Traffic Counter

Design Project Team 11

Matthijs Reyers
Everard de Vree
Floris Heinen
Leo Ruizendaal
Jagvir Singh Bal

November 2023

Abstract

This is the final report for the Design project of the UTwente technical computer science bachelor. During this project team 11 built a minimum viable product of a traffic counter for our client, Mindhash, a company based in Hengelo. The traffic counter hardware itself is an embedded device based on a ESP32 that can be mounted next to a road to count cars using two time of flight laser sensors. To demonstrate the system's capabilities we also developed a storage server for the devices to upload their data to and and a web dashboard to review the collected data.

Contents

1 Introduction **5**

2 Domain analysis **6**
2.1 Introduction to the domain . . . 6
2.2 Client and stakeholders . . . 6
2.3 Limitations of the current system . . . 7
2.4 Embedded hardware . . . 7
2.4.1 ESP32 Wrover . . . 8
2.4.2 Time of flight sensors . . . 8
2.4.3 LTM modem . . . 8

3 Project definition and approach **9**
3.1 Project definition . . . 9
3.2 Project approach . . . 10
3.3 Requirements formation . . . 10

4 Requirements specification **11**
4.1 Functional Requirements . . . 11
4.1.1 Must have . . . 11
4.1.2 Should have . . . 11
4.1.3 Could have . . . 11
4.1.4 Won't have . . . 11
4.2 Quality Requirements . . . 11

5 Preliminary research **12**
5.1 Bandwidth analysis . . . 12
5.1.1 Strategies . . . 12
5.1.2 Overhead . . . 13
5.1.3 Total data usage per year estimation . . . 13
5.1.4 Conclusion . . . 13
5.2 Sensor accuracy analysis . . . 14
5.2.1 Aliasing . . . 14
5.2.2 Worst case overestimation . . . 14
5.2.3 Worst case underestimation . . . 15
5.2.4 Generalization . . . 15
5.3 RTC accuracy . . . 16

6 Global Design **17**
6.1 Components . . . 17
6.2 Programming languages and frameworks . . . 17

7 Detailed design **19**
7.1 Counter firmware . . . 19
7.1.1 Safety vs performance . . . 20
7.2 API Design . . . 20
7.2.1 API Keys . . . 20
7.2.2 Devices . . . 21
7.2.3 Detections . . . 22
7.2.4 Detection data over time . . . 24
7.2.5 Debug . . . 26
7.3 Dashboard . . . 27
7.3.1 Functions . . . 28
7.3.2 Framework . . . 28

7.3.3 Chart.JS . . . 28
7.3.4 Leaflet . . . 28
7.3.5 User friendliness . . . 29
7.3.6 User error prevention . . . 31
7.4 Detection algorithm . . . 32
7.4.1 Characterizing the sensor output . . . 32
7.4.2 Sensor data averaging . . . 32
7.5 Sensor rest position determination . . . 33
7.6 Distance thresholds . . . 34

8 Testing **35**
8.1 Embedded system . . . 35
8.1.1 Car detection algorithm . . . 35
8.1.2 Sensor metrics . . . 35
8.2 Detection accuracy . . . 36
8.2.1 Methodology . . . 36
8.2.2 Results . . . 36
8.2.3 Conclusion . . . 36
8.3 API integration testing . . . 37
8.3.1 Results . . . 37

9 Future Planning **38**

10 Conclusion **38**
10.1 Must . . . 38
10.2 Should . . . 39
10.3 Could . . . 39
10.4 Won't . . . 39
10.5 Quality Requirements . . . 40

11 Discussion **40**

1 Introduction

Many municipalities in the Netherlands collect road usage data to estimate wear and tear and inform urban planning decisions. This data is currently often collected using a so called "telslang" (literally *counter snake* in Dutch), shown in figure 1. These are pneumatic sensors in the form of rubber tubes that can detect when wheels pass over them. However, there are significant downsides to the use of "telslangen". The installation of the system damages the asphalt of the road and, because sensors count wheels rather than vehicles, the collected data needs undergo processing after collection in order to be turned into usable statistics.

Figure 1: Tel slang installed in Alkmaar, source: Noordhollands Dagblad[2]

The Hengelo based company Mindhash, hopes to address these problems by developing an alternative system that uses laser based time-of-flight sensors to detect cars rather than the pneumatic tubes. This reduces the complexity of the system installation and allows for detecting cars rather than wheels. Additionally Mindhash's system also adds internet connectivity via an LTE-M modem which allows for immediate uploading and remote viewing of the collected data, meaning that the integrity of an installation can be checked without visiting it in person. After an initial prototype was built with some off-the-shelf hardware and development boards Mindhash began the development of some custom hardware in the form of a PCB with all the sensors and chips integrated together.

Initially, the goal for this project was to develop the firmware for the new custom hardware using the Rust programming language and then evaluate the performance of the new hardware, as well as building a simple storage server and web dashboard to demonstrate a fully functional system. However, due to production and shipping issues neither the hardware nor the extra development boards actually arrived when expected. Because of this, in week 5 (halfway through the project) the decision was made to pivot to using the old prototyping hardware for the project instead. The prototypes were built on different sensors and ESP32 micro controllers rather than the STM32 chips used for the new hardware, which meant that the firmware development essentially had to be started from scratch again. However after a discussion with the client this was deemed to be a safer choice than waiting for the new hardware to arrive and risk discovering that it contained design flaws, which would leave the project unfinished.

Figure 2: Mindhash company logo

The switch to the prototyping hardware still held value for the client as the prototype was developed in microPython which caused some performance issues that made on device processing of the sensor data not possible. The goal shifted to showing that the ESP32 is capable of doing all the required data processing, to be achieved by rebuilding the firmware for the prototype hardware from scratch in Rust. This is very relevant information for the client, since ESP32 chips are far cheaper to obtain than the STM32 chips used in the new hardware. Additionally, we systematically measured the system performance to provide insight into the data usage and system measurement accuracy.

2 Domain analysis

In order to streamline the development process and to investigate the scope of the project, a domain analysis was conducted. The system spans several domains and a detailed understanding of them can help prevent excluding requirements and features that later turn out to be crucial. Conversely, having a clear idea of where the system begins and ends prevents wasting time and resources on the implementation of unnecessary features.

2.1 Introduction to the domain

The system will span several domains. The main three areas are:

- Real world data collection
- Online data communication and storage
- Data visualization and user interaction

First, real world data collection is at the center of the proposed system. An embedded system will have to analyze sensor data to draw conclusions about real vehicles. Potential challenges in this domain are limited processing power and sensor inaccuracies. Next, uploading the processed data to an online database could be limited by bandwidth constraints, and scalability problems. Finally, the data visualization subdomain will contain challenges regarding user-friendliness and completeness.

2.2 Client and stakeholders

The main stakeholder is the client, Mindhash. They are looking to develop a marketable system as an alternative to traditional pressure-sensor traffic counters. Mindhash might sell the system to other stakeholders without a relevant technical background, like government officials or traffic analysts. Additionally, the mechanics who might install our system in the future could be stakeholders. A straightforward installation process would be in their best interest. Finally, residents of the area around a traffic counter are indirect stakeholders in the sense that they see the system every day and might experience some aesthetic impact from it. However, these last two stakeholders mostly interact with the physical form of the system, which is beyond the scope of this project.

2.3 Limitations of the current system

The predominant method of traffic/road usage data collection is currently using pressure-sensors which comes with significant drawbacks. To give a better understanding of why this new system is relevant and what problems for the customers Mindhash hopes to address, the most prominent of these downsides are listed below. Each problem is paired with an explanation of how the new system is meant to address it.

- **Wear and tear:** The pneumatic tubes experience significant wear from car tires driving over them, which means that the old system requires significant maintenance and replacement of parts. This is costly, and financial considerations are important for many of Mindhash's clients. The new system has no moving parts that come into contact with anything and thus experiences significantly less wear.
- **Online communication:** The current system only passively collects data, which then has to be retrieved and processed at a later date. This means that if any issues cause the data to be incorrect, or not be collected at all, customers will not know about it until the system is removed from the road. The new system adds online communication that allows for remote monitoring and potentially even theft and tamper detection.
- **Installation damage:** The old system requires the road on which it is installed, to be damaged in order to mount the pneumatic tubes. The new system has no such issues as it can be mounted on any existing light or signpost in a non-destructive way.
- **Speed limit:** The pneumatic tubes can only handle cars going up to a certain speed before they experience significant damage or break free from their mounting points on the road. This means that the old system is not usable in all situations, while the new system is theorized to handle much higher speeds, with only the measurement accuracy being slightly affected.

2.4 Embedded hardware

As mentioned in the introduction the goal for this project was initially to develop the firmware for the newer, STM32-based hardware, but availability and production issues led us to pivot to reusing the old prototype hardware. Although significant time and effort was put into researching and understanding the original STM32 chip and new time of flight sensors, this research has been left out here since it is not relevant to understanding the final product, though these findings and certain issues regarding the application of the new sensor drivers in Rust have been communicated to the client since this was valuable information to them.

2.4.1 ESP32 Wrover

The main micro controller at the heart of the embedded system is an Espressif Systems ESP32 WROVER-E. This is a dual core 32-bit micro controller with built in Bluetooth and Wi-fi connectivity running at 80Hz (though not exactly 80Hz, as we later discovered). It also has support for several peripheral interfaces such as UART, IC2, SPI, SDIO, and TWAI.

Normally the ESP32 runs an operating system called freeRTOS that handles the scheduling of tasks, threads, interrupts, and dynamic memory allocation. However, because of our focus on performance and stability we choose to forgo freeRTOS and even heap allocations completely and instead use only the `esp32-hal` Rust crate. This in turn means more work had to be done configuring the hardware and more time be spent on gaining a good understanding of the hardware, but this paid of in the improved sensor pulling performance demonstrated in section 8.1.2.

Figure 3: Espressif Systems ESP32 WROVER-E

2.4.2 Time of flight sensors

The time of flight sensors used in this project are a pair of *TFmini Plus 12M IP65 LiDAR* sensors (pictured in figure 4). Despite the product name these sensors are not what we would traditionally think of as LiDAR as they do not produce a point cloud but a single distance measurement (so called single-point LiDAR). Fundamentally these sensors function by sending a laser pulse with a certain frequency which then bounces off the object in front of the sensor after which the distance to the object can be computed using the shift in the returning signal and the speed of light.

All this processing happens within the sensor itself and the resulting distance measurements are then sent to the ESP32 over a UART (serial) connection. The sensor support sending these measurements at different intervals, with anywhere from 1-1000 Hz being supported. The data sheet for the sensor specifies that the "stability" of the measurements becomes worse at higher measurement intervals, but does not specify in what way. This was investigated by us in section 7.4.1, where it was discovered that the sensors actually keep an internal moving average value at lower measurement intervals.

Figure 4: Benewake TFmini Plus A02 IP65 sensor.

2.4.3 LTM modem

To facilitate the online communications and GPS location capabilities of the device the SIMCom 7000G LTM modem is used. This modem also connects to the ESP32 over UART (serial) where it can be configured and used by sending AT commands. Some of these commands are standardized as part of the *ETSI TS 127 007* standard, but SIMCom also added several proprietary commands to the SIMCom 7000G that can be used to easily make use of higher level network protocols such as HTTP, HTTPS, MQTT, and FTP without the esp32 having to implement any parts of those protocols.

Figure 5: SimCom 7000G Modem.

3 Project definition and approach

A good start to a project is to have a clear definition of the problem that needs to be solved, and the road to get there. The problem statement was provided by the client, and a general approach to solving it was written by the team before starting development.

3.1 Project definition

Figure 6 contains the initial proposal provided by Mindhash.

Design Project Proposal - Optical Counter

Dear Students,

Congratulations on being selected for an exciting computer science project that involves designing an optical traffic counter utilizing a custom PCB equipped with two Time-of-Flight (TOF) sensors. Your task is to create embedded software that facilitates sensor data reading, processing, and transmission via LTE-M to a central server. The ultimate goal of this project is to develop a user-friendly dashboard that visually presents the traffic data collected from the sensors. The focus will be on categorizing motorized traffic into three segments: small, medium, and large vehicles. Through this project, you will gain hands-on experience in embedded systems, sensor integration, data processing, wireless communication, and data visualization. As a team of five students, effective collaboration, task delegation, and timely progress updates will be crucial to the successful completion of this project. Best of luck, and we look forward to witnessing your innovative solution to real-world traffic monitoring challenges.

What we provide

Get the support you need with Mindhash. During the 10-week period, we will be available to answer questions and clear up any uncertainties via Slack or in person at our Hengelo office. We will also provide you with the necessary custom hardware and a short list of high level requirements.

About Mindhash

Mindhash is a 4.5-year-old tech start-up, based at the High Tech Systems Park (HTSP) in Hengelo. At Mindhash, we build software and occasionally hardware to run that software: we work on a few of our own products, on experiments and prototypes that help us explore new ideas and technologies, and we help other companies innovate through consultancy and building proofs-of-concept. We love technology and try to pick our projects based on whether they provide an interesting challenge for us.

Figure 6: Initial proposal by Mindhash

3.2 Project approach

In order to ensure a successful workflow, the team will organize stand-up meetings every other morning at 10:00. This will either happen remotely, on campus, or at the Mindhash office. During these meetings, team members will explain what they are working on, what they expect to finish before the next stand up meeting, and what problems they are facing. Finally, upcoming deadlines for the project will be recapped. This is the designated time to adjust expectations and provide support if a team member mentions falling behind. The team will work in a shared GitLab repository, and will have the option to remotely push code to parts of the system like the backend data logging server.

The client has indicated that they are available for weekly meetings. The team will meet with them or contact them via Slack when important problems arise. The client will also provide at least one complete set of PCB and sensors to use for development and testing.

3.3 Requirements formation

Based on the initial meeting with the client, two sets of requirements were proposed based on their wishes for this product. The main priority for Mindhash is to obtain a functional prototype to demonstrate the functionality of the project at an upcoming conference, as well as to learn more about the hardware and its possible pitfalls and problems. One of the key takeaways from this meeting was that the power consumption is not a big consideration for this version of the product, while the data consumption is.

4 Requirements specification

4.1 Functional Requirements

4.1.1 Must have

- The system must be able to detect passing vehicles
- The system must be able to categorize detected vehicles into three length classes (small, medium, large).
- The system must be able to detect the travel direction and speed of passing cars.
- The system must store all detected cars in an online database.
- The system must use the precompiled firmware for the AFBR-S50 time-of-flight sensor because the laser is otherwise not guaranteed to be safe.
- The system must display the recorded data to an end user in some way.

4.1.2 Should have

- The dashboard should be able to display the car detection statistics based on the logged data.
- The system should encrypt communication between the hardware and the backend (e.g. SSL).
- The system should be able to communicate the hardware state (battery SoC, firmware version, GPS location) to the backend server.
- The central server should be able to handle multiple embedded devices sending data.

4.1.3 Could have

- The system could give live updates/individual car detections (via MQTT/tcp sockets).
- The system could store all detected cars in a local SD card.
- The system could support over the air firmware updates of the embedded hardware.
- The system could detect cars on a second road behind the primary road.
- The system could have tamper detection via the accelerometer (against vandalism).
- The system could have a Microsoft Excel export feature for the recorded data.

4.1.4 Won't have

- The system will not have power saving features like putting the microcontroller to sleep until a car comes near.
- The system will not be able to detect or count pedestrians or cyclists.

4.2 Quality Requirements

- The communications over the SIMCom component of the embedded system should use less than 500MB per 10 years when it counts less than 2.9 million cars that year, based on the assumption of 8000 cars per day on average. This leaves 17.12 bytes per detection on average.
- The system should correctly detect more than 90% of passing cars.

5 Preliminary research

5.1 Bandwidth analysis

In this section, three different strategies for transmitting the car detection data back to the central storage server are described and analyzed for their bandwidth usage. This serves to give the client a sense of the expected data usage before development started and to inform the decision for which communication strategy was suitable for this project.

5.1.1 Strategies

The following three strategies were analyzed.

Aggregate statistics This strategy reduces the data sent to simple statistics about the processed data. It greatly reduces the amount of data bytes sent per car detection, without considering protocol overhead. In this case, the statistics will be sent through a HTTPS POST request. The message will contain information about the vehicle counts for every size, as well as information about the minimum, maximum, mean, and standard deviation of the speed distribution for every vehicle class.

For every vehicle class, 2 bytes are allocated per statistic: count, minimum speed, maximum speed, mean speed, and speed standard deviation. Multiplied by the 3 vehicle sizes, this results in data packets of 30 bytes. As can be seen in the formula below. Note that this method is the only one with a constant cost, irrespective of the amount of passing cars.

$$\text{cost}() = \text{TCP} + \text{TLS} + \text{HTTP} + 30 \tag{1}$$

Batched detections With this strategy the detections are still aggregated in a given interval. However, rather than sending the statistics of the detections to the server, a list of all detections is sent to the server in a single https POST request. This would give the end users a more detailed overview of the traffic while also removing the step of retrieving the logs from the SD card, which simplifies the workflow of using the traffic counter.

Every detection is represented as 4 bytes, with the first two bytes used for a timestamp, two bits for the car type (small, medium, large), one bit for the travel direction, and the remaining 13 bits for the speed. Which means the total data used by this scenario can be described by the following formula:[1]

$$\text{cost}(detections) = \text{TCP} + \text{TLS} + \text{HTTP} + 4 * detections \tag{2}$$

Individual detections In this strategy every detection is sent to the server individually as soon as the detection takes place. This gives end users a real time view of the counter. Since the packet for a detection is sent as soon as the detection takes place, the packets in this scenario do not need a timestamp because the server can simply record the time of arrival of the packet. This does mean that there is a small discrepancy between the recorded detection time and the real detection time, but this is an acceptable compromise to reduce data usage since ms-level accuracy is not a requirement for this project. Without the timestamp, the data can be packaged into 2 bytes as described in the previous section.

To give this strategy a chance in the comparison it uses a raw TCP socket to communicate. The data usage cost of this strategy can be described using the following formula:

$$\text{cost}(detections) = \text{TCP_INIT} + (\text{TCP_SEND} + 2) * detections \tag{3}$$

Using the RFC specification[1] for TCP, the TCP overhead for establishing a connection (SYN, SYN-ACK, ACK) was determined to be approximately 96 bytes, while sending a packet (PSH, ACK) has an overhead of approximately 64 bytes. The major share of the bandwidth will be used by the encryption layer. Roughly approximated, establishing a connection will use 6500 bytes of overhead. [3] Using the RFC specification for TLS [4], the protocol will need an overhead of circa 40 bytes per packet.

For the HTTP overhead, 112 bytes were assumed based on the following minimal POST headers:

```
POST /data HTTP/1.1
Host: traffic.matthijsreyers.nl
Content-Type: application/octet-stream
Content-Length: XXX
```

5.1.2 Overhead

To evaluate the different strategies we computed the total amount of transmissions, the total amount of bytes sent, and the percentage of bandwidth dedicated to overhead and graphed it as a function of the amount of detected cars.

5.1.3 Total data usage per year estimation

Based on a sample report given to us by Mindhash, the expected amount of cars per day is around 8000. Given intervals of 15 minutes the system will see an average of 83.3 cars per interval. Using the calculations from

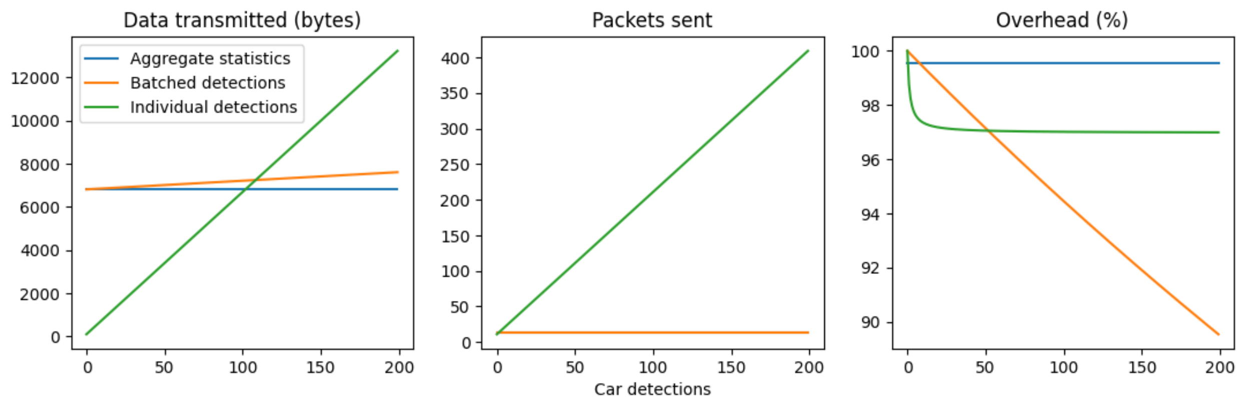


Figure 7: Estimated data usage of the different methods.

above (depicted in 7), it was determined that batched detections are preferable due to the fine granularity of the data and use of secure communications with simmilar data usages to aggregate statistics and unencrypted detections. Using this choice, the average total data usage is calculated as follows:

$$\text{cost_per_interval} = (\text{TCP}+\text{TLS}+\text{HTTP}+API_KEY)+4\cdot 83.3 = 96+64+40+36+6500+333.2 = 7069,2\text{bytes} \tag{4}$$

$$\text{cost_per_year} = \text{cost_per_interval} \cdot 4 \cdot 365 = 7069,2 \cdot 4 \cdot 365 = 10321032\text{bytes} = 10,321032\text{MB} \tag{5}$$

This is well below the 50MB of data usage allowed each year on a 500MB 10 year plan, leaving room for the device status updates and error reporting.

5.1.4 Conclusion

Despite our initial intuitions favoring the aggregate statistics technique, the results clearly show that once we consider all the bandwidth required for setting up a HTTPS connection the data saved by sending the statistics of the data rather than the full data itself is fairly insignificant. From these calculations it becomes clear that sending data in batches is efficient enough to satisfy our requirements with the system using an estimated 10.3MB per year where 50MB per year is allowed, leaving almost 400% margin.

5.2 Sensor accuracy analysis

5.2.1 Aliasing

The discrete nature of the sensors introduces uncertainty to the system through an aliasing effect.. The TFmini Plus LiDAR module has a maximum poll rate of 1000 Hz [5]. When the ESP32 is connected to the rest of the system, polling for cars and sending updates, the frequency at which the sensor data is read, is lower than 1000 Hz. For these calculations, a lower bound of 800 Hz was assumed, to account for some performance loss due to data processing and transmission. This means that there are, on average, $T = 1/800 = 0.00125$ seconds between sensor polls. This will be used in calculations of error ranges in observed speed and vehicle length. The speed of the vehicles $v_{calculated}$ is calculated by averaging the measured entry and exit speed.

$$v_{calculated} = \frac{v_{entry} + v_{exit}}{2}$$

v_{entry} and v_{exit} are calculated by dividing the distance between the sensors $d_{sensors}$ by the time a vehicle was detected by one sensor, but not both. The client provided information about the distance $d_{sensors}$ between the two TFmini sensors: they are set 0.400 m apart. For vehicle lengths that are multiplications of $d_{sensors}$, v_{entry} and v_{exit} are the same. This negates the averaging effect and allows for more extreme errors. Therefore, v_{entry} and v_{exit} will be assumed to be equal in the calculation of maximum errors. The measured speed is calculated by dividing the distance between the sensors by the measured time to cross that distance.

$$v_{measured} = v_{entry} = v_{exit} = \frac{d_{sensors}}{\Delta t_{measured}} = \frac{0.400}{\Delta t_{measured}}$$

Aliasing results in three possible scenarios regarding the accuracy of the measured vehicle speed: overestimation, underestimation, and perfect measurement. Consider the time t_{real} it takes a vehicle to cross the distance between the sensors.

$$t_{real} = \frac{0.400}{v_{real}}$$

With real time to cross t and poll rate T, the detected time to cross can differ from the real time. Due to the discrete nature of the sensor polls, the possible different times measured t_1 and t_2 are expressed in the number of poll intervals p_1 and p_2 it takes a vehicle to cross both sensors . Here, t_1 can result in an overestimation of vehicle speed and t_2 can result in underestimation. Note that when the true time to cross t is a multiple of the poll time T, the measured times t_1 and t_2 will both equal the true t.

$$p_1 = \left\lfloor \frac{t}{T} \right\rfloor,\ p_2 = \left\lceil \frac{t}{T} \right\rceil \tag{6}$$

$$t_1 = T * p_1,\ t_2 = T * p_2 \tag{7}$$

Which of the two times is measured, depends on the vehicle position relative to the first sensor at poll interval t_0. If the sensor is polled soon enough after a vehicle crosses its line of sight, the system will overestimate the time it took the vehicle to cross the sensors, and consequently underestimate the vehicle's speed. If the sensor is polled relatively late after a vehicle crosses its line of sight, the system will underestimate the time it took the vehicle to cross the sensors, and overestimate the vehicle's true speed.

5.2.2 Worst case overestimation

The maximum possible overestimation can be found close to the maximum speed registered by the device. The highest speed supported by the device-server communication protocol is 40.955 $\mathrm{m\,s^{-1}}$. This results in a time to cross t of $\frac{0.400}{40.955}$ s. A vehicle that passes the sensors at this speed, is registered as taking either 7 or 8 poll intervals to cross the sensor distance:

$$p_1 = floor(\frac{0.400/40.955}{0.00125}) = 7,\ p_2 = T * ceil(\frac{0.400/40.955}{0.00125}) = 8$$

This results in possible detected speeds v_1 and v_2

$$v_1 = \frac{d_{sensors}}{t_1} = \frac{d_{sensors}}{T * p_1} = 45.71,\ v_2 = \frac{d_{sensors}}{t_2} = \frac{d_{sensors}}{T * p_2} = 40.00$$

v_1 is above the 40.955 $\mathrm{m\,s^{-1}}$ speed limit set by the device-server protocol, so $v_2 = 40.00\ \mathrm{m\,s^{-1}}$ is the highest speed that can be measured.

The highest overestimation occurs at the lowest true speed for which the measured poll intervals can still be 8. This happens when the aliasing effect induced by the floor function in (6) is at its largest, as $\frac{t}{T}$ approaches 9. The time to cross t then approaches $9 * T = 0.01125$ and the real speed v_{real} approaches

$$v_{real} = \frac{d_{sensors}}{t} = \frac{0.400}{0.01125} = 35\frac{5}{9}\mathrm{m\,s^{-1}}$$

The maximum possible error approaches 12.5%.

$$error = \lim_{v_{real} \downarrow 35\frac{5}{9}} |\frac{v_{calculated} - v_{real}}{v_{real}}| * 100\% = |\frac{40 - 35\frac{5}{9}}{35\frac{5}{9}}| * 100\% = 12.5\%$$

5.2.3 Worst case underestimation

The largest underestimation occurs at the highest true speed for which the measured poll intervals can still be 8. This happens when the aliasing effect induced by the ceiling function in (6) is at its largest, as $\frac{t_{real}}{T}$ approaches 7. The time to cross t then approaches $7 * T = 0.00875$ and the real speed v_{real} approaches

$$v_{real} = \frac{d_{sensors}}{t_{real}} = \frac{0.400}{0.01125} = 45\frac{5}{7}\mathrm{m\,s^{-1}}$$

The maximum possible error approaches 12.5%.

$$error = \lim_{v_{real} \uparrow 45\frac{5}{7}} |\frac{v_{calculated} - v_{real}}{v_{real}}| * 100\% = |\frac{40 - 45\frac{5}{7}}{45\frac{5}{7}}| * 100\% = 12.5\%$$

5.2.4 Generalization

In the general case, the real time t_{real} it takes a vehicle to cross $d_{sensors}$ can be underestimated by t_{under} or overestimated by t_{over}. The values for these variables depends on the precise time respectively lost or gained by using the floor and ceiling functions in (7). If T divides t_{real}, both the ceiling and the floor function return the same number of intervals and there is no difference between the real and calculated times (and speeds). If T does not divide t_{real}, t_{under} and t_{over} can be found by calculating the 'remainder' of the floor and ceiling functions using the modulo operator.

$$t_{under} = t_{real} \bmod T \tag{8}$$

$$t_{over} = T - (t_{real} \bmod T) \tag{9}$$

The underestimation and overestimation error can be found by calculating the ratio between the t_{real} and the overestimated or underestimated time.

$$e_{over} = \frac{t_{over}}{t_{real}} * 100\%,\ e_{under} = \frac{t_{over}}{t_{real}} * 100\% \tag{10}$$

At a true vehicle speed v_{real}, the the total error range is simply the can be then be found by adding e_{over} and e_{under}.

$$error = e_{over} + e_{under} \tag{11}$$

The table below shows the detailed error ranges for some speeds close to standard Dutch motorway speed limits. Note that $v_{real} = 40.0$ results in a 0% error range, because the corresponding $t_{real} = \frac{d_{sensors}}{v_{real}} = \frac{0.40}{40.0} = 0.01$ can be divided by $T = 0.00125$. When v_{real} approaches such a value, the difference between e_{under} and e_{over} increases dramatically, as discussed in sections 5.2.2 and 5.2.3.

v_{real} ($\mathrm{m\,s^{-1}}$)	v_{real} ($\mathrm{km\ h^{-1}}$)	e_{under} (%)	e_{over} (%)	error (%)
9.00	32.4	1.25	1.56	2.81
14.0	50.4	0.63	3.75	4.38
23.0	82.4	0.63	6.56	7.19
28.0	100.8	5.00	3.75	8.75
39.9	143.64	12.22	0.25	12.47
40.0	144	0	0	0
40.1	144.36	0.25	12.22	12.47

5.3 RTC accuracy

In order to compute the speed of the passing vehicle the esp32 needs an accurate source of time as a reference for how much time passed between a vehicle appearing before the left and right sensor. We used the RTC Subsystem, which counts how many clock cycles have passed since boot, to compute how many microseconds have passed since boot, which was then used to get the relative time difference between two sensor data points.

Unfortunately, it quickly became apparent that the accuracy of the RTC Subsystem is directly bound to the accuracy of the 80Hz crystal on the esp32, which is not nearly accurate enough. To determine the factor by which the RTC clock was off, we performed a simple test where we recorded the raw sensor data and device time of the esp32 and then waved a hand in front of the sensor every 60 seconds using a stop watch on a mobile phone. Using the collected data shown in figure 8, we could see that for one of the ESP32 devices we used, the RTC timer was off by a factor of 1.162. The time between waves was consistently recorded by the ESP32 as being 69 seconds instead of 60.

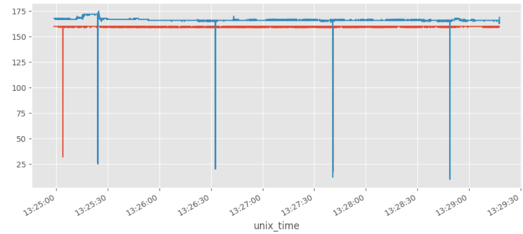


Figure 8: Raw sensor distance data with device time on x-axis.

To correct for this, a more accurate clock signal could be attached to the ESP32. However, in order to prevent the project from being even more delayed due to hardware issues we opted to fix it in software instead. This essentially means that every esp32 device now needs a unique `RTC_CORRECTION_FACTOR` constant that is used in all time calculations to correct for the real frequency of the esp32 crystal.

6 Global Design

This section provides justification and a simple overview of the major design choices made early on in the project.

6.1 Components

When composing the requirements, it quickly became clear that the final system would need four relatively distinct parts:

- an embedded system that collects and processes data about passing cars
- an API server that connects to a database to store data sent by the embedded system, and retrieve data requested by end users
- a database that stores the uploaded vehicle data
- a dashboard that shows relevant data in an organized, visual manner

In figure 9, these four elements are drawn in a simple diagram. The embedded system is pictured in slightly more detail. The diagram shows the ESP32 microcontroller at its core and its peripheral components: two time-of-flight sensors, an SD card, and an LTE-M modem.

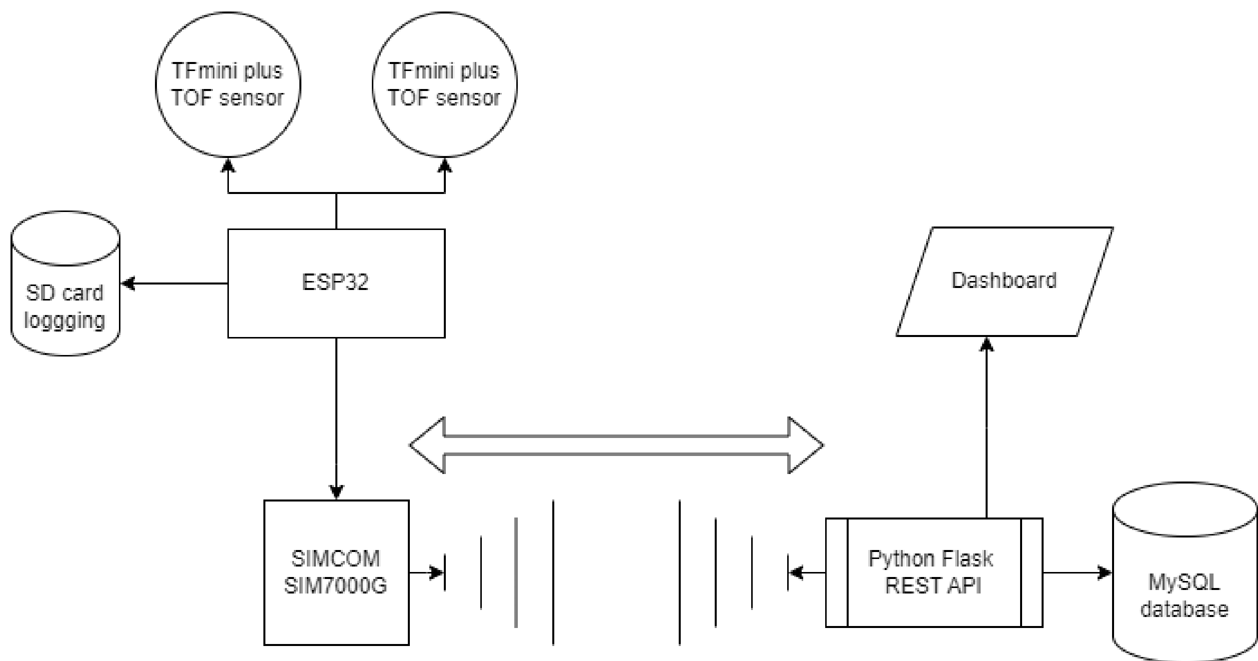


Figure 9: Simple global overview of the system

6.2 Programming languages and frameworks

When working under time constraints, it is especially important to choose the right tool for the job. One part of the team's toolkit was already decided by the client: the embedded system was to be programmed using Rust. This choice was justified by Rust's increased performance over MicroPython, which did not live up to the task in a previous prototype, and its focus on memory safety. The other tools were decided on by the team, and were mostly compared against others in learnability and familiarity. The latter is why the API server was written in Python's Flask microframework, as all team members had some experience with API development in its context. Additionally, the flexible and concise nature of python allowed for quick additions to the API, as extra paths were added for debugging or dashboard elements. The dashboard

itself was written in JavaScript using Vue.js, a front-end library designed for developing user interfaces and web applications. None of the team members were very familiar with front-end development, so Vue.js was selected based on its approachability and learnability.

7 Detailed design

This section goes into more detail concerning the design choices and architectures of the different components of the system.

7.1 Counter firmware

As mentioned previously the ESP32 has two processing cores. We assigned the second (APP) core the responsibility of reading and parsing the right sensor data as well as actually converting the raw detection data (mostly timestamps for what time cars appeared at what distance) into a full detection (actually useful data like car speed and length). This is conceptually the most simple as the second core essentially enters an infinite loop of waiting for serial data to be received, checking if the raw detection can be processed already and then waiting for data again.

The first core (CPU) does more work, being responsible for sending the periodic updates to the server as well as parsing and receiving the left sensor data. As can be seen in figure 10 rather than putting the data logging and heartbeat sending code inside their own interrupts we put them in the main loop. The main program loop is basically continuously checking if the send data or send heartbeat flags are set and will begin sending the data if so.

This is more complex than simply putting the send data/send heartbeat code directly inside the periodic timer interrupts that set the flags, however it is quite necessary as setting up the modem and writing the data to the SD card can take several seconds. If this were to happen inside of an interrupt the receive serial data interrupt would not be handled for several seconds (note that interrupt priority on the esp32 only indicates which interrupt should be handled first, not if they are allowed to interrupt each other; interrupts will only interrupt the main loop).

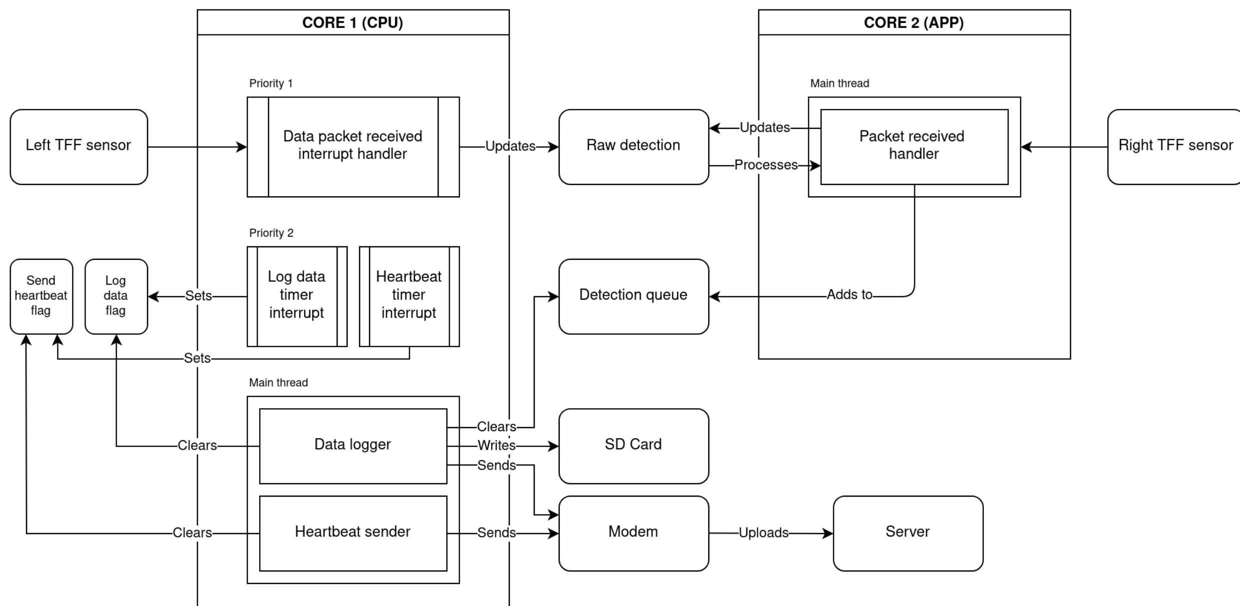


Figure 10: High level overview of counter firmware

Based on figure 10 it might seem like the APP core could be doing more, however the choice to keep the APP core from interacting with the rest of the system was a conscious one. Note how all other tasks performed by core 1 require interacting with the SD Card or Modem interfaces, which means that if any of those tasks were to be moved to the other core we would have to implement a multi-thread safe access mechanism for them. This would severely hurt performance for the reasons discussed in the next section. (Additionally, there is no performance actually going to waste on the APP core as it cannot even keep up with the sensor, as shown in section 8.1.2, so the added complexity would not improve performance regardless).

7.1.1 Safety vs performance

There were a few data structures which needed to be accessed from multiple points in the code, this meant we had to take concurrency safety into account. Even without freeRTOS or heap allocations embedded Rust has a `Mutex` system based on critical sections. This essentially means that in order to access the data within a `Mutex` you must enter a critical section. When inside a critical section the CPU will not stop execution to perform interrupts or let the other CPU core enter a critical section. This is useful for data which must be accessed by both cores, like the global `RawDetection` and `DetectionQueue`, however, because a core cannot enter a critical section while the other is inside a critical section this also means that that if one core is using the detection queue the other has to wait to access the shared `RawDetection` object even though the other core is not using it.

This could be addressed by implementing our own Mutex with read and write locking using atomic integers, however we did not pursue this approach because of time constraints. Instead we chose to make careful design choices such as only using a Mutex for data that had to be shared across both cores and avoiding entering a critical section wherever possible and trying to do as little processing inside as possible. Generally this means that we only enter a critical section to modify or copy a data structure and then immediately leave again.

There were a few instance were we opted to use a `SyncUnsafeCell` instead of a `Mutex` to deliberately bypass Rust's multi threading safety systems (and their effects on system performance) for certain data structures that we knew would only ever be accessed by a single core and never within interrupts.

In order to reduce the chances of a bug being created by the use of the `SyncUnsafeCell` we established a rule to only allow static `SyncUnsafeCell` variables to exist as private members of a sub module, as this would prevent them from accidentally being imported and used somewhere else in the code. This way only the safety of the code of the sub module had to be manually checked for rather than that of the whole program.

7.2 API Design

In order to provide an interface for storing and retrieving data of passing cars and detection devices, we designed a RESTful API. It was specified according to the OpenAPI 3.0 Specification. This OpenAPI specification is also used to generate documentation and way to test endpoints on our server at /api/docs, by using Swagger UI. The API provides endpoints for posting and fetching information about detections, devices, API keys, status updates and error messages.

7.2.1 API Keys

Whenever a device adds data to the database, this happens by means of a POST request containing either a batch of detections, or a status update. In order to prevent unknown parties from modifying the database, the relevant endpoints are protected by an API key security layer.

/devices/{device_id}/keys
Keys can be generated by sending a POST request to /devices/{device_id}/keys. Once a key is generated, it is exposed to the user once. From then, it can be passed in the 'Authorization' header in protected POST requests. The user can delete keys, access information about the API keys for a device, but will never again see the api key after its generation. Additionally, a GET request can be sent to this path to obtain information about available keys. If there are any API keys in the database for the provided device id, the server will return the key id and the date of creation for these keys.

```
[
  {
    "id": 1,
    "createdOn": "2009-07-13 16:32:02"
  }
]
```

Listing 1: Example body of a successful GET request response

/devices/{device_id}/keys/{key_id}
Using a key id obtained from a GET request to /devices/{device_id}/keys, the user can construct a DELETE request to remove it from the database.

7.2.2 Devices

This path contains information about all devices, as well as device-specific information.

/devices/
Sending a GET request to this path will return a list with general information of all devices, including the time they were last seen. This could be the last time the device sent a batch of car detections, or the last time it sent a status update.

```
[
  {
    "id": 10,
    "name": "test counter",
    "last_online": "2023-10-02 15:34:02"
  }
]
```

Listing 2: Example body of a succesful GET request response

Sending a POST request to this path will add a new device to the database. The only input needed is a name, a device ID is generated automatically by the database.

```
{
  "name": "test counter"
}
```

Listing 3: Example body of a valid POST request

/devices/{device_id}
Sending a GET request to this path will return device information of the device with its id equal to device_id.

```
{
"id": 10,
"name": "test counter",
"last_online": "2023-10-13 20:09:54"
}
```

Listing 4: Example body of a succesful GET request response

/devices/{device_id}/stats
Sending a GET request to this path will return simple statistics about the amount of cars the device with id equal to device_id has detected. It includes the total number of detections, and datetimes for the first and last detections.

```
{
  "detections": 24,
  "firstDay": "2023-10-04 15:34:02",
  "lastDay": "2023-11-27 09:18:55"
}
```

Listing 5: Example body of a successful GET request response

/devices/status
Sending a GET request to this path will return the following information about a specific device (if available): device ID, the timestamp of the last status update, the firmware ID, the last known latitude and longitude, and the last known state of charge.

```
{
  "device": 24,
  "received": "2023-11-10 17:26:15",
  "firwareID": "1.000.000",
  "latitude": 54.002,
  "longitude": 62.15,
  "SoC": 100
}
```

Listing 6: Example body of a successful GET request response

7.2.3 Detections

Devices send information about the cars they detect in batches to save bandwidth and energy. Individual batches are extracted by the API server and stored in the database. From there, they can be accessed through the /detections/ path.

/detections/
Car detections can be added to the database by sending a POST request to the server with a valid API key in the 'Authorization' header. The POST request should be of content type application/octet-stream and should contain information about cars detected in a certain interval.

A list of all car detections by all devices can be obtained by sending a GET request to this path.

```
[
  {
    "id": 1,
    "device": 3,
    "received": "2023-10-04 15:39:02",
    "vehicleType": "small",
    "direction": true,
    "speed": 54.5
  }
]
```

Listing 7: Example body of a successful GET request response

/detections/{device_id}
Sending a GET request to this path will return all detections uploaded to the database by the device with its device ID equal to device_id

```
[
  {
    "id": 1,
    "device": 3,
    "received": "2023-10-21 22:22:22",
    "vehicleType": "small",
    "direction": true,
    "speed": 54.5
  }
]
```

Listing 8: Example body of a successful GET request response

/detections/{device_id}/month/{month}
Sending a GET request to this path will return a list of days. These are all the days in the month specified by {month} for which the device with ID equal to device_id has recorded any detections. This is a useful way to find the last relevant data to display in the dashboard.

```
[
 1,
 2,
 13,
 19,
 23,
 31
]
```

Listing 9: Example body of a successful GET request response

/detections/{device_id}/month/{month}
Sending a GET request to this path will return a list of days. These are all the days in the month specified

by {month} for which the device with ID equal to device_id has recorded any detections. This is a useful way to find the last relevant data to display in the dashboard.

```
[
  {
    "id": 1,
    "received": "2012-12-12 06:43:13",
    "vehicleType": "small",
    "direction": true,
    "speed": 54.5
  }
]
```

Listing 10: Example body of a successful GET request response

7.2.4 Detection data over time

In order to facilitate drawing graphs of detection data over time, the API can provide data of user-specified resolution for user-specified intervals. The paths that implement this, take three extra query parameters: day, interval, and limit. The day parameter specifies the day on where the time series data should start. The interval parameter specifies the resolution of the time axis: it denotes how many minutes should be in one interval. The limit parameter specifies how many intervals should be in the resulting array.

/devices/{device_id}/counts?day={day}&interval={interval}&limit={limit}
Sending a GET request with the proper parameters will return a JSON object consisting of 4 arrays: { small: number[], medium: number[], large: number[], timestamps: string[] } Each of the first three arrays stores detections for a specific vehicle type, and each element in an array stores how many cars of a certain vehicle type were detected in an interval. The timestamps array stores the timestamps for which detection data was aggregated. For example: small[0] contains the number of small vehicles that were detected between timestamps[0] and timestamps[1]. The timestamps array starts at the provided start day, and then increments with the provided interval.

```
{
  "small": [
    5,
    9,
    10
  ],
  "medium": [
    2,
    1,
    4
  ],
  "large": [
    0,
    1,
    0
  ]
  "timestamps": [
    2023-11-05T00:00:00,
    2023-11-05T01:00:00,
    2023-11-05T02:00:00
  ]
}
```

Listing 11: Example body of a successful GET request response

/devices/{device_id}/speeds?day={day}&interval={interval}&limit={limit}
Sending a GET request with the proper parameters will return an object consisting of 3 objects: left, right and avg. Each object consists of 2 arrays: mean and v85. These arrays store the mean and v85 speeds for a specific direction, over a specified time range. The timestamps array stores the timestamps for which detection data was aggregated. For example: left['mean'][0] contains the average speed of vehicles that were going left between timestamps[0] and timestamps[1]. The timestamps array starts at the provided start day, and then increments with the provided interval.

```
{
  "avg": {
    "mean": [
      54,
      32.8
    ],
    "v85": [
      60.3,
      40.2
    ]
  },
  "left": {
    "mean": [
      54,
      32.8
    ],
    "v85": [
      60.3,
      40.2
    ]
  },
  "right": {
    "mean": [
      54,
      32.8
    ],
    "v85": [
      60.3,
      40.2
    ]
  }
  "timestamps": [
    2023-09-16T00:00:00
    2023-09-16T00:01:00
 ]
}
```

Listing 12: Example body of a successful GET request response

7.2.5 Debug

The API also includes some paths used for debugging purposes only.

/
Sending a GET request to this path will return a simple 'Hello, World!' message.

```
Hello, World!
```

Listing 13: Example body of a successful GET request response

/keys
Sending a GET request to this path will return a complete list with all information of every API Key currently in the database.

```
[
  {
    "id": 1,
    "apiKey": "e779f7bc-73d1-11ee-b962-0242ac120002",
    "createdOn": "2023-10-04 15:34:02"
  }
]
```

Listing 14: Example body of a successful GET request response

/errors
Sending a POST request to this path allows a device to store its error messages in the database for debugging purposes, as the on-board modem can be nontransparent and difficult to debug.

```
This is an error message
```

Listing 15: Example body of a valid POST request

/devices/{device_id}/errors
Sending a GET request to this path will return all error messages posted by the device with its ID equal to device_id.

```
[
  {
    "id": 23,
    "device": 1,
    "received": "2023-10-04 15:34:02",
    "message": "This is an error"
  }
]
```

Listing 16: Example body of a succesful GET request response

/devices/{device_id}/errors/{error_id}
Sending a DELETE request to this path will delete an error if its device ID equal to device_id and its error ID is equal to error_id.

7.3 Dashboard

After data collection and processing, the data must be accessible in a meaningful way. This is done with a web-based dashboard. The dashboard utilizes the RESTful API described in the previous section.

7.3.1 Functions

The dashboard must consist of a multitude of functionalities:

- Display statistics about the measured data from the API server per device
- Create new Device endpoints and corresponding Keys
- Control or modify existing Devices or Keys
- All in a meaningful and user friendly way

7.3.2 Framework

The dashboard is made in the Vue3 framework. Vue is a JavaScript framework that works well for building a user interface in our case, due to a focus on Singe page applications and compartmentalization of web components. In addition, extensions such as ChartsJS for displaying graphs and Leaflet are simple to integrate into this Vue framework; ChartsJS and Leaflet being the most important ones.

7.3.3 Chart.JS

To display the data we make use of the Chart.JS library which generates two graphs on the device page. The charts are first set up using Vue by defining the type of chart needed as a Vue component. In our case that is a line chart, this allows for the use of a line chart anywhere in the application. In the linechart component call <Linechart /> on the device page two parameters are required to generate a graph. That is "chartData" and "options", the chartData is pulled from the server and the options are pre-defined. The options specify the x-as and y-as labels and the type of data that will be shown. After pulling the data from the server a is added to the data points to ensure the data is visually distinguishable.

7.3.4 Leaflet

Vue leaflet is the JavaScript library that is used in the application to display a map with the location of the device. The device information in pulled from the database using the API. To display the device location the lat (latitude) and long (longitude) are used, if the device has not sent a status update "status unknown" is displayed. The lat and long is used to query OpenStreetMap which provides a human-readable address. This is displayed under that map along with other device information that is available on the server.

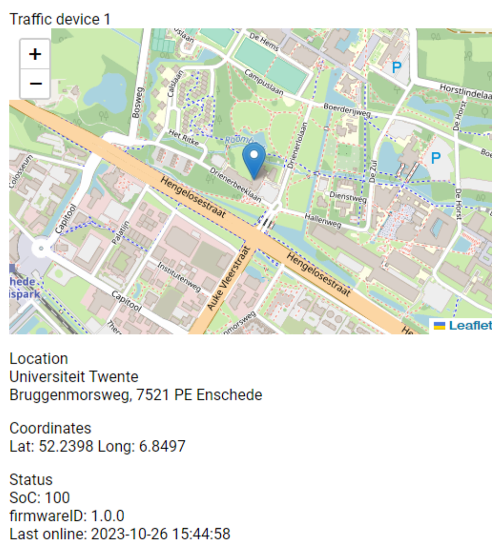


Figure 11: Device location known

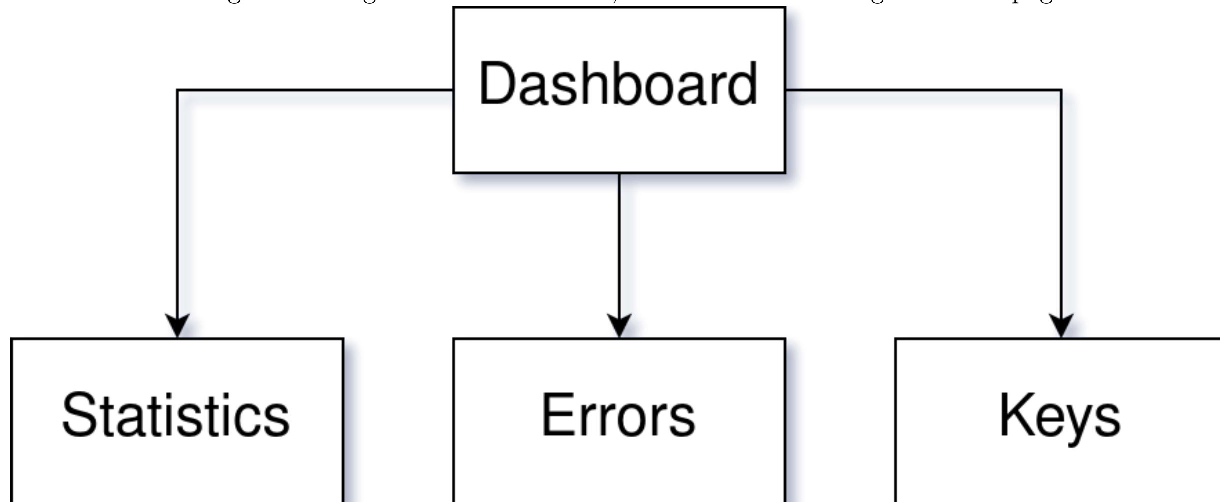
Figure 12: Device location unknown

7.3.5 User friendliness

Metrics for user friendliness of our dashboard is decided to be based upon three things: The simplicity of our website page tree, the small amount of clicks necessary to reach certain functionality and lastly, the overall readability of each web page.

Website page tree We opted for one page (the dashboard) that handles all devices themselves, a page where you can see the statistics of measurements per device, a page for keys and a page for errors. Because the latter three are bound per device, we wound that the overview of all devices seems like a convenient page to redirect the user to these specific pages of a device. This resulted in the following website page tree, seen in figure 13

Figure 13: Page tree of the website, with Dashboard being our index page



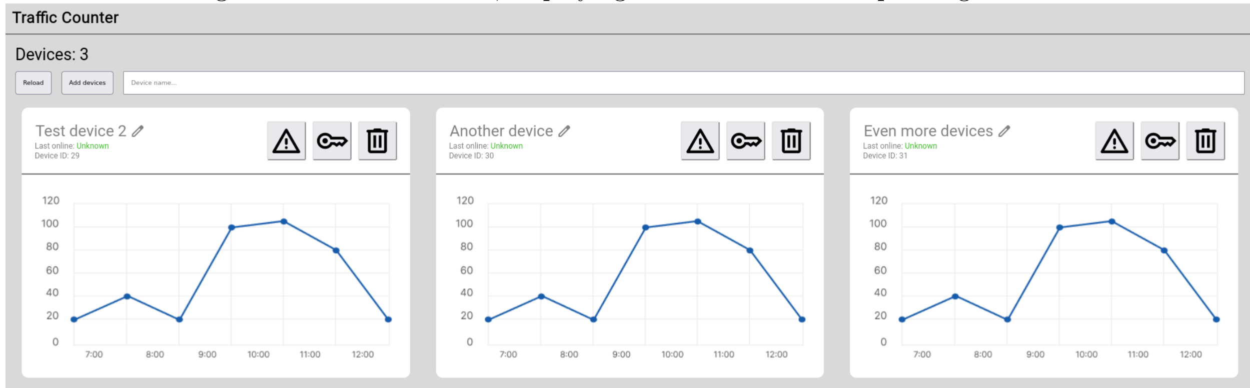
Dashboard page

On this index page, as shown in figure 14, you see a list of all devices, with each device displaying a device name, id, and last seen date and time. In addition, each device object on the web page will contain a few buttons to redirect the user to its corresponding page:

- The "errors" button:
  This button redirects the user to a page containing a table with all errors that have accumulated over the lifetime of the device.
- The "keys" button:
  This button redirects the user to a page containing a table with all associated API keys with this device.
- The "delete" button:
  This button will delete the instance of the device in the API server, in addition to all its corresponding statistics. To prevent accidental deletion, a pop up will appear after clicking to confirm whether the user indeed intended to delete this device.
- The "statistics" button:
  This button is disguised as a graph image to indicate that this will redirect the user to the device specific page that includes all statistics and other information.

In addition to visualizing already existing devices, the option exists to create new ones as well. This is done through the input at the top of the page where the user can enter a new device name, followed by clicking the "Add device" button or simply pressing enter. Lastly, devices can also be renamed by clicking the tag symbol next to the device names.

Figure 14: The dashboard, displaying 3 devices with corresponding buttons



Keys page (per device)

This page, as shown in figure 15 is for managing all API keys that the selected device has, by allowing the user to create and delete keys. Because the secrecy of keys ensures the security, this page will not show the actual keys, but only their corresponding ID. Only when the user creates a new key, he will be able to briefly see the key value through a pop up, but after that, this key value will not be recoverable anymore. Deleting a key will prevent this specific device from using this key to upload data to the API server.

Figure 15: The keys page, displaying all keys of this device

Traffic Counter

Keys for device: 10

Reload Add key

ID	Created on	Delete
7	Sun, 12 Nov 2023 20:54:01 GMT	
8	Sun, 12 Nov 2023 20:54:29 GMT	
9	Sun, 12 Nov 2023 21:01:27 GMT	
10	Sun, 12 Nov 2023 21:01:34 GMT	

Errors page (per device)

The errors page, shown in figure 16, will display all received errors from this device, convenient for debugging. These errors can be deleted if they are deemed unnecessary by clicking the "delete" button.

Figure 16: The errors page, displaying all errors of this device

Traffic Counter

Errors of device: 7

Back Reload

ID	Received	Message	Actions
516	11/4/2023, 5:13:25 PM	Device online!	
517	11/4/2023, 5:16:17 PM	Device online!	
518	11/4/2023, 5:16:37 PM	Saved 1 detections to sd card: Some(Detection { time: 27771413, speed: 1.0810635503208057, length: 0, going_left: false })	
519	11/4/2023, 5:20:03 PM	Device online!	
520	11/4/2023, 5:20:34 PM	Saved 2 detections to sd card: Some(Detection { time: 31373613, speed: 1.838970544289307, length: 0, going_left: false })	
521	11/4/2023, 5:23:43 PM	Saved 1 detections to sd card: Some(Detection { time: 248197946, speed: 2.5398921815768922, length: 0, going_left: false })	

Statistics page (per device)

The statistics page shown in figure 17 displays the data this device has measured in the form of graphs.

Figure 17: The statistics page, displaying all measurements of this device

7.3.6 User error prevention

To prevent users from making accidental mistakes, a few fail-safes have been implemented. These include:

- No name prevention
  Creating or renaming a device will fail if the entered device name is an empty string.
- Accidental device deletion
  Clicking on the delete button of a device will not only delete this device, but also all records this device made. In order to prevent the accidental deletion of this device and its corresponding measurements, a pop-up will appear asking the user for a confirmation if this button press was intended.

7.4 Detection algorithm

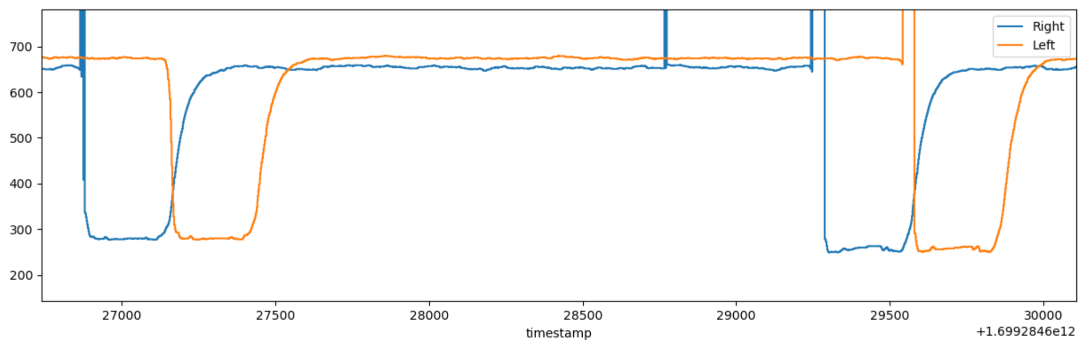


Figure 18: Raw sensor data recording of two passing objects.

7.4.1 Characterizing the sensor output

In order to develop an algorithm capable of detecting cars, we first had to understand the what the raw sensor data looked like. To achieve this we wrote a special firmware for the esp32 device that did nothing but parse the incoming sensor data and then send it back to the pc over a usb serial connection with a timestamp. We could then take the device and a laptop to a place near a busy municipal road and record the raw sensor data for later review.

The first discovery made was that when no car is being detected both sensors have a pretty consistent "rest distance" during which there are very few outliers or temporary changes in height. However a car detection is usually started with and followed by several "invalid" data points, (these are sensor data messages where the distance is 0 and the signal strength is less than 10). Additionally we also found out that the sensors' internal processing systems performs some kind of data averaging, the implications of which are discussed in the next section.

7.4.2 Sensor data averaging

Consider the sensor data of a car driving by shown in figure 19, ignoring the effects of the outliers for a moment we can get a good estimate of the speed at which the car was travelling based on the offset in the two graphs. The offset between the sensors was measured at 27 ms, while the distance between the sensors for this measurement was 350 mm. Which gives us the following speed estimate:

$$\frac{350}{27} \approx 12.96m/s \approx 46.66km/h$$

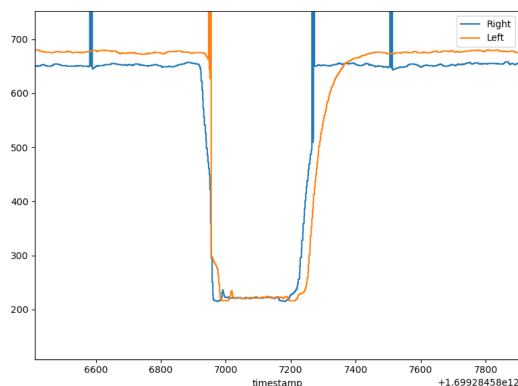


Figure 19: Raw sensor data at 500 hz of a car driving past, y-axis in cm, x-axis in ms.

Again ignoring the effects of the outliers and focusing only on the car leaving the left sensor (orange line) starting around t=7350, we can see there is a smooth transition in the data. This is not entirely unexpected as the manual[5] describes that when two objects partially cover the light cone of the sensor some value between the two distances will be output. What is unexpected however, is that it takes nearly 200 ms for the measurement to return to the "no car" distance of around 675 cm. Given that the car was detected at a distance of roughly 222 cm, the light cone should be around 12 cm[5], if the smoothing was entirely the result of the car partially covering the light cone that would mean it took 200 ms for the car to travel 12 cm. Which would mean that the car was driving at:

$$\frac{120}{200} \approx 0.6m/s \approx 2.16km/h$$

This is completely inconsistent with the more accurate speed we computed earlier and indicates that the sensors are internally computing an average value over a given time interval, although no such behaviour is explicitly documented in the manual[5]. Even accounting for the fact that the measurement in figure 19 was recorded at 500hz rather then the maximum 1000hz by assuming that each data point was actually the average of two 1000hz data points that would still only account for 2 ms of extra time rather then the full $200 - 27 = 173$ ms we found.

Initially it seemed that this data averaging performed by the sensor might not be an issue for speed calculations, since we compute the car speed based on the offset between the two sensor's data points and both sensors seem to perform the same amount of data smoothing and the offset would thus remain roughly the same. However the problem becomes significant when the effect of outliers or invalid measurements is taken into account. Notice how around 7255 ms in figure 19 an outlier occurs in the right sensor, which seems to reset the sensor's internal smoothing system causing the data points to instantly return to the newest value. This would be very problematic if we had chosen 600 cm as the distance to compute the offset between the graphs since the offset would appear to be almost triple the real value.

A similarly tricky situation can be seen with the outliers in the left sensor (shown in orange) around 6950 ms when the car enters. If we computed the offset between the graphs there it would seem that the car was traveling somewhere around the speed of sound. This means that the solution is not as simple as computing the offset in multiple places and taking the maximum or the minimum value.

7.5 Sensor rest position determination

When no car is passing by the sensor usually measures some far away distance, in the case of the sensor data recording in figure 20 this is about 8 meters. For ease of communication we will call this distance the "rest distance" from here on out. In order to know whether or not a change in the distance measurements constitutes a car entering before the sensor we needed to know the rest distance. This is more complicated than it sounds because the rest distance is different for every road and may even change slightly over time.

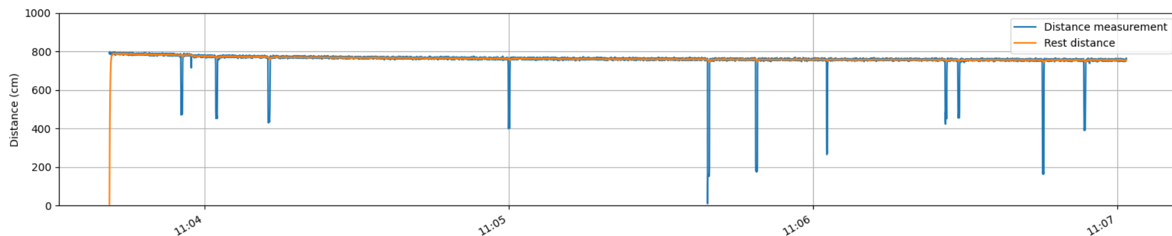


Figure 20: Simulation of the rest distance value generated by the algorithm when applied to raw sensor data. Note that the 0 distance outliers have been filtered from the graph.

The most simple and accurate way to compute the rest distance would be by simply keeping a few seconds of data in a buffer and determining the maximum value based on that. This would be very impractical on the esp32 however because at a sensor frequency of 900Hz even 5 seconds of data would take up $900 * 5 * 2 =$ 9000b= 9Kb of RAM. Not to mention the fact that several clock cycles would be wasted on linearly searching through the whole buffer for the maximum value every time we need to get the rest distance.

To address this we developed an algorithm that approximates the rest distance in a much more efficient way taking a sort of moving average while only using two atomic integers worth of RAM (one for each sensor). Essentially the rest distance is updated every time a sensor reading is received using equation 12.

$$rest_distance = \frac{rest_distance * (n-1) + d}{n} \tag{12}$$

The key innovation that makes this different from a normal moving average is the fact that the value of n changes based on the current rest distance and the incoming distance. If the current distance is bigger than the rest distance $n = 100$ is used, while $n = 1000$ is used if the difference between the current distance and the rest distance is less than 140 cm and $n = 1000$ if the current distance is closer.

The idea being that we usually expect the rest distance to only move further away from the sensor so if the current distance is further back than the rest distance we should update the rest distance very quickly, while if the current distance is closer than the rest distance this usually just means that a car passed in front of the sensor in which case we do not want to adjust the rest distance as much.

The one weakness of this approach is that any sensor data outliers that cause very far away values like the theoretical maximum of $2^16 = 65536$ cm would result in the rest distance nose diving as well, which would take several seconds to recover and might cause missed detections. Fortunately no such outliers have been found during the sensor data analysis as they usually take the form of 0 cm measurements that are easily filtered out. Though since our testing is limited by time constraints it is possible that this system would break on certain roads.

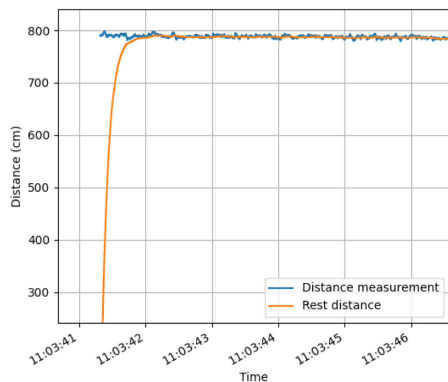


Figure 21: Rest distance quickly taking the correct value after startup.

7.6 Distance thresholds

With a system for obtaining the rest distance in place (as described in the previous section) we still needed to actually find the cars in the sensor data. To achieve this we divided the sensor detection area into virtual subsections of 50cm with thresholds at 50cm, 100cm, 150cm, 200cm, etc. When the sensor data gives a distance over the threshold this threshold (and all the ones below it) be marked as entered (if they were not already entered). And when the distance signal leaves the threshold it will be marked as left.

This system might seem convoluted compared to something more intuitive like only marking the time the signal becomes larger then the rest distance and the time the signal returns to the rest distance. However such an algorithm would not work since the data averaging performed by the sensors and described in section 7.4.2 would skew the length measurement. Using the threshold system we can take the enter/leave times of the threshold between the resting distance and the closest threshold the car entered to get a time value less affected by the data averaging. This is technically still less accurate than keeping the full last three seconds worth of sensor data in a buffer and computing the enter/leave times based on the true middle between the rest distance and the car distance, but this requires far less computational power and RAM.

8 Testing

In order to ensure the release of a robust system, it will need to be tested extensively. The system is clearly divided in three parts which can be tested separately: the embedded system, the API server, and the dashboard. Testing strategies will differ per component: the embedded system lacks the resources for proper unit testing, and will be tested using integration tests. The API server will be evaluated using automated integration testing with the pytest framework.

8.1 Embedded system

The complete embedded system could not easily be unit tested as a whole because of all its integrated parts like the modem and the sensors. To address this each part was individually tested during development and code was written with careful consideration of the data sheets of the peripherals. The modem boot sequence, for example, performs several checks to ensure that the internal state of the modem was as expected and will retry command with a short delay upon failure.

8.1.1 Car detection algorithm

The most critical piece of embedded code, that also proved to be the most difficult to write and validate, was the car detection algorithm. The final version of the algorithm involved dozens of functions that all had to read or manipulate quite complex data structures. To ensure the correctness of these functions and allow for easy testing and development of the algorithm, the code for the car detection algorithm moved to a separate rust crate that could than be imported by the firmware as well as by a "testing program" that could then import the exact same code and run unit tests for all the functions. This also allowed for the algorithm to be developed and very quickly run and tested on several minutes of pre-recorded data on a fast desktop PC, before being tested outdoors on the real hardware with live sensor data.

8.1.2 Sensor metrics

Methodology To determine how many sensor packets were actually processed every second by the device we created a profiling module (found in `src/profiling.rs`) that is hidden behind the `profiling` feature flag. This means no performance profiling code is included in the binary when this flag is disabled and no performance overhead is induced by metric calculations during normal operation.

The profiling module is relatively simplistic, two atomic counters are incremented every time a packet is successfully parsed. The sensor metrics are then computed at the start of the data logger, this has very little performance overhead because most of the computations can be freely interrupted. The counters are always updated using the `Ordering::SeqCst` ordering, which means that no packet is ever not counted.

8.2 Detection accuracy

Recall that one of the quality requirements specified in section 4.2 is that the system must correctly detect 90% of passing cars. (Passing cars here referring only to the cars on the lane the sensor is installed on, not the cars on the other lanes). To evaluate this requirement we tested the system on a roadway

8.2.1 Methodology

To obtain a set of ground truth measurements to compare the device detections against we placed a DLSR camera on the other side of the road to film the road at 50 fps. By using the known distance between two chalk lines on the road and a set of image processing Python notebooks we created we could extract the speed of the passing cars from the recorded video. To illustrate what the sensors see and help with debugging the algorithm we also recorded the raw sensor data (shown in the bottom graph of figure 23) with a second esp32. Note that because of lens distortion and motion blur it is difficult to determine exactly when a car crosses the chalk lines so the camera detections should not necessarily be interpreted as a 100% accurate ground truth but rather an approximation of the real value with some error.

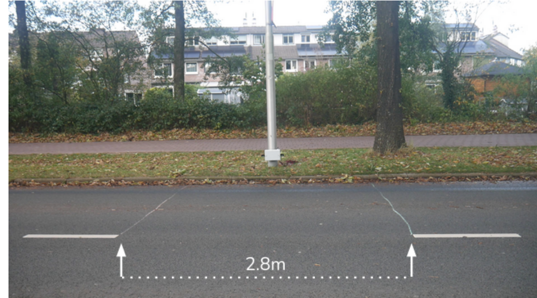


Figure 22: Chalk lines on road as seen from camera opposite to sensor.

8.2.2 Results

On the 16 cars in the evaluation recording there was an average error of 1.19 ± 5.26 km/h between the counter and camera speed detections. The error notably does not skew towards any direction, which likely indicates that the error originates from measurement inaccuracy rather than a fault in the algorithm.

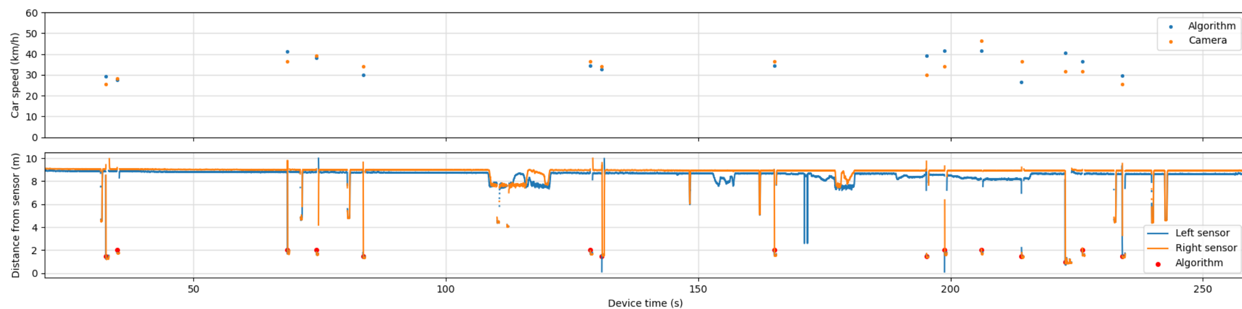


Figure 23: Speed measurements (top) and raw sensor data (bottom).

8.2.3 Conclusion

100% of the cars on the right lane of the road were detected (the cars on the other side were also detected but discarded because they were not in the right lane), this means we have reached our quality requirement. While we had no quality requirement for the accuracy of the measurements, 1.19 ± 5.26 km/h of error in speed is also perfectly sufficient. While it might not seem great at first glance it is important to remember that individual detections is not what the system will be used for. The end users are mostly interested in the road statistics in chunks of days or hours, which will be more accurate since the error is not consistently skewed in one direction and will thus average out. Indeed even in the 4 minutes of recording used in this evaluations the average speed detected by the counter and by the camera differ only 1.2 km/h (being 34.9 km/h and 33.7 km/h respectively), this is well within the margin of error of our camera car detection algorithm.

8.3 API integration testing

Another integral part to the system was the link between the embedded system and the dashboard: the API server. In order to ensure that the API worked according to specification, every response code of every path had to be tested. Since proper testing of some functionalities requires a non-empty database, it was necessary to send POST requests before being able to test a unit. That is why many test cases test both POST and GET requests in one, resulting in integration testing. These tests were ran inside the pytest framework, which provides a simple way to send HTTP requests to a test instance of the Flask API server. This instance works together with a separate testing database. The database is constructed from a docker compose file that uses the same database schema as the production database, but without persisting data. API keys and test devices are inserted into the database before tests that require them, and deleted directly after the tests are ran. For the API integration test, the code coverage goal was set to 100%.

Coverage report: 99%

coverage.py v7.3.2, created at 2023-11-08 17:54 +0100

Module	statements	missing	excluded	coverage
C:\School\Module 11\project\traffic-counter-monorepo\api-server\src\app.py	18	0	2	100%
C:\School\Module 11\project\traffic-counter-monorepo\api-server\src\auth.py	25	0	0	100%
C:\School\Module 11\project\traffic-counter-monorepo\api-server\src\counts.py	41	0	0	100%
C:\School\Module 11\project\traffic-counter-monorepo\api-server\src\database.py	27	0	3	100%
C:\School\Module 11\project\traffic-counter-monorepo\api-server\src\detections.py	61	0	4	100%
C:\School\Module 11\project\traffic-counter-monorepo\api-server\src\devices.py	102	0	0	100%
C:\School\Module 11\project\traffic-counter-monorepo\api-server\src\errors.py	8	0	6	100%
C:\School\Module 11\project\traffic-counter-monorepo\api-server\src\keys.py	37	0	0	100%
C:\School\Module 11\project\traffic-counter-monorepo\api-server\src\routes.py	11	0	11	100%
C:\School\Module 11\project\traffic-counter-monorepo\api-server\src\speeds.py	45	0	0	100%
C:\School\Module 11\project\traffic-counter-monorepo\api-server\src\swagger.py	4	0	0	100%
C:\School\Module 11\project\traffic-counter-monorepo\api-server\src\utils.py	7	0	0	100%
__init__.py	0	0	0	100%
conftest.py	51	0	0	100%
test_auth.py	13	0	0	100%
test_counts.py	83	0	0	100%
test_detections.py	73	0	0	100%
test_devices.py	157	0	0	100%
test_keys.py	56	0	0	100%
test_speeds.py	97	0	0	100%
testutils.py	79	2	0	97%
Total	**995**	**2**	**26**	**100%**

coverage.py v7.3.2, created at 2023-11-08 17:54 +0100

Figure 24: pytest coverage report

8.3.1 Results

The integration testing for the API was extremely useful throughout the entire development process. It allowed for test-driven implementation of API features, which helped catch a variety of bugs. One major problem with the device-server communication was the way the MySQL database misinterpreted the time zones of incoming python datetime objects. The test cases caught incongruities between expected outcomes and actual outcomes, leading to the problem being investigated: the MySQL database automatically appends its own, local, time zone to incoming datetime objects if no timezone is provided. The problem was solved by adding UTC+0 time zones to datetime objects before inserting them to the database. Additionally, the tests caught countless data formatting mistakes, and exposed insufficient user input sanitation. The test coverage reached 100% for the relevant source code. Note that some statements were excluded from the test coverage report. This was done because they either contained debug-only code, or, in the case of database.py, they

are only executed in case internal error handling fails. This should not occur under any circumstances, test cases included.

9 Future Planning

Although we are happy with the minimum viable project we have built, there are some areas in which it should be improved before it could be turned into a commercial product.

- API Server
  - **Authentication:** To avoid wasting any time on unneeded features, it was decided that this minimum viable product would not include any authentication, except for the endpoints used by the counter which require an API key. This is obviously unsafe for any real product which is why a login system or OAuth integration would be valuable.
- Dashboard
  - **Live feed:** In order to give the user feedback in what is currently measured by a device, a live feed could be implemented. Once use case for this live feed would be an easier way of debugging. The downside of this live feed is a significant higher data usage, resulting in more costs, so this was one of our lower priorities to implement.
- Embedded counter system
  - **Long term validation:** Because of time constraints caused mainly by the hardware availability issues we did not get a chance to validate that the embedded system keeps functioning over the course of multiple days or weeks. Although the choice to use Rust and no heap allocations should result in a very stable system it would be wise to actually validate this.
  - **SD card settings:** The counter settings like API key and network APN are currently hardcoded in the device itself. It would be much more user friendly to load these configuration settings from a settings file on the SD card.
  - **RTC self calibration:** Since the clock on the ESP32 is not exactly 80Hz each counter needs a correction factor for its time calculations, this factor is currently measured by hand and hardcoded in the firmware. It would be nice if counters could use their GPS time to compute their own correction factor and store it in their flash storage.
  - **Auto APN detection** The modem currently requires the APN (Access Point Name) of the mobile network it needs to connect to. However it might be possible to automatically detect the APN which would mean the end user does not have to manually look up and provide the APN in the config, which would in turn create one less chance for a setup mistake to be made.

10 Conclusion

To conclude this report we will revisit the requirements from section 4.1 and mark whether or not they have been met by the final system.

10.1 Must

✓ The system must be able to detect passing vehicles

✓ The system must be able to categorize detected vehicles into three length classes (small, medium, large).

✓ The system must be able to detect the travel direction and speed of passing cars.

✓ The system must store all detected cars in an online database.

- ~ ~~The system must use the precompiled firmware for the AFBR-S50 time of flight sensor because the laser is otherwise not guaranteed to be safe.~~
- ✓ The system must display the recorded data to an end user in some way.

The system fulfills all the must-have requirements, with the exception of including precompiled firmware for the AFBR-S50 sensor. This requirement naturally became obsolete when the client changed the assignment to use different sensors due to problems with hardware delivery. The implementation of the features specified in all other requirements in this section can be found in the report.

10.2 Should

- ✓ The dashboard should be able to display the car detection statistics based on the logged data.
- ✓ The system should encrypt communication between the hardware and the backend (e.g. SSL).
- ✓ The system should be able to communicate the hardware state (battery SoC, firmware version, GPS location) to the backend server.
- ✓ The central server should be able to handle multiple embedded devices that send data.

The final product fullfils all the should-have requirements. The encryption requirement is met by hosting the API and dashboard on an SSL-certified server. Multiple embedded devices are supported, but not extensively tested due to limited hardware availability. One important note for the hardware state communication: the communication protocol is in place, but the current system sends placeholder data in the SoC field. This is because the microcontroller does not have access to a Battery Management System.

10.3 Could

- × The system could give live updates/individual car detections (via MQTT/tcp sockets).
- ✓ The system could store detected cars in a local SD card.
- × The system could support over the air firmware updates of the embedded hardware.
- × The system could detect cars on a second road behind the primary road.
- × The system could have tamper detection via the accelerometer (against vandalism).
- × The system could have a Microsoft Excel export feature for the recorded data.

The only could-have implemented was SD card data logging. This feature proved to be very useful for debugging and was implemented relatively early in the development process. The other features were considered too time-consuming to gain priority over refining the implemented must-haves and should-haves.

10.4 Won't

- ✓ The system will not have power saving features like putting the microcontroller to sleep until a car comes near.
- ✓ The system will not be able to detect or count pedestrians or cyclists.

These requirements were deemed out of scope by the client and subsequently not implemented.

10.5 Quality Requirements

- ✓ The communications over the SIMCom component of the embedded system should use less than 500MB per 10 years when it counts less than 2.9 million cars that year, based on the assumption of 8000 cars per day on average. This leaves 17.12 bytes per detection on average.
- ~ The system should correctly detect more than 90% of passing cars.

The bandwidth requirement was met, as the implementation does not differ from the protocol that is discussed in the preliminary research section. For the last quality requirement: in the live test, the system did detect cars with a success rate of 100%, but this test was conducted in daylight and dry weather. To investigate the robustness of the system, tests during nighttime and different weather circumstances are required.

11 Discussion

The system is supposed to be able to withstand any type of weather, and be able to function for large amounts of time without maintenance. One thing that still needs to be verified is the rigidity of the system, by which we mean how well the system withstands outside forces like weather and how well it handles its function over extended periods of time. The system was tested mostly during dry and partly sunny/cloudy weather. We have experienced difficulties with measuring passing cars during rain, so we assume the same difficulty holds for heavy rain, hail, snow and fog. Unfortunately, due to constraints that come with the nature of this project, deadlines and other limitations, we were unable to properly test during different weather types. This should be done thoroughly before this system can turn into a full product.

Appendices

References

[1] Wesley Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293. URL: https://www.rfc-editor.org/info/rfc9293.

[2] Gert van der Maten. “Geen camera’s maar wel snelheidsmetingen en verkeerstellingen op de Oudegracht in Alkmaar”. In: *Noordhollands Dagblad* (2022). URL: https://www.noordhollandsdagblad.nl/cnt/dmf20220426_87047434?utm_source=google&utm_medium=organic.

[3] Nasko. URL: http://netsekure.org/2010/03/tls-overhead/.

[4] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[5] *TFmini Plus LiDAR module Short-range distance sensor*. SJ-GU-TFmini-Plus-01 A02. Benewake (Beijing) Co., Ltd.