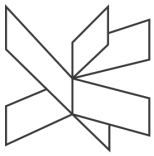

VIA University College

Large Object Detection Next to Waste Containers using IoT

PROJECT REPORT

Eimantas Sipalis 254018
Romans Kotikovs 252550

supervised by
Ib Havn

93834 characters (not including spaces)

Software Engineering, 7th semester
June 4, 2020

Document versions:

Version	Change	Date
0.1.0	Initial project report structure that follows ICT specific guidelines	2020/02/25
0.2.0	Analysis & Design sections finished	2020/05/27
1.0.0	Final version of the report	2020/06/04

Contents

List of Figures

Listings

Abstract

The purpose of this project is to envision a system that would improve the garbage collection process from underground trashcan containers. A common problem is that there are large objects placed on top of the platforms of the underground containers. The large objects disrupt the workflow of lifting up the platform to clean out the underground container. The goal of this envisioned system is to scan the platform for such objects and notify the kommune so the workers are aware of the impediments. The system has to be power efficient as it is running on a battery and able to communicate long distances using IoT technologies.

This is a research paper rather than a development of a "product to market". The designed prototype proved that such system is possible as the core functionality of the system was tested with positive results. If the system was to be implemented fully it could greatly improve the work flow of cleaning out the underground trash can containers by notifying the workers about possible impediments ahead of time. Used technologies are categorized and listed below.

Hardware technologies:

- Microcontroller - ATMega2560.
- Ultrasonic Ranging Module HC - SR04.
- Servomotor - Parallax servo motor.

Software technologies:

- C# .NET Core 3.1, ASP.NET Core, EF Core.
- C and FreeRTOS.

Networking technologies:

- LoRaWAN

Source code can be found in appendix 9.1 on page 76.

Glossary

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

Terminology

Terms that are listed here have the following meanings:

- *Use case* - a written description of how the task will be performed.
- Kommune - municipality in Denmark.
- Platform, scanning device and end node - used interchangeably, refers to the system that the monitored platform is equipped with. Example usage - "register platform", "register (scanning) device".

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.
- UP - Unified Process.

1 Introduction

The basis for this introduction is the project description found in appendix 9.2 on page 76.

Waste removal plays an important role in the way cities' infrastructure is handled in developed countries. It is very important for waste to be used in an efficient way with as less impact on the environment as possible, and Denmark has come far (Danish Environmental Protection Agency, 2020) in this regard where waste is collected in different containers based on their type. The goal is for the rate of recycling be the highest possible, but for that to happen, waste collection needs to be optimal with no obstacles in the way. There are three types of waste collection systems (Dr. Reinhart, 2020):

- Refuse Collection Systems - Household waste removed from the home.
- Commercial Waste Collection - Commercial waste removed primarily using dumpsters.
- Recyclable Material Collection - Collection of recyclable materials separated at the source of generation.

In Denmark, the third option is used on a large scale, where waste is separated at the origin. This presents a set of advantages such as better control over the waste and better management over the collection. However, collecting the waste can be faced with certain obstacles. The hereby project, conducted within the waste management industry, has the objective of improving the aspect of waste collection in Horsens, Denmark. The stakeholder of this project is Runik Solutions that operates with Horsens Kommune who is the customer.

The waste in Horsens is collected in underground waste bins, which are one of the most advanced and used waste collection methods in Europe. What makes them an excellent decision is the fact that they do not take a lot of space over ground, but they have a large underground size leading to fewer pickups. Also, they are enclosed with a double drum door, so no unpleasant odors get released.

Nevertheless, there is a downside to this solution, and that is the fact that people leave large and heavy objects such as household furniture, fridges, washing machines, etc. right next to the underground waste bins, or more specifically on top of the trash can platforms - this disrupts the workflow of lifting up the trash can platform to clean out the underground container, which is a common problem for waste collectors everywhere, not just Horsens. For this reason, it is important to keep the waste collector workflow without impediments, and therefore have no obstacles (i.e. foreign objects) on the platforms of the bins.

The purpose of this project is to design a system that scans the platform periodically and notifies the kommune's systems about the possible impediments.

This section is followed by the analysis section where the customer and the stakeholder requirements were captured and analyzed. Based on that, research was conducted and most suitable technologies for the project itself and its team

members were chosen. Then, the design section establishes the main system entities and describes their design and interaction with each other in detail. It is followed by the implementation section that showcases interesting snippets of the system's implementation that is based on the design of the system. Next, the test and result sections describe the outcome of the aforementioned design and implementation. The report ends with a conclusion section and a section for discussion about possible improvements of the project.

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 of unified process (Wikipedia contributors, 2020d), main requirements were established - functional requirements are expressed as user stories (which are later transformed to use cases) and non-functional requirements are stated in a list below.

2.1.1 User stories

1. As a kommune's dashboard system, it shall be notified about objects exceeding 0.4m width and 0.3m height placed on the trashcan's platform (highlighted in red in figure 1 on the following page) so that it can alert waste collectors about possible impediments. 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 so the workers can avoid false positive notifications.
3. As a kommune's dashboard system, it shall be notified about possibly malfunctioning devices on the platforms (e.g. out of battery) so that technician can fix or change the malfunctioning device.
4. As a kommune's dashboard system, it sometimes encounters system crash so the notifications from detection system should be resent at a later time until kommune's dashboard system responds so the kommune receives all notifications.
5. As a system administrator, I should be able to register new devices to their corresponding address so that kommune's dashboard system can receive alerts from new platforms.
6. As a system administrator, I should be able to configure the time of the day when the scan should be performed so waste collectors are notified with the newest information because waste collection time changes.
7. As a system administrator, I should be able to configure the "I'M ALIVE" message period so platforms that are not of crucial importance do not need to send messages often to save battery. The "I'M ALIVE" message period determines how quickly the malfunctioned devices are discovered.

2.1.2 Non-functional requirements

1. System should perform one scan a day.
2. The notification should include the platform's address, deviceEUI, notification arrival time and estimated object width (if possible).
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. The scanning device should operate on a battery.

Figure 1: Trash can container with platform highlighted in red (Elkoplast, 2020)

2.1.3 Use cases

Figure 2 on the next page 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.

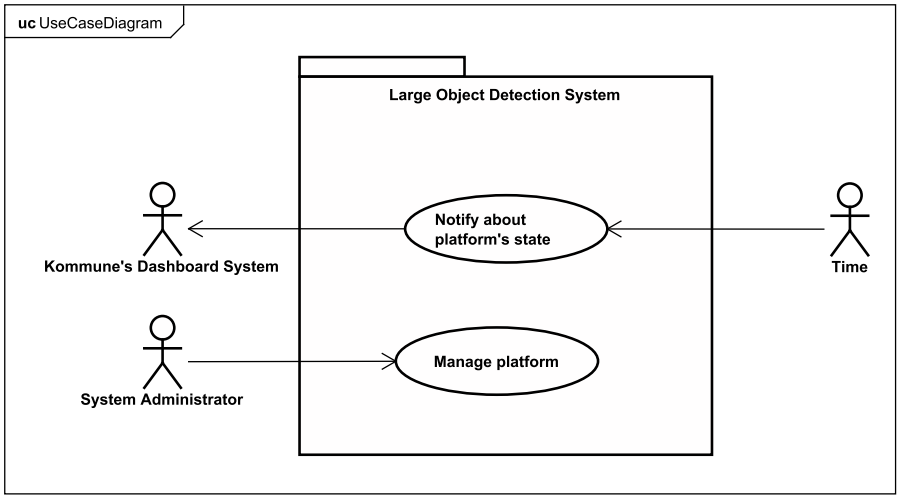

Figure 2: Use case diagram

Actor description

- **Kommune's dashboard system**
  An internal system used by the Horsens Kommune.
- **System administrator**
  Person who manages the system.
- **Time**
  Triggers the scanning procedure.

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 cases are formed from, then a brief use case description is given and a well detailed user description.

2.2.1 Use case description: Notify about platform state change

Actors: Time, Kommune's dashboard system.
Derived from user stories: 1, 2, 3, 4.

Brief use case description: A large object is placed on the underground bin platform. After some time the system scans the platform and notifies the Kommune's dashboard system where the object has been detected. If the previously detected object is removed from the platform before the next scan, updated information needs to be sent to Kommune's dashboard after the next scan.
Precondition: Platform state has changed (either an object was placed or removed since the last scan).
Post-condition: Kommune's dashboard system is notified about the platform state change.
Base sequence:

1. Time triggers the scanning procedure.
2. Scanning procedure detects a platform state change.
3. The system notifies kommune's dashboard server about the platform state change.

For the purpose of clarifying the base sequence of the use case an activity diagram is made which can be seen in figure 3 on the following page.
Exception sequence 1 (scanning device malfunctioning and cannot communicate):

3.1 Platform state change is not sent to the kommune dashboard server.

3.2 Detection system error.

3.3 The kommune dashboard server is notified about possibly malfunctioning device.

Exception sequence 2 (kommune dashboard server is not responding to the notifications):

3.1 Platform state change is not sent to the kommune dashboard server.

3.2 Kommune's dashboard system error.

3.3 The notification is resent at a later time to the kommune dashboard server.

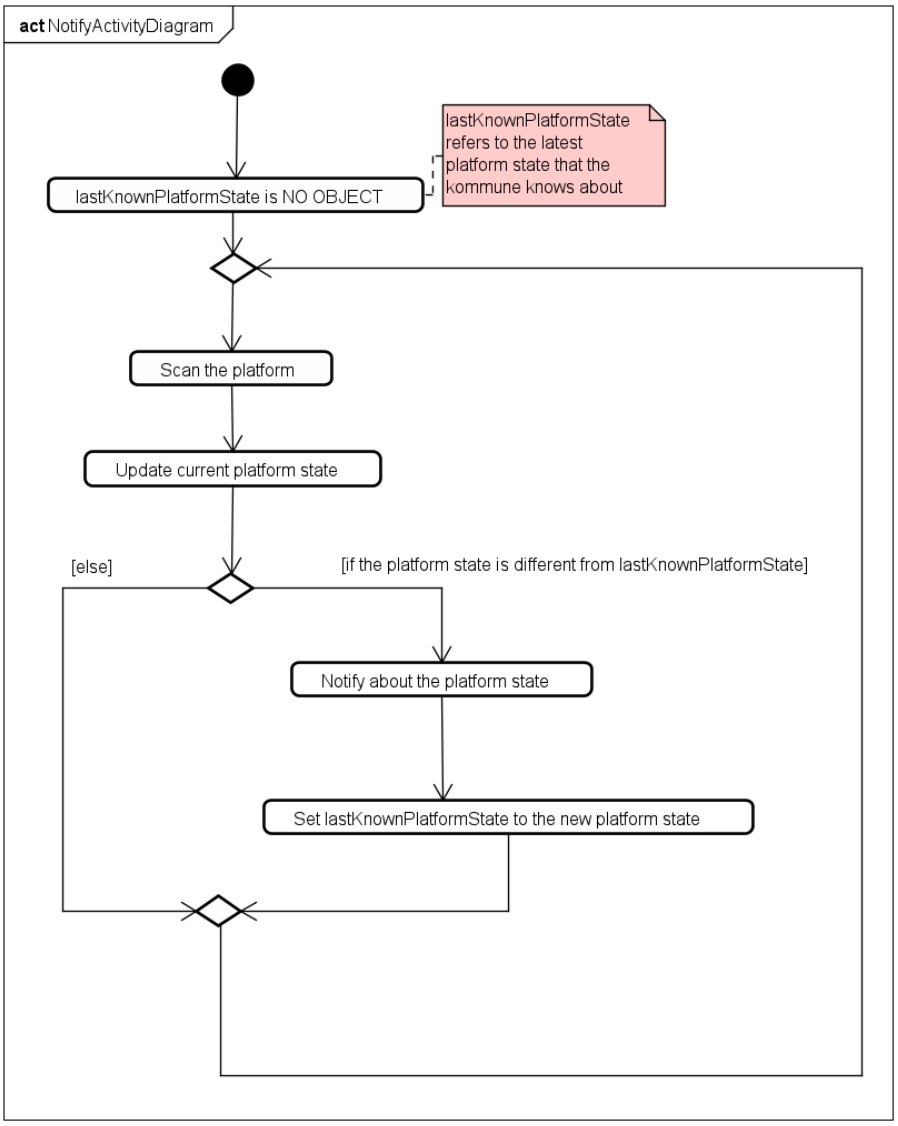

Figure 3: Activity diagram that represents the "sunny scenario" for notifying about platform's state change

2.2.2 Use case description: Manage platform

Actor: System administrator.
Derived from user stories: 5, 6, 7.
Brief use case description: The system administrator should be able to register new platform (device) and configure them later if needed. Devices should be

remotely configurable.
Scenario 1: Register platform
Precondition: The platform has not been registered before.
Post-condition: A new platform is monitored by the detection system.
Base sequence:

1. Register a new platform by providing the device eui, address and the initial configuration.

Scenario 2: Configure platform
Precondition: The platform has been registered.
Post-condition: New configuration is sent to the platform.
Base sequence:

1. Update platform's configuration by providing the new configuration.

2.3 Hardware and software investigation

This section serves 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 (platform is highlighted in red in figure 1 on page 10). 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 infrastructure 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.

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 4 and 5 indicate that.

Figure 4: Trash cans inline on different height levels (Google Maps, 2020)

Figure 5: Trash cans in a square shape (Google Maps, 2020)

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 6.

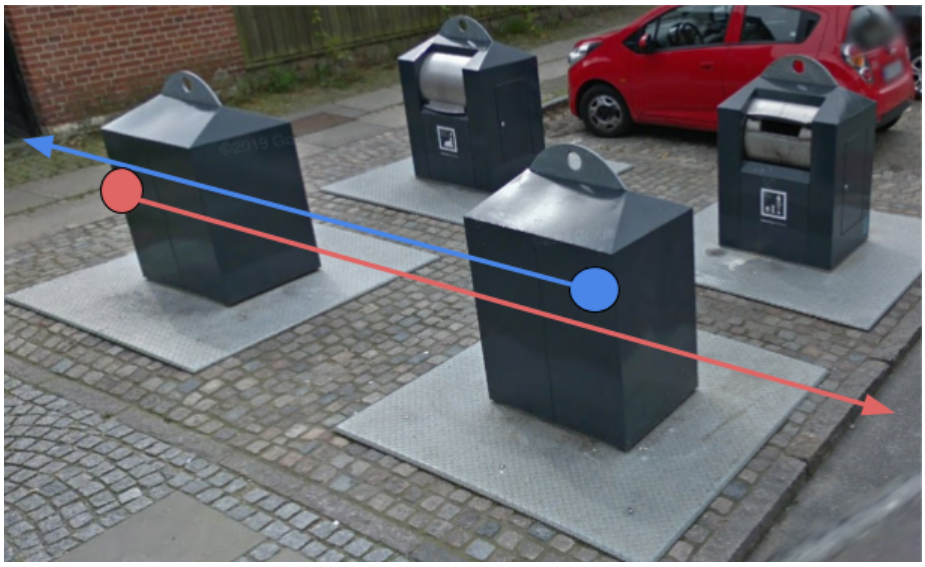

Figure 6: Sensors optimised for two trashcans (Google Maps, 2020)

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 7 for an example.

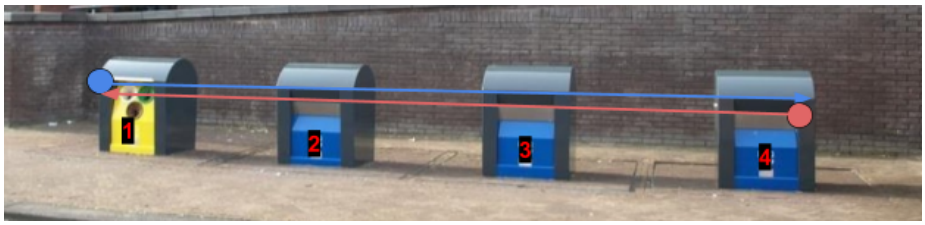

Figure 7: Sensors optimised for four trashcans (Google Maps, 2020)

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 7) 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 8. 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.

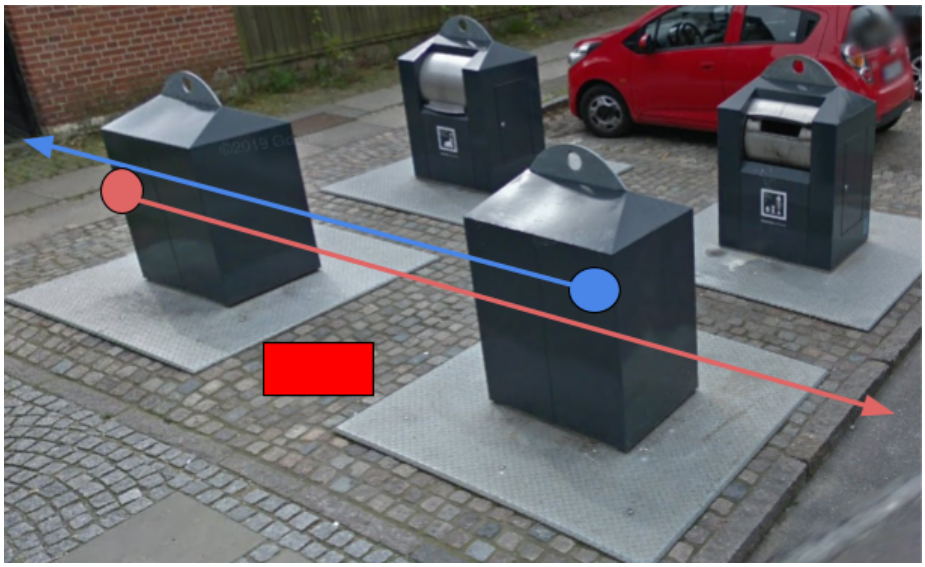

Figure 8: Object placed in between the platforms (Google Maps, 2020)

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 9 on the following page. 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 9: Object placed in between the platforms (Elkoplast, 2020)

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 10.

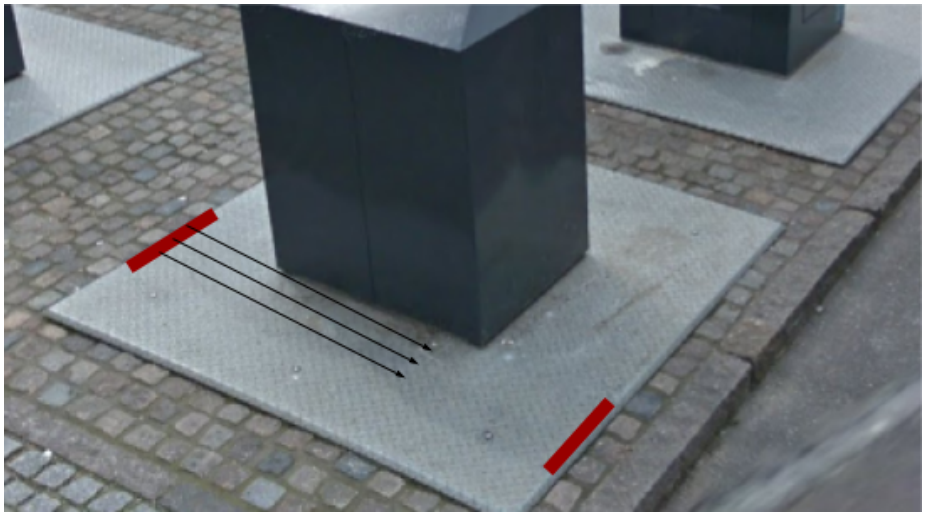

Figure 10: Light array sensor positioning (Google Maps, 2020)

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 experience 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, 2020b). 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 11.

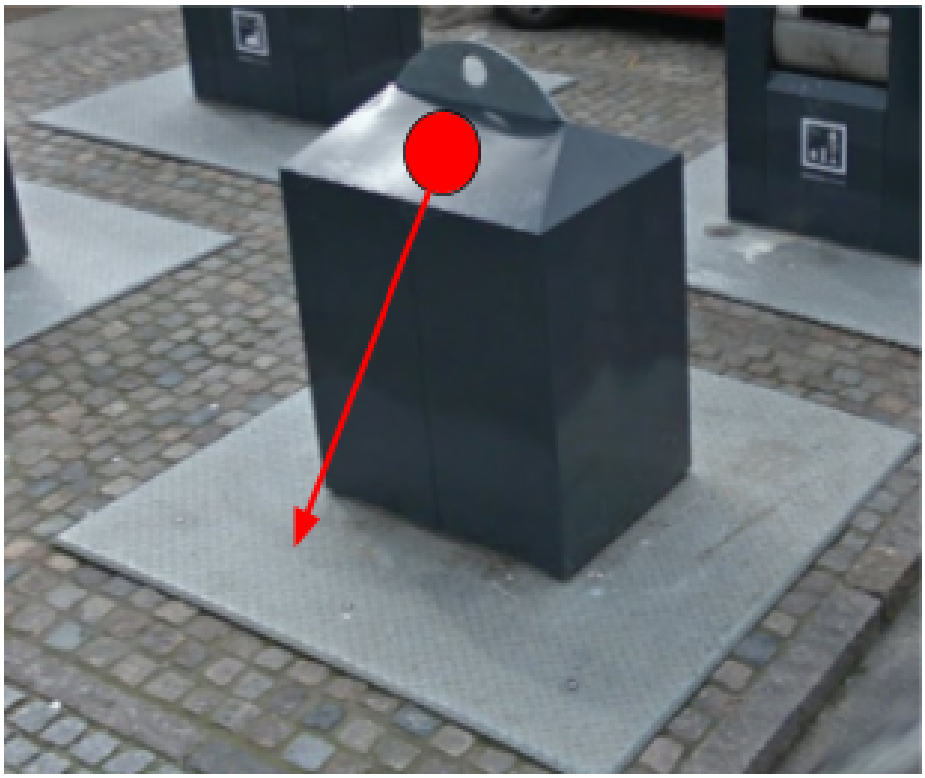

Figure 11: Single lidar positioning (Google Maps, 2020)

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 on the next page

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 12 on the following page illustrates the sensor and motor set placed on the corner of the container and rotating and taking constant measurements to detect any objects. The sensor set will take distance measurements every 5 or so degrees until the motor has rotated 270 degrees to cover two sides of the platform. The platform sides that will be scanned by the sensor set are highlighted in red and the sensor set measurements that are taken when the sensor set is rotating are indicated by the arrows. The same sensor and motor set is placed on the opposite corner of the container for covering the other 2 sides of the platform.

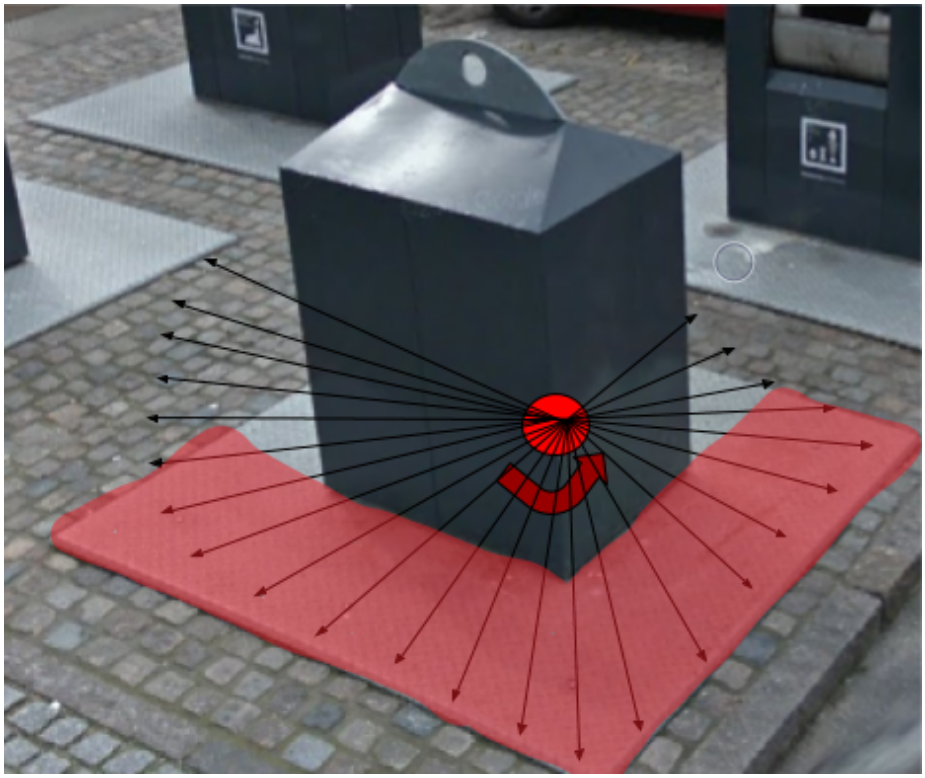

Figure 12: Rotating sensor (Google Maps, 2020)

However, tinyLiDAR is not the only sensor that would work in this configuration, 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.

Security and authorization are not considered in this project - anyone can use the register / configuration interface to register and configure the devices, you do not have authenticate as a system administrator. Same is for the simulated kommune dashboard server, anyone can send notifications to the kommune dashboard server. The security is not considered as this project serves only for the demonstration purposes.

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, which is shown in figure 13.

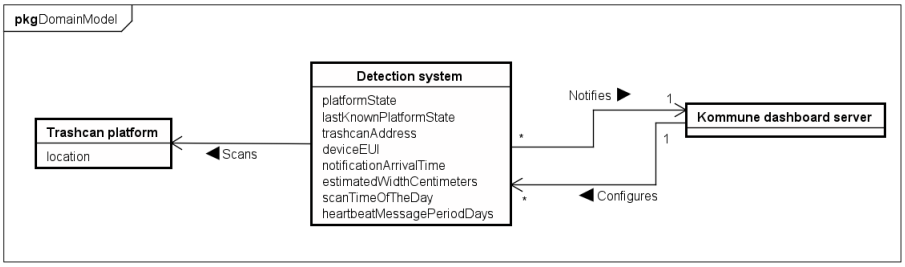

Figure 13: Domain model diagram

The domain model indicates that the detection system scans the trash can's platform for platform state changes and notifies kommune's dashboard server about any platform state changes (notify about newly placed objects or about the objects that were removed that were previously notified about) with the trash can's address, device eui, notification arrival time and an estimated object's width (measured in centimeters). The detection system is able to recognize when it should send a notification as it stores the last platform state value in *lastKnownPlatformState*. If the new platform state (*platformState*) is different than the *lastKnowPlatformState* it means a notification should be sent to the kommune dashboard server. The detection system can also be configured - its *scanTimeOfTheDay* and *heartbeatMessagePeriodDays* values can be changed by the system administrator. *ScanTimeOfTheDay* indicates at what time of the day

the scan should be performed. Since the detection system sends notifications only when the platform state change occurs, it might be possible that the state will not change for a very long time and no communication will be established - to make sure that the scanning device is still functioning a *"I AM ALIVE"* message is sent periodically, this period is set in *heartbeatMessagePeriodDays* field.

Kommunes dashboard server is mocked, because it was not possible to use actual system.

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 is 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 as the systems are designed in a layered way with low coupling between the layers allowing parts of the system to be replaced with ease.

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 on page 29 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 on page 43 describes the design of the simulated kommune dashboard server.
- Section 3.4 on page 44 describes the design of the application server and how it communicates with other subsystems.

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 14 on the following page.

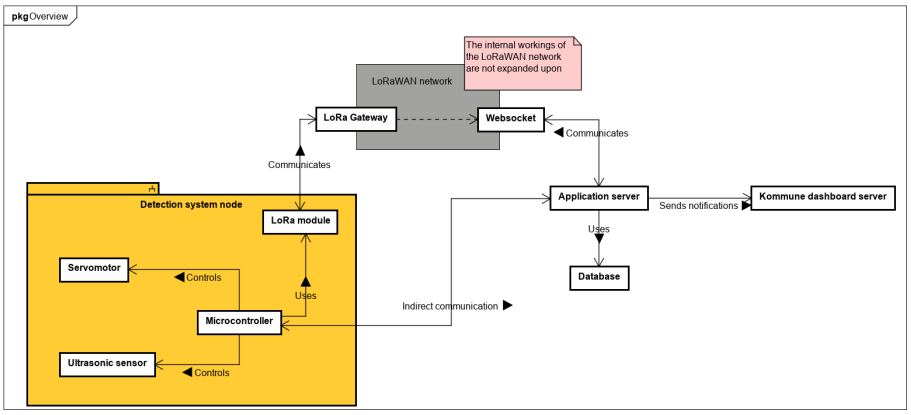

Figure 14: Overview block diagram

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 15 on the next page. 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.

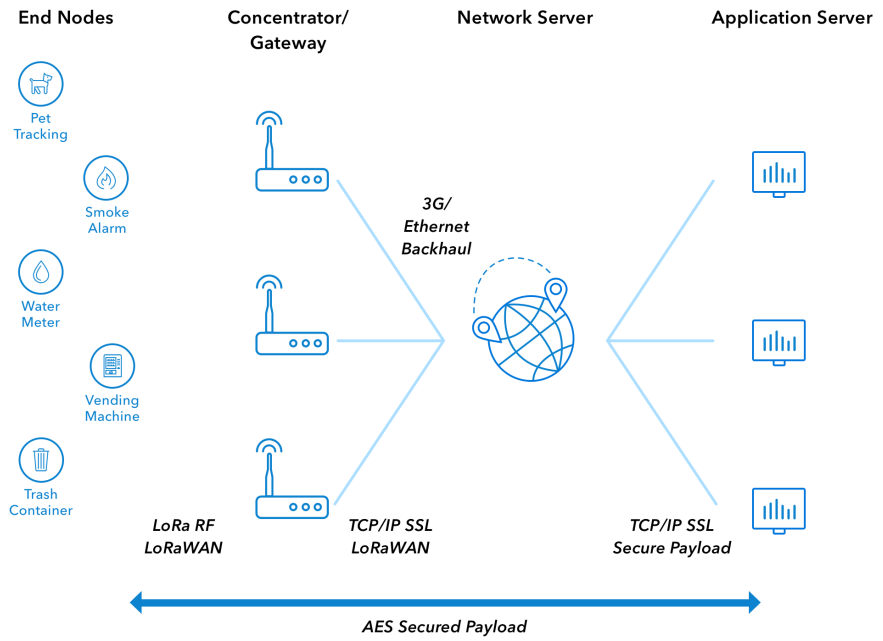

Figure 15: LoRaWAN architecture overview (The Things Network, 2020b)

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 is 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. All peripherals are connected to the Via shield and they are as follows:

- Servo motor 1:
  - GND - J13 PIN 1
  - VCC - J13 PIN 2
  - PWM - J13 PIN 3
- Servo motor 1:
  - GND - J14 PIN 1
  - VCC - J14 PIN 2
  - PWM - J14 PIN 3
- Ultrasonic sensor 1:
  - VCC - J902 PIN 16
  - GND - J902 PIN 15
  - ECHO - PORTK PIN 3
  - TRIGGER - PORTC PIN 1
- Ultrasonic senor 2:

- VCC - J903 PIN 16
- GND - J903 PIN 15
- ECHO - PORTK PIN 5
- TRIGGER - PORTC PIN 2

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.

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.

3.2.2 Software

A class diagram for the system design is shown in figure 16. The system will use the C programming language, FreeRTOS (FreeRTOS, 2020a) - a real time operating system library and the LoRaWAN (The Things Network, 2020a) network stack. The following sections focus on describing the technologies with their impact on the design and some other decisions that impacted the design.

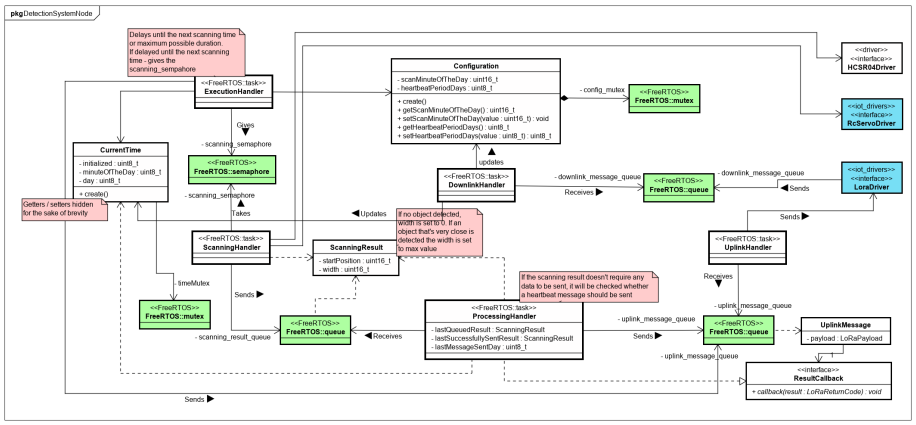

Figure 16: Detection system node class diagram

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 - the quoted text is italicized (FreeRTOS, 2020c). *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.*

  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, 2020a) 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 17 on the next page.

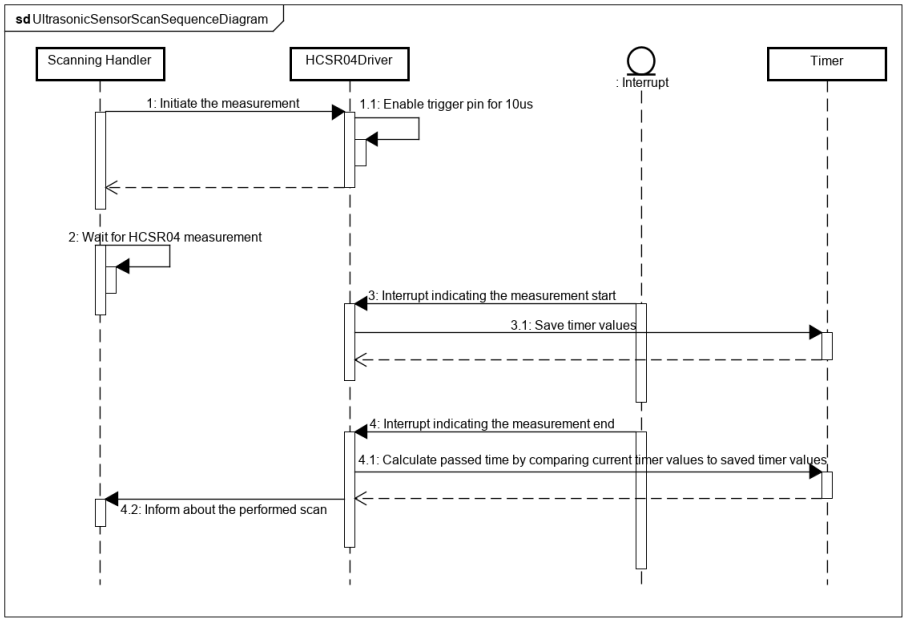

Figure 17: Ultrasonic sensor sequence diagram

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 (a sempahore can be used to wait for the distance measurements results and get informed about them). 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 save current timer value. 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 subtract saved timer value from current timer value (calculating how much ticks the timer performed between the interrupts) and informs the caller (ScanningHandler) about the elapsed time. HCSR04 driver uses a timer, that's running all the time - this allows to use multiple sensors at the same time with just one timer instead of allocating a timer for each sensor. 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 18 on the following page. The driver can be initialized by calling the create method - it takes the number of sensors that is going to be used. Then the sensors can be added by providing the sensor number and the

pins that the sensor is connected to. Before taking a measurement the *powerUp()* method has to be called that starts the timer. If the timer needs to be stopped (for power saving purposes) the *powerDown()* method can be called. 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 - it is preferred to keep the interrupt service routine as short as possible. This approach also allows the driver that acts like hardware abstraction layer to be used by different microcontrollers that have different timer / counter clock speeds (crystal frequencies). However, the driver interface also provides a utility functions for converting from the timer ticks to distance by providing the timer speed (amount of timer ticks per second) and for checking if the measurements is valid. The method that is checking if the measurement is valid will return *true* for values that indicate the measured distance to be lower than HC-SR04 sensor's maximum measurement range (400cm).

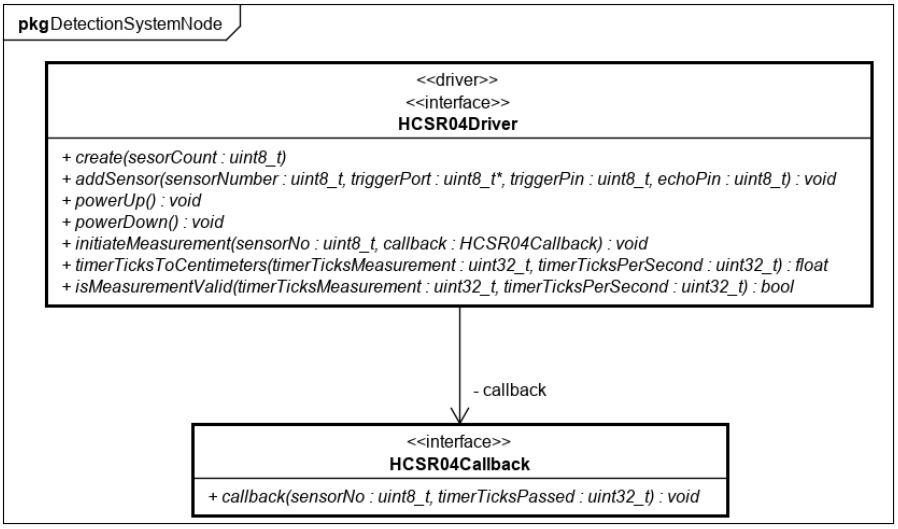

Figure 18: Ultrasonic sensor driver class diagram

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 ExecutionHandler 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. Execution task checks the current minuteOfTheDay and pauses its own execution until the scanMinuteOfTheDay, that's set in the Configuration. If the *initialized* field of the CurrentTime is set to *false* (it means the device is set up for the first time), the *ExecutionHandler* will queue up a heartbeat message to *uplink_message_queue*, which in turn will prompt the configuration to be sent from the gateway. The *DownlinkHandler* will receive the message, which includes the *timestamp*, which is used to update the *CurrentTime* values and set *initialized* to *true*.

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 (it means the platform state has changed), 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 + heartbeatPeriodDays > currentDay) 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 (message structures are explained later in the section). 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. Everytime a message is received the *CurrentTime* fields are updated to correct values using the *timestamp* value of the received downlink messages. This allows to keep the time synchronized if any offset is experienced from inaccurate microcontroller crystal. It also gives the possibility to synchronize the *minuteOfTheDay* when time is changed to summer or winter time. 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 ExecutionHandler 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, 2020b).

It is not shown in the diagram for the sake of brevity, but all tasks take relevant parameters when initialized. The parameters consist of the references that the entity has to other entities except for drivers. For example, the *ExecutionHandler* task will be initialized with references to *CurrentTime*, *Configuration* and the scanning semaphore objects. Whereas, the *ProcessingHandler* task will be initialized with references to scanning result queue and uplink message queue.

This makes the tasks more flexible to changes as their references are not hard-coded.

The current design is well layered - the HAL allows to change hardware connections and the implementation needs only minimal changes - either to update the driver or the way the driver is initialized if the driver allows configuration during initialization. It is also modular, the modularity allows to replace parts of the system with ease. For example, if a different sensor set is to be used for scanning the platform, only the *ScanningHandler* task implementation has to be changed, the rest of the system would stay the same. Similarly, if a new feature is requested that requires to send uplink messages, new task(s) can be created that would enqueue elements into *uplink_message_queue*.

The scanning process described in step 3 on page 35 is explained in more detail in a sequence diagram shown in figure 19 on the following page. 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 17 on page 33). 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 in steps of around 5 degrees). 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. Wasted time is not the biggest concern here, but it is mostly done for better power consumption optimization. For the same reason, after the scan is performed the sensor set is not rotated back to its original position, but instead the next scan will be performed in a reverse direction (going from 270 degree position to 0 degree position).

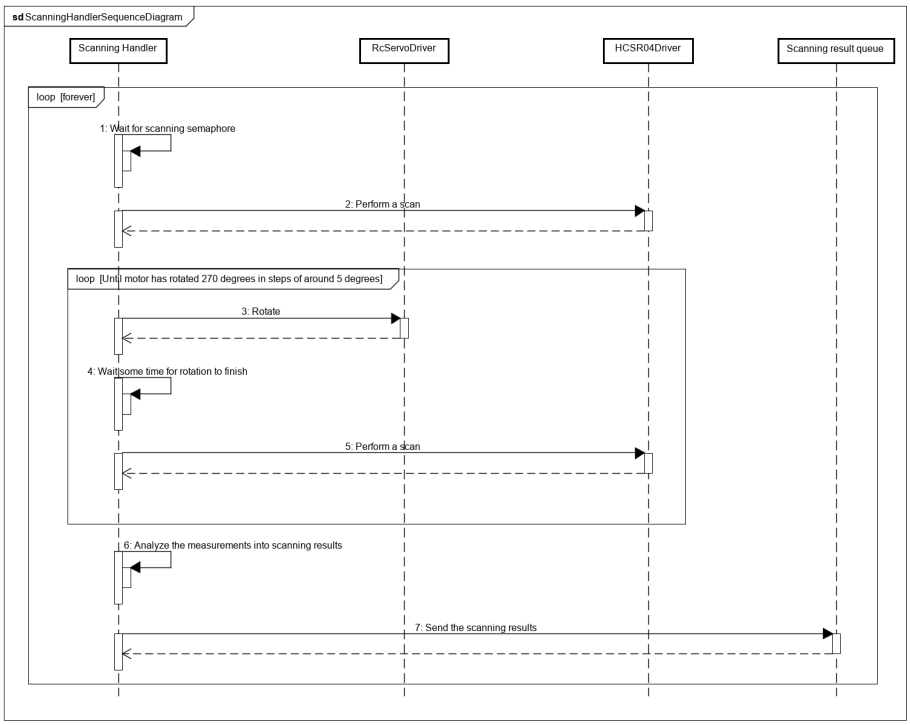

Figure 19: Scanning handler sequence diagram

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 20 on page 40. 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. Imagine, the sensor set is placed in a way so it is facing the platform edge in a perpendicular way. At this position the sensor set will be distance *a* away from the edge of the platform, by rotating the sensor set *x* degrees the distance from the sensor set to the edge is *b* now. It is clear that distance *a* and distance *b* are not the same length. First, distance *a* value can be found by subtracting container length from trashcan platform's length and dividing the result by 2 (a = (160-60)/2 = 50) - this is valid 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 the distance *b* as distance *a* is known and 2 angles of the triangle are also known (x°- the rotation that the sensor set made, and the 90°angle). Distance *b* is calculated in the following way: $b = a * \sin(90°) / \sin(90° - x°)$. Similarly distance *c* can be calculated: $c = a * \sin(90°) / \sin(90° - y°)$ - notice how *y*° is used instead of *x*° when calculating distance *c*. Thus, as long as the platform's length, container's length are known and the container is centered on the platform the distances of all the dashed lines by using the respective perpendicular lines that form a triangle with the respective edge of the platform can be calculated.

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. Starting position is used to check if there might be a new object placed on the platform even when the previous object that was notified about was removed. In other words, it is useful in the following scenario:

1. An object is placed on the platform.
2. Scan is performed and kommune is notified about it.
3. The workers remove the object.
4. Another object is placed on the platform.
5. Scan is performed and kommune is notified about it.

If starting position of the object was not saved the system would not be able to recognize that on the second scan it is a different object and the system would think that the object was not removed yet and there is no need to notify about the same object again. However, this might give false positives if the first object is just moved and it also will not work if the second object is placed at the exact same position as the first object and their widths match.

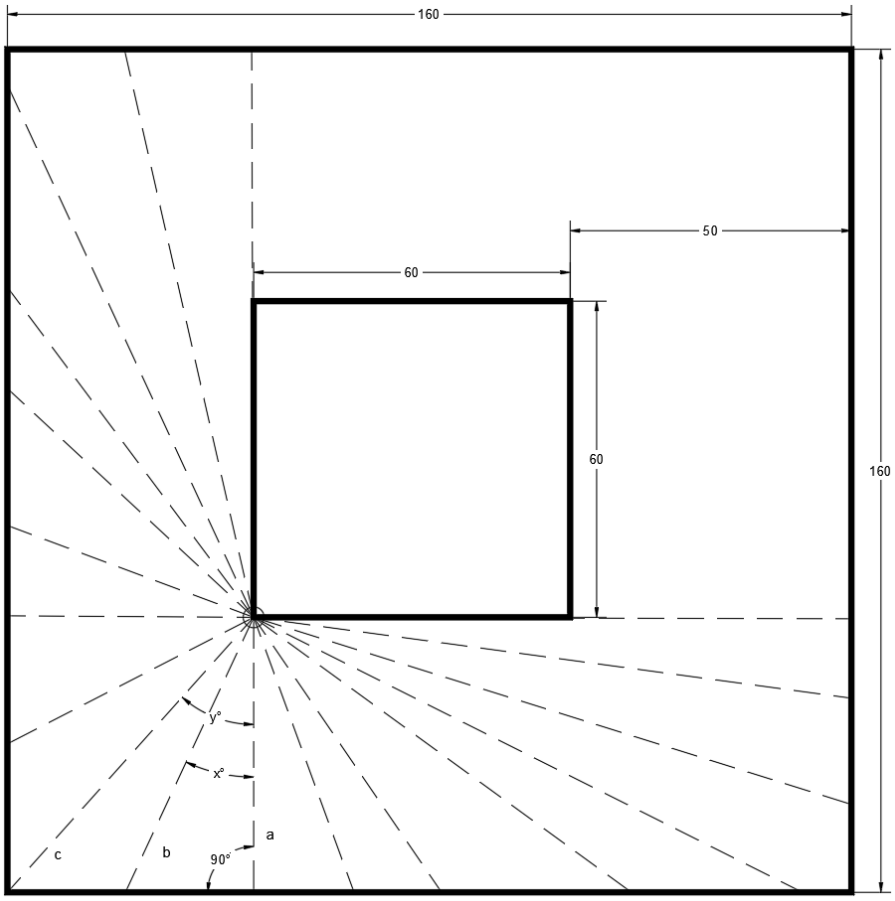

Figure 20: Trash can view from the top

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 21 on the next page.

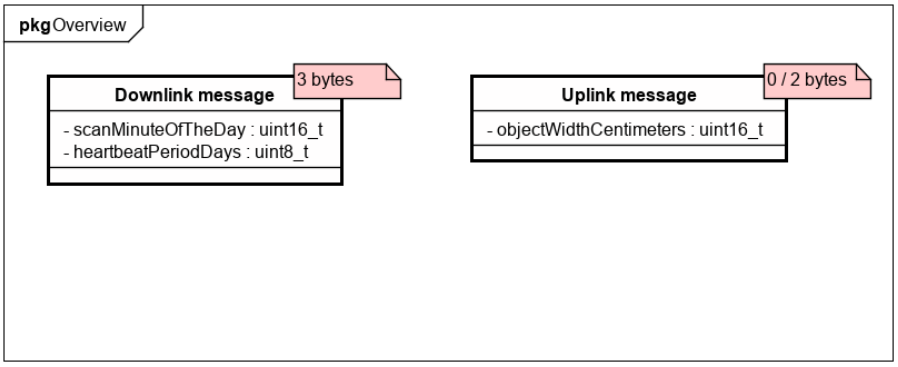

Figure 21: Uplink & Downlink message structures

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 21 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 22.

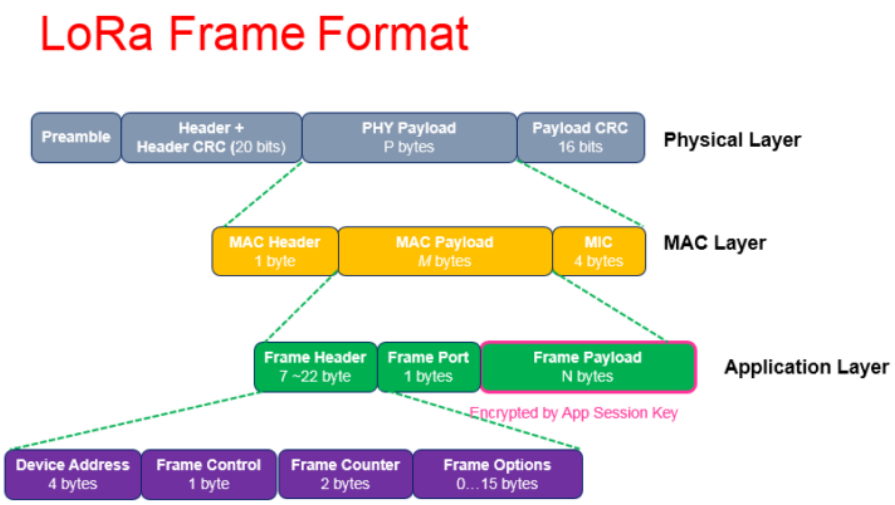

Figure 22: LoRaWAN frame structure (Techplayon, 2018)

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 23. 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 23: LoRaWAN message types (Prajzler, 2019)

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, 2020c). 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 16 on page 30. 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 Kommune dashboard server

This section focuses on the design of the kommune dashboard server. As mentioned in section 3.1 on page 26, 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 which can be found in section 2.1 on page 9, the design can be seen in figure 24 on the following page. 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.

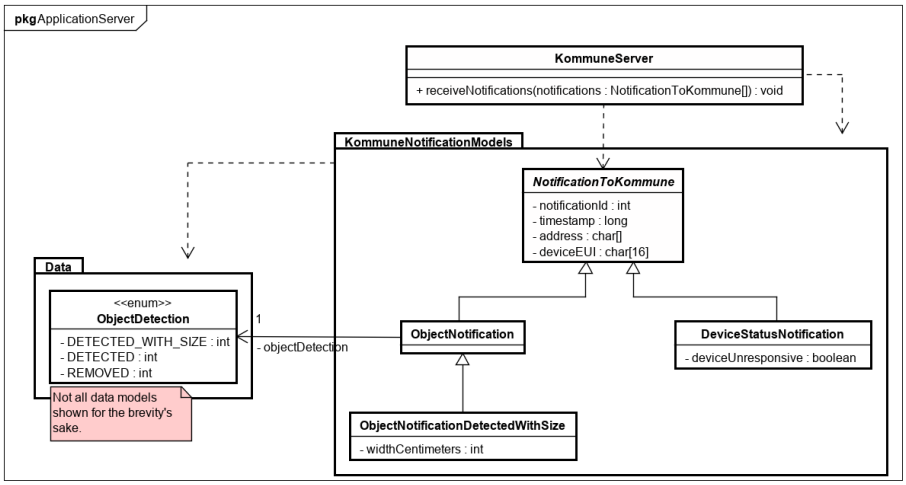

Figure 24: Kommune dashboard server endpoints & messages

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.

3.4 Application server

This section focuses on the design of the application server and its communication with other subsystems (detection system nodes and kommune dashboard 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. Even though the designs of these systems are completed, there's a middle-man between the end nodes and the application server - LoRaWAN network, its functionality also affects the design of the application server, specifically the design is influenced by the input / output messages of the websocket.

3.4.1 Websocket messages

Message structures (specifically how to deal with acknowledgements) that are read from / written to the websocket have influence on the design of the application server. The structure of an uplink message that is read from the websocket can be seen in figure 25. Some of the fields are ignored and not used by the system - *fcnt*, *port*, *encdata* - *port* could be used to indicate the message type, if it's a heartbeat message or an object notification message, however this is achieved in the *data* field as different message types will have different structure inside the *data* field. Uplink messages will have *cmd* field set to *rx*, so when listening to the socket if an *rx* message is received, it means it is an uplink message. The *ack* field plays an important role in the acknowledgement process - if the field is set to *true*, that means the device has received the latest downlink message that required acknowledgement, if it is set to *false*, it means no downlink message that required acknowledgement has been received by the device since the previous uplink message. This is used when a downlink message containing the device configuration is sent out, as the configuration messages require acknowledgements.

Parameter table

Parameter	Type	Description
`cmd`	string	identifies type of message, always 'rx' for uplink data messages
`EUI`	string	device EUI, 16 hex digits (without dashes)
`ts`	number	server timestamp as a number of seconds from Linux epoch
`ack`	boolean	**NEW!** acknowledgement flag as set by device
`fcnt`	number	frame counter, a 32-bit integer number
`port`	number	port number as sent by the end device
`encdata`	string (optional)	encrypted data payload as hexadecimal string, only present if APPSKEY is not assigned to device
`data`	string (optional)	decrypted data payload as hexadecimal string, only present if APPSKEY is assigned to device

Figure 25: Websocket uplink message structure (ihavn, 2020b)

Downlink message structure can be seen in figure 26 on the following page. Same as the uplink message, the *port* field is not used. When sending a downlink message the *cmd* field has to have the value of *tx* and if the message requires acknowledgement the *confirmed* field has to be set to true. There are 2 more message types that are related to sending downlink messages:

1. **Acknowledgement of send request** - indicates whether the downlink message that was written to the websocket by the application server is accepted by the network. As an example, it can return an error if the format of the provided downlink message was incorrect.
2. **Downlink confirmation event** - is sent when the provided downlink message has reached the gateway and is ready to be sent out to the end device.

Sequence of a successful downlink message that has been sent to and acknowledged by the end device can be described as follows:

1. Write a downlink message to the websocket.
2. Read *acknowledgement of send request* message from the websocket that indicates successful write to the network.
3. Read *downlink confirmation event* message from the websocket stating that the downlink message has reached the gateway.
4. Next time the end node sends an uplink message and opens up a receive window, the gateway sends out the downlink message, which is received by the end node.
5. Next uplink message from the end node is sent out with *ack* flag set to *true*.
6. Uplink message is read from the websocket that indicates that the last downlink message has reached the end node (*ack* field of the read uplink message is set to *true*).

Both, uplink and downlink, messages include a *data* field, which is binary data encoded as a string where each character represents a HEX value. All messages are formatted using JSON.

Parameter table

Parameter	Type	Description
`cmd`	string	identifies type of message, always 'tx' for downlink messages
`EUI`	string	device EUI, 16 hex digits (without dashes)
`port`	number	port number (1 to 223)
`confirmed`	boolean (optional)	**NEW!** request confirmation (ACK) from end-device
`data`	string	data payload (to be encrypted by our server) as a hexadecimal stringm, minimum two hex characters (equivalent of one byte).

Figure 26: Websocket downlink message structure (ihavn, 2020b)

3.4.2 Database

As shown in figure 14 on page 27 the application server is using a database, specifically a relational database - the design of the database is shown in figure 27 on the following page. A relational database is chosen as that's what project members have experience with and it fits the project requirements well. The database is mainly used for storing the devices (and their respective statuses and configurations, an entry in the device table represents a single detection system node, or more specifically a single LoRa module) and notifications. The main tables (*Device* and *Notification*) use auto generated int values as their 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.

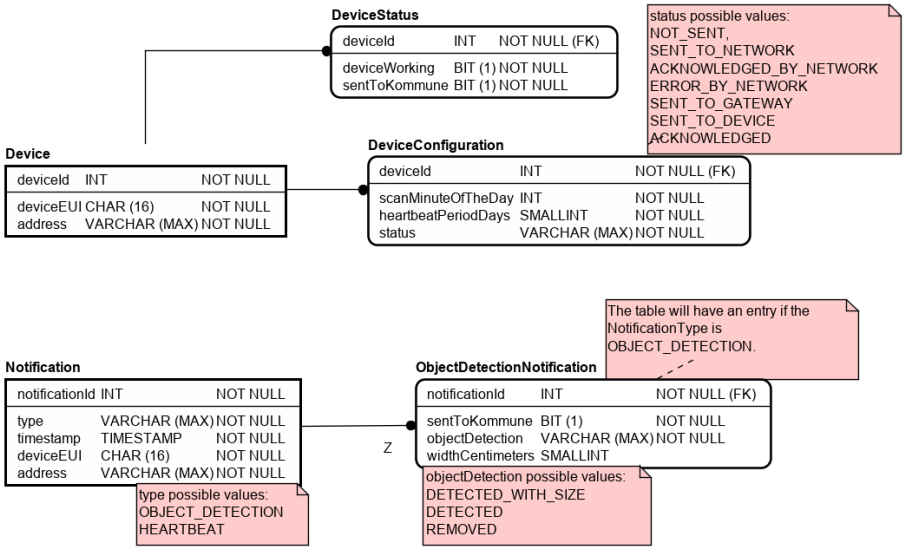

Figure 27: Database design

The device table consists of 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 the configuration's status (if it has been not sent, sent to lora network, acknowledged by the network, sent to the gateway, sent to the device and acknowledged by the end device). 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, they are mainly stored for 2 reasons:

1. So it is possible to resend the notifications when the kommune dashboard server was not able to accept notifications while for whatever reason.
2. The latest notification from specific device timestamp is used to find devices that have not sent notifications for longer than their heartbeat period.

The *Notification* table consists of:

- *notificationId* - auto generated int used as the primary key.
- *type* - type of notification, either *OBJECT_DETECTION* or *HEARTBEAT*, it's basically an enum value stored as *varchar*. It could be stored as integer (with values being 0 or 1), which would be much more storage efficient.

However, storage is very cheap nowadays, so clarity is preferred over saving storage as it makes the database easier to understand for the next possible developer of the system.

- *deviceEUI* - the device eui that the notification had at the time of the arrival, stored as 16 hex digits represented as characters.
- *address* - stores the address of the device at the time when the notification arrived.

Instead of having both the deviceEUI and the address, the notification table could just have a reference to the *Device* table by having *deviceId* as a foreign key. However, if the customer later ever wanted to change the address of the device, the change would make it look like old notifications were received from the new address even though they were received at the time the device was at the old address. It could be possible to use the *deviceId* foreign key and still points the old notifications to the old address by whenever the customer wants to change the address instead of just changing the value of the address in the database, a duplicate entry of the corresponding device entry is made just with updated address. This way the old notifications will still point to the device that has the old address and new notifications will point to the new device which has the new address. However, it was preferred to just store the deviceEUI and the address in the notification table to keep track what were the values at the time of the notification arrival.

As there are 2 types of notifications and *OBJECT_DETECTION* notifications contain additional fields compared to *HEARTBEAT* notifications another table is created - *ObjectDetectionNotification*. Instead of adding additional fields as nullable fields to the *Notification* table that would always be *NULL* for *HEARBEAT* notifications, an entry to the *ObjectDetectionNotification* table is added when the notification is of type *OBJECT_DETECTION*. *ObjectDetectionNotification* table contains only the fields that are specific to the *OBJECT_DETECTION* notifications:

- *sentToKommune* - boolean value indicating whether the notification has been received by the kommune dashboard server, not necessary for heartbeat notifications as only the object detection notifications are sent to the kommune dashboard server.
- *objectDetection* - enum value indicating how the platform state changed, possible values - *DETECTED_WITH_SIZE*, *DETECTED*, *REMOVED*. It's stored as varchar for the same clarity reasons as mentioned above.
- *widthCentimeters* - nullable int value that is filled when *objectDetection* is of type *DETECTED_WITH_SIZE*.

This structure is a way of implementing a subtype - *ObjectDetectionNotification* is subtype of *Notification*. No heartbeat notification table is needed, as heartbeat notifications do not contain additional fields compared to general notification.

Alternatively, instead of storing all notifications, only the *OBJECT_DETECTION* type notifications that have failed to reach the kommune dashboard server could be stored. And for keeping track of when the latest notification from a device was received a *lastNotificationTimestamp* field could be added to the *device* table. However, as mentioned the project team is not concerned with storage shortage, so it's preferred to keep a notification history.

3.4.3 Application server functionality

From the requirements and overall system design it was analyzed that the application server has the following tasks:

- Create an interface that allows the system administrator to register and configure detection system nodes. In this case, REST API endpoints are exposed.
- Listen to the websocket for messages, forward them to the kommune dashboard server and store them in the database.
- Periodically resend notifications that were not received by the kommune dashboard server.
- Periodically check for devices that have not communicated for longer than their heartbeat period, mark them as not working and send the status update to the kommune dashboard server.

In the following sections, each task is shown in a sequence diagram with a brief description provided.

API endpoints controller

The exposed API endpoints are shown in figure 28 on the next page, where each operation indicates an HTTP endpoint.

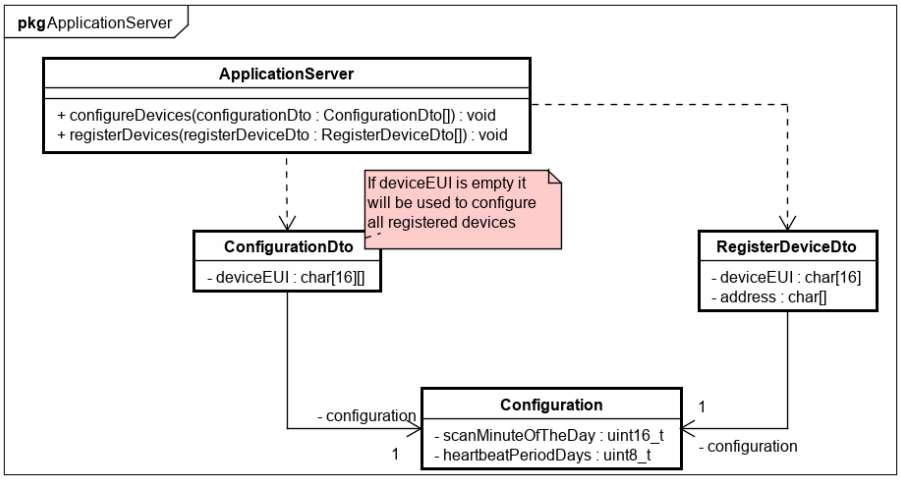

Figure 28: Application server endpoints

The *configureDevices* endpoint accepts *ConfigurationDto* array as a parameter, where each *ConfigurationDto* has an array of *deviceEUI* values and configuration values stored in *configuration* field. The *deviceEUI* array allows to configure multiple devices to the same kind of configuration at the same time and the *ConfigurationDto* array allows to send multiple configurations to the application server at once. This allows the endpoint to receive multiple configurations, where each configuration is applied to multiple devices, at once. The *registerDevices* endpoint accepts an array of *RegisterDeviceDto*, which contains the device EUI, an address and the initial configuration. For same reasons as for previous endpoint, the endpoint accepts an array of values instead of a single value as it allows to register multiple devices at once.

Both endpoints accept complex parameters, as such they will be exposed as *HTTP POST* endpoints, as complex objects can be passed in the *HTTP* request body. The endpoints accept *JSON* formatted messages.

The API endpoints controller's functionality is shown in 29 on the following page. It basically just adds devices and corresponding configurations to the database and sends the configurations to the websocket.

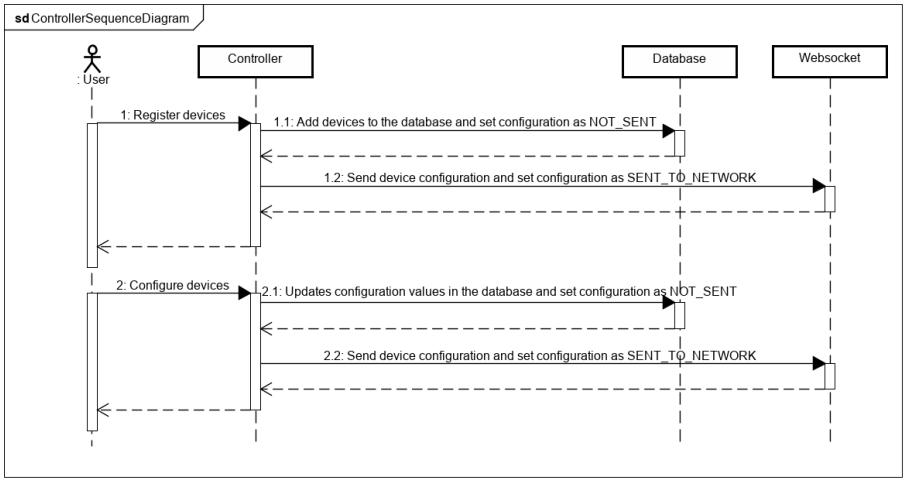

Figure 29: Controller sequence diagram

Message retrieval task

As mentioned, the application server has to continuously listen to the websocket as messages are not stored anywhere by the LoRa network. The main purpose of this task is to listen for uplink messages, the flow of this listening is shown in a sequence diagram in figure 30 on the next page. Besides just forwarding the notifications to the kommune dashboard server it also has to deal with the acknowledgements as the acknowledgements for downlink messages are indicated by the *ack* property of the uplink message. When an uplink message is received and configuration's status is *SENT_TO_DEVICE* it checks the uplink message's *ack* property - if it is *true* it means that the device has received the configuration and configuration's status is updated to *ACKNOWLEDGED*, if it is false then the configuration is resent.

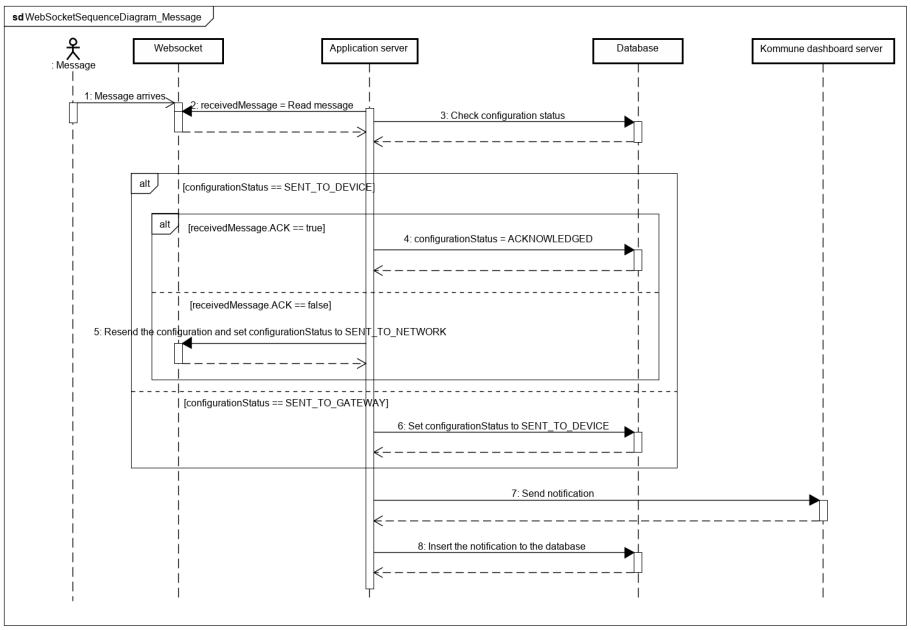

Figure 30: Websocket sequence diagram for receiving uplink messages

Websocket does not output only uplink messages, but other message types too - their execution flow is pretty simple, so no sequence diagrams are shown, instead they are explained in the following list, which lists the message type as the *cmd* field value and how such message type is dealt with:

- *cmd* value of *tx* - acknowledgement of downlink message send request, indicates whether the request was successful or erroneous. Accordingly the configuration status in the database is updated either to *ACKNOWLEDGED_BY_NETWORK* or *ERROR_BY_NETWORK*.
- *cmd* value of *tx* - indicates that the downlink message has reached the gateway, as such the configuration status in the database is updated to *SENT_TO_GATEWAY*.

All messages are kept track of and sorted out by the *EUI* property in the messages, which indicates the device EUI value.

Resend failed notifications task

As mentioned, the kommune server sometimes is not able to accept the notifications and as such these notifications have to be resent later. It was decided to resend these notifications in batches periodically - the flow is shown in a sequence diagram in figure 31 on the following page. When the timer expires, it loads the notifications that did not reach the kommune server, resends them and if the resend was successful, it updates the database to indicate that the sent notifications were sent successfully.

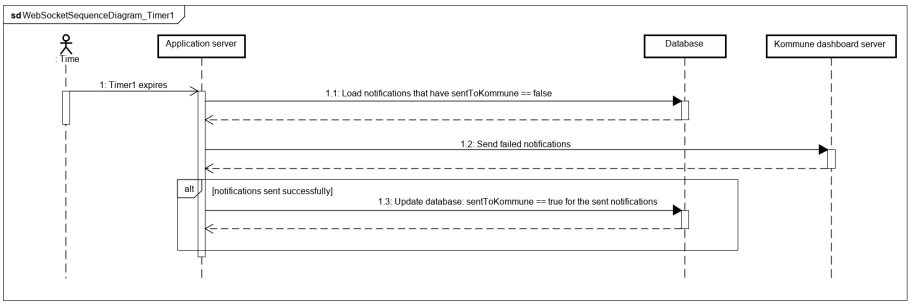

Figure 31: Sequence diagram for resending failed notifications

Refresh and send device statuses

The application server is also responsible for periodically checking if any of the registered devices have not communicated in their heartbeat period and report them to the kommune dashboard server as not functioning. The flow is shown in a sequence diagram in figure 32 on the next page. Timer expiration triggers the following flow. First, it updates devices that are registered as working and have not sent messages for longer than their heartbeat period as not working and inverts the *sentToKommune* field value, then it updates devices that are registered as not working, but have sent new messages to be registered as working devices and also inverts the *sentToKommune* field value. The *sentToKommune* field is inverted because the device status value is always inverted from the previous device status - if the kommune knew (*sentToKommune* == true) about the previous device status, that means the new device status has to be sent (*sentToKommune* is set to *false*), if the kommune didn't know about the previous device status (*sentToKommune* == false), there's no need to send a device status that kommune already has registered last. The first two steps that can also be merged into a single step with somewhat complex SQL query. After refreshing the device statuses and whether they should be sent to kommune, the application server queries the database for statuses that have to be sent and sends them to kommune dashboard server. If the statuses were sent successfully, the database is updated accordingly and *sentToKommune* is set to *true* for the sent statuses.

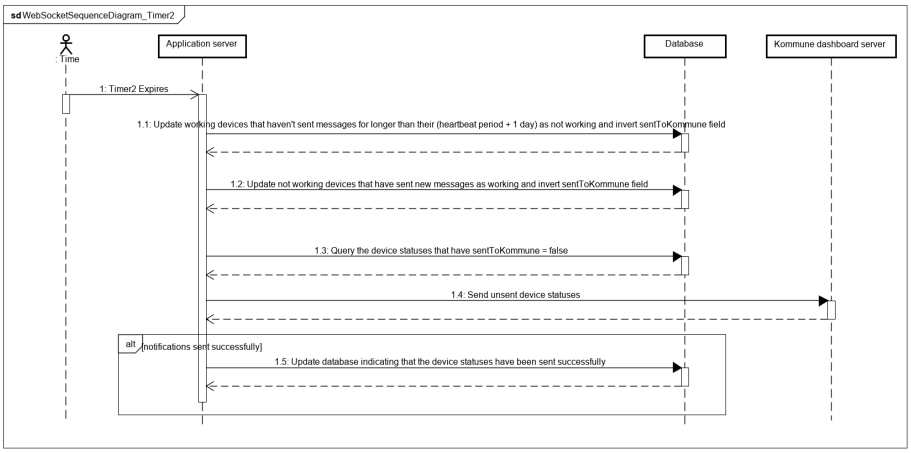

Figure 32: Sequence diagram for refreshing and sending device statuses

3.4.4 Class diagram

From the designs described above a class diagram for the application server is designed which is shown in figure 33 on the following page. The model classes inside *KommuneNotificationModels* and *Data* packages are not shown in this diagram. The *KommuneNotificationModels* package classes can be seen in figure 24 on page 44 and *Data* package classes can be seen in figure 34 on page 58.

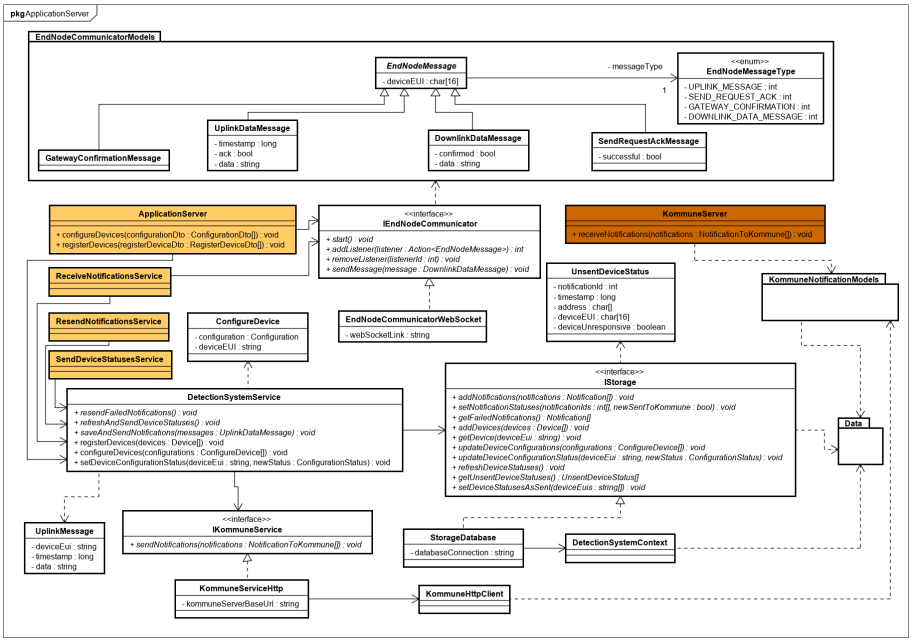

Figure 33: Application server class diagram

The system is designed in a modular approach that offers loose coupling between the modules. For example, for interacting with persistence storage - storing data in it and retrieving data from it - an *IStorage* interface is designed. The interface itself doesn't describe HOW the functionality is supposed to be established, only what kind of functionality it offers. *DetectionSystemService* has a reference to the interface, but it cannot just call methods on the interface, a concrete implementation of the interface has to be used. *StorageDatabase* is such implementation, it implements the functionality of the *IStorage* interface by using a database as a persistence unit. *StorageDatabase* uses *.NET Core EF core* nuget package (Microsoft docs, 2020d) to interact with the database. *EF Core* functions as an ORM layer (Wikipedia contributors, 2020c) allowing the system to interact with the database by interacting with *.NET* model classes. That is why it has a reference to *DetectionSystemContext* (which is a *DbContext* from *EF Core* package (Microsoft docs, 2020b)) that has dependency on the *Data* package, which contains model classes that represent the database. To avoid *DetectionSystemService* having a direct reference to *StorageDatabase* dependency injection of *ASP .NET Core* is used (Microsoft docs, 2020c). *StorageDatabase* is registered as the implementation of *IStorage* in the dependency injection, so whenever an implementation of *IStorage* interface is asked for, the dependency injection provides the *StorageDatabase* implementation. This way the *DetectionSystemService* is asking for an implementation of the *IStorage* interface, but it is not aware what implementation specifically it is using. If later a different implementation

is to be used, perhaps an implementation that uses files to store data persistently, only the part that registers which implementation to use for *IStorage* in dependency injection has to be changed. Similarly, *DetectionSystemService* is using *IKommuneService*, which has an implementation that uses an HTTP client to comminucate with kommune dashboard server. It's probable that the kommune would like a different way of communication and with this design only a new implementation of the *IKommuneService* would have to be made and registered in the dependency injection register and the rest of the system remains the same.

Since both, the kommune dashboard server and the application server, are made in the same programming language, their implementation can share *KommuneNotificationModels* package to not duplicate code and implement the notification model classes twice. *KommuneHttpClient* is using the models to know how to phrase the notifications and *KommuneServer* is using the model classes to define what kind of notifications it wants to receive. If kommune later wants different structure messages, the model classes have to be changed only in one place.

DetectionSystemService is like a facade class (Wikipedia contributors, 2020a), basically exposing most of the functionality that the application server is capable of. Unlike like the *IKommuneService* and *IStorage*, no interface was made for the *DetectionSystemService* as it is very unlikely that an implementation would be required that is not using *IStorage* and *IKommuneService*. Even if that happens it probably means the system is changing in a drastic way, which incorporates much more changes. Moreover, if such scenario occurs where an interface would have been useful, with today's tooling it is not too difficult to extract an interface from the implementation and replace its uses with the extracted interface.

Each application server functionality that was described in section 3.4.3 on page 49 is implemented in different classes (later referred to as execution tasks):

- *ApplicationServer* - exposes API endpoints for registering and configuring devices. Uses *DetectionSystemService registerDevices* and *configureDevices* methods. Also it uses the *IEndNodeCommunicator* to send device configurations in downlink messages.
- *ReceiveNotificationsService* - uses *IEndNodeCommunicator* to receive messages (most importantly new uplink messages) and uses *DetectionSystemService* to save them by calling the *saveAndSendNotifications* method.
- *ResendNotificationsService* - periodically calls the *resendFailedNotifications* method of *DetectionSystemService*.
- *SendDeviceStatusesService* - periodically calls the *refreshAndSendDeviceStatuses* method of *DetectionSystemService*.

The services are implemented using ASP.NET core worker services (Microsoft docs, 2020a) and the *ApplicationServer* is implemented using ASP.NET core web api (Microsoft docs, 2020e). Another advantage of this modular design is that it is possible to separate those execution tasks into separate execution units -

greatly increasing the scalability of the system. For example, the *Application-Server* could be running on a different server than the other tasks or even all 4 tasks could be running on separate servers. This requires the database to be deployed somewhere, where it would be accessible by all different servers, but that's not a big issue.

IEndNodeCommunicator exposes methods for communicating uplink and downlink messages (in this case communicating with the websocket). The methods *addListener* and *removeListener* allows any client class to start or stop listening for messages. When a class wants to start listening for messages it must provide an *Action* of *EndNodeMessage* - *Action* is a delegate that points to a method that accepts *EndNodeMessage* as a parameter and returns *void*. This way there's no reverse dependency from *IEndNodeCommunicator* to the listeners. This allows any class that has a method that takes *EndNodeMessage* as a parameter to become a listener. The *addListener* method returns an ID that can be used to stop listening by calling the *removeListener* method with the received ID. To send a downlink message the *sendMessage* can be called. The *IEndNodeCommunicator* interface has a dependency on *EndNodeCommunicatorModels* package, which contains the model classes for different messages. The client of *IEndNodeCommunicator* can then use these classes to interface with messages in an easier way. An implementation *EndNodeCommunicatorWebSocket* that uses the websocket to send / receive messages to / from end nodes. The implementation is responsible for converting the received messages from the websocket to the model classes in *EndNodeCommunicatorModels* package. The reason *IEndNodeCommunicator* is not used by the *DetectionSystemService* facade class is because not all of the execution tasks need to communicate with the end nodes. This way only the execution tasks that need to use it will have a dependency on it.

Implementations that communicate with external services use configuration to know how to connect to the external services:

1. *KommuneServiceHttp* - uses *kommuneServerBaseUrl* to know to what URL to send the notifications.
2. *StorageDatabase* - uses *databaseConnection* to know how to connect to the database.
3. *EndNodeCommunicatorWebSocket* - uses *webSocketLink* to know how to connected to the websocket.

All these configurations are also injected using dependency injection, allowing to easily use the same code with different configurations enabling 2 different *ReceiveNotificationsService*s sending notifications to 2 different kommunes.

A few classes in the diagram have a dependency on *Data* package that contains the model classes that represent the database entities. The *data* package with its classes can be seen in figure 34 on the following page. The classes are designed based on the database entity design - it is done like this so the object relational mapper can be used to interact with the database.

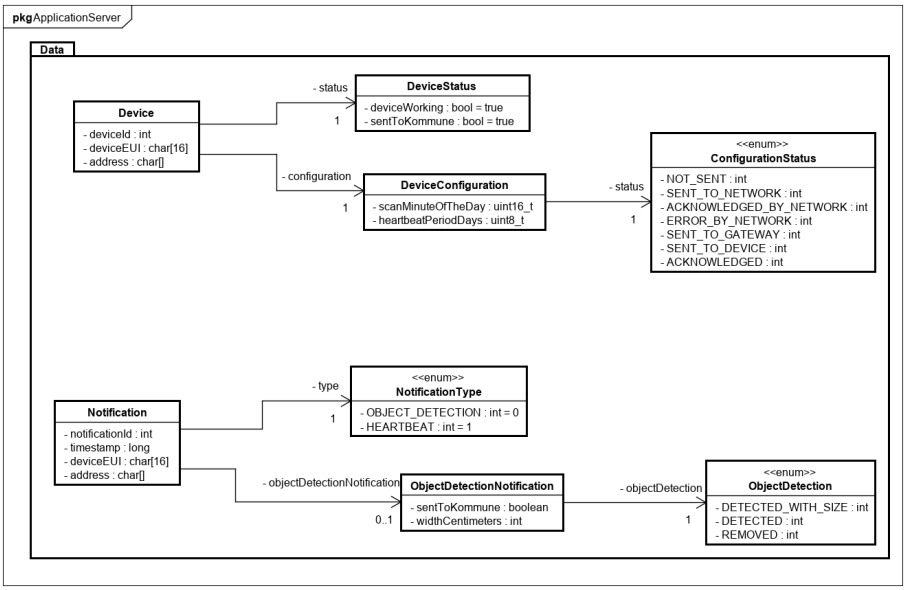

Figure 34: Data package class diagram

4 Implementation

This section showcases a few code examples with brief descriptions provided, for the sake of brevity some of the code examples are shortened. Section 4.1 shows code listings from the detection system end node's implementation. Section 4.2 on page 61 shows code listings from the application server's implementation. Source code can be found in appendix 9.1 on page 76.

4.1 Device

Mutex section macro, conditional debugging output and a timer usage in the HCSR-04 driver are described from the device's implementation in the following sections. It also should be noted that the implementation is not optimized for battery usage and could be improved by making the microcontroller go to deep sleep and powering off the peripherals when they are not in use.

Mutex section macro

As shown and described in section 3.2.2 the device design contains model classes whose fields are encapsulated by setters and getters, which also ensure mutual exclusion to the fields by using a mutex. To avoid repeating code in all setters and getters - taking the mutex at the start and giving it back after the operation - a mutex section macro is made. The macro accepts code as a parameter and

wraps that code with mutex interaction, that is - takes the mutex before the provided code and gives back the mutex after the provided code. The mentioned macro can be seen in listing 1 and an example usage of the macro can be seen in listing 2, which contains the *setScanMinuteOfTheDay* method of *Configuration* class.

```
#define mutexSection(...) xSemaphoreTake(self->mutex,
    portMAX_DELAY); __VA_ARGS__ xSemaphoreGive(self->mutex);
```

Listing 1: Mutex section macro

It is worth noting, the method self parameter has to be named *self* and the struct has to contain the mutex in a field named *mutex*. The macro could be further improved by accepting the variable names as parameters, but that was deemed unnecessary as the naming convention makes sense.

```
void set_scan_minute_of_the_day(configuration_t self,
    uint16_t scan_minute_of_the_day) {
    mutexSection(
        self->scan_minute_of_the_day =
    scan_minute_of_the_day;
    )
}
```

Listing 2: Mutex section macro usage, *setScanMinuteOfTheDay* method

Conditional debugging output

To make sure the execution flow is as expected a lot of print statements are added to the implementation. The print statements send the messages to the microcontroller's serial port, which is connected to a computer that has a program (HTerm, 2020) running that prints the received messages from the serial communication. However, the print statements are useful only when debugging - they serve no purpose when the system is deployed and instead just waste processing power. For this reason a wrapper function for the print statement is made, that prints messages only when it is in the debugging environment, otherwise it performs no operations. The wrapper function can be seen in listing 3.

```
void debugPrint(const char *format, ...) {
#ifdef DEBUG_PRINT
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
#endif
}
```

Listing 3: Conditional debugging output function

In the implementation, the wrapper function is used everywhere instead of directly calling a *printf* function. If the code is compiled with *DEBUG_PRINT* symbol defined then the *debugPrint* method will print the message. If it is compiled

without this symbol defined, then the *debugPrint* function will contain no statements. If the code is compiled without the symbol and with optimization enabled, the *debugPrint* calls most likely will be removed as the function itself does nothing. This approach allows to have print statements in the debugging environment without any performance wasted and code changes required in the production environment - only the *DEBUG_PRINT* symbol needs to be defined when compiling the code.

HCSR-04 Driver's timer usage

As mentioned in the design section, the HCSR-04 driver uses just one timer for multiple sensors instead of one timer per each sensor. To achieve this the driver has to mark the timer value when the measurement starts and then subtract the saved value from the current timer value to calculate the amount of timer ticks passed. However, a problem occurs when the timer overflows, especially as the timer is used without any prescalers (to get the best possible resolution) the timer overflows multiple times during single measurement. To counteract this problem, a timer overflow interrupt is enabled and a field is created that keeps track of how many overflows have occurred. Thus, when the measurement starts the current number of overflows and the timer value are saved. When the measurement finishes the current number of overflows and the current timer value are compared to the stored values. This approach allows to use multiple (possibly infinite) amount of sensors just with a single timer and achieve a high resolution. Code listing 4 showcases how the timer values are saved when the measurement is started and how the elapsed timer ticks are calculated when the measurement is done.

```
1  if (PIN_ECHO & _BV(sensor->echo_pin)) { // HIGH
2      sensor->numberOfTimerOverflows = numberOfTimerOverflows;
3      sensor->timerValue = TCNT1;
4  } else { // LOW
5      uint16_t overflows = (256 + (uint16_t)
       numberOfTimerOverflows - (uint16_t)sensor->
       numberOfTimerOverflows) % 256;
6      uint16_t timerValue = TCNT1;
7      int32_t timerDifference = (int32_t)timerValue - (int32_t
       )sensor->timerValue;
8      uint32_t timerTicksPassed = 65536 * overflows +
       timerDifference;
9
10     if (sensor->callback != NULL) {
11         sensor->callback(a + 1, timerTicksPassed);
12     }
13 }
```

Listing 4: HCSR-04 Driver performing a measurement

In the code, *sensor* refers to the *struct* of the sensor that is performing the measurement. When the measurement starts (when the sensor's echo pin is set to *HIGH*) the current timer values are stored in the respective fields of the

sensor struct. When the measurement is done (the echo pin is set to LOW), the amount of passed timer ticks is calculated and the sensor callback is called with the result. To calculate the amount of timer ticks passed, first a number of overflows that happened during the measurement has to be calculated. An 8 bit field *numberOfTimerOverflows* in the driver is used to hold the value of how many overflows of the timer have occurred. Each time a timer overflow occurs the field is incremented. However, the field itself can overflow so when calculating the number of overflows that occurred during the measurement a simple subtraction is not enough as can be seen on line 5. The calculation on line 5 keeps the results correct even when the field overflow has occurred during the measurement. This approach will achieve correct results as long as the number of timer overflows occurring during the measurement is less than 256 (number of values in an 8 bit field). If the number is higher than that then a 16 bit field can be used to store the number of timer overflows. The calculated number of timer overflows passed is then used to calculate the total number of ticks passed by multiplying it by 65536 (as a 16 bit timer is used) and adding the number of timer ticks passed.

4.2 Application server

This section shows and describes an example usage of the EF Core *DbContext* and how the periodic execution tasks (*ResendNotificationsService* and *SendDeviceStatusesService*) are implemented.

Example EF Core usage

The *refreshDeviceStatuses()* method of *StorageDatabase* is shown in listing 5. The method is responsible for refreshing the device statuses (steps 1.1 and 1.2 in the sequence diagram shown in figure 32 on page 54).

```
1  List<Device> devices = await _context.Device
2      .Include(device => device.Configuration)
3      .Include(device => device.Status)
4      .ToListAsync();
5  long currentTimestamp = DateTimeOffset.UtcNow.
       ToUnixTimeSeconds();
6  foreach (Device device in devices)
7  {
8      bool deviceHasNotifications = await _context.
       Notification
9          .Where(notification => notification.DeviceEui ==
       device.DeviceEui)
10         .AnyAsync();
11     if (!deviceHasNotifications) // If device has not sent
       any notifications at all, do not refresh its status.
12     {
13         continue;
14     }
15
```

```
16      double deviceHeartbeatPeriodInSeconds = (device.
    Configuration.HeartbeatPeriodDays + 1) * 86400;
17      bool anyRecentNotifications = await _context.
    Notification
18          .Where(notification => notification.DeviceEui ==
    device.DeviceEui)
19          .Where(notification => currentTimestamp -
    notification.Timestamp < deviceHeartbeatPeriodInSeconds)
20          .AnyAsync();
21      bool shouldChangeStatus = device.Status.DeviceWorking ?
    !anyRecentNotifications : anyRecentNotifications;
22
23      if (shouldChangeStatus)
24      {
25          device.Status.DeviceWorking = !device.Status.
    DeviceWorking;
26          device.Status.SentToKommune = !device.Status.
    SentToKommune;
27      }
28  }
29
30  await _context.SaveChangesAsync();
```

Listing 5: *refreshDeviceStatuses* method

First it retrieves all devices and stores them in *devices* list. Then it loops through each device in the list and checks if it has any notifications at all as the device statuses of the devices that have not sent any notifications at all are not refreshed. The configured hearbeat period of the device is converted to seconds from days, as this value is used to check if there any notifications that have been sent from this device recently as can be seen on line 17. If the device is working the *shouldChangeStatus* bool is set to invert value of *anyRecentNotifications* and if the device is not working then *shouldChangeStatus* is set to *anyRecentNotifications* value. Meaning the working devices should change status when no recent notifications have been sent from them and not working devices should change status when there have been recent notifications from these devices. If the device status should be changed then it is done so, additionally *SentToKommune* property is also inverted. After the looping is done all changes to the properties of the devices are saved on line 30. There's no need to indicate anywhere which properties should be updated as when the devices are fetched from the database on line 1 EF Core framework begins tracking these entities to know the occurred changes when the *_context.SaveChangesAsync()* is called. The method implementation is great for prototyping, however it does not scale well as all devices have to be fetched into memory and then the program loops through each device individually. If the system is to grow the method implementation should be replaced with an efficient SQL statement that does not require fetching the devices into memory.

Timed execution tasks

As the ASP.NET Core framework does not provide a nice interface for executing periodic tasks, to avoid duplicating code between execution tasks that need to execute periodically an abstract class is made that contains the boilerplate code - *TimedBackgroundService*. However, the framework does provide an interface for implementing background tasks - *IHostedService*. A part of the abstract class can be seen in listing 6.

```
protected TimedBackgroundService(TimeSpan period, ...)
{
    _period = period;
    ...
}

public Task StartAsync(CancellationToken stoppingToken)
{
    _logger.LogInformation($"{_serviceName} service started");
    _timer = new Timer(DoWork, null, TimeSpan.Zero, _period);
    return Task.CompletedTask;
}

private void DoWork(object state)
{
    DoWorkAsync().Wait();
}

protected abstract Task DoWorkAsync();
```

Listing 6: Parts of *TimedBackgroundService* abstract class

The *StartAsync* method is used by the ASP.NET Core framework to start the background task. It initializes a timer that calls the *DoWork* method periodically. The period is defined in the *_period* field, which is initialized in the constructor. *DoWork* method is just a wrapper method that executes the *DoWorkAsync* method synchronously. This is necessary as the timer accepts only synchronous methods. The *DoWorkAsync* method is an abstract method and it is implemented by each execution task implementation. With this approach a periodic task can be achieved by extending the *TimedBackgroundService* class, passing the task period to the constructor and implementing the *DoWorkAsync* method. An example can be seen in listing 7.

```
public class ResendNotificationService :
    TimedBackgroundService
{
    public ResendNotificationService(...) : base(TimeSpan.
    FromMinutes(1), ...) {}

    protected override async Task DoWorkAsync()
    {
```

```
        using IServiceScope scope = ServiceProvider.
    CreateScope();
        var detectionSystemService = scope.ServiceProvider.
    GetRequiredService<DetectionSystemService>();
        await detectionSystemService.
    ResendFailedNotifications();
    }
}
```

Listing 7: Period task's *ResendNotificationsService* implementation

The code creates a task that will resend failed notifications every minute. However, for it to actually work it has to be registered to the framework, which is fortunately just a one liner in listing 8.

```
services.AddHostedService<ResendNotificationService>();
```

Listing 8: Registering a hosted service

5 Test

The level of testing used in this project was positive testing (pp_pankaj, 2020) as this is a prototype project and its purpose is to prove that such system is achievable. Moreover, almost no automated testing was made, as automated testing is mostly useful for long living projects to make sure that code changes do not break the system, whereas the scope of this project is relatively small. However, there are a few unit tests written where project members were not sure about the implementation and unit tests were a quick way of testing the code. For example, a unit test is written for a *StorageDatabase.refreshDeviceStatuses()* method that is described in section 4.2 on page 61. All testing was performed while the end node's implementation was running on Atmega2560 microcontroller and the application server was running on a Windows 10 machine.

5.1 Use case testing

In this section the use cases described in section 2.2 on page 11 are tested. For each use case scenario a brief description, a precondition, testing steps and testing results are stated. When testing is performed the periods of the execution tasks of the application server are changed to 1 minute instead of the usual 1 day as in production. All parts of the system are started before the test is performed unless specified otherwise in the precondition.

5.1.1 Use case: Notify about platform's state change

Base sequence

Test that the notification arrives correctly from the end node to the kommune

dashboard system. The device's implementation was set to send out a different platform state every minute instead of actually performing a scan. This allowed for an easier testing and the project members were not able to implement the actual scanning implementation because of time constraints. However, the ultrasonic sensor driver was implemented and tested together with RC servo motor, which would indicate that it is possible to implement the scanning functionality.
Precondition: None
Testing steps:

1. The device sends out a notification with new platform state.
2. Notification reaches the kommune dashboard server.

Testing results: The notification has successfully reached the kommune dashboard server and it was also saved into the application server's database.

Exception sequence 1 (device malfunction)

Test that when the device has not communicated for longer time period than their heartbeat message period it should be detected as a malfunctioning device and kommune should be notified.
Precondition: Data to the database is added that indicates that the device has not communicated for longer than their heartbeat period. More specifically, 8 devices are added - 4 registered as working, 4 as not working and then each group contains 2 devices that have sent their statuses to kommune and 2 that have not. Then for one of those devices notifications are added that indicate that the device has recently communicated and for other that the last communication was longer than their heartbeat period. This is done to test all possible cases shown in sequence diagram in figure 32 on page 54.
Testing steps:

1. The fake data is inserted into the database.
2. *SendDeviceStatusesService* executes because its period has passed.
3. The device statuses are updated as excepted.
4. All unsent device statuses (including the ones that were marked as sent before, but got updated) are sent to kommune.

Testing results: The device statuses were correctly updated and successfully sent to the kommune.

Exception sequence 2 (kommune server unresponsive)

Test that when the kommune dashboard server is not accessible the notifications are resent at a later time.
Precondition: Fake notifications are added to the database and they are marked as not sent to kommune. Also the simulated kommune dashboard server is not running at the start of testing.
Testing steps:

1. The fake data is inserted into the database.
2. The *ResendFailedNotificationService* tries to send unsent notifications to the kommune dashboard server, but the server is down.
3. The notification *sentToKommune* statuses remain the same.
4. The kommune dashboard server is launched.
5. Service's execution period has passed and the service is executed again.
6. The notifications reach the kommune dashboard server.
7. Notifications are marked as sent to kommune.

Testing results: The failed notifications were successfully resent to the kommune dashboard server until the server has received the notifications.

5.1.2 Use case: Manage platform

Scenario 1: Register platform

Test that it is possible to register a new device with initial configuration provided.
Precondition: The device has not been registered before.
Testing steps:

1. The database is checked to see how it currently looks
2. The application server "register device" endpoint is called with device eui, address and initial configuration.
3. The configuration is sent out to the LoRaWAN's websocket.
4. The database is checked to make sure that the device was added correctly and no other unnecessary changes occurred.
5. The device was added to the database correctly.
6. The websocket did not respond to the downlink message request.

Testing results: The device is added to the database successfully with its initial configuration, however the websocket did not respond to the downlink message request for unknown reasons.

Scenario 2: Configure platform

Test that already registered device's configuration can be updated
Precondition: The device has been registered
Testing steps:

1. The database is checked to see how the current configuration looks.

2. The application server device configuration endpoint is called with device eui and the updated configuration.
3. The configuration is sent out to the LoRaWAN's websocket.
4. The configuration in the database was updated correctly.
5. The websocket did not respond to the downlink message request.

Testing results: The device configuration in the database is updated successfully, however the websocket did not respond to the downlink message request for unknown reasons.

In the scenario tests the LoRaWAN websocket does not seem to respond to the downlink message requests for unknown reasons, so instead a new task was added to the end node's implementation that periodically pushes fake messages to the *downlink_message_queue* to test if the system reacts correctly to downlink messages. Fortunately, this was easy to do because of the modular design of the system.

It should be noted that during use case testing the project team found the LoRaWAN coverage in Horsens to not be perfect.

5.2 Non-functional requirement testing

This section describes the testing of non-functional requirements. Each non-functional requirement will have a brief description.

- **System should perform one scan a day** - The system is designed to perform one scan a day. The scanning time of the day is configurable.
- **The notification should include the address, deviceEUI, notification arrival time and estimated object width (if possible)** - The system was designed to include these parameters in the notification. Estimated object width is included when its calculation is possible. It is not possible to estimate object width when the object is very close to the sensor set. During use case testing it was noted that the notifications include the required parameters.
- **The project equipment should not interfere with the worker's usual way of cleaning out the underground containers** - The scanning method and where the sensor sets are attached do not interfere with worker's usual way of clearing the underground container.
- **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)** - the scanning method does not require to damage the equipment in any way.
- **The delay of notifying the kommune's dashboard server after the large object detection scanning process should not exceed a 60 minute mark** - during the use case testing the notifications reached the simulated kommune dashboard server quicker than the 60 minute mark.

- **The scanning device should operate on a battery** - the hardware of the detection system node allows it to be connected to a battery. The end node's system was also designed so the microcontroller would operate in deep sleep mode for most of the time and would wake up and power up all the peripherals when the scanning time occurs. However, it was never tried to connect the hardware to a battery and test the end node's power consumption to estimated the expected battery life.

6 Results & Discussion

Test	Result
Use case: Notify about platform's state change	
Base sequence (notify about platform state change)	Passed [1]
Exception sequence 1 (device malfunction)	Passed
Exception sequence 2 (kommune server unresponsive)	Passed
Use case: Manage platform	
Scenario 1: Register platform	Incomplete (Passed)
Scenario 2: Configure platform	Incomplete (Passed)
Non-functional requirements	
"System should perform one scan a day"	Passed
"The notification should include the address, deviceEUI, timestamp and estimated object width (if possible)."	Passed
"The project equipment should not interfere with the worker's usual way of cleaning out the underground containers."	Passed [2]
"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)."	Passed [2]
"The delay of notifying the kommune's dashboard server after the large object detection scanning process should not exceed a 60 minute mark."	Passed
"The scanning device should operate on a battery."	Unknown

Discussion

It can be seen that most tests passed , but not all of them were done as full integration tests. For example, as mentioned in section 5.1.1 on page 64 during the base sequence test of *notify about platform state change* use case, no actual scanning was performed, but instead fake scanning results were used for easier testing and as the project team did not implement fully functioning scanning functionality because of time constraints. However, the sensor set was tested separately and no problems could be foreseen that could occur while imple-

[1] Tested with fake scanning results.
[2] Tested only from the software engineering point of view.

menting the functionality. Similarly, during the test of *manage platform* use case scenarios the LoRaWAN websocket was not reacting to the downlink message requests as expected, thus a task was created in detection system node's implementation that would put fake downlink messages in the *downlink_message_queue* to test the rest of the system. The reason those tests are marked differently in the result table is because the base sequence test just used fake scanning results, but the rest of the implementations were left as they are and the notifications proceeded through all system parts from the end node to the kommune dashboard server successfully. Whereas, for the *manage platform* use case scenarios testing, the communication between the application server and the end node was not established so it had to be mocked. Moreover, some of the tests were looked at only from software engineer's point of view, but it is possible that the results of the test would change if they are analyzed by different kind of engineers. For example, the test for the non-functional requirement stating that the project should not damage the trash can's equipment in any way passed. However, if the detection system is to be placed on the trash cans in a way that the detection system equipment would stay sturdy for years, it is possible that the trash can needs to be damaged. Nevertheless, no time was spent for analysing this as this is out of scope for this project. The end node's hardware and software design indicate that it could operate on a battery well, but no actual testing was done for it.

Overall, even though parts of the system implementation had to mocked, it is believed that the testing proves that the core functionality of the system is possible.

7 Conclusion

The hereby designed system is able to scan trash can platforms for large objects and notify the kommune dashboard server. The chosen software technologies and the hardware proved to be the correct choices in the end. They allowed for quick prototype implementations because of wide community support and provided frameworks. For example, ASP.NET Core allowed to make a quick implementation of the Web Api with background execution tasks. Whereas, FreeRTOS allowed to build a task driven system with little boilerplate and overhead. LoRaWAN fit the project requirements well, but as mentioned in section 5.1 on page 64 the coverage in Horsens was not perfect, so perhaps different networking technologies could be looked into. The designs of the systems proved to be modular and well layered allowing parts of the project to be replaced or for additional functionality to be added easily. This is well showcased in section 5 on page 64, where parts of the system were not functioning or implemented as wanted and were replaced with mocking implementations so the rest of the systems could be tested. The testing results proved that the desired system described in introduction is possible. While the technology choices might not be the most efficient ones and there is a lot of room for improvement in the implementation, however the modular design and well layered design allows to

replace the chosen technologies or improve the implementation without much struggle.

All in all, the project can be considered a success as it has been analysed that the desired system can be achieved with a powerful design. However, more of the functionality has to be implemented and tested for the project to be considered a minimum viable product. The project would also require input from mechanical engineering to develop the product for testing in environment that is similar to production.

8 Project future

Possible areas of improvement will be listed bellow for this project. These improvement areas are considered only from software engineering point of view.

8.1 Security

When exposing API endpoints it is also important who can use them. A good approach could be to allow the usage of the endpoints only for the users who provide a valid authorization token and the tokens would be handed out to valid users. For more user friendly approach an OAuth protocol could be used. OAuth (OAuth, 2020) is an open protocol for secure authorization for server side APIs. It is important when the user is using the endpoints in a suspicious way (possibly a DoS (denial-of-service) attack) then that authorization token that is usually granted per application can be suspended and requests from that source ignored.

8.2 Hardware

Hardware could be encapsulated into a single device allowing for more compact design and more power efficient device. Also different scanning hardware could be tested while observing the battery life and scanning result accuracy.

8.3 Communication

In section 2.3 on page 14 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 in the batch 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. Allowing for better power efficiency and less communication noise generated.

Additionally, different communication protocols could be tested while considering different factors such as battery life and network coverage.

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>.

Danish Environmental Protection Agency, 2020. *Waste in Denmark*. [Online; accessed 25-May-2020]. Available at: <http://www.seas.columbia.edu/earth/wtert/sofos/Denmark_Waste.pdf>.

Dr. Reinhart, 2020. *Solid Waste Collection*. [Online; accessed 25-May-2020]. Available at: <http://msw.cecs.ucf.edu/Lesson3-wastecollection.html>.

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

Elkoplast, 2020. *Underground Containers*. [Online; accessed 3-June-2020]. Available at: <https://www.elkoplast.eu/underground-containers>.

FreeRTOS, 2020a. *FreeRTOS*. [Online; accessed 27-May-2020]. Available at: <https://www.freertos.org/>.

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

— 2020c. *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/>.

Google Maps, 2020. *Images from google maps*. [Online; accessed 30-March-2020]. Available at: <https://www.google.com/maps>.

HTerm, 2020. *HTerm*. [Online; accessed 27-May-2020]. Available at: <http://www.der-hammer.info/pages/terminal.html>.

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

— 2020b. *Websocket output format*. [Online; accessed 25-May-2020]. Available at: <https://github.com/ihavn/IoT_Semester_project/blob/master/LORA_NETWORK_SERVER.md>.

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

Microsoft docs, 2020a. *Background tasks with hosted services in ASP.NET Core*. [Online; accessed 27-May-2020]. Available at: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-3.1&tabs=netcore-cli>.

— 2020b. *Configuring a DbContext*. [Online; accessed 27-May-2020]. Available at: <https://docs.microsoft.com/en-us/ef/core/miscellaneous/configuring-dbcontext>.

Microsoft docs, 2020c. *Dependency injection in ASP.NET Core*. [Online; accessed 27-May-2020]. Available at: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1>.

— 2020d. *Overview of Entity Framework Core*. [Online; accessed 27-May-2020]. Available at: <https://docs.microsoft.com/en-us/ef/core/>.

— 2020e. *Tutorial: Create a web API with ASP.NET Core*. [Online; accessed 27-May-2020]. Available at: <https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-3.1&tabs=visual-studio>.

OAuth, 2020. *What is OAuth?* [Online; accessed 3-June-2020]. Available at: <https://oauth.net/>.

pp_pankaj, 2020. *Difference between Positive Testing and Negative Testing*. [Online; accessed 01-June-2020]. Available at: <https://www.geeksforgeeks.org/difference-between-positive-testing-and-negative-testing/>.

Prajzler, V., 2019. *LoRaWAN Confirmations and ACKs*. [Online; accessed 27-May-2020]. Available at: <https://medium.com/@prajzler/lorawan-confirmations-and-acks-ba784a56d2d7>.

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>.

Techplayon, 2018. *LoRa- (Long Range) Network and Protocol Architecture with Its Frame Structure*. [Online; accessed 27-May-2020]. Available at: <http://www.techplayon.com/lora-long-range-network-architecture-protocol-architecture-and-frame-formats/>.

The Things Network, 2020a. *LoRaWAN*. [Online; accessed 27-May-2020]. Available at: <https://www.thethingsnetwork.org/docs/lorawan/>.

— 2020b. *LoRaWAN Architecture*. [Online; accessed 27-May-2020]. Available at: <https://www.thethingsnetwork.org/docs/lorawan/architecture.html>.

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

Wikipedia contributors, 2020a. *Facade pattern*. [Online; accessed 27-May-2020]. Available at: <https://en.wikipedia.org/wiki/Facade_pattern>.

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

— 2020c. *Object-relational mapping*. [Online; accessed 27-May-2020]. Available at: <https://en.wikipedia.org/wiki/Object-relational_mapping>.

— 2020d. *Unified Process*. [Online; accessed 20-May-2020]. Available at: <https://en.wikipedia.org/wiki/Unified_Process>.

9 Appendices

9.1 Source code

Source code can be found in the attached .zip file or on GitHub - https://github.com/ESipalis/BPR

9.2 Project description

See next page.

Project Description

Large Object Detection Next to Waste Containers Integrated with LoRa

Romans Kotikovs 252550

Eimantas Sipalis 254018

Software Technology Engineering

October 2019

Table of Contents

Background Description

Denmark is one of the leading countries in Europe developing renewable energy resources and improving the sustainability of the environment in all kinds of ways, starting from alternative transportation ways to excessive use of renewable resources[1]. 30% of all energy used in Denmark comes from renewable sources[2] and 44% of electricity is provided by wind and solar power[3].

In 2011 Denmark produced about 9 million tonnes of waste. 61% of this waste was recycled, 29 % was incinerated and 6 % was landfilled. The goal is to recycle more and incinerate less. Separation of “wet” and “dry” waste is important due to many factors. Paper and cardboard (“dry waste”) burn well but can also be used to produce recycled paper. Organic (“wet”) waste or so-called food waste from households, can be used to produce biogas or as fertiliser on fields[4]. That’s why segregation is an important part and involves innovation.

Underground waste containers are the semi-new way of waste collection and segregation at the same time in Horsens. All waste is underground and enclosed most of the times with double drum door preserving unwanted odours from escaping and animals from getting in. This kind of containers helps maximise efficiency due to their large size, meaning fewer and faster pick-ups[5].

However, there’s a problem with this solution when people place large objects such as household furniture and kitchen equipment on the platforms of the underground containers disrupting the cleanup workflow because large objects complicate the process of lifting the platform up. To improve the efficiency of cleaning up the city, these objects must be automatically detected and waste collectors should be notified about their placement beforehand.

[1] From https://denmark.dk/innovation-and-design/sustainability
[2] From https://denmark.dk/innovation-and-design/clean-energy
[3] From https://denmark.dk/innovation-and-design/green-solutions
[4] From Denmark without waste (See ref)
[5] From Van Dyk recycling solutions (See ref)

Problem statement

There's a problem of people leaving large and heavy objects next to the trash cans which disrupts the usual waste collection workflow by not allowing to lift the platform of the underground containers.

Sub questions:

- How / What sensors can be used to detect large objects next to the trash cans?
- What should be the placement of the sensors for the most reliable object detection and the most efficient use of sensors?
- What's the structure of the messages sent by sensors to the gateway?
- Should the data be sent by each sensor separately or grouped together and sent as a group (since trash cans are usually placed in groups of 4 and each trash can will have its own sensor)?
- How often should the data be sent?
- What spreading factor should be used for most reliable/efficient communication?
- Should sensors inform about low battery or the central system would just warn about the sensors that have not sent data for more than a specific interval of time?

Purpose

The purpose of this project is to design a system that notifies waste collectors about large objects placed next to the trash cans that cannot be collected during (might even disrupt) the usual waste collection routine.

Persona / Scenario

Persona

Horsens municipality waste collectors.

Scenario

1. Someone throws out a fridge next to the sorting containers.
2. Sensors identify a large object next to trash cans and notify waste collectors.
3. Waste collectors are aware of a possible problem and can tackle it before sweeping the containers.

Delimitation

- The system will not be able to differentiate between heavy and light objects. So even if a large light object (like a big cartoon box) is placed next to the trash cans it will still detect it as a potential problem.

Planned Model and Methods

During the project period, a Kanban methodology in combination with the Agile Unified Process will be used. This will allow the team to work on tasks in iterations ensuring that the progress on the project will be made from the beginning of the project and it will allow the progress to be clearly observed.

Project Plan

The time scope for the project is 1100 hours - 550 hours per team member. The phase plan is estimated as follows:

Date	03/02/2020	17/02/2020	16/03/2020	20/04/2020	15/05/2020
Phase	Inception	Elaboration	Construction	Transition	Final deadline
Weeks	2	4	5	4	
Hours	147	293	367	293	

Risk assessment

Risk	Likeli-hood	Severity	Risk mitigation	Identifiers	Responsible
No access rights to LoRa gateway	5%	100%	Use different gateway	No communication from person responsible for the gateway.	Eimantas Sipalis
No connection to Kommune or waste collector company	10%	90%	Project can still be done without communication to Kommune.	No communication from Kommune	Romans Kotikovs

References

Van Dyk recycling solutions [online] Available at: <https://vdrs.com/underground-storage/> [Accessed 11 October 2019].

Ecoloxia [online] Available at: <http://www.ecoloxia.ca/en/underground-waste-container/> [Accessed 11 October 2019]

Demark without waste [online] Available at: <https://eng.mst.dk/media/mst/Attachments/Ressourcestrategi_UK_web.pdf> [Accessed 19 October 2019]

Pioneers in clean energy [online] Available at: <https://denmark.dk/innovation-and-design/clean-energy> [Accessed 30 October 2019]

Sustainability in Denmark [online] Available at: <https://denmark.dk/innovation-and-design/sustainability> [Accessed 30 October 2019]

Green Solutions [online] Available at: <https://denmark.dk/innovation-and-design/green-solutions> [Accessed 30 October 2019]

9.3 System requirements specification

See next page.

Large Object Detection Next to Waste Containers Integrated with LoRa

Runik Solutions

System Requirements Specifications

IEEE 830-1998

Romans Kotikovs 252550

Eimantas Sipalis 254018

Software Technology Engineering

6th semester

November 2019

Table of contents

Introduction

Purpose

The purpose of this document is to give the reader a detailed and elaborated overview of the requirements for the project “Large Object Detection Next to Waste Containers Integrated with LoRa”. The intended customers are waste collector companies who are using underground waste containers (in our case study, specifically in Horsens).

Product scope

“Large Object Detection Next to Waste Containers Integrated with LoRa” project developed by Romans Kotikovs and Eimantas Sipalis. This product will identify and notify waste collectors about large object placement next to the underground waste container platforms. It will benefit waste collectors by being aware in advance about possible disrupt in a specific location of the waste containers. This is achieved using sensors integrated with LoRa network, therefore achieving low-cost, long lifetime solution.

References

“Project Description, Large Object Detection Next to Waste Containers Integrated with LoRa”, October 2019, by Romans Kotikovs and Eimantas Sipalis.

Overall Description

Product Perspective

This product's hardware is dependent on the underground waste container but they are manufactured in accordance with EN 13071 standard.

Product's software solution will be built on top of existing LoRa Alliance network available in Horsnes.

Product functions

Product's main functionality is:

1. Identify potentially large, heavy objects around the waste container's platform.
2. Notify waste collector company about objects in a specific location of the waste container.

User characteristics

The product is intended to be used by the supervisor of the waste collector company in a specific area. The persona is expected to know the locations of the waste containers since the product's hardware and software are mapped to waste containers.

Constraints

1. Sensors that will be used will identify object size dimensions not weight. Therefore the system will not be able to differentiate between heavy and light large objects.
2. Sensors are used only a couple of times throughout the day, therefore waste collector company might not be notified on time.

Assumptions and dependencies

It is assumed that:

- The team will get access to the LoRaWAN gateway in VBI park.
- The team will be able to acquire the necessary sensors.
- The team will get permission to test their work with waste containers that are placed in Horsens.

Specific requirements

Functional requirements

Must:

- User should be able to see which trash cans have large objects placed next to them.

Should:

- User should be able to see the dimensions of the object that's placed next to the trash cans.
- User should be able to see which trash cans have out of order sensors and they need to be replaced.

Non-Functional requirements

Must:

- The system should detect any objects placed on the trash can plate up to 5 cm margin. Meaning that an object that's 5 cm on the plate won't be notified about.
- The system should scan for objects 2 times a day. (The specific times are right now unknown and should be set after further discussion with Horsens municipality).
- The system should use LoRaWAN for communication between nodes that are placed next to the trashcan platforms and the gateway.
- The system will use the Teracom network.
- The system should store received information in the database (timestamp, address and possibly the size of the object).

Should:

- The system should use acknowledgements to make sure that the data packages arrive successfully.
- The system should be able to detect non-functional nodes.

External interface requirements

User interfaces

The user interface is very low priority in this project, so right now there's no time allocated for the design of user interface. For now it's decided that the system will just show live data of addresses that have large objects blocking the trash can plates.

Software interfaces

The application layer will use .NET

Hardware interfaces

It's not fully decided what sensors will be used for object detection. Right now the main focus is on ultrasonic sensors, but after further analysis we might decide to use Lidar sensors. The most compatible ultrasonic sensor that we have found so far is: **DL-MBX** - it has up to 10m measurement distance with 1mm precision and it's compatible with LoRaWAN.

Communication interfaces

The communication between the nodes and the gateway will be done using LoRaWAN.