University of Twente

Thesis: UAV-based data streaming for remote processing and real-time information retrieval

Marc Bouma

March 29, 2025

Abstract

Unmanned Aerial Vehicles (UAVs), commonly known as drones, are increasingly used in a variety of applications. This thesis explores if 4G/5G would be a viable option to send data from the drone to a server/workstation where the data can be used for further data processing. In this thesis, the estimation of a vehicle's speed was considered as use case.

A system was designed to send images with Real-Time Streaming Protocol (RTSP) and telemetry data using the Message Queuing Telemetry Transport (MQTT) protocol from the drone to a server/workstation. This RTSP method is reliable and has a latency (time needed to send and fully receive the image) of around 0.45 seconds for a data stream of 10 FPS with an image size of 724x1280 pixels.

These received images and telemetry data are then used in algorithms to estimate the speed of the vehicles that can be seen in the images. This is done by making a 3D map with ORB-SLAM,3 which is connected with 3D GPS coordinates, so a sparse point cloud is created, and these 3D points are in GPS coordinates. Then, the vehicle's position is estimated by detecting the objects with YOLO and tracking these objects with a Bot-SORT tracker. The pixel values of these vehicles are converted to 3D points with the help of the 3D map, and these 3D locations are then used to estimate the speed of the vehicle. This implementation is expandable with multiple drones, and the map creation can even be combined with the help of ORB-SLAM3.

Experimental validation at 120 meters altitude demonstrates that the system that was designed is capable of sending images and telemetry data using 4G/5G. This system proposes a scalable and flexible alternative to traditional static infrastructure, enabling real-time traffic analysis and other remote sensing applications.

Contents

1 Introduction

A drone, also known as an Unmanned Aerial Vehicle (UAV), is a device that is operated without a human pilot. It can be remotely controlled with the help of sensors like GNSS, comes in various shapes and sizes, and they are increasingly used for a large number of applications. Examples of such applications are surveillance, environmental mapping, search-and-rescue missions, agriculture, and construction inspections. (1)

As drones do not have a human pilot, there are many safety concerns if the device loses control or is taken over. Drones make use of flight controllers, which are used in combination with controllers on the ground so a person on the ground can control the drone, but what happens once this connection is broken or, worse, the drone is taken over by another person. These are valid concerns, but these concerns are mitigated by the fact that flight controllers need explicit messages to perform a task, and these messages are different for every session/drone you have. If a wrong message is sent to the flight controller, it will fly the drone back to its start location where it took off with autopilot where it hopefully will not crash somewhere. In addition, there is always a risk that an unwanted party connects to the remote control and then uses these for their own goals. However, the implementation of an encrypted control message and return to home fail-safe decreases the risk of someone being able to hijack the drone and control it.

Another source of problems is given by GNSS spoofing that "alters" the correct positioning of the drone (i.e. it looks like the drone is in another location), giving information that can endanger the flight. The GNSS signal can also be jammed, which can cause the drone to lose control or return home.

Drones collect data with the help of sensors for different applications. For example, surveillance cameras are used, and the data from these cameras are processed. This processing is mostly done in post-processing, which means that the data is collected, and afterwards, the data as a whole is used for processing. It is also possible to process the data in real-time to retrieve the information immediately. So, there are two ways to do this:

- 1. Processing the data on board (using onboard processing units).
- 2. Doing the remote processing, the data needs to be sent during the flight from the drone to a server on the ground, which does the processing.

This processing in Real-Time/close to real-time is of importance for applications like surveillance and rescues where immediate analysis is of core importance. However, the problem with method 1 is that for heavy data processing, it is not possible to do the processing on board, and therefore method 2 with transferring the data in real time to a server/workstation requires a stable and efficient communication with the ground.

Communication between UAVs and ground stations was traditionally guaranteed by radio or wireless communications. However, these solutions have shown several problems of poor connection, especially in complex environments such as urban areas where obstacles forbid their efficient use. In the last few years, UAVs have been increasingly used within an Internet of Things (IoT) network, where the UAV can communicate with other devices or users over an internet connection.

The use of 4G/5G can, therefore, be considered an alternative, but the question remains if sending data with these networks is sufficient for this task. As there is not always coverage for a 4G/5G connection, these networks are optimized for on-the-ground use. So, the question is if the network is still able to send and receive data at high altitudes (between 50 and 150m above the ground).

An example of an application for this real-time data transfer with 4G/5G could be surveillance of the roads to track and estimate the speed of the cars. This task was previously done manually by the police force, but recently, it all became more automated as the use of cameras and video surveillance increased. However, this last solution still requires infrastructure to be built and maintained, which can be very costly. Drones, on the other hand, offer a flexible and scalable alternative. As it is capable of covering large areas and can be deployed in all kinds of environments. By using the images and telemetry data from the drone, the speed of the seen vehicles can be estimated, and a lot more can be done with the gathered information afterward.

For this to work, some challenges need to be tackled, such as the data transfer in real time to a server/-workstation from the drone with a reliable stream of data with minimal transfer time (time needed to send and fully receive the data). Once the data is received on the server/workstation this data needs to be analyzed with algorithms that make use of the images and telemetry data to assess the speed of the vehicles which present computational and algorithmic challenges. This thesis aims to address these challenges by designing and assessing a reliable and fast data transfer system and making an algorithm to estimate the speed of vehicles.

The project has supported the development of a dedicated research undertaken by the Dutch Drone Delta (2). This research aimed to assess the reliability of 4G data streaming in urban environments and assess the use of this network to replace the classical radio remote control of drones in case of emergency. The consortium, composed by KPN, NLR, and the Amsterdam Drone Lab, among others, has completed several tests in the Marine Terrain of Amsterdam. The most meaningful results of these tests are an integral part of this thesis.

1.1 Problem Statement

In this thesis, the focus will be on sending data with 4G/5G from the drone to the server so it can be used to estimate the speed of vehicles.

Data transfer is critical for applications that require real-time decision-making, such as surveillance, search-and-rescue missions, and emergency response. To do this effectively, the data transfer needs to be reliable and needs to have a low transfer time. This presents multiple challenges as the 4G/5G network is optimized for ground level, and the drone will be flying high, but also the network latency and bandwidth limitations are a challenge.

Traditional methods to measure the speed of the vehicles often rely on a static camera or static sensors, which limit the use case of these sensors and the infrastructure. While drones can be multi-use and are not static.

This research focuses on solving two key problems:

1. Reliable Data Transfer: How can a reliable data transfer method be implemented on the drone with a minimal transfer time so the server/workstation can do a close to real-time analysis of the data?
2. Car's speed Assessment: How can the images and telemetry data from the drone be used to estimate the speed of the vehicles?

The aim is to show that a drone can send data to a server effectively and that the speed of the vehicles can be estimated with this information. If this is successful, this could further increase the use of drones and demonstrate the potential of the current communication infrastructure.

1.2 Thesis objectives

This project aims to estimate the speed of vehicles depicted in the images by sending the data from the drone to a server/workstation and then processing the data on this device. The sub-objectives for this goal are:

1. To develop and improve the UAV data streaming to enable efficient data streaming from drones to UT servers enabling the transmission of telemetry data and images.
2. To test the existing 4G network (in the Netherlands) and its speed in data streaming and carry out experiments that benchmark the performance of the designed system.
3. Implement the algorithms (to run on the remote server) to detect and track cars and estimate their speed by leveraging both images and the drone's telemetry.

By achieving these objectives, this thesis will show that it is possible to send data effectively from the drone to a server and use this information to estimate the speed of vehicles, demonstrating the potential of 4G/5G connections for real-time data collection and analysis.

1.3 Research questions

Based on the sub-objectives from before, some research questions are formed, and these research questions are:

- What is the best protocol to send the telemetry data and images from the drone to the workstation? (from sub-objective 1)
- What metrics are important for this data streaming? (from sub-objective 2)
- How can these metrics be measured? (from sub-objective 2)
- What algorithms are needed to estimate the speed of a car? (from sub-objective 3)
- How reliably can a vehicles speed be estimated by these algorithms? (from sub-objective 3)
- Is this approach fast enough to run in real-time? (from sub-objective 2 and 3)

1.4 Outline

This thesis starts in Chapter 2 with the state of the art on video streaming, telemetry streaming, linking the received telemetry and image data, estimating the speed of the vehicle with object detection methods and tracking methods, converting the 2D pixels value to a 3D point, methods of making a 3D map, the speed estimation calculations, and the time synchronization between the drone and server/workstation.

This is followed up by Chapter 3, the methodology, which explains how everything is implemented and why these methods are chosen. This chapter is divided into two parts: the drone and the server.

Then, in Chapter 4, the tests to benchmark the performance of the designed system are explained, and in Chapter 5, the results of these tests are shown. These results are discussed in Chapter 6, where conclusions are made, challenges and changes during the thesis are mentioned, and finally, the future work is mentioned.

2 State of the art

In this chapter, an overview of the background and state of the art is given. In the following, both video streaming (section 2.1) and telemetry data streaming (section 2.2) are considered. The possible methods to link the telemetry data and images are reported in this chapter too (section 2.3). Beyond data streaming, this chapter discusses object detection (section 2.4.1), tracking methods (section 2.4.2), multiple methods to get the 3D location (section 2.5), and the different methods to calculate the speed (section 2.5.2). Finally, possible methods for time synchronization between devices are discussed (section 2.6).

2.1 Video streaming

Multiple protocols are available for sending and showing the live feed of a video. The two largely used protocols are RTMP and RTSP. An example of how to stream a video from a drone and live view it can be seen in (3) where, the video is compressed and sent with the real-time messaging protocol(RTMP). Where it can be sent to the server and the live feed can be shown. RTMP is also used in the following paper (4). where RTMP is used to send images from the drone to a server so the users can detect animals with the help of some algorithms. In (5) Real-time streaming Protocol (RTSP) is used for a smart security and live video surveillance system.

Both RTSP and RTMP are based on the TCP and UDP protocols. Transmission Control Protocol (TCP) is one of the main protocols of the Internet. TCP provides reliable, ordered, and error-checked delivery of a stream of data. TCP is connection-oriented as the sender and receiver need to be connected before the data can be sent between them. A "handshake" between the sender and receiver is used for this. This handshake, combined with the retransmission and error detection, adds reliability to the connection but also increases the latency, which is not optimal for some real-time applications (6). UDP User Datagram Protocol is one of the core communication protocols of the Internet to send messages to other hosts. UDP is a connectionless protocol, meaning that it does not need a handshake like TCP does. This means that the data is being sent even without knowing if a connection is made, which means that there is a possibility that some data will be lost. Time-sensitive applications often use UDP because dropping packets is better than waiting for delayed packages due to retransmission and not being in real-time anymore (7) (6).

2.1.1 RTMP

Adobe's Real-Time Messaging Protocol (RTMP) was designed for high-performance transmission of audio, video, and data between Adobe Flash Platform technologies. It is mainly used for low-latency video delivery and for broadcasting live content to media servers. RTMP uses TCP for reliable transmission, and it provides a bidirectional message multiplex service over a reliable stream transport for both control commands and data messages transmission over the same TCP channel through a fixed port. An RTMP connection begins with a handshake to establish a stable connection between the server and viewer (3).

A typical delay for RTMP would be around 5 to 30 seconds, but this can be reduced by using H264 encoding. The delay measured with this H264 encoding ranges between 137.48 ms and 146.02 ms (8). Some disadvantages that RTMP has are that it requires Flash Media Player, which is not widely supported now, and it does not have HTML5 support.

2.1.2 RTSP

Real-time streaming Protocol (RTSP) is an application-layer protocol used to control the transmission of real-time data (8) (9) (10). It is non-connection-oriented, but it can operate over TCP to control the session if necessary, and Real-time Transport Protocol (RTP) over UDP is used to transmit the video and audio data (9). RTP is a network protocol designed to deliver audio, video, and other real-time data over the internet. It divides the data into chunks, adds timestamps or sequence numbers to it, and sends these chunks with UDP, as mentioned earlier (11).

UDP is preferred because it is considered the optimal streaming protocol compared to TCP because it has less delay. To make it more reliable, TCP uses retransmission and an “addictive increase, multiplicative decrease” sliding window to control the congestion. This makes it less suitable for real-time communication, but if reliability is a big concern, there is still the option to implement TCP (8).

RTSP is designed to control and deliver real-time media streams, for example, for video surveillance. In other words, RTSP acts as a "network remote control" for multimedia environment (10) and is commonly used for IP cameras, security systems, and IoT video. This can also be seen in the following papers (5) and (12) where RTSP is used for surveillance systems.

2.2 Telemetry streaming

Telemetry data needs to be sent from the drone to the server. Telemetry data is GNSS data, IMU values, and time of measurement. During my ASP(Academic skills project) my assignment was to synchronize the images that are taken on the drone with the telemetry data. This information is then set up in a way that the drone continuously captures data and creates images with metadata of the telemetry data or an image and a JSON file with the telemetry data. This data package needs to be sent to the server, and this can be done with multiple protocols. In the following sections, some of these protocols are reported.

2.2.1 AMQP

Advanced Message Queuing Protocol (AMQP) is a protocol specifically developed for IoT (Internet of Things) applications. It has reliable queuing and uses publishers and subscribers, which makes it very scalable. However, it is limited because it is less suitable for resource-constrained environments and real-time applications (13).

2.2.2 HTTP

Hypertext Transfer Protocol (HTTP) and web sockets are commonly used to send and receive data. HTTP is, in fact, the foundation of data communication for the World Wide Web. HTTP is designed to make communication between two systems at a time using its client-server-based model. It has high overhead and relies on continuous connections. Which makes it less suitable for constrained environments and real-time applications (13).

2.2.3 CoAP

The Constrained Application Protocol (CoAP) is specifically made for IoT (Internet of Things) applications for data sharing, applications, and services. As it makes use of UDP with lightweight communication with minimal overhead. For low-power, resource-constrained networks.

CoAP predominantly functions via UDP, but it also offers the possibility of utilizing CoAP over DTLS over UDP to increase the security. Employing CoAP over TCP increases the overhead, adds round trip delays, uses more RAM, and increases the resources needed. Because of this, CoAP over UDP remains the preferred option for IoT devices.

Also, CoAP relies on best-effort delivery, and this can result in unreliable message transmission, which can cause problems in applications where message delivery is critical (13).

2.2.4 MQTT

Message Queuing Telemetry Transport (MQTT) makes use of a publish/subscribe model with a central broker in the middle that manages the messages between publishers and subscribers. MQTT is built on TCP, providing reliable message delivery with Quality of Service (QoS) levels.

QoS is set up between sender and receiver with the broker in the middle, and MQTT has 3 QoS levels that define the level of delivery guarantee, where 0 means that the message needs to be sent at most once and is not retransmitted if lost. 1 means that the message needs to be sent at least once and retransmitted until it is successfully received. On the opposite, 2 means that the message needs to be sent exactly once, so no duplicates are received (14).

MQTT is renowned for its simplicity, energy efficiency, and small memory usage. It is secured with the help of authentication, authorization, and data confidentiality. Because MQTT makes use of TCP, it is possible that it needs more resources. Which makes it more difficult in constrained environments compared to UDP-based protocols like CoAP (13).

2.2.5 MQTT broker

The communication needs to be secure, and this can be done by using TLS Encryption and Client certificates for authentication. For this to work, a broker needs to be found, or an own broker needs to be made.

Possible choices for brokers are HiveMQ and Mosquitto. HiveMQ is paid, and Mosquitto is free since, with Mosquitto, you need to set up the broker yourself. It must be noted, however, that this also needs additional infrastructure to work, which in turn costs money. In addition, HiveMQ is more scalable than Mosquitto, it has some extra security features, and the documentation from HiveMQ is professional and has tutorials. The documentation of Mosquitto is community-based, it still supports TLS/SSL encryption and client certificates for security but it also needs maintenance. HiveMQ has higher availability and fault tolerance than the standard Mosquitto broker. The time needed to implement a own broker is very extensive and not compatible with many projects such as this one (15) (16).

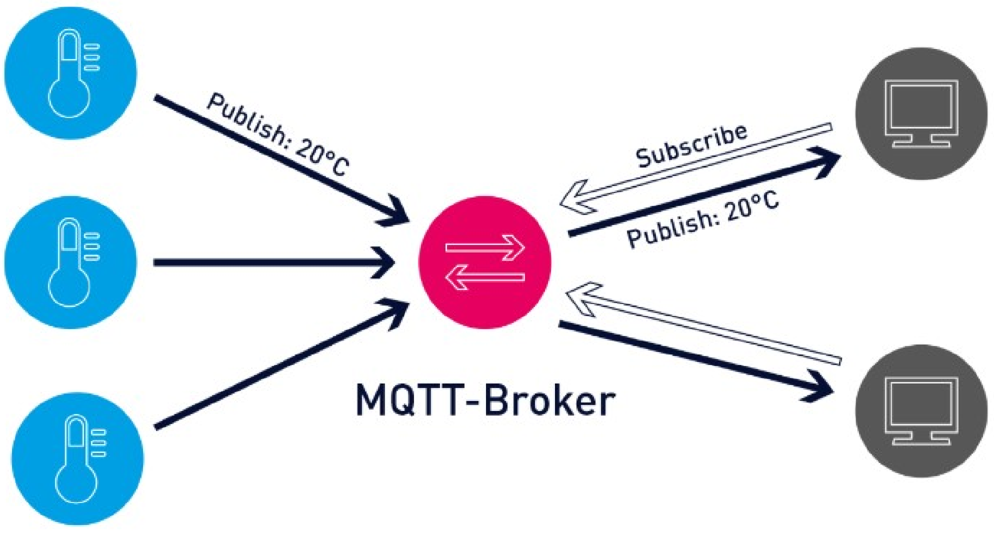


Figure 1: Scheme MQTT source (17)

In the figure above Figure 1, the working of MQTT with the broker is depicted. The publishers send data to the broker, and the subscriber subscribes to the MQTT topic and gets the data because the MQTT broker publishes the data to the subscriber.

2.3 Linking the files

As mentioned earlier on, the telemetry data and images need to be linked to correctly run the algorithm. Although these files are sent separately. To link the images and telemetry files, a common denominator needs to be determined. For this project, the time of acquisition of the image was chosen as this common denominator with which you can check if you have the right image with the right telemetry file. To link the images and telemetry files, multiple methods are possible: and the most promising methods are discussed below.

A method called Least Significant Bit (LSB) could be used to embed the timestamp of when the image is taken into the image itself. This is done on new blank pixels underneath the original image.

Another method would be to put the timestamp as text onto the image and use Optical Character Recognition (OCR) to extract the timestamp (18).

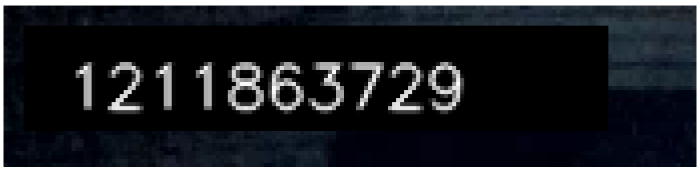


Figure 2: OCR example

An example of this can be seen above in Figure 2, where new pixels are made on these pixels, and the timestamp is set with the use of text.

An additional option is to use another version of LSB, where the most significant bit (MSB) would be used to transfer data as this bit would not easily change because of encoding, decoding and conversion.

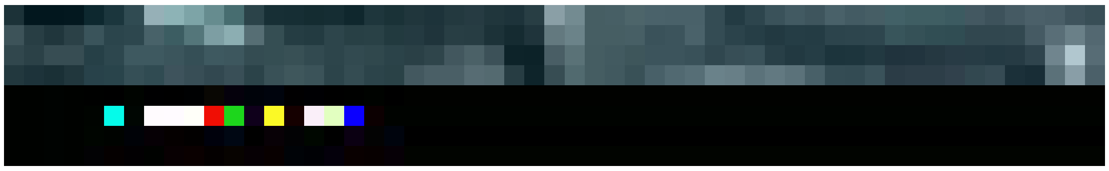


Figure 3: MSB example

An example of this can be seen above in figure 3, where some pixels are added underneath the original image, and the pixel values are changed to encode the timestamp into it. This is done by converting the time in an integer and then converting this integer in a 52-bit long binary value. These binary values are then used to change the R, G, and B channels of the new pixels underneath the original image. As shown in Figure 3, where the time is embedded into the image using the pixels.

Then, to link the telemetry data and the images, the timestamp needs to be collected from the image, and the telemetry data. The time in the image is extracted from the pixels and the time is a part of the telemetry data. How this is done in detail is later explained. So, if these times are the same, the files are connected/linked.

Then, the algorithm can be run with this telemetry data and the image.

2.4 Speed estimation

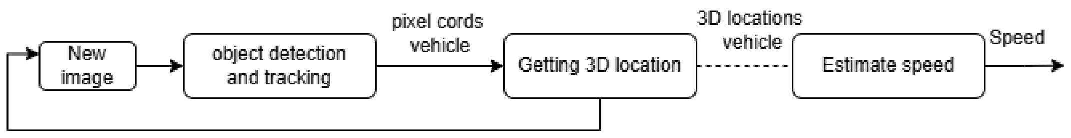


Figure 4: Scheme Speed estimation

The estimation of the speed of a vehicle can be done once the locations of the vehicle are known. Once these locations are known, they can be used over time to get the speed.

However, the vehicles need to be detected and tracked using some algorithms to get its location. The object detection algorithm gives the pixel values of where the vehicles are detected. These vehicles are given IDs and are tracked in time.

Then, the pixel coordinate needs to be converted in a 3D geolocalization of the object. To do this the localization and attitude of the drone need to be retrieved first and then the position of the car can be inferred thanks to 3D projective geometry. In particular, this is done by projecting the vehicle in the image to get its point position into the 3D space. This ray is obtained with the use of the inverse of the intrinsic parameters matrix and the rotation and translation matrix of the camera.

All these elements will be further explained in this chapter.

2.4.1 Object detection

To detect and track the vehicles in images, detection and tracking algorithms are used. Object detection is the process of getting the location of certain objects in the image. This is most commonly done with the help of bounding boxes. There are several approaches for object detection, but this section will focus on the most famous methods.

Fast RCNN
Fast RCNN is an improved version of RCNN(Region-based Convolutional Neural Network) and is designed to detect objects. Fast R-CNN integrates feature extraction and object detection into a single, unified network, enabling shared computation and faster training. Fast RCNN makes use of Region Of Interest (ROI) pooling. It takes the entire image as input and generates pooling feature maps. Each of these features in the pooling feature map is an ROI. The created ROI map is then used with three fully connected layers for object classification. This can also be seen in Figure 5 (19).

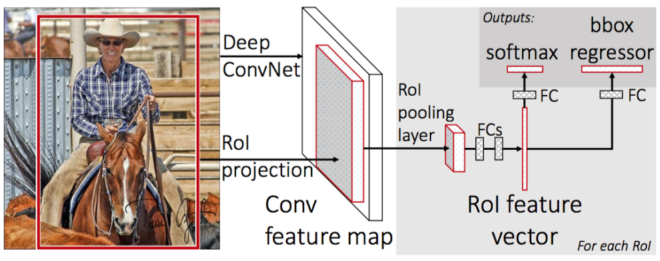


Figure 5: Fast RCNN source (20)

Faster R-CNN

Faster RCNN, in turn, is an improved version of Fast RCNN and is designed to be faster and more accurate than its predecessor. This is possible thanks to the use of the Region Proposal Network (RPN), which is a fully connected CNN that efficiently generates region proposals. Therefore, there are no separate proposal generation methods such as selective search. This makes Faster RCNN significantly faster and betters the accuracy. The RPN then scans the feature maps from the CNN and gives the potential object regions, which are further refined and classified. The structure of this can be seen in Figure 6. Faster RCNN is approximately 10 times faster than Fast RCNN with VGG16 network and Nvdia K40 GPU with a testing time of 198ms (19) (21).

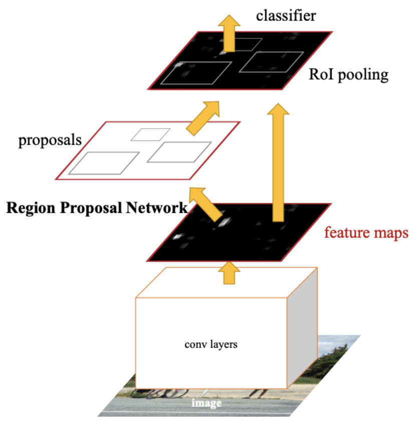


Figure 6: Faster RCNN source (21)

Yolo

Despite Faster RCNN's improvements, it was still not fast enough for some applications. This, in turn, sparked a search for a faster solution. YOLO has seen many versions, and these versions are discussed below.

YOLOV1 is "You Only Look Once" also known as YOLOs first version. YOLO is a real-time object detection system. YOLO processes the entire image in a single pass through the single convolutional neural network to predict the bounding boxes(position) and class probabilities for all the objects in the image (22) (23). The conventional methods needed to run hundreds or thousands of times per image and the more advanced methods divide the task into two steps like R-CNN, where the first step detects possible regions with objects or region proposals and the second step runs a classifier on the proposals (24). The single pass makes it significantly faster than its predecessors, but it comes at a cost of some accuracy. As the localization error was larger than state-of-the-art methods like R-CNN. This is because it faced challenges regarding localizing objects with aspect ratios that were not present in the training data but also could only detect up to two objects of the same class in the same grid cell (24).

However, with newer versions of YOLO, the accuracy increased significantly. YOLO is widely known and used for object detection. An example is (25) where it is used for weed detection, but it can be used in a variety of real-time applications such as autonomous driving and video surveillance and so much more (19) (26) .

After YOLOV1 YOLOV2 was created and YOLOV2 incorporated several improvements, including: Batch normalization to stabilize training and improve convergence, a high-resolution classifier for better feature extraction, passthrough layers bring fine-grained spatial information from earlier layers to later layers, helping with small object detection, multi-scale training to make YOLOV2 robust to different input sizes. This was done by training the model randomly, changing the input size. Anchor boxes to improve localization accuracy. Anchor boxes are boxes with predefined shapes used to match the prototypical shapes of objects.

These changes made the model more robust and increased the accuracy (24).

After YOLOV2, YOLOV3 was made. The main changes made compared to YOLOV2 are that a new backbone network is implemented, and the introduction to multi-scale predictions is added to the network. This new backbone, called Darknet-53, is a larger feature extractor composed of 53 convolutional layers with residual connections. Multi-scale Predictions are analyzing three different-sized grids. This helps with gathering finer detailed boxes and significantly improves the prediction of small objects, which was one of the main weaknesses of the previous versions of YOLO. Some other changes are that YOLOV3 predicts an objectness score for each bounding box. This score is 1 for the anchor box with the highest overlap, and the other anchor boxes have a score of 0. So, YOLOV3 only assigns one anchor box to each ground truth object. Instead of using a softmax for classification, it now uses binary cross-entropy to train independent logistic classifiers, which makes multilabel classification possible. This means you can assign multiple labels to the same box like a person can be labeled as a man and a person at the same time (24).

YOLOV4 tried to find the optimal balance by experimenting with many changes categorized as bag-of-freebies and bag-of-specials. Integrating bag-of-freebies (BoF) for an Advanced Training Approach. It also introduced features like mosaic data augmentation besides regular augmentations such as random brightness, contrast, scaling, cropping, flipping, and rotation. This technique combines four images into a single one, allowing the detection of objects outside their usual context and also reducing the need for a large mini-batch size for batch normalization. On the other hand, bag-of-specials are methods that slightly increase the inference cost but significantly improve accuracy. Examples of these methods are those for enlarging the receptive field, combining features, and post-processing, among others. The best-performing architecture was a modification of Darknet-53 named CSPDarknet53-PANet-SPP: CSPDarknet53 enhances gradient flow and reduces redundancy, PANet improves feature fusion for better object detection, and SPP (Spatial Pyramid Pooling) expands the receptive field. Apart from these changes, YOLOV4 also used self-adversarial training. This means that the model is made more robust by executing an adversarial attack to think the object is not present in the image anymore. The model is then retrained with the correct label to improve the capability to detect objects under difficult circumstances, which should make the model more robust. Genetic Algorithms are also used to optimize the hyperparameters such as learning rate, momentum, and activation function (24).

YOLOV5 was a significant advancement for object detection as it was developed in Pytorch instead of Darknet, which made it more accessible and flexible for researchers and developers. It also had multiple versions named n,s,m,l, and x. These versions made it possible to use different configurations of hardware and still run it. Even if this was at the cost of performance. All this combined made YOLOV5 a versatile solution for both research and practical applications (27) (28).

YOLOV6 outperformed previous state-of-the-art models on accuracy and speed metrics. This is because it uses a more fully convolutional design and replaces CSPDarknet. It replaced CSPDarknet with EfficientRep, which improved feature extraction. It also introduced RepVGG-based re-parameterization, which enhances inference efficiency while maintaining training-time flexibility. Just like YOLOV5, YOLOV6 has multiple models named n,s,m, and x for different hardware configurations. (24)

YOLOV7 surpassed all known object detectors in speed and accuracy in the range of 5 FPS to 160 FPS. Like YOLOv4, it was trained using only the MS COCO dataset without pre-trained backbones. YOLOv7 proposed a couple of architecture changes and a series of bag-of-freebies, which increased the accuracy without affecting the inference speed, only the training time. Compared to YOLOv4, YOLOv7 achieved a 75% reduction in parameters and a 36% reduction in computation while simultaneously improving the average precision (AP) by 1.5% (24) (29).

YOLOV8 offered cutting-edge performance in terms of accuracy and speed. Building upon the advancements of previous YOLO versions. YOLOv8 also supports multiple vision tasks, such as object detection, segmentation, pose estimation, tracking, and classification. It incorporates an anchor-free split Ultralytics head and state-of-the-art backbone and neck architectures and offers optimized accuracy-speed tradeoff, making it ideal for diverse applications. Evaluated on MS COCO dataset test-dev 2017, YOLOv8x achieved an AP of 53.9% with an image size of 640 pixels (compared to 50.7% of YOLOv5 on the same input size) with a speed of 280 FPS on an NVIDIA A100 and TensorRT (24) (30).

YOLOV9 introduced Programmable Gradient Information (PGI) and the Generalized Efficient Layer Aggregation Network (GELAN). PGI preserves essential data across network layers, while GELAN optimizes parameter utilization and computational efficiency. So, these are used to address the problems of information loss in deep neural networks. This ensures efficiency, accuracy, and adaptability. Like the versions before it, it has different models and the YOLOv9c model uses 42% fewer parameters and has 21% less computational demand than the YOLOV7 AF, although it achieves comparable accuracy. The YOLOV9e model has 15% fewer parameters and has 25% less computational need than YOLOV8x, although it has a 1.7% increase of AP. So, it achieved enhanced efficiency without the loss of precision (31).

YOLOV10 eliminates the need for non-maximum suppression (NMS), by constantly using dual assignments during training. It also optimized model components for superior performance with reduced computational overhead (reduces inference latency). YOLOv10 outperforms previous YOLO versions and other state-of-the-art models in terms of accuracy and efficiency. YOLOv10b has 46% less latency and 25% fewer parameters than YOLOv9-C with the same performance. And YOLOv10s is 1.8x faster than RT-DETR-R18 with similar AP on the COCO dataset (32).

The latest version of YOLO is YOLOV11. It has significant improvements in architecture and training methods compared to its predecessors. It uses an improved backbone and neck architecture, making feature extraction better and making object detection more precise. With these improvements in the model's design, YOLOV11 can achieve a higher mean Average Precision (mAP) on the COCO dataset, although it uses 22% fewer parameters than YOLOv8m. So, it is more computationally efficient without a decline in accuracy. YOLOV11 also supports object detection, instance segmentation, image classification, pose estimation, and oriented object detection (OBB) (33).

2.4.2 Object Tracking

YOLOv8 extends its detection capabilities by providing a range of trackers. Two popular options amongst these are Bot-SORT and ByteTrack. All the trackers are customizable, and users can fine-tune parameters like confidence threshold and tracking area (34). But these are not the only possible object-tracking algorithms there are also some based on deep learning.

ByteTrack
ByteTrack is a simple, fast, and strong multi-object tracker (35) (36) Multi-object tracking (MOT) aims at estimating bounding boxes and identities of objects in videos.

ByteTrack works in two stages: it first associates high-confidence detections with existing tracks using a tracking-by-detection framework and then incorporates low-confidence detections to recover missing objects.

Traditional tracking algorithms often discard low-confidence detections, which can result in fragmented tracks or missed objects. But by this use of low-confidence detections, ByteTrack achieves superior performance compared to other tracking methods (35) (36).

Bot-SORT Tracker
Bot-SORT is a fast and efficient multi-object tracking (MOT) algorithm. It is an extended version of BYTETracker, designed for object tracking with ReID (Re-identification) and GMC (Global Motion Consistency) algorithms. Bot-SORT works in two stages: it first uses a Kalman filter to predict the location of the object in the next frame using the motion history. Then, it uses the Hungarian algorithm to match the new destinations with existing tracks to ensure continuity.

Unlike traditional tracking methods, which rely on motion information, Bot-SORT leverages both motion and appearance data. Which increases tracking accuracy. Because of this, Bot-SORT is suitable for real-time applications for things like video surveillance and crowd monitoring where accuracy is critical (37) (38).

Compared to the Bytetracker, it is a bit more intensive to run because of the ReID (Re-identification) and GMC (Global Motion Consistency) algorithms but has an increased tracking accuracy.

Deep learning-based Tracker
Fully-Convolutional Siamese Networks (SiamFC) is a deep-learning object tracker that tracks an object by using similarity learning with the help of a fully convolutional Siamese Network. The advantage of it being fully convolutional is that the sizes of the candidate and input images can be different.

Then, the cross-correlation of the two inputs is commutated, which generates a map. In this map, the peak value indicates the new location of the tracked object. This will be done for every image so that the object is tracked over time (39).

In the image below Figure 7, this is also shown in a schematic and compared to the other trackers mentioned earlier. This SiamFC tracker is more suitable for single-image object tracking than multi-object tracking.

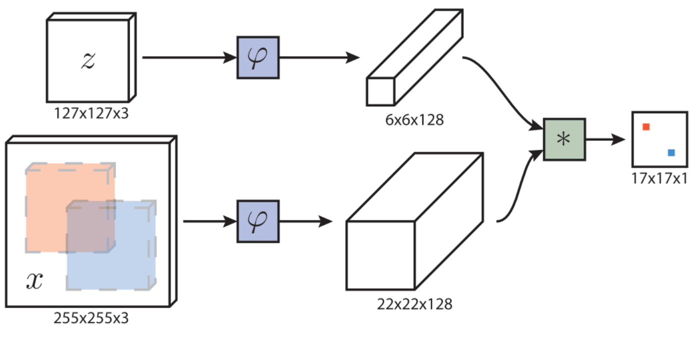


Figure 7: Fully-Convolutional Siamese Networks (SiamFC) source (39)

2.5 Converting 2D pixel to 3D point

To convert a 2D pixel point to a 3D point, a projection can be done.

To do this back projection, the following formulas are needed:

$$K = \begin{bmatrix} f & 0 & u_0 \\ 0 & f & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

Where K is the intrinsic parameter matrix with f being the focal length and u_0 and v_0 together are coordinates of the principal point. The principal point is the coordinate of the optical center in the image. It is usually around the center of the image.

Extrinsic parameters are Rotation matrix R and Translation vector T in reference to the external reference system.
The rotation matrix R can be obtained with the gyroscope and accelerometer values from the IMU. Using sensor fusion techniques such as Kalman filtering, it is possible to obtain a rotation matrix. (40)

The translation vector T can be made by interpolating the IMU measurements and the GNSS data so the location of the drone can be determined at the time the image is taken. This translation vector T can also be determined using sensor fusion algorithms like the Extended Kalman Filter.

Pixel coordinates(x) can be gotten from world coordinates(X) and a projection matrix named P.

$$x = PX \tag{2}$$

Where P is given by:

$$P = K \cdot R[I| - T] \tag{3}$$

To get the 3D point from a 2D point, the following formulas can be used:

$$ray_{camera} = K^{-1} \cdot x \tag{4}$$

This formula gives a ray/direction in the camera frame. To transform the ray to the world frame, a rotation is needed.

$$ray_{world} = R^T \cdot ray_{camera} \tag{5}$$

The origin of the ray is at the Translation matrix T. ray_{world} gives the direction of the ray, and the formula for the full ray is then:

$$ray = T + s \cdot ray_{world} \tag{6}$$

Where s is the length of the ray.

Then the full formula is:

$$ray = T + s \cdot (R^T \cdot (K^{-1} \cdot x)) \tag{7}$$

(41) (42) (43)

So, if the intrinsic parameters and the rotation and translation matrix's are known of the camera, it is possible to project a 2D pixel into a 3D ray. Without depth information, the point can lay anywhere along this 3D ray. Once the depth is known, the 3D point can be extracted.

2.5.1 Getting 3D position

To convert 2D pixels to 3D points, multiple solutions exist. These multiple solutions are explained in this section.

Standardized object scaling
For example, the pixel sizes can be discovered by using standardized objects. What is meant by this is that the objects' dimensions are known and used to extract the pixel sizes in centimeters or meters. This way, the pixel size can be converted, and the relative 3D location can be calculated. This is then only the relative 3D position, and no GNSS coordinates are needed.

An example of this is given in the figure below in Figure 8, where the assumption is made that all cars are 180 cm wide so the pixel size can be calculated:

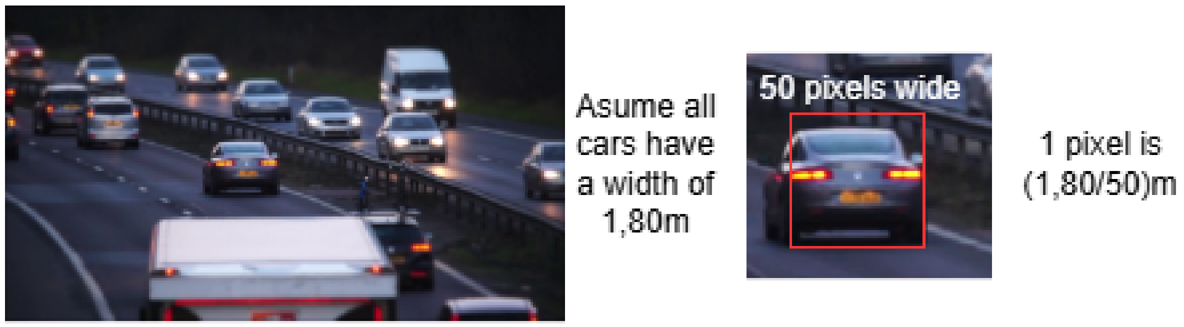


Figure 8: Standardized object scaling

Single image depth estimation
Single image depth estimation(SIDE) is a method to estimate the depth of a single RGB image. For each pixel, it gives a depth value.

Humans use the following cues to estimate depth:

- Occlusion, covered objects are further away
- Perspective: the same objects can look visibly larger as it is closer to the camera, the texture of the objects also increases once the distance increases.
- Atmospheric cue: objects get blurry as they are further away.
- Patterns of light and shadows, objects casting shadows onto another object
- Height cue: objects closer to the horizon seem farther away.

Advancements in deep learning have made it possible to use visual cues like the ones stated above. As neural networks can learn and predict depth by analyzing patterns, shading, and other contextual cues in the image.

Convolutional Neural Networks (CNNs) and Transformers are trained on large datasets with known depth maps to learn the relationships between image features and depth. A lot of training data is needed for this. These models generalize to predict new images. But if these new images contain new or not-so-detailed patterns, it is difficult for the neural network to get an accurate depth estimation (44).

An example of this is shown in the figure below Figure 9 where the depth information is reconstructed from a single image:

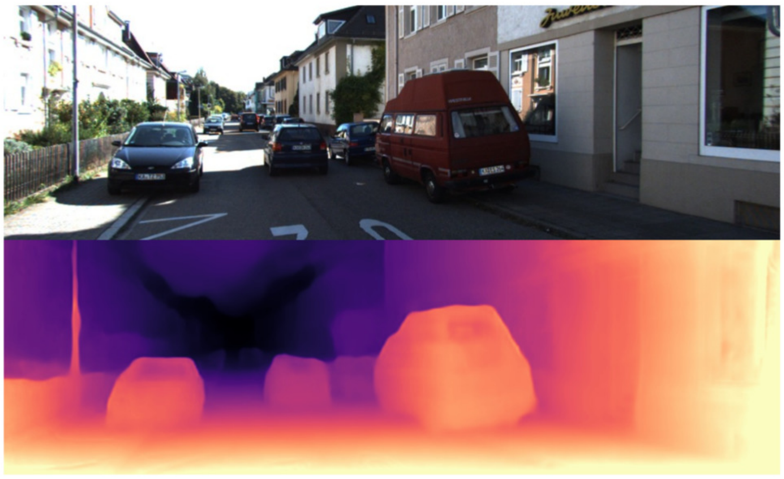


Figure 9: Single image depth estimation source (45)

Structure from Motion (SfM)
Another option is to generate a 3D point cloud using structure for motion(SfM) in post-processing from a sequence of images. This way, a 3D map is made, and with the help of back projection of the pixel coordinate, the 3D location of the vehicle (or any object) can be extracted from the generated point cloud.

Structure from motion works as follows: it makes use of 2D images to reconstruct 3D points. For this to work, the same features need to be found on multiple images. This is done by first extracting features(edges, corners, etc) and then matching them between overlapping images. Then, the camera pose needs to be estimated, and the relative position and orientation of the camera for each image are calculated based on the matched features. Once this is done and the camera position and orientation are known, the 3D reconstruction of the matches' feature points can be triangulated by solving geometric equations (as described above).

Then, the bundle adjustment is run to refine the 3D points and camera poses for the best possible accuracy (46) (47).

An example of this is shown in the figure below in Figure 10:

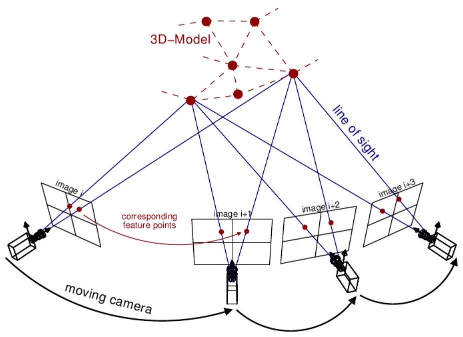


Figure 10: Structure from Motion source (48)

SLAM

Simultaneous Localization and Mapping (SLAM) (49) is a technique used for mobile robots to build and generate a map from, an unknown environment it is exploring. While simultaneously it also localizes itself inside this generated map. SLAM operates in real-time in contrast to SfM which is done during post processing, SLAM processes sensor data from for example, LiDAR, cameras, IMU or other sensors to estimate motion and to reconstruct the surroundings. To do this SLAM uses feature extraction, sensor fusion, and optimization methods such as loop closures to minimize drift over time. SLAM is widely used in robotics and autonomous navigation.

ORB-SLAM3

The most prominent example of SLAM is ORB-SLAM3 which is a cutting-edge open-source Simultaneous Localization and Mapping (SLAM) system. It is the first system able to perform visual, visual-inertial, and multi-map SLAM with monocular, stereo, and RGB-D cameras, using pin-hole and fisheye lens models. It is a system that operates robustly in real-time, in small and large, indoor and outdoor environments, and is 2 to 5 times more accurate than previous approaches (50).

A key feature of ORB-SLAM3 is ORB-SLAM Atlas. It is the first complete multi-map SLAM system able to handle visual and visual-inertial systems in monocular and stereo configurations, can represent a set of disconnected maps, and apply to them all the mapping operations smoothly: place recognition, camera re-localization, loop closure, and accurate seamless map merging. This allows the automatic use and combination of maps built at different times.

This is handy because when it gets lost, it starts a new map. This new map can be seamlessly merged with previous maps when revisiting mapped areas. Which makes it able to survive long periods with poor visual information.

Also, ORB-SLAM3 is the first system able to reuse all previous information in all the algorithm stages. ORB-SLAM3 is as robust as the best systems available in the literature and significantly more accurate (50).

It generates a 3D point cloud as the map. An example can be seen in Figure 11 where on the left the points are shown in red with the image path in green, and on the right side of the image green boxes can be seen are features with the corresponding image at the time:

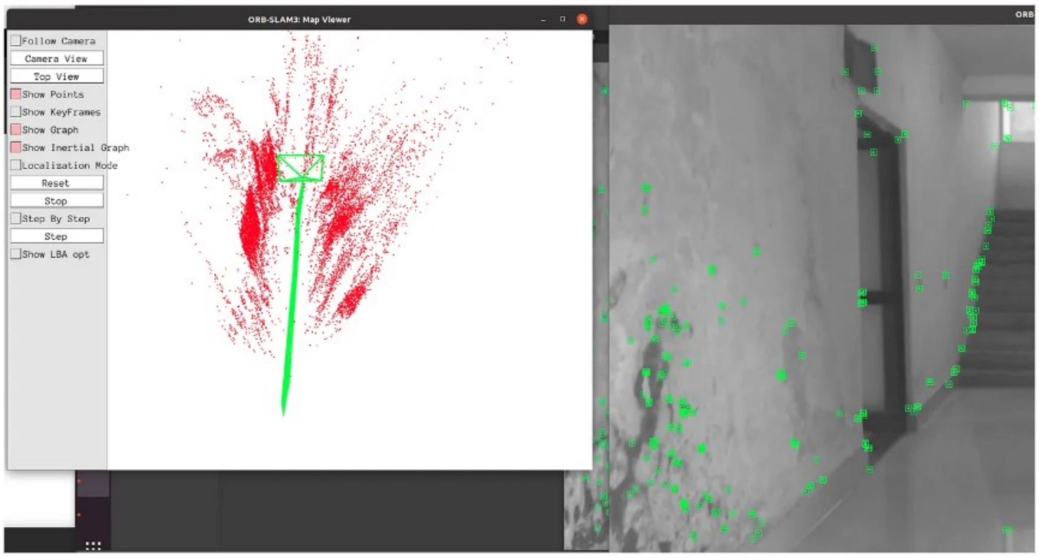


Figure 11: ORB-SLAM3 point cloud creation source (51)

ORB-SLAM3 also makes rotation and translation matrixes. This is handy because these calculated rotations and translation matrixes can be used to calculate the backray from 2.5.

2.5.2 Speed estimation

To get the speed of a vehicle v, the location of a vehicle needs to be known over time.

$$v = d/t \tag{8}$$

Where d is the distance in meters and t is time in seconds. And v is measured in m/s (52).

An example of how the speed can be calculated between frames can be seen in the figure below:

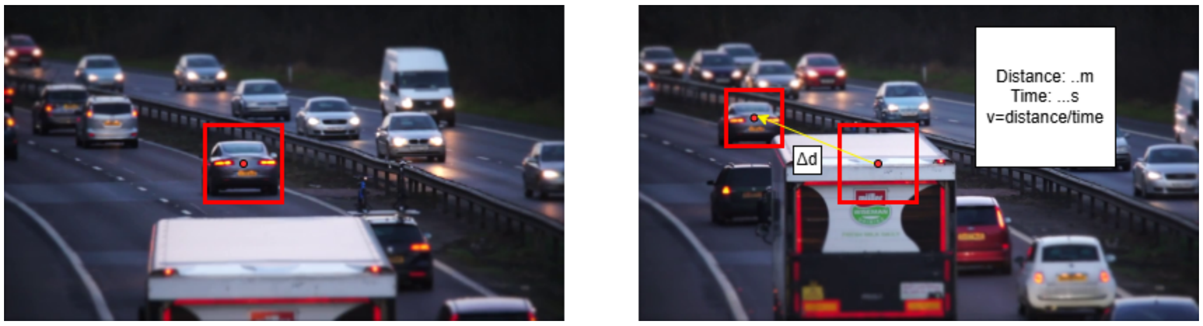


Figure 12: Speed estimation between frames

In Figure 12 two frames are considered: the location of the same vehicle is highlighted with a red dot so the difference in location can be determined. Knowing the time difference between frames, the speed can be calculated with the previously stated formula 8.

The way to measure the speed can depend on the operational condition. For example, it can be used between two close frames, but it can also be used between two frames far apart to get a more averaged-out speed if the vehicle is moving at a constant speed. Depending on the accuracy in measuring positions and time, the latter option can be preferred as it "averages" errors more.

A longer interval could be used to get a better understanding of the speed progression in time. Since vehicles can accelerate and also brake. So, if you take two times far apart, this still does not guarantee a good averaged-out speed.

In addition, if two frames are picked that are close time-wise, the error of the measurement of the vehicle's position may be larger than the measured distance traveled. This gives a large error in the speed estimation, and the time between frames should thus be taken into account.

2.6 Time synchronization

To reliably measure the delay and speed, the times of the drone and server need to be the same. A couple of methods to synchronize these clocks are: (53)

- Global Positioning System(GNSS)
- Network Time Protocol(NTP) (54) (55)
- Precision Time Protocol(PTP)

Some advantages and disadvantages of these methods are that:
For the method with GNSS, there is extra hardware needed, but it is by far the most accurate method as the time from the GNSS can have an accuracy of 100 nanoseconds to 1 microsecond. The implementation of this method is complex as both devices need to have a GNSS receiver, and for both receivers, code needs to be written to extract the GNSS central time.

The NTP method has a precision of tens of milliseconds for public internet, and it is the most common time synchronization mechanism in use today. NTP clients request time from NTP servers and adjusts its clock accordingly. To implement this you need to add the NTP servers you want to use to a configuration file and restarting NTP on the device. To find NTP servers, websites like "https://www.ntppool.org/nl/" provide thousands of NTP servers you can use around the world.

The PTP method's accuracy is in the sub-microsecond range on local networks but is not designed for devices that are not on the same network. It is also known as the more precise but more complex version of NTP but then for devices that are on the same network.

For the NTP method, it is also critical that there is an internet connection. A bad internet connection can increase latency and jitter, reducing time synchronization accuracy.

However, for the GNSS method, you need to keep in mind that the time is dependent on the GNSS being able to receive the time.

So, for both methods, you need to keep in mind that they need to be reachable.

3 Methodology

This thesis is about gathering images and telemetry data (IMU and GNSS data) and sending this data from the drone that collects them to a ground server/workstation. On this server/workstation, this data is then used in real time to run some algorithms to detect and track vehicles so the speed of these vehicles can be estimated based on the remote acquired data.

In this chapter, the methodology behind this approach is explained. This chapter is divided into two parts: drone (section 3.1) and server (section 3.2). For every module, a flowchart is presented with the functions that are being used. For each of these functions, a subsection is given to describe them in detail.

3.1 Drone

In this section, the flow of the drone's module is discussed in detail, as reported in the figure below. Here, it can be seen that the code is started (start in Figure13), an image is taken, and telemetry data is taken. The telemetry data is continuously measured which means it always collects data, and once there are enough telemetry measurements for the current image, the data is packaged and sent to the server. What is meant with enough telemetry data is the set amount of IMU measurements the user wants before and after the image is taken, this can be set to 10 for example then the code will wait until these 10 IMU measurements are captured after the image is taken and then the telemetry data is packaged and sent to the server. Once there are enough IMU measurements the package is thus sent but also the trigger to get a new image (trigger new image in Figure 13) is called to make a new image and send it to the server, after adding the timestamp to it (add timestamp). This loop keeps going until the code is stopped manually or the number of images wanted to be sent is reached. In the following, each of the blocks presented in Figure 13 will be discussed in detail.

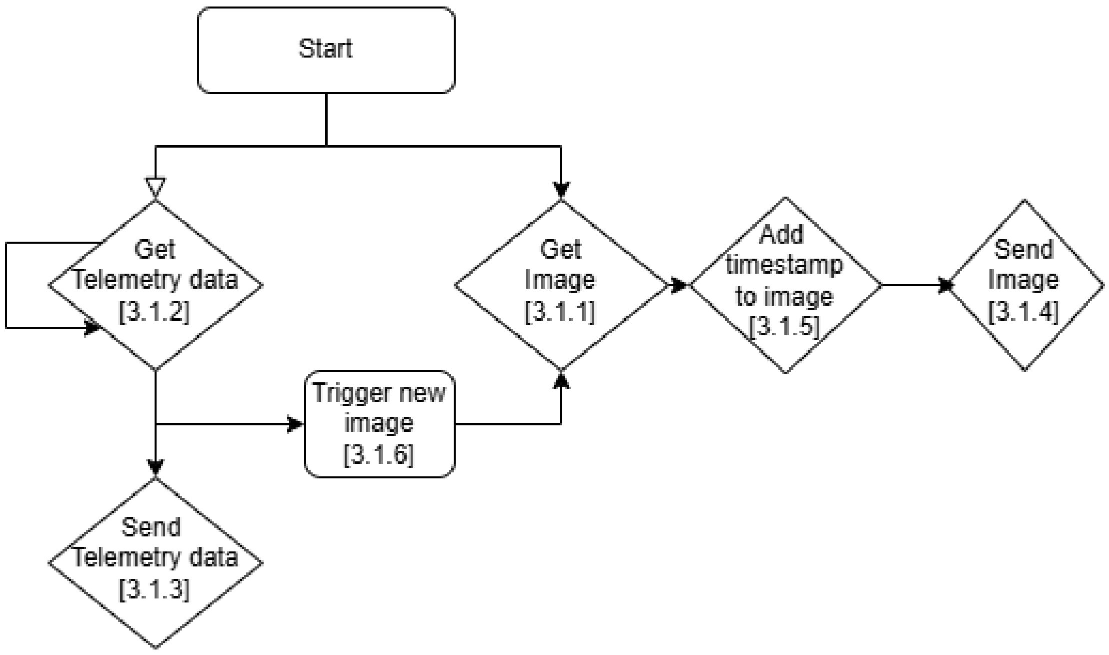

Start
Get Telemetry data [3.1.2]
Get Image [3.1.1]
Add timestamp to image [3.1.5]
Send Image [3.1.4]
Trigger new image [3.1.6]
Send Telemetry data [3.1.3]

Figure 13: Scheme drone

3.1.1 Get image

To get the images, a library called open-cv (56) is used, which can use USB devices such as webcams and cameras to take images and store them. This is exactly what happens in the function of "Get image" it captures an image and saves the time when the image is taken. Then, once an image is taken, the "Add timestamp to image" function is called, which adds the timestamp of the acquisition to the image itself. Then, the "Send image" function is called, which sends the image from the drone to the server.

3.1.2 Get telemetry data

When a drone is flying and collecting images and location information, telemetry data is the information that records data about the location and attitude of the drone. So, the GNSS coordinates and the IMU measurements are stored in the the telemetry data.

In this work, these GNSS and IMU measurements are obtained from a Pixhawk 4 which can be seen in Figure 14 (57).

Figure 14: Pixhawk 4 (57)

Pixhawk 4 is an advanced autopilot which also has on-board sensors such as accelerometers and gyroscope meters (57).

These GNSS and IMU measurements are captured at different sampling rates. In this case the maximum sampling rate of the IMU for the Pixhawk 4 was around 210 per second, and the sampling rate of the GNSS was 5 per second (58). These sampling rates can be modified with the help of some commands from Pixhawk.

The pixhawk data then needs to be sent from the pixhawk to the onboard processing unit (Jetson/computer) with the help of the Mavlink library (59). Mavlink, or Micro Air Vehicle Link, is a very lightweight messaging protocol that is made for communication with drones and drone components (59). It can be used to send data like GNSS, orientation, and IMU data (60).

For this project, the drone module uses the Mavlink library to connect to the pixhawk and listens to messages the pixhawk sends. Specific messages of "SCALED_IMU" and "GPS_RAW_INT" are continuously captured and stored in a deque, which stores the last n amount of measurements (61).

To get the accurate location of the drone, multiple IMU measurements before and after the image need to be taken. The number of IMU measurements captured can be changed in the config file of the code of the drone. These IMU measurements will help determining the location of the drone more accurately with the combination of the GNSS data. Then, after the image is taken you wait until there are enough IMU measurements and this telemetry data is then sent to the server with the "send telemetry data" function, and the "trigger new image" function is called to make a new image.

3.1.3 Send telemetry data

A protocol needs to be chosen to send the telemetry data. A good protocol should have low latency as it is critical for surveillance and many other applications to have it as close to real-time as possible. Reliability is another important element to ensure that all the data that you want to send is actually being sent and fully received on the other side.

In this project, a drone with 4G connection is used to send and receive the telemetry data. However, 4G introduces challenges primarily with overhead and bandwidth usage. Overhead is the extra data a protocol needs to add to the actual data that you want to send, this extra data is to manage communication between devices so there is an accurate data transmission. It includes information such as metadata, error checking, and control messages. So, if there is a high overhead, more data needs to be sent, and this causes the bandwidth to be used more and the time needed to receive the data to increase. Bandwidth efficiency is also important as this minimizes the data usage, which improves the transmission performance. Since the project is based on a drone, the power consumption of the protocol needs to be low too as there is a limited amount of energy in the system. Additionally, the security of the data transfer needs to be kept in mind as if there is no security, the data can be intercepted and maybe even changed along the line, and this will give a false view of reality or even worse give the information to someone who should not have access to set information. So, the important parameters for this protocol are overhead, latency, QoS(reliability), power consumption, security, and bandwidth efficiency.

A Design Space Exploration (DSE) (62) took place to get the best suitable protocol, and the result of this DSE can be seen below in Table 1. These results are based on the results presented in (63), where all these protocols are compared to each other and ranked. These ranks are then filled into this table, and that is what those numbers represent. A score of 5 means good, and a score of 1 means bad. Then, these parameters, are given a weight factor of how important they are. Latency and QoS are of utmost importance, and therefore, they are given the weight of 5. As we want a protocol as close as possible to a real-time telemetry data sending protocol, which also can send every message reliably as the extraction of the location of the UAV is dependent on this information. Besides these two parameters, the overhead and bandwidth efficiency are the most important. This is because these go about how much data needs to be sent and if the bandwidth is used optimally. This is important to have a good running and fast protocol. Then comes security, which is not of utmost importance and therefore given a 3, but it is not the main objective of this protocol. Then, at last, the power consumption. This is because although the setup has limited power, the power needed for such a protocol is lower than keeping the drone in the air and is, therefore, not the most important parameter in the list.

	weight	MQTT	CoAP	AMQP	HTTP	
overhead	4	4	5	3	1	
latency	5	4	5	3	1	
Qos	5	5	2	5	1	
power consumption	2	4	5	3	1	
security	3	3	4	5	3	
bandwidth efficiency	4	5	5	3	2	
		98	97	85	33	Score

Table 1: DSE telemetry data sending protocol

These scores are gotten by the formula: $score = \sum weight \cdot points$ for every parameter. So, an example is for MQTT $score = weight1 \cdot points1 + weight2 \cdot points2 + weight3 \cdot points3 + weight4 \cdot points4 + weight5 \cdot points5 + weight6 \cdot points6$
$score = 4 \cdot 4 + 5 \cdot 4 + 5 \cdot 5 + 2 \cdot 4 + 3 \cdot 3 + 4 \cdot 5 = 98$

MQTT and CoAP are the best-scoring protocols. Because of the preferred way of how MQTT works (publisher/subscriber), this protocol will be implemented for this project. This protocol can handle multiple devices sending data to the broker, and the subscriber can subscribe to all the topics at once or individually, which makes it a good protocol for a system where multiple devices send data to a single server.

There are multiple brokers available and some were discussed in 2.2.5 and the best suitable broker is HiveMQ with the free plan, which can be changed to a paying plan later on. A broker is a middleman which is able to receive and send data so it acts as a publisher and subscriber. HiveMQ is the most suitable broker as with Mosquitto, you need to set up the broker yourself and do maintenance. HiveMQ is also more scalable and has some extra security features, the documentation from HiveMQ is professional and has tutorials, whereas the documentation of Mosquitto is community-based. Also, HiveMQ has higher availability and fault tolerance than the standard Mosquitto broker. So HiveMQ was implemented as the MQTT broker.

This is then implemented with Python with the help of MQTT libraries (64) You can use a Python MQTT client to connect to an MQTT broker, publish messages, and subscribe to topics to receive messages. The Paho Python Client (64) is one of the most trusted and widely used libraries for MQTT communication. This client provides a robust solution for connecting to MQTT servers, publishing messages, and subscribing to topics with ease. Therefore, it is being used in this project.

With Python, this MQTT client can be imported. You can connect to HiveMQ with port 8883, which is the default port for MQTT. To ensure the security of the data transfer, Transport Layer Security (TLS) is used to encrypt data, prevent interception, and prevent tempering. Besides TLS, client identifiers and username and password credentials are used to increase security.

Once a connection is made, it is possible to publish messages and subscribe to topics to receive messages (65).

3.1.4 Send image

The current setup of the University of Twente already offers the possibility to send videos with the protocols RTMP and RTSP. The protocol changes depending on what device you want to stream the data from. As for DJI drones, RTMP is used, and RTSP is used for other drones. With the help of FFmpeg(66), the video from the drone can be sent to the media server of the University of Twente. FFmpeg is open-source software that is capable of decoding, encoding, transcoding, muxing, demuxing, streaming, filtering, and playing files like images and video. Here the streaming option is used to stream the images that are being captured to a MediaMTX server of the UT (66). MediaMTX is a ready-to-use real-time media server that allows a user to publish, read, proxy, record, and playback video and audio streams. It routes media from one to another (67).

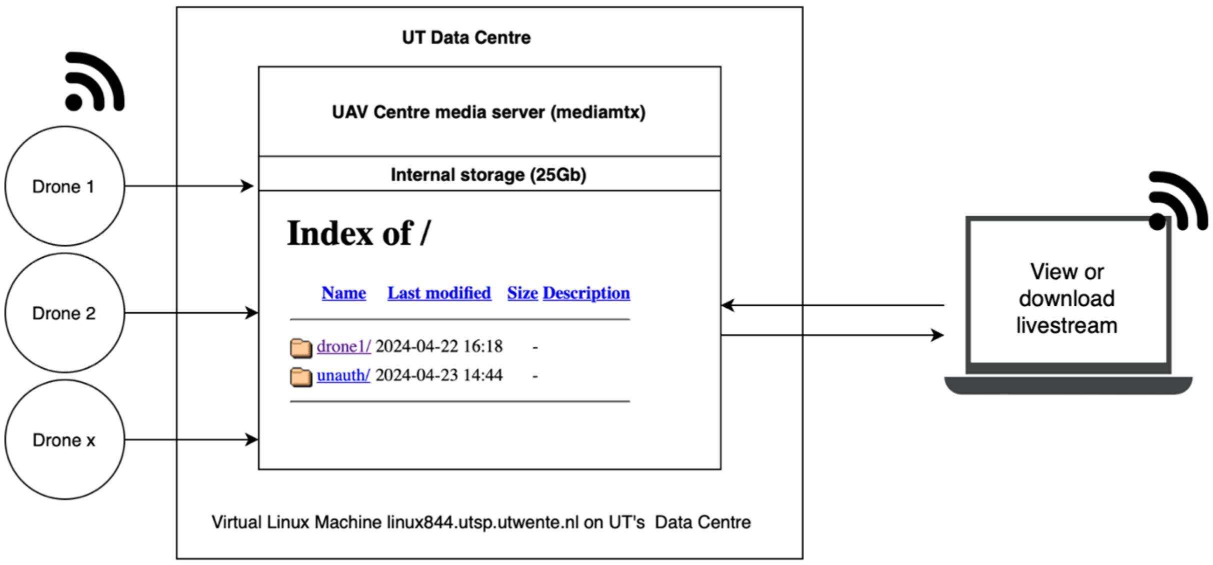

UT Data Centre
UAV Centre media server (mediamtx)
Internal storage (25Gb)
Index of /
Name
Last modified
Size
Description
drone1/ 2024-04-22 16:18 -
unauth/ 2024-04-23 14:44 -
Drone 1
Drone 2
Drone x
View or download livestream
Virtual Linux Machine linux844.utsp.utwente.nl on UT's Data Centre

Figure 15: MediaMTX server

In the figure above Figure 15, a scheme can be seen of how this MediaMTX server works. As it can be seen in this figure, the drones stream data to the server, the server stores the data, and exchanges this data with any device that wants to connect and see the live view of the stream or wants to download the stream. There are two ways to access the streams: 1) view and download the stream "live" on your laptop, or 2) download from the web server for post-processing. The first can be done by opening VLC media player and using the URL of the stream. The second can be done by going to the designated website, which is hosted for showing the streams in this case this website is: `https://uavcentre-datastreaming.itc.nl`. On this website, the designated drones stream can be found, this stream can be viewed but also downloaded. [UAV research/Document/General/UAV Centre media server/Intructions UAV Centre media server.docx]

However, this data streaming needs some adjustments, as the capability to send telemetry data and images to the server needs to be present and the current data streaming implementation only sends images.

The protocols RTMP and RTSP are chosen because it is used in (3), (5), (12), and (8) and are a proven method to send images reliably and fast to another device, It was therefore, assumed that this solution would be good enough to handle the incoming video stream from the drones. This, of course, would need some optimization and some changes as with the current method the transfer time (time needed to fully send and receive the image was around 2 seconds) was high for a real time application.

This sending of images with RTSP was implemented with Python with the help of FFmpeg, which makes it possible to send images from the drone to the server.

3.1.5 Add timestamp to image

Multiple methods were possible to send this timestamp across. In this section the LSB, OCR and MSB methods were all preliminary tested, allowing to draw some conclusions on their applicability.

LSB The first method was LSB, with this method the least significant bits are changed of some pixels so that the timestamp data can be embedded into the image itself. Then, once the image is sent, the timestamp can be extracted, and the same timestamp will be extracted from the telemetry data. However, this method ended up not working properly because of the encoding, decoding, and conversion loss. So, a new method was needed.

OCR For this method, the timestamp was added as text below the original image and OCR was used to extract the timestamp again, this was tested but it could not 100% correctly read out the text and the speed of this method was below par it took around 0.5seconds to extract the time. So again, a new method was needed.

MSB The most significant bit (MSB) method is the method where new pixels are added underneath the full image, and in these new pixels, the R, G, and B channel values are changed to embed the timestamp. The timestamp of the time the image is taken is converted to a H:M:S.f format and then converted to binary, which is extended so it always has a binary length of 52.

These binary values are then encoded in the new pixels. If a binary value of the timestamp is 0, the channel value of the color is set to 0. But if the binary value of the timestamp is 1, the channel value of the color will be set to 255. This is done from the R, G, and B channels. Once these are used, it moves on to the next pixel. This is done until all the binary values of the timestamp are used.

The scheme below shows an example of this process in Figure 43:

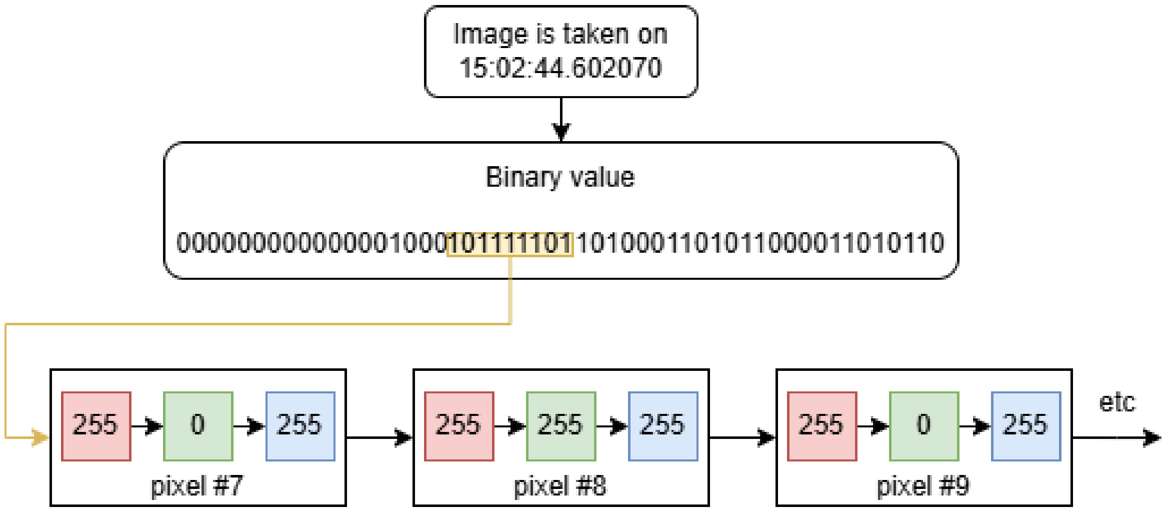


Figure 16: MSB encoding

The yellow values of the binary values of the timestamp are used in this example, and the corresponding R, G, and B channels values are changed accordingly.

Once the timestamp is fully encoded in the pixels, the image can be sent to the server. Once the image arrives on the server, these pixels are then extracted and de-encoded. This means that the R, G, and B channels are read from these pixels, and the timestamp is reconstructed.

MSB was tested, and this method is fast enough as it at it worst takes 0.001 seconds to re-encode the timestamp, and it has no mistakes with reading out the timestamps. So, this method was ultimately used to send and receive the timestamp from the image.

3.1.6 Trigger new image

Once the wanted (this is a setting in the config file) amount of IMU measurements after the image is taken, the "Trigger new image" function is called. This function has the option to save the telemetry data in a JSON file.

JavaScript Object Notation (JSON) is a lightweight format for storing and transporting data. An example of a JSON file can be seen below. Here, there are 3 employees with first names and last names saved.

```
{
"employees":[
"firstName":"John", "lastName":"Doe",
"firstName":"Anna", "lastName":"Smith",
"firstName":"Peter", "lastName":"Jones"
]
}
```

(68) (69)

JSON filenames use the extension .json.

The telemetry data JSON file was organized to store all the needed information. An example of the content of this file is shown below:

{"web_start_cam": 1739356709.8401132, "web_end_cam": 1739356709.840831, "time_boot_ms": [6844825, 6844829], "xacc": [-2, -2], "yacc": [18, -19], "zacc": [-999, -999], "xgyro": [1, -1], "ygyro": [0, -3], "zgyro": [1, 0], "xmag": [127, 127], "ymag": [-127, -127], "zmag": [353, 353], "time_IMU_RTC_CLOCK": [1739356709.920386, 1739356709.9232512], "time_usec": [6844691808, 6844816804], "fix_type": [0, 0], "lat": [0, 0], "lon": [0, 0],"alt": [-17000, -17000], "eph": [9999, 9999], "epv": [9999, 9999], "vel": [0, 0], "cog": [0, 0], "satellites_visible": [0, 0], "time_GPS_RTC_CLOCK": [1739356709.7888496, 1739356709.9116318], "count": 213, "received_MQTT": 1739356710.152383}

To give a better and clearer overview of this data, Table 2 describe all the data stored in each JSON file where different measurements of GNSS and IMU and their attributes as well as all additional information such as a counter and the time the image is taken are stored.

General information	**IMU measurement**	**GPS measurement**
Time image is taken	Acceleration (x,y,z)	Latitude
Counter	Gyroscopes angular velocity (x,y,z)	Longitude
	Magnetic field (x,y,z)	Altitude
	Time the measurement is done	Number of satilites visible
		Time the measurement is done

Table 2: Telemetry data JSON information

For the IMU the x,y, and z acceleration(acc), the angular speed around that axis(gyro), the magnetic field(mag), and the time the measurement is done(time_IMU_RTC_CLOCK) are stored. The GNSS data is the latitude(lat), longitude(lon), altitude(alt), number of satellites visible(satellites_visible), and the time the measurement is done(time_GPS_RTC_CLOCK) (61).

The telemetry data is captured and kept in memory continuously, independently from the images. Once there are enough IMU measurements after the image is taken, the telemetry data is packed in a JSON file and sent to the server, also is the process continued with taking a new image and a new set of telemetry data.

In the tested implementation, 14 IMU measurements are recorded before and after the image is taken. This number stems from using the 3D mapping function where the location of the drone needs to be known, and around 14 IMU measurements before and after seem to be a good number to get fused with position (GNSS) and image information to get a good accuracy for the location.

3.2 Server

In this section, the flow of the server's module and its related code is discussed in detail. The flow of the code can be seen in Figure 17. From this scheme it can be seen when the code is started, the image sent from the drone can be received, and this process is run continuously until no new image comes in from the drone (Receive images function). Also, the telemetry data from the drone is received continuously just like the images (Receive telemetry data in Figure 17).

Once a new image is received, the image with the corresponding telemetry data is linked (Link image and telem function). If these are linked, the algorithm can be run. The algorithm starts by making two new threads, with the first sends data to the algorithms for 3D map generation (make 3D map in Figure 17) and the second thread detects and tracks the objects seen in the images(Object detection and tracking function).

Once the 3D map is generated, the point cloud (receive 3D map function), and the rotation and translation matrixes will be received (receive R, T matrixes function) that are made in the code which makes the 3D map. As a matter of fact these rotation and translation matrixes will be needed to make the trace from 2D to 3D.

After object detection and tracking, the pixel location of the vehicles is known. The rotation and translation matrixes describing the position of the camera are then combined with the pixel location to convert the 2D coordinates to a 3D ray (trace 2D to 3D function). With this 3D ray, the 3D location of the object can be determined(Get 3D point object function) by intersecting the the generated 3D map (i.e. point cloud). This 3D location of the objects in multiple frames can then be used to estimate the speed of the vehicles(Estimate the speed function). This loop runs continuously until no new images are received or it is stopped manually.

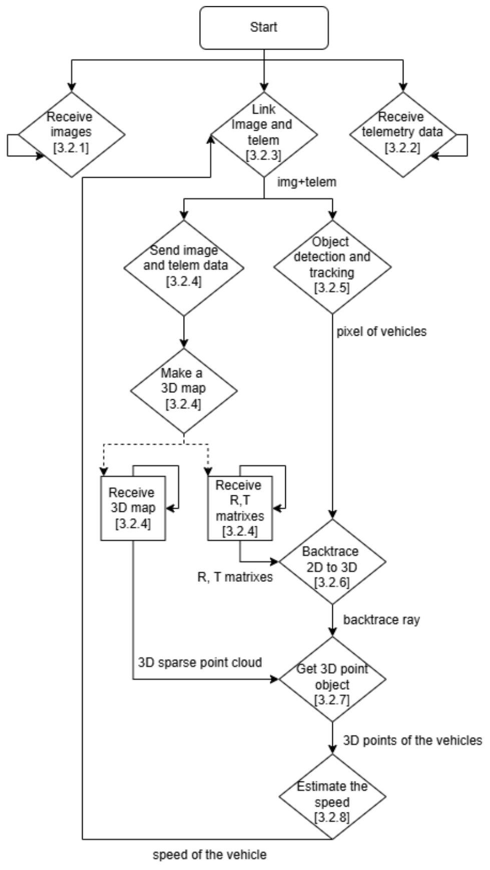

Start
Receive images [3.2.1]
Link Image and telem [3.2.3]
Receive telemetry data [3.2.2]
img+telem
Send image and telem data [3.2.4]
Object detection and tracking [3.2.5]
pixel of vehicles
Make a 3D map [3.2.4]
Receive 3D map [3.2.4]
Receive R,T matrixes [3.2.4]
R, T matrixes
Backtrace 2D to 3D [3.2.6]
backtrace ray
3D sparse point cloud
Get 3D point object [3.2.7]
3D points of the vehicles
Estimate the speed [3.2.8]
speed of the vehicle

Figure 17: Scheme of the server code

3.2.1 Receive images

To receive the images on the server, multiple methods could be used. And multiple are tried. The methods that were tried are open-cv and FFmpeg to receive the images. Open-cv and FFmpeg use the RTSP stream URL to listen in on the stream and receive the images that way.

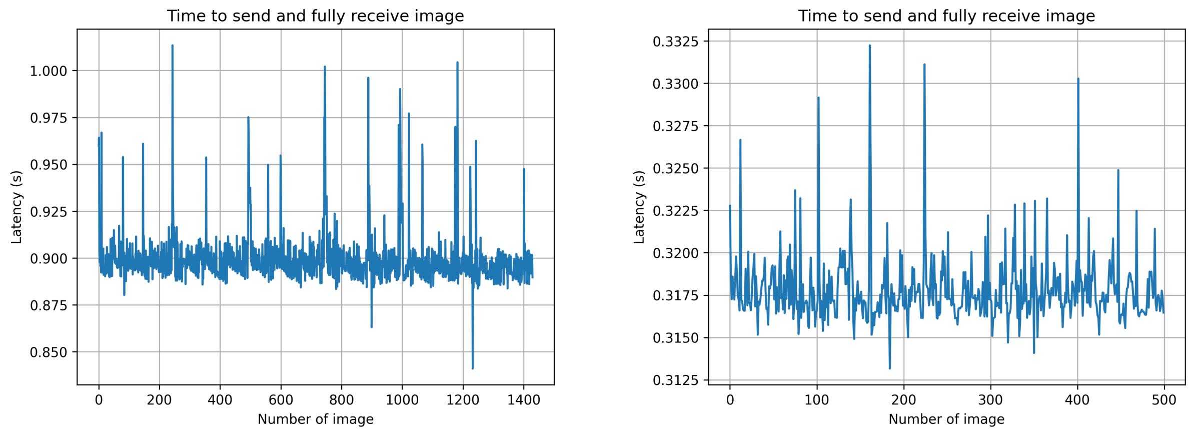
These two methods differ in implementation methods and results, as open-cv uses the cv2.VideoCapture(rtsp_url) function in python while in FFmpeg, a process needs to be run for a command like the one below:

```
FFMPEG_CMD = [
"ffmpeg",
"-rtsp_transport", "tcp",
"-fflags", "nobuffer",
"-flags", "low_delay",
"-max_delay", "0",
"-probesize", "32",
"-analyzeduration", "0",
"-i", RTSP_URL,
"-f", "rawvideo",
"-pix_fmt", "bgr24",
"-vf", "fps=25",
"-vcodec", "rawvideo",
"-an", "-",
```

] This command states which transport protocol is used how many buffers are assigned and to what RTSP stream the data is received from alongside with the datatype and frame rate.

Both methods receive the images, and as soon as it is received, the time is also saved this is done with a function in python which can get the current time by "time.time()" so the time can be measured between sending and receiving the images. As a preliminary test, both OpenCV and FFmpeg were tested, this test exists of sending data from a laptop to the same laptop. A WIFI network was used to send the images through the UT mediamtx server and the images were received on the same laptop connected to the UT network (i.e. Eduroam). The results of this test can be seen in Figure 18. As can be seen in this figure, the time needed with the open-cv method is significantly bigger than that of the FFmpeg method.

In particular, the time needed for open-cv is around 1 second and the time for FFmpeg is around 0.33 seconds: this suggested to implement the FFmpeg solution in the deployment of the algorithm.



(a) open-cv **(b)** FFmpeg

Figure 18: Open-cv and FFmpeg delay results for 10fps

3.2.2 Receive telemetry data

To receive the telemetry data sent with MQTT from the drone, the libraries of MQTT are used again to receive the data.

With these libraries, you can subscribe to topics and receive the data. To do this, you first need to be subscribed to the topic before anything is published to it, otherwise, you might miss the incoming data. This is done in a loop to continuously capture the telemetry data. Once the data is received, the data is stored in a JSON file, which is named after the timestamp that is from the time the image is taken. This is done to later on link the images with the corresponding telemetry data.

3.2.3 Link image and telem

To link the images and telemetry files, multiple methods were possible, and these methods are discussed in 3.1.5. It was decided that the most significant bit (MSB) method would be used. This is because, as stated in 3.1.5, this is the best possible method to send the timestamp as this is fast enough and has no mistakes in reading out the timestamp after sending. This is the method where new pixels are added underneath the full image, and in these new pixels, the R, G, and B channel values are changed to embed the timestamp. Once the timestamp is fully encoded in the pixels, the image is sent to the server. Once the image arrives on the server, these pixels are then extracted and de-encoded.

An example can be seen in the figure below in Figure 19:

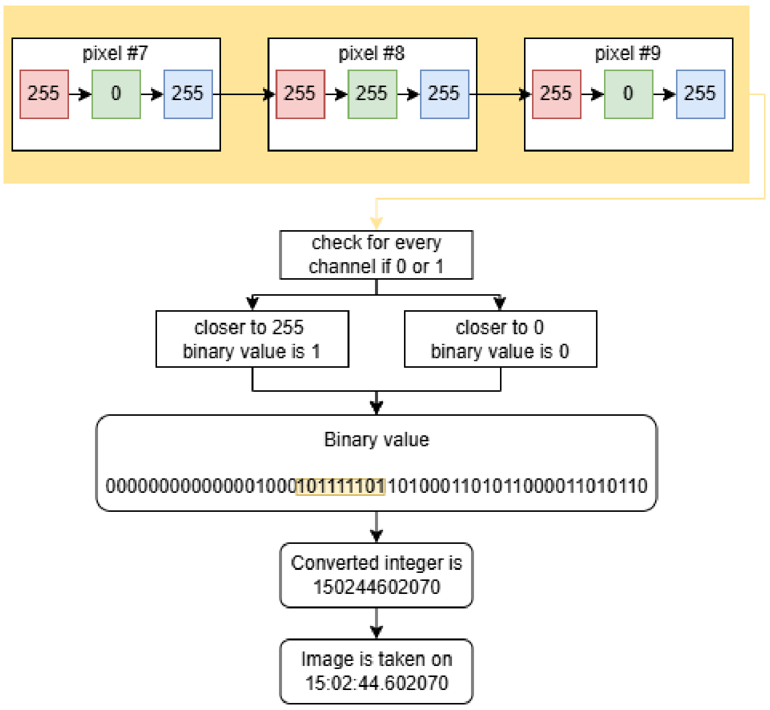

pixel #7
255
0
255
pixel #8
255
255
255
pixel #9
255
0
255
check for every channel if 0 or 1
closer to 255 binary value is 1
closer to 0 binary value is 0
Binary value
0000000000000010001011111011010001101011000011010110
Converted integer is 150244602070
Image is taken on 15:02:44.602070

Figure 19: MSB de-encoding

The values of the R, G, and B channels are used and assessed if they are closer to 0 or 255. Dependent on this, the binary value is expanded with a 0 or 1. 0 for closer to 0 and 1 for closer to 255. This is done 52 times until the full timestamp is extracted. For the example in Figure 19 this number will be 150244602070 which corresponds to time 15:02:44.602070.

Once this timestamp is fully extracted, it can be compared to the timestamp provided in the filename of the saved MQTT message as the MQTT message also has the time the image is taken saved inside of it and is used as the filename. If they are equal, you have successfully linked the image with the corresponding MQTT message.

3.2.4 Make a 3D map

OrbSLAM3 was used to generate a 3D map in real time and retrieve the object's (i.e. car) position. This decision was made as Standardized object scaling was not reliable and only works in environments where objects' sizes are known. Single image depth estimation was also not the optimal choice as this only gives relative depth. Structure from Motion (SfM) was a good contender, but it works in post-processing while ORB-SLAM3 is a SLAM (real-time) solution. In addition, ORB-SLAM3 can estimate the absolute scale using cameras and IMU data, which makes it the preferred option. There exist multiple solutions outside ORB-SLAM3, but for this thesis, this was chosen as it is a reliable, well-known solution.

To make this work with the current code, ROS needed to be implemented in Python as ORB-SLAM3 makes use of ROS for incoming and outputting data. ROS publishers needed to be made to send the data to the code and subscribers where needed to receive the output data (see figure 20).

The input data are the images, IMU, and GNSS data, and the output data is the rotation matrix, translation vector, and the 3D point cloud for every new image added to the block.

In the image below in Figure 20, the flow of the ROS subscribers and publishers is shown. The publishers send data to ORB-SLAM3, and the subscribers get the output data from ORB-SLAM3.

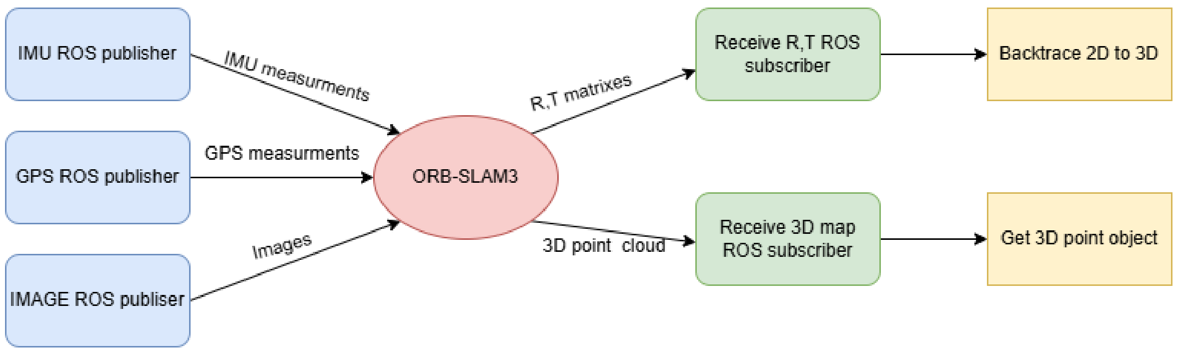


Figure 20: ORB-SLAM3 ros topics

To work with ROS, it is encouraged to work with Linux, and for ORB-SLAM3 ROS noetic is used, ROs noetic is a version of ROS (70)So, a system with Ubuntu 20.04 is needed, not 22.04 or any further versions, as these versions do not support ROS noetic. Once this is all setup and ROS noetic is installed, it is then possible to use ROS in Python with the help of a library called rospy (71).

3.2.5 Object detection and tracking

In the state of the art, a couple of object detection methods, Fast RCNN, Faster RCNN, and YOLO, are discussed. All the YOLO versions are discussed with what is different between them. Based on this information, it was decided to use YOLO as this is the most widely used algorithm for object detection and because it achieved better performance. The performance metrics of the different models can be seen in the figure below in Figure 21:

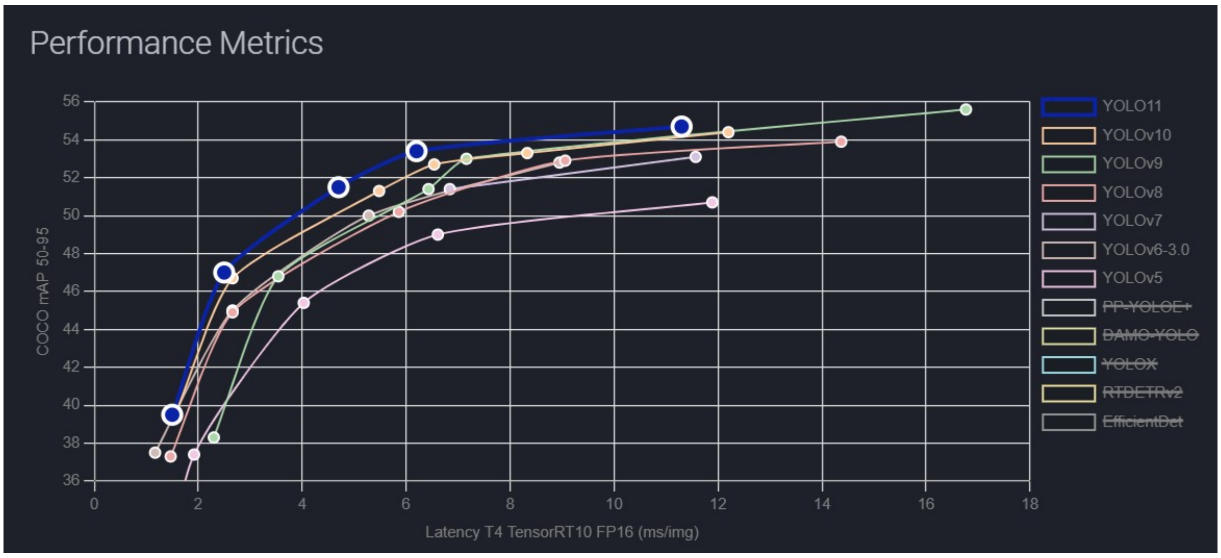


Figure 21: Performance YOLO versions source (33)

Based on the performance metrics shown in Figure 21, the YOLOV11 model was chosen as this is the newest model with computational efficiency and good accuracy. It also supports multiple-sized models and object detection, instance segmentation, image classification, pose estimation, and oriented object detection (OBB).

And to track the object over time the Bot-SORT, ByteTrack and deep learning trackers were reviewed in the state of the art. Bot-SORT and ByteTrack trackers are made for multi-object tracking, while the SiamFC tracker is more suitable for single-image object tracking, which makes the SiamFC tracker less suitable for this implementation as there will be multiple cars/vehicles that need to be tracked.

Bot-SORT takes more time to run than ByteTrack but has increased accuracy, so the decision on which tracking to use depends on whether the speed of the tracker is more important or, on the opposite, the accuracy is more important. Here, the decision was to count more for the accuracy and the Bot-SORT tracker was ultimately chosen.

YOLO and the tracker are implemented using Ultralytics (72) , which makes it easy to implement. This implementation can be seen below:

```
results = model.track(image, persist=True, tracker=botsort.yaml,
    save=True, device=cuda)
```

The output of this object detection and tracking are bounding boxes with IDs assigned to them so the same vehicle/object keeps the same ID, and they can be tracked in time. An example can be seen in the figure below in Figure 22:

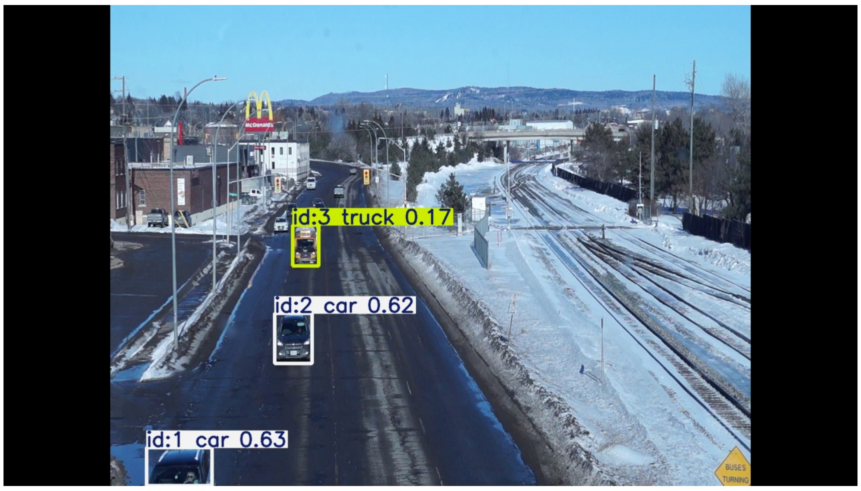


Figure 22: Object detection and tracking

3.2.6 Backtrace 2D to 3D

After using the detection and tracking algorithms, the pixel location of the object is known. To then transform this 2D pixel to a 3D location, a projection ray is needed.

In the section 2.5, it is explained how this back projection can be made using the pixel coordinate, K (intrinsic parameter matrix), R (rotation matrix), and T (translation vector) by using the following formula:

$$ray = T + s \cdot (R^T \cdot (K^{-1} \cdot x)) \tag{9}$$

Which is implemented with the help of Python.

An example of such a backtrace projection ray can be seen in the Figure 23 the original 3D point and the camera position can be seen while, using the equation (8), the projected ray is traced, intersecting with the original 3D point.

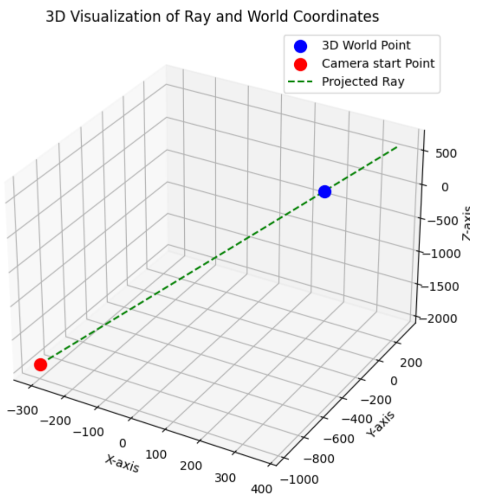


Figure 23: Backtrace 2D to 3D

3.2.7 Get 3D point object

Once this projected ray is known. The next step is to estimate the 3D coordinates of the vehicle. To get the estimated 3D coordinate of the vehicle, the distance between the 3D points of the 3D point cloud and the ray needs to be calculated.

The distance from points to a line can be calculated with the following formulas (73):

$$\mathbf{distances} = \frac{|\mathbf{AP} \times \mathbf{ray_{world}}|}{|\mathbf{ray_{world}}|} \tag{10}$$

$$\mathbf{AP} = \mathbf{P} - \mathbf{A} \tag{11}$$

Where A is the camera position and P are the points from the point cloud.

The formula (10) calculates the distances from points to the ray in 3D space.

The 3D location of the vehicle is computed in two ways, and they are controlled with the parameter closest. If closest is True, the closest point to the ray is used as the vehicle's position. But if closest is False, the deepest point that complies with a defined threshold (value xxx) is used as the vehicle's position. An example of this can be seen in the figure below in Figure 24.

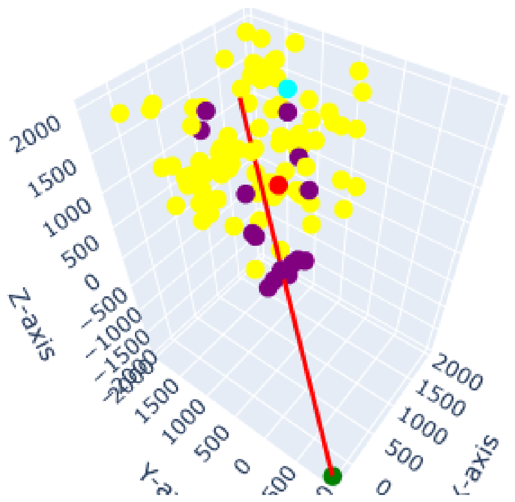


Figure 24: Closest point vs deepest threshold point

Given a projecting ray (red line) can be seen, the red dot is the closest point to that line. On the other hand, the purple points are the points that comply with a set threshold, which checks how far the points are from the line. The aqua-colored point is the deepest point that also complies with the threshold.

As can be seen, there is a significant difference between the closest point and the deepest threshold point. So, it is important to think about which mode to use. If the point cloud detects objects close to this line but not at the location of the vehicle you want to track, this can cause issues, as you then get the wrong location. So choose this mode carefully.

This position is saved and will later be used to estimate the speed of the vehicle. This is done for every vehicle detected.

3.2.8 Estimate the speed

To get the speed of a vehicle, the location of a vehicle needs to be known over time.

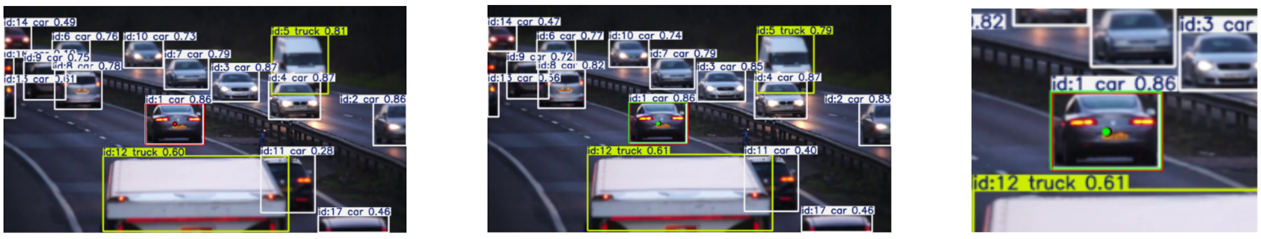
Every ID(vehicle) is tracked along the image sequence: for each frame, the location of the object is defined and the distance is then calculated between the previous known location and the current location. Also, the time difference between this old location and the new location can be calculated as the amount of images and the framerate are known. Once the distance and time difference are known, the speed can be calculated. This can be done using the formula:

$$v = d/t \tag{12}$$

Where the speed is calculated by dividing the distance traveled in meters by the time in seconds. This speed is then saved, and the parameter that keeps track of which image the vehicle is seen in is updated so the code knows the last time the vehicle has been seen.

This speed is only taken between two frames at the moment, but this can, of course, be changed by averaging the values over several frames, reducing the possible errors.

An example of getting the speed between two upcoming frames can be seen in Figure 25. In this figure, the red box is of the previous image, and the green box is of the current image. Here, it can be seen that the bounding boxes of the same vehicle between two images are really close. Thus, a minimal amount of displacement is measured, and this displacement measured can be smaller than the error from the position estimation, which can cause big errors in the speed estimation.



(a) image 1 **(b)** image 2 **(c)** Close up of image 2

Figure 25: Speed estimation between 2 upcoming frames

Another example can be seen in the Figure 26, where two images are used that are a bit further apart. In this figure, the green and red box can be seen again and the red box is the location of the vehicle from the previous image, and the green box is the location of the vehicle in the current image. With images further apart, these locations are not as close anymore, which makes the displacement measured no longer smaller than the position error. The displacement can be calculated as the change in location from the red square to the green square, and the time between these locations is also known since the FPS is known. And then, with the formula from before, $v = d/t$ the speed can be calculated.

(a) image 1 **(b)** image 53

Figure 26: Speed estimation over a longer period of time

3.3 Time synchronization

Based on the information from 2.6, the decision was made to implement the NTP, and if later on it shows that because of an unreliable internet connection the time could not accurately be gotten, then the GNSS method with two GNSS receivers can be implemented to synchronize the time.

The decision for NTP was made because it does not require additional hardware and also because the UT has an NTP server.

4 Tests

During this thesis, some experiments need to be done to measure some metrics on how good the current solution is. The experiments mentioned in this chapter will measure the performance in speed, transfer time, and CPU overhead to benchmark the performance of the solution. How to measure the performance in speed, transfer time, and CPU overhead is also explained in this chapter.

4.1 Performance metrics measurement methods

The parameters transfer time, speed, and CPU overhead need to be measured, and in this section, the explanation is given on how these parameters are measured with the needed steps.

4.1.1 Transfer time

To measure the transfer time, a file needs to be sent from the drone to the server, and the time between sending and receiving the data needs to be captured. To measure this, the timestamp of the sending of the file needs to be saved/sent to the server, and the time of arrival needs to be saved. These times need to be linked with the previously discussed use of a Network Time Protocol(NTP). Then, the time difference can be calculated.

So the steps to measure the transfer time are:

1. Send a file from the drone:
   In this thesis, telemetry data and images are being sent from the drone to the MQTT broker and the MediaMTX server.
2. Record the timestamp when the file is sent from the drone:
   The times when the images and telemetry data are being sent are saved on the drone with the help of a list that is expanded for every file that is being sent, and once the stream is done, this list is saved in a JSON file.
3. Record the timestamp when the full data arrives at the server:
   Once the telemetry data or the full image arrives on the server/workstation, the time is saved just in the same way as it is saved on the drone. It is added to a list, and this list is saved at the end once no data is coming in anymore.
4. Calculate the time difference:
   Now that the time when the images and telemetry data are sent and received, the transfer time can be calculated.

4.1.2 Speed

To measure the speed of data transfer, some files (of different sizes) need to be sent from the drone to the server, and the time between sending and fully receiving the data needs to be captured. Using the following formula, the speed can be calculated $Speed[Mb/s] = \frac{Filesize[Mb]}{TransferTime[s]}$.

So the steps to measure the speed are:

1. Send a file from the drone:
   In this thesis, telemetry data and images are being sent from the drone to the MQTT broker and the MediaMTX server.
2. Record the timestamp when the file is sent from the drone:
   The times when the images and telemetry data are being sent are saved on the drone with the help of a list that is expanded for every file that is being sent, and once the stream is done, this list is saved in a JSON file.
3. Record the timestamp when the full file arrives at the server:
   Then, once the telemetry data or the full image arrives on the server/workstation, the time is saved

just like the time is saved on the drone, it is added to a list, and this list is saved at the end once no data are coming in anymore.

4. Calculate the speed with the above formula:
   Then the time of sending and receiving is known, and also the size of the data is known. Then, with the formula $Speed[Mb/s] = \frac{Filesize[Mb]}{TransferTime[s]}$, the speed can be calculated.

4.1.3 CPU overhead

The CPU overhead can be measured by the difference in CPU usage during the baseline and baseline+task. So the steps to measure the overhead would be:

1. Capture the baseline:
   Run the baseline and capture how much CPU is used during this.
2. Capture the baseline+task:
   Run the baseline with the task and capture how much CPU is used during this.
3. Calculate overhead:
   Then, the CPU overhead is the difference in CPU usage between the baseline and baseline+task.

4.2 Ground test

The ground test is a test that takes place on the ground in an office where the drone has access to a 4G network and the workstation has access to the Eduroam network. During this test the transfer time and thus, also the speed is tested, but that is not all. The CPU overhead is also measured for the workstation. This CPU overhead is measured with regard to a baseline to see how intensive the method of sending and receiving data really is. This test is done to show the capabilities of the solution concerning data transfer on the ground to show a sort of baseline, as 4G and 5G are optimized for ground level and not for the sky. This test should then give perspective to the other test. This test is done for code where only data is sent and received, but it is also done for a version of code where YOLO is incorporated, and also the test is done for the complete version where the 3D map is created and YOLO is run. These different versions are used to see where, if any, bottlenecks come from.

4.3 Flight tests

A flight test is organized to see if the full implementation of the code would work with the drone in the air. This test is done by running the drone and server code when the drone is in the air so the drone can transmit the data it is gathering and the server can receive the data and use it for the algorithms to estimate the speed of the vehicles. During this test, the times needed for the sending and receiving of images, the time YOLO takes, and the time the remainder of the code takes are saved so it can later be analyzed to see where the bottlenecks of the system are. So transfer time and speed can be measured with this test. Also, during this test, another experiment is done where the drone is flown to a high altitude to see if 4G/5G can still send the data at those altitudes. This test will be at 120 meters high.

4.4 Amsterdam tests

There were also tests done in Amsterdam with Dutch Drone Delta (DDD). These test came from the interest of multiple companies that wanted to gather information about human perception, 4G/5G usage and coverage in the air, and the 4G/5G capabilities of sending data with a drone in the air. For this thesis, the focus was on the 4G/5G capabilities of sending data with a drone in the air. These tests were set up on the marineterein in Amsterdam where access was granted to fly around a building where multiple flights were done with different heights.

During the test, the drone was flying around a building on the marineterein, and the drone's code was sending the telemetry and image data. The server/workstation was receiving this data. During this test, the times when the data is sent and received are also saved. This was done to determine the transfer time (time needed to send and fully receive the data).

5 Results and discussion

In this chapter, the results from the tests in Chapter4 are shown and discussed. The results of the ground test being transfer time, speed, and CPU overhead, the results of the Flight test being transfer time and the assessment of the bottlenecks of the implementation, and the results of the Amsterdam test being transfer time.

5.1 Ground test

The ground test is performed with a workstation and a drone. The workstation is an OptiPlex7050 with an intel i7-7700 CPU @3.6GHzx8 and a Mesa Intel HD Graphics 630 (KBL GT2) and the drones onboard computer is a Jetson Xavier NX.

The results of the ground test can be found in this subsection, where the CPU overhead, transfer time, and speed are shown.

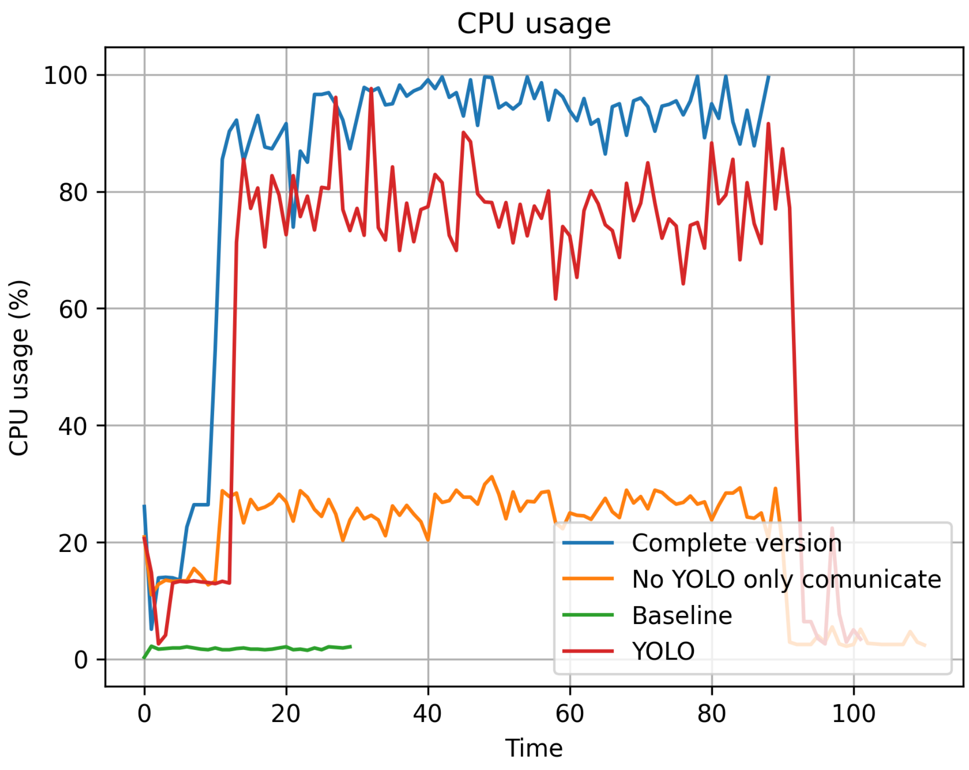


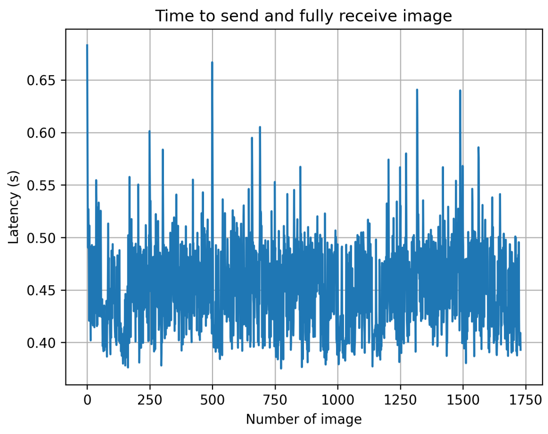
Figure 27: CPU usage ground test

In Table 3 the CPU usage and CPU overhead can be seen CPU overhead is the difference between the CPU usage from a baseline and when the device is performing the task you want to measure:

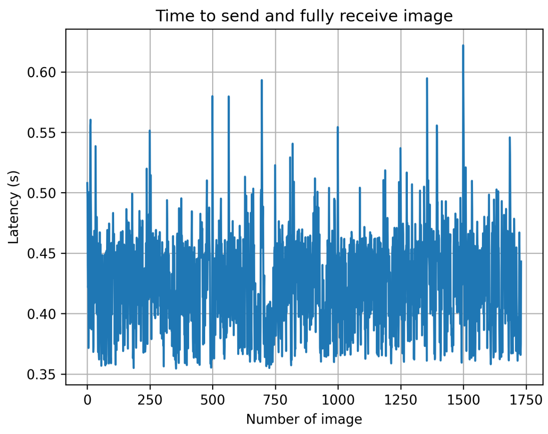
	CPU usage %	**CPU overhead**
Complete version	84.8%	84.8% 1.8% = 83.0%
No YOLO only communication	20.7%	20.7% - 1.8% = 18.9%
Baseline	1.8%	-
YOLO	62.4%	62.4% 1.8% = 60.6%

Table 3: CPU usage and overhead

As can be seen from the results, the CPU usage increases significantly once YOLO is implemented on the workstation. This is caused by the fact that the workstation can not make use of CUDA, so the CPU is used for YOLO. Also, it can be seen that with the complete version, the CPU usage is around 100%, and it was visible during this test that the time needed for YOLO increased significantly. As with only YOLO, it took around 40 ms and now sometimes 1000 ms or even more per image this is caused by the high CPU usage and the fight for resources. This is a problem, and this could be fixed by switching to a workstation that can make use of CUDA.



(a) Transfer time communication

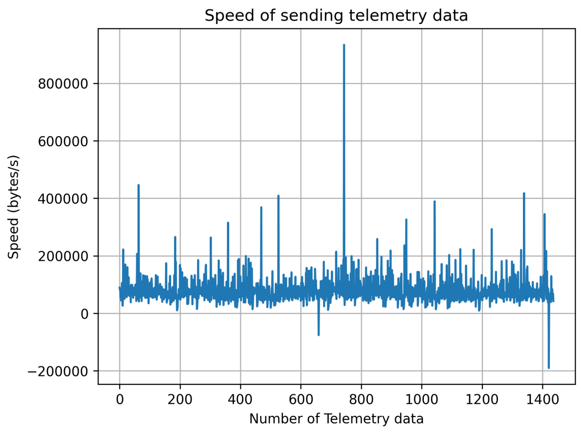


(b) Transfer time with YOLO

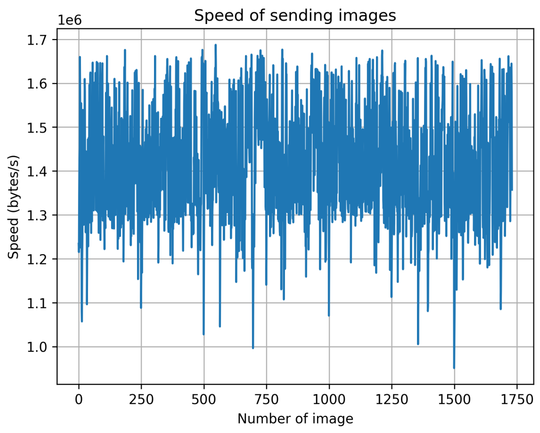
Figure 28: Transfer time images Ground tests

These latencies are obtained by subtracting the receive time by the time you start sending the data, and the results can be seen in Figure 28 The average transfer time for the images is $0.4516sec$ for only communication and $0.425sec$ if also YOLO is added. So, roughly, the transfer time is $0.45sec$ for every image. This result is good enough for the application of mapping, where a transfer time of 1 second is roughly the maximum, but it is not fast enough to pilot the drone based on the incoming images.

Then, the speed was calculated with these transfer time measurements. This is possible since the file sizes are known as they are being stored on the workstation. During this test images of size 724x1280 are being transferred with 10 FPS. The results of these can be seen in the figures below in Figure 29 and Figure 30:

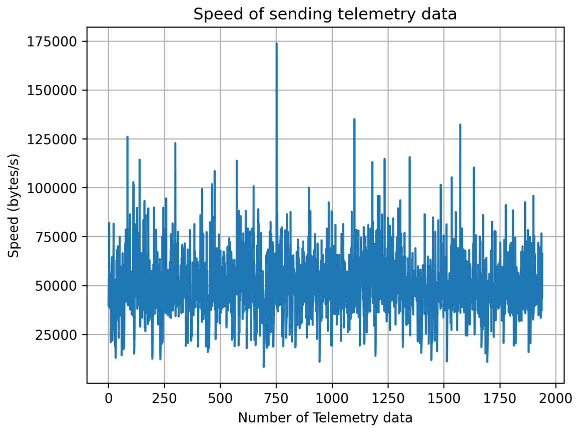


(a) Telemetry

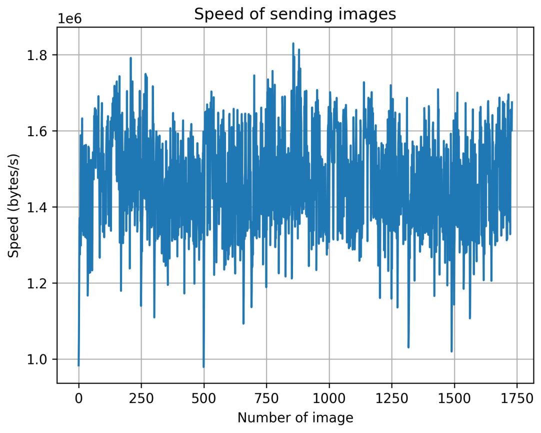


(b) Image

Figure 29: Speed with YOLO and communication



(a) Telemetry



(b) Image

Figure 30: Speed no YOLO only communication

Then the speed was calculated this was done by getting the file sizes and the transfer times. For the YOLO and communication, the average speed was $78555.2 bytes/s = 0.0785 MB/s$ for the telemetry, and the average speed was $1414072.64 bytes/s = 1.41 MB/s$ for the images.

For the no YOLO only communication, the average speed was $49601.78 bytes/s = 0.049 MB/s$ for the telemetry, and the average speed was $1466370.62 bytes/s = 1.47 MB/s$ for the images. These speeds are lower than I expected, but it still suffices for the file sizes we want to send.

In additional tests, other resolutions were tested such as 1920x1080 or 4K, but these resulted in the stream to drop below 10FPS. For 1920x1080, the stream could only handle roughly 5FPS with 4G, and the 4K stream could only handle 1FPS. Because of these drops in FPS, the decision was made to go on with 720x1080 as this could stream at 10FPS, what was the framerate that was given as a requirement.

5.2 Flight tests

During the flight test, two experiments took place. The first experiment is to fly at a high altitude of 120 meters to see if the 4G/5G connection can still send the data. During this test, images of a size of 724x1080 are being sent at 10FPS, and the transfer time result of this experiment can be seen in the figure below in Figure 31:

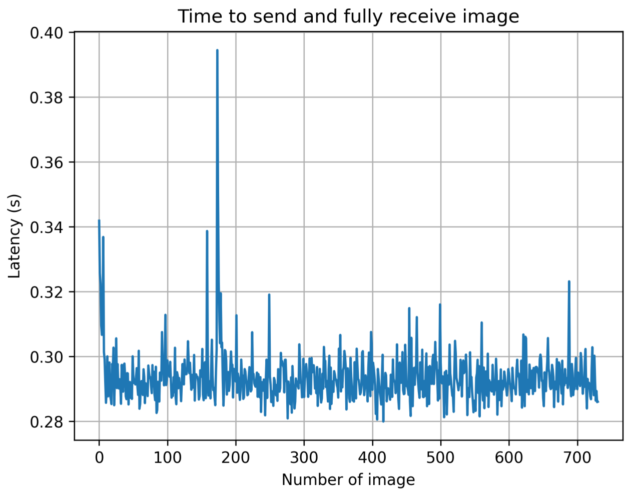
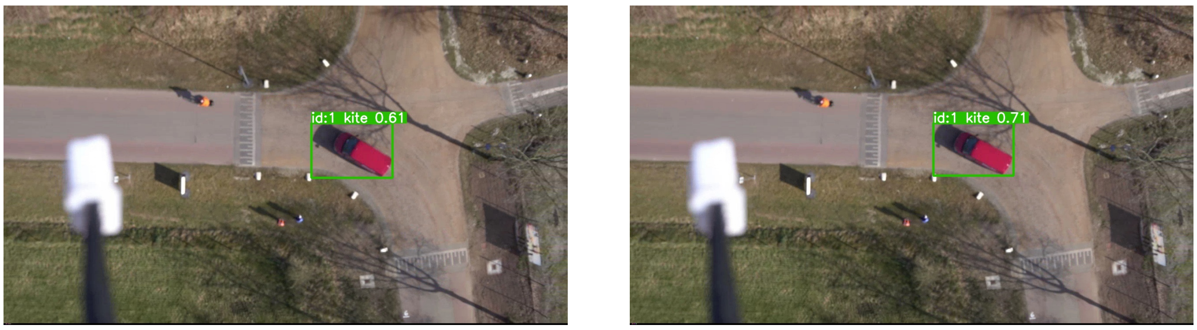


Figure 31: Transfer time at 120 meters height

This test was successful, as the drone could still send the data from that altitude. It was even faster (averaged 0.3s) than the ground test. There could be multiple reasons for this, such as network usage and cell coverage in the place where the ground test was performed. As this was done in a room with a lot of concrete walls, the 4G/5G connection might not be ideal. So, it can be said that it is possible to send images and telemetry data from 120 meters high.

The other experiment is running the full implementation to see if everything is operational and to discover the bottlenecks. During the test, the speed of vehicles is estimated with the algorithms. The objects are detected using YOLO, and a result of this can be seen in the images below in Figure 32:



(a) Image 1666 **(b)** Image 1667

Figure 32: YOLO object detection

As can be seen in the images, the car is not correctly being detected, this is because of the camera's view. The images are now taken from a top-down point of view, which cause YOLO to have difficulties with detecting objects as it is trained on images that are taken with a slight tilt or are taken face-on.

Non the less, a car is detected but with a wrong class assigned to it this car's detection is used in the algorithm to estimate its speed. And the pixel coordinates of the car in image 1666 is [790.62, 328.36], and in image 1667 the pixel coordinate is [780.70, 325.82].

Then the 3D map was created but there were some issues with IMU measurements so the 3D map that could be made is only based on the images and not on the GNSS and IMU data which causes the map to not be scaled. And this 3D map that was made using ORB-SLAM3 can be seen in Figure 33 in this figure it can be seen that the intersection is made into a 3D point cloud.

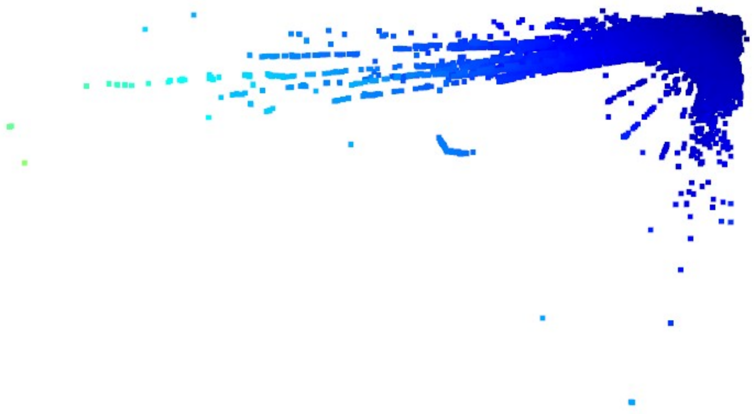


Figure 33: 3D point cloud of Flight test

Using this 3D map in combination with the rotation and translation matrix's the 2D pixel coordinates of the object detection and tracking can be converted to 3D points, and the rays with the 3D estimated point in the 3D point cloud can be seen in Figure 34. In this figure the green lines are the rays, the magenta points are from the point cloud and the red and blue dots are the estimated location of the vehicle.

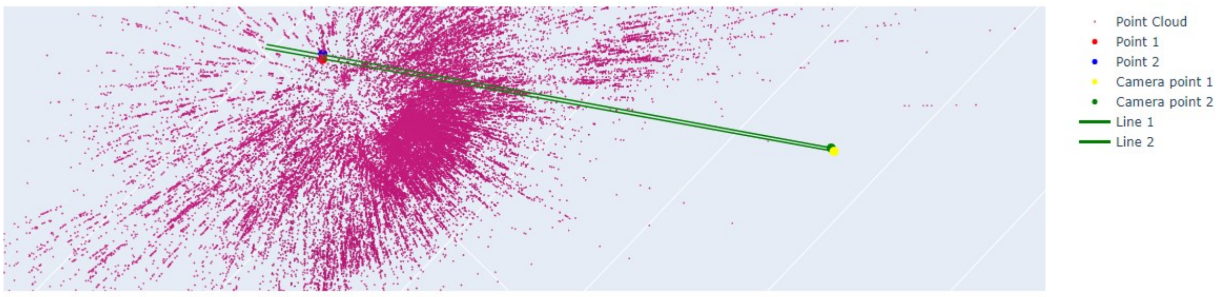


Figure 34: 3D point with pixel rays

For image 1666 the estimated 3D coordinate of the car is [-1.013e-03, 3.13e+00, 3.693e01] and for image 1667 the estimated 3D coordinate is [0.13, 3.26, 0.28] these points are not scaled as previously mentioned the 3D map is only made up from the images and without the GNSS and IMU data.

Then these 3D points are used to estimate the speed of the vehicle and this is done using the formula $v = d/t$ and since the distance can be calculated and the time between the images is known as the stream is 10FPS and the consecutive images are used the time between the images is 0.1 seconds, the speed can be calculated. The speed is 2.07 and this result is in a unit from a relative reference system as this is now done without GNSS coordinates and done in the image frame.

This needed to be done in post-processing because the workstation was not able to handle all the processes in real-time.

So, it is possible to estimate the speed with the designed algorithm but it could not be done on the current workstation and also does YOLO have problems with face-down taken images.

5.3 Amsterdam tests

The test results from the Amsterdam test are shown in this subsection. These tests are performed with a workstation and a drone. The workstation is an OptiPlex7050 with an intel i7-7700 CPU @3.6GHzx8 and a Mesa Intel HD Graphics 630 (KBL GT2), and the drones onboard computer is a Jetson Xavier NX.

Rooftop:
The first test was a test roughly 10 meters above the building where we flew around the building and the transfer time of this test can be seen in Figure 35 from a stream of 724x1080 pixels with 10FPS. These results are as expected as we also see similar results from the "Flight tests".

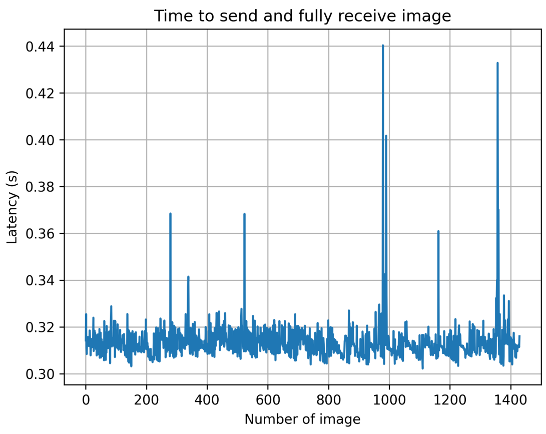


Figure 35: Transfer time Amsterdam test V1 HD

Top floor:
The second test was a test at the top floor of the building where we flew around the building and the transfer time of this test can be seen in Figure 36 from a stream of 724x1080 pixels with 10FPS. Like the results from the "Rooftop" these are expected the only abnormality is the outlier at the end, this is probably caused by the fact that TCP is used and not UDP so it ensures all images arrive but if it is stuck with one the times increase. These spikes could also be caused by the fact that we are surrounded by buildings but to be sure what causes these spikes some additional tests needs to be executed.

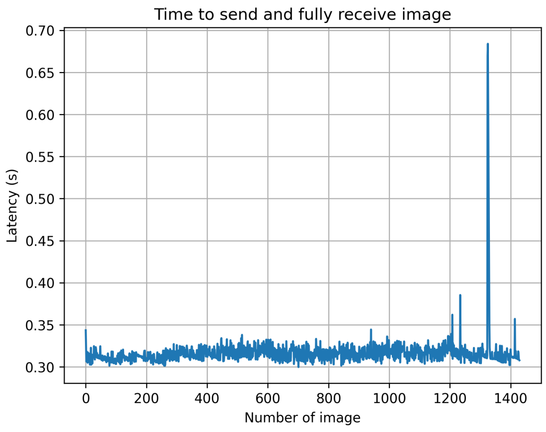


Figure 36: Transfer time Amsterdam test V2 HD

Lower around the building:
The third test was a test around the building at a lower altitude and the transfer time of this test can be seen in Figure 37 from a stream of 724x1080 pixels with 10FPS. These results are not as expected as the spikes are way bigger then 3 seconds sometimes, this can be explained with using TCP and not UDP or maybe even because of the interference of the surrounding buildings but non the less it remain big spikes.

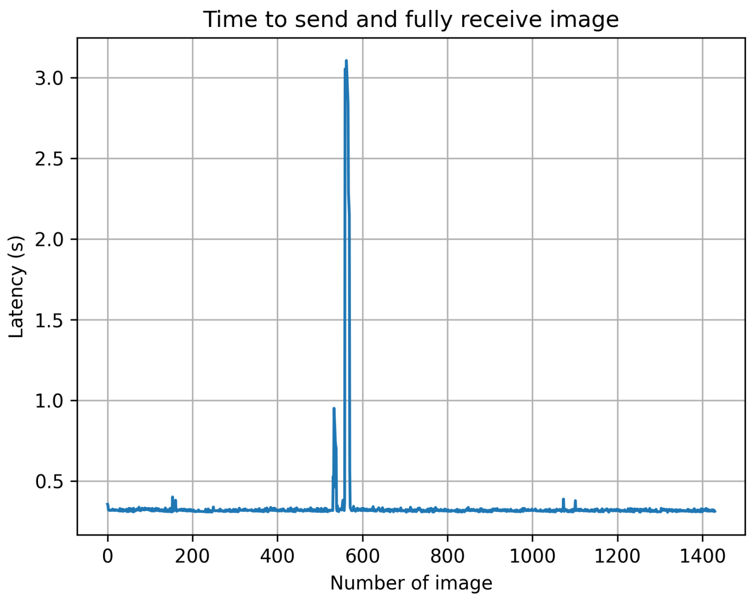


Figure 37: Transfer time Amsterdam test V3 HD

Also was Full HD tested here with a image size of 1920x1080 which reached roughly 5FPS and the transfer time of this test can be seen in Figure 38. These results where not expected with these big spikes but the average transfer time of 0.80 seconds is expected as the time for bigger images would be higher than that of smaller ones.

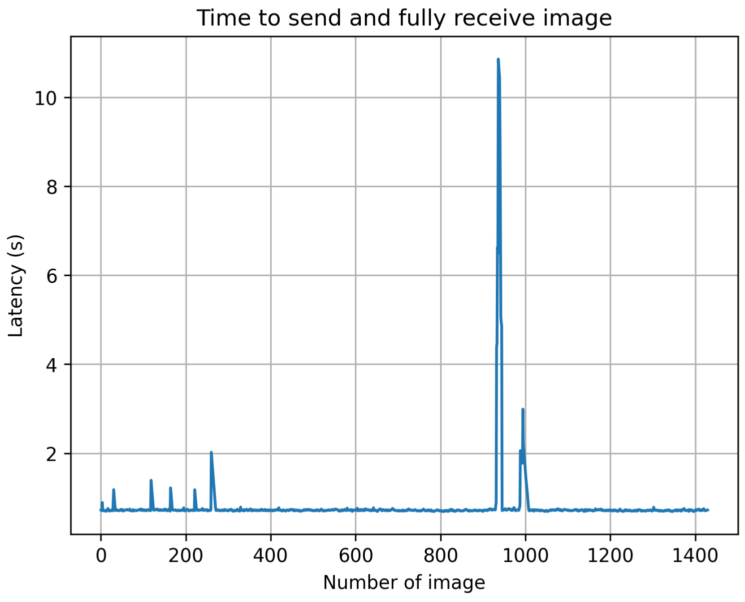


Figure 38: Transfer time Amsterdam test V3 FullHD

Away from the building:
The fourth test was a test around the building with some more space between the drone and the walls and the transfer time of this test can be seen in Figure 39 from a stream of 724x1080pixels with 10FPS. These results are good and more like the once seen in the earlier "Flight tests" in Enschede with a average transfer time of around 0.31 seconds.

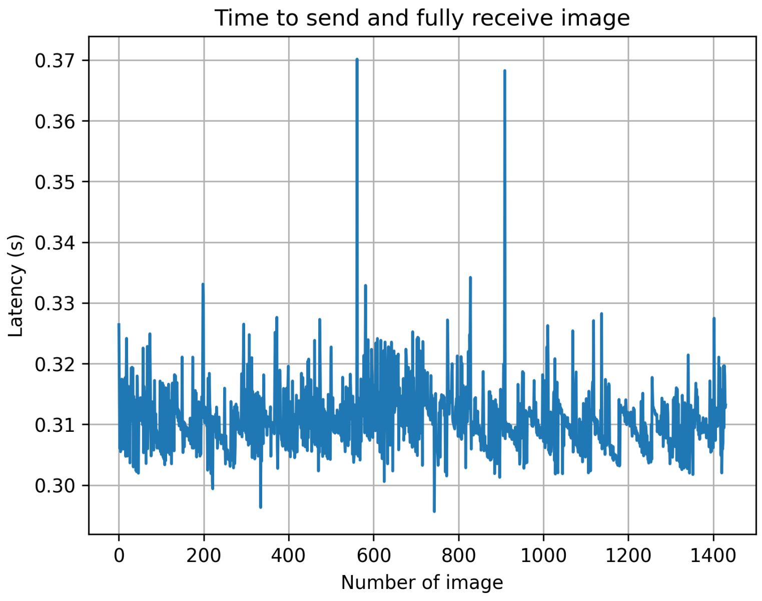


Figure 39: Transfer time Amsterdam test V4 HD

Also was Full HD tested here with a image size of 1920x1080 which reached roughly 5FPS and the transfer time of this test can be seen in Figure 40. These results are as expected with average transfer time of around 0.74 seconds.

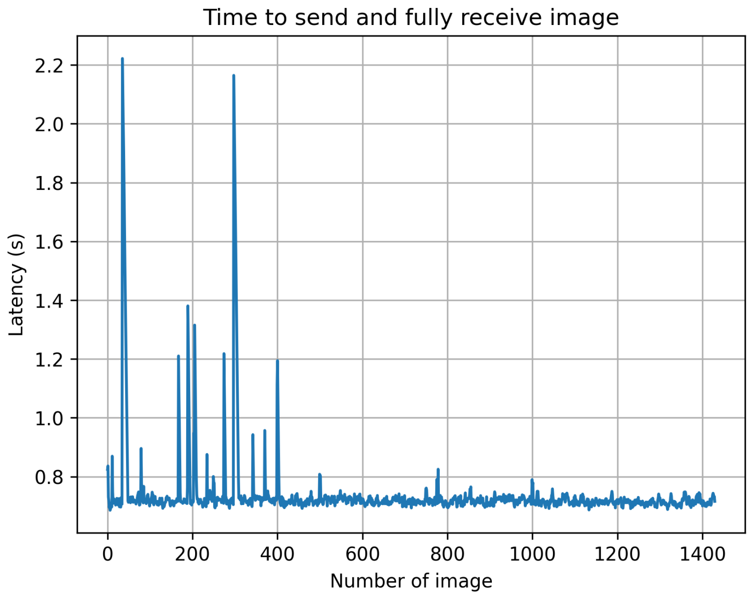


Figure 40: Transfer time Amsterdam test V4 FullHD

Rooftop V2:
The fifth test was a test roughly 10 meters above the building where we flew around the building and the transfer time of this test can be seen in Figure 41 from a stream of 724x1080 pixels with 10FPS. These results are not as expected as the spikes are way bigger then 4 seconds sometimes, this can be explained with using TCP and not UDP or becuase of the interference of the surrounding buildings but it remains a big spike.

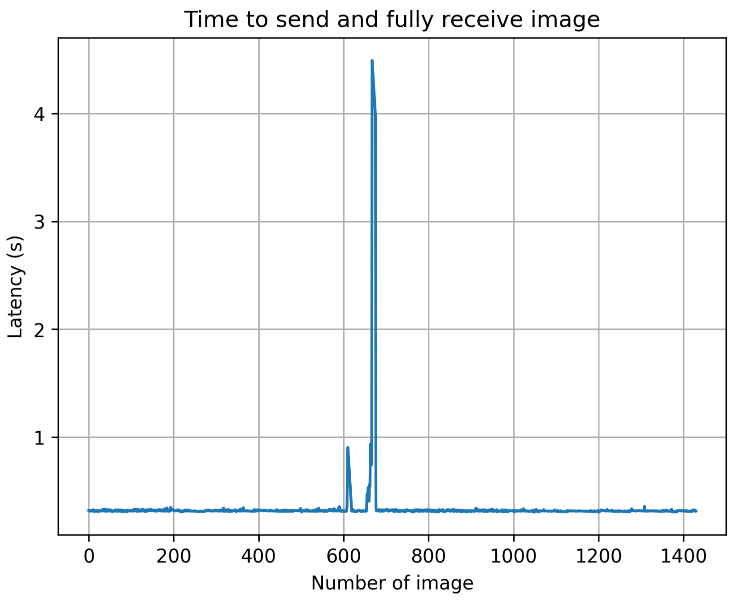


Figure 41: Transfer time Amsterdam test V5 HD

Grass field:
The last test was a test on a grass field besides the building and the transfer time of this test can be seen in Figure 42 from a stream of 724x1080 pixels with 10FPS. These results are as expected with an average transfer time of around 0.74 seconds.

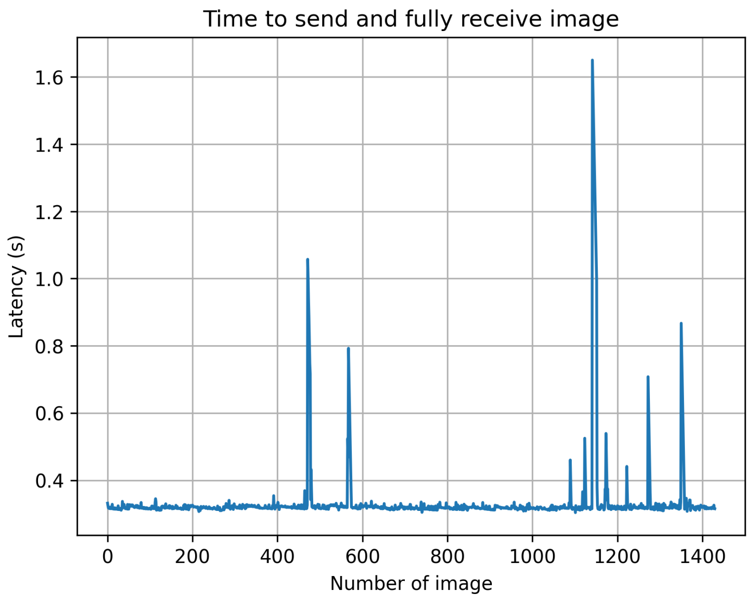


Figure 42: Transfer time Amsterdam test V6 HD

6 Conclusion

In this chapter, the conclusions are drawn, the research questions are answered, and the challenges and recommendations are discussed.

6.1 Conclusion

In this subsection, conclusions are made from the results gathered in Chapter5 and from everything else discussed earlier in this thesis.

The goal of this thesis was to develop and improve the UAV data streaming to enable efficient data streaming from drones to UT servers enabling the transmission of telemetry data and images, testing the 4G network and carrying out experiments that benchmark the performance of the designed system, and implementing the algorithms (to run on the remote server) to detect and track cars and estimate their speed leveraging both images and the drone's telemetry.

The UAV data streaming was a success as it is now possible to send telemetry data and images from the drone to a server/workstation remotely. The transfer time for the images is roughly 0.45 seconds so not real time yet but closer than it previously was with the 2 to 3 seconds it took.

The performance of the designed system was tested with the help of some tests and measurements and it became apparent that it is possible to send 724x1280 pixel-sized images with a frame rate of 10 fps from the drone, even on a height of 120 meters which shows that 4G is capable for such applications.

Then the data that was sent from the drone would be used for some algorithm that for this use case ended up being the speed estimation of vehicles that were seen in images. This was possible but with the current workstation (an OptiPlex7050 with an intel i7-7700 CPU @3.6GHzx8 and a Mesa Intel HD Graphics 630 (KBL GT2)) it was not possible because it had no supported drivers so YOLO could not make use of the GPU so it was using CPU which ORB-SLAM3 was also using so they where "fighting" for resources which caused the time needed for an image for YOLO to increase significantly from somewhere around 40 ms per image to around 1000 ms per image if not more. The code that was created was able to handle the data, make a 3D map(to a certain extent as there where problems with IMU data), detect and track vehicles from images, and convert these 2D pixels to a 3D location so the speed could be estimated. But the workstation was just not good enough so the tests with the full implementation were not good as the workstation was in a "fight" for resources for ORB-SLAM3 and YOLO which is also shown in the result from Figure 27 where it can be seen that the CPU usage is roughly 100%.

Also does YOLO have problems with detecting objects if the images are taken face down.

So, to give an overview the CPU usage is too high with YOLO and the 3D map creation on the current workstation, the time needed to send and fully receive the image (transfer time) of 724x1280 pixels at 10FPS is below 1 second, and even below a half second as the results show a transfer time of around 0.3 and 0.45 seconds, the drone is capable of sending data from a height of 120 meters without a noticeable negatively effected transfer time. And the designed code can estimate the speed of vehicles but because of the workstation, this could not correctly be shown and because of the problems with YOLO it is not optimal for a camera that is facing downward. We know the algorithm can estimate the speed as the components separately are tested and it is then able to detect, track, convert 2D to 3D, estimate the location of the vehicle, and then estimate the speed. Which is also shown in the results of the flight test where some post processing have been done.

During the Amsterdam test it became apparent that the transfer time for some tests had some spikes in it and it is not clear why these spikes appeared but some predictions can be made. As it could be assumed that this is because of the use of TCP instead of UDP as with TCP you send all the data and if something is not received you keep sending it until it eventually is but this increases the transfer time for some images. Another reason could be that the coverage of the 4G network is not as good there as on the rest of the places we tested, this can be confirmed with the data that is gathered from the different parties that where involved in the tests. This is not yet confirmed and could be part of a future investigation.

6.2 Answering research questions

Using all this information from the thesis, the research questions can be answered:

What are the best protocols to send the telemetry data and images from the drone to the workstation?
The best protocol for telemetry data is MQTT, and the best protocol for images is RTSP. These are the best as RTSP is used often for this implementation and is therefore a proven method to send images reliably and fast to another device. MQTT is the best as it scores the highest score for the DSE(Design Space Exploration) where the attention was on overhead, latency, QoS(reliability), power consumption, security, and bandwidth efficiency as these are the most important parameters for this project. These parameters were of the most importance because a good protocol should have low latency as it is critical for surveillance and many other applications to have it as close to real-time as possible. It needs to be ensured that all the data is transferred so the QoS(reliability) is important. As the implementation is a drone with a 4G connection it is critical that the overhead and bandwidth usage stay low. Also since a drone is used it has limited power, so the power consumption needs to be kept in mind when choosing a protocol. Additionally, security is also important as data can be stolen or can be altered by a hostile party. Based on the results from the DSE on these parameters MQTT came out as the best so this is the best protocol to send the telemetry data from the drone to the server/workstation.

What metrics are important for this data streaming?
The metrics important to asses the designed system would be the time needed to send and fully receive the images and telemetry data as this says a lot about the speed of the solution and if this is an improvement from the implementation of RTSP by the UT before I started working on this Thesis as for that implementation it took roughly 2 to 3 seconds to send and fully receive an image. This time is not the only important metric as then with this time that is needed to send and fully receive the image the speed of the data transfer can also be determined as the time is known and the file size is also known so, another important metric is the speed of the data transfer. That is not all also the workload on the computer itself is an important metric as you want an as low as possible workload on the computer so other things can also be run on the device or even multiple of these solutions can be used on the same device. It is also important for information about bottlenecks and resource usage. So the important metrics to assess the designed system are transfer time (time needed to send and fully receive the data), speed (data transfer in bytes/s), and CPU overhead(in % of usage).

How can these metrics be measured?
Transfer time can be measured by measuring the times from receiving the data fully and the time you start sending the data. Speed can then be calculated by using the transfer time and the file sizes by dividing the file size by the transfer time. CPU overhead can be measured by measuring the CPU usage while the system is doing nothing (baseline) and while it is during the task, then the overhead is calculated by subtracting the baseline from the CPU usage during the task.

What algorithms are needed to estimate the speed of a vehicle?
To estimate the speed of vehicles seen in an image algorithms need to be used that detect the objects, which can and is done with YOLO, and these objects need to be tracked which is done with the Bot-SORT tracker. YOLO was chosen as it is the most widely used algorithm for object detection and because it achieved better performance than its counterparts. Bot-SORT tracker was chosen as Bot-SORT takes more time to run than ByteTrack but has increased accuracy, the accuracy is more important and therefore the decision was made to use the Bot-SORT tracker. With this, the 2D pixel coordinate of the vehicle is known but this still needs to be converted to 3D so the speed can be calculated. For this to work a 3D point cloud of the environment was needed and this 3D point cloud is made with the help of ORB-SLAM3 which makes the map and also calculates rotation and translation matrices. With this information, the 3D coordinate of the vehicle can be estimated. This is done by tracing the 2D pixel back to 3D with the help of the rotation, translation, and intrinsic camera matrix, with this a 3D ray is made and with the help of the 3D map, an estimation can be made of the 3D location of the vehicle. If this vehicle is tracked over time the 3D position of it is known over time and with this 3D position information the speed can be estimated.

How reliably can a vehicles speed be estimated by these algorithms?
During the flight test from Chapter4.3, it became clear that the object detection algorithm YOLO was not able to correctly detect objects from downward facing images which means that the designed method is not reliable to estimate the speed of a vehicle. Besides this it is also not able to run on the current workstation because YOLO and ORB-SLAM3 both use the CPU and there were some issues with IMU measurements for ORB-SLAM3. But besides these points it is able to estimate the vehicles speed so if the vehicles are being detected reliably the algorithm would work reliably.

Is this approach fast enough to run in real-time?
The transfer time measured during the tests in this Thesis show that the time to send and fully receive images is around 0.45 seconds for an image stream at 10 fps, which is not perfectly real-time but is close to it. Altough the transfer speed is fast enough the workstation is not, so the workstation needs to be upgraded to make the designed system be operational in real-time.

6.3 Research objectives

Now the research objectives are mentioned:

1. To develop and improve the UAV data streaming to enable efficient data streaming from drones to UT servers enabling the transmission of telemetry data and images.
This objective has been met as the UAV can now stream its images and telemetry data to a server/workstation with the help of MQTT(for telemetry data) and RTSP(for images). It is efficiently done as the time needed to send and fully receive the image on the workstation takes roughly 0.45 seconds which is way faster than the previously measured 2 to 3 seconds.

2. To test the existing 4G network (in the Netherlands) and its speed in data streaming and carry out experiments that benchmark the performance of the designed system.
This objective has been met as there were multiple tests done on campus and in Amsterdam with Dutch Drone Delta to get a better understanding of the possibilities of 4G for data transfer. From these tests, it can be said that 4G is capable of sending images and telemetry data at even a height of 120 meters. The performance of the designed system is benchmarked with the help of a test that measures the time needed to send and fully receive the images and telemetry data but also was the CPU usage measured.

3. Implement the algorithms (to run on the remote server) to detect and track cars and estimate their speed leveraging both images and the drone's telemetry.
During this project and the flight test, this was tested and the data coming in from the drone was used to estimate the speed of the vehicles that could be seen in the images. This experiment was limited as there were problems with the GNSS and IMU. Which means that the 3D map is now not linked to 3D GNSS coordinates but is in a relative scale. Non the less, this objective has been reached to a certain extend as the algorithm that is designed is able to estimate the speed of the vehicles it sees, there are only some things that are not optimal about this designed system. As YOLO does not always detect objects correctly with the top down view of the camera and the problem with GNSS and IMU data but, apart from that it is demonstrated that the algorithm works in a relative scale and is thus able to estimate the speed of the vehicles.

6.4 Challenges and Adjustments

The research project required to switch between different operative systems and libraries, adding complexity to the work. ORB-SLAM3 was proposed for making the 3D map, but no attention was given to the integration beforehand. So, I needed to switch to Linux for the server code and add ROS to my code, which brings its share of problems and complexity.

ORB-SLAM3 was beyond my thesis scope, and is still under development by a PhD student. Which caused significant delays during my thesis.

This was a big challenge as there was little time left at the end of my thesis once I received a version of the ORB-SLAM3 code that could be implemented making it difficult to make any changes and doing tests. As I now did not have the expected time to complete the tests.

To implement ORB-SLAM3 with the current code ROS needed to be used in python which I had no prior knowledge of making it difficult.

Then the revelation came that ROS subscribers can't be put in threads with the method I wanted, because they need to be in the main thread for ros.spin() to work. This ros.spin() is important because it keeps the subscriber alive and then it can receive data but because this ros.spin() can not be used the code needed to change. It was changed to use while loops which are more demanding but make it possible to run it in threads to keep the main-thread of the code clear for the other code.

Then, there came a problem with the usual workstation of the department because it was down during my thesis. It then took a long time to get in conversation with the people in control of servers and workstations.

Once there was contact, a workstation was quickly assigned to me, but this workstation was far from optimal for this use case. There is no GPU driver available for the workstation, so CUDA (GPU) can not be used for YOLO therefore, YOLO is slower. Now, because it uses CPU, it uses the same resources ORB-SLAM3 wants to use, so they are in a fight for resources, which slows the code down drastically. So, in turn, YOLO is not fast enough with the tracker to analyze every image as ORB-SLAM3 hordes resources, so anything besides it being run is hurt.

Also, the amount of IMU measurements per second are nowhere close to what was advertised. As in the specifications it states the device can easily do more than 1000 measurements per second but I could only extract around 120 and sometimes 140 measurements per second. This limits the amount of IMU measurements before and after the image is taken, which limits the accuracy of the location, as ideally you want more IMU measurements.

Also, during my thesis, I tried to synchronize time on two devices using GPS receivers, but the GPS receivers had problems with extracting the GPS UTC real time. So, there was a time difference of 0.5s to 0.7s. Which, of course, can't be used to synchronize time. Because of this, the decision was made to use NTP, which has a time difference of around 0.15s which is good enough for this implementation.

6.5 Future work

For the continuation of this project, my recommendations would be to first focus on getting a good workstation that has a good GPU for YOLO and can run the rest of the code adequately. The second thing I would suggest is looking into using YOLO with images that are taken from a top down view. The third thing I would suggest is to look into the estimation of the speed once the locations are known. As of now, the location is taken from 2 images after each other, and we know this is not the optimal method.

After this, I would suggest working on the optimization of the code and integrating ROS better.

The fourth suggestion would be to look into narrowing down the search for the 3D point based on the ray trajectory so you do not search the full point cloud.

The fifth suggestion would be to look into getting a better IMU if the location of the drone is not good enough with the now-limited IMU measurements per image.

My last suggestions would be to expand the solution to work with multiple drones and alternatively look into using YOLO on the drone itself to detect objects and send the pixel location of these detected objects with MQTT to the workstation so the workstation does not have to use resources for YOLO. Also, some time could be used to look into GStreamer for sending and receiving images to see if this is even faster than the already used FFmpeg method.

Another thing that could be done is to analyze the network coverage data and location data in combination with the Amsterdam test results to see what causes the peaks in transfer time.

A Appendix

A.1 Time line

This is the Gantt chart of all the things I have done during the project.

A.2 Github

A.3 Possible additional tests

A.3.1 Computational overhead

To measure the computational overhead, the baseline needs to be measured, and the difference once a task is done needs to be determined.

To determine the computational overhead, the time difference between the baseline and baseline+task needs to be measured. The computational overhead is then calculated with the following formula $CO = \frac{t_1 - t_0}{t_0} \cdot 100\%$. Where t_1 is the time with the task in seconds and t_0 is the time from the baseline in seconds.

So the steps to measure the overhead would be:

1. Capture the baseline:
   Measure the time it takes for the baseline task.
2. Capture the baseline+task:
   To then measure the baseline+task time, the time is measured from when the task is executed.
3. Calculate overhead:
   Then, the computational overhead can be calculated using the following formula $CO = \frac{t_1 - t_0}{t_0} \cdot 100\%$.

A.4 CPU usage drone

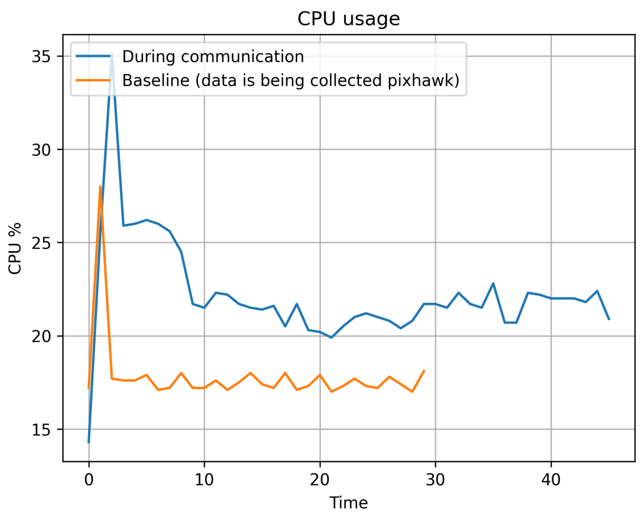


Figure 43: CPU usage drone

As can be seen in the figure above the average CPU usage during the communication and collecting data is around 22.2%. And during the collection of data it is around 17.8%.

References

[1] Jean-Paul Yaacoub, Hassan Noura, Ola Salman, and Ali Chehab. Security analysis of drones systems: Attacks, limitations, and recommendations. *Internet of Things*, 11:100218, 2020. URL: `https://www.sciencedirect.com/science/article/pii/S2542660519302112`, `doi:10.1016/j.iot.2020.100218`.

[2] Ddd. URL: `https://www.dutchdronedelta.nl/`.

[3] Baolong, Hou Jie Fan Zhifei, and Guo. Implementation of a drone-based video streamer. pages 67–74. Springer International Publishing, 2018. URL: `https://link.springer.com/chapter/10.1007/978-3-319-63859-1_9`.

[4] C Chalmers, P Fergus, C Aday Curbelo Montanez, Steven N Longmore, and Serge A Wich. Video analysis for the detection of animals using convolutional neural networks and consumer-grade drones. *Journal of Unmanned Vehicle Systems*, 9:112 – 127, 2021. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-85107386549&doi=10.1139%2fjuvs-2020-0018&partnerID=40&md5=722af2d3946c643034016e16be038fe4`, `doi:10.1139/juvs-2020-0018`.

[5] M S Punith, I Mamatha, and Shikha Tripathi. Home intrusion: smart security and live video surveillance system. *Engineering Research Express*, 6, 2024. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-85213693783&doi=10.1088%2f2631-8695%2fad9fd7&partnerID=40&md5=a9811bd37f6f94e70400f64579535a89`, `doi:10.1088/2631-8695/ad9fd7`.

[6] Fahad Al-Dhief, Naseer Sabri, Nurul Muazzah Abdul Latiff, Nik Noordini Nik Abd Malik, Mohd Khanapi Abd Ghani, Mazin Mohammed, R.N. Al-Haddad, Yasir Dawood, Mohd Ghani, and Omar Ibrahim Obaid. Performance comparison between tcp and udp protocols in different simulation scenarios. *International Journal of Engineering Technology*, 7:172–176, 01 2018.

[7] K. W Kurose J. F.; Ross. *Computer Networking: A Top-Down Approach.* Boston, MA: Pearson Education, 5 edition, 2010.

[8] Amritpal Kaur and Sarbjeet Singh. A survey of streaming protocols for video transmission. page 186 – 191, 2021. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-85123764984&doi=10.1145%2f3484824.3484892&partnerID=40&md5=4bc370b6d94f67151bf451b97eaff3ef`, `doi:10.1145/3484824.3484892`.

[9] I Santos-González, Alexandra Rivero, Jezabel Molina-Gil, and Pino Caballero-Gil. Implementation and analysis of real-time streaming protocols. *Sensors (Basel, Switzerland)*, 17, 3 2017. `doi:10.3390/s17040846`.

[10] Yan Syaifudin, Imam Rozi, Rudy Ariyanto, ROHADI Erfan, and Supriatna Adhisuwignjo. Study of performance of real time streaming protocol (rtsp) in learning systems. *International Journal of Engineering and Technology(UAE)*, 7:216–221, 3 2018. `doi:10.14419/ijet.v7i4.44.26994`.

[11] Stephen Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. *Internet Engineering Task Force, RFC*, 07 2003.

[12] Chunfang Xue, Peng Liu, and Weiping Liu. Studies on a video surveillance system designed for deep learning. 2019. URL: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-85081990603&doi=10.1109%2fIST48021.2019.9010234&partnerID=40&md5=002f97f56886d724c0bc5a21e9585570`, `doi:10.1109/IST48021.2019.9010234`.

[13] Vikas Lalhriatpuii, Ruchi, and Wasson. Comprehensive exploration of iot communication protocol: Coap, mqtt, http, lorawan and amqp. pages 261–274. Springer Nature Switzerland, 2025.

[14] What is mqtt quality of service (qos) 0,1, 2? – mqtt essentials: Part 6, 2 2024. URL: `https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/`.

[15] Mosquitto. URL: `https://mosquitto.org/`.

[16] Hivemq cloud. URL: https://www.hivemq.com/products/mqtt-cloud-broker/.

[17] It explained: Mqtt. URL: https://www.paessler.com/it-explained/mqtt.

[18] Krupa, Badelia Pratixa, Ghosh Soumya K Chaudhuri Arindam, and Mandaviya. *Optical Character Recognition Systems*, pages 9–41. Springer International Publishing, 2017. doi:10.1007/978-3-319-50252-6_2.

[19] Sankar K Pal, Anima Pramanik, J Maiti, and Pabitra Mitra. Deep learning in multi-object detection and tracking: state of the art. *Applied Intelligence*, 51:6400–6429, 2021. doi:10.1007/s10489-021-02293-7.

[20] Ross Girshick. Fast r-cnn, 2015. URL: https://arxiv.org/abs/1504.08083.

[21] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. volume 2015-January, page 91 – 99, 2015. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84960980241&partnerID=40&md5=18aaa500235b11fb99e953f8b227f46d.

[22] Tausif Diwan, G Anirudh, and Jitendra V Tembhurne. Object detection using yolo: challenges, architectural successors, datasets and applications. *Multimedia Tools and Applications*, 82:9243 – 9275, 2023. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85135697931&doi=10.1007%2fs11042-022-13644-y&partnerID=40&md5=0f5745224d43a55fec88243079fac8ad, doi:10.1007/s11042-022-13644-y.

[23] Mujadded Al Rabbani Alif and Muhammad Hussain. Yolov1 to yolov10: A comprehensive review of yolo variants and their application in the agricultural domain, 2024. URL: https://arxiv.org/abs/2406.10139.

[24] Juan Terven, Diana-Margarita Córdova-Esparza, and Julio-Alejandro Romero-González. A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas. *Machine Learning and Knowledge Extraction*, 5:1680–1716, 11 2023. URL: http://dx.doi.org/10.3390/make5040083, doi:10.3390/make5040083.

[25] Jun Wang, Zhengyuan Qi, Yanlong Wang, and Yanyang Liu. A lightweight weed detection model for cotton fields based on an improved yolov8n. *Scientific Reports*, 15, 2025. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85213971951&doi=10.1038%2fs41598-024-84748-8&partnerID=40&md5=89f14285828ff2165ebe1c73ae3f0336, doi:10.1038/s41598-024-84748-8.

[26] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. volume 2016-December, page 779 – 788, 2016. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84986308404&doi=10.1109%2fCVPR.2016.91&partnerID=40&md5=79a23b9cc271b6f314eea447fb88cc7d, doi:10.1109/CVPR.2016.91.

[27] Ultralytics yolov5. URL: https://docs.ultralytics.com/models/yolov5/.

[28] Rahima Khanam and Muhammad Hussain. What is yolov5: A deep look into the internal features of the popular object detector, 2024. URL: https://arxiv.org/abs/2407.20892.

[29] Yolov7: Trainable bag-of-freebies. URL: https://docs.ultralytics.com/models/yolov7/.

[30] Explore ultralytics yolov8. URL: https://docs.ultralytics.com/models/yolov8/.

[31] Yolov9: A leap forward in object detection technology. URL: https://docs.ultralytics.com/models/yolov9/.

[32] Yolov10: Real-time end-to-end object detection. URL: https://docs.ultralytics.com/models/yolov10/.

[33] Ultralytics yolo11. URL: https://docs.ultralytics.com/models/yolo11/.

[34] Top 10 video object tracking algorithms in 2025. URL: https://encord.com/blog/video-object-tracking-algorithms/.

[35] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. Bytetrack: Multi-object tracking by associating every detection box, 2022. URL: https://arxiv.org/abs/2110.06864.

[36] Bytetrack. URL: https://github.com/ifzhang/ByteTrack.

[37] Nir Aharon, Roy Orfaig, and Ben-Zion Bobrovsky. Bot-sort: Robust associations multi-pedestrian tracking, 2022. URL: https://arxiv.org/abs/2206.14651.

[38] Bot-sort. URL: https://github.com/NirAharon/BoT-SORT.

[39] Luca Bertinetto, Jack Valmadre, João F Henriques, Andrea Vedaldi, and Philip H S Torr. Fully-convolutional siamese networks for object tracking. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9914 LNCS:850 – 865, 2016. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84996921146&doi=10.1007%2f978-3-319-48881-3_56&partnerID=40&md5=4c0117eff2df6e96106893377bcf0a33, doi:10.1007/978-3-319-48881-3_56.

[40] M. Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp. A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174 – 188, 2002. Cited by: 10302; All Open Access, Green Open Access. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0036475447&doi=10.1109%2f78.978374&partnerID=40&md5=ca2ddf967fdcbda3c26c9e03c1347515, doi:10.1109/78.978374.

[41] Fransesco Nex. lecture slide ai for autonomous robots. AI for Autonomous Robots/3D reconstruction and SLAM/Stereo_reconstruction_2023.pdf.

[42] ray. URL: https://en.wikipedia.org/wiki/Camera_resectioning.

[43] Radhwan Madhkour, Matei Mancas, and Thierry Dutoit. A taxonomy of camera calibration and video projection correction methods. *EAI Endorsed Transactions on Creative Technologies*, 02 2015. doi:10.4108/ct.2.2.e2.

[44] Alican Mertan, Damien Jade Duff, and Gozde Unal. Single image depth estimation: An overview. *Digital Signal Processing*, 123:103441, 2022. URL: https://www.sciencedirect.com/science/article/pii/S1051200422000586, doi:10.1016/j.dsp.2022.103441.

[45] Monodepth2. URL: https://github.com/nianticlabs/monodepth2.

[46] Johannes L Schonberger and Jan-Michael Frahm. Structure-from-motion revisited. volume 2016-December, page 4104 – 4113, 2016. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84986328012&doi=10.1109%2fCVPR.2016.445&partnerID=40&md5=e007b7e836b2b044d07dbdb63db80a97, doi:10.1109/CVPR.2016.445.

[47] M J Westoby, J Brasington, N F Glasser, M J Hambrey, and J M Reynolds. 'structure-from-motion' photogrammetry: A low-cost, effective tool for geoscience applications. *Geomorphology*, 179:300 – 314, 2012. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84870300327&doi=10.1016%2fj.geomorph.2012.08.021&partnerID=40&md5=276117cb50ccf812ad06bbc5cd5241a2, doi:10.1016/j.geomorph.2012.08.021.

[48] Sjoerd van Riel. Exploring the use of 3d gis as an analytical tool in archaeological excavation practice. 3 2016. doi:10.13140/RG.2.1.4738.2643.

[49] Alif Khairuddin, Shukor Talib, and Habibollah Haron. Review on simultaneous localization and mapping (slam). pages 85–90, 11 2015. doi:10.1109/ICCSCE.2015.7482163.

[50] Carlos Campos, Richard Elvira, Juan J Gomez Rodriguez, Jose M M. Montiel, and Juan D. Tardos. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE*

Transactions on Robotics, 37:1874–1890, 12 2021. URL: http://dx.doi.org/10.1109/TRO.2021.3075644, doi:10.1109/tro.2021.3075644.

[51] Place recognition and loop closure in orb-slam3. URL: https://medium.com/@maurya.19bcd7178/place-recognition-and-loop-closing-in-orb-slam3-473fd951978.

[52] Speed. URL: https://en.wikipedia.org/wiki/Speed.

[53] Time synchronization. URL: https://www.trianglemicroworks.com/help/6tsp/Content/Getting%20Started/Time%20Synchronization.htm.

[54] Network time synchronization: Why and how it works. URL: https://www.auvik.com/franklyit/blog/syncing-clocks-network-devices/.

[55] How to synchronize clocks on network devices. URL: https://bigtimeclocks.biz/blogs/news/how-to-synchronize-clocks-on-network-devices.

[56] Opencv. URL: https://opencv.org/.

[57] Holybro pixhawk 4. URL: https://docs.px4.io/main/en/flight_controller/pixhawk4.html.

[58] m10-gps. URL: https://holybro.com/collections/standard-gps-module/products/m10-gps.

[59] Mavlink developer guide. URL: https://mavlink.io/en/.

[60] Mavlink. URL: https://github.com/mavlink/mavlink.

[61] Mavlink common message set (common.xml). URL: https://mavlink.io/en/messages/common.html.

[62] Dse. URL: https://www.sciencedirect.com/topics/computer-science/design-space-exploration.

[63] Nitin Naik. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. pages 1–7, 2017. doi:10.1109/SysEng.2017.8088251.

[64] Mqttlibrarie. URL: https://pypi.org/project/paho-mqtt/.

[65] Implementing mqtt in python. URL: https://www.hivemq.com/blog/implementing-mqtt-in-python/.

[66] About ffmpeg. URL: https://www.ffmpeg.org/about.html.

[67] Mediamtx. URL: https://github.com/bluenviron/mediamtx.

[68] Introducing json. URL: https://www.json.org/json-en.html.

[69] What is json? URL: https://www.w3schools.com/whatis/whatis_json.asp.

[70] Rosnoetic. URL: http://wiki.ros.org/noetic.

[71] rospy. URL: http://wiki.ros.org/rospy.

[72] ultralytics. URL: https://www.ultralytics.com/.

[73] Distance from a point to a line. URL: https://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line.