POLITECNICO DI TORINO

Master's Degree in Computer engineering

Master's Degree Thesis

SOFTWARE DESIGN FOR UAV APPLICATIONS IN GNSS-DENIED ENVIRONMENTS

Supervisors

Prof. Elisa CAPELLO

Dott. Francesco SOTTILE

Candidate

Giacomo CACIAGLI

July 2025

Summary

Today, unmanned aerial vehicles (UAVs) are gaining more and more interest, their velocity and precision make them a powerful tool for a wide range of applications, from aerial photography to search and rescue operations. These systems heavily rely on the global navigation satellite system (GNSS) as primary source of position information for navigation. Without reliable satellite signals, a UAV's ability to maintain accurate positioning and autonomy is compromised, for this reason the UAVs are not suitable to work in environments where GNSS signals are either weak or unavailable, so-called GNSS-denied environments. However, there are many operations, such as indoor and urban inspections, that take place in GNSS-denied environments that would benefit from the use of a drone.

The design and development of UAV software capable of enabling reliable navigation in the absence of GNSS signals has become an area of intense research. Currently, most of the solutions to this problem leverage alternative sensor modalities such as visual odometry, LiDAR, inertial measurement units (IMUs) and barometric pressure sensors. These modalities, in conjunction with advanced estimation algorithms, ensure autonomous flight. Another possible solution is to triangulate the position of the UAV using radio signals to measure the distance from known points, but the estimate could be inaccurate due to obstacles in GNSS-denied environments. The ultra-wideband (UWB) technology is an innovative technology with low power consumption that implements a communication defined by a wide frequency spectrum, this characteristic offers high precision and robustness against any type of interference, making it ideal for indoor and GNSS-denied environments.

This thesis focuses on the design and development of an application to enable UAVs to operate in GNSS-denied environments. The architecture uses ROS2 humble, PX4 (v 1.15.2 dev) and the DDS-XRCE protocol to create the necessary architecture to enable and control the flight of the drone. It was developed with the help of the LINKS foundation for a specific hardware setup, for this reason some parts of the software are strictly hardware dependent. After the development, the software has been tested in two different environments that differ in the localization system used: the first leverages a MoCap system to obtain position information, while the second uses the UWB technology to obtain those data. The first scenario

was used mainly to test the correctness of the features of the application and enhance the performance of the UAV, while the second was used to check the performance of the system in real-world scenarios.

Acknowledgements

I would like to express my sincere gratitude to my supervisors Elisa Capello and Francesco Sottile for the opportunity they have given me. I am also very grateful to the supervisors and my colleague Delos Campos of LINKS for their invaluable support and for making this work such an enjoyable experience.

I am deeply grateful to my family for their patience and unwavering support throughout this journey.

My thanks also go to the Giochini 2.0 group and all my Automation colleagues for the joys and hardships we shared, both during lectures and beyond.

I would also like to thank the Cirotroni, the Giochini 2.0 group, my high school friends and Miriam for their help, the wonderful experiences and the lasting memories we have created during this journey.

Thank you all for making this experience truly unforgettable.

Table of Contents

List of Tables

List of Figures

Acronyms

API
Application Programming Iterface

CPU
Central Processing Unit

DDS
Data Distribution Service

EKF
Extended Kalman Filter

FCC
Federal Communication Commission

GNSS
Global Navigation Satellite System

GPS
Global Positioning System

GUI
Graphical User Interface

ID
Identification

IEEE
Institute of Electrical and Electronics Engineers

MoCap

Motion Capture

OSRF

Open Source Robotics Foundation

PID

Proportional-Integral-Derivative

QoS

Quality of Service

RC

Radio Control

ROS

Robot Operating System

RTOS

Real-Time Operating Systems

SDK

Software Development Kit

ToA

Time of Arrival

ToF

Time of Flight

TWR

Two Way Ranging

UART

Universal Asynchronous Receiver-Transmitter

UAV

Unmanned Aerial Vehicle

UDP

User Datagram Protocol

UWB

Ultra-Wideband

UTS

UNIX Timesharing System Namespace

XRCE

eXtremely Resource Constrained Environments

uORB

Micro Object Request Broker

Chapter 1

Introduction

Unmanned Aerial Vehicles (UAVs), commonly known as drones, have witnessed an increase in popularity across numerous sectors due to their high agility, precision and ease of deployment. These capabilities have made UAVs indispensable tools in diverse applications, ranging from aerial photography and infrastructure inspection to agricultural monitoring and search-and-rescue missions. The increase in the usage of UAV technology continues to redefine the boundaries of what is possible in automated aerial operations. However, one of the most critical dependencies of UAV functionality is the Global Navigation Satellite System (GNSS), which serves as the primary source of positioning information for autonomous flight and navigation.

Despite the reliability and global coverage that GNSS provides, its effectiveness is severely reduced in environments where satellite signals are weak, obstructed or entirely unavailable, these environments are the so-called GNSS-denied environments. These conditions can be found in many common places such as urban canyons, underground locations, heavily forested regions and indoor areas. In such environments, the loss or degradation of GNSS signals can make conventional UAV systems incapable of maintaining accurate localization and trajectory control, compromising their operational utility, its safety and the safety of its surroundings.

However, many high-impact use cases for UAVs are precisely in those GNSS-denied environments. For example, indoor inspections of warehouses, tunnels, or manufacturing facilities are ideal tasks for drones due to their ability to access hazardous and confined spaces with ease and precision. Similarly, in urban environments where tall buildings often block or reflect satellite signals, UAVs could be instruments for surveillance, maintenance or emergency response missions. Therefore, overcoming the limitations imposed by the dependency of GNSS is a key step in expanding the operational envelope of UAVs.

Addressing this challenge has become a focal point of contemporary research on UAV autonomy. A promising strategy involves integrating alternative sensor

modalities to replace or enhance the GNSS-based navigation. Visual odometry, LiDAR, barometric sensors and Inertial Measurement Units (IMUs) are the most popular options. These sensors enable the UAV to extract its position relative to the environment through a combination of data fusion and advanced estimation algorithms such as Simultaneous Localization and Mapping (SLAM) or Extended Kalman Filter (EKF). Although effective, these solutions often face limitations in computational complexity, power consumption or performance under certain environmental conditions such as poor lighting or lack of visual features.

An increasingly promising alternative is the use of Ultra-Wideband (UWB) technology. UWB is a radio-based communication protocol characterized by the transmission of signals across a large frequency spectrum at low power levels. This broad spectrum allows high temporal resolution in signal detection, which facilitates precise ranging between UWB-equipped nodes. Due to its robustness to interference and its ability to perform an accurate localization in cluttered indoor environments, UWB is particularly well suited for UAV navigation in GNSS-denied conditions. When integrated with other sensors onboard and robust control algorithms, the UWB can serve as an important solution for reliable UAV autonomy in constrained settings.

This thesis investigates and presents the design and implementation of an application that allows a UAV to operate in GNSS-denied environments, with a focus on indoor navigation. The system architecture is built on the Robot Operating System 2 (ROS2) Humble, integrated with the PX4 (v1.15.2-dev) autopilot firmware and the DDS-XRCE communication protocol. This combination provides a modular and scalable foundation for real-time control, sensor data integration and mission planning. This project was developed in collaboration with the LINKS Foundation using a specific hardware setup tailored for research and development purposes. Certain parts of the overall implementation strictly depend on the specific setup of the provided drone, although the core concepts and the architecture remain applicable to similar UAV platforms.

To validate the proposed system, two distinct testing environments were utilized. The first employed a motion capture (mocap) system to provide highly accurate position and orientation data, facilitating performance optimization and software debugging. The second test environment relied on UWB-based localization for position information, simulating near-real-world operational conditions. This dual stage evaluation allowed for both precision calibration and performance assessment.

In the chapters that follow, the thesis will detail the methodology, software architecture, system integration and experimental results that sustained the development of this UAV application. Through this work, the research aims to contribute to the growing body of knowledge surrounding autonomous drone operation in complex environments, ultimately supporting a wider deployment of UAVs in critical applications where GNSS cannot be relied upon.

Chapter 2

Software and technologies preliminaries

2.1 Docker

Docker is an open-source platform designed to automate the deployment, scaling and management of applications through containerization, a technology that encapsulates an application with its libraries, configuration files and dependencies into a single unit. Introduced in 2013 by Docker Inc., the platform provides an efficient method to package applications and their dependencies into portable containers, ensuring consistency across various computing environments [1]. Unlike traditional virtual machines, Docker containers share the kernel of the host operating system, making them more resource efficient while maintaining isolation between applications.

The core Docker components are:

1. Docker daemon: a background process on the host system responsible for managing Docker objects (such as images, containers, networks and volumes). The daemon enables the creation of images and the initialization and monitoring of containers;

2. Docker Images: the building blocks of Docker containers. They are immutable templates that contain the application code, runtime libraries and dependencies required to execute the application. Each docker image is created using a Dockerfile, a file that specifies the instructions to build the image step by step;

3. Containers: lightweight, runtime instances of Docker images. They encapsulate the application and its environment, ensuring consistent behavior across various host systems. Containers operate in isolation, utilizing Linux kernel features

(like namespaces and cgroups) to separate processes, allocate resources and manage permissions.

2.1.1 Containers life-cycle

The creation of a container starts with the creation of a Docker image using a Dockerfile, a file that contains a set of instructions to define all the dependencies, runtime configurations and codes to make the desired applications work. The Docker images are built from the Dockerfile by the Docker daemon using the *docker build* command. After the image has been created, a container can be initialized from it anytime by calling the Docker daemon with the *docker run* command. To efficiently run the container without overloading the host system, each container has access to a limit amount of hardware resources (such as CPU, memory and I/O) and operates as an independent process with its own filesystem, network stack and resource allocation.

The isolation between containers is obtained by dividing the system resources; this operation is done through namespaces, a particular feature provided by the kernel that isolates and virtualizes system resources for a specific set of processes. The namespaces that a container can have access to are:

- **Process ID namespace**: isolates the process ID space for each container, this brings the processes inside a container to be able to see only other processes created in the same container, not those running on the host or in other containers;
- **Mount(MNT) namespace**: manages filesystem mount points, allows the containers to have their own file system and ensures isolation and independent management;
- **Network namespace**: with this namesapce each container gets its own virtual network stack, which includes unique network interfaces and an IP address, this enables independent network configurations and avoids interference between containers;
- **UTS Namespace (UNIX Timesharing System Namespace)**: different host and domain names, presenting the container as an unique machine to the processes running inside it;
- **Interprocess Communication namespace**: provides isolation in the inter-process communication;
- **User namespaces**: remaps user IDs and group IDs inside the container to a different UID/GID on the host, this means that a root user inside the

container is seen as a non-root user outside of it or viceversa, enhancing the security of the container.

All of these namespaces can be set in the *docker build* command to obtain the desired behavior from the container and the application itself.

After the creation of the container, it is possible to use it as an independent application or even as an entire system.

When a container is deleted or stopped, all of its data is lost, to address this, there are two possible solutions: docker volumes or bind mounts. The docker volumes are persistent data stores created and managed by docker on the host file system, while bind mounts means setting up the mount namespace in order to directly map a host directory or file into the container.

2.1.2 Containers characteristics

The characteristics common to all containers are:

- **Portability**: the containers package all the dependencies and configurations into a single unit;
- **Efficiency**: containers are lightweight and consuming minimal system resources compared to traditional virtual machines;
- **Consistency**: Docker containers ensure a uniform runtime environment, reducing discrepancies across development, testing, and production stages;
- **Scalability**: containers can be easily replicated or scaled horizontally to handle varying workloads;
- **Flexibility**: Docker supports multi-platform deployment, enabling seamless operation on Linux, Windows, or cloud environments.

2.2 Robotic operating system 2

The robotic operating system (ROS) 2 is a standardized framework for robotic development [2] [3]. It was born in 2014 from the Open Source Robotics Foundation (OSRF) as an improvement of the previous ROS 1. The idea behind its creation is to maintain the communication structure of ROS 1 but solve its limitations in real-time and distributed systems, limitations caused by its reliance on a single-threaded architecture and the use of a centralized master node, characteristics no longer present in ROS 2.

2.2.1 Key Features of ROS 2

ROS 2 introduces a series of architectural enhancements that address the shortcomings of its predecessor:

- **Middleware abstraction with the data distribution service (DDS)**: DDS is an industry-standard protocol designed for real-time systems and distributed applications. This abstraction enables peer-to-peer communication, which allows decentralized communication and enhances scalability, and to support various Quality of Service (QoS) settings, developers can configure QoS policies to prioritize data delivery, ensure reliability or reduce latency. DDS ensures reliable communication in distributed and resource-constrained environments;
- **Real-time capabilities**: ROS 2 integrates real-time capabilities by allowing developers to use real-time operating systems (RTOS). These systems ensure deterministic behavior by prioritizing high-priority tasks and minimizing latency;
- **Security enhancements**: ROS 2 incorporates DDS Security, providing tools for authentication, encryption and access control. These features mitigate the risks associated with malicious actors and unauthorized access, ensuring the integrity of robotic systems;
- **Cross-platform support**: ROS 2 is designed to run on multiple operating systems, including Linux, Windows and macOS. This flexibility makes it accessible to a broader range of developers and applications. The modularity of ROS 2 allows developers to use only the components they need, reducing complexity and resource usage.

2.2.2 Structural components of ROS 2

The architecture of ROS2 is based on 4 core elements, these elements form the backbone of ROS 2's decentralized architecture, enabling perfect interactions between distributed system components and a highly adaptable communication infrastructure for robotic systems. The elements are as follows:

- **Nodes**: nodes serve as the origin or destination of all data flows in ROS 2 and are the fundamental units of computation in ROS 2, encapsulating distinct functionality in a highly modular form. Each node is responsible for executing specific tasks, such as sensor data acquisition, control algorithms or actuation commands. Furthermore, nodes can operate independently or collaboratively in a distributed environment, enhancing system robustness and fault tolerance;

- **Topics**: topics are the elements that truly manage and allow the exchange of messages in the ROS2 architecture; they create an asynchronous, many-to-many communication through a publish-subscribe model. Nodes publish messages to topics and other nodes subscribe to these topics to receive the data. This approach is particularly well-suited for high-frequency unidirectional data streams, such as sensor readings or telemetry data. The topics are enhanced using Quality of Service (QoS) policies, which provide control over message delivery reliability, durability and latency;
- **Services**: Services provide a synchronous one-to-one communication model between nodes. They are analogous to function calls, where one node sends a request and awaits a response from another node. Services are typically used for operations that require immediate feedback, such as querying a robot's current state, triggering a specific behavior or setting parameters. This tightly coupled communication pattern is well-suited for tasks requiring precise coordination between components;
- **Actions**: Actions are an extension of services; they are designed to handle long-running tasks that require progress feedback and cancellation capabilities. Unlike the request-response model of services, actions allow for ongoing interaction between nodes during task execution. An action consists of three message types: goal, feedback and result. The goal initiates the task, feedback provides updates on the progress of the task, and the result signals its completion. Actions provide a robust mechanism for managing complex behaviors that take time.

An example of connection between the above elements is shown in Figure 2.1.

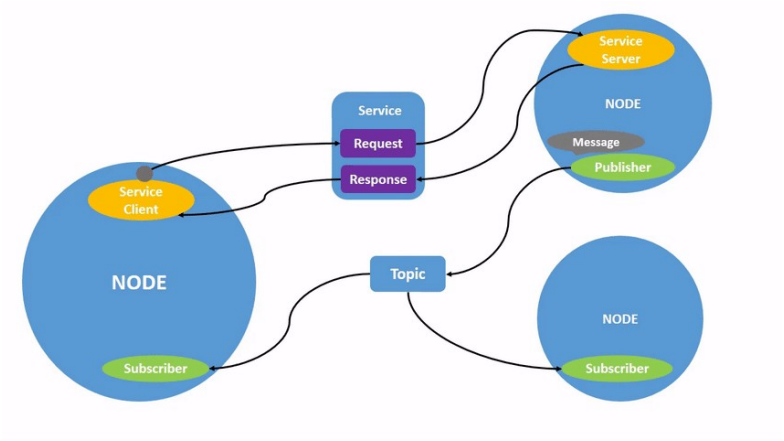

Figure 2.1: ROS2 communication structure. Image taken from [3]

2.3 PX4

PX4 is an open source flight control software for drones and other unmanned vehicles [4][5]. The project provides a flexible set of tools for drone developers to share technologies to create customized solutions for drone applications. PX4 provides a standard to deliver drone hardware support and software stack, allowing an ecosystem to build and maintain hardware and software in a scalable way.

2.3.1 System architecture

The architecture of the PX4 system is composed by three main parts (as shown with three different colors in Figure 2.2):

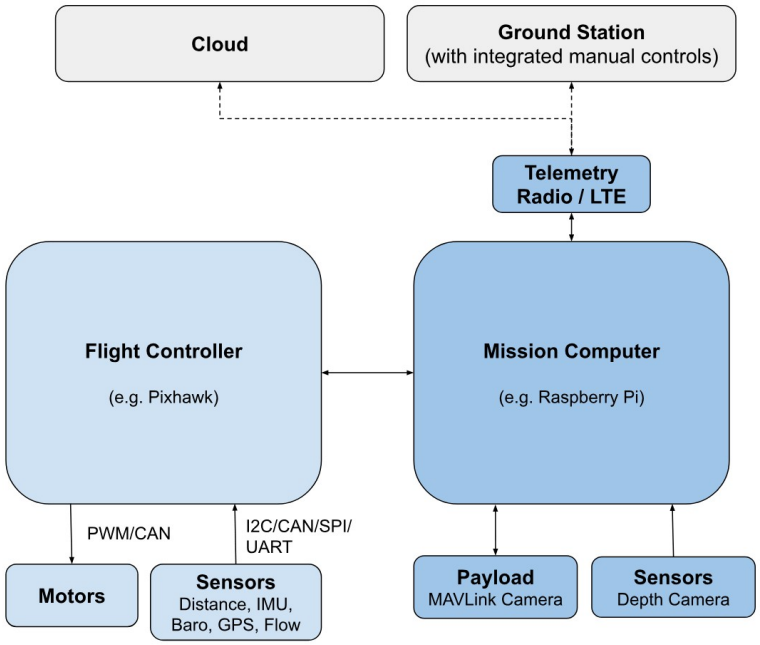

Figure 2.2: PX4 system architecture. Image taken from [5]

- **Remote controller** (an RC system or a ground station): the only part of the system architecture outside the vehicle and the one which the user interacts to. This element remotely controls the behavior of the unmanned vehicle. If the controller is a ground station, it usually runs QGroundControl (or some other ground station software) and a robotic software like MAVSDK or ROS (1 or 2) to communicate with the flight controller;
- **Mission computer** (also called companion board): a small board with computation capacities, the only non-mandatory component in the system

architecture. If present, it is directly connected to the flight controller through a serial connection or IP link and communicates with the ground station; it can also be connected to external sensors to improve the information about the position and orientation of the system. It provides advanced features to the vehicle like mission managing and computer vision applications (collision avoidance, collision prevention, visual odometry, safe landing, etc.). Its software part is composed of the applications it provides and a middleware to communicate with the flight controller, the type of middleware depends on the software architecture and PX4 version. In this work, the connection between the companion board and the flight controller is obtained through the DDS-XRCE protocol, a protocol that is well suited to communicate with ROS2 nodes;

- **Flight controller**: the most important part of the vehicle, contains all the software needed to make the vehicle fly, it directly communicates with the motors and some sensors through serial connection. On the software side, we have two main parts: the middleware and the flight stack. The middleware consists of two parts: device drivers for embedded sensors and communication with the external world (companion computer, GCS, etc.) and the μORB asynchronous publish-subscribe message API. The μORB API consists of a bus used for interthread/interprocess communication, its publish-subscriber scheme makes the system reactive and thread safe and parallelizes all the operation and communication. The flight stack is a collection of guidance, navigation and control algorithms for autonomous drones, its building blocks are shown in Figure 2.3: the estimator is composed by an EKF2 that takes in input all the data coming from the sensors and/or a companion board and computes the odometry of the vehicle, it works as explained in [6]; the position, attitude and rate controllers compose the control architecture of PX4 and depends on the UAV type; the navigator manages the autonomous flying modes; the mixer takes force commands (such as "turn right") and translates them into individual motor commands while ensuring that vehicle velocity and acceleration limits are not exceeded.

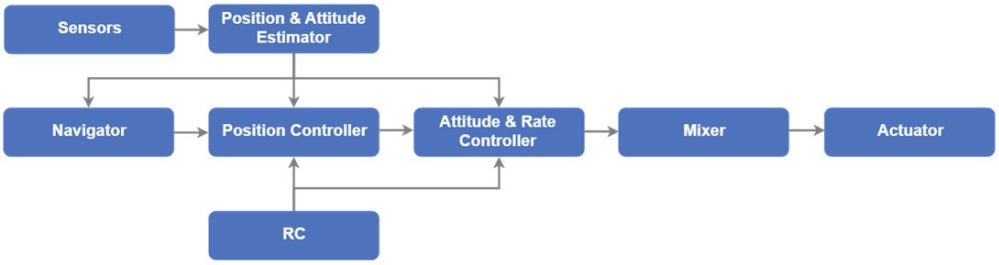

Figure 2.3: PX4 flight stack overview. Image taken from [5]

2.3.2 DDS-XRCE protocol

The DDS-XRCE protocol is a specification that enables a resource-constrained, low-power device to communicate with a DDS network, like ROS2, and ensures compatibility between applications based on different vendors' implementations of the XRCE protocol. In this work, the protocol is implemented with the eProsima Micro XRCE-DDS library [7], which respects the protocol guidelines specified in the *DDS for eXtremely Resource Constrained Environments* standard as defined and maintained by the Object Management Group (OMG) consortium [8].

The communication defined by the protocol is based on a client-server architecture where the server (XRCE Agent) acts as an intermediary between the clients (XRCE Clients), which run on the constrained devices, and the DDS global data space. The communication between XRCE Clients and XRCE Agents is based on operations and responses. Clients request operations to the agents, which generate responses with the result of these operations, and once the clients are informed about the result of the operation, they can perform subsequent actions or request further operations. The architecture can be created with both serial and wireless connections.

The setup of the architecture of the PX4 system is shown in Figure 2.4, the XRCE-Client runs on the flight controller, while the XRCE-Agent runs on the companion board.

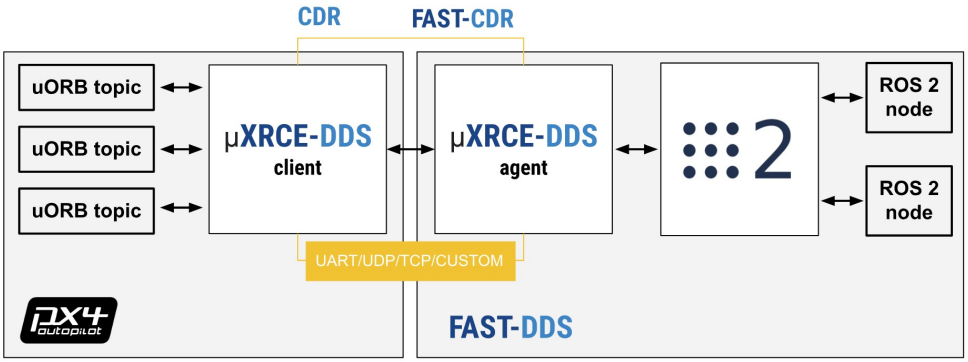

Figure 2.4: PX4 flight stack overview. Image taken from [5]

2.3.3 Control architecture

The control architecture varies with the UAV type, since a multicopter was used for this work, only its architecture will be analyzed more in-depth.

The high-level composition, comprehensive of the mixer block, is shown in Figure 2.5, it is a cascaded mix of P and PID controllers. Depending on the flight mode, the outer loop (the position loop) can be bypassed and behaves like a multiplexer;

this loop is only used when holding position or when the requested velocity on an axis is null. The input of the architecture are the desired position and yaw angle, the output is the computed thrust force to apply to the rotors. All of the estimates needed, from any block, come from the EKF2.

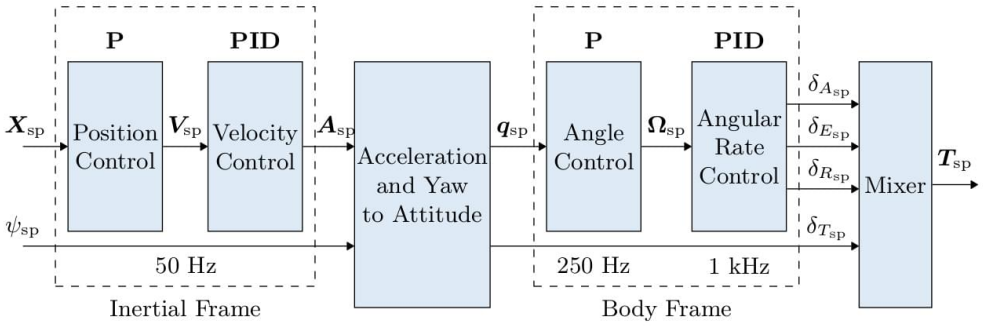

Figure 2.5: PX4 global control architecture. Image taken from [5]

Starting from the left, we have:

- **Position control**: is composed by two P controllers, one for the altitude and one for the horizontal position, which command in velocity. The output is saturated to not damage the vehicle. Its simple structure is shown in Figure 2.6;

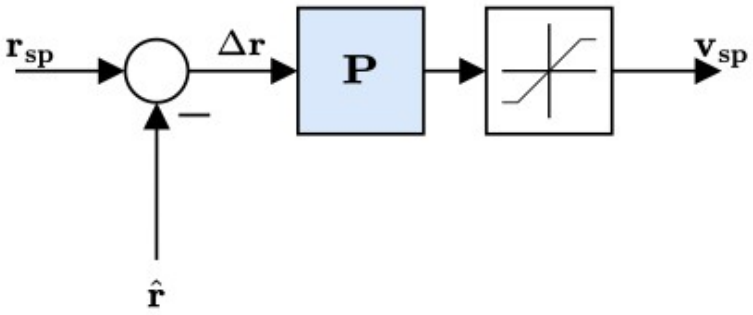

Figure 2.6: PX4 position controller architecture. Image taken from [5]

- **Velocity control**: contains three PID controllers, one for each axis, that stabilize velocity and command in acceleration. In this case, the output is not limited because the saturation will be applied in the next step. The controllers have a low pass filter in the derivative path to reduce noise (as shown in Figure 2.7);

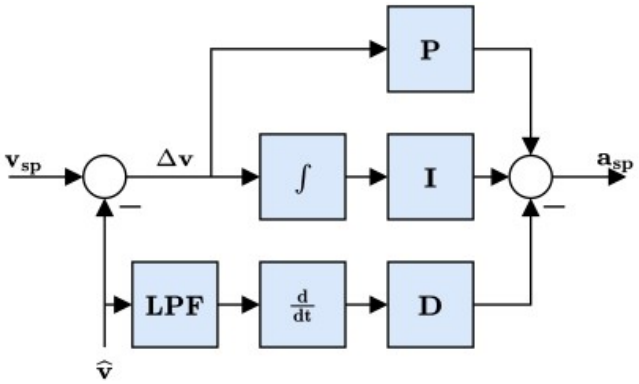

Figure 2.7: PX4 velocity controller architecture. Image taken from [5]

- **Acceleration and yaw to attitude**: this part of the architecture converts the acceleration generated by the velocity controller and the input yaw angle into desired orientation, in the quaternion form, and thrust;
- **Angle (or attitude) control**: this controller takes as input the desired orientation, derived from the transformations in the previous block, in quaternion form. The controller scheme is shown in Figure 2.8 and implements the controller explained in [9];

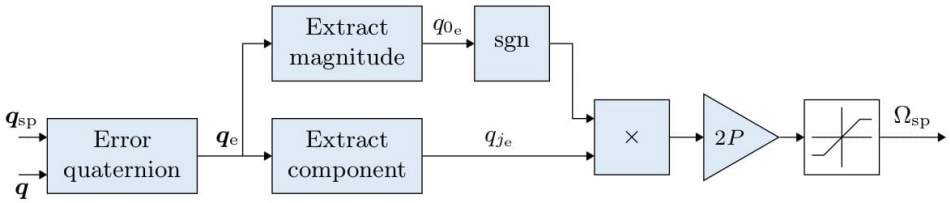

Figure 2.8: PX4 attitude controller architecture. Image taken from [5]

- **Angular rate control**: controls the three body rates (roll, pitch and yaw) with a K-PID controller each. The K-PID controller is a controller that, depending on the values of P and K, can change its form: with K equal to 1 we have the *parallel form*, the form commonly used in textbooks; with P equal to 1 we have the *standard form*, a form that decouples the proportional gain tuning from the integral and derivative gain; this facilitates the tuning phase because it is possible to tune the controller starting from the parameters of a drone with similar size and inertia and tune only the K parameter.

In each controller (as shown in Figure 2.9) there is an LPF to reduce noise and a limitation in the integral path to reduce wind up;

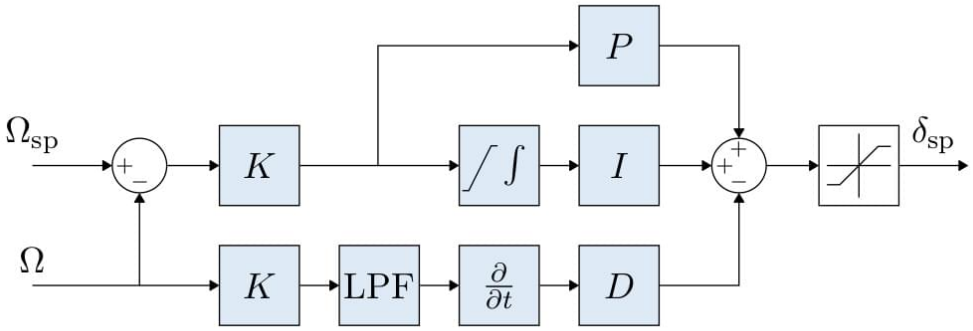

Figure 2.9: PX4 rate controller architecture. Image taken from [5]

- **Mixer**: translates the commands that arrived from the previous blocks into torque to assign to each actuator to achieve the desired movement of the vehicle.

2.3.4 Multicopter flight modes

PX4 offers several multicopter flying modes; they can be divided into two macro categories: manual and autonomous. Manual modes are programmed to maintain the drone stable and execute the command received from an RC, autonomous modes instead focus on the execution of one or more specific maneuvers without the necessity of external command sources, with the exception of the *offboard* mode that receives external commands but not from an RC. The flight modes are:

- **Manual**:
  - **Position mode**: maintain the vehicle stable at the current horizontal position; in this mode the radio control can only change the horizontal orientation and the altitude of the drone, if one of them is changed, this mode will maintain the new position and orientation. This mode requires a position fix or a GPS;
  - **Position slow mode**: this mode works just like the Position mode, but with limitations on the velocity and yaw rate;
  - **Altitude mode**: similar to position mode, but in this case the GPS and a position are not needed. This mode only maintains the altitude and does not actively hold the horizontal position;

  - **Stabilized mode**: focuses on the horizontal posture of the vehicle, the vehicle will continue to move with momentum, and both altitude and horizontal position may be affected by wind;
  - **Acro mode**: only acrobatic mode. This mode only stops the rotation of the vehicle if not required, but otherwise does not stabilize the vehicle.

- **Autonomous**:
  - **Hold mode**: vehicle stops and maintains its current position and altitude against wind and other forces;
  - **Return mode**: the vehicle ascends to a safe altitude, flies a clear path to a safe location (home or a rally point) and then lands. This mode requires a global position estimate (GPS);
  - **Mission mode**: activating this mode, the vehicle executes a predefined mission/flight plan that has been uploaded to the flight controller. A GPS is required for this mode;
  - **Takeoff mode**: vehicle automatically takes off vertically, after reaching a desired altitude, it switches to Hold mode;
  - **Land mode**: the vehicle immediately starts the landing procedure;
  - **Orbit mode**: vehicle flies in a circle, ensuring that the yaw axes always face towards the center of the orbit. RC control can optionally be used to change the orbit radius, direction and speed;
  - **Follow Me mode**: vehicle follows a beacon that provides position setpoints. RC control can optionally be used to set the follow position;
  - **Offboard mode**: Vehicle obeys position, velocity or attitude setpoints provided via MAVLink or ROS 2.

2.4 Vicon tracker

The Vicon tracker is a MoCap system developed and sold by Vicon Inc., its purpose is to analyze and register objects or human movement. The main features of this application are that it can easily be integrated in various external platforms and can achieve a micrometer accuracy while maintaining immediate feedback. The performance of Vicon tracker system are obtained by integrating high-performance hardware with advanced real-time software processing.

2.4.1 System architecture

The system architecture of the Vicon tracker is made up of four elements:

- **High-speed optical cameras**: high-resolution cameras capable of capturing thousands of frames per second. They must be strategically placed to obtain a comprehensive coverage around the desired volume and capture markers from multiple angles (for accurate triangulation). To obtain a clear representation of the object, at least three cameras must be able to see and identify the object. Furthermore, the cameras are synchronized to capture data at the same time to increase the accuracy of the reconstruction algorithm;
- **Infrared illumination and reflective markers**: the cameras use infrared illumination to uniformly illuminate the capture volume. To identify and track an object, it must be covered with markers, small items covered with retro-reflective material that reflect back the infrared light of the cameras to obtain a precise detection. The disposal of the markers must be: visible, most of the markers must be always visible by at least one camera in any position and orientation; unique, to prevent incorrect identification; asymmetric, to increase the orientation precision;
- **Central processing unit**: the cameras transmit image data to the central unit through a high-bandwidth connection to ensure minimal delay; the raw image data are passed through a data pipeline where real-time processing algorithms extract and compute the 3D position of the object;
- **Communication interfaces**: the system uses high-speed network interfaces that transmit processed tracking data to external devices with minimal latency. The tracker supports various communication protocols and APIs, making its integration with other applications almost immediate.

2.4.2 Software and algorithms

The data processing pipeline is made up of three steps:

- **Image processing and marker detection**: each frame is processed to identify and isolate the markers, the identification is based on the intensity of the reflected infrared light, the markers should be the brightest points in the cameras view. Once the markers are identified, the tracker assigns unique identifiers to each to ensure consistent tracking across frames, even when they temporarily disappear;
- **Data filtering and noise reduction**: to ensure smooth motion tracking, temporal filters, such as the Kalman filter, are used to predict the position of the marker and maintain the identity of each marker between frames. The software continuously monitors the data for anomalies or outliers caused by occlusions or reflective interferences and corrects them in real time;

- **3d reconstruction**: after the identification of the object, using the known camera positions and orientations and the 2D coordinates of the markers in the frames received, the software applies triangulation techniques to compute the 3D position and orientation of the object, with respect to the origin and orientation of the tracker volume, in real time.

The entire process is designed to be scalable and modular, allowing additional cameras or processing units to be added if needed, and to facilitate the update or integration of new tracking algorithms or fusion techniques.

Even if the tracking is extremely precise, it is possible to incorporate other sensors into the tracking system to increase its robustness even in challenging conditions.

2.5 UWB technology

The ultra-wideband (UWB) technology was developed in the early twentieth century and was used in the military field until the 1980s. In the 1990s the United States Federal Communication Commission (FCC) released the first regulations for UWB communications, and because of the potential of this technology, in the 2000s also the IEEE started regulate it, opening its commercial use.

The ultra-wideband technology is optimized for short-range communication, from a few meters up to 100 meters. Its main characteristics are:

- **High bandwidth**: as the name suggests, the main feature of this technology is its high bandwidth (from 3.1 to 10.6 Ghz), which allows transmission of signals with a wide frequency band (typically greater than 500 Mhz) that corresponds to a high data rate;
- **Low power consumption**: since power is spread over a large bandwidth, UWB communication systems are power efficient and operate at very low power levels, causing minimal interference to other systems;
- **High temporal resolution**: the UWB technology operates by sending short-duration pulses (nanoseconds or less), and it focuses on the precise timing of pulse arrival rather than the frequency, these features achieve fine temporal resolution.

These characteristics are particularly useful in the localization problem [10], the high bandwidth can penetrate obstacles and take advantage of multipath reflection (where the signal bounces on the obstacles in the environment), giving reliable performances even in non-line of sight (NLOS) environments, while the temporal resolution, when integrated with time-based localization algorithm, offers

a position estimation with a centimeter-level precision. Despite these benefits, UWB technology requires precise synchronization and complex signal processing to obtain good performance. Moreover, although the low power consumption makes the UWB resilient to interference, it is not immune.

However, considering its characteristics, the UWB technology remains a suitable candidate for applications operating in challenging and GNSS-denied environments, both indoors and outdoors.

Among all the possible localization techniques, in this work the localization with UWB is obtained through trilateration with a two-way ranging time of flight (TWR-ToF).

Trilateration is a technique that enables the determination of the absolute or relative position of a point by measuring distances and exploiting the geometry of circles, spheres and triangles. This localization process is performed measuring the distance between the target (or tag) and multiple fixed references (or anchors), the target position is located at the intersection of these measurements. Since the distances are obtained using waves without the aid of directional antennas, a minimum number of three anchors is needed to obtain a precise location. With one antenna there could be infinite possible positions along the circumference of a circle, while with two the intersection of the corresponding distance circles gives two possible locations, resulting in ambiguity. Therefore, in both scenarios, the localization is not achieved. With three (or more) anchors, we obtain a setup as the one shown in Figure 2.10 that gives only one possible solution to the localization problem.

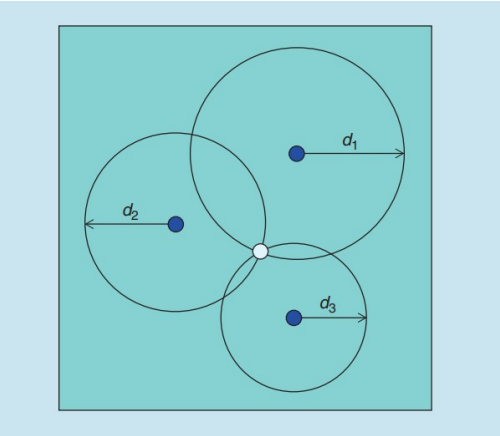

Figure 2.10: Trilateration obtained with three references. Image taken from [10]

The TWR-ToF is a technique to measure the distance between a target (or tag) and a transmitter (or anchor), it falls under the time of arrival (ToA) umbrella, where the distance is obtained with the time of flight. Usually, the time of flight is

computed with a single message exchange that the anchor uses to know the total round-trip time and compute the time of flight with:

$$ToF = \frac{(T_{tx-recv} - T_{tx-send}) - (T_{rx-send} - T_{rx-recv})}{2} \tag{2.1}$$

($(T_{tx-recv} - T_{tx-send})$ is the total roundtrip time and $(T_{rx-send} - T_{rx-recv})$ is the response time). After the computation of the ToF, the total distance is obtained with:

$$Distance = Speed of Light * ToF \tag{2.2}$$

The main problems with this approach are that the transmitter must know the reply time of the target and both the entities must be perfectly synchronized. To address these problems, the TWR-ToF implemented in the LINKS laboratory uses a Poll-Response-Final method to compute the ToF [11]. The method is shown in Figure 2.11 (*DEVICE A* is the tag, *DEVICE B* is the anchor and T_{prop} is the ToF), in the final message the tag sends its T_{round} and T_{reply} to the anchor that computes the ToF with:

$$T_{prop} = \frac{T_{round1} * T_{round2} - T_{reply1} - T_{reply2}}{T_{round1} + T_{round2} + T_{reply1} + T_{reply2}} \tag{2.3}$$

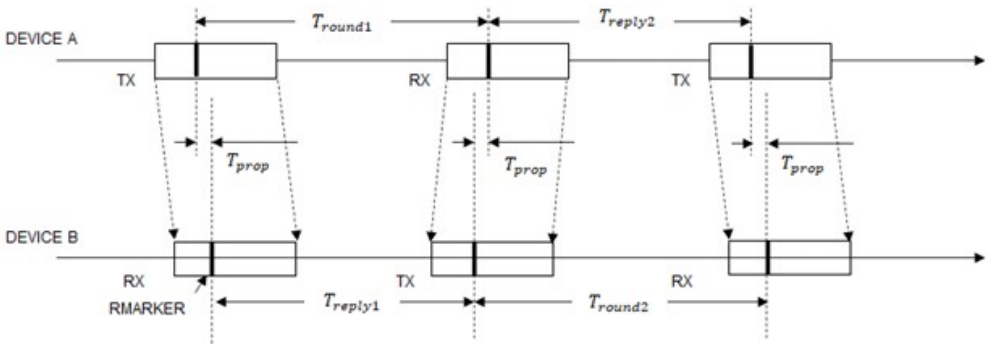

Figure 2.11: Poll-Response-Final method messages exchange. Image taken from [11]

2.6 UAV components

The UAV used in this work was assembled and provided by the LINKS foundation, it is a small quadcopter composed by a flight controller and a companion board connected through a UART port.

The flight controller, shown in Figure 2.13, is a Kakute H7 v 1.3, the main features of this board are the presence of an IMU sensor and a barometer sensor, both helpful during flight, and the absence of a network adapter, which can be a useful alternative communication channel in some scenarios. This board comes with a preinstalled version of BetaFlight, a software to manage the flight of a vehicle, but it can be easily substituted with the PX4 firmware. For a deeper understanding of the hardware components of the flight controller, refer to [12].

The companion board mounted on the UAV is a VOXL board (Figure 2.12), a board with high computational and memory capacities developed to be an onboard computer. It has a Yocto version of Linux as its operative system and has preinstalled several software and services to enhance its role as onboard computer, the most important preinstalled software and service for the development of this work is an older (but perfectly working) version of Docker and its service "docker-autorun", which automatically starts a container when the board is started. Another important feature of this board is that all serial ports and GPIO pins are internally mapped to the Sensors DSP (SDSP), with the benefit that the low-level and time-sensitive interaction with sensors and telemetry communications can be handled by the SDSP's real-time operating system, freeing up CPU cycles on the applications processor. However, this means that it is not possible to talk to serial ports with reads and writes to /dev/i2cX or /dev/ttyOX as in other embedded Linux systems. Communication is permitted by a library provided by the VOXL development team, called libvoxl-io. More information on VOXL components and software can be found in [13].

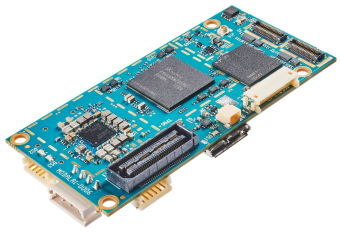

Figure 2.12: VOXL board. Image taken from [13]

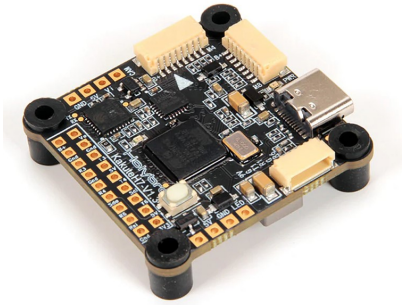

Figure 2.13: Kakute H7 v 1.3 board. Image taken from [12]

Chapter 3

Software design and implementation

The software architecture is built upon PX4 and ROS2 Humble, these two software were chosen because they are at the forefront in their field and have plenty of documentation and community support. The specific ROS2 distribution was chosen to match the one used in other LINKS foundation works in order to simplify a possible integration or communication with them. Both software have a specific role: PX4 manages vehicle flight, from data acquisition to rotor commands; ROS2 builds the network to enable communication from external sources to the flight controller. The "external sources" are entities physically detached from the drone but still included in the architecture; they are: a ground station from which a user can activate and control the drone and, since we are in a GPS denied environment, the external source of position and orientation. In this work, the Vicon tracker and UWB technology were used as external sources of position and orientation.

3.1 UAV setup and internal connections

The components chosen require a specific setup to make this application work, the VOXL board has to install and run the ROS2 libraries while the flight controller has to install the PX4 firmware and change its parameters to match the shape and the characteristics of the drone to obtain good performances. After installation, it is necessary to use the serial connection of the two boards to set up the communication channel that allows the two software to communicate with each other.

3.1.1 VOXL configuration

The VOXL board naturally does not support any ROS2 distribution. However, it is possible to use the ROS2 libraries on the board using the preinstalled docker daemon. The first step is to create a docker image that builds an environment that contains all ROS2 dependencies and nodes. Starting from that image, it is possible to run a docker container with all the requirements needed for this application. Since this container is the only solution to use ROS2 in the VOXL, it must always be active. To address this, it is possible to set the container to start at the start of the board using the docker-autorun service present it. With this procedure, the VOXL board is virtually transformed into a system with ROS2 installed and running. However, this solution has several problems: even if the container is set to be removed when it is properly stopped, it will not be removed if the board is turned off while the container is still running; is not possible to create more than one container with the same name and it is not possible to create again the same container with a new name every time the board is restarted, it will fill the memory of the board. These problems have been resolved by assigning to the container a specific name and, before creating the new container at the start of the board, removing the old container, if present.

Since the VOXL system will be virtually replaced by a docker container to allow the execution of this application, the container needs some properties to carry out its task. Those properties are set during the initialization of the container. The command to start the container is:

docker run --rm --privileged --net=host --name <container name> -v /home/-root:/root/yoctohome/:rw -w /root/ <docker image> /entrypoint.sh

Each of the options of this command gives the container a particular feature:

- *--rm*: completely removes the container from the memory of the board after it is properly stopped;
- *--privileged*: gives the container the possibility to do all the operations that a privileged user can do;
- *--net=host*: connects the network of the container to the one of the host, with this property the container can sand and receive messages to all the systems connected to the same network of the board;
- *--name*: sets the name of the container, it must be unique;
- *-v*: mounts a directory on the host to the container;
- *-w*: sets the working directory of the container;

- the last two parameters are the starting docker image and the file to be executed at the start of the container, respectively.

3.1.2 PX4's parameters setup

The Kakute h7 board comes with a BetaFlight firmware, the procedure to upload PX4 to the Kakute h7 is not hard and well explained in [5].

Normally, the firmware cannot work in this kind of application, it is programmed to use GPS's data to fly a vehicle, however, it requires small changes to be suitable for the purpose. The firmware requires two fundamental features to be set before uploading it on the board (they cannot be modified after). These features are:

- **Disable GPS module**: Disabling the GPS module prevents the upload of the GPS drivers on the flight controller. All the positional information sent to the flight controller is received and handled by the PX4 estimator, it combines the data and computes the current position and orientation of the drone. This process is not executed if the estimator does not receive the GPS data, the firmware is programmed to merge the received data only if at least one of them comes from a GPS device, if this requirement is not satisfied, the EKF2 will ignore all the data. Disabling the GPS module before the upload of the firmware removes the block, it prepares the EKF2 to work without GPS data and compute the position and orientation with other sources, which is fundamental for the target purpose of this work;

- **External vision module enabled**: this module set the firmware to receive data from an external vision source, this setting enables several parameters to manage the messages from this source and creates a communication channel that sends the external vision messages to the EKF2 to combine them. In this application, information on the position and orientation of the drone will be transmitted using the external vision communication channel.

These two modules are the only two that have to be set as explained to make the application work; every other module can vary the performances of the flight but does not change the flow of the application. All modules and drivers to be installed can be selected using a GUI provided by the PX4 development team, it can be found with all the default firmware versions for all the supported boards at this link:https://github.com/PX4/PX4-Autopilot.git. The GUI can be accessed with the command:

make holybro_kakute_h7 boardconfig

3.1.3 Components communication: voxl-parser

As mentioned in chapter 2.3, the communication between ROS2 and PX4 is obtained through the DDS-XRCE protocol; in this work, the protocol is implemented with the *eProsima Micro XRCE-DDS* library (recommended by the PX4 documentation). This library creates a client-server architecture to permit resource-constrained devices to take part in DDS communication, it can create the architecture with both serial and IP-based (e.g. UDP or TCP) communication channels. The Micro XRCE-DDS client is pre-installed within the PX4 firmware and uses the communication type supported by the flight controller it runs on, the serial one in this case. The Micro XRCE-DDS agent runs on the VOXL inside the docker container created to use the ROS2 libraries; this solution was adopted since some of the programs needed to install and run the agent are not present on the board and cannot be installed or are present but with an old and not compatible version that cannot be updated.
However, the hardware choices brought a big problem to this communication setup, the lack of an IP-network interface on the flight controller precludes the IP-based connection while the management of the serial ports by the VOXL system unite to the necessity to run the agent inside a container blocks the serial connection (the docker containers do not have the authorization to use the libapq8096 library). To address this problem, I created the voxl-parser.

The voxl-parser is a software module that I created to enable the communication between the Micro XRCE-DDS client and agent in this hardware setup. It runs as a service on the VOXL board, outside of docker, and serves as a bridge between the serial connection wanted by the flight controller and the IP-based connection needed by the agent (in this case the UDP protocol was chosen), its addition creates the communication architecture shown in Figure 3.1.

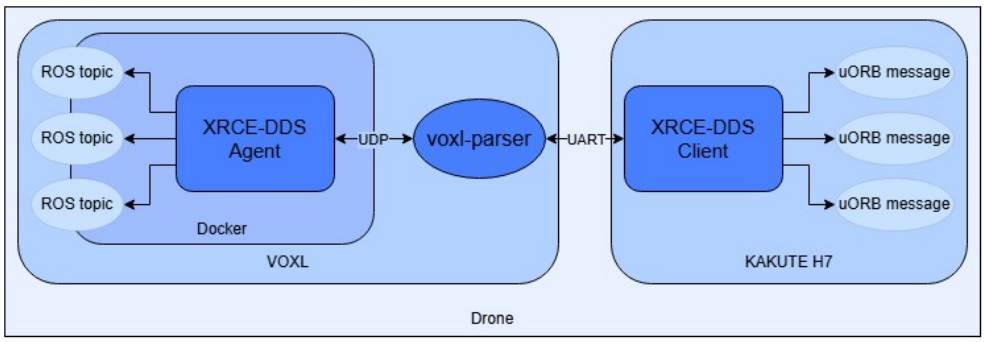

Figure 3.1: UAV internal communication architecture

The voxl-parser is a program written in C language composed of three threads, two threads handle the messages from the client to the agent and one thread for

the opposite ones. The number of threads and their purposes have been chosen looking at the operation that the program has to do. The program has to handle an asynchronous bidirectional communication, to not lose any message and to increase the overall speed, each side of the communication has to be handled separately. The different number of threads for each side of the connection comes from the amount of messages that each side has to handle. After some tests, it has been observed that the stream of messages from the agent to the client can be handled by one thread without worsening the performance, while the other part of the connection, with longer and more numerous messages, benefited from the use of two threads, the division between the reception and manipulation of bytes and the submission improved the overall performance. The three threads work as follows:

- The first thread handles the first part of the communication client-agent (Algorithm 1), it continuously tries to read the bytes arriving at the serial port from the client, if there is something to read, it extracts the messages contained and stores them in a queue waiting to be processed by the other thread.
  The read operation is executed using the specific voxl function *voxl_uart_read()*, this function reads all the bytes arrived on the UART port and stores them in a vector, if the vector cannot contain all the incoming bytes, an error occurs. The extraction operation is performed with the *read_msg()* function, this function takes the vector of bytes and extracts all the payloads of all the messages contained in it one at a time, the extraction is possible by following the known message structure of this type of communication and the serial protocol. The function processes each byte to find and extract the payload, it also verify the correctness of the message by comparing the cyclic redundancy check (CRC) bytes contained in the message to the ones computed during the processing of the bytes. The extracted payloads are stored in a shared queue waiting to be send to the agent;

- The second thread work is to send the messages arrived from the client to the agent (Algorithm 2), this thread pops the messages from the queue and send them to the agent with an UDP message, if no message is present, this thread waits without spend machine cycles;

- The last thread (Algorithm 3) solely handles the communication agent-client, it uses a blocking function (*recvfrom()*) to wait, without use any machine cycle, a message from the agent, every message that arrive is slightly modified and sent to the client.
  The necessary changes to the messages are made with the function *recv_from()*, this function adds the serial header to the message given as input and modifies it to respect the serial communication protocol.

Algorithm 1 Handles the incoming messages from the client and stores them in a queue. The *queue* is protected by a mutex lock

```
1: procedure CLIENT TO QUEUE(queue)
2:     ▷ queue contains the message queue
3:     ▷ Initialization
4:     rb ← 0                                   ▷ bytes received from the client
5:     msg ← 0                                  ▷ bytes to send to the client set to 0
6:     ▷ Execution
7:     while 1 do
8:         rb ← voxl_uart_read()                ▷ read bytes arrived to the uart port
9:         if no errors occured then
10:            while rb is not end do
11:                msg ← read_msg(rb)           ▷ extract a msg from the bytes
12:                push msg into queue
13:            end while
14:        end if
15:    end while
16:    free memory
17:    return 0
18: end procedure
```

Algorithm 2 Sends the messages to the agent. The *queue* is protected by a mutex lock

```
1: procedure QUEUE TO AGENT(sd,queue)
2:     ▷ sd contains all the information about the udp communication
3:     ▷ queue contains the message queue
4:     ▷ Execution
5:     while 1 do
6:         pop msg from queue
7:         sendto(sd,msg)                       ▷ send msg to the agent
8:     end while
9:     return 0
10: end procedure
```

Algorithm 3 Agent to client transmission algorithm.

```
1: procedure AGENT TO CLIENT(sd)
2:     ▷ sd contains all the information about the udp communication
3:     ▷ Initialization
4:     wb ← 0                                  ▷ bytes to send to the client set to 0
5:     msg ← 0                                 ▷ bytes received from the agent set to 0
6:     ▷ Execution
7:     while 1 do
8:         msg ← recvfrom(sd)                  ▷ bytes sent by agent
9:         if no errors occured then
10:            wb ← compose_msg(msg)           ▷ adapt msg to serial comm.
11:            send wb to the client
12:        end if
13:    end while
14:    free memory
15:    return 0
16: end procedure
```

3.2 Software architecture

The software architecture, shown in Figure 3.2, is a network to enable communication from any source of position and orientation to PX4 and to control the drone from a ground station without the use of a radio controller. The communication between the *external source of position and orientation* and the drone is obtained through UDP messages, every other communication in the network uses ROS2 topics. The structure of the network has been chosen to permit an easy change of the source of position and make the architecture adapt to any application; with this modular design, if the external source changes, only one node (the coordinate publisher) needs to be changed. The only prerequisite of this application is that PX4 must be set in offboard mode, every other mode will not execute any command received from the companion board.

3.2.1 External source of position and orientation

The external source of position and orientation can be any source, even more than one, the only requirement is that the results of the measurements can give information about both position and orientation of the object. In this work, two slightly different setups were used; the first setup used only a MoCap system (VICON) to obtain all the necessary information, while the second setup used the UWB technology for the horizontal position and the Mocap system for the

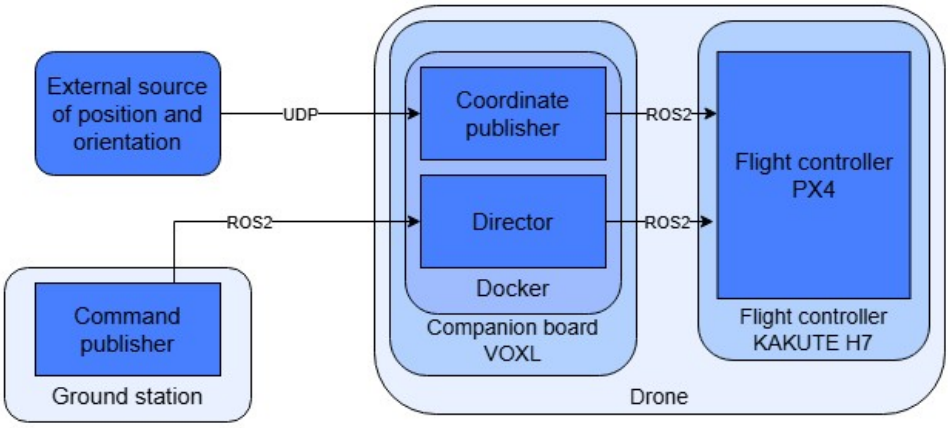

Figure 3.2: Logical environment setup

vertical position and the orientation. Both setups have a centralized structure, the computation of the position and orientation is made by a single unit and these units communicate with the drone through an IP-based communication. The MoCap system used can broadcast a single VICON message containing all the information needed with a minimum frequency of 100 hz, the central unit of the UWB setup can instead send a message with a frequency of 40 hz. In both cases, it is up to the *coordinate publisher* node to handle the messages received. The physical implementation of the two types of external sources is explained in Section 4.1.

3.2.2 Coordinate publisher

The Coordinate publisher in one of the two ROS2 nodes running inside the VOXL board, it is posed between the external source of position and orientation and the firmware. Its role is to acquire the vehicle odometry measures and send them to PX4 in the correct format. This node has a dual purpose: translating the incoming data into a format understandable by the destination and guaranteeing that the messages arrive to the firmware with the frequency required by it (between 30 and 50 hz); otherwise, the data will be ignored. The messages are sent to the firmware as a *VehicleOdometry* message through the *vehicle_visual_odometry* topic, in this way, the data are sent to the EKF2 and used to compute the estimated position. Since during this work two different external sources that communicate in two slightly different ways were used, I implemented two different coordinate publisher nodes:

- The node for the setup with the MoCap system has to communicate with VICON and handle its messages (the node implemented in this work is a modified

version of a node specifically designed to translate the messages of a MoCap system into ROS2 messages, the original node can be found in this repository: https://github.com/MOCAP4ROS2-Project/mocap4ros2_vicon.git). The node works as follows: it connects to the VICON data stream and every 20 ms it extracts a message and uses the data inside to create a new message to send to PX4 (the extraction frequency has been set to the maximum to deal with possible delays). A summary of the handling procedure is shown in Algorithm 4, the procedure can be divided into two parts: data acquisition and message construction. The data acquisition is made using several functions specifically developed for the VICON messages, these functions take the data of the segment of the object whose identifiers are passed as argument (in this work a single object with a single segment was used, so the identifiers are not so relevant). The message construction is a simple allocation of the information previously retrieved in the corresponding fields of the PX4 message.
Unfortunately, VICON does not measure the velocity of the objects it is tracking. Anyway, to enhanced the precision of the drone, i decided to derive the velocity on all the axes from the position measurements and pass them to the firmware with all the other data;

- With the second setup, the messages that contain the data are a defined amount of bytes directly send to the drone over UDP. In this case, the node has to read the incoming messages and translate the bytes. The message handling procedure is shown in Algorithm 5, this procedure acquires the incoming messages, one by one, with the *recvfrom()* function, then, knowing the structure of the message, translates each sequence of 4 bytes as a single *double* number. After the conversion of the data, they are copied in the PX4 message and sent to the firmware. This procedure is the callback of a 10 ms timer, this high frequency prevents the formation of a message queue and, thanks to the blocking function *recvfrom()*, the entire procedure does not overload the board.
  In this case, only the vertical velocity is calculated (the one obtained from VICON) since the estimation error of the UWB technology for the horizontal position is too large to obtain a precise and useful value.

Furthermore, both nodes apply some changes to the incoming data. The firmware requires all the input measures to be with respect to a NED (north, east, down) reference frame, but the systems used measure the odometry of the object with respect to its own reference frame, which is an ENU (earth, north, up) reference frame. To resolve this mismatch, it is necessary to transform both the position coordinates and the orientation measures. To transform the position it is necessary to invert the Y and Z position coordinates, the orientation needs to be pre- and post-multiplied by the quaternion that corresponds to a rotation of 180 degrees

on the X axis; the result of this operation is equal to an inversion of the Y and Z components of the quaternion given by the MoCap system. The two implemented coordinate publisher nodes execute these transform operations during the extraction of the data.

Algorithm 4 Extracts the data from the VICON data stream and sends them to the firmware. This algorithm is a function of an object, all the variables used are created and initialized inside the initializer of the object.

```
1: procedure StreamReading
2:     ▷ msg contains the information about the position and orientation of the drone
3:     ▷ Execution
4:     subj ← GetSubjectName()                          ▷ Obtain the object identifier
5:     seg ← GetSegmentName()  ▷ Obtain the identifier of the segment of the object
6:     t ← now()
7:     time_delay ← t − old_time
8:     old_time ← t                                          ▷ Get the timestamp
9:     pos ← GetSegmentGlobalTranslation(subj, seg)
10:    move ← pos − old_pos
11:    old_pos ← pos                                          ▷ Get the position
12:    q ← GetSegmentGlobalRotationQuaternion(subj, seg)      ▷ Get the orientation
13:    vel ← move/time_delay                                 ▷ Compute the velocity
14:    ▷ Copy the information in the msg
15:    msg ← t
16:    msg ← pos
17:    msg ← q
18:    msg ← vel
19:    send msg
20:    return 0
21: end procedure
```

3.2.3 Command publisher

The command publisher is the simplest ROS2 node among all in this architecture and it is the only one that runs outside of the drone, its role is to send to the drone a message containing the user command to be executed. The command publisher directly communicates with the *director* node through an ROS2 topic, sending over that topic a message containing the word that identifies the desired command and,

Algorithm 5 Reads the UDP message received from the central unit. This algorithm is a function of an object, all the variables used are created and initialized inside the initializer of the object.

```
1: procedure MSG HANDLER(sd)
2:     ▷ sd contains all the information about the UDP communication
3:     ▷ msg contains the information about the position and orientation of the drone
4:     ▷ Execution
5:     bytes ← recvfrom(ip)  ▷ Stores the bytes received from the UWB central unit.
6:     t ← now()
7:     time_delay ← t − old_time
8:     old_time ← t                                          ▷ Get the timestamp
9:     pos ← read_bytes(bytes)
10:    move ← pos[2] − old_pos_z
11:    old_pos_z ← pos[2]                                    ▷ Get the position
12:    q ← read_bytes(bytes)                                 ▷ Get the orientation
13:    vel ← move/time_delay                                 ▷ Compute the velocity
14:    ▷ Copy the information in the msg
15:    msg ← t
16:    msg ← pos
17:    msg ← q
18:    msg ← vel
19:    send msg
20:    return 0
21: end procedure
```

based on the specific command, it can also contain up to three numbers that can be coordinates or the yaw angle. Since this work has been developed in a research environment, this node checks only if the inserted command is valid, all other data inserted are sent as they are written, the *director* node has the duty to translate the string received into a proper command.

3.2.4 Director node

The director node is the most important part of the architecture; it is the core of the entire application. Its role is to translate the strings received from the *command publisher* into a proper command and ensure its execution. As mentioned above, communication between the *director* node and the firmware is obtained through ROS2, this means that every action we want to perform corresponds to a specific

message to be sent to a specific ROS2 topic.

The commands available at the moment are:

- **start**: PX4 needs to receive an *OffboardControlMode* message at 2 hz as proof that the external controller is healthy, without it, it is not possible to set PX4 in the offboard mode and externally control it (the firmware requires to receive this message for at least 1 second to execute any external command), this message also defines the kind of control the firmware has to perform, since the focus of this work is to make a working application and not an optimal one, i chose to perform the position control, the simplest. The *start* command initiates the transmission of the required healthy message, since the 2 hz required are a lower bound, i decided to transmit the message with a frequency of 5 hz to cover possible delays or loss of messages. This has to be the first command to be executed to enable the control of the drone;

- **offboard**: set the firmware to offboard mode. The request is made sending a *VehicleCommand* message (parameters: 176, 1, 6). This request message doesn't work if it is sent before one second from the start of the healthy message stream;

- **arm**: activates the drone rotors. This is done by publishing a *vehicle command* message with the arming request (parameters: 400,1). This command can be executed only after at least one second from the start of the healthy messages stream and if the preflight checks performed by the firmware give positive results;

- **go**: this command sets a new destination for the drone by sending a *TrajectorySetpoint* message with the desired coordinates. To enhance the positional precision of the drone, the position is sent every 200 ms, the stream of messages stops if the drone is disarmed, killed or is landing. This command can be executed only if the drone is armed;

- **wait**: stops the drone at the current position. Sends a *TrajectorySetpoint* message with the coordinates of the drone when the command is received. The message is sent only if the vehicle is moving;

- **yaw**: change the yaw angle of the drone. Sends a *TrajectorySetpoint* message with the current position of the drone and the desired yaw angle. This command can be sent only if the drone is armed and still:

- **k** or **kill**: stops the rotors of the drone and the stream of healthy messages, this message is developed to guarantee the safety of the drone or the environment. This command sends a *VehicleCommand* message to the firmware requiring

the immediate disarm of the vehicle (parameters: 400, 0 and 21196), the difference with the *disarm* message is the addition of a third parameter, this parameter tells the firmware to bypass any flight check and execute the disarm immediately;

- **land**: land the drone. To make a safe landing i call the predefined maneuver saved in the firmware. The maneuver is called by sending a *VehicleCommand* message with the landing request (parameter: 21). After the landing, the drone is disarmed;

- **disarm**: stops the drone rotors. For safety reason, this command can be executed only if the firmware detects that it is on the ground. As for the *arm* command, the desired behavior is obtained publishing a *VehicleCommand* with the disarming request (parameters: 400 and 0);

- **mission**: make the drone follow a predefined path by sending a sequence of coordinates and orientations to it. This command automatically send several trajectory point to the firmware, the new point is sent after the previous one is considered reached, a point is considered reached if the drone stays within an acceptance radius around it with the desired orientation for a given amount of time. Every set of coordinates is sent with a *TrajectorySetpoint* message, like the *go* and *yaw* commands. This command can be executed only if the drone is armed. More details on the path followed are available in Section **??**;

- **stop**: immediately land the drone, stops all streams of messages and resets all the variables of the node.

The *director* node has to execute the above commands as explained if the conditions of the specific command are satisfied and verify the correct behavior of the drone. To verify if the conditions of a command are satisfied, the *director* node has to have information about the state and odometry of the drone, the firmware gives all the necessary data about the current odometry of the vehicle but does not give enough information about its state, the node needs to know the maneuvers the drone is performing while the firmware *VehicleStatus* messages contains only the arming state and the flight mode of the vehicle. To address this problem, the *director* node implements a state-space machine with five states:

- **disarm**: this state implies that the drone is on the ground with the rotors still and is ready to be controlled, the only way to exit from this state is through the *arm* command. This is the starting state of the application;

- **hold**: in this state the drone is armed but steady in its current position, both on the ground or in the air;

- **move**: the drone is moving, this state includes both a rotation on itself and a shift to reach a position. When the desired orientation and/or position is reached, the state return to be *hold*;
- **landing**: this state means that the drone is executing a landing maneuver. In this state, is not possible to execute any command until the drone is correctly landed and disarmed;
- **kill**: entering this state means that something went wrong and the drone had to be forcefully stopped. From this state, it is necessary to repeat the starting procedure (*start* and then *arm* command) to control the drone again.

Figure 3.3 shows the implemented state-space machine with the necessary commands to pass from one state to another. The *start* and *stop* commands activate and deactivate the machine, respectively. The arrows without a command represent a forced state change; when the maneuver characterized by that state ends, the state is automatically changed.

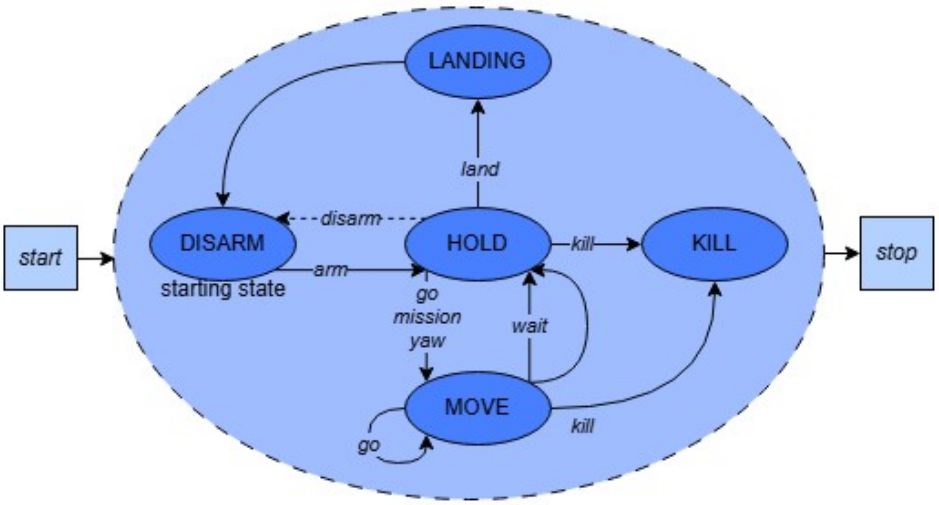

Figure 3.3: State machine of the director node

3.2.5 PX4

The last component of the architecture is the vehicle firmware. During the implementation of the architecture, i did not modify any part of its code, i changed its editable parameters to make it better fit the purpose of the application and the vehicle used. In addition to the setup mentioned earlier in this chapter (Section

3.1.2), i performed a PID tuning to increase drone precision and set the external vision source as the most important for data fusion. The main focus of the PID tuning was to reduce overshooting in the horizontal movement to prevent possible collisions with the environment.

Chapter 4

Results

The focus of this work was to implement an architecture that would allow the execution of a controlled flight with a drone without the use of a GPS. To test the application, i developed a small mission that uses most of the commands implemented and analyzed the performance. All tests were conducted in the same space, but, due to the different precision between the two sources of position, the missions have slightly different waypoints to maintain the safety of the drone. Even though the waypoints are different, the structure of the mission is the same, the drone has to reach five waypoints at a given altitude and then reach other five waypoints at a different altitude, the first five points have to be reached moving forward while the remaining moving sideways, after the tenth point the drone lands. A waypoint, which is composed by the X,Y and Z coordinates of the destination and the yaw angle to have there, is considered reached if the drone position and yaw angle are inside an acceptance radius for two seconds (the acceptance radii are checked separately, creating a "cube" of acceptance instead of a sphere), the time requirement is used to analyze the capacity of the drone to maintain the position. Between two different waypoints, the vehicle has to match the destination yaw angle before moving there (after this change of orientation, the check to verify that the position has been reached is repeated), in this way we obtain more control over the flight, but the time needed to complete the mission increases.

The data shown in this chapter were taken from the messages exchanged between the ROS2 nodes inside the drone and the firmware; the collection has been carried out by the system where the *command publisher* was running.

4.1 Physical environment

All tests were performed indoors at the LINKS foundation. The area used for flight is a cage 3.5 x 7.5 x 2.5 prepared to use the MoCap system and the UWB

technology (Figure 4.1).

The MoCap system is enabled by the installation of six cameras on the cage scaffold, four of them are positioned on the four upper corners while the remaining two are set in the middle of the long sides of the structure, all cameras are at the same height and oriented to cover all the space inside the cage. With this camera disposal, we obtain a working area of dimensions 2.5 x 4.5 x 2 centered at the center of the cage, inside this area, every move and orientation of the drone is tracked by the system, with any (or, at least, not remarkable) loss of position.

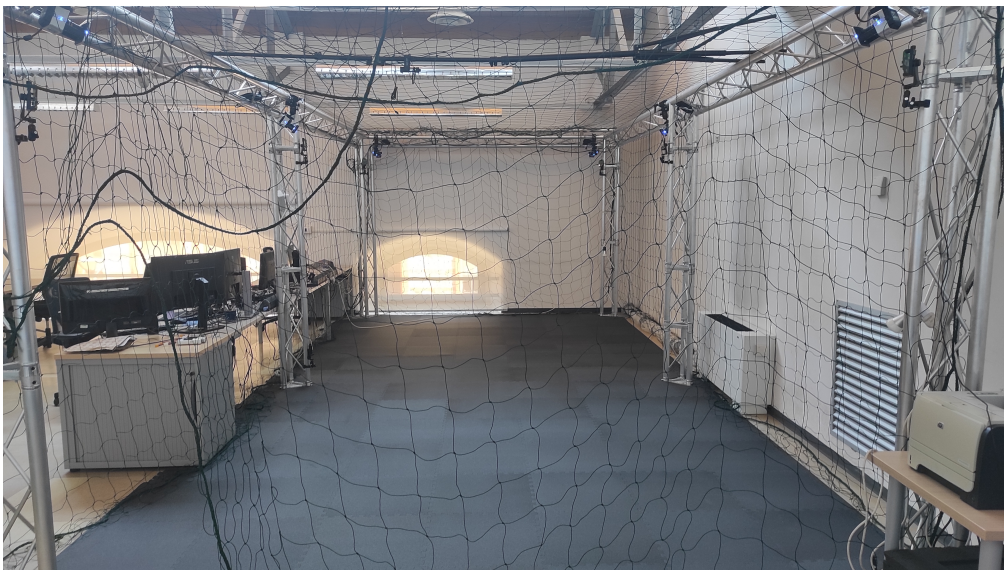

Figure 4.1: Area used for all tests

To enable the VICON tracker to recognize the vehicle, the drone has been equipped with four markers (shown in Figure 4.2), their position is extremely important for the definition of the working area, after several tests we discovered that with this disposal we obtain the biggest working area, the three markers on the top of the drone help to recognize the drone at lower altitude while the fourth marker is extremely important to identify the orientation of the drone and not lose track of it at higher altitude. After recognition, the VICON tracker continuously computes the vehicle odometry and sends it over the network as a broadcast message with a frequency of 100 hz.

The UWB setup is enabled by using six anchors (shown in Figure 4.3) mounted right below the cameras for the MoCap System at slightly different heights. The UWB technology does not have the limits of the tracker in terms of area covered, but, since this setup needs part of the measurements of the tracker, the working area is the same as the other setup, slightly reduced to avoid critical areas where are present communication delays too high to maintain a position.

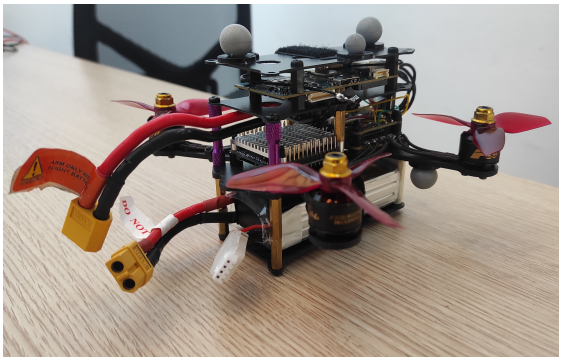

Figure 4.2: UAV used for all tests

In this setup, the drone is equipped with a tag (Figure 4.4), a small board that, combined with the anchors, enables the measurement technique explained in Section 2.5 to obtain position information with the UWB technology. All anchors communicate the measured ranges to each other and one anchor is connected to a server through a gateway, this gateway extracts all the measurements received from the anchors and sends them to the server, this server combines the measurement received to obtain an estimate of the X and Y positions of the tag. After the computation, the server extracts the Z position and orientation of the vehicle from the VICON data stream to obtain all the necessary odometry values, and, as last step, it sends all the data to the drone with a single UDP message. This message is sent at a rate of 40 hz.

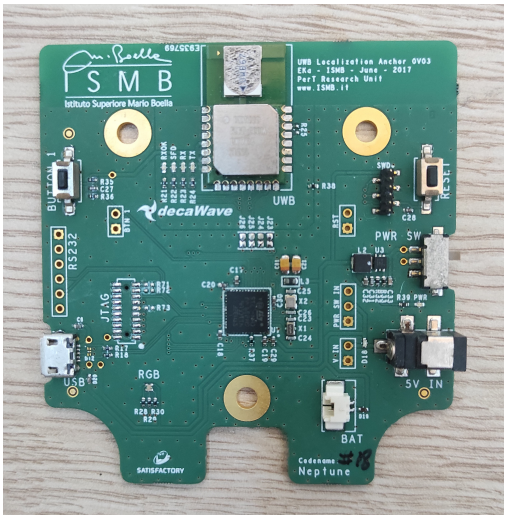

Figure 4.3: UWB anchor

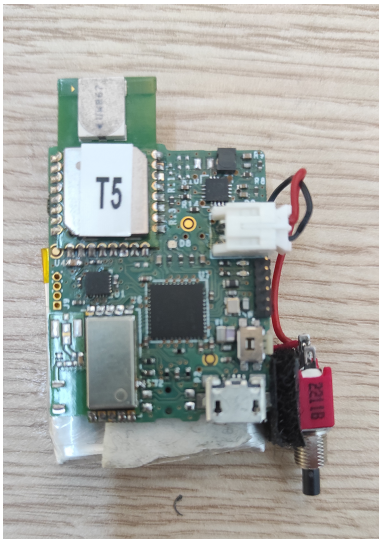

Figure 4.4: UWB tag

4.2 Test with VICON

The waypoints of this mission are shown in Table 4.1, and the desired yaw angles for each waypoint are shown in Table 4.2, for this setup the acceptance ranges are: [± 10] cm for the X coordinate, [± 10] cm for the Y coordinate, [+5, -10] cm for the Z coordinate and ± 0.03rad for the yaw angle. The initial position and orientation of the drone are not relevant for the purpose and success of the mission.

	1	2	3	4	5	6	7	8	9	10
X	0	-1.2	0.6	0.6	-1.2	0.6	-1.2	0.6	-1.2	0
Y	0	-1.8	-1.8	1.8	1.8	-1.8	-1.8	1.2	1.2	0
Z	0.9	0.9	0.9	0.9	0.9	1.7	1.7	1.7	1.7	1.7

Table 4.1: Waypoints of the mission with the MoCap system. Measured in meters in an ENU reference frame.

	1	2	3	4	5	6	7	8	9	10
Yaw angles	3.14	2.15	0	-1.57	3.14	1.10	1.57	-2.60	-1.57	2.35

Table 4.2: Yaw angles of the mission with the MoCap system. Measured in radiant in an NED reference frame.

The results of this test are shown in Figures 4.5, 4.6, 4.7 and 4.8. As you can see, the mission was a success. The spatial performance of the system is illustrated in the 3D trajectory graph (Figure 4.5), where the drone path (blue line) intersects all 11 predefined waypoints (red dots) with high positional precision, showing the vehicle's ability to operate in three-dimensional environments. The corresponding position plots on the X, Y and Z axes (Figure 4.6) show that the vehicle follows the reference trajectory with small errors over time. The controller exhibits minimal steady-state error and rapid convergence on all axes with a focus on altitude, the initial overshoot in the altitude position is due to the dynamic computation of the weight of the drone, the firmware requires a few moments to correctly estimate the right amount of force to apply to precisely execute aerial maneuvers. The transition from a waypoint to an other is quite low, as the horizontal velocity of the vehicle has been limited for safety reasons.

The attitude response, shown through the evolution of roll (ϕ), pitch (θ) and yaw (ψ) angles over time (Figure 4.7), reveals that the system maintains a consistent angular stability throughout the mission. Both roll and pitch remain around zero, with transient oscillations during movement, reflecting good stability. The yaw angle shows distinct step transitions corresponding to the planned heading changes between the waypoints. These precise changes indicate that the system can change its orientation without a significant delay or overshoot. The angular velocities

profiles (Ωx, Ωy, Ωz shown in Figure 4.8) further confirm the responsiveness of the system, with variation in rotational rates during trajectory transitions.

Overall, the combined analysis of attitude, angular velocity, position trajectories and spatial path visualization confirms the effectiveness of the proposed architecture and control system. The platform demonstrates precise waypoint navigation, dynamic stability and accurate trajectory tracking in all degrees of freedom. These results validate the reliability and performance of the control architecture under realistic conditions, supporting its suitability for complex autonomous missions.

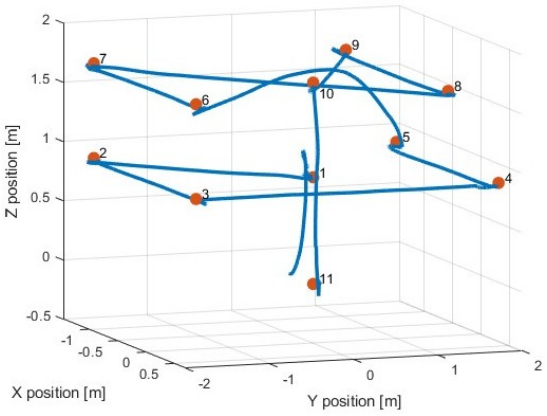

Figure 4.5: Mission path executed by the drone using the MoCap system.

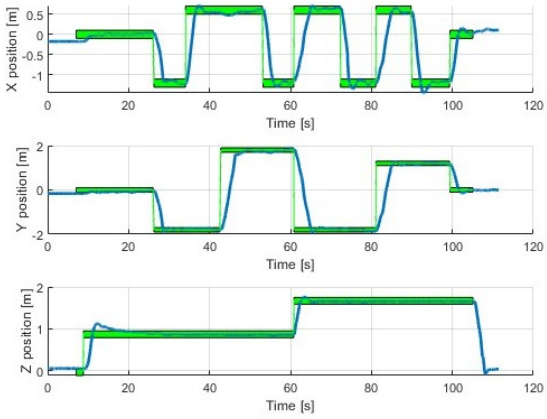

Figure 4.6: X,Y and Z coordinates of the drone, in meters, during the mission with the MoCap system

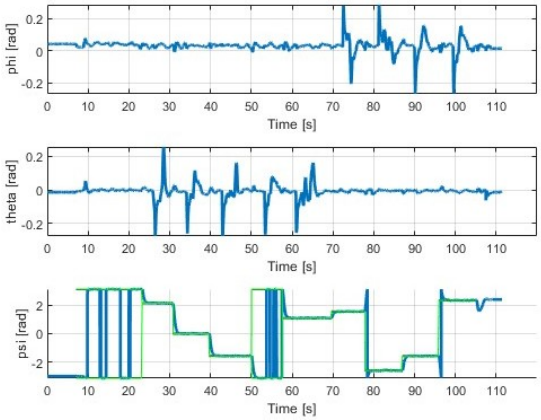

Figure 4.7: Euler angles of the drone, in radians, during the mission with the MoCap system

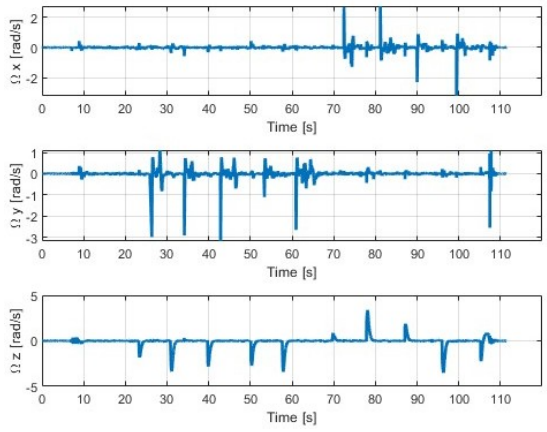

Figure 4.8: Angular velocities of the drone, in radians over seconds, during the mission with the MoCap system

4.3 Tests with UWB

For the mission in this environment i have slightly approached the waypoints to cope with the inaccuracy of the UWB technology, the new waypoints are shown in Table 4.3 and with the position i also changed the desired yaw angles to maintain the structure of the mission, the new ones are shown in Table 4.4. For this environment, also the acceptance ranges needed a modification to cope with the lower precision of the localization system, the new ones are: [± 25] cm for the X coordinate, [± 25] cm for the Y coordinate, [+5,-10] cm for the Z coordinate and [± 0.03] rad for the yaw angle. The initial position and orientation of the drone are not relevant for the purpose and success of the mission.

	1	2	3	4	5	6	7	8	9	10
X	0	-0.9	0.3	0.3	-0.9	0.3	-0.9	0.3	-0.9	0
Y	0	-1.2	-1.2	1.2	1.2	-1.2	-1.2	1.2	1.2	0
Z	0.9	0.9	0.9	0.9	0.9	1.7	1.7	1.7	1.7	1.7

Table 4.3: Waypoints of the mission with the UWB technology. Measured in meters in an ENU reference frame.

	1	2	3	4	5	6	7	8	9	10
Yaw angles	3.14	2.21	0	-1.57	3.14	1.11	1.57	-2.68	-1.57	2.5

Table 4.4: Yaw angles of the mission with the UWB technology. Measured in radiant in an NED reference frame.

The results of the mission are shown in Figures 4.9, 4.10, 4.11, and 4.12. This time, the mission has been completed with some difficulties. The 3D trajectory plot (Figure 4.9) illustrates the system's capabilities to reach all the predefined waypoints, although the actual path (blue) displays more curvature and loop compared to the lines seen in the other experiment. These deviations and overshoots are caused by inaccuracies and inconsistencies in the position given by the UWB technology, which is extremely visible during the landing maneuver, but the actual position remains within the tolerance band (Figure 4.10).

The attitude plots (ϕ, θ and ψ, shown in Figure 4.11) show that the system maintains roll and pitch angles within a bound of ±0.2 radians, with higher noise levels and fluctuations compared to the previous experiment, particularly in the pitch channel. These transient deviations are caused by rapid changes in the UWB position measurement. Despite the noisy roll and pitch, the yaw angle continues to precisely follow the command reference values. This behavior is mirrored in the angular velocity graphs (Figure 4.12), where the Ωx and Ωy channels exhibit a less stable behavior than before, the Ωz graph is comparable to previous results.

All graphs show that this technology does not affect the movement of the drone but negatively affects its capacity to maintain the position. Despite the increase of the acceptance radii and the reduction in the distance between waypoints, the mission lasted longer than the MoCap one. It is important to observe that the positional error of this technology is not proportional to the environment and is accentuated by the dimension of the mission, with larger mission it can be negligible.

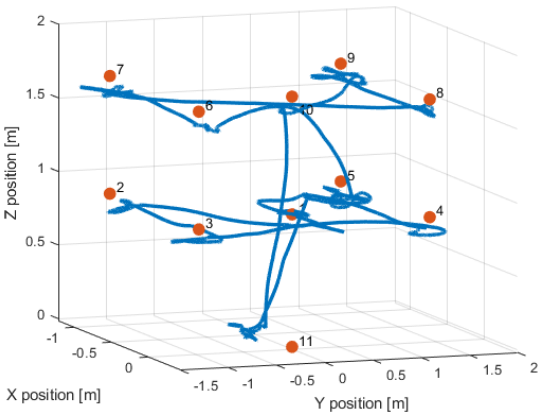

Figure 4.9: Mission path executed by the drone using the UWB technology.

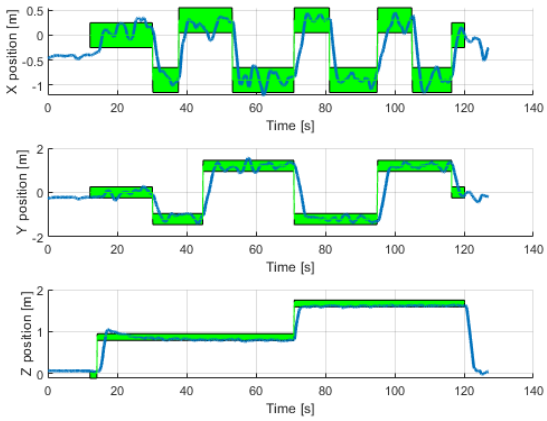

Figure 4.10: X,Y and Z coordinates of the drone, in meters, during the mission with the UWB technology

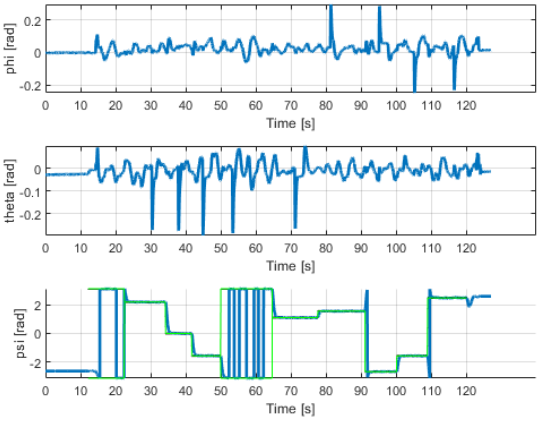

Figure 4.11: Euler angles of the drone, in radians, during the mission with the UWB technology

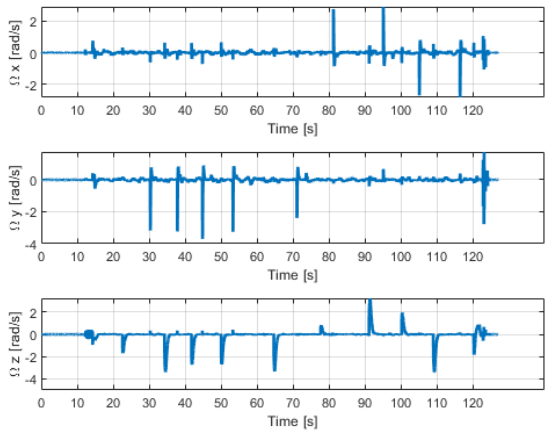

Figure 4.12: Angular velocities of the drone, in radians over seconds, during the mission with the UWB technology

4.4 Discussion

The performance presented earlier are direct results of the implementation carried out during this thesis. These results were achieved by realizing the system architecture illustrated in Figure 4.13. As detailed in Chapter 3, the implementation of this architecture required several key developments:

- The design and deployment of the *Voxl-parser* service, which enables Micro XRCE-DDS communication between the companion board and the flight controller firmware;
- The implementation of three ROS2 nodes (*Coordinate publisher*, *Director* and *Command publisher*) which constitute the core of the communication network and encapsulate the application logic;
- Modify the firmware to cope with the structure of the drone and the specific purpose of the application;

To streamline the operation of the system, all components that are executed onboard the drone, both on the companion board and the flight controller, were configured to start automatically upon system boot.

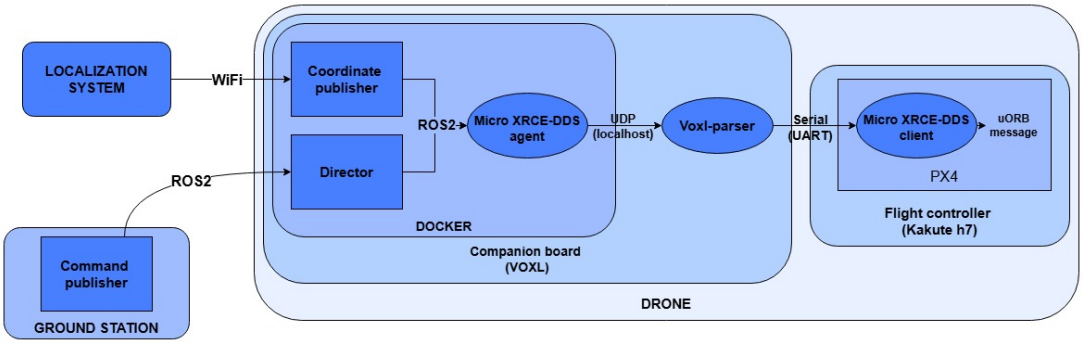

Figure 4.13: Complete architecture

The development of the application was guided by a series of tests conducted in simulated and real world environments. The simulated environment served primarily to support an initial understanding of the firmware and to facilitate the early stages of the application logic design. However, due to discrepancies between the simulation and the actual use case, real-world testing was essential for refining the application and obtaining accurate feedback on its performance.

Although the implementation phase consumed most of the project timeline, it was relatively straightforward in terms of technical obstacles. The most complex

aspect involved the development of the *Voxl-parser* component, which was necessary to establish a robust communication link between the companion computer and the flight controller. This task posed a particular challenge, as it required familiarity with technologies previously unexplored during the course of this work: the Micro XRCE-DDS communication protocol and UART serial communication.

In contrast to the implementation phase, the testing phase was characterized by a series of technical difficulties which, although not critical, contributed to a slowdown in the overall development process:

- **Slow deployment times**: the creation and upload of the Docker image to the companion board proved to be significantly time-consuming. Despite various optimization attempts, each update to the ROS2 nodes required approximately thirty minutes to be deployed to the vehicle.
- **Latency in localization communication**: the communication between the localization system and the firmware suffered from noticeable delays, resulting in a wavelike oscillation in the drone's motion. This behavior stemmed from a mismatch between the drone's actual position and the position data received from the localization system. Consequently, the firmware would attempt to correct a position the drone had already moved beyond, creating a continuous cycle of overcompensation. This issue was particularly evident in the Ultra-Wideband (UWB) environment and was successfully mitigated by directly connecting the UWB server to the network router, thereby reducing latency in IP-based communication, which was the primary source of the problem;
- **Inconsistent behavior on the companion board**: some inconsistencies were observed in the execution of the software on the companion board. Although the application was configured for automatic startup at boot, certain components, particularly the Voxl-parser, occasionally failed to launch or exhibited significant startup delays. Unfortunately, these issues occurred sporadically and could not be reproduced systematically, making cause analysis difficult. However, their infrequent nature and the possibility to resolve them manually ensured that they did not significantly slow the project's progress.

Despite these challenges and the associated delays, the main objective of this thesis was achieved successfully. However, several aspects of the application could be further refined or extended to enhance performance and system reliability, such as:

- Resolve the previously explained startup inconsistencies observed in the companion board software;
- Conducting more accurate tuning of firmware parameters, which was limited during development due to connectivity issues and frequent hardware changes;

- Enabling drone control through velocity commands, in addition to position-based control, to allow for smoother and more responsive flight behavior;
- Integrating an Inertial Measurement Unit (IMU) and improving the UWB localization system to estimate altitude (z-axis) as well. These additions would allow the drone to operate independently of an external motion capture (mocap) and enable fully autonomous operation in environments where external tracking systems are not available.

Chapter 5

Conclusion

This thesis presented the development and implementation of an application that allows an unmanned aerial vehicle (UAV) to perform controlled indoor flight, with a focus on remote operation and flexibility in the system architecture. The primary objective was to design and validate a functional framework rather than optimize for peak performance, yet the experimental results were highly satisfactory under different test conditions. The UAV demonstrated the ability to perform stable and precise maneuvers in a three-dimensional space, even with varying degrees of localization accuracy.

A key insight from this work is the strong dependency of UAV performance on the quality of position and orientation data. The system integrated two external localization methods: the VICON motion capture system and Ultra-Wideband (UWB) technology. The VICON-based setup delivered exceptional results, with the UAV maintaining stable flight, accurate trajectory tracking and smooth transitions, ideal for operations requiring high precision and the possibility to install high definition cameras to obtain a good cover of the operational area such as industrial inspection, warehouse navigation or research applications. In contrast, the UWB-based localization, though less precise and less stable than the MoCap one, still enabled mission completion. This suggests that UWB technology can be a viable alternative for tasks where ultra-high accuracy is not essential and have maneuvering spaces large enough, such as urban monitoring, environmental exploration or initial mapping in large indoor environments.

The flexibility and modularity of the system also highlight its potential for adaptation to various applications and environments. Future work could focus on optimizing control algorithms and improving the robustness of communication between nodes, potentially incorporating onboard localization fusion techniques to compensate for lower-quality external data sources. Furthermore, integrating real-time obstacle avoidance and dynamic path planning would expand the applicability of the system to more complex and unstructured environments.

In conclusion, this thesis demonstrates that a UAV can be effectively controlled indoors using a versatile architecture that supports multiple localization systems. While high-precision systems like VICON yield superior results, cost-effective alternatives such as UWB provide a functional solution for less demanding scenarios. These findings contribute to the broader field of UAV autonomy by emphasizing the trade-offs between performance and practicality in localization-driven control strategies.

Bibliography

[1] Dirk Merkel. «Docker: lightweight linux containers for consistent development and deployment». In: *Linux journal* 2014.239 (2014), p. 2 (cit. on p. 3).

[2] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. «Robot Operating System 2: Design, architecture, and uses in the wild». In: *Science Robotics* 7.66 (2022), eabm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074 (cit. on p. 5).

[3] *ROS2 Humble online documentation*. https://docs.ros.org/en/humble/index.html. Accessed: 2025-02-12 (cit. on pp. 5, 7).

[4] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. «PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms». In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 6235–6240. DOI: 10.1109/ICRA.2015.7140074 (cit. on p. 8).

[5] *PX4 online documentation*. https://docs.px4.io/main/en/index.html. Accessed: 2025-02-12 (cit. on pp. 8–13, 22).

[6] Mauricio Caceres. *Kalman Filter Applications in Localization Algorithms for Wireless Sensor Networks*. Tech. rep. Radio-Mobile Technologies Lab, Sept. 2007 (cit. on p. 9).

[7] *MicroXRCE-DDS Documentation*. eProsima, Inc. 2024. URL: https://micro-xrce-dds.docs.eprosima.com/_/downloads/en/latest/pdf/ (cit. on p. 10).

[8] *DDS for eXtremely Resource Constrained Environments*. Object Management Group. 2020. URL: https://www.omg.org/spec/DDS-XRCE/1.0/PDF (cit. on p. 10).

[9] Dario Brescianini, Markus Hehn, and Raffaello D'Andrea. *Nonlinear Quadrocopter Attitude Control*. Tech. rep. ETH Z¨urich, Oct. 2013 (cit. on p. 12).

[10] Sinan Gezici, Zhi Tian, Georgios B. Giannakis, Hisashi Kobayashi, Andreas F. Molisch, H. Vincent Poor, and Zafer Sahinoglu. «Localization via Ultra-Wideband Radios». In: *IEEE signal processing magazine* (July 2005), pp. 70–84 (cit. on pp. 16, 17).

[11] *The implementation of two-way ranging with the DW1000*. Qorvo, Inc. 2024. URL: `https://www.qorvo.com/products/d/da008448` (cit. on p. 18).

[12] *Kakute H7 v1.3 (MPU6000)*. `https://holybro.com/products/kakute-h7`. Accessed: 2025-02-14. Holybro, Inc. (cit. on p. 19).

[13] *ModalAi tecnical documentation*. `https://docs.modalai.com/`. Accessed: 2025-02-14. ModalAI, Inc. (cit. on p. 19).