Table of Contents

Chapter 1. Introduction

1.1. Project context

The context of the COVID-19 pandemic has brought about many changes in our daily lives and routines, altering the way we live, work, and interact. To say that the world has changed would be an understatement.

In order to prevent the spread of the virus and protect citizens, many nations have put in place various rules to limit direct interaction between people in public areas. Among such rules are mask wearing, social distancing and travel restrictions, going as high up as total lockdowns, which were enforced in many countries where the infection counts could not be held down anymore.

To enforce these rules, cities and municipalities rely on staff and personnel to monitor people in public areas and make sure they follow the local rules of social distancing and mask wearing. Additional methods of enforcing these rules include using physical barriers, lines drawn on the ground or pedestrian traffic restrictions on streets and in stores.

Many times, though, the people responsible for enforcing these rules become overwhelmed and reduce the efficiency of their judgement, leading to overcrowding in public areas, irregular flows of pedestrians and the disrespecting of social distancing rules. This mostly happens in large open public area, such as squares and intersections, where the dispersion of people in large physical areas make it difficult for personnel to gauge if social distancing is respected. In the case that there are personnel enforcing rules in these areas, which is the case most of the times, the risk becomes much greater.

Thus, large public areas can very easily become overcrowded, making them prime grounds for spreading infection, which is contrary to the local administration’s intentions and poses a serious health risk for the actual people present in the area.

1.2. Problem statement

As described above, crowds pose a very serious health hazard for people and lead to the inevitable prolonging of lockdowns due to their ripple effects, that can only be seen weeks after exposure.

The problem of identifying crowds and monitoring those crowds is an issue of utmost importance nowadays, with more and more countries resorting to total lockdowns to prevent people from strolling around and creating dangerous situations.

Chapter 2. Project Objectives

In order to solve the aforementioned problem of crowd identification and monitoring, an innovative solution was required.

The project set out to provide a complete software and hardware solution for crowd identification and monitoring using computer vision. The chosen solution uses a very versatile UAV drone (Unmanned Aerial Vehicle) as the main platform. The advantage of this approach is the high mobility of the system, being capable of deployment virtually anywhere. Additionally, the use of edge computing devices enables real-time processing and data streaming to ground-based control stations.

The following two sections will discuss the hardware and software sides of the system for a more detailed look, defining objectives for both of them separately.

2.1. Hardware

As mentioned previously, the platform used for the system is an aerial one, namely a UAV quadcopter, more commonly known as a drone. The drone was designed and built using modern techniques such as CAD modelling and 3D printing.

2.1.1. Functional requirements

The main functional objectives of the hardware platform are:

a) To accommodate the main computer and various sensors for data gathering and processing;
b) To be compact and easy to carry to any location.

2.1.2. Non-functional requirements

Due to the nature of the UAV, the following non-functional requirements have been set:

- Low weight – the chassis of the UAV (the structural components) should not be too heavy so as not to hinder the flight of the quadcopter
- Rigidity – the chassis should be strong enough to maintain its shape even in difficult flying conditions
- Modularity – in case of damage to the UAV or its components, the design should allow easy and fast replacement of the damaged parts
- Reliability – the craft should be designed in a way that limits maintenance operations and uses long-lasting components

2.2. Software

The software system is the main functional part of the project, and it was critical to define good objectives first, before searching and settling for a stack of technologies that would be most efficient.

As mentioned previously, this thesis and project aims to provide an innovative way of identifying crowds from the air and estimate social distancing conditions and overcrowding.

2.2.1. Functional requirements

The functional objectives of the system are the following:

a) To detect people on the ground and estimate overcrowding;
b) To estimate the respecting of social distancing rules;
c) To provide insight into data by computing statistics, charts, graphs etc.

2.2.2. Non-functional requirements

Due to the nature of the application and project, the following non-functional requirements have been identified:

- Latency – the system should respond to input commands without a large delay
- Performance – the gathering and processing of data should be done in real time
- Usability - the system should be easy to understand and operate even by someone with low computer experience
- Reliability – the system should work well in various conditions

Chapter 3. Bibliographic Research

The first step when designing a new system is to research existing or similar system and learn from their successes and mistakes. Also, there may be existing methods and state-of-the-art solutions for such a problem that is being tackled in a certain project, so finding out what other have come up with that solves part of that problem is a definite help when thinking of your own solution.

Since this project is based on person detection on edge, I will first talk about what edge computing is, and how it has grown in popularity and efficiency in the past years. Then, I will compare various person detection methods that are popular in the computer science community for their good results, and talk about the chosen type of method used in this project. Finally, I will show how these methods (and others) tie into crowd detection as an adjacent research field to person detection.

3.1. Edge computing

3.1.1. Overview and benefits

Edge computing has become a very efficent way of combating the inherent problems of the more traditional form of cloud computing, in cases where latency and locality are important [1], such as in the case of this project, where data has to be processed in real time.

As opposed to the traditional types of distributed architectures, edge computing aims to make use of the client devices' computing capabilities [1], which brings data processing closer to the sensors and to the end user, which impacts latency in a positive way.

Below is a representation of the edge computing paradigm, offered by IEEE:

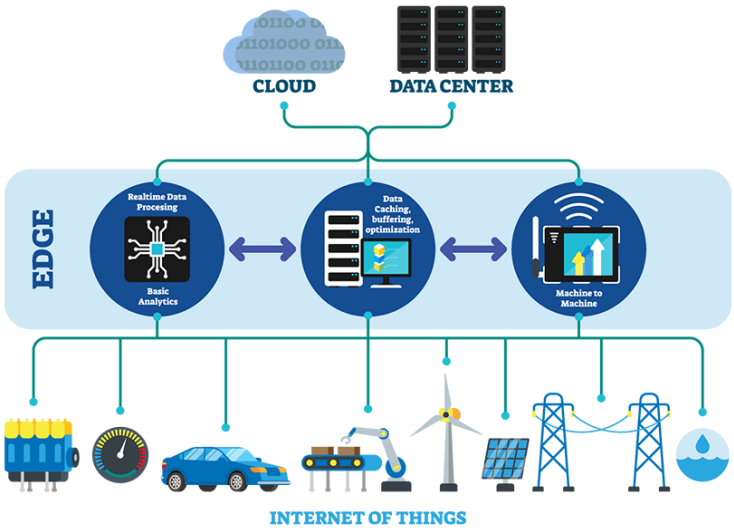

Figure 3.1 – Edge computing architecture[1]

[1] https://innovationatwork.ieee.org/real-life-edge-computing-use-cases/

Edge computing is a logical step in the evolution of devices and distributed architectures, since as devices became more powerful, more computing power was available in the hands of the user and in a more compact package. Furthermore, edge computing also relies on the local or close-by placement of servers and data-centers in the case of distributed services, to further aid latency and responsiveness [1].

3.1.2. Challenges

There are many challenges when implementing a novel system or architecture, regardless of industry or field. Edge computing, with its distributed nature and (usually) sensor-driven data acquisition, is especially difficult to implement reliably.

In the article at [2], the authors describe eloquently a few major challenges of edge computing, and I found that the one most relevant to this project, from the ones presented, is the challenge of data abstraction. The challenge of data bastraction refers to how data from the sensors is transmitted from one part of the system to the other, and how services down the road from the sensors could miss some important information due to abstraction [2]. This problem is one of great importance in large distributed systems, but also in smaller scale systems such as the one developed here and detailed in the next chapters.

In addition to this, edge devices are notorious for being underpowered when compared to cloud devices, that are usually very powerful server machines with fast processors and graphics cards. Edge devices are usually smaller and more resource-constrained, prompting the programmer to find clever ways of optimizing their systems to be more efficient and still be reliable.

3.2. Person detection

3.2.1. Overview

Person detection is a computer vision method by which we try to identify people (humans) in a given image or video stream (which is essentially just a stream of static images). Usually, it is not implemented on its own (with some exceptions, next sections), but as a part of an object detection neural network, that is designed to detect multiple classes of objects, and are usually trained on a large dataset such as Microsoft's COCO dataset [3].

There have been many methods researched and used for this task, not necessarily including neural networks, such as methods based on the histogram of gradients [4], thermal imaging [5] or stereo vision systems [6]. However, with the rise of Machine Learning models optimized for object detection (detailed in the next section), these have taken the center stage in state-of-the-art person detection algorithms.

In the following sections we will only talk about computer vision based methods for person and crowd detection, since they are the main focus of this project and research process.

3.2.2. Single-Shot Detector (SSD)

A very important step in object detection technology came with the development of the Single-Shot Detector, a method presented in [7]. This method is based on predicting bounding boxes for objects based on a set of pre-defined fixes bounding boxes, by adjusting these bounding boxes with offsets to match the objects [7] . This increases performance a lot compared to previous methods, by detecting bounding boxes directly in a single pass (hence the single-shot detector name). The architecture can be visualised below, in comparison to the previous YOLO architecture:

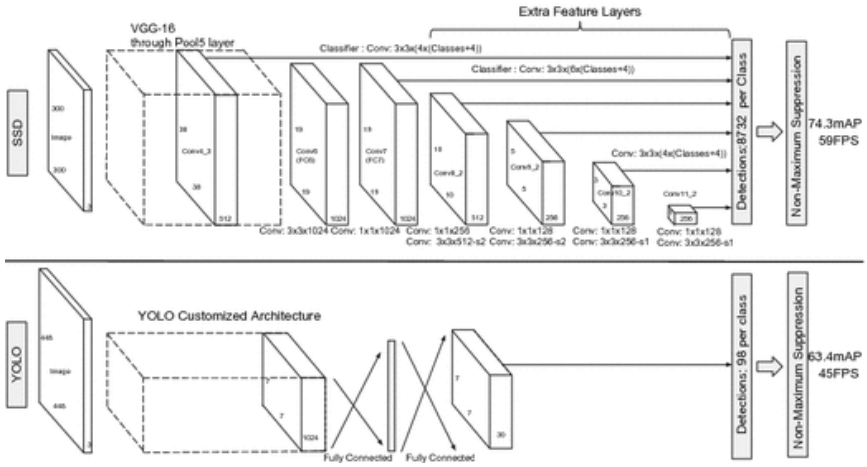

Figure 3.2 – SSD architecture [7]

As can be seen, the SSD has additional feature layers at the back of the initial architecture (VGG classifier or the YOLO detector), which are simple image classification models. The SSD generates a finite fixed set of bounding boxes (based on the number of pre-defined boxes, called anchors), together with class probabilities for each box, and in the end, a non-maximum suppression step which gives the final output [7].

3.2.3. State-of-the-art computer vision methods for person detection on edge

Even though any detection method could theoretically be adapted to work on edge, the unique constraints of the UAV environment, such as a low weight and compact size, make the choice of computing devices fairly limited. In these cases, when device size and weight is limited, it usually also means that the edge device's power is also limited.

Hence, since most algorithms are developed without such limitations in mind (usually on powerful computers or clusters), they are not optimized for running on resource-constrained devices, such as those most commonly found in edge applications. Moreover, the UAV environment constitutes an extra layer of difficulty for these devices, since all the resources (including power and storage) have to be integrated into the UAV.

Computer vision detection methods are, in this case, preferred, since only a data acquisition device (a camera) and a computer is required, whereas in the case of other methods, more devices are required (two cameras and more processing for stereo systems, an infrared camera, which is heavy and expensive, for thermal methods etc.).

Due to the above concerns, popular object detectors' creators also developed lower-parameter versions, most times called *tiny* versions, meaning that they have less parameters and a faster runtime. Of course, the tradeoff here is that the precision is generally lower than the complete parameter model.

Many object detection models come as two variants: a regular model, and a tiny model. Sometimes there are also lite versions of models, built for devices without a dedicated GPU, but these will not be discussed here. One of the most used and well-known object detection model that has a tiny version is the YOLO model [8] and its smaller version called YOLO-tiny.

Besides these, there are those object detection models that are specifically designed for resource-constrained devices. The most well-known and used of these is called MobileNet [9] a deep-learning model developed by Google, which can be used for image classification purposes. The latest version

The object detection version of this model, called MobilNetV2SSD [10] combines the original image classifier from MobileNetV2 with the SSD architecture [7] to obtain an object detector. This object detector is the one used in this project and will be further detailed in Chapter 4.

3.3. Crowd detection and crowd counting

Usually, when we talk about crowd detection, we talk about detecting a large number of people that may or may not be close together. Compared to simple person detection, crowd detection aims to make sense of more people at once and is generally an operation that is carried away further from the crowd.

Crowd detection can be done in multiple ways, but the ones we will focus on here are the computer vision based methods. Finally, we will look at how these methods could be implemented on a UAV.

Crowd counting involves estimating the number of people in a crowd by counting the number of people present in an image using various methods. These methods can inlcude neural networks, simple image processing or various other methods. It is easily understandable that we can detect a crowd by finding the people in a given image, and using other methods or computations (such as the ones described in Chapter 4 of this work) we can decide wheter they are a crowd or not.

Additionally, we might want to obtain several other parameters, such as the crowd's density, which indicated wether the people in the image are standing close or far from each other. This has been the subject of many papers, including the one at [11], where the researchers proposed a real-time method for crowd density estimation using textures and a distributed computation architecture. The real time aspect is very relevant for this project, since the final application will need to run in real time, and therefore also be a multi-processor application.

One of the most comprehensive pieces of research that I read is the report at [12], which goes over the entire problem of computer vision crowd identification and detection, giving examples of possible applications of such technologies, and showing how each type of crowd-related problem exists as part of a larger whole. The authors developed a taxonomy of the crowd-related problems, visible below:

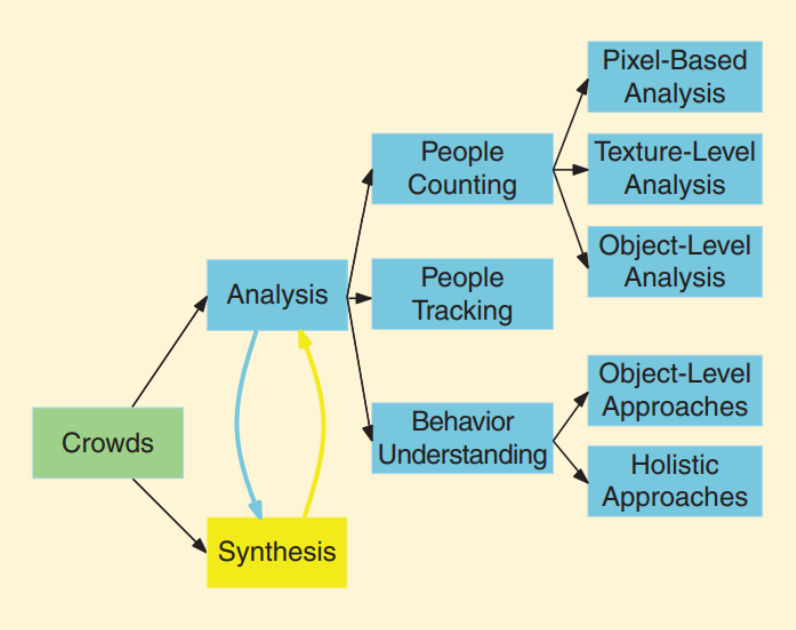

Figure 3.3 – Taxonomy of crowd related problems [12]

As can be seen in the above taxonomy, there are three main areas of crowd analysis: people counting, people tracking and crowd behavior understanding. The purpose of this project is focused on people counting, mainly on an object-level analysis, using object detection neural networks.

Object-level analysis of crowds aims to determine individual objects in the scene, such as people in this case. According to [12], this method produces better results than pixel- and texture-level analysis, because these methods only operate on lower-level features and only provide an estimate of the general density of the crowd, while individual identification of the people in that crowd is a more difficult task for these methods. They are generally only used for density estimation.

Object-level analysis is most usable in conditions when the crowd is of a generally lower density, such as the case in most public places or venues, and do not produce good results in very dense crowds, due to the fact that the detection is hindered by the complete or partial covering of the shapes of the people in the crowd by the surrounding people. This happens at crowded concerts, as a simple example, when all the people pile up in the front of the stage and are hard to identify.

Other notable advancements in the field of crowd detection and counting came from a contest called VisDrone [13], which is a contest based on images and footage taken from UAVs. Researchers and teams compete to develop the best computer vision models for one of three tasks: object detection (of objects on roads, such as cars, pedestrians, buses etc.), object tracking and crowd counting. As of the latest contest from 2020, the results presented at [14] show that the best model so far for crowd counting is FPNCC – Feature Pyramid Network for Crowd Counting. It is important to note, though, that even if this is the best available crowd counting model, it is not meant to run in real-time on a resource-

limited device, only on pre-existing aerial footage from UAVs and using powerful computing hardware.

3.4. UAV object detection

In a very interesting article I read at [15], the authors proposed a UAV-based system for decision support during emergency situations. The logic was that, when an emergency happens, a UAV could be deployed very fast and act as a surveillance device to help make quicker safety decisions.

The proposed system used a NVidia Jetson board, namely the Jetson TX2 to perform real-time object detection of response personnel and dangerous materials, as seen below:

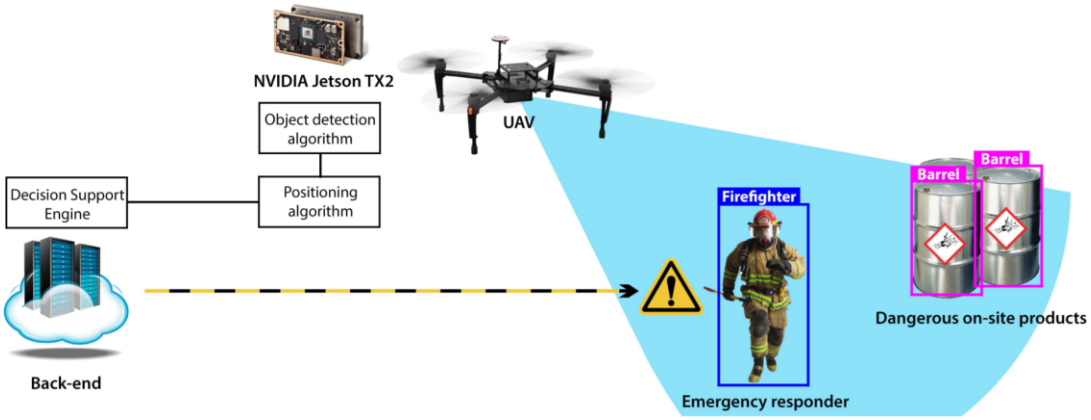

Figure 3.4 - System architecture of a decision support system based on UAVs

The goal was to enable a lightweight UAV to perform real-time object detection and identify various objects, which could then support an off-site decision system [15]. The board used in this system is a powerful mini-computer, with 256 GPU processing cores and a quad-core CPU, running at a maximum of 15W, meaning that it is very efficient. Using this board, the authors managed to obtain a maximum framerate of about 5FPS with a mean average precision of about 90% using the YOLOv2 model [8] [15].

Chapter 4. Analysis and Theoretical Foundation

This chapter presents the theoretical foundations and working principles of the proposed solution and explains the main concepts used in the implemented system. It will start with an overview of the system and then dive deeper into its components and how they interact with each other.

4.1. Outline

The proposed system is comprised of a number of different components that work together as a whole to produce the desired results that relate to the objectives defined in Chapter 2. The diagram below showcases the proposed system and has three main areas of interest: the inputs of the system (colored in blue), the processing that is done on the input data (green color) and the system's outputs (red color).

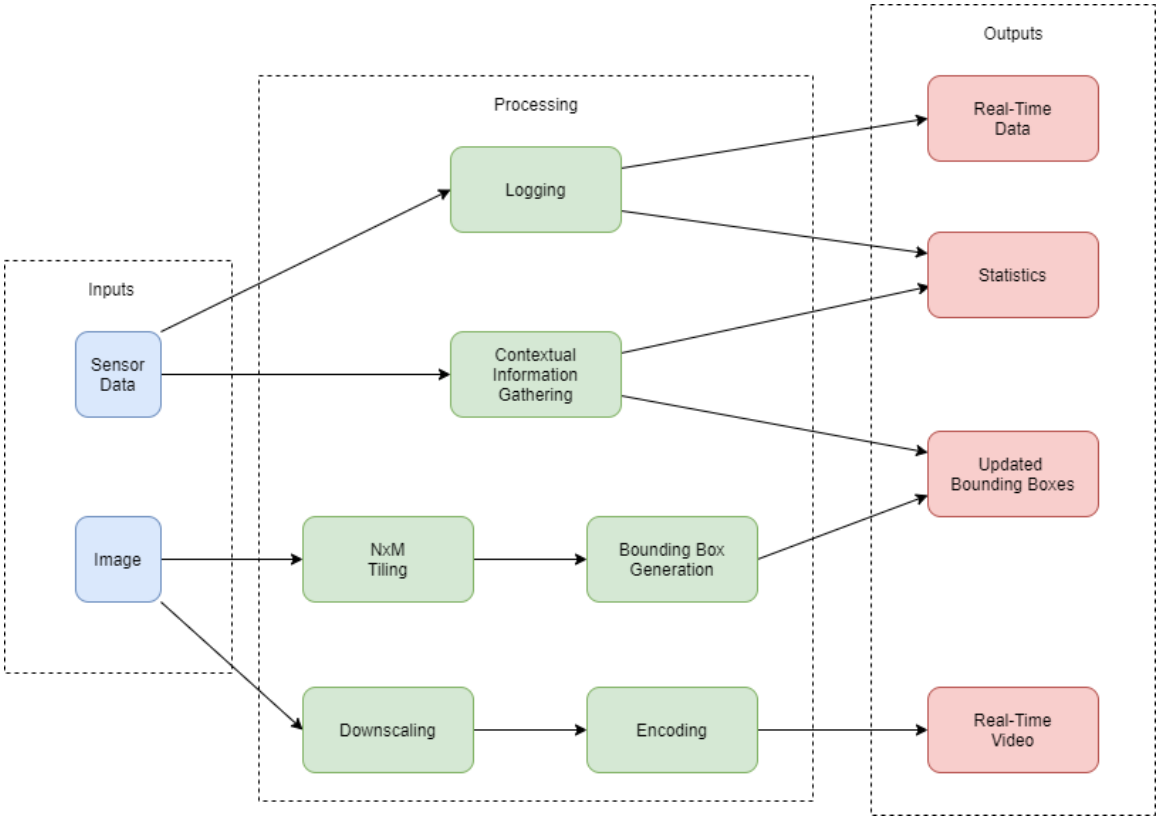

Figure 4.1 – Conceptual system diagram

4.2. Crowd counting

The most important task that the system performs is crowd counting via person detection. This means that the system must be able to detect persons in an image or a video feed and estimate the total number of people present in the scene.

This is achieved using person detection algorithms and other image processing techniques, which will be discussed in the following subchapters.

4.2.1. Person detection

Since there are physical limitations to the size and weight that a computer can have when mounted on a moving aerial platform of a UAV, an algorithm had to be selected that is optimized to run on small computers that have low processing power, but still perform accurately and fast enough for the system to be reliable and function in real time.

Therefore, the selected algorithm used for person detection is a deep-learning model created by Google, called MobileNetV2 [16], which is a model developed specifically for a mobile processing environment, made to perform well in resource-limited devices such as the ones that will be installed on the drone when it is completed.

MobileNetV2 is a modern deep-learning model that builds on top and improves the previous 2017 MobileNetV1 [9] model by using depthwise separable convolutions as its building blocks [16]:

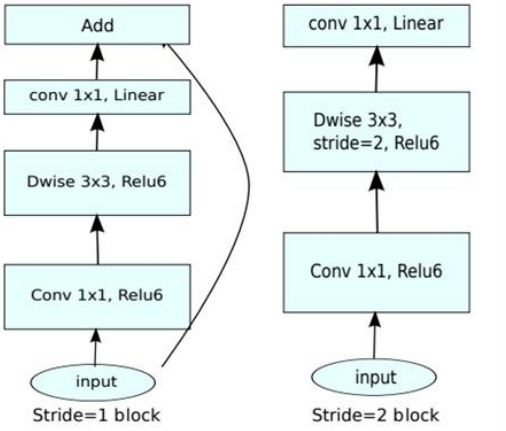

Figure 4.2 - MobileNetV2 Convolutional Block[2]

The V2 variant also adds two important characteristics to the existing architecture, namely linear bottlenecks and skip connections between these bottlenecks [16], which can be seen below in the overview of the network below:

[2] Belongs to [16]

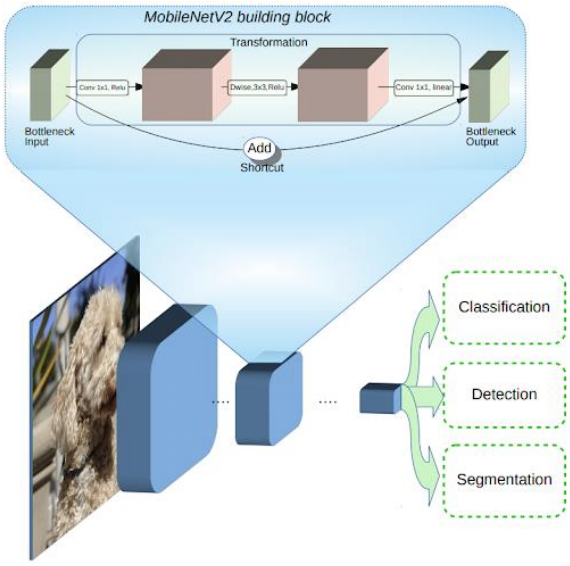

Figure 4.3 – MobileNetV2 architecture overview[3]

The MobileNetV2 deep learning model can be used either for Classification, Object Detection or Segmentation tasks. In this project, however, only the Object Detection task is relevant, since we want to produce easily understandable bounding boxes around the identified people, to help the involved parties to visually identify the people after the detection was made.
Generating bounding boxes around the identified objects (in this case humans) is the main task of this object detector model. The results we wish to obtain look like the following example of the algorithm being applied to a crowded street of people:

Figure 4.4 – Result of applying MobileNetV2 to an image of a crowded street

[3] https://ai.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html

The bounding boxes produced by the algorithm give clear indications of where people are located in the scene, which is easy to understand even for people with limited computer experience, an objective that we aimed for from the start.

In order to highlight the people who are standing too close to each other according to social distancing rules, those people will be outlined with a different color bounding box, to easily identify those who do not respect the social distancing rules, as seen in the example below:

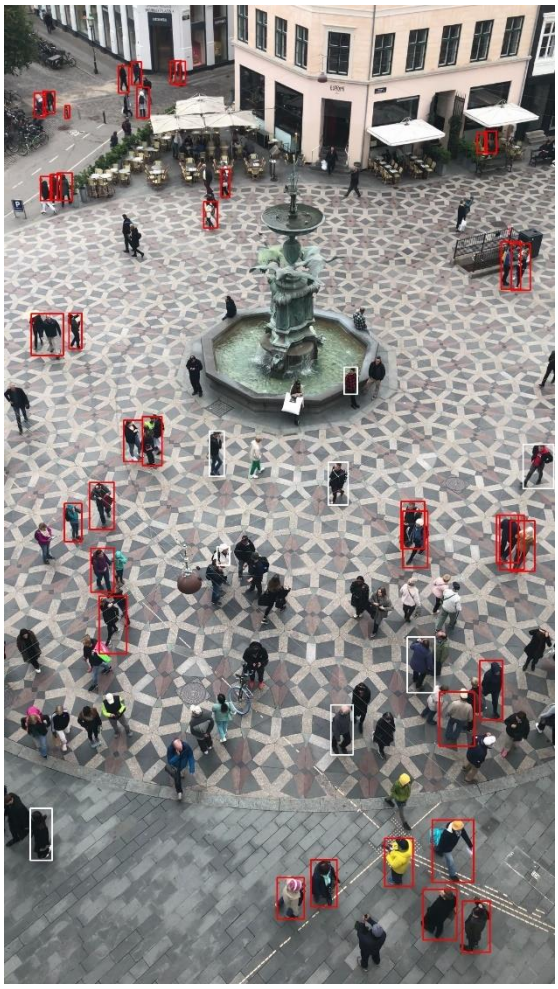

Figure 4.5 – Social distancing warnings applied on a sample image

This provides an extra layer of details in the images, due to the fact that both types of persons are highlighted, those who respect social distancing laws and those who do not.

4.2.2. Image tiling

As we have seen in the above section, the MobileNetV2 object detector produces a set of bounding boxes (also called detections) that identify the detected people in the input image. However, the detector operates on small input images (around 300x300 pixels) and therefore, if we give it a large input image (1920x1080 pixels, for example), the input image

will be resized and downscaled to fit the input resolution of the model, effectively destroying the information within the image and producing a sub-standard result. This is due to the fact that when the input image is downscaled to fit the input of the model, the details in the image get washed out, but because drone images tend to be taken at quite a high altitude, those missing details when downscaling could mean that many detections will be lost in the process, and the system will not identify the total number of people in the scene correctly.

To solve this issue, we make use of a technique called image tiling, which is helpful in detecting small objects [17]. The technique of image tiling is based on splitting the large initial input image into smaller images, that are closer in size to the size of the input layer of the object detection model [17]. The image is divided into *NxM* (or similar) smaller pieces (tiles) and detection is performed individually on each piece. In the end, the individual results are then combined to produce the final output image. The authors of [17] propose a fixed image tiling and a merging of the rsult of the individual detections with the initial image, while in [18], a dynamic tiling approach is proposed, but since there is a restrictive computational environment on the UAV, a fixed tiling was preferred, without any merging, as seen below:

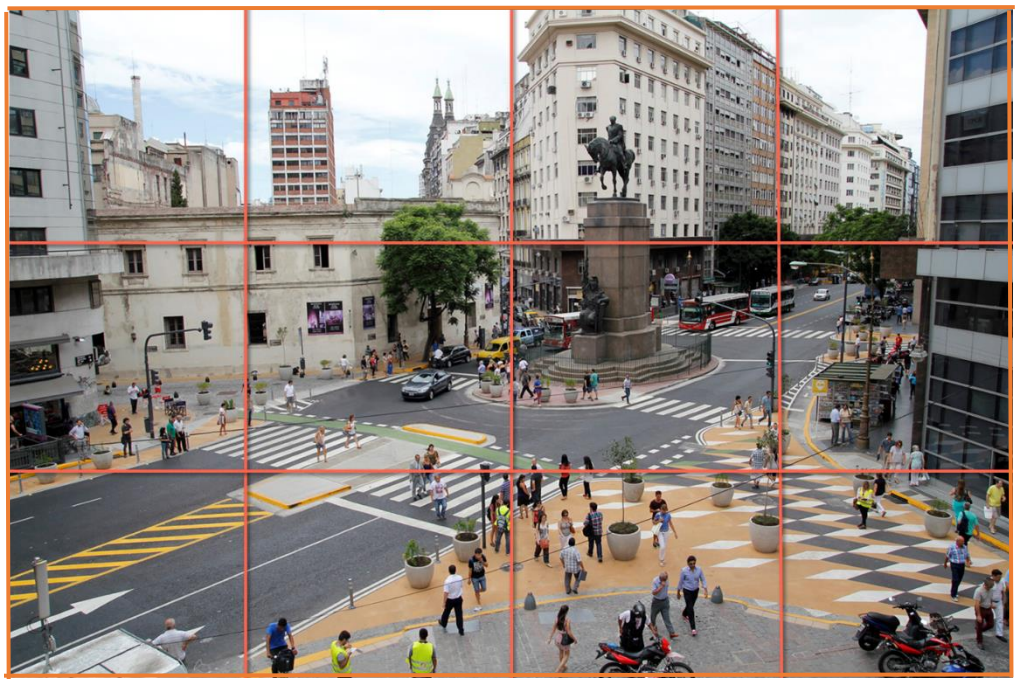

Figure 4.6 – 4x3 image tiling on a sample image

This method is the simplest one, but still gives very good results compared to simply feeding the input image in its entirety to the model. Of course, there is a trade-off with processing speed, since we need to run the detection algorithm individually *NxM* times, reducing our overall framerate by 12-fold in this case.

As mentioned previously, detection is performed individually on all the image tiles, and a set of detections is produced for each tile. If we take, for example, the second tile from the left on the bottom row and its detections:

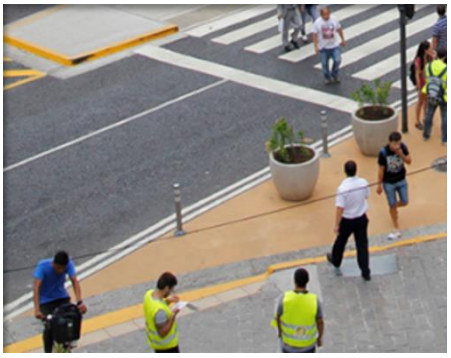

Figure 4.7 – sample tile

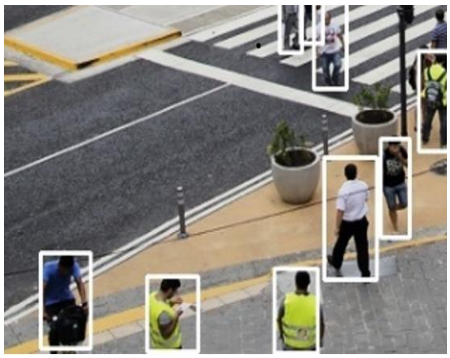

Figure 4.8 – sample tile with overlaid bounding boxes

We can see that the people have been identified correctly, but there can be the usual person that is caught in between the tiles and that does not get identified correctly, or not identified at all. When this process is applied to all the tiles, the final sets of detections can be merged, and the final output generated.

4.3. Contextual information

4.3.1. Area and distancing estimation

In order to estimate distances between the people in the scene, additional information is required, namely the height and angle of the camera pointed at the ground.

First of all, cameras obtain an image through an image sensor and a lens, which gives cameras a horizontal and vertical field of view (FOV). This field of view defines the camera's view frustum, which is the projection of the image sensor onto the scene, as seen below:

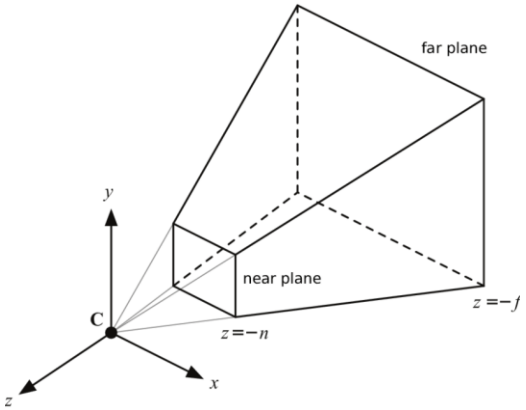

Figure 4.9 – View frustum of a camera[4]

[4] https://www.researchgate.net/figure/Projection-of-image-plane-to-ground-plane_fig3_224219714

The view frustum is a bounded region where models in the world appear on the viewing screen [19]. The near and far planes are the boundaries which define the frustum's top and bottom sides.

For a camera meant to look down at the ground, as it is the case in this project that aims to provide an UAV based solution to social distancing estimation, the far plane will be defined by the ground plane, and the near plane will be defined by the camera's image sensor, similar to the image below:

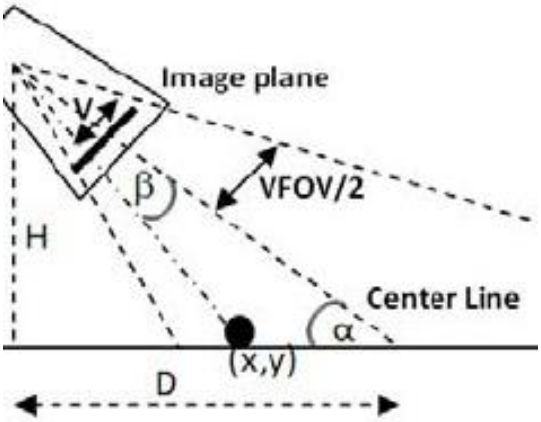

Figure 4.10 – Projection of image plane to ground plane[5]

Since cameras and sensors are very small in size, I considered the image plane as being a single point, similar to an eye. Therefore, the shape of the projection changes from a frustum to a four-sided pyramid. The base of this pyramid is a rectangle when the camera is facing straight down to the ground and gradually shapes into a symmetrical trapezoid as the camera is tilted upwards. For simplicity purpouses and for ease of computation, I considered that the base of the projection pyramid is always a rectangle.

Due to this fact, we can map the pixels of the image sensor to the projection on the ground plane and obtain the projected size of the image pixels onto the ground plane, allowing us to measure a physical distance on the ground plane between two objects by knowing the distance between them in pixels in the image plane. Of course, for this computation to be possible, the height of the camera relative to the ground plane must be known, along with the angle at which the camera is oriented to the ground, which can be between 0 and 90 degrees.

[5] https://www.researchgate.net/figure/Projection-of-image-plane-to-ground-plane_fig3_224219714

We first estimate the area of the projection on the ground plane, by computing the length of the sides of the rectangle that defines the ground projection:

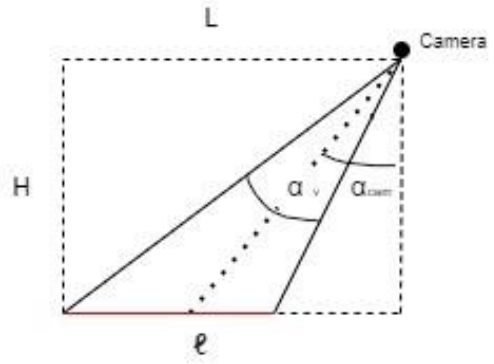

Figure 4.11 – Camera projection schematic

where *H* is the height of the camera relative to the ground plane, α_{cam} is the camera's angle towards the ground, α_v is the vertical FOV of the camera, and l is the size of the rectangle that makes up the pyramid base. Since in this scenario we considered the view from the side with the vertical FOV, the computed l value will be the height (small side) of the rectangle base of the projection pyramid. The deduced formula for the length of l is:

Equation 4.1 – Camera area side length computation

$$l = H(\cot\left(90 - \frac{\alpha_v}{2} - \alpha_{cam}\right) - \sin\left(\alpha_{cam} - \frac{\alpha_v}{2}\right)$$

Similarly, we apply the same formula for the width of the base rectangle to obtain both the width and height of the base rectangle of the projection.

Having both the width and height of the projection's base is not entirely necessary, since we could calculate the projected pixel size from one single value of these two, but we use both of them and average out the result for a more accurate computation.

After obtaining either the width or the height of the projection (or both), we simply divide its length by the corresponding pixel count of the input image in that direction.

For example, if the input image is 1920 wide by 1080 pixels high and the projection is computed to be 100m wide, it means that each projected pixel on the ground plane is 100 / 1920 = 0.052m or 5.2cm wide. This means that, if on the image we measure the distance between two objects in the scene to be 200 pixels in the input image, the physical distance on the ground plane between the same two objects is 200*5.2 = 1040cm or 10.4m.

Using this logic we can quite accurately estimate the physical distance between people on the ground from aerial images taken from the UAV, which will enable real-time estimations of how well social distancing is respected in the given scene of people.

4.3.2. Geographic positioning

Geographic positioning is an important parameter in the system, since it allows locating the crowd of people precisely within the surrounding environment, which helps the system administrator produce various statistics from the data at a later date, based on location.

Geographic positioning of the craft comes from a Global Positioning System (GPS), which sends out radio waves that are intercepted by a GPS module located on the drone (more about GPS in the next chapter).

The information received by the module from the satellites is the following:

- **Latitude and longitude**
  This is the most basic information we want to obtain by using a GPS receiver. The latitude and longitude numbers indicate a North-South and East-West meridian, at the intersection of which lies the GPS module.

- **Number of satellites involved**
  A GPS module needs at least 4 satellites to compute its location in 3D space accurately [20], so knowing the total number of satellites involved in the computation is helpful. Generally, the more satellites involved, the better.

- **Precise time**
  GPS satellites also relay the exact time-of-day from their onboard atomic clocks [20], which is also intercepted by the GPS module.

- **Ground speed**
  The ground speed is computed by using a tracking algorithm [20] that is internal to the GPS module.

4.3.3. Height and altitude

The height of the craft relative to the ground plane must be known in order to compute the area visible and thus the other parameters such as overcrowding and social distancing, according to section 4.3.1 above. Absolute altitude is not necessary for a functional reason, but purely for extra information regarding the crowd of people.

There are multiple methods for computing the relative altitude of a craft in the air [21], but the one we chose to go with in this project was the pressure-based altimeter, also known as a barometer [21]. A barometric sensor outputs air pressure in hPa with an absolute accuracy of 1 hPa [22], which is sub-meter accuracy. However, in order to calculate the correct altitude, one must also take into account air temperature, since air's density varies with temperature, using a mathematical equation called the *Barometric Formula* [23]. The equation that allows the computation of altitude based in the barometric pressure and air temperature is the following, deduced from the Barometric Formula:

Equation 4.2 – Altitude computation

$$altitude[m] = -\log\left(\frac{p}{p_0} * 100\right) * \frac{T_0 * R_0}{g * M} \text{ [23]}$$

where p is the pressure returned by the BMP280 sensor, p_0 is the standard atmospheric pressure at sea level, T_0 is the reference temperature, R_0 is the universal gas constant, M is the molar mass of dry air and g is Earth's gravitational constant.

4.3.4. Data logging

When the craft detects people on the ground (after detection has been started by the operator, data is logged continuously into .csv files and on the disk. This allows the creation of statistics at a later time and allow the administrator of the system view the overcrowding situation unfolding in time over a larger period of time. Each .csv file is saved as "*ddMMMYY.csv*" with *dd* indicating the day, *MMM* representing the first three letters of the month and *YY* representing the year. All subsequent detections from that particular day will be saved to this location.

The data that is saved into the .csv file has the following format:

Table 4.1 – logging file format

Index	Column name	Description	Unit
0	Time	Time of the data row (for ease of understanding).	HH:mm:ss
1	Stamp	The timestamp in seconds of the data row (for precision).	s
2	People	The total number of people detected by the system	-
3	Warning	The number of people from the total that are standing too close to each other according to social distancing rules.	people
4	Density	The average density of the crowd of people.	People/10m^2
5	Latitude	The latitude of the craft GPS module.	N/S
6	Longitude	The longitude of the craft GPS module.	E/W
7	Sat	The number of satellites picked up by the GPS module (indicated GPS fix quality)	-
8	Altitude	The absolute barometric craft altitude from sea level	m
9	Height	The relative craft altitude from the ground plane	m
10	Temp	The air temperature	°C
11	Angle	The tilt angle of the camera	deg
12	Area	The ground plane area captured by the camera (section 4.3.1)	m^2
13	Battery	The battery percentage of battery left	%

Additionally to the above data fields, each detection result (the image with the overlaid bounding boxes) is saved to the disk with the name format *timestamp_people*, which allows each image to be identified from the above .csv file by combining the second and third data fields.

4.4. Real-time data

Being a real-time system, there is constant data flow between the components that make up the system. The most important pieces of real-time information are discussed below.

4.4.1. Live video feed

One of the most critical pieces of real-time information is the real time video feed coming from the drone to the ground station computer in the dashboard. This helps the pilot navigate the terrain and obtain a good understanding of where the camera is pointing at. This is of course critical when trying to position the drone to obtain a good view of the crowd.

In order to obtain a low-latency image from the drone to the ground station computer the image that comes from the camera needs to be first downscaled and then encoded into an image format for viewing on the remote screen.

4.4.1.1. Image downscaling

Downscaling is the process of reducing the resolution (and therefore size) of an image [24]. It is therefore a handy tool for sending images over the network (especially over wireless connections), because the image size can be greatly reduced.

In order to perform image scaling (downscaling in this case), the input image, which is a 4:3 aspect ratio image with a resolution of about 1632x1232 will be downscaled to an image of about 410x308 resolution, which is a 4x downscaling, meaning that the image will be 16 times smaller than the original, and will be transferred wirelessly with much greater efficiency and much less packet drops. This also means that higher video transmission framerates are possible. The interpolation algorithm used is the nearest neighbor type, since it provides very fast computations [24], with low regard for image quality, since this is not an important factor in this part of the system.

4.4.1.2.Image encoding

In order to send an image over a network connection we must first encode the image, which is just a multi-dimensional array, into a regular image of a known image format, such as .jpeg. This image is then serialized and sent over the network by specialized libraries and functions that will be described in the next chapter.

4.4.2. Craft parameters

Control commands, craft parameters (orientation, battery etc. - list), location

Another area of real-time data is the data associated with the actual aircraft. This includes the craft control information for flying and the information of the craft itself, such as its orientation in space, geolocation and its altitude. The following list contains a comprehensive description of all the real-time parameters going to and from the craft:

- **Control axes**
  The drone utilizes 4 control axes, called roll, pitch, yaw and throttle. This is similar to how an aircraft flies, and the first three of these axes control a direction of rotation around each axis (for X, Y and Z), while the throttle axis controls the power of the motors. This data is sent to the drone, all the other pieces of data are sent by the drone.

- **Craft orientation**
  The craft orientation is a tuple made up of three elements, that represent the rotation angle of the craft on all of the three axes.

- **Battery charge level**
  The battery charge level is calculated from the battery's voltage and is displayed as a percentage from 0 to 100. The minimum battery voltage is 10V and the maximum is 12.6V, and the 0 to 100 range is mapped between these values linearly.

- **Power draw**
  The total power draw of the motors is displayed (since this is the largest power consumer) as the product of the battery voltage and the current draw from the battery.

- **GPS location and satellites**
  The GPS location is updated once per second via NMEA format messages (discussed in the next chapter). This gives the latitude and longitude and the number of fixed satellites used in the geolocation estimation.

- **Relative and absolute altitude**
  Relative altitude is the height of the craft relative to the ground plane, while absolute altitude is the height of the drone relative to sea level.

4.4.3. Detection results

The detection loop of the system runs on-demand, meaning that it can be started and stopped as desired by the drone operator. The detection loop produces images with bounding boxes, and by combining this data with the other contextual information described above, all the outputs of the system are generated.

The generated bounding boxes, together with the overcrowding and social distancing estimation constitute the detection results. The sent data is as follows:

- **Total people**
  The total number of people detected by the system.

- **Social distancing**
  The total number of people standing too close to each other according to social distancing rules.

- **Overcrowding**
  The average density of the detected people in the frame.

- **Total area**
  The total area on the ground visible to the camera.

Chapter 5. Detailed Design and Implementation

This chapter will present in detail the design, modeling and construction of the quadcopter and the design and inner workings of the algorithms and processing pipelines used in the software application. The hardware and software sections will be discussed in depth separately. The system's architecture will be presented, and each module will be discussed thoroughly.

5.1. Hardware

5.1.1. Overview

Below is the diagram showing the main physical components of the system and how they are connected. They will each be presented in detail in the following subsections.

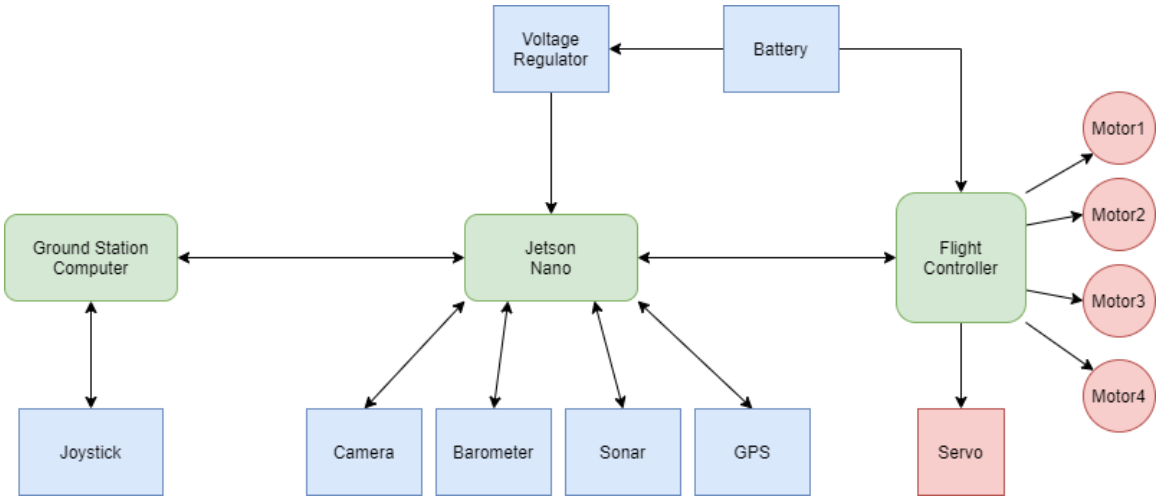

Figure 5.1 – Hardware Overview

It is important to note that items colored in blue are system inputs, items colored in green are computational and functional units and red items are physical outputs of the system.

5.1.2. Quadcopter chassis

When designing any kind of vehicle, whether it is a land, air or water-based, the designer must tailor the design to the specific application and adapt the construction and physical characteristics of the vehicle to the situations it will be used in.

In the case of this project, important design considerations had to be considered in the design stage, such as overall dimensions, weight (very critical in aeronautical applications) and stability. Of course, the craft should be able to host all the components needed for the application, including computers, sensors, and batteries for flight. Additionally, it should comprise of modular components that can be easily replaced.

Therefore, the main material chosen for building the chassis of the quadcopter was plastic, built to a desired shape using the technique of 3D printing. The plastic used was PETG, a strong and slightly elastic co-polymer, similar to the material that plastic bottles are made of (PET). This material was chosen due to its durability, ductile strength and relatively low material and manufacturing cost.

The entire quadcopter was designed in CAD, using Dassault Systèmes® *SolidWorks*™ modelling tool.

Below is a screenshot of the 3D model of the quadcopter:

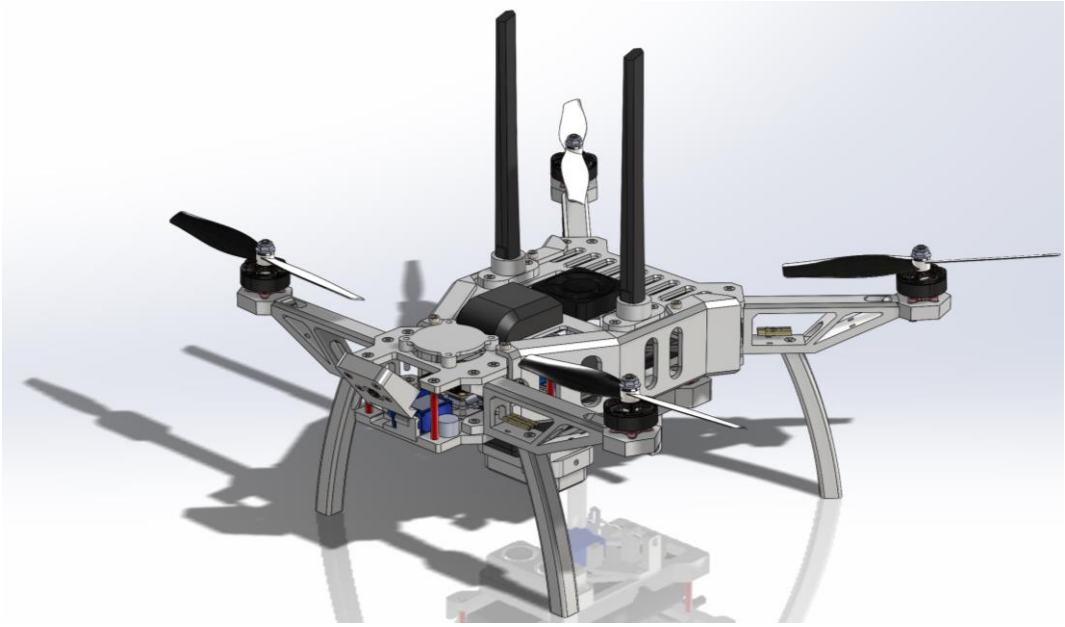

Figure 5.2 – 3D CAD model of the UAV in SolidWorks

It features a 3-deck design, where in the middle deck lie the most important components (Jetson Nano, Flight Controller, Camera). The upper deck hosts the GPS and the Wi-Fi antennas, while the lower deck contains the LiPo battery and the ultrasonic rangefinder. These components will be discussed in the following subsections.

The propellers are configured in a modified X pattern (the front arms are pulled back in comparison to a symmetrical X quadcopter) in order for them not to interfere with the camera's image) and are placed high, towards the top of the craft, to provide stability. Additionally, the motors are slightly inclined inwards by 3 degrees, giving even more flight stability to the design.

Because there are four motors, two of them spin clockwise and the other two spin counterclockwise. This is done to counteract the angular forces that the motors exert on the craft and prevent it from spinning out of control. The craft uses 2-blade propellers, which provide higher efficiencies than propellers with a higher number of blades [25].

The entire craft is suspended on 4 independent legs that can be easily replaced. The motor arms are also modular and can be easily replaced if broken. The sides are covered for extra protection in case of falls.

In order to fulfill the hardware and software objectives defined in Section 2.2, a number of critical components had to be fitted to the quadcopter. The components had to be chosen carefully, according to compatibility, compactness, and efficiency.

The following subsections will look at each individual component or group of components (in case they are similar) used in the construction of the quadcopter.

5.1.3. Main Computer

The main processing unit in the system is an NVidia® Jetson Nano™ development kit, a powerful single-board computer equipped with a CPU and GPU, ideal for deep learning and image processing tasks that are necessary in this project. The included CPU is a quad-core ARM® A57, while the onboard GPU is a 128-core Nvidia ® Maxwell architecture chip. [26]

The Nano is also equipped with USB3.0 and 2.0 ports, Ethernet, HDMI, and DisplayPort outputs, which can be handy when setting it up. It also has two MIPI-CSI camera connectors for fast camera access to the CPU. Additionally, the M.2 connector on the board accepts standard laptop Wi-Fi network cards, giving the Nano wireless capabilities, which have been used in this project.

The Jetson Nano chip and the carrier board boast impressive dimensions, being almost as compact as a Raspberry Pi, while offering a lot more performance. Due to its compact size and powerful chips, the Nano was the ideal computer for this project.

Figure 5.3 - Jetson Nano Developer Kit[6]

5.1.4. Main image sensor

The main image sensor used on the drone is the Raspberry Pi camera V2, capable of 3280 x 2464 resolution, with a MIPI-CSI serial interface for direct CPU access to the camera's image buffer. This camera has a high-quality Sony IMX219 sensor and

[6] https://developer.nvidia.com/embedded/jetson-nano-developer-kit

integrated auto white balancing for scene luminosity compensation. It has many useful modes for capturing video and images and serves as the main data acquisition device of the quadcopter.

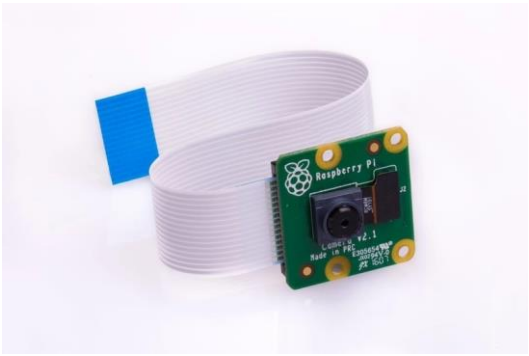

Figure 5.4 - Raspberry Pi Camera V2[7]

5.1.5. Flight Controller

This is the component that directly interacts with the motors and the camera tilt servo. The FC contains 4 separates ESCs (Electronic Speed Controllers) to control the 4 separate motors. It also provides a pass-through interface to control the servo to which the camera is attached, for tilt action. It receives MSP (MultiWii Serial Protocol) commands on separate channels and runs PID loops and stabilization algorithms to keep the drone level. It uses Betaflight firmware and has an integrated IMU (Inertial Measurement Unit) with an accelerometer and gyroscope. It is mounted in the head of the craft, behind the camera and the tilting servo, on soft rubber feet, in order to absorb vibration. It uses a 32-bit ARM® processor and has battery voltage and current draw measurement capabilities, useful for seeing the status of the battery during flight.

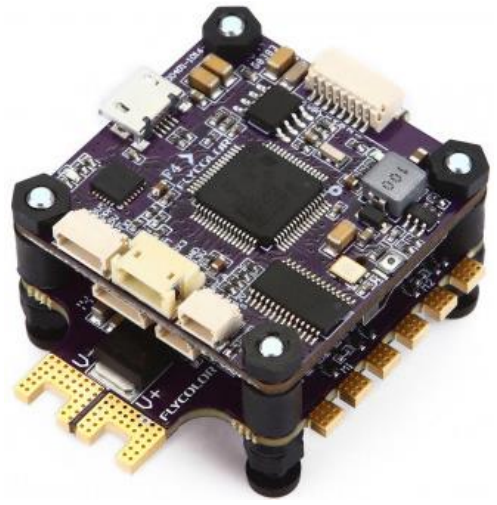

Figure 5.5 - Flight Controller[8]

[7] https://www.raspberrypi.org/products/camera-module-v2/
[8]https://hobbyking.com/en_us/flycolor-x-tower-40a.html

5.1.6. GPS Module

The GPS receiver used is a VK-162 module, based on a u-blox® GPS receiver. It has a USB interface and can connect to multiple satellites simultaneously for a high location accuracy. It is compact and lightweight, making it ideal for this application. It sends standard NMEA-formatted data strings, which make interaction with the device very easy, and the GPS coordinates and other relevant information such as elevation and ground speed can be easily access with simple code. It is mounted on the topside of the drone for the best signal reception.

Figure 5.6 – VK 162 GPS module[9]

5.1.7. Barometer

The Bosch BMP-280 is a MEMS (Micro-ElectroMechanical System) barometer and pressure senor using I2C as the communication interface. It is used to measure the absolute altitude of the craft while in the air. The integrated temperature sensor is there for more than just convenience, it is there because it is required in altitude computation, since the temperature of the air influences its density, and therefore, its pressure. Without the temperature sensor, the altitude would be measured wrong.

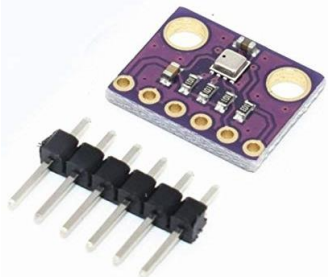

Figure 5.7 – Bosch BMP280 barometer break-out board[10]

[9]https://www.rhydolabz.com/wireless-gps-c-130_186/mini-gmouse-vk162-gps-receiver-usb-p-2026.html

[10]https://www.reichelt.com/de/en/developer-boards-temperature-and-pressure-sensor-bmp280-debo-bmp280-p266034.html

5.1.8. Ultrasonic rangefinder

This is a very common sensor used in many applications, and in this application, it has the role of finding the altitude of the craft relative to the ground. It works on the principle of measuring the time that sound travels the distance between the sensor and an object in front of it and determining the distance to that object using the known speed of sound. It is mounted on the underside of the craft.

Figure 5.8 – HCSR04 ultrasonic rangefinder[11]

5.1.9. Camera tilt servo mechanism

The camera is mounted on a servo arm that allows its tilting from 0 to 90 degrees. This is a helpful feature to have when taking shots from above and is commonly found on cinematic drones. In addition to allowing the camera to tilt, this servo also provides stabilization features, keeping the camera steady during flight. The servo used is common 9g micro servo with PWM (Pulse Width Modulation) control.

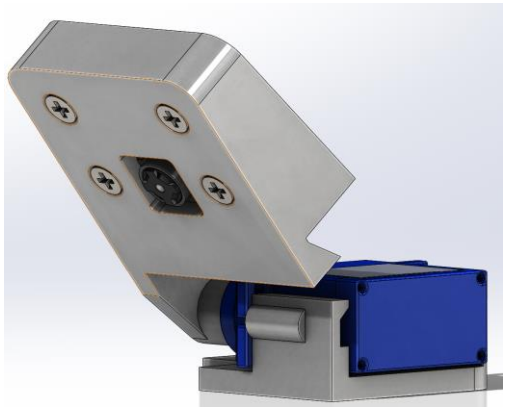

Figure 5.9 – 3D CAD model of the camera tilt mechanism

[11] https://www.robofun.ro/senzori/sparkfun-hc-sr04-senzor-distanta-ultrasonic.html

5.1.10. Wi-Fi Card

An Intel® 8265NGW Wi-Fi card mounted on the Jetson Nano provides wireless connectivity and allows wireless communication between the drone and the Ground Control Station. It features dual-band connections, 2.4 and 5GHz and dual antennas for better reception. The antennas have been mounted on the top of the drone and can be tilted and adjusted for the best reception.

Figure 5.10 – Intel 8265NGW[12]

5.1.11. Motors

The motors used on this quadcopter are a type of motors designed specifically for use in multi-rotor aircraft. They are brushless DC motors (called BLDC) that run on alternative current supplied by the ESCs on the flight controller. This type of motor is one of the most reliable and silent types of motors. They are rated at 2600KV, which is their speed constant, and means that they will spin at 2600RPM per volt applied to them. This motor rating is a good balance between power and speed. Their maximum rating is 300W for each motor, which means that four of them have more than enough power to fly the quadcopter. Two motors spin clockwise and two of them counterclockwise, and they all have self-tightening nuts on the top to hold the propellers in place during flight.

This type of motor is called an outrunner since the rotor is placed on the outside of the motor, instead of on the inside as is common. This type of geometry gives the motor more torque and a higher efficiency, as explained in [27]. The requirements of quadcopters and aerial vehicles in general fit very well with the capabilities that these motors offer, making them an ideal choice for such applications.

[12] https://www.zipy.ro/p/ali/brand-new-for-intel-dual-band-wireless-ac-8265-8265ngw-bluetooth-4-2-867mbps-m2-wireless-networkcard-better-than-7265-7260-8260/32793888109/

Figure 5.11 – Turnigy C2206 2600KV motor[13]

5.1.12. Power supply

Power to all the components is supplied by a 55Wh Polymer Lithium-Ion battery, rated at 11.1V, with 3 cells in series (commonly known as a 3S Lithium-Polymer battery, or shortly LiPo). This connects directly to the Flight Controller and power the motors through the integrated ESCs. Power to the Jetson Nano and the low-voltage sensors is converted from the battery voltage to 5V by a switching buck converter rated for 5A, which is less than the Jetson Nano will ever consume, so it is safe to use this converter for this scenario.

Figure 5.12 – 3 cell Lithium Polymer battery[14]

Figure 5.13 – XL4015 Switching Buck Converter[15]

[13] https://hobbyking.com/en_us/brushless-motor-c2206-2600kv-cw.html

[14] https://hobbyking.com/en_us/turnigy-high-capacity-5200mah-3s-12c-multi-rotor-lipo-pack-w-xt60.html

[15] https://www.optimusdigital.ro/en/step-down-power-supplies/2410-sursa-dc-dc-coboratoare-xl4015-de-5-a-intrare-de-4-38-v.html

5.1.13. Completed quadcopter

Below is an image showing the completed, fully functioning quadcopter, with all the above components installed:

Figure 5.14 – Completed quadcopter

When fully assembled, the quadcopter weighs about 1.2kg and has a propeller span of about 50cm end-to-end. The average flight time is about 20 minutes, depending on weather conditions and flight pattern.

5.1.14. Ground Station

The Ground Station is simply a computer or laptop that serves as the communication device with the airborne drone. It has a gamepad connected to it, with which the drone operator can fly the drone, and it receives real-time data from the JetsonNano computer during flight over a Wi-Fi connection. This device creates a Wi-Fi hotspot that the JetsonNano connects to, so they reside on the same network. In this project I have used my personal laptop, however any type of laptop can be used instead. The created Wi-Fi network has a format of 192.168.137.X, where X is the host identifier.

Figure 5.15 – Gamepad controller[16]

[16] https://www.amazon.com/Microsoft-Xbox-360-Wireless-Controller-Black/dp/B007EHO03A

5.2. Software

5.2.1. System architecture

The software component of the application can be split in three different applications, namely the main detection and processing application running on the JetsonNano on the quadcopter (Airborne System), the real-time dashboard used by the operator to view flight data (Client Application) and a small application whose main role is to send control commands to the drone (Control Application).

Each of these will be detailed separately in the following sub-sections. The following is an overview of the system architecture:

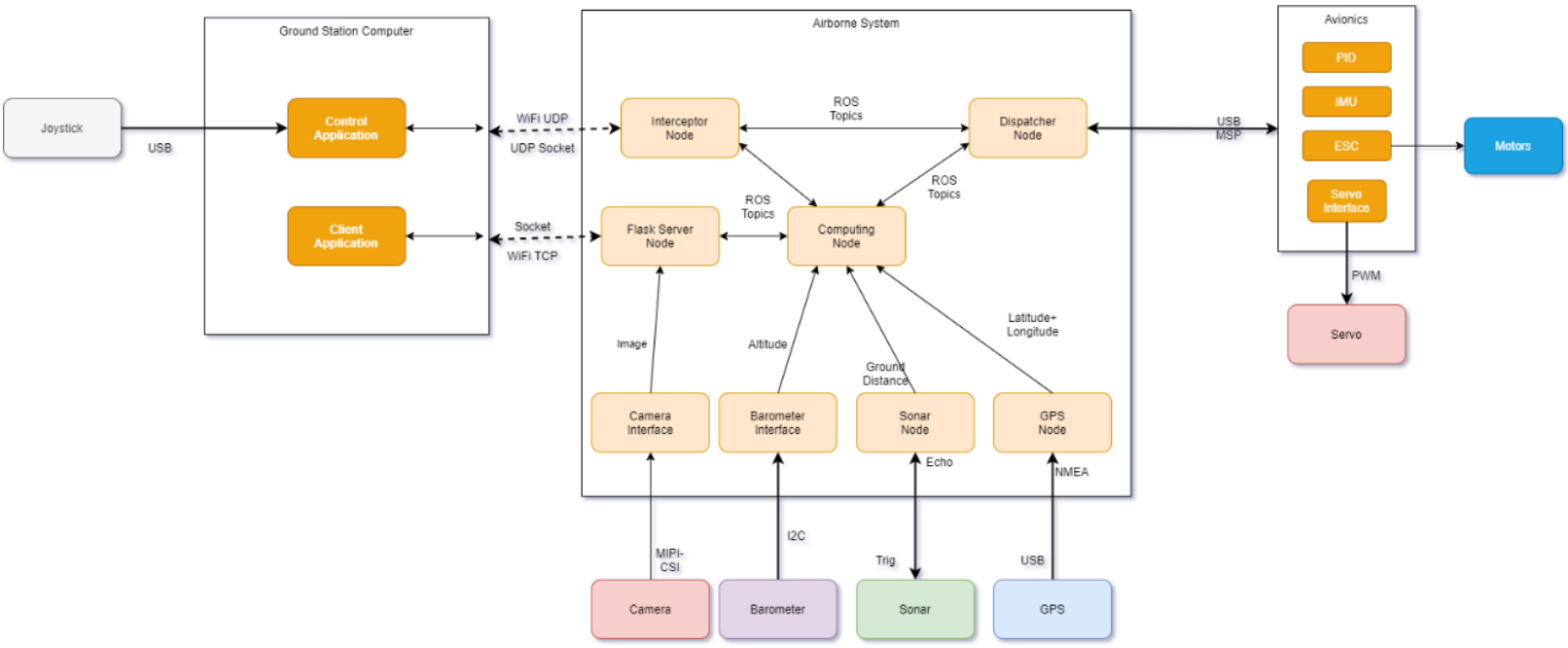

Figure 5.16 – System architecture diagram

What is important to note is that the Ground Station computer constitutes a separate device from the rest of the system. The joystick is also connected to this computer, and they together constitute the stationary part of the system. The airborne part (on the quadcopter) is made up of the Airborne System and the Avionics parts. The Avionics part is also a separate device (the Flight Controller), but it is installed in the drone, same as the Jetson Nano.

5.2.2. Technologies

The following section contains an overview of the used technologies in the software applications and the motivation behind using them.

5.2.2.1. Python

Python is a very widely used programming language, mostly due to its ease of use and readability. Python is a declarative language, with functional, object-oriented, and imperative programming influences. Python programmers enjoy a wide variety of choices when it comes to programming styles and paradigms, making this language the preferred one for computer vision and robotics applications.

The very useful thing about Python is the fact that it is not strongly-typed and can infer variable types automatically, allowing the programmer to focus on the problem they are trying to solve and not on choosing the correct data type at each step.

Python is used in the most part of the software application. Alternatives would include C++ and/or MATLAB since both of them are well known and used in these types of engineering and vision projects.

5.2.2.2. JavaScript

JavaScript is a modern language designed for the web. It is loosely typed and declarative, and has very powerful object-oriented programming paradigms, along with functional programming methods, that make it a very useful language to have in a programmer's toolbox. In addition, many frameworks and libraries have been built on it, increasing its usefulness and appeal.

5.2.2.3. Frameworks and libraries

Frameworks and libraries extend the capabilities of a language and allow a programmer to leverage a vast amount of additional resources compared to the standard language. Below the most important frameworks used in the project are presented:

- **ROS** (Robotic Operating System) Python – the most widely used framework for robotic control.

  ROS is a multi-processing framework and an environment for running Python projects (it is also available for C++). It is based on Nodes, which are separate Python processes that run the user's scripts. Communication between these nodes is not trivial (since they are separate processes), thus the ROS frameworks includes methods for inter-process communication too. These are called Topics, which is simply a communication channel that uses the local loopback network to transmit messages between Nodes. Messages can be user-defined, or a standard set can be used (including Float, Integer etc.). [28]

  Each Node in a ROS environment would normally be a Publisher, a Subscriber or both. Publishers are Nodes that only send messages (like a GPS node only sends GPS information and does not receive anything), while Subscribers are Nodes that only receive information. Usually Nodes do both, sending and also receiving messages. This is the most common usage and also the most used in this project as well. The ROS framework also provides a parameter server, which is like a central location where all Nodes can store values that can then be accessed globally by all other Nodes. This is useful for storing global values and for rudimentary node-to-node communication.

  ROS projects have to be well-structed and built using an external tool called *catkin*. They run in an environment and make use of a Master Node, which is a Node that starts before all others and routes messages between them. The

version used in this project is called ROS Melodic. In this project, the ROS environment consists of about 7-8 Nodes, each with their own specific functions, which will be detailed in the following subsections. The version of ROS used is called ROS Melodic.

- **OpenCV** – the most popular computer vision framework [29]
  It has been built from source with CUDA support, which means that computations are accelerated by having an NVidia GPU, which the JetsonNano has. It also has C++ libraries, but this project used the Python implementation. In the context of this project, OpenCV is used as an image and video processing tool.

- **Flask Server** – a Python web server [30]
  Flask is Python's library for building web servers. It uses various methods for responding to HTTP requests and sending information to client devices. In this project, it is used to send live video data to the Ground Station Computer, where it is visible in the dashboard application.

- **ReactJS** – JavaScript framework [31]
  React is one of the most used JavaScript frameworks, and it helps programmers build functional and good-looking web components easily in a dynamic manner, which would have been more difficult with simple plain JavaScript. Many component libraries have been developed for this framework, making it very easy to create websites that use data and work in real time. In this project React is used in the dashboard application to display the real-time data coming from the quadcopter.

- **ROSLib & ROSLibJS** [32]
  ROSLib and ROSLibJS are a pair of ROS libraries that make communication between a ROS Node and a ReactJS application possible. On the server side (the quadcopter ROS environment), a ROSLib Node is started separately, which waits for incoming connection requests from client applications, in this case the React application. In the React application, the ROSLibJS library is used to intercept the communication between the Nodes and display it in a browser window. In other words, the React application subscribes to the desired ROS topics, the same as any other Node, with the important distinction that messages are routed to a different device (the Ground Station Computer).

- **GoogleMapsAPI** [33]
  This is Google's API for adding maps into any application and provides programmers with tools for adding markers on a map, setting various locations and general map manipulation. It is used to display the current location of the quadcopter as it flies through the sky, information which is obtained from the GPS sensor described in the previous sub-chapter.

- **YAMSPy** [34]
  A Python library for MSP (MultiWii Serial Protocol) for communicating with MSP-enable devices such as Flight Controllers.

5.2.3. Control Application

This application reads information coming from a gamepad that is connected to the Ground Station computer, formats this data, and transmits it over UDP to the JetsonNano, wirelessly, in order for the operator to control the quadcopter.

The code is written in *Python* and uses the popular *PyGame* library to extract Gamepad information from the physical device. A gamepad has axes and buttons. An axis is a proportional input, meaning it outputs values in a range, usually between 0 and 1 or -1 and 1. Buttons are like regular buttons, and have only two states, pressed and depressed.

There are six axes on a regular computer gamepad: the two main joysticks, which make up four axes, because each stick moves both horizontally and vertically, and two additional „trigger" axes, located on the back.

The axes are mapped as follows:

- Roll – the right stick horizontal movement
- Pitch – the right stick vertical movement
- Yaw – the left stick horizontal movement
- Throttle – the left stick vertical movement

These represent the standard 4-channel control scheme for drones.

There are also twelve buttons: a group of four, with letters „A", „B", „X", „Y", a directional pad (also called a D-pad) with four buttons, only a single one pressable at once, two „shoulder" buttons on the back, and „Start" and „Back" buttons. This layout is used on gamepads mainly because of its usefulness in video games.

The *PyGame* library is used to decode this information and make it available programmatically in the application. Axes then are rounded to three decimals, and buttons are kept as they are, in their own variables. An array-shaped message is composed by concatenating the axes and the buttons, resulting in an array of size 13 (some buttons are not used). This array is packed into bytes and sent over a previously created UDP socket.

The UDP socket is created using the *socket* Python library and addressed to the IP address of the JetsonNano. Since the Ground Station computer and the JetsonNano are on the same network, the connection can be established, and the data sent continuously. The UDP socket is targeted at the IP address 192.168.137.113, which is the address of the Jetson Nano on the Wi-Fi network created by the Ground Station Computer.

The entire loop of polling the gamepad for data, assembling it, and sending it via UDP is done at a fixed configurable rate, usually around 30Hz. This rate was chosen to avoid putting unnecessary stress on the processor of the JetsonNano, because a higher rate would have meant more processing would need to be done. This rate controls the response rate of the craft to the inputs of the operator. This includes regular flight, but also controls for other functions of the system, such as starting and stopping the detection loop. This

script is adapted from a script written by Aldo Vargas in his DronePilot project, a project aimed at developing a control system for quadcopters using a motion capture system [35].

5.2.4. Airborne System

This is the main functional part of the software system. It is composed of an environment of ROS Nodes that work together to gather data, process it, and provide results. Each Node will be presented and discussed separately, along with other useful classes and files, such as the custom defined messages.

5.2.4.1. Interceptor Node

This node, together with the *Dispatcher Node* constitute the main control Nodes of the application, responsible for sending control commands to the quadcopter flight controller. The *Interceptor* runs a UDP listener that listens for the messages sent by the Ground Control Station, interprets them, does some processing to transform them into useful information and then passes them along to the dispatcher.

The UDP listener runs in a loop in a separate thread and uses the *twisted* Python library [36] for intercepting UDP communication. The intercepted message is the array sent by the Control Application above.

After receiving a message asynchronously, the Node transforms it into the format expected by the Flight Controller. The axes of the gamepad give a signal between -1 and 1 and this is directly sent to the Interceptor by the Control Application. The Flight Controller, however, only accepts PWM signals, which are numerical values between 1000 and 2000 (from 1us and 2us period of the PWM signal). Therefore, the incoming values have to be mapped to the 1000-2000 range before they can be passed on to the Dispatcher, which will then forward them to the Flight Controller.

The Flight Controller accepts 4 PWM input channels, called Roll, Pitch, Yaw and Throttle. These are specific to aircraft and control the orientation of the quadcopter. Through the mapping done by the *Interceptor*, the axes on the gamepad are mapped to these channels, meaning the drone can be controlled easily using the gamepad, by a human operator.

Additionally, this is also the place where the command input is interpreted. There are three main actions that can be performed by the drone operator, that are interpreted in this node:

- **Arming and disarming the quadcopter**
  Arming is a mandatory step before taking off with the quadcopter, as it provides extra safety measures that ensure the craft does not fly off by mistake. If the craft is not armed, take-off is not possible. Arming and disarming can be done by pressing the right trigger (RT) on the back of the gamepad. The input is debounced to one second, and disarming is only possible when the quadcopter is on the ground, to avoid accidental disarms in mid-air, which would cause the craft to fall out of the sky. When arming or disarming the craft, the Node sets a ROS parameter to a value of true or false (true = armed), which is then interpreted by the *Dispatcher Node* and sent to the Flight Controller.

- **Selecting the camera angle**

The camera is mounted on a servo, which means that precise angle control of the camera orientation is possible. When the operator presses the directional pad up and down buttons, the *Interceptor* cycles through an array of possible camera values. These can have values between 90 (vertical) and 0 (horizontal) degrees, allowing the operator to choose the orientation of the camera. Similarly to arming, a ROS integer parameter is set, containing the camera angle.

- **Starting and stopping the detection**
  Since the detection algorithm and the data acquisition and storage algorithms are computationally expensive, they are not started automatically. They have to be started by the operator after the craft has taken off. This helps conserve battery life and storage space. The detection can be started by pressing the button labeled „A" on the gamepad and stopped by pressing the button labeled „B".

The *Interceptor* also contains failsafe's in the case that the connection to the ground station is lost during flight. Initially, a warning is shown, but if the connection drops for more than a set number of seconds, the craft is disarmed, so as to avoid it flying away.
This Node is registered as a Publisher in ROS since it sends the control input to the *Dispatcher*.

5.2.4.2. Dispatcher Node

The separation of *Dispatcher* and *Interceptor* exists due to the fact that there should only be one process that interacts directly with the flight controller, and only one that interacts directly with the control input from the Ground Station Computer. The *Dispatcher* Node is subscribed to the control messages coming from the *Interceptor*.

The *Dispatcher* connects to the Flight Controller via a USB serial connection using MSP (MultiWii Serial Protocol). This connection is used to send control commands to the Flight Controller and to obtain information from it, such as the battery level, the power draw, and the orientation of the craft. This information is then published so that other nodes can access it and it can be displayed on the dashboard.

In its main loop, this node reads the ROS parameters set by the *Interceptor* (armed, the camera angle and the detection start) and assembles a dictionary of commands that are sent to the Flight Controller. The dictionary contains 6 entries, one for each configured channel of the Flight Controller. The first four are for controlling the craft, the fifth is the PWM signal that controls the camera servo, and the last one is the channel that arming is done on. A value of 1000 or 2000 on this channel signals disarming and arming, respectively. The camera servo PWM value is calculated from the angle found in the ROS parameter using the following formula:

Equation 5.1 – Servo PWM signal computation for a set camera angle

$$pwm = 1000 + (11 * camera\_angle)$$

where *camera_angle* is the value of the ROS parameter set by the *Interceptor* based on the received input from the gamepad.

Additionally, the camera is stabilized by the *Dispatcher* at the desired angle set by the operator, using the forward pitch of the UAV to correct the camera angle.

Since the *Dispatcher* is the only Node that has direct access to the Flight Controller board, it is also the Node that collects information about the flight parameters accessible only to the Flight Controller. These parameters gathered are the following:

- **Battery voltage**
  The nominal voltage of LiPo batteries is 3.7V, but when charged to 100% the voltage is 4.2V per cell [37], and usually a limit of 3.3V per cell is set as 0% charged. Therefore, the total voltage of the battery (the voltage per cell multiplied by the number of cells, since the cells are in series) is an indicator of the battery's state of charge. This value is read by the circuits present on the Flight Controller and can be read by the Python program using the MSP communication protocol. For example, for the used 3-cell LiPo battery, the 100% charge voltage would be 3x4.2V = 12.6V, and the minimal set voltage would be 9.9V (3x3.3V). Therefore, the *Dispatcher* Node requests the voltage value from the Flight Controller via MSP, and converts the result into a very familiar percentage that is easy to understand, between 0 and 100%, using the following formula:

  Equation 5.2 – Battery remaining percentage computation

$$percentage = \frac{(voltage - battery\_min)}{(battery\_\max - battery\_\min)} * 100$$

  where *percentage* is the final value between 0 and 100, voltage is the battery voltage returned by the flight controller. *battery_min* and *battery_max* are maximum charge levels of the battery; in the case of the used 3-cell battery it is 2.7V (12.6 – 9.9). The times 100 multiplication is done to convert the result to a percentage.

- **Power draw**
  Additionally, the Flight Controller is also equipped with a shunt resistor that allows it to measure current draw from the battery. This current draw comes from using the motors, the camera servo and the PID control loop of the microcontroller installed in the Flight Controller. Having both the battery voltage and the total current draw from the battery, we can compute the total power draw from the battery as the product of these two values and display it in the operator's dashboard. Usually, the UAV will draw about 8W when idle (not flying) and about 120W while hovering (stationary in the air). Maneuvering the craft or increasing the throttle input also increases the power requirement, making the battery last shorter on a single charge.

- **Craft Attitude**
  „Attitude" is the name given to the three-value set of numbers indicating the rotation of the craft along the X, Y and Z axes. This is measured using a gyroscope in the Flight Controller and computed by the Flight Controller's

firmware. The output of the computation is the rotation angle with respect to the horizontally flat position of the drone on all three axes. These are called Roll, Pitch and Yaw and correspond to the control axes mapped on the control gamepad that the operator uses. The values range from 180 to -180, with 0 being the default horizontal position, meaning the UAV is not tilted in any direction. When the craft tilts, the change is measured and can be seen on the operator's dashboard.

The *Dispatcher* Node publishes messages about all these parameters, so that the ROSLibJS library running on the Client Application can intercept them and display them to the operator's dashboard.

5.2.4.3. Flask Server Node

This node is responsible for creating a Python server using the popular *Flask* Python library [38]. The purpose of this server is to stream the live video feed from the onboard camera of the drone to the Client application, so that the operator can see from the perspective of the UAV.

The Flask server is initialized in the IP address 192.168.137.113, port 60500, which corresponds to the address of the Jetson Nano. This address will later be used in the Client Application to remotely access the video feed.

In order to access the camera's video feed, this node imports the *Camera* file from within the project, where the camera's parameters have been configured previously (more details in the sub-section about the camera interface). This node is the only process that accesses the camera feed, due to the fact that concurrent access to the camera is prohibited. This problem is solved by saving the current frame to the disk, before accessing it from another process.

The *Flask Server* Node is a separate process (as are all the nodes) that also contains two separate threads in order to increase parallelism and improve running times.

The threads are the following:

- **Camera Thread**
  This thread manages the camera interface and performs the task of running the image capture function of the camera continuously. This ensure that the *frame* variable in the *Camera* interface is always being updated, so that there is always a new frame available for processing;

- **Frame Writer Thread**
  This thread is responsible for writing the frames received from the camera to the permanent storage of the Jetson Nano. The file name remains the same to ensure that the image is always overwritten with the newest frame available. This writing of the frame to disk ensures that other processes that require the camera feed (the *Computing* node for example) can access the latest image without maintaining a handle for the *Camera* interface. In order to avoid artifacts when another process tries to read the image, this node publishes a message on a ROS topic that signals when a frame has been saved. Other nodes that want to read the frame wait for this signal and only after receiving this signal do they start reading the frame from the disk. Additionally, the

Computing node sends another signal, in which it requests a frame to be written, which signals to the *Flask Server* Node to start writing a frame to the disk. When the frame is written, the node sends the confirmation signal back, allowing the *Computing* Node to read a complete frame without artifacts.

Sending the frames to the Client Application is done when a browser (on the Ground Station Computer) access the link at *192.168.137.113:60500/live_feed*, an URL which is linked to a controller method defined in the Python file and annotated with *@app.route(„/live_feed")*, which is the Flask API's annotation for defining controller methods. The frame is sent as a Multipart HTTP response, a type of HTTP response that can be sent more than once, in this case, each response containing an image, the last frame captured from the camera. The image captured from the camera is downsized to a smaller resolution to increase the transmission speed to the Client Application. The transmission speed can be configured in FPS (Frames Per Second) and is currently set at 20FPS, which allows a smooth image while not cluttering up the communication lines unnecessarily.

5.2.4.4. Camera Interface

This Python file is used to access the camera feed of the onboard camera on the UAV. It uses Gstreamer [39] and OpenCV's *VideoCapture* to open a streaming pipeline to the camera. The pipeline is configurable, and it has been configured to retrieve a full-size image from the camera (full sensor size and full field-of-view) and downsizing it by a factor of 2 on both dimensions, effectively reducing the total size of the image by a factor of 4, without much loss in visual quality. After retrieving a frame, a global variable is written with the new frame, which can then be accessed by the process that imported this interface, in order to gain access to the camera's frame.

5.2.4.5. Barometer Interface

The purpose of this interface is to communicate with the Bosch BMP280 barometer and temperature sensor, which is connected via I2C to the JetsonNano. It is based on the bmp280 library by Feyzi Kesim [40].

The BMP280 sensor outputs air pressure in hPa with an absolute accuracy of 1 hPa [22], which is sub-meter accuracy. However, in order to calculate the correct altitude, one must also take into account air temperature, since air's density varies with temperature, using a mathematical equation called the *Barometric Formula* [23]. The equation that allows the computation of altitude based in the barometric pressure and air temperature is the following, deduced from the Barometric Formula:

Equation 5.3 – Altitude computation using the barometric formula

$$altitude[m] = -\log\left(\frac{p}{p_0} * 100\right) * \frac{T_0 * R_0}{g * M} \text{ [23]}$$

where p is the pressure returned by the BMP280 sensor, p_0 is the standard atmospheric pressure at sea level, T_0 is the reference temperature, R_0 is the universal gas constant, M is the molar mass of dry air and g is Earth's gravitational constant.

By using this formula in a simple Python function and using the bmp280 library, it is easy to obtain the absolute altitude of the UAV above sea level.

5.2.4.6. Sonar Node

The *Sonar* node is tasked with providing the relative altitude information of the craft, which it achieves by using the onboard Ultrasonic sensor on the UAV. As it runs in a separate process, it is able to run the computation algorithm multiple times per second, independently of other nodes or processes.

Ultrasonic distance measurement is based on measuring the time it takes for a sound wave to return to a receiver after being emitted. In the case of this sensor, a signal is sent on the TRIG pin for a very short time, then a signal is received on the ECHO pin after some time. Knowing the difference between these times and the speed of sound, one can compute the distance between the sensor and the object that reflected the sound wave.

The node also contains code for the cases in which the signal never returns, timeouts defined to prevent the locking of the processing loop. Additionally, the values computed are smoothed out by averaging the last values together. This ensures that random spikes in the calculated distance are smoothed out. The node then publishes the computed distance on a ROS topic, so that it becomes available to other nodes.

Of course, ultrasonic sensors such as this HR-SR04P only perform reliably up until about 4m, and are unreliable on non-reflective surfaces, such as grass. Therefore, extra processing is required in another node (the *Compute* node) to switch between computing the craft relative altitude from the ultrasonic sensor or from the barometric sensor.

5.2.4.7. GPS Node

This node manages the GPS module onboard the UAV. The GPS module sends serial lines of text containing NMEA formatted data, which is a standard specification of communication data between GPS devices [41].

Listed below are the types on NMEA standard GPS messages:

Message	Description
GGA	Time, position and fix type data
GLL	Latitude, longitude, UTC time of position fix and status
GSA	GPS receiver operating mode, satellites used in the position solution, and DOP values
GSV	Number of GPS satellites in view satellite ID numbers, elevation, azimuth, & SNR values
MSS	Signal-to-noise ratio, signal strength, frequency, and bit rate from a radio-beacon receiver
RMC	Time, date, position, course and speed data
VTG	Course and speed information relative to the ground
ZDA	PPS timing message (synchronized to PPS)
150	OK to send message
151	GPS Data and Extended Ephemeris Mask
152	Extended Ephemeris Integrity
154	Extended Ephemeris ACK

Figure 5.17 – Types of NMEA messages [52]

Out of these, only *GGA, GLL, RMC* and *VTC* are used in the GPS Node, since they provide the essential information needed for location, satellites, time, and speed.

Each NMEA message has to be decoded in order to extract useful information from it. Each NMEA message has different data fields, with their respective meaning. Below is an example of an *RMC* message, which gives time and location information:

Name	Example	Unit	Description
Message ID	$GPRMC		RMC protocol header
UTC Time	161229.487		hhmmss.sss
Status[1]	A		A=data valid or V=data not valid
Latitude	3723.2475		ddmm.mmmm
N/S Indicator	N		N=north or S=south
Longitude	12158.3416		dddmm.mmmm
E/W Indicator	W		E=east or W=west
Speed Over Ground	0.13	knots	
Course Over Ground	309.62	degrees	True
Date	120598		ddmmyy
Magnetic Variation[2]		degrees	E=east or W=west
East/West Indicator[2]	E		E=east
Mode	*A*		*A=Autonomous, D=DGPS, E=DR*
Checksum	*10		
<CR> <LF>			End of message termination

Figure 5.18 – NMEA RMC message type [52]

The GPS node runs at 1Hz (once per second) since that is the communication frequency of the GPS module.

5.2.4.8. Computing Node

The *Computing* node is the main processing node of the application. It gathers information from the other nodes, interprets it and produces results.

It uses the separately defined *Detector* (next subsection) to detect persons in the image taken by the camera, an image which was previously written to the disk by the *Flask Sever* node (explained in section 5.2.4.3). It is also responsible for logging important flight data and computing statistics from that data.

The main functions of the *Computing* node are listed and explained below:

- **Person detection**
  The *Computing* node requests a frame from the *Flask Server* node (which is the only node with access to the camera interface), which then in turn writes a frame to the disk and signals back to the *Computing* node via a ROS Topic that the frame is ready to be read. The *Computing* node read the frame and passes it through a method defined in the *Detector* (defined separately, more details in the next section). The resulting detections from this process (bounding boxes of the identified people in the received frame) are then processed to obtain a series of key metrics, such as the total number of people present, their average density (based on the area computation described in Chapter 4) and the number of people who are too close with respect to social distancing rules. This entire processing pipeline runs on a separate thread, so as not to interfere with the running of the main node program.

- **Social distancing and overcrowding estimation**
  In order to estimate social distancing, the distance between the detected people must first be computed by computing the projected size of camera pixels on the ground. By knowing the projected size of each pixel and the distance between two people in pixels from the camera image, it is possible to estimate the physical distance between those people on the ground, as explained in Chapter 4. By calculating the projected pixel size, it is also possible to compute the total area visible by the camera.
  Overcrowding estimation is done by taking the total number of detected people and dividing them by the total area visible by the camera. Assuming that the average density should not exceed one person per $10m^2$, any value that is above 1 person / 10sqm is regarded as an overcrowding situation.

- **Relative height computation**
  As stated previously, the UAV is equipped with two different altitude sensors. One is the BMP280 barometric sensor and the other one is the HC-SR04P ultrasonic distance sensor, which is mounted on the bottom of the craft and gives relative altitude information. Ideally, we would only want to use the ultrasonic sensor for relative altitude measurement, due to its better accuracy and the smaller influence of external factors upon it (such as air density). However, the ultrasonic sensor only works up until 4m, and is also unreliable over non-reflective surfaces, such as grass. Therefore, additional logic must be implemented in order to switch from the ultrasonic sensor to the barometric sensor in cases when the ultrasonic sensor can no longer function properly (above 4m and over non-reflective surfaces). This is done simply by looking at the readout from the ultrasonic sensor, and if it falls between certain values, the program knows that it has become unreliable, and switches to computing the relative altitude from the barometric sensor by diving the absolute altitude of when the craft took off by the current absolute altitude. This process happens automatically, and the relative altitude sent to the Client Application is always the correct one after this resolution.

- **Area computation**
  In order to compute the area visible by the camera, the projected pixel size of the camera's pixels must be calculated from the relative height of the drone relative to the ground and the angle of the camera with respect to the horizontal plane. After applying the formula in section 4.3.1 from Chapter 4 that considers these factors, the area can be estimated, and the projected pixel sizes can be computed.

- **Data logging**
  During flight and when detection is enabled (after the operator has taken off and pressed the „A" button on the gamepad connected to the Ground Station Computer), the *Computing* node logs relevant data about the detection results, the state and location of the UAV and the atmospheric conditions. The data is logged to CSV files on the disk that can be later accessed for viewing.

The fields of data that are saved are the following:

- *Time & Timestamp*
  The local time and the integer timestamp are recorded separately, due to the fact that it is easier for a person to understand time in hours, minutes and seconds, and for a computer to understand it as an integer.

- *Total number of people*
  As the number of total detections from the *Detector*, gives a rough indication how many people are present in the scene.

- *Number of people that are standing too close to each other*
  From the total number of people, these are the ones who are standing too close to each other, based on social distancing rules.

- *Average density of the people*
  The result of dividing the total number of people by the total area, then multiplying by 10, so that the density represents the number of people per $10m^2$

- *Latitude & Longitude of the UAV*
  The GPS location information obtained from the GPS node, from the translated NMEA messages.

- *GPS satellites fixed*
  The number of satellites involved in determining the GPS latitude and longitude. Generally, the more the better, but this is not user controllable on this GPS module.

- *Absolute altitude*
  The barometric altitude relative to sea-level, taken from the on-board barometric sensor.

- *Relative altitude*
  The altitude relative to the take-off altitude, computed with the barometer or ultrasonic sensor.

Additionally, when each row is saved in the CSV file, an image is saved with the detection results. In this image, people are highlighted with a white bounding box, while people who are standing too close to others are highlighted with a red bounding box. The image is saved in the following format: *timestamp_people*, so that it can be easily identified later.

- **Producing statistics**
  The *Computing* node continuously logs the flight and detection information, as explained in the previous point. Additionally, however, it also produces

understandable statistics from this data, in the form of charts and graphs. The process of computing these statistics is done when the detection process is stopped by the operator, with the previously generated charts being replaced with updated ones, that use the latest data available.

5.2.4.9. Detector

The *Detector* is the Python file in which the person detection methods are defined. It uses the *NVidia Jetson Inference* library [42] to load a MobileNetV2 SSD [16] pre-optimized using TensorRT. This ensures that the model runs in inference mode in the fastest time possible and gives results in the shortest amount of time.

Additionally, this is where image tiling is performed on the input image, as explained in section 4.2.2 of Chapter 4.

The model detects all the pre-defined 91 classes from the COCO dataset [43], however, only the „Person" class is considered for the scope of this project. This makes sure that the TensorRT optimized model is kept intact, which greatly speeds up the detection. Each detection candidate produced by the *Jetson Inference* library function call has the following format:

ClassID: 1
Confidence: 0.608221
Left: 299.436
Top: 349.765
Right: 524.057
Bottom: 1003.88
Width: 224.6211
Height: 654.115
Area: 146928.03
Center: (411.746, 776.823)

Listing 5.19 – Detection object format

where *ClassID* is the identifier of the class of the detected object (1 means „Person"), *Confidence* is the detector's confidence score attributed to this object, while the other fields represent the coordinates, in image pixels, of the bounding box surrounding the object, along with an automatically computed *Area* of the bounding box, and its center in image pixels. Of course, there cannot be fractional pixels in the image, so these numbers get rounded to the closest integer at a later stage.

First, the image is tiled into smaller pieces, and each piece is fed into the model using the *Jetson Inference* library as a single image. Running the inference on a single image produces an array of data elements with the above format. The size of this array gives us the total number of detected objects.

Next, the results are filtered, so that only those detections that have the *ClassID* field equal to 1 are left (i.e. the detected people). After inference is performed on a single image tile, the resulting bounding boxes are translated to their respective location in the final image, according to the steps explained in section 4.2.2 of Chapter 4.

After this step, the final image, with the drawn bounding boxes around the detected humans is returned to the *Computing* node. It is important to note that, at this step in the processing phase, the system does not yet have information about which people are standing too close to each other (i.e. disrespecting social distancing rules) and therefore all the bounding boxes are drawn in a white color. The *Computing* node will further process the detection information and the sensor data to determine which people are standing too close to each other, and will re-draw their respective bounding boxes in a red color and save the updated image to the disk.

The *Detector* is not a ROS node, it is simply a script used by the *Computing* node, used only for performing person detection on a single image.

Therefore, there are *N*x*M* image inferences made per camera frame, *N* being the number of rows of tiling and *M* being the number of columns of the tiling. This improves accuracy, but also increases the processing time of a camera frame by a factor of *N***M*. The result is a processing time of around 700ms, giving the detection loop about a 1.5FPS speed, which is acceptable in this case since we do not expect the crowd of people on the ground to make any fast moves, therefore there will only be a small difference between the processed frames, and a high FPS number would not have helped in this case. That is why it is acceptable to trade speed for accuracy, in this case.

5.2.5. Client Application

The Client Application is the application that contains the real-time dashboard that the drone operator sees when flying the drone. It provides an overview of important information about the drone and the people it detects on the ground. Below is a screenshot of the interface that the operator sees:

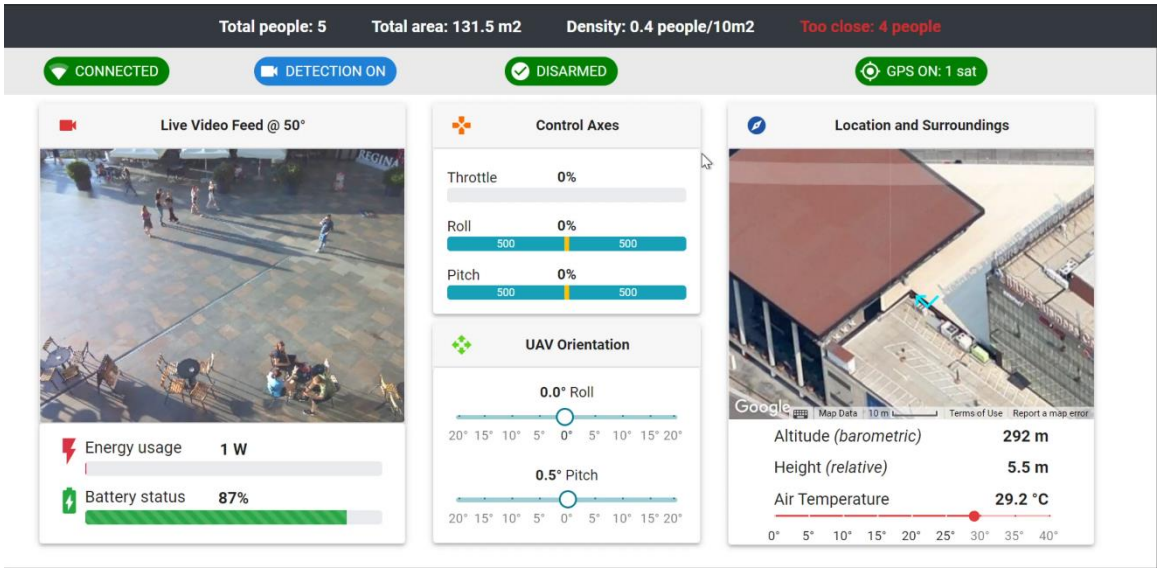

Figure 5.20 – Dashboard user interface

The interface is broken down into six main components.

On the upper left side, there is a window containing the live video feed of the camera onboard the drone, and the angle that the camera is at:

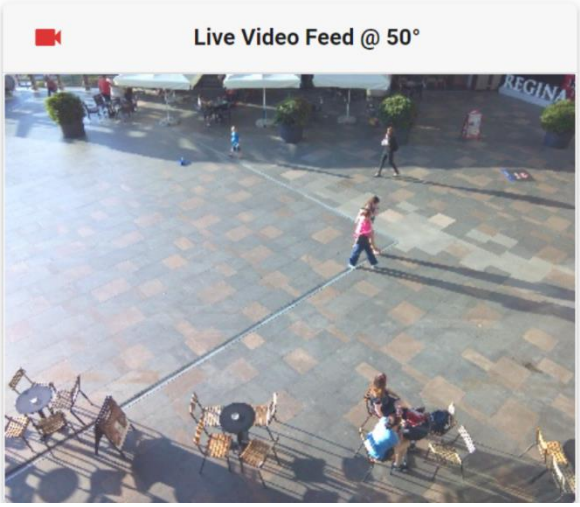

Figure 5.21 – Live Video Feed Window

Here, the operator can see the real-time image that the application is processing and can also know how the camera is angled towards the crowd, by looking at the number in the title bar of the window.

Under this window sits another window that contains power information, such as the battery percentage left and the instantaneous power consumption of the drone, measured in Watts:

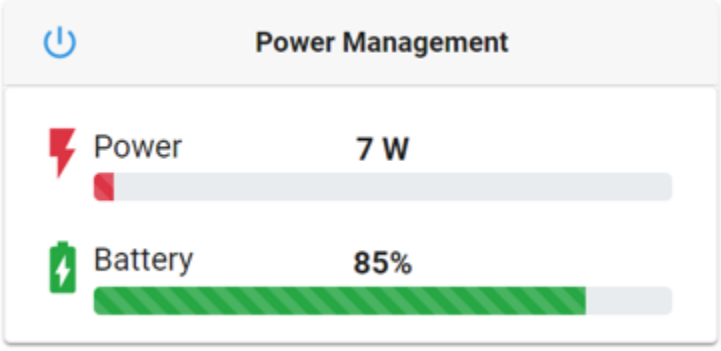

Figure 5.22 – Power Management Window

This helps the operator adjust their flight patterns to reduce the power consumption of the drone and increase its flight time.

In the middle of the screen there are two windows regarding the commands sent to the drone and its positon in space:

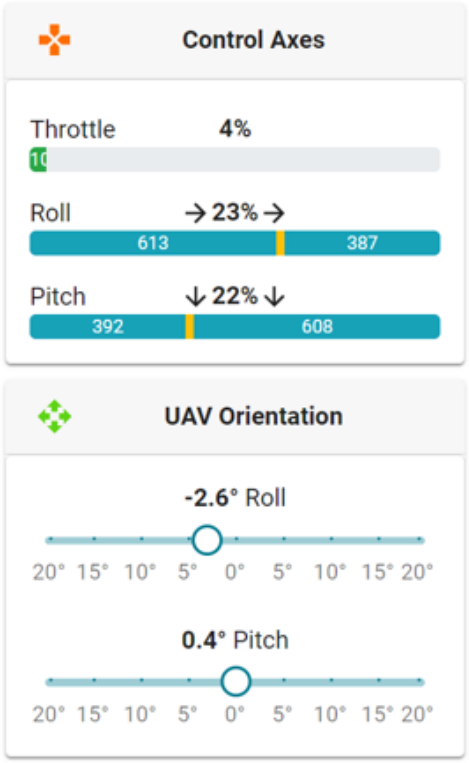

Figure 5.23 – Control Axes and Orientation

The top window represents the control axes that the operator uses to command the drone, which correspond to the three of the four axes of the gamepad (Roll, Pitch and Throttle). Yaw was left out, since the orientation of the drone on the horizontal plane is done by looking at the camera feed. Percentages of the total stick deflection are shown, as they are easy to comprehend.

Below this window there are two sliders that indicate the physical orientation of the quadcopter in space, in the Roll and Pitch directions. Moving the Roll stick up, the drone will tilt forward and start to accelerate forward, which will be visible in this window as a positive Pitch angle on the second slider. The same goes for Roll, which signals that the drone is tilted to the sides.

On the right side, there is GPS location information of the drone, along with vertical absolute and relative altitudes:

Figure 5.24 – Location and Surroundings window

Absolute elevation is the altitude of the drone relative to sea level, computed from the Barometric Formula [23] and the air pressure obtained from the barometric sensor onboard the drone.

The following piece of information available to the operator is located in the top bar of the screen:

Figure 5.25 – Status bar

Finally, the detection results are shown in real time at the top of the window:

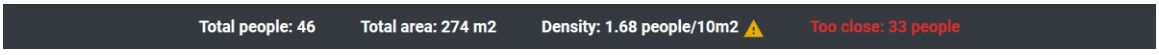

Figure 5.26 – Detection results bar

Here, the operator can see in real time the most important detection parameters, such as the total number of people, the total area visible, the average density and the number

of people who do not respect social distancing. In the case there is more than one person per $10m^2$ (configurable), the system alerts the operator with a visible warning indicator.

5.2.6. Communication Protocols

The implemented application relies on a variety of communication protocols to achieve communication between components. These will be explained and detailed below.

5.2.6.1. User Datagram Protocol (UDP)

This protocol is used in the communication beteween the Ground Station Computer and the Jetson Nano. It is used in a wireless communication channel to send commands from the gamepad attached to the Ground Station Computer to the Jetson Nano on the drone.

UDP is a connectionless protocol (unlike TCP) [44], making it ideal for low-latency, real-time applications such as this one. UDP is lightweight, meaning there are no connections to be made, no acknowledgements and no ordering of messages, so overhead and latency are inherently low [44].

UDP messages are called datagrams, which are simply packets of data, whose structure can be seen below:

UDP datagram header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Length																Checksum															

Figure 5.27 – UDP message structure [44]

It is easy to see that the size of a datagram packet is much smaller compared to the size of a TCP packet (4 bytes for UDP compared to up to 60 bytes for TCP [45]). This gives UDP the inherent advantage of being faster than TCP, but also less reliable, since data packets that are sent sequentially are not guaranteed to arrive in the same sequence at the other endpoint, or arrive at all [44]. This is not such an important issue in this project, since the connection between the computer and the Jetson Nano is strong and the drone is not that sensitive to sudden control changes, so even if a few data packets were to arrive out of sync or not arrive at all it would not pose a problem.

5.2.6.2. MultiWii Serial Protocol (MSP)

MultiWii is an open-source serial protocol designed by a community of drone-enthusiats and is used in almost all modern Flight Controllers for their functions. It a light and efficient request-reply protocol that is generic and applicable to almost all drones. [46]

In order for the Jetson Nano to interact with the onboard Flight Controller on the drone, it must use this protocol to send and request data from the Flight Controller.

Since MSP is a bi-directional protocol, there are two separate message formats to be considered when communicating with this protocol, one type of message for sending data and another type for requesting data. MSP Flight Controllers do not send out any information on their own [46], so for each piece of data that an external program wants to know, it must first send a request for that piece of data.

Below are the two types of MSP messages:

$M<[data length][code][data][checksum]
$M>[data length][code][data][checksum]

Listing 5.28 – MSP message types [46]

The request and output messages, respectively. '$', 'M' and '<' or '>' are each one byte, as are all the other parameters in brackets. *Code* is a specially designated MSP code that represents a certain field of data. For example, *MSP_ATTITUDE* is the code for retrieving the 3D orientation and rotation of the drone in space, and *MSP_BAT* is used for retrieving the battery voltage information [46].

5.2.6.3. ROS WebSocket

Rosbridge_server is the ROS node included in the default ROS Melodic distribution that is capable of opening a WebSocket connection, in order for the Client Application to be able to use ROSLibJS to listen to the ROS topics that are sent between Nodes. [47]

The Rosbridge server converts JSON messages that it receives from the clients (the Client Application in this case) into ROS calls and vice-versa, allowing clients to receive ROS messages from ROS topics like any other Node in the ROS runtime. Data is sent over a WebSocket connection, which is a bidirectional low-latency communication method. [47]

5.2.7. ROS Launch File

The ROS Launch file specifies to the ROS Core system which ROS Nodes to start and which parameters to set [48]. This is useful in multi-node applications such as this one, since it is easier to write in a single location all the files that are involved in running the application and starting them all at once as opposed to running all the files separately.

Below is a screenshot of the actual launch file used to launch all the nodes of the application:

```
<launch>

    <param name="/physical/camera_angle" type="int" value="90"/>

    <param name="/run/armed" type="bool" value="False"/>
    <param name="/run/detection_started" type="bool" value="False"/>

    <param name="/udp/timeout_threshold" type="double" value="2.0"/>

    <param name="/serial/flight_controller" type="string" value="ttyACM0"/>
    <param name="/serial/gps_vk" type="string" value="ttyACM1"/>

    <node name="ComputingNode" pkg="drone" type="ComputingNode.py" output="screen"/>
    <node name="Sonar" pkg="drone" type="SonarNode.py" output="screen"/>
    <node name="GPSProvider" pkg="drone" type="GPSNode.py" output="screen"/>
    <node name="Interceptor" pkg="drone" type="InterceptorNode.py" output="screen"/>
    <node name="Dispatcher" pkg="drone" type="DispatcherNode.py" output="screen"/>
    <include file="$(find rosbridge_server)/launch/rosbridge_websocket.launch">
        <arg name="port" value="9090"/>
    </include>
    <node name="FlaskServer" pkg="drone" type="FlaskServerNode.py" output="screen"/>

</launch>
```

Figure 5.29 – ROS Launch File used in the project

The launch file is an XML-type file that has formats for specifying nodes to be started and parameters to be set.

For each parameter, a path (name) is set, so that a tree-like structure is created, and parameters can be grouped into categories and components more easily. In addition, a type is specified, along with an initial value, which can be changed later from code. For example, the first parameter in the image is the value that represents the camera angle. The default value is 90, meaning that the camera will always be facing forward when starting the ROS nodes. This value is changed later when the operator presses the button to tilt the camera down or up, modifying the value in the parameter.

In order to access and modify the ROS parameters, the following two functions are used in the Python code:

value = rospy.get_param(“param_name”)

and

rospy.set_param(“param_name”, value)

Listing 5.30 – ROS parameter retrieval and writing

where *param_name* can be any of the above names, like *"/physical/camera_angle"* for example, in order to set the camera angle.

In the second part of the file, the nodes that are started are specified with a name, a package (the ROS package name given when creating the project) and an actual Python file that contains the script to run, which are the Nodes discussed in the previous subsections.

The <*include*> tag specifies Nodes that do not pertain to the current package, that should also be run. In this case, it is the Rosbridge server, with its parameters for the WebSocket, which is port 9090. The Client Application will use this port and the ROSLibJS library to intercept communication between the launched nodes

Chapter 6. Testing and Validation

The following chapter will describe the testing scenarios that we considered and the results we obtained. We will first look the choice was made when selecting various hardware components and detection parameters, then follow up with detection results from aerial footage and in the end, we will look at some statistics generated from the recorded data.

6.1. Main computer comparison

One of the most challenging implementation choices to make was the selection of the drone's onboard comoputer.

Taking into consideration the stated software objectives from Chapter 2, the main computer would need to be powerful enough to run all the needed computations and have all the required interfaces, both wired and wireless. Due to the fact that the computer would also need to be airborne, it would have to be relatively small in size and lightweight. Edge computing devices usually are smaller in size and weight, but not always, since even a laptop could be considered an edge computing device if it is used in a self-driving vehicle. Of course, a laptop is completely out of the ballpark for this application, since I knew a small device was necessary.

The main points I took into consideration when choosing a computer were the following:

- **CPU core count**
  The number of CPU cores and their architecture gives a good indication of the responsiveness of the computer and the processing capabilities.

- **GPU TOPS**
  The GPU aids image processing commputations and neural network inference speeds. Performance is measured in trillion operations per second (TOPS)

- **Communication interfaces**
  Very important in order to be able to communicate with the sensors and the ground station computer.

- **MobileNetV2 inference speed [ms]**
  Since this is the algorithm I decided to use for the project, the selected computer's inference speed on this algorithm was an important consideration.

- **Weight [g]**
  A very important point in UAV edge systems, weight has an impact on flight time and craft maneuvrability.

- **Total Cost [$]**
  The price of the computer and the necessary accessories.

- **Price to performance ratio**
  The total cost multiplied by the inference speed. Lower is better

The contenders for this selection were the folowing computers and accesories:

- **Raspberry Pi 4 Model B**
  A very popular single board computer with very capable specifications, a good CPU, but no dedicated GPU. [49]

- **Raspberry PI 4 Model B + Google® Coral™**
  The same board as above, but with an added Google® Coral™ USB ML accelerator capable of an added 4 TOPS (Trillion operations per second) [50]

- **NVidia Jetson Nano developer kit**
  A small module and carrier board with a quad-core CPU and a 128-core GPU that delivers very good performance. [26]

- **NVidia Xavier NX developer kit**
  A high-end large module with more CPU and GPU cores. [51]

Below is the table with the above data:

Table 6.1 – Main Computer Comparison

Board	CPU	TOPS	Interfaces	Speed	Weight	Cost	PPR
RPi4	4	0	USB2+3,HDMI, UART, I2C, SPI	483ms	46g	$50	24,150
RPi4 + Coral	4	4	USB2+3, UART, I2C, SPI	25ms	~100g	$110	2750
Jetson Nano	4	0.5	USB3, DP, HDMI, UART, I2C, SPI	25ms	250g	$100	2500
Jetson Xavier	6	21	USB3, DP, HDMI, UART, I2C, SPI	1.25ms	400g	$400	500

After these factors were taken into consideration, the Jetson Nano was chosen due to its good balance of performance, weight, and compactness. The closest contender was the RaspberryPi4 plus a Google Coral USB accelerator, but this setup would have been harder to hold together in a drone, since there are two separated components connected with a wire. The Jetson Xavier did offer more power, but the cost and weight were too large. The Raspberry Pi on its own offered dismal inference speed and was therefore not suitable.

6.2. Area computation experiment

To evaluate whether the algorithm described in section 4.3.1 estimates the visible camera area correctly, I set up a testing environment with a known physical area that I measured beforehand.

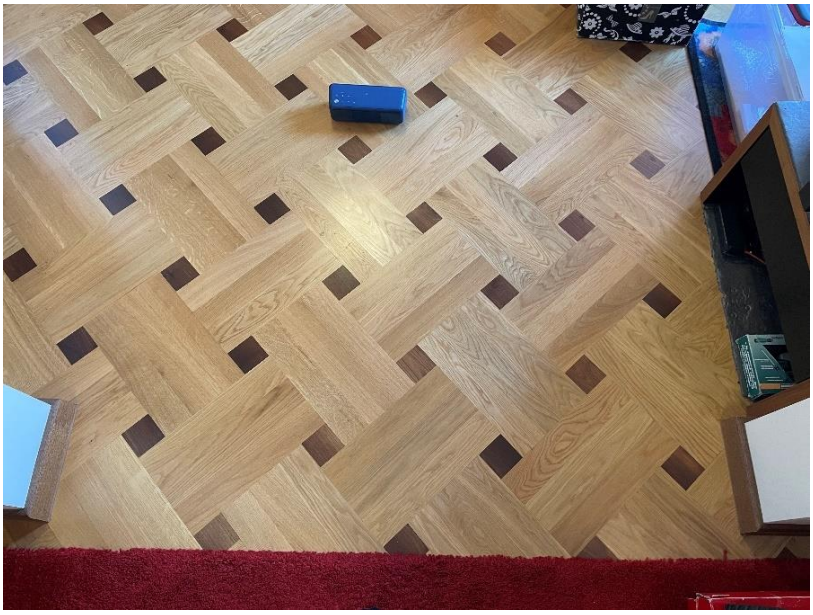

Figure 6.1 – Test setup for area computation

The above image shows a section of my room that I measured with measuring tape, giving me the following measurements:

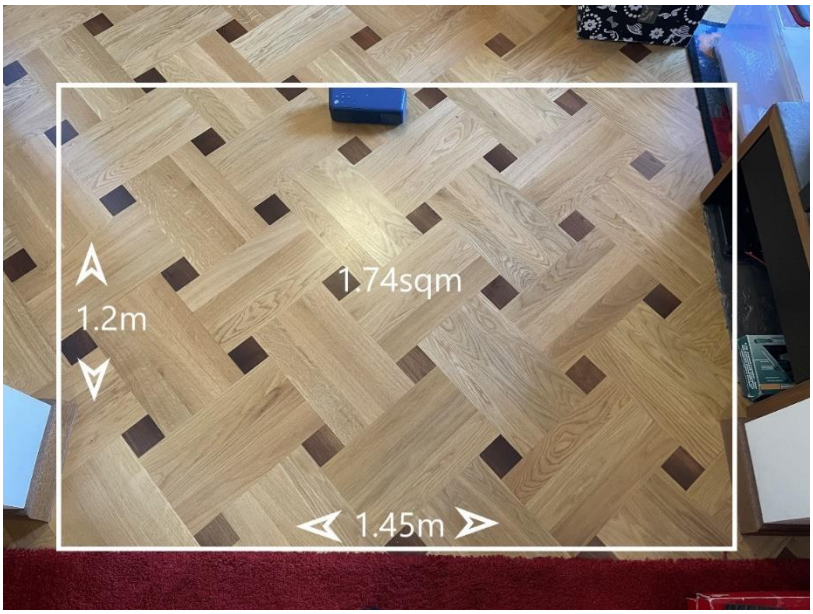

Figure 6.2 – Measured area

As can be seen from the above picture, the area measures 1.2 by 1.45m in height and width between the chosen markers. Multiplying these two values give a 1.74m^2 total area of the indicated zone.

Next, I held the drone at a height of 1m off the ground, with the camera pointing straight down (0 degrees, parallel to the ground) and ran the algorithm for finding the visible area of the camera, as described in section 4.3.1.

The view of the onboard drone camera is presented below:

Figure 6.3 – Onboard camera view

Given this image, the height of approximately 1m and a camera angle of 0 degrees (accurate due to gyroscope reading and servo stabilization), the algorithm estimated a total area of 1.7m^2, only about a 3% error from the actual area measured with the measuring tape. Even though this experiment is not highly scientific, we can however confirm that the intuition for the area computation is correct and that the algorithm produces a reasonably accurate estimate of the total area.

Extrapolating this information to an area of 300m^2, for example, the error would only be about 9m^2 using the same camera angle. It is important to note that since we do consider the area to always be of a rectangular shape, the error will be larger for camera angles above 0 degrees, since the actual projection of the camera image will be a trapezoid.

6.3. Image tiling experiment

In the next experiment I wanted to find the optimal image tiling of the input image, whose size is 1632x1232 pixels.

Knowing that the object detection model's input size is 300x300 (MobileNetV2), naturally it is clear to see that without tiling, the input image would be resized so much that a lot of the detail will be lost, leading to very bad results.

Therefore, the goal was to find the best NxM tiling of the input image that produces the most accurate result in the shortest time. N represents the horizontal tiles and M represents the number of vertical tiles. In order to make sense of this experiment, I considered the following parameters, besides N and M:

- **Tile Width**
  The pixel width of an individual tile

- **Tile Height**
  The pixel height of an individual tile

- **Runtime per tile [ms]**
  The time it took to run inference on a single image tile

- **Runtime per image [ms]**
  The total time it took to run inference on all the tiles and combine the results

- **Correctness [%]**
  The percentage of people detected from the total people in the scene. This is what we want to optimize, along with the total runtime.

I considered tilings that tile the image in tiles that are as square as possible, so as not to distort the input image. This means that there should be as close to a 4:3 relationship between N and M, since 4:3 is the aspect ratio of the input image. The tilings that were tested were the following:

Table 6.2 – Image tiling comparison

N	M	Tile Width	Tile Height	Runtime/Tile	Runtime/Image	Correctness
3	2	544	616	50ms	305ms	65%
4	3	408	410	54ms	650ms	67%
5	4	326	308	45ms	905ms	73%
6	4	272	308	43ms	1050ms	76%
8	6	204	205	44ms	2100ms	67%

As can be seen from the above results, a tiling of 6x4 resulted in the most correct results (the most people detected from those present) and therefore was the tiling selected for use.

6.4. Stationary tests

In this testing scenario, the drone was held stationary at a fixed height above the ground, while looking down at the people below, with the motors turned off.

This enables the use of the drone in crowded areas without worrying about the legal considerations that would have come up if the drone was being flown normally.

The input images will be presented, together with the detection output of both the simple person detection algorithm and the social distancing overlays. The contextual information, such as height, camera angle etc. will be shown, along with the .csv files where data was logged.

6.4.1. Setup

I chose the location of a shopping center for this test, due to high people concentration and possible overcrowding conditions appearing. I chose an outdoor and an indoor setting to compare different environmental conditions, and because that is where I found suitable places at an appropriate altitude.

I mounted the drone on a stand and started the detection as it would normally be started during a regular flight, to come as close as possible to real in-flight detection without actual flying. The detection was monitored on the dashboard application (screenshots in the following sections) and the terminal output of the ROS nodes. The results were visualized from images saved on the disk of the Jetson Nano's SD card via a network mount on the Ground Station Computer running Windows 10. This allows the detection results to be seen from the GS Computer, without having to connect the Jetson Nano to a separate display.

I chose two testing locations, one from a vantage point above the entrance to the mall and one in the adjacent park. The setups are shown below:

Figure 6.4 – Test setup in the park

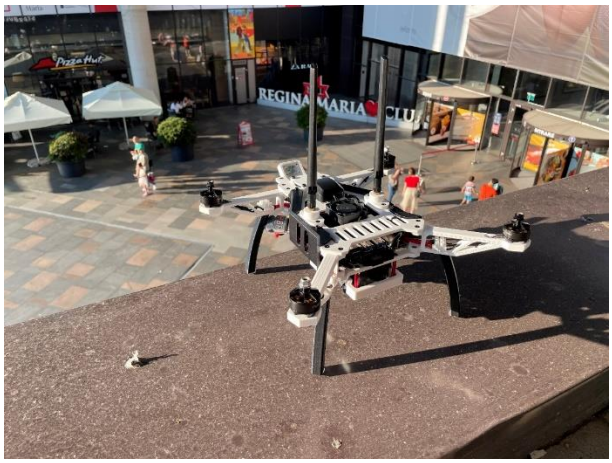

Figure 6.5 – Test setup at the entrance

In both scenarios the drone was stationary (not flying) and the detection was run normally. The only difference between these scenarios and regular use was the absence of fligh, which was the testing objective.

Flight tests will be presented in the next section.

6.4.2. Inputs

I selected four images from the multitude of recorded images, which can be seen below:

Figure 6.6 – Image 1
4m high, 60deg camera

Figure 6.7– Image 2
4m high, 60deg camera

Figure 6.8– Image 3
2.9m high, 60deg camera

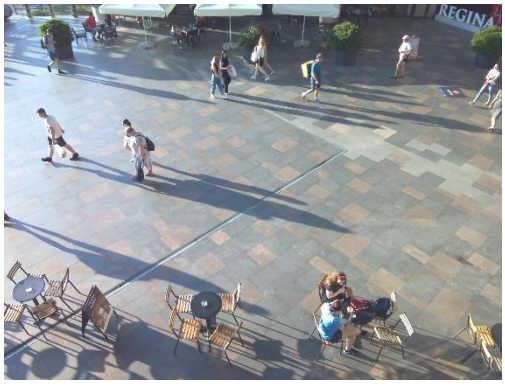

Figure 6.9– Image 4
5.5m high, 50deg camera

The images contain the reported relative height and the camera angle. These will be used in the social distancing and overcrowding estimations to compute the visible camera area.

6.4.3. Detections

All the detections used the 6x4 input image tiling, since this was proven to be the best tiling for the input image resolution, as discovered in section 6.3.

Each image has bounding boxes overlaid and shows the total number of people detected by the algorithm in comparison to the true total number of people in the frame, which were manually counted later. The percentage shows the number of people correctly identified by the algorithm to the total number of identifiable people. Unidentifiable people (partly visible, very small etc.) were not considered.

The detections on the input images are shown below, cropped to a smaller size for easier visualization:

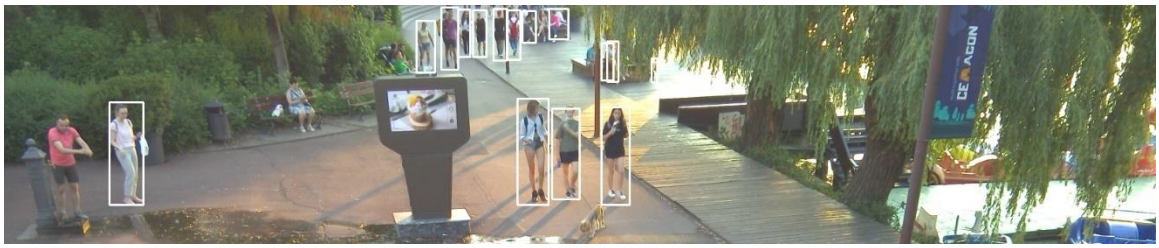

Figure 6.10 – Image 1 detections

21 people, 15 detected = 71.4% correct

We can see a large concentration of people in the back area, with a few people present in the front of the image. They were mostly detected correctly, except for the people sitting on the bench on the left side of the image.

Figure 6.11 – Image 2 detections

18 people, 10 detected = 55.5% correct

Almost the same image, but this time the woman sitting on the bench was correctly identified, and the girls in the middle have moved further away, which will be relevant when showing social distancing in the next section.

Figure 6.12 – Image 3 detections

16 people, 11 detected = 69.7% correct

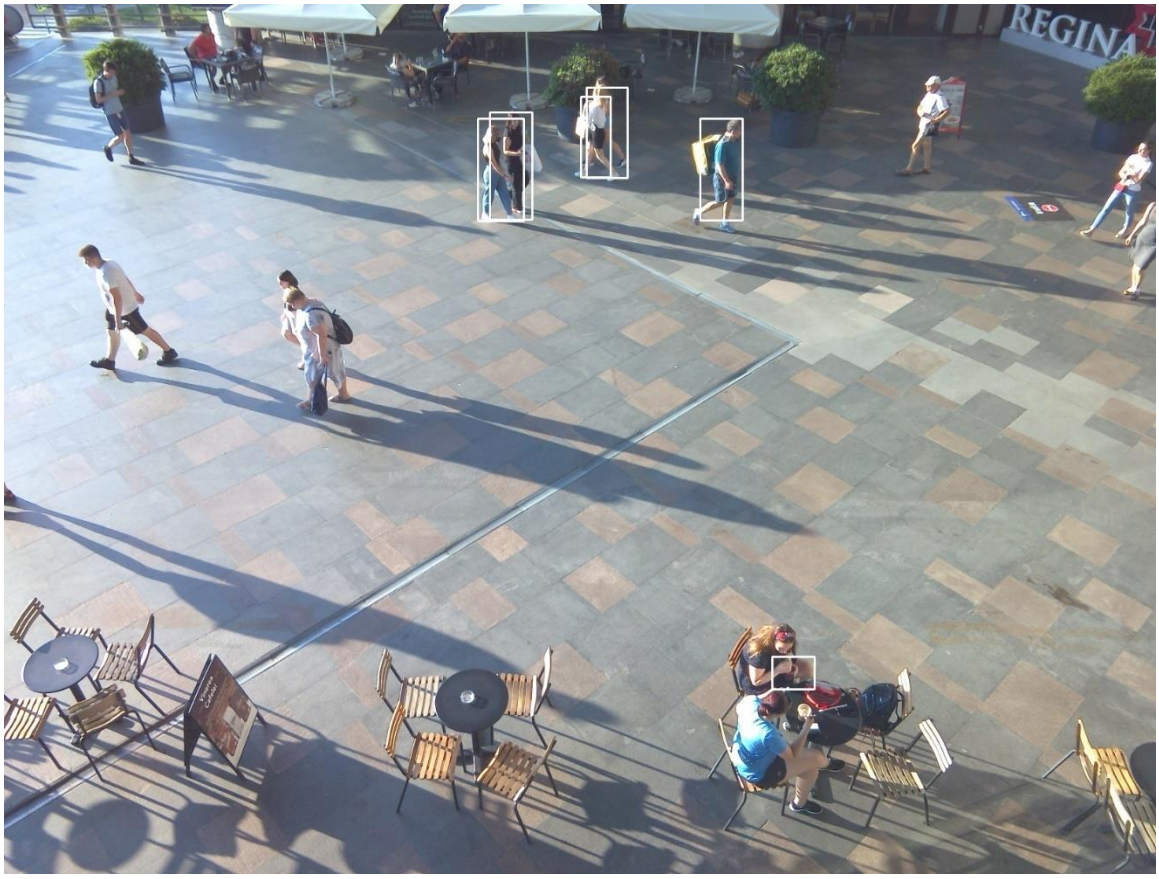

Figure 6.13 – Image 4 detections

13 people, 6 detected = 46.1% correct

Although the last imge had a lower corectness, I chose it because the detected people are relevant for showing the social distancing computations in the next section, and to show that the system produces lower success rates sometimes. The 4 women and the courier were correctly identified, as was part of the woman in the lowe section of the image.

6.4.4. Social distancing and overcrowding

Social distancing and overcrowding estimations used the height and camera angle to determine the visible camera area, the projected pixel size and the, as explained in Chapter 4. The captions indicate the number of people standing too close to each other according to a 1.5m social distance, and also indicate the average density of the crowd, which is calculated according to the algorithms described in Chapter 4 and Chapter 5.

The updated red bounding boxes were drawn over the previous images and the results are presented below:

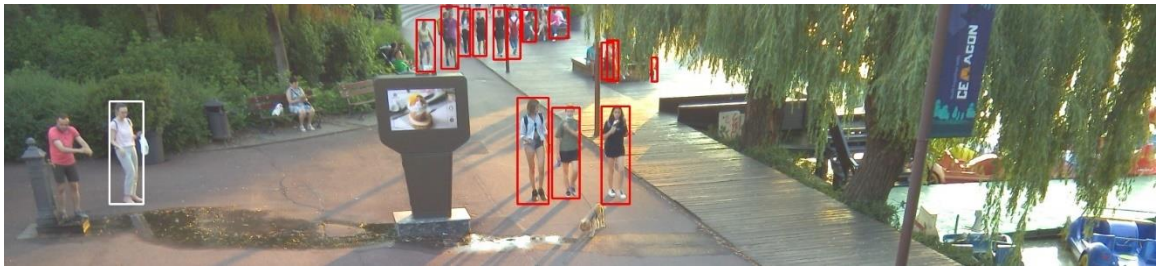

Figure 6.14 – Image 1 social distancing and overcrowding

15 people detected, 14 too close, 0.83 people/10m^2 density

Figure 6.15 – Image 2 social distancing and overcrowding

10 people detected, 7 too close, 0.47 people/10m^2 density

It is interesting to note that between Image 1 and Image 2, the 3 girls in the middle are standing too close in Image 1, but the one on the left goes furter away in Image 2, passing the 1.5m threshold and, therefore, is no longer close to the middle girl. The middle girl was detected too much to the right, which contributed to this result.

Figure 6.16 – Image 3 social distancing and overcrowding

11 people detected, 9 too close, 0.99 people/10m^2 density

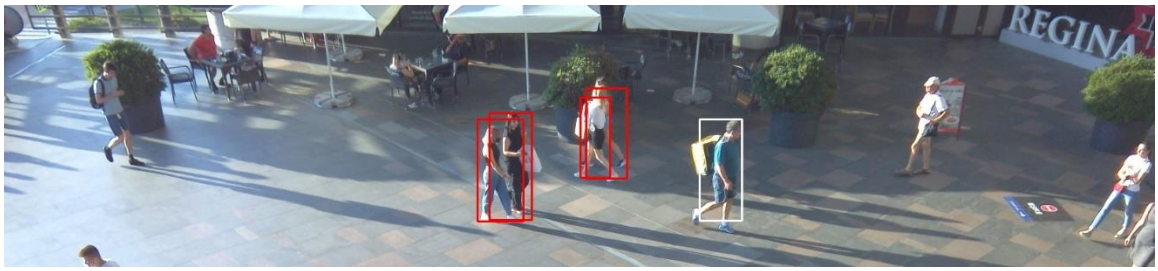

Figure 6.17 – Image 4 social distancing and overcrowding

6 people detected, 4 too close, 0.43 people/10m^2 density

Here it can be seen that the courier is at a distance larger than 1.5m from the other detected persons, but the two pairs of girls are standing closer than 1.5m from each other. This means that the social distancing estimation method is working correctly, as far as these tests went.

Below is a screenshot of the live dashboard application while running some detections same as seen above:

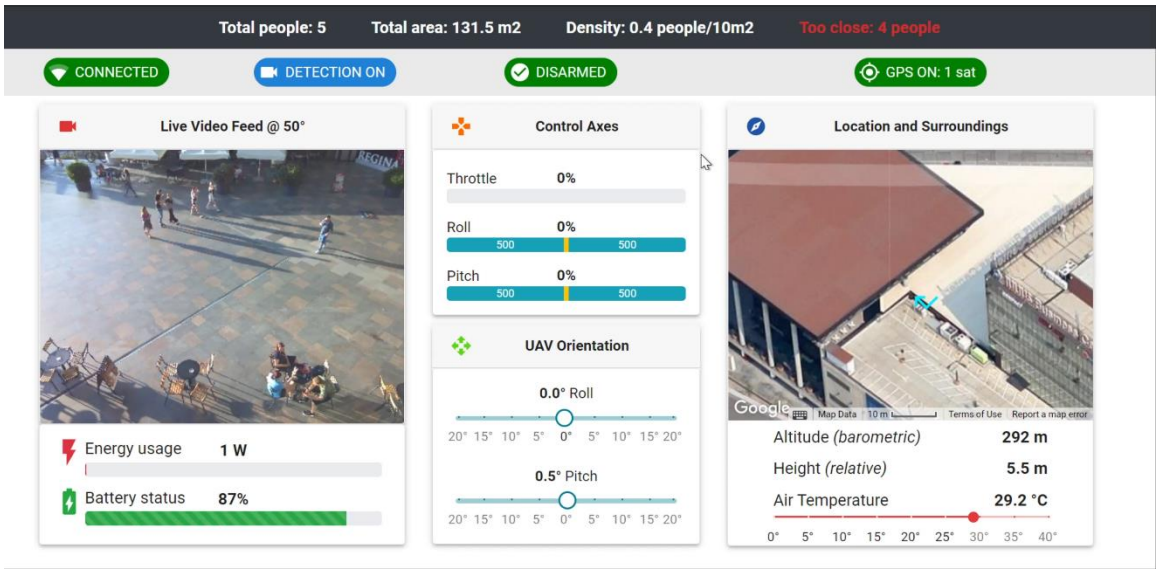

Figure 6.18 – Live dashboard during the tests

Since the drone was not being flown, the middle column with the control axes was irrelevant in this case. The other columns did, however, provide useful real-time information, along with the top status bars.

6.4.5. Contextual information

The logged data of the prevoius four used images is presented below, extracted from the .csv file:

Table 6.3 – Data log of performed tests

time	stamp	people	warning	density	latitude	longitude	sat	altitude	height	temp	angle	area
19:45:52	1624898753	6	4	0.46	46.7715	23.6261	1	289.3	5.5	27.3	50	131.5
20:47:55	1624902475	11	9	0.99	46.7728	23.6265	1	279.2	2.9	27.1	60	111.6
20:48:13	1624902493	15	14	0.83	46.7728	23.6265	1	279.1	3.7	27.2	60	181.7
20:48:14	1624902494	10	7	0.47	46.7728	23.6265	1	279.4	4	27.2	60	212.3

6.4.6. Statistics

The recorded data was post-processed to produce the following graphs, showing samples of the evolution of the crowding of the people in the scenes.

First of all, I charted the denstity and number of people against each other in the two locations:

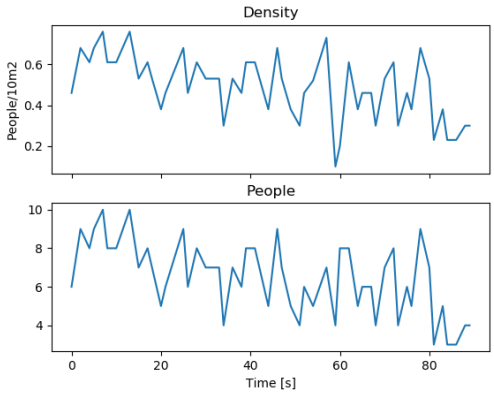

Figure 6.19

Entrance people and density sample

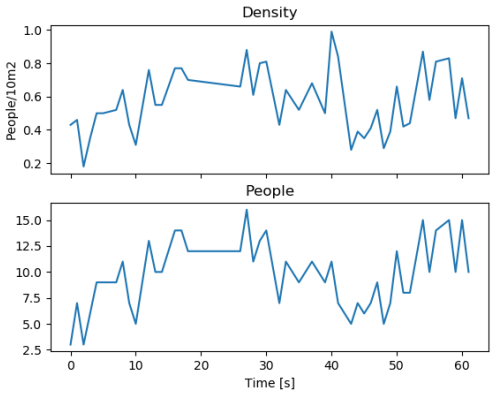

Figure 6.20

Park people and density sample

As can be seen from the above charts, the entrance to the mall was generally seeing a slight decline, as the tests were performed at around 8PM in the evening, when people usually leave the mall. In the park the trend is less clear. It appears to be fluctuating, due to the fact that people come and go all the time, which also happens at the entrance.

Second, I looked at how socially distanced people were, and tried to find a correlation between this and the average recorded density. The charts are presented below:

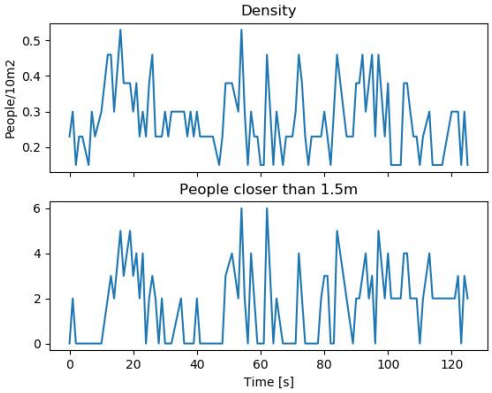

Figure 6.21

Entrance density and social distancing sample

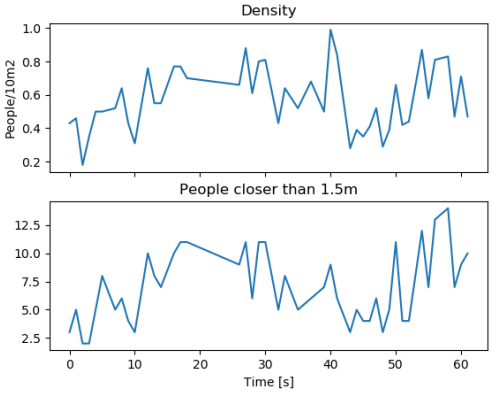

Figure 6.22

Park density and social distancing sample

I selected different time samples for these tests to explore different parts of the created dataset.

It can be seen that density usually relates closely to the number of people who are not socially distanced, although this could be further analyzed using more advanced statistical methods, that are out of the scope of this project.

6.5. Deployment tests

In this testing scenario, the drone was flown in an open area in a park. The drone was piloted by myself and pointed at the ground at various camera angles, while detection was turned on and the results were being recorded.

This is the most relevant testing scenario, since the drone is actually flown, not just stationary, like in the previous test.

6.5.1. Setup

As stated, the test took place at a park in Cluj-Napoca. It was about 20 in the afternoon, with still a lot of light left. The park was very crowded, so we chose an empty field at the end of the park to launch the drone:

Figure 6.23 – Deployment test view 1

Figure 6.24 – Deployment test view 2

In the above images, the drone can be seen hovering over the chosen test location. This location was selected for safety reasons, due to the fact that it was far from people and objects that could be damaged.

The people we were trying to identify were in the following locations, as seen from above:

Figure 6.25 – Test area overview 1

Figure 6.26 – Test area overview 2

6.5.2. Inputs

I selected a few representative images from the recorded ones, seen below:

Figure 6.27 – Image 1
3.2m high, 65deg camera

Figure 6.28– Image 2
7.7m high, 60deg camera

Figure 6.29– Image 3
7.6m high, 60deg camera

Figure 6.30– Image 4
2.1m high, 60deg camera

6.5.3. Detections

All the detections used the 6x4 input image tiling, since this was proven to be the best tiling for the input image resolution, as discovered in section 6.3.

Each image has bounding boxes overlaid and shows the total number of people detected by the algorithm in comparison to the true total number of people in the frame, which were manually counted later. The percentage shows the number of people correctly identified by the algorithm to the total number of identifiable people. Unidentifiable people (partly visible, very small etc.) were not considered.

The detections on the input images are shown below, cropped to a smaller size for easier visualization:

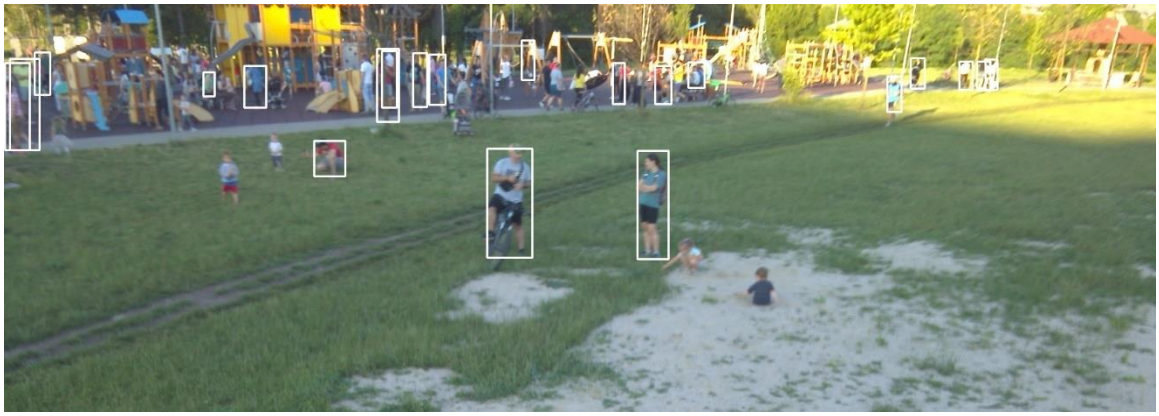

Figure 6.31 – Image 1 detections

32 people, 21 detected = 65.6% correct

We see a fairly even distribution of people accross the back side of the image. Due to the vibration of the craft, however, the image does become less sharp than the ones from the stationary tests, which will also be visible in the other images.

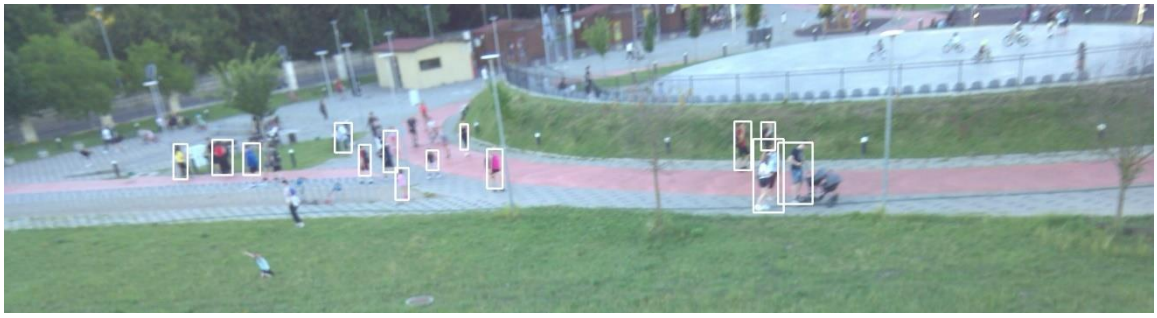

Figure 6.32 – Image 2 detections

23 people, 14 detected = 60.8% correct

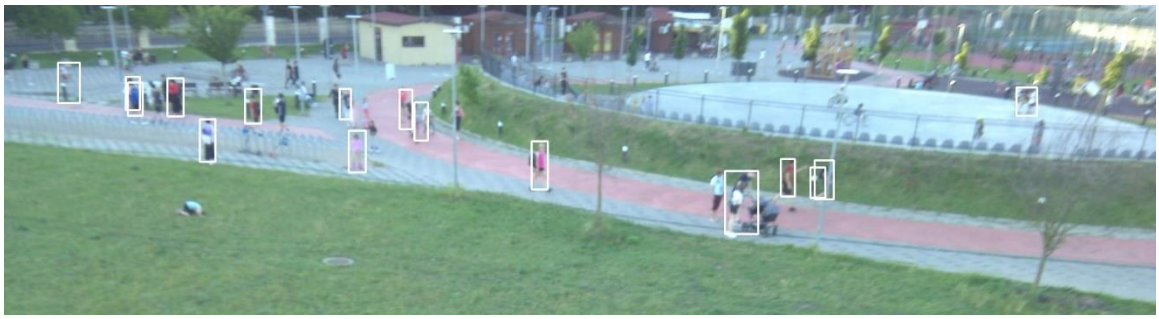

Figure 6.33 – Image 3 detections

29 people, 16 detected = 53.3% correct

Figure 6.34 – Image 4 detections

34 people, 27 detected = 79.4% correct

Here we see again a large concentration of people far away, with an impressive detection accuracy.

6.5.4. Social distancing and overcrowding

Social distancing and overcrowding estimations used the height and camera angle to determine the visible camera area, the projected pixel size and the, as explained in Chapter 4. The captions indicate the number of people standing too close to each other according to a 1.5m social distance, and also indicate the average density of the crowd, which is calculated according to the algorithms described in Chapter 4 and Chapter 5.

The updated red bounding boxes were drawn over the previous images and the results are presented below:

Figure 6.35 - Image 1 social distancing and overcrowding

21 people detected, 6 too close, 0.16 people/10m^2 density

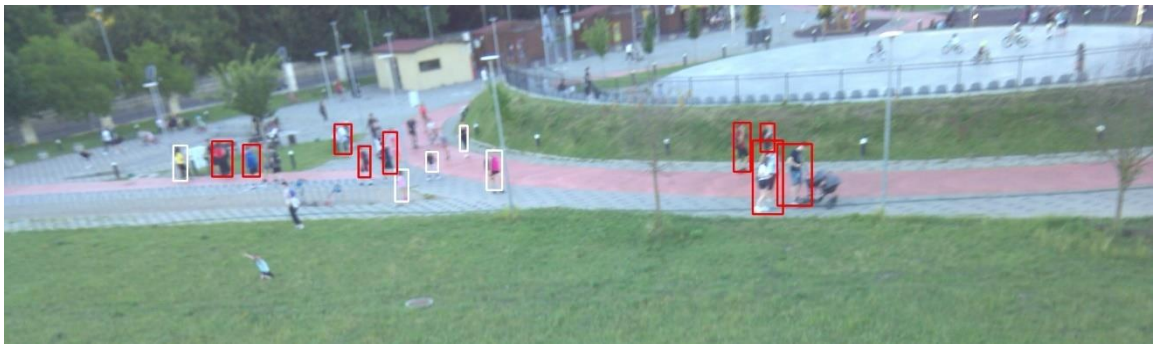

Figure 6.36 – Image 2 social distancing and overcrowding

14 people detected, 9 too close, 0.18 people/10m^2 density

Figure 6.37 – Image 3 social distancing and overcrowding

16 people detected, 7 too close, 0.24 people/10m^2 density

Figure 6.38 – Image 4 social distancing and overcrowding

27 people detected, 6 too close, 0.47 people/10m^2 density

I have also attached a screenshot of the dashboard application while running the above detections:

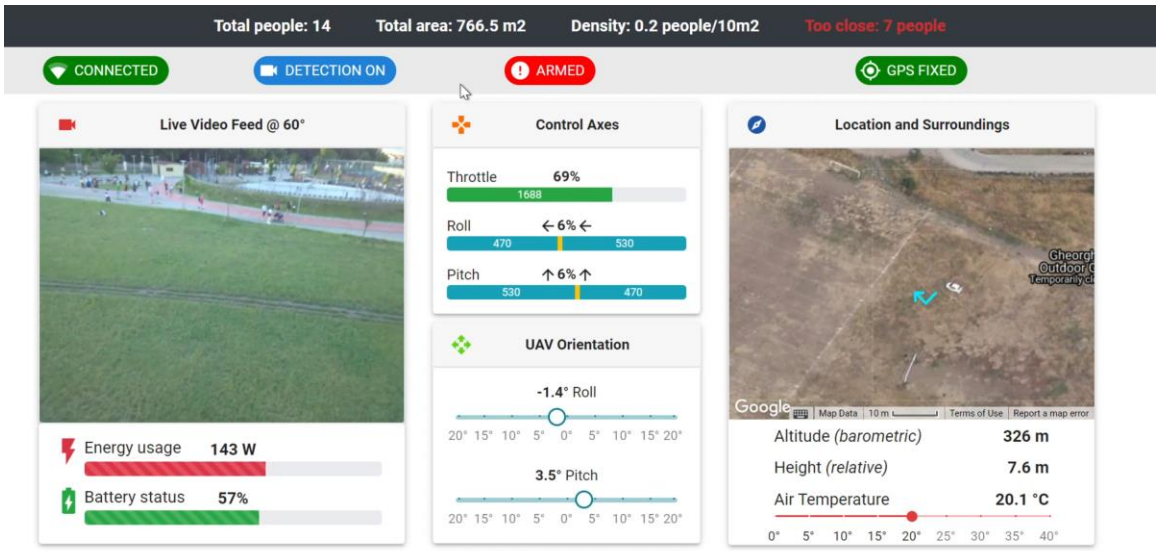

Figure 6.39 – Live dashboard while running the tests

It can be seen that, compared to the static tests, in these tests the control axes are active for flying, and power is drawn from the battery (energy usage parameter in the lower left). The Roll and Pitch axes of the quadcopter are also indicating the current orientation of the craft in the air, and the height is computed at 7.6m in the screenshot. GPS information is also accurate, showing the location of the craft on the field in the sporting park.

6.5.5. Contextual information

The logged data of the prevoius four used images is presented below, extracted from the .csv file:

Table 6.4 – Data log of performed tests

time	stamp	people	warning	density	latitude	longitude	sat	altitude	height	temp	angle	area
19:51:13	1625503874	27	6	0.47	46.76818	23.63239	1	326.8	2.1	24.9	65	577.2
20:15:31	1625505331	21	6	0.16	46.76858	23.6325	1	321.7	3.2	19.9	65	1340.2
20:13:49	1625505230	16	7	0.24	46.76857	23.63234	1	325.9	7.1	20.1	60	669
20:13:46	1625505227	14	9	0.18	46.76859	23.63239	1	326.2	7.7	20.2	60	786.8

6.5.6. Statistics

The recorded data was post-processed to produce the following graphs, showing samples of the evolution of the crowding of the people in the scenes.

I charted the following graphs:

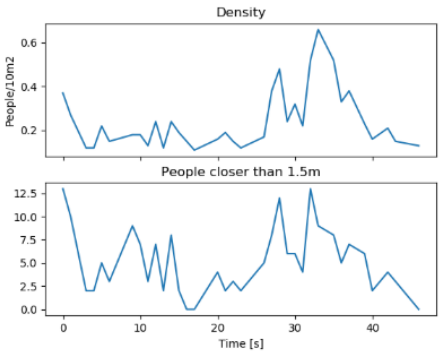

Figure 6.40

Density and social distancing sample

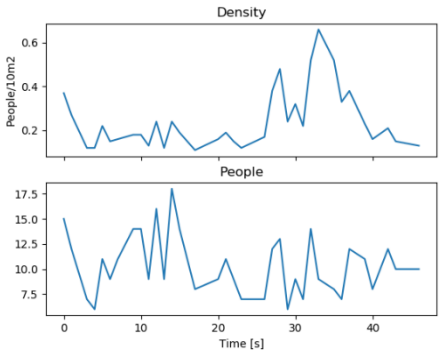

Figure 6.41

Density and total people sample

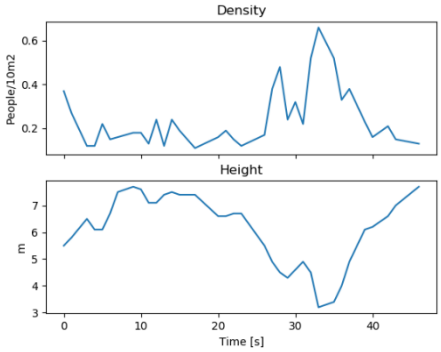

Figure 6.42

Density and craft height

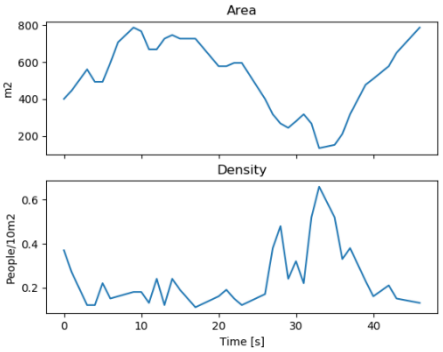

Figure 6.43

Camera area and density

The first two figures show a similar story to the static tests, while the second ones could not be generated from the static tests, due to the fact that the height was fixed and the drone was not flying.

Figure 6.42 shows that as the height of the craft increases, the average density of the crowd decreases, due to the fact that more area is visible to the camera, so the total area is larger. Since height and area are directly connected, the last figure (6.43) shows a similar story, where a decreasing area gives a higher average crowd density, assuming the same number of total people.

All computations and results were processed on the Jetson Nano.

Chapter 7. User's manual

This chapter will present the steps that a system adminstrator should take to install the project's applications and run them successfully.

7.1. Prerequisites

The following items are required to run the project:

- Jetson Nano developer kit with a Wi-Fi module
- Modern laptop computer
- Gamepad controller
- Drone

7.2. Dependencies

In order to install all the dependencies that the project needs, the following packages will be required:

7.2.1. Jetson Nano

First, install system dependencies:
sudo apt-get install git cmake
sudo apt-get install libatlas-base-dev gfortran
sudo apt-get install libhdf5-serial-dev hdf5-tools
sudo apt-get install python3-dev
sudo apt-get install nano locate
sudo apt-get install libfreetype6-dev python3-setuptools
sudo apt-get install protobuf-compiler libprotobuf-dev openssl
sudo apt-get install libssl-dev libcurl4-openssl-dev
sudo apt-get install cython3

Frameworks and larger libraries:

- ROS Melodic
  Installation instructions can be found here[17]

- OpenCV
  Needs to be built from source. The easiest way is to use the JetsonHacks repo and installer[18]:

  git clone https://github.com/JetsonHacksNano/buildOpenCV
  cd buildOpenCV
  ./buildOpenCV.sh |& tee openCV_build.log

[17] http://wiki.ros.org/melodic/Installation/Ubuntu
[18] https://www.jetsonhacks.com/2019/11/22/opencv-4-cuda-on-jetson-nano/

- Flask server
  pip install Flask

- YAMSPy library
  git clone https://github.com/ricardodeazambuja/YAMSPy.git
  cd YAMSPy
  sudo pip3 install .
  sudo usermod -a -G dialout $USER

7.2.2. Laptop

For ReactJS you need to install NodeJS and NPM from here[19] and here[20].
Python3 is also required and can be downloaded from here[21].

7.3. Downloading the applications

The entire project can be found at *https://github.com/andreirusu99/mUAV* and is currently private. If anyone needs access, I will provide it. I will probably make it open source in the future for others who want to make similar application to be able to learn from it. Run this command on both the laptop and the Jetson Nano:

git clone https://github.com/andreirusu99/mUAV

The repo contains five folders, but out of these, only three will be required:

- *GroundStation*
  Contains the Python script used for controlling the drone via UDP. Only needed on the laptop.

- *ros_client*
  Contains the React app with the real-time dashboard. Only needed on the laptop.

- *catkin_ws*
  The catkin workspace (ROS build tool) where the Python ROS code resides. Only used on the Jetson Nano.

7.4. Running the applications

7.4.1. ROS Application

The ROS applciation must first be built by running *catkin make* in the terminal in the root of the *catkin_ws* folder.

[19] https://nodejs.org/en/download/
[20] https://docs.npmjs.com/downloading-and-installing-node-js-and-npm
[21] https://www.python.org/downloads/windows/

After the build is done, the ROS application can be started with the following command:

roslaunch drone auto.launch
which uses the lauch file defined above to start all the needed processes.

7.4.2. React Client Application

The client applcation can be started by running *npm start* in the root of the *ros-client* folder. This will open a browser window which will be populated with the data coming from the previously opened ROS application.

7.4.3. Control Application

Finally, the control application can be started by running the following command from the *GroundStation* folder:

venv/Scripts/python3.exe JoyControlStation.py

This command will start sending UDP messages to the address of the Jetson Nano's Wi-Fi card, which has to be previously configured in the Python code.

To find the IP address of the Jetson Nano, type *ifconfig* in the terminal and take the IP address of *wlan0*. The Jetson Nano must be connected to the hotspot created by the laptop for this to work.

7.5. Piloting and using the drone

After the above steps are done, the gamepad attached to the laptop can be used to control the drone.

The two sticks control the axes of the drone, the A and B buttons control the detection, and the two shoulder buttons control the yaw axis. Arming the drone is done by pressing the right trigger with the throttle below 30% value. Disarming is done the same way.

Chapter 8. Conclusions

This bachelor's thesis proposed and implemented an innovative system for assessing overcrowding and the respecting of social distancing in public areas, using unmanned aerial vehicles.

The implemented solution used a custom-made 3D printed quadcopter fitted with a Jetson Nano computer, a camera and various sensors, which together enable the system to identify people in the ground, asses how close they are to each other and how crowded they are. The drone is a mobile deployable system that uses a ground station computer to allow the drone operator to see craft parameters and detection results in real time while flying. Additionally, the craft continuously records the most relevant data while flying and allows statistics to be generated post-flight, which allows the administrators of the system to have a clear understanding of how the crowd evolved in time.

8.1. Contributions

I would summarize my contributions as follows:

- Improved the design of 3D printable drones with a robust and space-efficient design that allows multiple components to fit together well
- Created a multi-processor application that uses the modern ROS framework to achieve inter-process communication and perform the tasks of person detection, overcrowding and social distancing estimations
- Created a unique end-to-end system with a dashboard and control application for viewing real-time data while flying in the desired environment and controlling the drone with a regular gamepad, removing the need to buy expensive radio control transmitters and receivers
- Implemented a real-time edge system that removes the need for offsite data processing and decreases latency considerably compared to traditional systems

8.2. Critical analysis of results

In order to create a system that would allow the identification and monitoring of crowds on edge in real time, I realized that a traditional approach using stationary cameras or other devices was not sufficient, so I set out to create an innovative and mobile deployable system that performs this task faster and more conveniently than just record some footage and send it somewhere else for processing.

While this has been achieved, there are some limitations to the system. First of all, the most important drawback is the relatively low detection accuracy. Since the used model is a person detection model, it is not optimal for use with crowd detection of large crowds, since the people are very small and most of the times partially or completely occluded by

others. A dedicated crowd detection algorithm such as the ones from VisDrone would have been necessary, but it would have to be optimized and minimized for edge devices.

In addition to this drawback, there is also the limitation of the control of the drone while in flight. A skilled operator can hover the drone very steady, and the camera is stabilized on one axis, but the drone is still very hard to control compared to a more traditional drone from DJI, for example, which stabilizes itself and requires very little operator input. This also improves the quality of aerial images, which become more clear than the ones obtained from this project.

8.3. Further improvements

Since the domain of UAV-based computation and processing is so broad, and so is the field of crowd analysis, there are many improvements that could be made to this system. I summarized some of them with the following list:

- Implement flight automation and autopilot for stabilized flight. This would require more processing power, so probably an upgraded main computer could be helpful as well

- Implement heat maps visualizations of the crowd which would allow a very quick understanding of how the crowd is structured

- Improve the camera area computation by taking into account the deformation of the camera area to a trapezoidal shape instead of a rectangular shape that is considered now

- Implement a dynamic tiling algorithm that would choose the tiling performed on the input image based on the height of the craft and other factors, such as the tilt of the camera

Bibliography

[1] J. Ren, Y. Pan, A. Goscinski and R. Beyah, "Edge Computing for the Internet of Things," *IEEE Network*, vol. 32, no. 1, pp. 6-7, 2018.

[2] S. Weisgon, C. Jie, z. Quan, L. Youhuizi and X. Lanuy, "Edge Computing: Review and Future Directions," *IEEE INTERNET OF THINGS JOURNAL*, vol. 3, no. 5, pp. 637-646, 2016.

[3] e. a. Tsung-Yi Lin, "Microsoft COCO: Common Objects in Context," 2015.

[4] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 2005.

[5] J. W. Davis and V. Sharma, "Background-subtraction using contour-based fusion of thermal and visible imagery," *Computer Vision and Image Understanding*, vol. 106, no. 2, pp. 162-182, 2007.

[6] L. Xia, C. C. Chen and J. K. Aggarwal, "Human detection using depth information by kinect," *Conference on Computer Vision and Pattern Recognition Workshop*, pp. 15-22, 2011.

[7] L. Wei, A. Dragomir, Dumitru Erhan, S. Christian, R. Scott, F. Cheng-Yang and B. Alexander, "SSD: Single Shot MultiBox Detector," in *European Conference on Computer Vision*, 2016.

[8] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2016.

[9] A. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto and A. Hartwig, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision," 2017.

[10] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreeto and A. Hartwig, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," 2017.

[11] A. Marana, M. Cavenaghi, R. Ulson and F. Drumond, "Real-Time Crowd Density Estimation Using Images".

[12] J. C. S. J. Jacques, S. R. Musse and C. R. Jung, "Crowd analysis using computer vision techniques," *IEEE Signal Processing Magazine*, vol. 27, no. 5, pp. 66-77, 2010.

[13] Z. Pengfei, W. Longyin, B. Xiao, L. Haibing and H. Qinghua, "Vision Meets Drones: A Challenge," *arXiv preprint arXiv:1804.07437*, 2018.

[14] D. Du, L. Wen, Z. Pengfei and F. Heng, "VisDrone-CC2020: The Vision Meets Drone Crowd Counting Challenge Results," *Computer Vision – ECCV 2020 Workshops*, vol. 12538, 2020.

[15] N. Tijtgat, W. van Ranst, B. Volckaert, T. Goedeme and F. de Turck, "Embedded Real-Time Object Detection for a UAV Warning System," in *International Conference on Computer Vision*, 2017.

[16] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and C. Liang-Chieh, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* pp. 4510-4520, 2018.

[17] O. Unel, B. Ozkalaycı and C. Cıgla, "The Power of Tiling for Small Object Detection," in *Conference on Computer Vision and Pattern Recognition*, 2019.

[18] G. Plastiras, C. Kyrkou and T. Theocharides, "Efficient ConvNet-based Object Detection for Unmanned Aerial Vehicles by Selective Tile Processing," in *Conference on Computer Vision and Pattern Recognition*, 2019.

[19] "Viewing Frustum - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Viewing_frustum.

[20] "Global Positioning System," [Online]. Available: https://en.wikipedia.org/wiki/Global_Positioning_System.

[21] "Altimeters," [Online]. Available: https://en.wikipedia.org/wiki/Altimeter.

[22] "Bosch BMP280 datasheet," [Online]. Available: https://www.bosch-sensortec.com/products/environmental-sensors/pressure-sensors/bmp280.

[23] "Barometric Formula," [Online]. Available: https://en.wikipedia.org/wiki/Barometric_formula.

[24] "Digital Image Scaling," [Online]. Available: https://en.wikipedia.org/wiki/Image_scaling.

[25] B. Theys, G. Dimitriadis, P. Hendrick and J. De Schutter, "Influence of propeller configuration on propulsion system efficiency of multi-rotor Unmanned Aerial Vehicles," in *International Conference on Unmanned Aircraft Systems (ICUAS)*, 2016.

[26] "Jetson Nano developer kit specifications," NVidia, [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[27] J. W. Sensinger, S. D. Clark and J. F. Schorsch, "Exterior vs. interior rotors in robotic brushless motors," in *IEEE International Conference on Robotics and Automation*, 2011.

[28] "ROS Official Documentation," [Online]. Available: http://wiki.ros.org/Documentation.

[29] "OpenCV Official Documentation," [Online]. Available: https://docs.opencv.org/master/d6/d00/tutorial_py_root.html.

[30] "Flask Official Documentation," [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/.

[31] "ReactJS Official Documentation," [Online]. Available: https://reactjs.org/docs/getting-started.html.

[32] "ROSLib," [Online]. Available: http://wiki.ros.org/roslib.

[33] "Google Maps API Documentation," [Online]. Available: https://developers.google.com/maps/documentation.

[34] R. De Azambuja, "YAMSPY Python MSP library," [Online]. Available: https://github.com/thecognifly/YAMSPy.

[35] A. Vargas, "DronePilot," 2016. [Online]. Available: https://github.com/alduxvm/DronePilot.

[36] Twisted Matrix Labs, "Twisted Python UDP library," [Online]. Available: https://twistedmatrix.com/documents/15.1.0/core/howto/udp.html.

[37] D. Linden and T. B. Reddy, "Chapter 35, Lithium-Ion Batteries," in *Handbook of Batteries, Third Edition*, McGraw-Hill, 2001, pp. 1074-1167.

[38] "Flask Python Server," [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/.

[39] "Gstreamer," [Online]. Available: https://gstreamer.freedesktop.org/.

[40] F. Kesim, "BMP280 Library," [Online]. Available: https://github.com/feyzikesim/bmp280.

[41] "NMEA specification," [Online]. Available: https://en.wikipedia.org/wiki/NMEA_0183.

[42] n. Corporation, "Jetson Inference Library," [Online]. Available: https://github.com/dusty-nv/jetson-inference.

[43] L. Tsung-Yi, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollar and L. C. Zitnick, "Microsoft COCO: Common objects in context," *ECCV*, 2014.

[44] "User Datagram Protocol - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/User_Datagram_Protocol.

[45] "Transport Control Protocol - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Transmission_Control_Protocol.

[46] "MultiWii Serial Protocol," [Online]. Available: http://www.multiwii.com/forum/viewtopic.php?p=11836.

[47] Open Robotics Foundation, "Rosbridge Server," [Online]. Available: http://wiki.ros.org/rosbridge_server.

[48] Open Robotics Foundation, "Roslaunch," [Online]. Available: http://wiki.ros.org/roslaunch.

[49] "RaspberryPi 4 Model B Technical Specifications," [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/.

[50] "Google Coral Datasheet," Google, [Online]. Available: https://coral.ai/docs/accelerator/datasheet/.

[51] "Jetson Xavier NX specifications," NVidia, [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/.

[52] " NMEA message formats," [Online]. Available: https://sites.google.com/site/vmacgpsgsm/understanding-nmea.