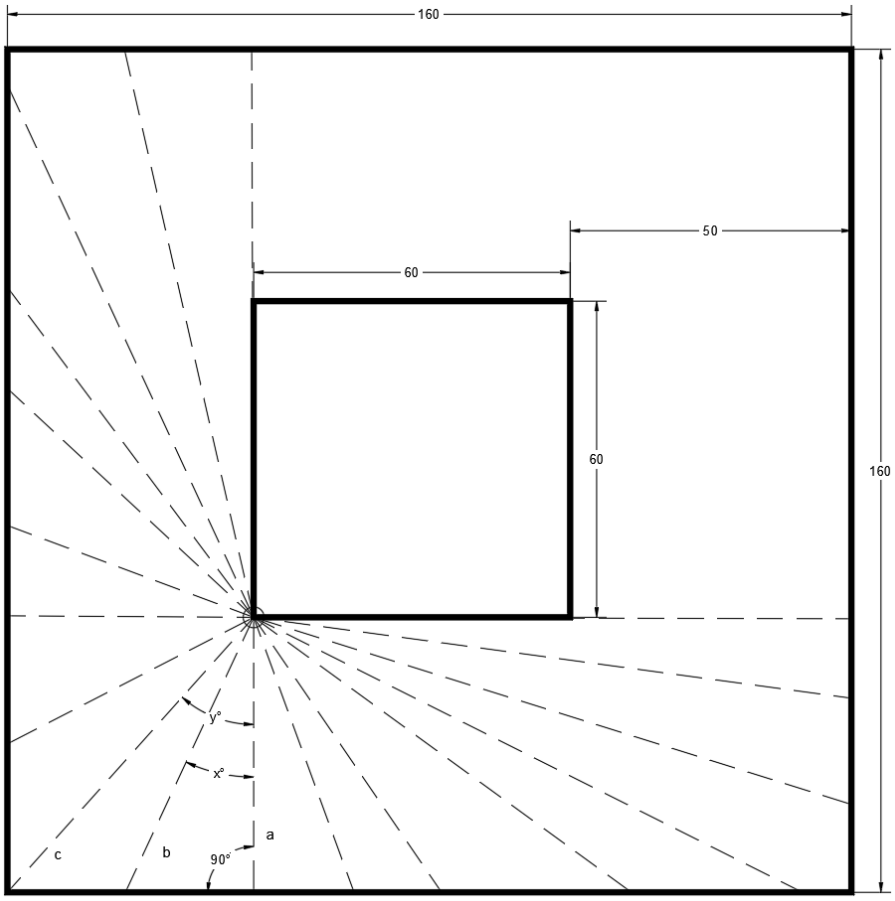

Figure 22: Trash can view from the top

Uplink, Downlink Messages and LoRaWAN

With the subsystem's class diagram designed and explained, the uplink (outgoing from the node) and downlink (incoming to the node) message structures are designed, which are shown in figure 23.

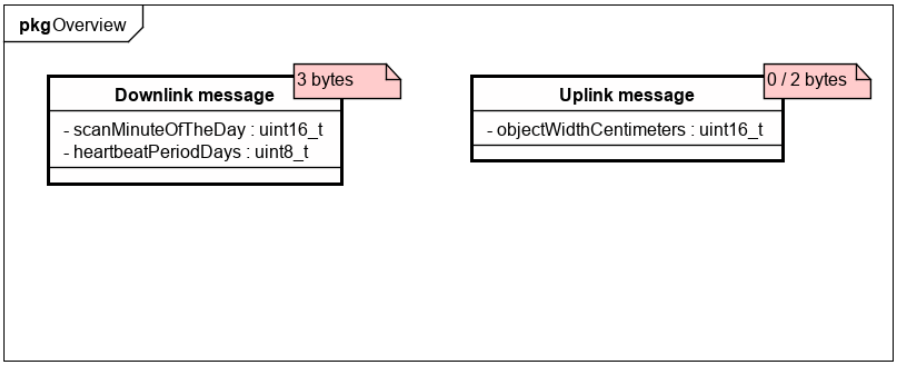

Figure 23: Uplink & Downlink message structures

The message structure highly depends on the system requirements, the design and the underlying LoRaWAN network. The network's functionality affects the structure of the outgoing and incoming messages - there's no need to include the message id, ACK field, device id, etc... as the network handles all of that. The message structures provided in figure 23 do not showcase the whole message that will be sent through the network, only the data (frame payload part). Full LoRaWAN message structure can be seen in figure 24.

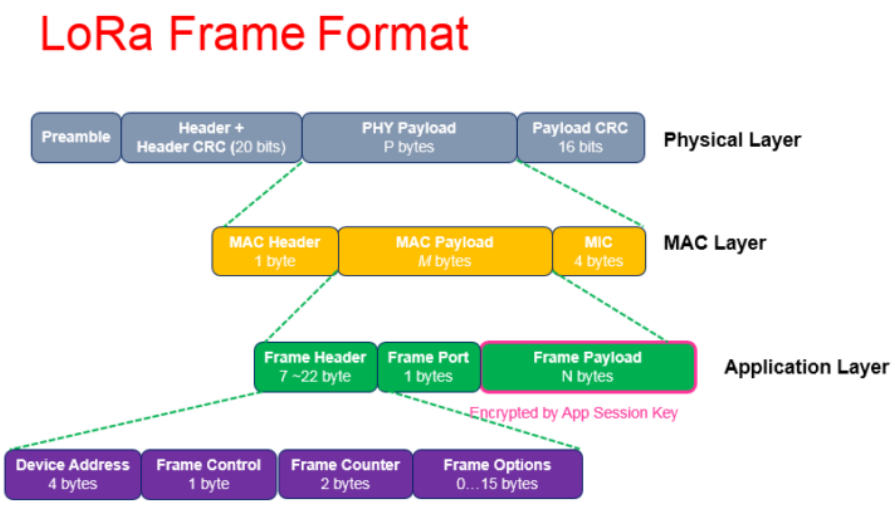

Figure 24: LoRaWAN frame structure

As mentioned in the beginning of the section, the notifications are sent only when the platform state changes or a heartbeat message should be sent. This means that the delivery of each message must be successful - for this message

acknowledgements are used. Fortunately, the underlying LoRaWAN network provides functionality to have message acknowledgements without manual implementation in the data part of the message. The design decision to use acknowledgements was made after analysing if the overhead of the acknowledgements is worth for the advantages it provides. To send messages that need to be acknowledged a correct message type must be specified. LoRaWAN messages can be of eight different types - the message type is set in the first three most significant bits of the MAC header. The types can be seen in figure 25. Confirmed data messages are the ones that require the acknowledgement.

MType Binary Value	**MType Decimal Value**	**LoRaWAN 1.0.3 Specification description**	**Plain English description**
000	0	Join Request	Uplink OTAA Join Request
001	1	Join Accept	Downlink OTAA Join Accept
010	2	Unconfirmed Data Up	Uplink dataframe, confirmation not required
011	3	Unconfirmed Data Down	Downlink dataframe, confirmation not required
100	4	Confirmed Data Up	Uplink dataframe, confirmation requested
101	5	Confirmed Data Down	Downlink dataframe, confirmation requested
110	6	RFU	Reserved for future use
111	7	Proprietary	Proprietary usage (ask me for suggestions of usage)

Message types as outlined in LoRaWAN™ 1.0.3 Specification, page 16, section 4.2.1
(final release, March 20, 2018, V1.0.3)

Figure 25: LoRaWAN message types

The uplink message needs to inform about the platform's state, if it has changed, and send a heartbeat message periodically. Uplink message is constructed based on the message type (not LoRaWAN message type). If the message is about the platform state change - it contains 2 bytes, which indicate the object's width in centimeters (integer value):

- If an object was removed the width field is set to 0.
- If an object was detected and its size CANNOT be estimated, the width field is set to max value.
- If an object was detected and its size CAN be estimated, the width is set to the estimated width.

If it's a heartbeat message, it contains no data. If the uplink message is sent as confirmed message a downlink confirmation message is received right after the uplink message. It doesn't require too much resources from the detection system node as listening for a message is not that resource intensive. However, if there are a lot of nodes that send out an uplink message that requires a confirmation, the network might not be able to send out all downlink confirmation messages because of the required maximum duty cycle of the fair use policy (The Things Network, 2020). Nevertheless, this is very unlikely to happen as the nodes send out uplink messages only when the platform state changes - this would require for a lot of the platform states to change on the same day.

The node receives downlink messages only for configuration updates. As a consequence, the downlink message fields match the configuration fields displayed in figure 18. Receiving configuration updates is also important, so acknowledgement for downlink messages is necessary too. It's also not too resource intensive as it just requires to set ACK flag to true on the next uplink message. The ACK flag is carried in the Frame Control byte of the Frame Header.

The uplink and downlink message data is encoded in binary to send the data efficiently.

3.3 Application server & Kommune dashboard server

This section focuses on the design of the kommune's dashboard server's endpoints and an overall application server design.

3.3.1 Kommune dashboard server

As mentioned in section 3.1, the kommune dashboard system was designed by the project team as no contact with the Horsens kommune's contact person overseeing this project could be made. The dashboard server created by the team will be a simple "simulation" server that will expose API endpoints for receiving the notifications and will just print them out into the standard output. The endpoints are designed accordingly to the requirements, the design can be seen in figure 26. From the requirements it can be seen that the user wants to receive 2 different types of messages:

1. **Object notification** - sent whenever the platform's state changes relating to large object existence. The message contains a property called *objectDetection* that is of enum type and indicates how the platform state changed - either an object was detected or the object that was previously detected was removed. If an object was detected and its size could be estimated, it will include *widthCentimeters* property.

2. **Device status notification** - sent whenever the detection system node's status changes. If the *deviceUnresponsive* boolean is set to true, it means the node is malfunctioning. It's possible that the device was blocked by some object (ironically) or because of the weather conditions or for some

other reason it was not able to send the heartbeat message and has been reported as malfunctioning, but since then the conditions have changed and the device sent the heartbeat message - in this case another *DeviceStatusNotification* message will be sent with *deviceUnresponsive* property set to false.

Both message types include *notificationId*, *timestamp*, *address*, *deviceEUI*. It includes both the *address* and the *deviceEUI* in case the device's address was entered incorrectly or the device has been moved since registering and the configuration has not been updated yet.

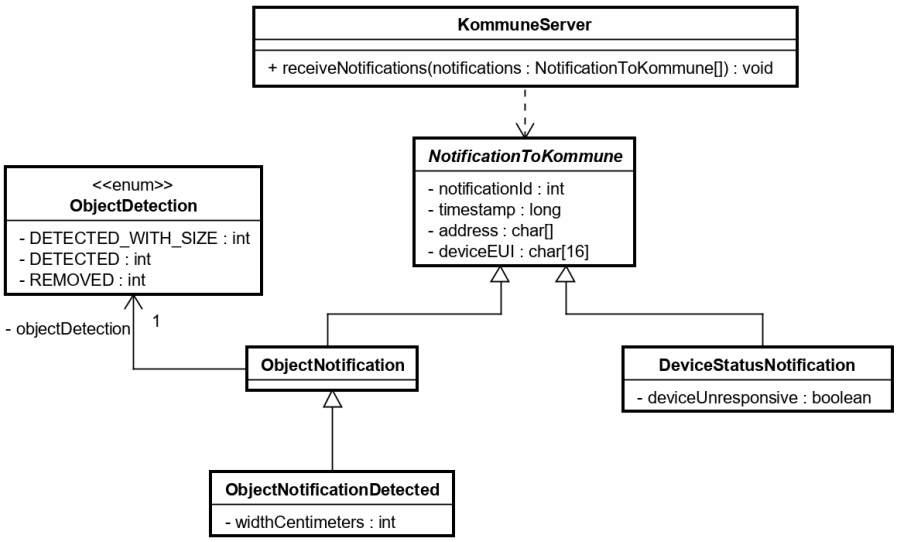

Figure 26: Kommune dashboard server endpoints & messages

There's only one endpoint and its functionality is to receive the notifications. It accepts an array of notifications instead of a single notification - this design allows to send multiple notifications at once, for example, report multiple malfunctioning devices at once. Nevertheless, this still allows to send a notification array with a single value inside of it.

3.3.2 Application server

The application server is designed last as it's mostly viewed as a communication / translation layer between the nodes and the kommune dashboard server, so its design is highly influenced by the design of those systems.

Database

As shown in figure 16 the application server is using a database, specifically a relational database - the design of the database is shown in figure 27. The database

is mainly used for storing the devices (and their respective statuses and configurations) and notifications. The main tables (*Device* and *Notification*) use auto generated int values as the primary keys. For example, *deviceId* is used instead of *deviceEUI*, as from the project members' experiences the user might enter the deviceEUI incorrectly and later will want to update it and updating primary key values can be error-prone. As such, it is preferred to keep the primary key values static since the row insertion. An entry in the device table represents a single detection system node. The device table contains the deviceEUI and the address - in a sense it's mostly used just for mapping from the deviceEUI to the address. The device also has a configuration associated with it and whether that configuration is pending. If the configuration is pending it means that the device has not yet acknowledged that it has received the configuration. The device also has a status associated with it, which contains a boolean field *deviceWorking* to indicate if the device is functioning and another boolean indicating whether the kommune dashboard server has received the latest device status. The database also stores notifications Database. Controller diagram. Uplink / downlink messages - websocket. Message sequence diagram. Timer sequence diagrams. Alternative design.

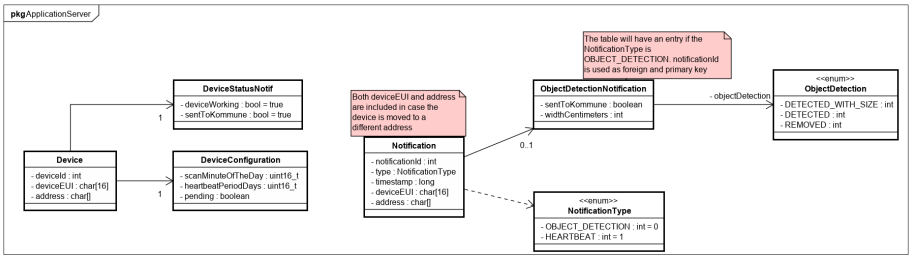

Figure 27: Database design

References

albzn, 2019. *OpenCV on ESP32*. [Online; accessed 15-April-2020]. Available at: <https://www.reddit.com/r/esp32/comments/bheh6j/opencv_on_esp32/>.

Ayuso, G., 2018. *OpenCV and ESP32: Moving a Servo With My Face*. [Online; accessed 15-April-2020]. Available at: <https://dzone.com/articles/opencv-and-esp32-moving-a-servo-with-my-face>.

Elecfreaks, 2020. *HC-SR04 Datasheet*. [Online; accessed 17-May-2020]. Available at: <https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf>.

FreeRTOS, 2020a. *FreeRTOS Overhead*. [Online; accessed 19-May-2020]. Available at: <https://www.freertos.org/FAQMem.html>.

— 2020b. *FreeRTOS Task*. [Online; accessed 18-May-2020]. Available at: <https://www.freertos.org/taskandcr.html>.

georgehelser, 2018. *LIDAR Resolution at Distance Calculator*. [Online; accessed 15-April-2020]. Available at: <https://precisionlaserscanning.com/2018/05/lidar-resolution-at-distance-calculator/>.

ihavn, 2020. *IoT drivers for VIA shield*. [Online; accessed 17-May-2020]. Available at: <https://github.com/ihavn/IoT_Semester_project>.

MicroElectronicDesign, Inc., 2020. *tinyLiDAR*. [Online; accessed 15-April-2020]. Available at: <https://microed.co/product/tinylidar/>.

Rockwell Automation, 2006. *Installation instructions PHOTOSWITCH Bulletin 45PVA Part Verification Array*. [Online; accessed 15-April-2020]. Available at: <https://literature.rockwellautomation.com/idc/groups/literature/documents/in/45pva-in001_-en-p.pdf>.

— 2020. *Light Arrays*. [Online; accessed 15-April-2020]. Available at: <https://ab.rockwellautomation.com/Sensors-Switches/Light-Arrays>.

Sick Sensor Intelligence, 2020. *Array Sensors | SICK*. [Online; accessed 15-April-2020]. Available at: <https://www.sick.com/dk/en/registration-sensors/array-sensors/c/g130174f>.

SparkFun Electronics, 2018. *LIDAR-Lite v3HP Operation Manual and Technical Specifications*. [Online; accessed 15-April-2020]. Available at: <https://cdn.sparkfun.com/assets/9/a/6/a/d/LIDAR_Lite_v3HP_Operation_Manual_and_Technical_Specifications.pdf>.

— 2020. *LIDAR-Lite v3HP - SEN-14599 - SparkFun Electronics*. [Online; accessed 15-April-2020]. Available at: <https://www.sparkfun.com/products/14599>.

The Things Network, 2020. *LoRaWAN Duty Cycle*. [Online; accessed 20-May-2020]. Available at: <https://www.thethingsnetwork.org/docs/lorawan/duty-cycle.html#maximum-duty-cycle>.

Wikipedia contributors, 2020. *Lidar — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-April-2020]. Available at: <https://en.wikipedia.org/wiki/Lidar>.