

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



ROS2-based Architecture for MAV Data Sensing

Rui Miguel Santos Carvalho

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Professor Pedro Santos

Co-Supervisor: Eng. Ricardo Faria

October 29, 2021

Resumo

Os *Micro Aerial Vehicles* (MAV) são uma plataforma flexível e de baixo-custo para executar medições sensoriais. A sua pequena escala e alta mobilidade em um espaço tridimensional tornam-nos particularmente eficientes em cenários *indoor*. No entanto, a sua integração em arquiteturas industriais, maiores e mais complexas, pode causar o que se pode denominar de "um desafio de interoperabilidade" devido aos diferentes protocolos de comunicação usados e à natureza crítica com poucos recursos dos MAVs.

Nesta dissertação, propõe-se um arquitectura de ponta a ponta baseada na *framework Robot Operating System 2* (ROS2) para resolver este desafio. Neste contexto, como os MAVs utilizados não são computacionalmente capazes de correr a versão *Desktop* do ROS2, utilizou-se o micro-ROS, uma versão reduzida do ROS2 desenvolvida com o intuito de ser utilizada em sistemas embarcados com baixos recursos. Esta funcionalidade é possibilitada pela utilização do eProxima Micro XRCE-DDS, um *middleware* que permite a fácil integração destes dispositivos no mundo ROS2/DDS e possibilita a comunicação e integração com outros dispositivos ROS2 incluindo uma *ground station*, outros sensores e até nós remotos em serviços *Cloud*. Adicionalmente, um agente Micro XRCE-DDS customizado foi desenvolvido para a utilização do CRTP, o protocolo de transporte proprietário utilizado pelos MAVs escolhidos para este projeto, os Crazyflie 2.1.

Como caso de uso exemplo, integrou-se um sensor de qualidade do ar a bordo do MAV, de forma a executar medições em pontos específicos, podendo ser utilizado para a monitorização ou localização de substancias nefastas no ar. Adicionalmente, desenvolveu-se um algoritmo de planeamento de missões baseado no problema do *Multiple Depot Multiple Travelling Salesmen* que permite a optimização do tempo total da missão através de não só a distancia entre pontos mas também o tempo necessário para fazer a medição.

A validação da arquitectura foi feita através de testes reais na "arena de teste" que mostraram que os MAVs são capazes de executar missões com sucesso enquanto publicam medições e interagem com outros sistemas através de ROS2. Além disso, uma avaliação do sistema de comunicação foi levada a cabo que mostrou que apesar da utilização do micro-ROS e o agente customizado poder causar um acréscimo considerável a nível de latência, os resultados mostram que a *framework* ainda pode ser viável para aplicações sensoriais e a interoperabilidade e flexibilidade que providencia torna-a uma framework poderosa em sistemas sensoriais robóticos embarcados.

Palavras-chave— ROS2, DDS, IIoT, UAV, micro-ROS, robótica, interoperabilidade, sensores aéreos, qualidade do ar, computação na nuvem.

Abstract

Micro Aerial Vehicles (MAV) are a flexible and low-cost platform for performing airborne sensor measurements. Their small size and high maneuverability in tri-dimensional space makes them particularly efficient in indoor scenarios. However, integration of said devices in larger, more complex Industrial Internet of Things (IIoT) architectures can cause what can be deemed as an "interoperability challenge" due to the different communication protocols used in these systems and the critical resource-constrained nature of MAVs.

In this dissertation, we propose an end-to-end sensing architecture based on the Robot Operating System 2 (ROS2) framework to solve this challenge. In this regard, as the MAVs are not capable of running the desktop version of ROS2, we use micro-ROS, a stripped-down version of ROS2 which was specifically designed to run on resource-constrained devices by using eProsima Micro XRCE-DDS as its middleware. The usage of this middleware allows the MAVs to be seamlessly integrated in the ROS2/DDS world and therefore communicate and cooperate with other ROS2-enabled devices including a ground station, other sensors and even cloud-based nodes. Furthermore, a custom Micro XRCE-DDS agent was developed to use CRTP, the proprietary transport protocol of the MAVs used, the Crazyflies 2.1 quadcopters.

As an example use-case we have integrated an Indoor Air Quality (IAQ) sensor on-board the MAVs, to perform measurements in user-provided waypoints which can be used for both monitoring and localization of airborne contaminants. Additionally, we have developed a mission-planning algorithm based on the Multiple Depot Multiple Travelling Salesmen problem that can optimize the total mission time by taking into account not only the distance between sensing waypoints but also the time needed to perform the measurement.

Validation of the architecture was done by performing real-world experiments in a "test arena" which showed that the MAVs were capable of successfully performing sensing missions and interact with other devices by leveraging the ROS2 framework. Furthermore, we have performed a preliminary evaluation of the communication performance which showed that although the usage of micro-ROS and our custom agent can cause a considerable latency overhead, the framework can still be viable for sensing applications and the interoperability and flexibility it offers make it a powerful framework for embedded robotic sensing systems.

Palavras-chave— ROS2, DDS, IIoT, UAV, micro-ROS, embedded robotics, interoperability, data sensing, indoor air quality, cloud-computing.

Acknowledgements

First and foremost, I would like to thank my advisor Professor Pedro Santos for all the support and feedback given during the this thesis, availability and understanding.

To Dr. Patrick Neumann from Bundesanstalt für Materialforschung und -prüfung (BAM) for sharing his SGP30 Crazyflie deck driver and to Pablo, Antonio, Maria and everyone from eProxima for the support in micro-ROS related troubleshooting.

To all the people from Capgemini Engineering, especially my co-supervisor Ricardo Faria for all the support and attention given, to my brother-in-arms and author of the integrated collision avoidance system Matheus. To my colleagues, Diana, Jorge and the rest of the Research and Innovation (R&I) team for all the enthusiasm and support and to Margarida for helping out with video editing for the demos. Last but not least, to Eduardo Mendes from Connect Robotics for introducing me to the world of aerial robotics.

Rui Miguel Carvalho

*“So that was our life, and it seemed like nothing would ever happen to change it.
You let your guard down, almost, and life comes running at you on your blind side.
Because life is nothing but change, even when it seems to stand still.
Standing still is a human thing, like a defiance we throw,
but we can never do it for long ”*

M. R. Carey (The Book of Koli)

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	2
1.3	Contributions	2
1.4	Dissertation Structure	3
2	Literature Review	5
2.1	Research methodology	5
2.2	Technical Background	6
2.2.1	ROS Overview	6
2.2.2	micro-ROS	10
2.2.3	Micro Aerial Vehicles	11
2.2.4	Cloud Robotics	12
2.2.5	Real-Time Systems	13
2.3	Related Work	14
2.3.1	Evaluation of ROS2	14
2.3.2	Summary	18
2.3.3	UAV-assisted IoT	20
2.3.4	IAQ sensing using UAVs	21
2.3.5	Cloud-enabled UAV applications	22
2.4	Main Takeaways	23
3	MAV Data Sensing using ROS2	25
3.1	System Description	25
3.2	Description of System Technologies	27
3.2.1	The Crazyflie Ecosystem	28
3.2.2	Additional tools and frameworks	31
3.2.3	Test Arena	32
3.3	Summary	33
4	Implementation	35
4.1	Communication Architecture	35
4.1.1	GS - MAV Communications	36
4.1.2	Custom CRTP Micro XRCE-DDS Agent	36
4.2	Motion control and plan execution	40
4.2.1	Waypoint-based IAQ sensing	40
4.2.2	Collision Avoidance System	41
4.3	Trajectory and mission planning	42

4.3.1	Optimal sensing trajectories	42
4.3.2	Auction-based algorithm for event-triggered (alarms) sensing	47
4.4	On-board IAQ sensing	50
4.4.1	Hardware	50
4.4.2	Firmware Integration	51
4.5	Cloud Services	51
4.5.1	Integration Service and cloud-based planning	51
4.5.2	AWS Cloudwatch monitoring	53
4.6	Visualization	55
4.7	Summary	56
5	Results and Discussion	59
5.1	Planning	59
5.1.1	Produced Mission Trajectories	59
5.1.2	Planner performance	61
5.2	Flight Tests	64
5.2.1	Test 1: Static Mission Execution	64
5.2.2	Test 2: Mission with alarms	65
5.2.3	Test 3: IAQ Sensing	66
5.2.4	Test 4: Battery consumption	67
5.3	Communications	68
5.3.1	GS-MAV communication	68
5.3.2	End-to-end latency	72
5.4	Summary	74
6	Conclusions and Future Work	75
6.1	Conclusions	75
6.2	Limitations and Future Work	76
A	Custom micro-ROS/ROS2 Interfaces and configuration files	77
A.1	Full Service/Topic Listing	77
A.1.1	MAV	77
A.1.2	RQT Plugin (GUI)	78
A.1.3	Planner	79
A.1.4	RVIZ	79
A.1.5	Alarm node	80
A.1.6	Task Manager	80
A.1.7	Auctioneer	80
A.1.8	Weather Publisher	81
A.1.9	AWS Cloudwatch	81
A.2	Custom Service/Topic definition	81
A.2.1	cf_messages	81
A.2.2	alarm_interfaces	83
A.2.3	auction_interfaces	83
A.3	Configuration Files	84
A.3.1	Integration Service	84
	References	89

List of Figures

2.1	ROS1/ROS2 Architecture Comparison	7
2.2	ROS2 layered architecture	8
2.3	DCPS model	9
2.4	Comparison and interoperability of the ROS2 and micro-ROS stack	10
2.5	Differences between cloud service models	13
2.6	Intra-process latency profiling of ROS2	15
2.7	Impact of RT settings in ROS2 performance	16
2.8	UAV-IoT interaction	20
2.9	OFERA's drone use-case overview	21
2.10	UAV-Edge-Cloud model	23
3.1	System Concept	26
3.2	System Breakdown Structure	27
3.3	Bitcraze Crazyflie 2.1 nano-quadcopter	28
3.4	Crazyflie 2.X system architecture	28
3.5	Positioning System	29
3.6	Stabilizer module	30
3.7	PID controller architecture	31
3.8	Test arena at Capgemini Engineering	32
3.9	System Hardware	33
4.1	Communication Architecture	35
4.2	Micro XRCE-DDS Agent working principle	36
4.3	Communication stack	37
4.4	Encapsulation of micro-ROS messages in MAV - GS communication	38
4.5	Serial Framing Protocol working principle	38
4.6	micro-ROS communication simplification	39
4.7	Main mission control FSM	40
4.8	Visual representation of the CAS	41
4.9	GUI representation in rqt_graph	44
4.10	Mission planning GUI	45
4.11	Sequence diagram of the trajectory planning triggered by a human operator using the GUI	46
4.12	Sequence Diagram for the "auction" algorithm	47
4.13	Node/topic interaction of the auction system	48
4.14	Bidding FSM	48
4.15	Supervision FSM	49
4.16	Main FSM with alarm functionality	49

4.17 Sensing Hardware	50
4.18 rqt_graph of sensor publishing and humidity correction.	51
4.19 WAN communication example using integration service	52
4.20 Cloudwatch diagram.	53
4.21 rqt_graph-based representation of the AWS Cloudwatch related nodes	53
4.22 AWS Cloudwatch dashboard	54
4.23 RVIZ visualization	55
5.1 Example planning	59
5.2 Example planning with different criteria	60
5.3 Planner RTT Test	61
5.4 Graphical representation of the growth of the median planning time w.r.t. MAVs and waypoints	62
5.5 Execution of test 1	64
5.6 Example mission with RVIZ visualization.	65
5.7 Artificial contamination source setup	66
5.8 IAQ Mapping Experiments	67
5.9 Impact of IAQ sensor on time of flight	68
5.10 QoS profile comparion and CRTP communication baseline ($p_L = 8B$)	69
5.11 Latency comparison for the different QoS profiles	70
5.12 Latency w.r.t. payload size for different QoS RELIABILITY policies ($f = 1Hz$)	70
5.13 Latency w.r.t. publishing frequency for different QoS RELIABILITY profiles ($p_L = 8B$).	71
5.14 Latency comparison for a stationary and airborne MAV.	71
5.15 Latency comparison for a single and two MAVs.	72
5.16 End-to-end latency measurement setup.	73
5.17 Median end-to-end latency profiling	73

List of Tables

2.1	Summary of the most relevant ROS2 evaluations	18
5.1	Comparison of the route length and total mission time considering or not the measurement time	60
5.2	Device specifications	61
5.3	Median planning time (in seconds) w.r.t. the number of waypoints and MAVs - both nodes are running on a Desktop PC. Values in bold represent decreases in latency w.r.t. number of MAVs	62
5.4	Median planning and communication time (in seconds) w.r.t. the number of waypoints and MAVs - comparison of using local and cloud computing.	63
5.5	QoS Profiles used in the evaluation	69
A.1	MAV Publishers	77
A.2	MAV Subscribers	78
A.3	Services used by the GUI (as a server)	78
A.4	Services used by the GUI (as a client)	78
A.5	Subscribers	79
A.6	RQT Plugin (GUI) Subscribers	79
A.7	RQT Plugin (GUI) Services used by the Planner (as a server)	79
A.8	RVIZ Subscribers	79
A.9	Actions used by the alarm node (as a client)	80
A.10	Actions used by the task manager (as a server)	80
A.11	Services used by the task manager (as a client)	80
A.12	Services used by the task auctioneer (as a server)	80
A.13	Auctioneer subscribers	80
A.14	Auctioneer publishers	81
A.15	Weather publisher subscribers	81
A.16	Cloudwatch-related subscribers	81

List of Abbreviations

5G	Fifth-Generation
API	Application Programming Interface
ARM	Advanced RISC Machines
ASIC	Application-Specific Integrated Circuit
AWS	Amazon Web Services
BE	Best Effort
BLE	Bluetooth Low Energy
BLOS	Beyond Line of Sight
CAD	Computer Aided Design
CAN	Controller Area Network
CF	Crazyflie
CI/CD	Continuous Integration/Continuous Delivery
CLI	Command Line Interface
COTS	Commercial off-the-shelf
CRC	Cyclic Redundancy Check
DCPS	Data-centric Publish-Subscribe
DDS	Data Distribution Service
DOF	Degrees of Freedom
EC2	Elastic Cloud Computing
eCO ₂	Equivalent Carbon Dioxide
EKF	Extended Kalman Filter
ESB	Enhanced ShockBurst
ESC	Electronic Speed Controller
GPS	Global Positioning System
GPU	Graphics Processing Unit
GS	Ground Station
GUI	Graphical User Interface
HLC	High-level Commander
HTTP	Hypertext Transfer Protocol
I ² C	Inter-Integrated Circuit
IaaS	Infrastructure as a Service
IAQ	Indoor Air Quality
IIoT	Industrial Internet of Things
IMU	Inertial Measurement Unit
IoT	Internet of Things
IP	Internet Protocol
LPS	Loco Positioning System
LTS	Long-term Support

MAV	Micro Aerial Vehicle
MCU	Microcontroller unit
MmTSP	Multi Depot Travelling Salesman Problem
OEM	Original Equipment Manufacturer
OS	Operating System
p2p	Peer-to-peer
PaaS	Platform as a Service
POSIX	Portable Operating System Interface
QoS	Quality of Service
RCL	ROS Client Library
RE	Reliable
REST	Representational State Transfer
RMW	ROS Middleware
ROS	Robot Operating System
RT	Real-time
RTOS	Real-time Operating System
RTPS	Real-time Publish Subscribe
RTT	Round Trip Time
S3	Simple Storage Service
SaaS	Software as a Service
SCADA	Supervisory control and data acquisition
SFP	Serial Framing Protocol
SH	System Handle
SOAP	Simple Object Access Protocol
SROS2	Secure Robot Operating System 2
TCP	Transmission Control Protocol
TDMA	Time Division Medium Access
TDoA	Time Difference of Arrival
TL	Transient Local
TVOC	Total Volatile Organic Compound
UART	Universal Asynchronous Receiver-Transmitter
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
USB	Universal Serial Bus
VoIP	Voice over IP
VRP	Vehicle Routing Problem
VT	Volatile
VTOL	Vertical Take-Off and Landing
WAN	Wide Area Network
WSN	Wireless Sensor Networks
XRCE-DDS	eXtremely Resource Constrained Environments - Data Distribution Service

Chapter 1

Introduction

This chapter firstly introduces the context and background for the dissertation in Section 1.1, the motivation for carrying out this project is then discussed in Section 1.2. Contributions from a scientific and enterprise standpoint are described in 1.3. Finally, in Section 1.4 a brief outline of the report's structure and content is given.

1.1 Context

In recent years, the usage of autonomous micro aerial vehicles, sometimes referred as micro-UAVs or MAVs in indoor applications has received growing attention from the robotics community due to their high mobility and ability to maneuver through highly constrained three-dimensional space [1]. MAV-based systems are emerging in real-world applications such as greenhouse monitoring [2], surveillance [3] or autonomous inventory management in industrial warehouses [4]. Using teams or swarms of multiple MAVs has become a common practice in such applications as it offers several advantages compared to using a single MAV such as fault-tolerance and increased time efficiency [5].

Interacting with sensors, actuators and other devices in the environment is also often desired, which leads to an increasing amount of complex systems that leverage the high mobility of MAVs in UAV-aided IoT wireless networks to collect data from ground-sensor nodes [6] and/or act as relays or airborne base-stations [7].

As MAVs in indoor settings typically maneuver through dynamic environments where adaptation and planning is crucial, real-time or near real-time computing for motion planning and control is usually a requirement. In sensing applications, data gathering and transmission normally has to comply with Quality of Service (QoS) requirements depending on the specific use-case.

ROS is an open-source software platform and meta-operating system designed for robotic applications [8] which allows developers to create modular software through the usage of packages contributed by the community. ROS has been widely adopted for UAV software development thanks to numerous open-source packages in various fields necessary to a UAV software stack such as control, planning, communications and mapping.

However, ROS has a few caveats that hinder its adoptance in industrial settings such as its inability to guarantee real-time performance [8]. To address this, a successor to ROS named ROS2 is currently being released and developed in parallel with ROS, henceforth referred to as ROS1 for the sake of nomenclature. ROS2 was developed according to design guidelines¹ such as: security, fault-tolerance, real-time support and QoS requirements.

1.2 Motivation

ROS2 solves many of the problems that ROS1 faced in industrial and real-world scenarios and shows great promise as a framework for Capgemini Engineering Portugal to develop systems compliant with real-time industrial standards in many areas including the automotive and aerospace sector.

We believe data sensing using MAVs is an example of a system that can greatly benefit from the new features that ROS2 offers namely its support for real-time and interoperability. The development of an end-to-end proof of concept would not only help validate the usage of ROS2 in said systems but serve as a platform to further explore the performance of the framework.

1.3 Contributions

As this thesis was done in collaboration with an external entity (Capgemini Engineering) the results obtained during the development of this dissertation project contribute both to the company's growth and to scientific development mainly in the field of robotics as ROS2 and micro-ROS are still fairly recent frameworks and their potential in IIoT applications has not been fully realized yet. The contributions can be summarized as follows:

- Validation of the use of ROS2/micro-ROS as frameworks for applications with critical embedded systems, in particular with MAVs, by performing an evaluation of the communication latency.
- A ROS2-enabled end-to-end communication architecture for collecting sensor data from MAVs that integrates subsystems with different computing capabilities, real-time requirements and operating systems.
- A mission planning algorithm suited to a target application, minimizing the time needed to visit a set of waypoints and perform sensor measurements.
- Validation of the proposed architecture with real-world experiments.
- Development of a MAV-based data sensing platform for the Capgemini Engineering Portugal Research and Innovation (R&I) team to show potential business partners in the relevant sectors, as well as to use in other R&D projects. During the course of this thesis, two demos, to clients and other internal R&D teams, were made showcasing our work.

¹https://design.ros2.org/articles/why_ros2.html

- Interaction and collaboration with the maintainers of micro-ROS and the open-source community through Github issues, bug reports and general discussions. In this respect, a presentation in the ROS2 Embedded Working Group is planned in order to share knowledge, experience with the framework and contribute to the community.

1.4 Dissertation Structure

This dissertation is structured in six chapters:

Chapter 1 is meant to introduce the topic of the dissertation and provide the reader with the context and motivation behind the project as well as a brief outline of the project's concept and goals.

Chapter 2 is dedicated to the results obtained from a literature review and intends to present the state of the art in the various fields or areas relevant to the dissertation as well as identifying related work and current research gaps.

Chapter 3 contains a high-level description of the concept and architecture of the system. It also provides an overview of the subsystems, tools, and testing facilities used in this dissertation in order to understand the limitations and benefits of the technologies used.

Chapter 4 details how each subsystem is implemented, their interaction as well as the design choices and limitations.

Chapter 5 presents and discusses the experimental evaluation results and procedures for data gathering.

Chapter 6 briefly summarizes the conclusions of the research done in this dissertation, considerations, limitations and future work.

Chapter 2

Literature Review

In this chapter, the results of the literature review and state of the art in relevant fields is presented. Section 2.1 starts by describing the methodology and criteria used in the literature review. Section 2.2 provides the technical background needed to understand the technologies and platforms used. Section 2.3 provides a more in-depth analysis of similar or highly relevant projects. Finally, Section 2.4 attempts to summarize relevant results as well as identify research gaps and trends.

2.1 Research methodology

In this review, a few guidelines were adopted:

1. The review process started with retrieving papers from Scopus, IEEEExplore and Engineering Village within five main topics: (i) Autonomous MAVs and Swarms, (ii) UAV-aided IoT and sensing applications (with focus on indoor air quality and gas sensing), (iii) ROS1 and ROS2, (iv) Cloud Robotics and (v) Real-time systems.
2. To further narrow-down the results, with the exception of ROS2 papers and a few other cases, papers were considered for a more in-depth analysis only if they included two or more of said topics and seemed relevant based on abstract.
3. As ROS2 is a relatively young framework and is currently under heavy development other more informal sources were used as a complement such as talks from developer conferences (i.e. ROSWORLD). For relevance's sake only sources from 2016 onwards were considered.
4. Retrieved papers were classified into two main overlapping categories: (i) applications within said topics and (ii) performance analysis of frameworks or systems (such as ROS2) which by generalization also includes surveys.
5. Research focused mostly on ROS2 as it is the core study subject of the thesis.

As one of the core principles of ROS2 is its high modularity, many of the analyzed applications may seem out of the scope of this thesis but were also considered as long as they took advantage of ROS2's innovative features.

2.2 Technical Background

2.2.1 ROS Overview

The first version of the Robot Operating System (ROS1) [9] is an open-source software framework for developing robotics software. The goal of ROS1 is to "encourage collaborative robotics development" by offering a modular design where developers can create, share and reuse packages from the community instead of building robots from scratch [10]. ROS1 has received wide acceptance for research and academic purposes as well as increasingly been adopted in industrial settings [11].

At its core, ROS1 consists of a peer-to-peer network of processes called the *computation graph*. These processes are named *nodes* and communicate between each other using *messages* either through a publish/subscribe pattern using *topics* or by following a request/reply model using *services*. Data such as algorithm parameters or sensor/actuator configurations can be stored by key and modified in run-time through the *Parameter Server*.

2.2.1.1 Limitations of ROS1 and the need for a redesign

ROS1 started as a development tool for the Willow Garage PR2 robot [12], a humanoid mobile manipulation robot that could navigate autonomously in a known environment. Although flexibility was one of the core software design guidelines, ROS1 was still created for this use-case which included a single robot with desktop processing capabilities, highly stable networks, no real-time requirements and academic or research settings.

However, as ROS grew and branched out into different robotics fields, this use-case no longer matched the requirements or needs of some applications. These new use-cases include but are not limited to [12, 13, 14]:

- **Multi-Robot Systems:** Although it is possible to build multi-robot distributed systems in ROS1, the underlying single master node architecture has a well known single point of failure [15].
- **Operating System:** ROS1 was designed to run on top of UNIX-based systems, having official support for Ubuntu and unofficial support by the community for other linux distributions such as Fedora and Arch Linux as well as macOS.
- **Embedded Systems:** As stated previously it is not possible to run ROS1 on Real-time Operating Systems (RTOS) or bare-metal. A common approach to integrate microcontrollers into ROS1-based systems is to run ROS1 on a computer or embedded device (i.e. a Raspberry Pi) and communicating with the micro-controller by wrapping and sending ROS standard serialized messages through a serial port or network socket using packages such as *roserial* [16].
- **Lossy or Unreliable networks:** ROS1 has been developed to run on a single machine or in multiple machines within a single robot where the various nodes are connected through

the local *loopback* interface or through wired networks such as ethernet. However, in multi-robot systems it is often desired to use wireless communication which is typically associated with reduced bandwidth and increased jitter. The lack of QoS settings and predominant use of TCP (although the middleware also supports UDP messages) often makes the usage of ROS impractical in lossy network settings [17].

- **Real-time Support:** ROS1 does not support hard or soft real-time computing due to being built on top of Linux and featuring other sources of non-determinism such as its memory allocation and management policies.

2.2.1.2 ROS2 architecture and comparison with ROS1

ROS2, the successor to ROS1 has been developed in parallel with ROS1 as a solution to the use cases referred in 2.2.1.1. As the use-cases of the two frameworks differ greatly, ROS2 has been designed with a vastly different architecture to its predecessor (shown in Figure 2.1) which means that there is no direct retro-compatibility between ROS1 and ROS2 nodes.

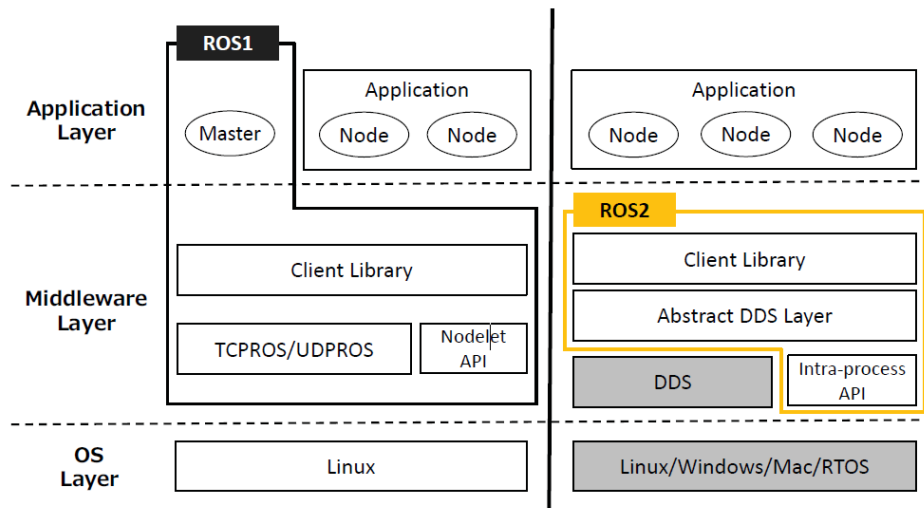


Figure 2.1: ROS1/ROS2 Architecture Comparison [18]

One of the most relevant differences, is how communication between nodes is carried out. ROS2 replaces the custom transport system of ROS (TCPROS/UDPROS) and has been built, by default, on top of the DDS (Data Distribution Service) middleware [19] which is a "data publish/-subscribe standard designed for distributed real-time systems" [20]. Data transmission is accomplished through the usage of the RTPS protocol which supports multicasting using UDP.

However, ROS2 does not use a specific implementation of DDS but instead allows the usage of different implementations of DDS (or even other types of middlewares such as Eclipse iceoryx¹ which is specifically tailored for intra-process communication) from multiple vendors through an

¹<https://github.com/eclipse-iceoryx/iceoryx>

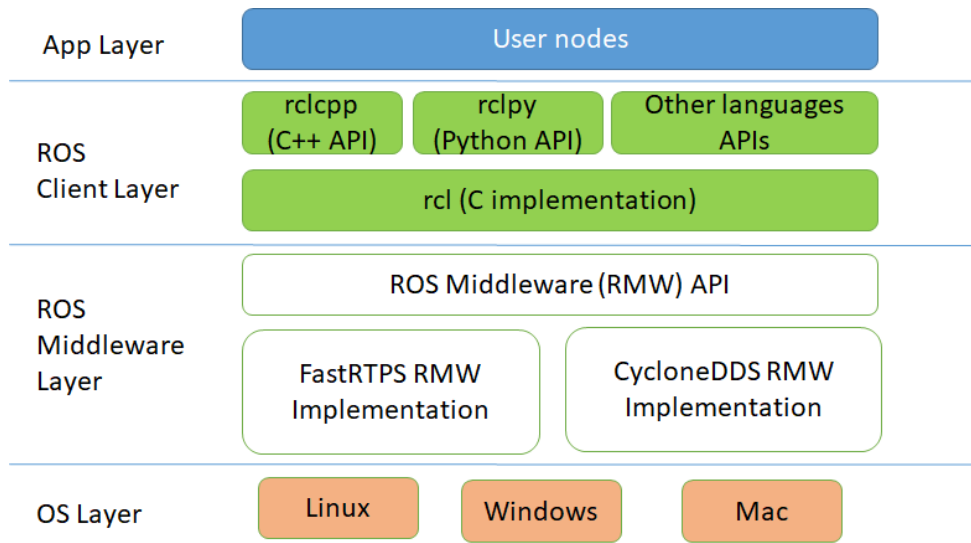


Figure 2.2: ROS2 layered architecture [22]

interface called the ROS Middleware layer (RMW). This layer sits between the ROS Client Library (RCL) and a middleware implementation providing a middleware-agnostic API to the RCL cf. Figure 2.2. Each specific implementation of the DDS standard requires a dedicated implementation of the RMW layer which translates messages between the common agnostic RMW interface and the vendor-specific DDS API [21]. This layer abstraction allows developers to create programs without focusing on the low level details of DDS [18].

The client layer consists of a pure C library, RCL, that implements core ROS2 logic and functionalities. Client libraries can then be built on top of RCL as wrappers. Client libraries typically use higher level programming languages such as C++, Python, Java etc.. ROS2 officially supports the *rclcpp* (C++), *rcl* (C) and *rclpy* (Python) APIs although many more are available as open-source community driven projects. ROS2 nodes written in different languages can co-exist and interact within the same project.

ROS2 also supports a growing number of operating systems including Windows, macOS, linux and RTOS.

2.2.1.3 ROS2 communications and DDS basics

ROS1 uses a publish-subscribe communication model that relies on the custom UDPROS and TCPROS transport systems. ROS1 discovery and node registration is done by the ROS MASTER.

ROS2 completely overhauls this system by using DDS as its messaging system, which relies on the Data-centric Publish-subscribe (DCPS) for data distribution who generates a global data space where each Domain Participant can read and write data to [18]. Each participant contains DCPS entities including *Subscribers*, *Publishers*, *DataReaders* and *DataWriters* which communicate through the RTPS protocol [23] as illustrated in Figure 2.3.

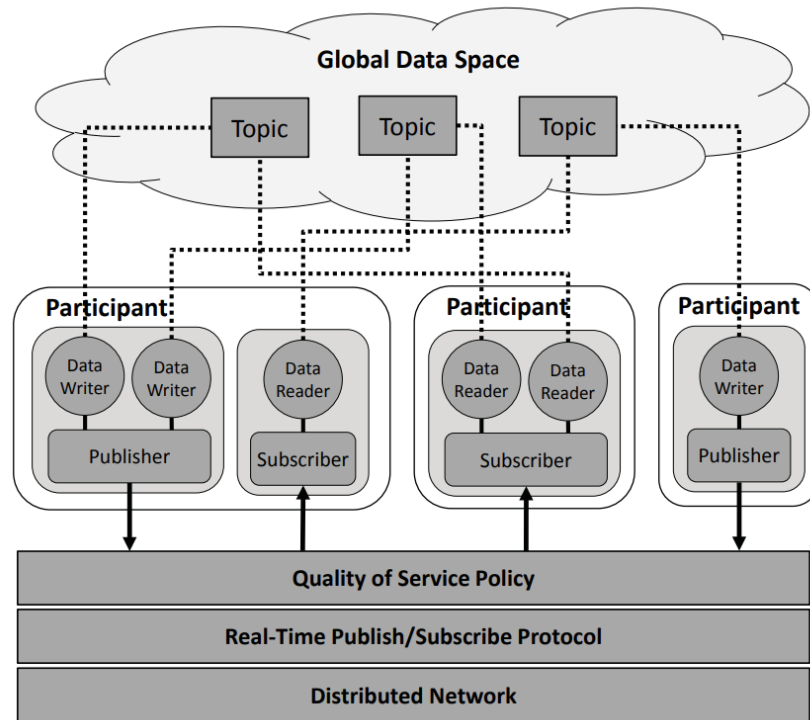


Figure 2.3: DCPS model [18]

DDS solves many of ROS1 communication system's issues. QoS policies can be customized through the RMW layer for a specific application temporal and reliability requirements. QoS profiles consist of a subset of QoS policies that are available in ROS2 and are detailed below [24]:

- **History:** Determines if all samples should be stored in the queue (*Keep all*) or only a number of samples (*Keep last*), this number is defined through the **Depth** policy.
- **Reliability:** Whether samples are guaranteed to be delivered (*Reliable*) or not (*Best Effort*).
- **Durability:** If publishers should save samples for future (late-joining) subscribers (*Transient Local*) or not (*Volatile*).
- **Deadline:** The maximum amount of time between consecutive messages are published to a given topic.
- **Lifespan:** Determines how long before a sample is considered to be "expired", can be used to avoid delivering old data.
- **Liveliness:** Can be *Automatic* if the system asserts that the publisher node is alive by making sure it publishes a message within a given time specified using the **Lease Duration** policy or *Manual* if it interacts directly with the publisher API.

Although QoS profiles are flexible, ROS2 offers default QoS profiles for common use-cases. It is important to understand that each DDS entity can have different QoS profiles and communication only happens if they are compatible.

This features allows for increased flexibility and robustness as a node can subscribe to topics published by nodes with different temporal requirements or demand the published topics to comply with a pre-defined QoS profile.

Aside from serialization and transport ROS2 also uses DDS for discovery and therefore does not need a master node like ROS1. SROS2² is a collection of tools to integrate features such as authentication, access control and cryptography [26] that are defined in the DDS-Security specification [27] Improving security is a major ROS2 design guideline due to the more production oriented nature of the framework.

At the time of writing the most recent LTS version of ROS2 is "ROS2 Foxy Fitzroy" or "foxy" for short. The default DDS implementation of foxy is Fast DDS by eProsima (former Fast RTPS) [28], nevertheless ROS2 offers many RMW implementations to support other DDS vendor implementations.

2.2.2 micro-ROS

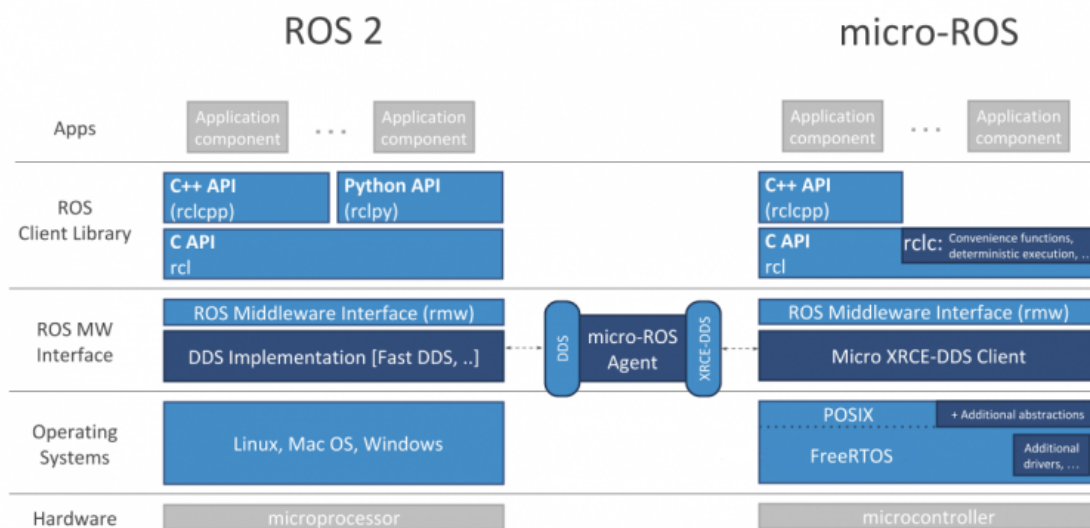


Figure 2.4: Comparison and interoperability of the ROS2 and micro-ROS stack. The micro-ROS agent acts as an intermediary between the two stacks. [29]

Targeting microcontrollers and resource-constrained embedded systems, micro-ROS is a framework that "puts ROS2 onto microcontrollers" [30]. As stated in section 2.2.1.1 distributed robotic

²SROS2's name is an homage to a ROS1 package called SROS[25] that attempted to improve ROS1's lacking security. Not to be confused with SROS, SROS2 is not an add-in but an integrate part of ROS2.

systems are an increasingly common use case and as such interoperability with environmental sensors and IoT-based systems is becoming more and more important.

Naturally, micro-ROS is optimized for microcontrollers and as such its architecture differs slightly from ROS2 as illustrated in Figure 2.4. Notably, micro-ROS uses by default the eProsima Micro XRCE-DDS middleware that implements the DDS-XRCE (DDS for eXtremely Resource Constrained Environments) standard enabling resource-constrained devices to access the DDS global data space.

Micro XRCE-DDS has a client-server architecture where constrained devices are clients that publish or subscribe data (using the Micro XRCE-DDS Client C99 library) through an agent by UDP or Serial (implemented through the Micro XRCE-DDS Agent C++11 library) who acts as a server between multiple clients and DDS.

2.2.3 Micro Aerial Vehicles

In [1] Kumar et. al, define MAVs as a subset of UAVs that are between 0.1-0.5m lengthwise and have a mass inferior to 0.5 kg. MAVs can be classified according to the type of frame and propulsion type. Most MAVs fall within the categories: i) multi-rotor (which are then sub-classified according to the number of rotors), ii) fixed-wing or iii) hybrid VTOL. Although MAV kinematics, dynamics, modeling and control are vast fields of study, in this thesis focus is put instead on the software and flight stack of these platforms specifically in multi-rotors. Furthermore, emphasis is given to UAVs that are capable of autonomous or semi-autonomous flight in opposition to manually piloted UAVs.

Software that enables autonomous flight is usually called *autopilot*, the on-board computer or embedded device responsible for state estimation, control and stable flight is called the *flight controller*. The *flight controller* interfaces both with sensors through standard interfaces such as UART, CAN and I^2C (common sensors include IMU, GPS, Cameras and barometers) and motors which are typically brushless DC motors controlled through ESCs.

2.2.3.1 Communications

For indoor applications short-range communication technologies such as Wi-Fi (IEEE 802.11), Bluetooth (IEEE 802.15.1) or ZigBee (IEEE 802.15.4) are common choices for MAVs due to low-cost and integration with other systems (i.e. WSN) [31]. For outdoor use, long-range technologies are generally necessary especially in beyond line of sight (BLOS) flight and can be grouped in: i) Point-to-point communication such as radio-based telemetry, ii) Satellite communications and iii) cellular networks. In recent years, the latter has received special attention for IoT and smart-cities applications due to the low-latency promise of 5G.

The most common protocol for GS-UAV communication is MAVLINK³, a lightweight protocol for communication with unmanned systems that is supported by a wide range of autopilots

³<https://mavlink.io/en/>

and ground control software. However, due to its lightweight nature it does not support QoS settings[32] and has known security flaws [33]

2.2.3.2 Autopilots and ROS integration

Several autopilots are available as open source software. In [34] Ebeid et al. perform a qualitative analysis of over 20 open-source autopilots as well as flight controllers and simulators. At the time of writing most autopilots rely on MAVLINK and use the MAVROS⁴ package as a bridge to communicate with ROS1/ROS2. PX4⁵ is a noteworthy exception that also supports direct interface with ROS2 through a fast-RTPS bridge⁶. Furthermore, there are plans to replace this bridge with Micro XRCE-DDS allowing the optional usage of micro-ROS.

2.2.4 Cloud Robotics

The term "Cloud Robotics" was coined by James Kufner in 2010 and refers to "any robot or automation system that utilizes the cloud infrastructure for either data or code for its execution"[35]. Since then, the integration of cloud computing in robotics and automation has grown in popularity and become a driving force in Industry 4.0 [36].

Cloud computing allows robots to offload computationally intensive tasks and to share or store data through the Internet. [37] Some common uses of cloud robotics are [35]:

1. **Storing large amounts of data:** Mobile robotic applications can generate large amounts of data such as sensor readings. On-board storage in these robots is usually limited due to payload, cost or size constraints.
2. **Accessing big data:** Applications such as object recognition, grasping or data-driven algorithms such as machine or reinforcement learning algorithms which typically require a knowledge base associated with large datasets of images, CAD models or 3D point clouds.
3. **Parallel or grid computing:** Algorithms for motion planning, statistical analysis, and deep learning are typically computationally intensive and largely benefit from external hardware acceleration through GPUs or custom ASICs. Cloud services also typically provide elastic on-demand computing.
4. **Collaborative robotic learning:** Sharing perception data and logs acquired during tasks between robots facilitates learning through data-driven approaches such as Reinforcement Learning [38]. The cloud not only provides a medium for knowledge sharing (even if the robots are geographically separated) but also the computing resources for these kind of applications.

⁴<http://wiki.ros.org/mavros>

⁵<https://px4.io/>

⁶<https://docs.px4.io/master/en/middleware/micrortps.html>, this feature is under heavy development and is still experimental.

Cloud computing in robotics is generally enabled through service models such as: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) or Software as a Service (SaaS), depending on the level of abstraction provided to the end user [39] (cf. Figure 2.5).

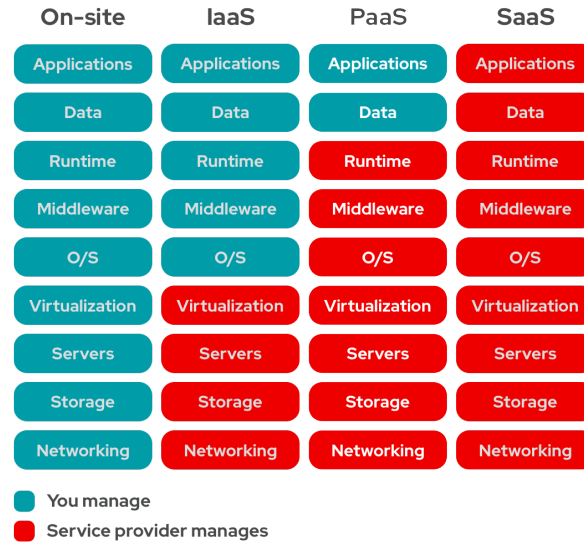


Figure 2.5: Differences between cloud service models [40]

Due to its inherent latency, using cloud computing may be unfeasible due to the real-time requirements of a given application. Hence, in robotic application where these requirements are present, other paradigms such as *fog*, *mist* or *edge* are commonly used.

2.2.5 Real-Time Systems

Real-time systems are systems where "the correctness of a computation depends not only on the logical correctness but also on the time at which the results are produced" [41].

This kind of systems are present in many applications such as in the automotive and aerospace sectors and are predominant in industrial applications for process control, automation and SCADA systems. An operating system that is designed to be used in real-time systems and therefore guarantees determinism is called a Real-time Operating System (RTOS).

2.2.5.1 Types of RT tasks

RT tasks are associated with a quantitative time constraint that is often described in terms of a deadline to complete, therefore tasks can be classified in terms of the consequences of failing said deadlines [42].

- **Hard-RT tasks:** Inability to complete the task within the defined temporal boundaries is considered a system failure (i.e. Collision Avoidance).

- **Firm-RT tasks:** Late-completion does not result in failure but late results are usually discarded (i.e. VoIP).
- **Soft-RT tasks:** Missed deadlines do not result in failure but cause system performance degradation. Late results can still be useful (i.e. Web browsing).

2.2.5.2 Linux for hard-RT

Linux is a general-purpose operating system and common choice for embedded devices and computers due to its wide collection of open source libraries and hardware support. In robotics, Linux OSs namely Ubuntu are especially used due to being the supported OS of ROS1. However, the standard Linux kernel can not handle real-time and thus two main different approaches are available to add real-time support to it (not including running Linux in a machine that commands an RTOS in another, although that is a common practice in robotics):

- **The PREEMPT_RT patch:** Like the name implies, it is a modification of the Linux kernel that makes it fully preemptible.
- **Using a co-kernel:** In this approach, a secondary microkernel is used in parallel with the linux kernel and is responsible for the real-time threads.

2.3 Related Work

In this section, an analysis is made of the work relevant to the proposed study. This analysis consists of both a review and comparison of evaluations of the performance of ROS2 and a brief survey of applications in the field of MAV data collection using interaction with wireless sensor nodes, environmental monitoring (namely Indoor Air Quality (IAQ)) or cloud computing.

2.3.1 Evaluation of ROS2

ROS2 evaluations can be categorized in four overlapping categories: i) ROS2 stack and communications, ii) real-time performance, iii) evaluations based on real-world or practical implementations and iv) proposals of improvements or modifications to ROS2.

2.3.1.1 ROS2 stack and communications

Maruyama et al. in [18] explored the performance of ROS2 and DDS by experimenting with different QoS profiles, different implementations of DDS and using several different forms of communication in ROS1/ROS2. Evaluation was based on latency, throughput and memory consumption for different data sizes. They found that the overhead caused by converting data to DDS and abstracting DDS from ROS2 is not suitable for large data and is dependent on QoS policies. Nonetheless, the authors believe that the advantages of using DDS in ROS2 outweighs its cost due to enabling real-time and fault-tolerance. However, it is important to keep in mind that their

article was published in 2016 before the first LTS release of ROS2 and some functionalities of the framework and DDS implementations have improved since then.

Chen in [43] evaluated ROS2's performance with varying QoS profiles and security constraints in a lossy wireless 802.11a ad-hoc network through integration of the ns-3⁷ discrete-event network simulator. The authors concluded that the security features in SROS2 lead to a large overhead regarding latency and that increasing the number of nodes leads to a significant increase in latency which they believe questions the viability of the framework in swarm robotics.

Similarly, Fernandez et. al [44] evaluation used differing QoS and security settings and found that the negative impact of security in latency and throughput is considerably higher than that of QoS. Additionally, their results suggest that the HISTORY policy has the most significant impact in the latency.

Kronauer et al. in [45] conducted experiments to test the scalability of the framework. Tests were run both in a desktop PC and a Raspberry Pi using a variable number of nodes, three different DDS implementations to profile the communication latency and identify bottlenecks. They concluded DDS and *rcldpp* notification delay are by far the largest sources of latency and that ROS2 can have up to 50% latency overhead when compared to low-level raw DDS communications as shown in Figure 2.6. Other important results are: i) latency was larger in the Raspberry Pi, ii) for large data sizes (superior to UDP fragmentation) latency increases with the payload, iii) latency was also found to decrease with publishing frequency and iv) latency has a linear relationship with the number of nodes.

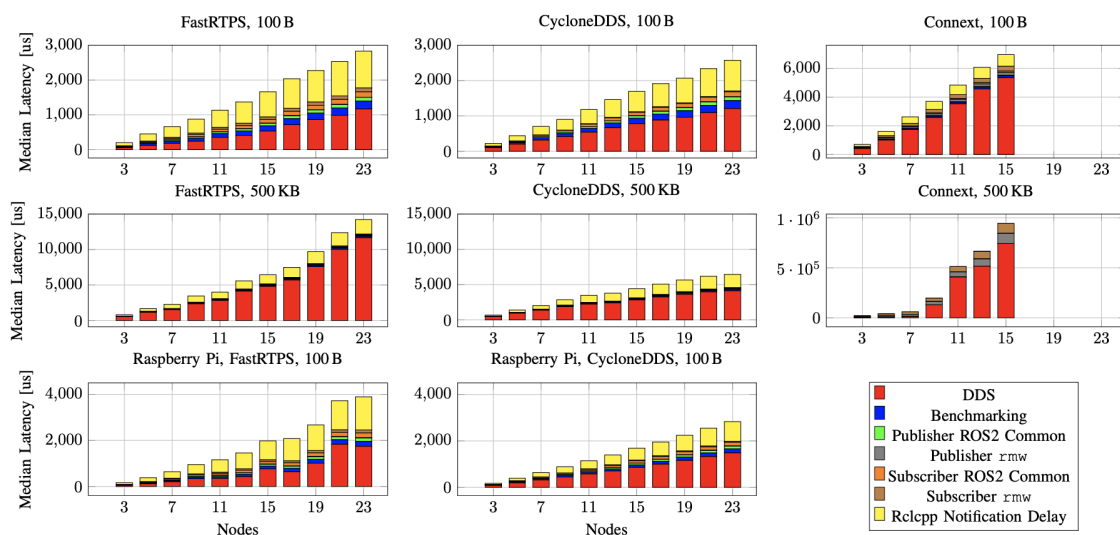


Figure 2.6: Intra-process latency profiling results of Kronauer et al. [45]. Evaluation consisted of simulating a linear data-processing pipeline of 3 to 23 nodes and evaluating the end-to-end latency. Publishing frequency was set 100Hz with BEST_EFFORT QoS policy. Evaluation was done on a Desktop PC and a Raspberry Pi 4.

⁷<https://www.nsnam.org/>

2.3.1.2 Real-time performance

Gutiérrez et. al in [46] further explored the communication stack of ROS2 using an experimental setup which consisted of a simple two node publisher/subscriber between a Ubuntu-based desktop PC and an ARM-based dual core embedded device. The main contributions added was the experimentation with different system and network loads, the impact in latency of using a PREEMPT-RT linux kernel and of using RT settings such as Two-Level Segregate Fit (TSLF) allocators, custom scheduling and QoS policies (depending on the DDS implementation) and tuning kernel thread priorities.

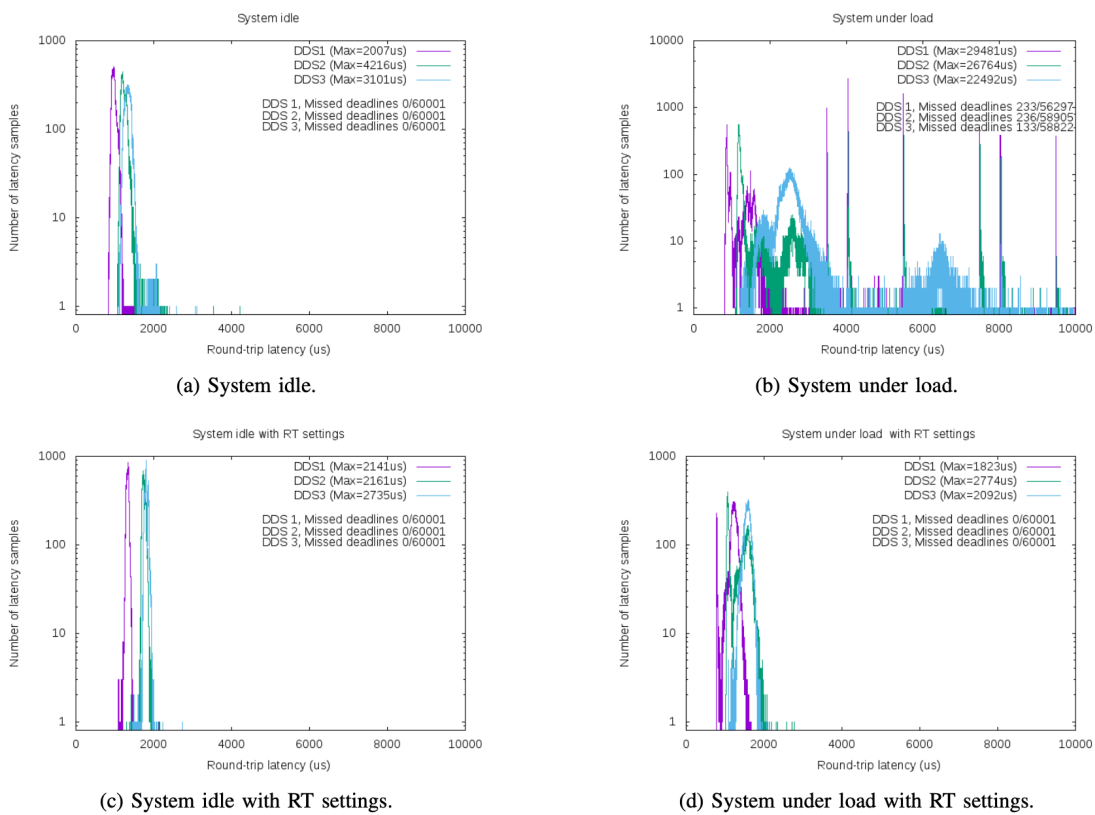


Figure 2.7: Results of Gutiérrez et al.'s work [46] evidencing a clear performance gain using RT optimized settings under system load. The authors used a round-trip test with a two node publisher/subscriber and a payload of 500Kb under 4 different test conditions: a) idle system w/o RT settings, b) system under load w/o RT settings, c) idle system w/ RT settings, d) system under load w/ RT settings

The authors' findings show that the best strategy to optimize ROS2's real-time performance is to use the real-time settings previously mentioned, configure kernel threads accordingly and limit the network and CPU of each machine as possible. The results show that proper real-time configurations can reduce by a fair amount the jitter and latency (figure 2.7). Based on these results the authors claim it is possible to achieve soft and firm real-time using the Linux Network

Stack but not hard real-time. It is also worth noting that the experiments used a wired connection (UDP/IP over Ethernet) instead of wireless communication.

2.3.1.3 Evaluations based on practical implementations

Reke et. al in [47] verified the real-time capability of ROS2 in a real application using self-driving cars (two modified KIA Niro) and showed that RT performance was significantly improved when using a real-time kernel and that the usage of ROS2 for RT is insufficient without careful scheduling and memory management.

Park et al. in [48] performed an exhaustive study of ROS2's real-time performance in comparison with ROS1 by evaluating both the software stack and communications. A multi-robot system was also developed in order to validate the real-time performance of ROS2 in a real-world system. The performed evaluation can be classified in three stages:

1. **Software stack:** The authors verified that real-time scheduling was possible by measuring the linux kernel latency in both idle and stressed environments and found that the maximum latency was significantly lower in ROS2 and that ROS1 was unable to satisfy the specified deadline. Node schedulability was tested by measuring the periodicity and response time for periodic tasks. The results showed that deviation was much higher in ROS1 especially under stress while ROS2 results had much smaller deviation showing deterministic behaviour.
2. **Communications:** Latency and message loss rate for a simple round-trip between two ROS nodes were the chosen metrics. Results showed that ROS1 performed very poorly in unstable environments while ROS2 only had a minor increase in latency which indicates that the framework is suitable for lossy networks.
3. **Real-world application:** In order to validate in a real environment, the node periodicity experiment was repeated with nodes in two different robots and the results were similar with ROS1 showing a large increase in jitter when under stress. Additionally, the authors performed a test where the robots had to follow a curved trajectory also under stress. ROS1 failed to satisfy RT deadlines for executing speed commands which lead to the robot diverging from the planned path. ROS2 was able to follow the path with almost no changes between the idle and stressed configurations.

The authors concluded that ROS2 vastly outperformed ROS1 in real-time performance and showed that the latter was not suitable for real-time applications.

2.3.2 Summary

A comparison and summary of the most relevant contributions to ROS2 evaluation is described in table 2.1.

Table 2.1: Summary of the most relevant ROS2 evaluations

Reference	ROS2 Version - Year	Setup	Conditions	Metrics	Findings
Maruyama et al. [18]	Alpha 2016	2 Desktop PCs all standard node communication methods for ROS1,ROS2 and ROS1-RO2 over Ethernet	Intra and inter-machine communication w/ variable QoS and different DDS implementation	Latency and throughput	DDS not suitable for large data sizes, causes significant overhead
Gutierrez et. al [46]	Ardent 2018	A desktop PC and an embedded ARMv7 device connected through Ethernet. Ubuntu 16.04 w/ RT kernel	Round-trip test with variable data size, traffic and stress. Different RT settings.	Latency, message loss and missed deadlines	Evaluation of the Linux Network Stack shows that soft and firm real-time communications are possible with appropriate RT settings
Chen [43]	Dashing 2019	NS-3 Simulation for ad hoc 802.11a network	Variable QoS and security settings	Message loss rate and latency	QoS and security result in significant overhead, poor scalability
Jiang et al. [49]	Bouncy 2019	Single server with Ubuntu 16.04	Different types of data (images, waypoint, lidar etc.)	Latency	Serialization in DDS causes a major bottleneck.
Continues in the next page					

Table 2.1 – continued from previous page

Reference	ROS2 Version - Year	Setup	Conditions	Metrics	Findings
Reke et al. [47]	Not Specified 2020	Two KIA Niro w/ i7 Intel NUCs connected by CAN	Comparison of standard and RT kernel for a Lane-keeping assist system	Jitter and positioning of the car in the lane.	Stability and jitter is improved by using the RT kernel. Validation of good RT behaviour.
Fernandez et al. [44]	Crystal 2020	Single Desktop PC with Ubuntu 18.04	Comparison of different QoS profiles for secured data and plaintext	Latency and throughput	Security had a larger influence on performance than QoS profiles.
Park et al. [48]	Dashing 2020	Multiagent wireless system w/ a Kobuki2, Tetra DS-IV and a back-end desktop PC. All running Ubuntu 18.04 w/ RT Kernel	Real robots and lossy networks, variable data sizes and network stress. Comparison with ROS1.	Message loss, latency, node periodicity and robot trajectory with variable stress	Validation of ROS2 in a RT multi-robot system in opposition of ROS1.
Kronauer et al. [45]	Foxy 2021	Desktop PC and Raspberry Pi 4 with standard kernels.	Different DDS implementations, variable publishing frequency, payload and number of nodes and reliability policy.	Median latency	For large payloads (>64kB) latency increases w/ payload, decreases with frequency. DDS and rclcpp notification delay are the biggest sources of latency

2.3.3 UAV-assisted IoT

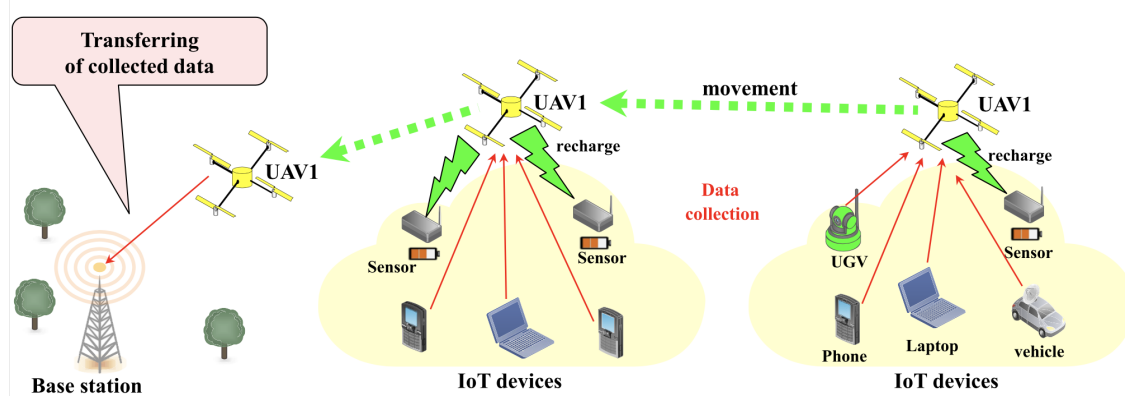


Figure 2.8: Example of UAV-IoT interaction [50].

Due to the limited energy and transmission range of IoT devices, multi-hop techniques can lead to unreliable or energetically inefficient network behavior [50]. As a possible solution, UAV-assisted IoT systems use UAVs as data carrier entities that hover around regions of interest with ground sensors, IoT devices or WSNs gathering data, transporting or relaying it to other sensors/base stations.

Energy Optimization: A major concern in UAV-assisted IoT systems is the energy consumption of the UAV or swarm of UAVs and the ground sensor nodes. Optimization of flight paths to reduce consumption is a hot topic in research. A common application is to have the UAV take off from the base-station, visit all nodes in the shortest possible path and return to the base-station which typically translates into a travelling salesman problem. Goudarzi et. al tackle this problem [51] which is proven to be NP-Hard and use Bezier curves to convert the obtained paths to smooth flyable trajectories. Along with gathering data, UAVs can also be used to transfer energy to depleted IoT devices through RF signals as explored in [52] by Arabi et al. and illustrated in Figure 2.8.

UAV-WSN: In an effort to reduce the energy consumption of the ground nodes, a common technique is to employ WSNs to cluster multiple sensors. In this configuration, instead of all sensor or IoT devices transmitting data, sensors are clustered and a node from that cluster, a cluster head, is used as a reliable data link between the UAV sink and the cluster[53]. Nonetheless, the challenges of the UAV-IoT paradigm still stand as the UAV flight path needs to be calculated with energy consumption in mind as well as the clustering methods used. In [54] Wu et. al propose a solution to optimize the flight between cluster heads using genetic algorithms for large-scale WSNs and in [6] Cao et. al take advantage of cloud computing to compute flying parameters, optimize the sequence of visited clusters and data analysis.

The OFERA drone use-case: An application for indoor drone sensing is described in the

OFERA's ⁸ "Drone use-case" specified in deliverable D6.1 [55]. This use-case⁹ consists of a single MAV, the Crazyflie 2.1 by Bitcraze AB, that works as a micro-ROS Client application and is commanded through a ground station (micro-ROS Agent) using the Crazyflie Real-time Protocol (CRTP). This communication is done through a serial agent and a *python3* bridge. The MAV flies around and retrieves data from multiple wireless sensors using a P2P connection. The sensor nodes consist of a Raspberry Pi 3A+ with a Sparkfun Weather Shield running micro-ROS agent lite. A simple overview of use-case is illustrated in figure 2.9.

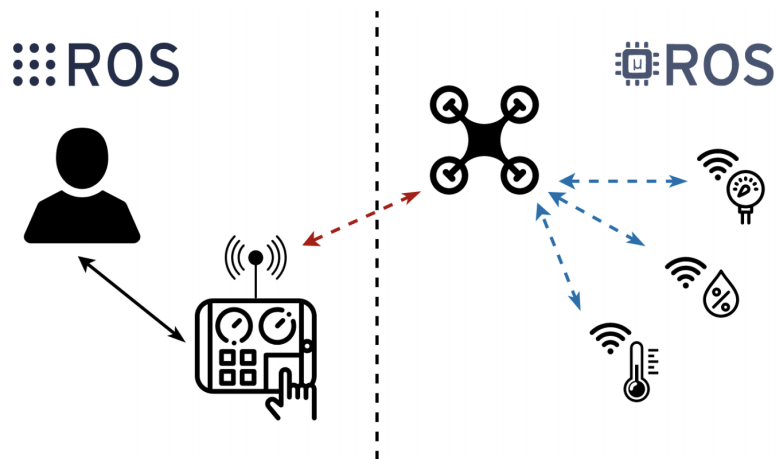


Figure 2.9: OFERA's drone use-case overview [55]

2.3.4 IAQ sensing using UAVs

The usage of UAVs fitted with environmental sensors is increasing in many scientific fields [56]. The small size of MAVs combined with their high maneuverability allows them to fly-over contaminated or hazardous locations and/or areas that are unaccessible by rovers or larger UAVs [57].

Neumann¹⁰ et. al in [58] propose an indoor IAQ monitoring solution based on a swarm of Crazyflie 2.0 quadcopters showing the viability of using nano UAVs for tasks such as gas distribution mapping with the goal of improving occupational health and safety of industrial workplaces.

The authors of [56] use the same quadcopter although with a different micro Metal OXide (MOX) sensor. They argue that the airflow (rotor downwash) causes the measurements to be inaccurate and therefore propose a *butterfly* inspired sensing approach where the MAV lands before taking a measurement in order to reduce the negative impact of the air flux caused by the propellers. In [57] Burgués et al. study the effect of the rotors on MOX sensor signals and show that even

⁸OFERA (Open Framework for Embedded Robot Applications) is a project funded by the European Union to extend ROS to microcontrollers through the micro-ROS framework described in 2.2.2. The project is under heavy development by the OFERA consortium which includes multiple companies such as Bosch and FIWARE and is lead by eProxima

⁹This use-case has since then been implemented as an example demo for the micro-ROS framework: https://micro.ros.org/docs/tutorials/demos/crazyflie_demo/

¹⁰The sensor (SGP30) deck driver used in this work has been kindly provided to us by the authors and is used in this dissertation.

tough the propellers have a significant impact on the quality of the measurements, information needed for Gas Source Localization (GSL) can still be extracted.

Duisterhof et al. developed the first fully autonomous solution for GSL in [59] using a swarm of resource constrained nano-quadcopters (Crazyflies) by developing "Sniffy Bug", an algorithm based on the bug and Particle Swarm Optimization (PSO) algorithms as well as a simulation pipeline "AutoGDM" for gas dispersion modeling.

2.3.5 Cloud-enabled UAV applications

As typically resource constrained systems, MAVs and UAVs in general can benefit from the additional computational power of cloud computing not only to perform more complex tasks but to improve battery life. Another benefit that UAV networks or Flying ad-hoc networks (FANETs) have from using cloud-based is knowledge sharing and collaboration.

2.3.5.1 Monitoring and UAV-UAV cooperation

The concept of Internet of Drones (IoD) was presented by Gharibi et al. in [60] as a conceptual control architecture that enables multiples UAVs to cooperate and coordinate themselves in a shared air space. However, the authors only provided an high-level description and an overview of the concepts needed to build such an architecture and not an actual implementation.

The IoD concept was then applied in Koubâa et al.'s *Dronemap Planner* described in [61]. *Dronemap planner* is a service oriented cloud-based management and deployment tool for controlling and monitoring a single UAV over the internet through SOAP or REST.

Hong et al. then developed a similar framework for multiple UAVs (albeit without cooperation between them) in [62]. This framework is based on a layered architecture that abstracts the UAV's specific hardware and control through the use of what the authors named a "shadow layer".

In [63] Mehrooz et al. propose a framework that integrates ROS1/ROS2 with a Kubernetes cluster in a client-server architecture for a power line inspection use case. Validation and testing were done through both a software-in-the-loop (SITL) and hardware-in-the-loop (HITL) simulation. The cloud services provide UAVs with information about the GPS locations of power towers and communication is done through REST over HTTP.

2.3.5.2 Cloud computing in UAVs

Industrial Settings: Salhaoui et al. in [64] developed a smart Industrial Internet of Things (IIoT) monitoring application for a concrete plant. Their application leverages cloud computing (IBM Watson) and deep learning to analyze images from the production line, captured using UAVs. The authors propose an architecture based on fog computing which uses a IoT Gateway that contains a Node-RED application to connect to the cloud, interface with the plant's PLCs based on OPC-UA and communicate with the sensing UAVs through Wi-Fi using the proprietary Node.js library. UAV supervision was successfully integrated into the plant system and the authors proposed a mathematical model for the end-to-end latency that proves the feasibility of the architecture for

the specific use-case. In their work, the authors address the integration of UAVs into IoT systems which they consider to be an "interoperability challenge" due to the different communication protocols used in IoT systems and the critical nature of UAVs. They consider that the challenge was overcome with the aforementioned fog computing architecture.

UAV-Edge-Cloud: In order to benefit from the computation advantages of the cloud while providing acceptable QoS, the authors of [65] proposed an hybrid UAV-Edge-Cloud computing model based on applications for smart cities, namely crowd sensing which use heavy computer vision and machine learning algorithms. This computing model is based on UAV-UAV collaboration in a ad-hoc network or swarm of UAVs, moving tasks with real-time requirements to the edge of the network and doing more resource intensive computation in the cloud as shown in Figure 2.10.

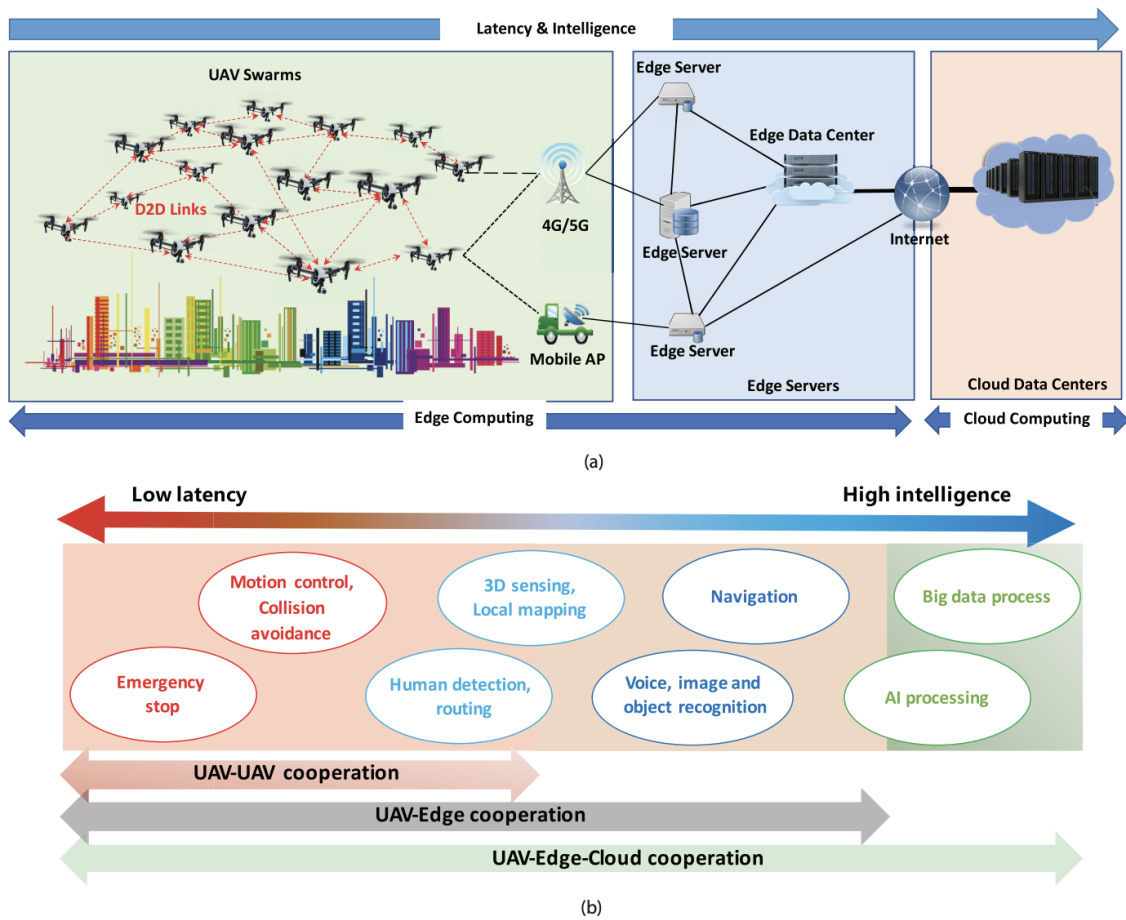


Figure 2.10: UAV-Edge-Cloud model proposed in [65]. a) Architecture overview, b) Applications in crowd sensing for smart cities.

2.4 Main Takeaways

Due the novelty of the framework, research on ROS2's performance is limited. However, QoS and security features were revealed to be a major factor in the performance of the framework in

terms of latency and throughput. Multiple authors also point out the overhead caused by DDS message conversion and serialization and conclude that the overall system performance is greatly dependent on the chosen DDS implementation as well as the underlying OS and system load.

Only a handful of publications that evaluate ROS2 use actual robotic platforms or multi-agent systems. Evaluation of ROS2 is typically done in desktop-grade PCs and/or embedded single board computers (i.e. a Raspberry Pi). In comparison, evaluation of microcontrollers and frameworks such as micro-ROS is quite limited. The Crazyflie 2.1, the MAV utilized in this project, is a microcontroller-based system and thus can also be used as platform to study ROS2/micro-ROS performance. Additionally, the MAV swarm proposed in this dissertation is a prime example of a multi-agent system.

Publications on IoT or cloud-based UAV sensing applications are also emerging due to the growth of enabling technologies such as 5G. The vast majority of UAV-IoT or UAV-WSN use-cases are for outdoor applications such as smart cities and agriculture and thus there is a great interest in the development of algorithms and methods to reduce energy consumption as well as optimize UAV trajectories and path planning. Cloud platforms are used to overcome the intrinsic computational limitations of UAVs allowing them to perform more demanding tasks such as vision-based object recognition and can act as a medium for knowledge sharing or cooperation between UAVs.

ROS1 is a recurring framework for UAV control and planning especially in indoor applications (GPS-denied) where perception and localization is crucial. However, to the best of the author's knowledge this dissertation the first application of autonomous UAVs running ROS2 for data sensing, possibly due to both the potential of ROS2 having not been fully explored and the UAV-IoT paradigm being relatively recent.

Integration of ROS1 or other proprietary libraries for controlling UAVs in IoT applications can pose what the authors of [64] deemed an "interoperability challenge". ROS2, using DDS as a middleware, has the potential of overcoming this challenge and seamlessly integrate IoT devices (and other robotic systems) in the same framework while providing real-time performance and empowering developers with the many tools and frameworks the ROS ecosystem offers. However, further research is needed to validate this approach. Our application aims to contribute to this validation by integrating ROS2 in a swarm-cloud architecture.

Chapter 3

MAV Data Sensing using ROS2

In this chapter, firstly a conceptual view of the use-case and proposed system is given in Section 3.1 which is then followed in Section 3.2 by a brief overview of the main technologies, software tools and frameworks to be used. Finally, Section 3.3 provides a brief summary of the key points of the use-case, design guidelines and how the available tools can be used to implement the concept as well as their limitations.

3.1 System Description

In this use-case proposed by Capgemini Engineering, MAVs are used as mobile airborne sensing platforms. Gathered data is then relayed to a ground station and forwarded to a cloud platform for storage, analytics and remote visualization. ROS2 is used to seamlessly integrate the various subsystems. In this dissertation, we aim to assess ROS2's performance and interoperability by creating an indoor data sensing application using swarms of MAVs (as an example use case we are performing IAQ measurements).

From this description, we can highlight three high level requirements:

1. Perform autonomous missions (trajectories) with one MAV or a swarm of MAVs to take sensor measurements at a series of waypoints.
2. Provide an operator with a graphical user interface to send basic commands to the MAVs and plan missions.
3. Send gathered sensor and mission data to a ground station (GS) and store it in a cloud-based platform for data storage and analytics.

The system concept is illustrated in Figure 3.1. It is straightforward to identify the main subsystems of the data sensing application. These subsystems have different functionalities and requirements that are briefly described below:

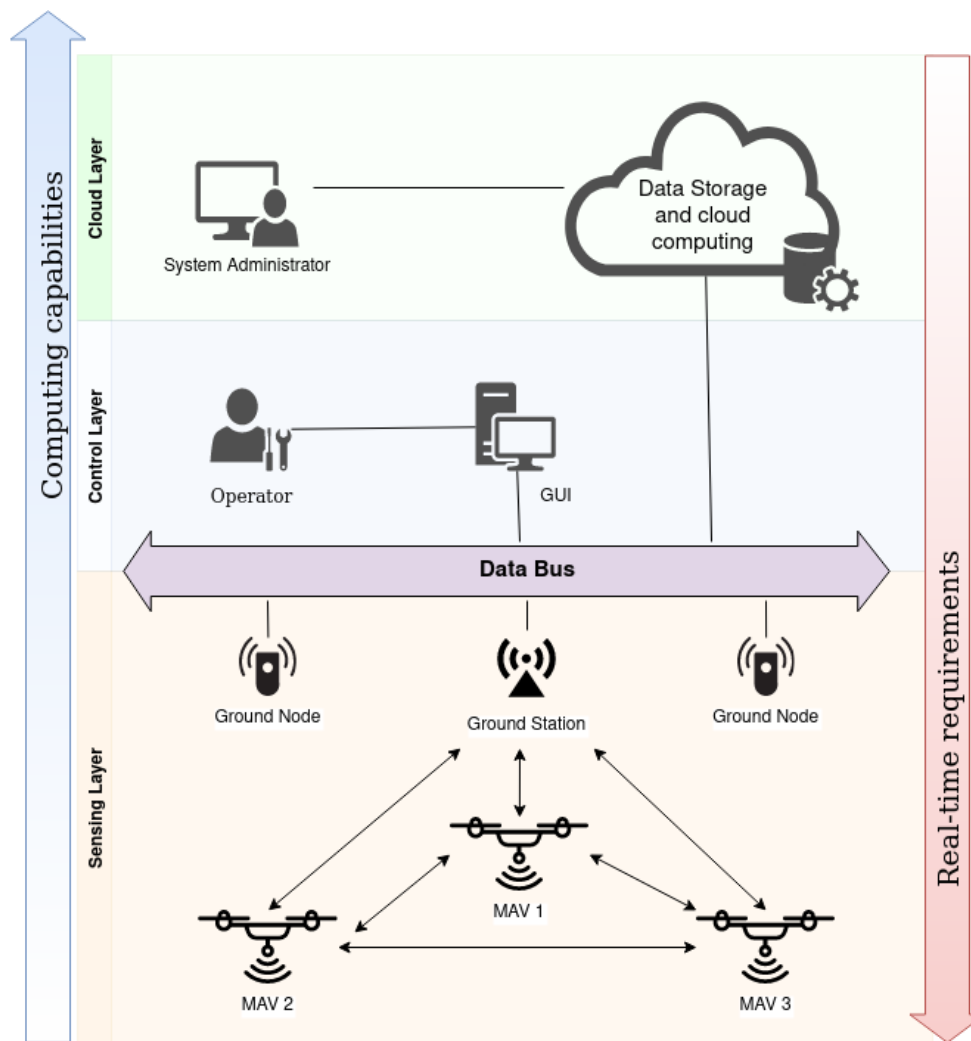


Figure 3.1: System Concept

- **Ground Station:** Responsible for mission planning, control and monitoring of the sensing MAVs. Should provide a GUI for an operator execute basic commands (i.e. "take-off", "land" and "fly to a given point").
- **MAVs:** Should be capable of autonomous flight and performing indoor air quality measurements. As multiple MAVs have to share a flight space, communication between them and collision avoidance maneuvering needs to be implemented in order for the MAVs to navigate through indoor environments, an absolute 3D positioning system is used as complement to the on-board sensors. This system relies on a set of "anchors" which are setup in the Capgemini Engineering "test arena" (Figure 3.8).
- **Cloud Services:** Cloud-based subsystem to store sensor-data and mission logs as well as provide visualization of historical data in a dashboard or web portal. Used to run complex planning algorithms, reducing workload and the need for computing resources in the Ground Station.

- **Ground Nodes:** IoT devices that publish sensor measurements (temperature and humidity) and can interact with the MAVs both by sharing information (i.e. providing humidity sensor readings for compensation of the IAQ values) or by requesting measurements at a given location (alarm behavior).

The system breakdown structure diagram depicted in Figure 3.2 briefly describes the main components of each subsystem. Components in light blue were developed for this project and will be detailed in Chapter 4 while the ones in green are part of the Crazyflie's native firmware and will be explored in the remainder of this chapter.

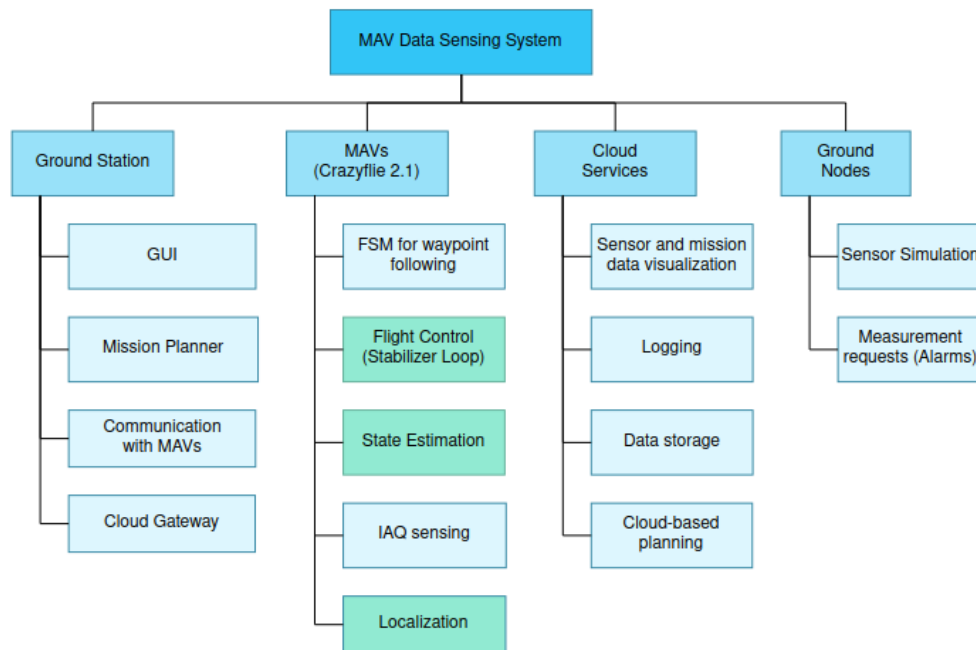


Figure 3.2: System Breakdown Structure

3.2 Description of System Technologies

In order to implement the various subsystems mentioned in the previous section, first it is paramount to study and understand the technologies available for their implementation.

Naturally, the MAVs are one of the more complex subsystems due to their embedded and critical nature. Therefore, an analysis of the hardware architecture, firmware and pre-developed tools must be made prior to implementing new functionalities to determine the viability and limitations of the platform.

3.2.1 The Crazyflie Ecosystem

The MAVs available for developing this application are the Bitcraze Crazyflies 2.1, an "open-source flying development platform" (figure 3.3).



Figure 3.3: The Bitcraze Crazyflie 2.1 nano-quadcopter [66]

Crazyflies are highly versatile due to their open-source nature giving developers and researchers access to both the firmware's source code and the hardware schematics. Additional expansions can be added through the use of *decks* which are functionally similar to shields in the Arduino ecosystem.

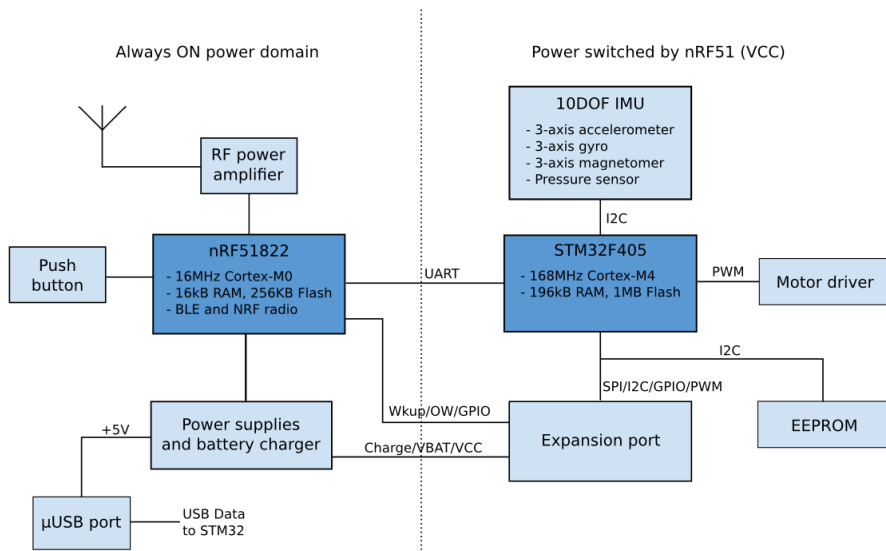


Figure 3.4: Crazyflie 2.X hardware architecture [67]

The Crazyflie has a dual MCU architecture, described in Figure 3.4, which uses a nRF51822 for communications (radio as well as BLE) and power management, the STM32F405 MCU for motor control, sensor measurements and all other computing. The Crazyflie is powered by a

250mAh 3.7V battery which can be charged through a standard micro-USB port. The MAV possesses an on-board 10-DOF IMU composed of a 9-DOF BMI088 IMU to measure orientation and a BMP388 pressure sensor for altitude.

Firmware: The OEM Crazyflie firmware is available at Bitcraze's github repository¹. It is under active development with the latest stable release dating of June 2021. The firmware itself is built on top of FreeRTOS. As the Crazyflie 2.1 is a deeply resource constrained device with a Cortex M4 CPU and a modest 192 kB of RAM, it is unable to run the desktop version of ROS2. However it is one of the supported systems of micro-ROS and therefore can interface with ROS2 systems through the micro XRCE-DDS Agent as described in section 2.2.2. An alternative method that is currently in development is to use the PX4 fast-RTPS bridge mentioned in 2.2.3.1.

Communication protocol: GS-MAV communication is done through the proprietary Crazy RealTime Protocol (CRTP). On the GS side, radio communication is implemented using the Crazyradio PA, a USB dongle which contains a Nordic Semiconductor nRF24LU1 transceiver.

3.2.1.1 Positioning system

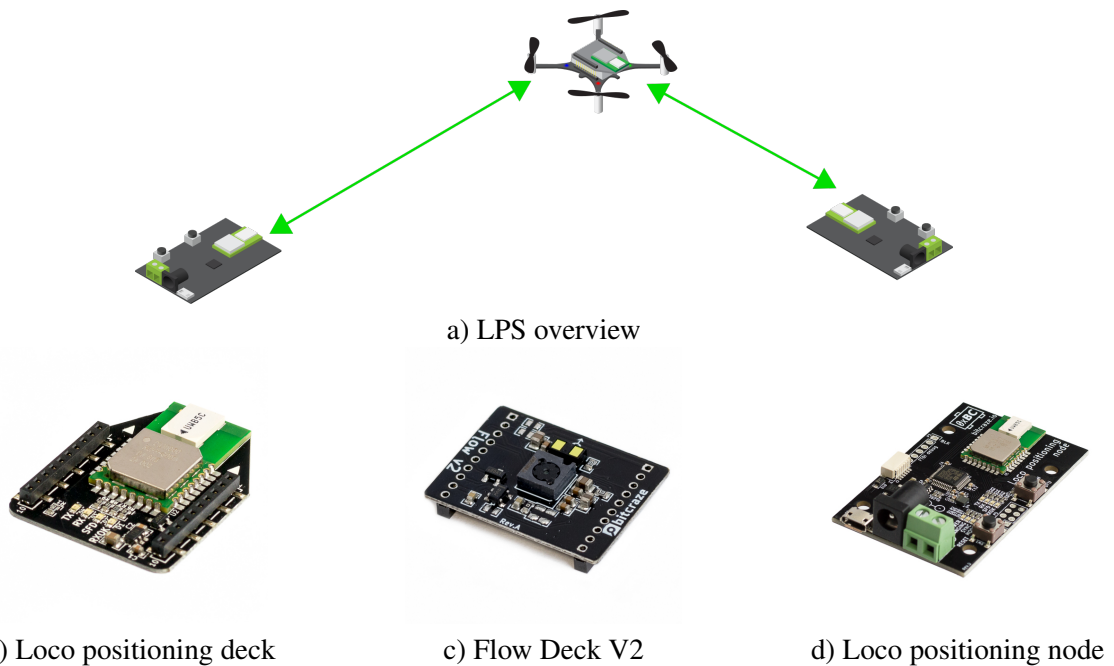


Figure 3.5: Positioning system and its components [68][69][70][71]. Note that the flow deck is not part of the LPS but can be added to the system through sensor fusion.

The absolute 3D positioning system available for this project is the Loco Positioning System (LPS) (cf. Figure 3.5), based on Ultra-Wideband (UWB) localization. The LPS is based on Decawave's DWM1000 IC which has an indoor precision of 10cm and has two components: a deck mounted on the Crazyflie and a set of "anchors" (analogous to satellites in a GPS system). Positioning is done based on Time of flight (ToF) either through Two Way of Ranging (TWR) or

¹<https://github.com/bitcraze/crazyflie-firmware>

Time Difference of Arrival (TDoA). Two separate modes of TDoA are available: TDoA2 in which anchors are pinged in synchronized slots similar to TDMA or TDoA3 which uses randomized transmission scheduling. Position computation is done in the MAV instead of resorting to a GS or edge node. A theoretical minimum of 3 anchors for trilateration are needed although the addition of more anchors improves accuracy and redundancy.

The flow deck V2 is another used deck that is mounted on the bottom of the aircraft and contains both a PMW3901 optical flow sensor and a VL53L1x ToF sensor to measure altitude. The three localization systems can then be fused within the Crazyflie firmware through the use of an extended Kalman filter.

3.2.1.2 The Stabilizer Loop

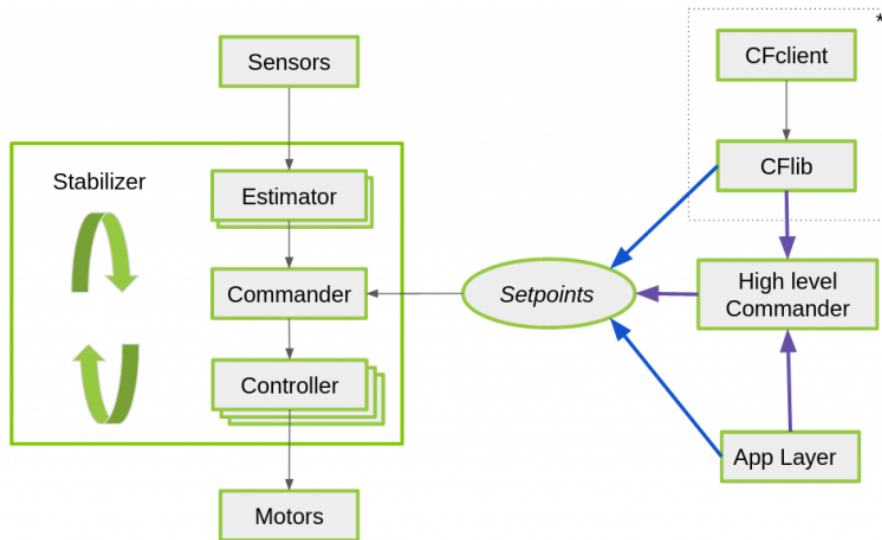


Figure 3.6: Stabilizer module [72]

In the Crazyflie 2.1, flight control and stabilization of the quadcopter is done in the firmware through the stabilizer (which can be seen in Figure 3.6), this module has three main components:

- **High-level commander (HLC):** The HLC is, like the name implies, a module that receives high-level commands such as "take-off", "land", "go-to" and uses a *planner* to generate smooth trajectories based on 7th-order polynomials. These trajectories are then translated to setpoints that are forwarded to the commander. In this project, commands are issued within the "app layer" - where custom code is implented in the Crazyflie's firmware.
- **Estimator:** State estimation in the Crazyflie is done using an Extended Kalman Filter (EKF). The EKF is a recursive filter for non-linear systems that is used to estimate the state of the MAV (orientation, position and velocity) based on multiple sensor inputs (in this case from the IMU, flow deck and LPS) as well as the measurement noise model and dynamic model of the MAV.

- **Controller:** The controller is responsible for navigating the MAV between the current position and the setpoint. The Crazyflie supports three different controller implementations: cascaded Proportional Integral Derivative (PID), Mellinger, and Incremental non-linear dynamic Inversion (INDI) controllers. As the MAV only needs to perform simple movements and the absolute positioning system used is not the most accurate, the PID is used for this project due to its simplicity. The cascaded PID controls the Crazyflie in three different levels: position/velocity, attitude and angle rate according to Figure 3.7.

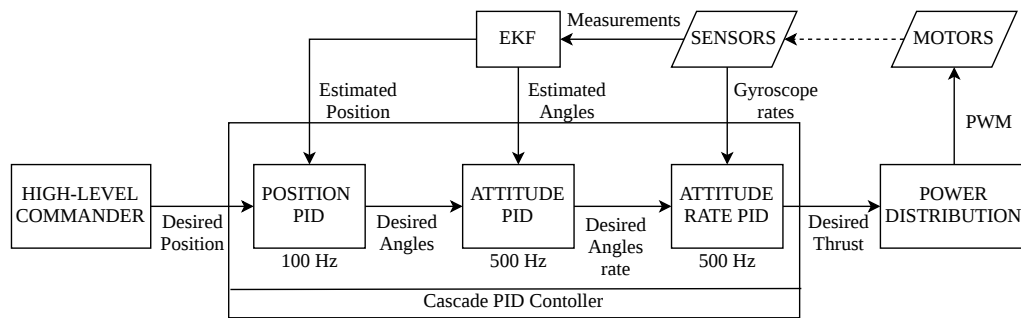


Figure 3.7: PID controller architecture [73]

3.2.2 Additional tools and frameworks

Naturally, as the main goal of the dissertation is to validate and evaluate ROS2, most of the tools and frameworks lie within ROS2 ecosystem. Due to being an heterogeneous full-stack system, frameworks and programming languages vary for each subsystem:

- **Crazyflie firmware:** The firmware of the Crazyflies is written in C due to being an embedded system with hard real-time requirements. ROS2 integration is done using micro-ROS and its *rcl* client library which is also written in C.
- **Ground Station:** Desktop version of ROS2 Foxy using the *rclpy* or *rclcpp* client libraries which are written in more high-level languages. Connection with the MAVs is done through a micro-ROS agent and can also be cross-compiled to the ARM architecture allowing the GS to be run on a embedded device such as a Raspberry Pi.
- **GUI:** For GUI development QT² is a common choice for ROS based systems with various GUI tools for ROS are available. QT is a cross-platform C++ framework with bindings for python available in PyQt. The GUI can be run on the Ground Station or on another PC.
- **Ground nodes:** ROS2 foxy running on Ubuntu Server 20.04. For simplicity, these nodes are simulated using Raspberry Pis 4.
- **Cloud Services:** AWS EC2 IaaS for running ROS2 nodes on the cloud and AWS Cloudwatch for data storage and visualization.

²<https://www.qt.io/>

3.2.3 Test Arena

At Capgemini Engineering, flight tests are performed in a enclosed space that will henceforth be referred to as the "arena" (Figure 3.8). The arena not only allows the MAVs to be flown safely as the entire arena is enclosed by a safety net in case of crashes but also hosts the absolute positioning system described in subsection 3.2.1.1.

The LPS is setup with 8 anchors positioned as the vertices of a 2.17x2.01x2.10 meter cuboid. UWB communication is done using TDOA2; we use TDOA instead of TWR as it is desired to fly multiple MAVs and opted for TDOA2 instead of TDOA3 as the arena has a box shape and in this case TDOA2 has an alleged slightly better precision³.

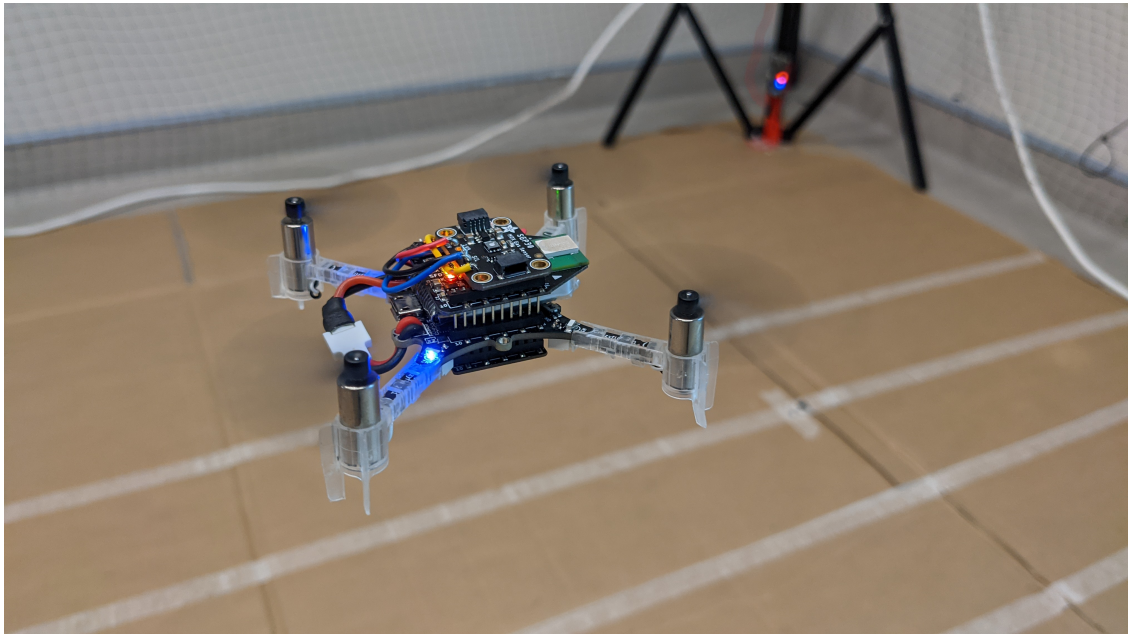
In Figure 3.8, the ground-station can also be seen. It consists of a Raspberry Pi with a Crazyradio PA used to communicate with the MAVs connected to a router, allowing devices in the Local Area Network (LAN) to interact with the Crazyflies. Additionally, in order to avoid interference, the router only uses the 5GHz band for Wi-Fi as the MAVs use the 2.4GHz for GS-MAV communication and peer-to-peer communication.



Figure 3.8: Test arena at Capgemini Engineering. Color code: Orange - LPS anchors (bottom right one is used as the referencial for the *odom* frame), Red - MAVs, Green - Ground Station (Raspberry Pi) with a crazyradio PA

³<https://www.bitcraze.io/documentation/repository/lps-node-firmware/master/functional-areas/tdoa2-vs-tdoa3/>

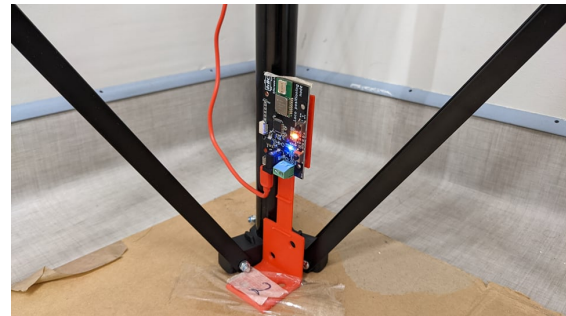
The main hardware components of the system can be seen in Figure 3.9.



(a) Crazyflie hovering in the middle of the arena



(b) Crazyflie 2.1 (with IAQ sensor mounted)



(c) LPS anchor

Figure 3.9: System Hardware

3.3 Summary

In this chapter, an high-level explanation of the MAV-based IAQ sensing use-case was provided. The proposed system can be broken down into four highly intertwined subsystems: Ground Station, MAVs, Cloud Services and Ground Nodes. Our approach is to use ROS2 (and micro-ROS) as a way to increase the interoperability of these systems. An analysis of the hardware/software architecture of these systems resulted in some considerations that should be taken into account during the implementation:

- The Crazyflie's firmware is completely open-source and provides an "app layer" which can be used to run custom code. Due to the low processing power of the MAV the desktop

version of ROS2 can not be used. However, the MAV is compatible with micro-ROS which can be used to interface with ROS2/DDS with the help of an *agent* running on the GS.

- Communication with the MAVs uses a proprietary radio link. An implication of this is that all communication with the MAVs must be done through the GS which is the only entity that possesses a crazyradio PA. Nonetheless, using the DDS middleware can provide abstraction and facilitate the communication of other devices with the MAVs.
- The Crazyflie firmware provides high-level functions for motion control through the HLC module. This functionality can be exposed as micro-ROS/ROS2 services, allowing other DDS participants to interact with the MAV.
- Peer-to-peer (brokerless) communication between MAVs is available through radio broadcast. However, micro-ROS does not support this functionality implicating that all MAV-MAV communication must be done through the agent or without using micro-ROS.
- Position estimation is done in the crazyflie's firmware through the sensor fusion of its on-board sensors and the external positioning system (LPS using TDOA2) which is setup in the "test arena".

Chapter 4

Implementation

This chapter aims to provide a detailed description of the implementation and interaction of all software and hardware components of the proposed system. Section 4.1 covers the working principle of communicating with MAVs, Section 4.2 details how the MAV performs a mission internally, mission planning is then explained in section 4.3. Section 4.4 encompasses the hardware and firmware integration of the IAQ sensor. Cloud services are described in section 4.5 and finally the visualization tools are presented in Section 4.6.

4.1 Communication Architecture

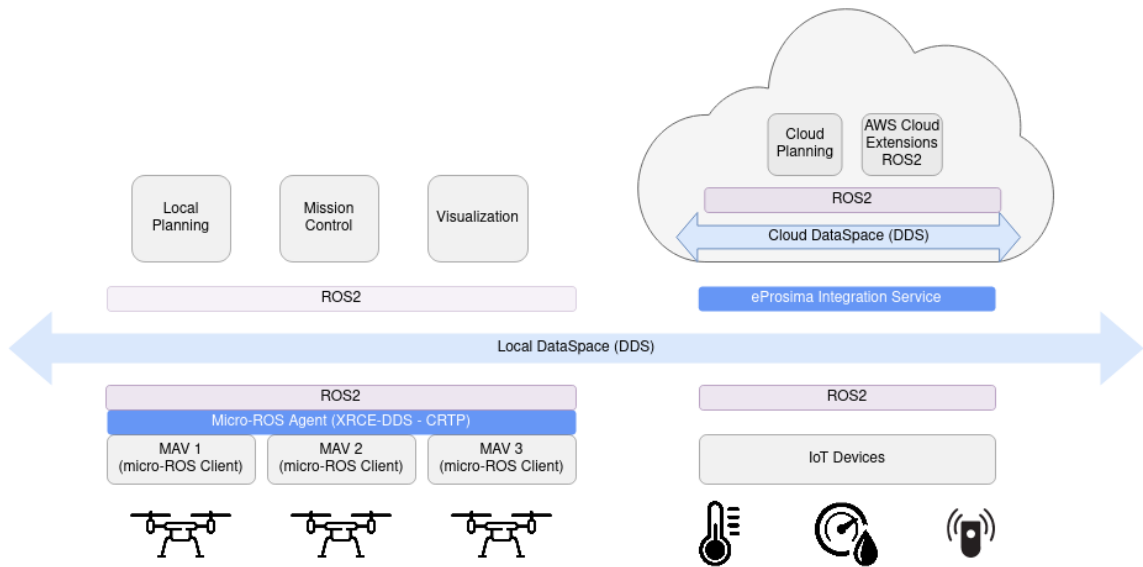


Figure 4.1: Communication Architecture

The diagram in Figure 4.1 summarizes the communication architecture of the proposed system. At the core of communication architecture is the DDS databus, the data-centric middleware

used by all software components as the default middleware of ROS2 and the micro-ROS Agent. Having a common middleware allows interoperability between all systems despite having very different computing resources and operating systems.

4.1.1 GS - MAV Communications

As stated in section 2.2.2, micro-ROS communication uses Micro XRCE-DDS. In this middleware, the MAVs are clients and the Ground Station is a server. The GS runs an agent who acts as a proxy to the MAV and performs operations in the DDS world on its behalf (figure 4.2). As the agent is in charge with external communication from/to the MAVs, all communication to the MAVs has to pass through the ground station as represented in figure 4.3. Nevertheless, the use of the DDS middleware makes this transparent to other machines as they only need to interact with the logical ROS2 nodes representing the MAVs that are created by the agent.

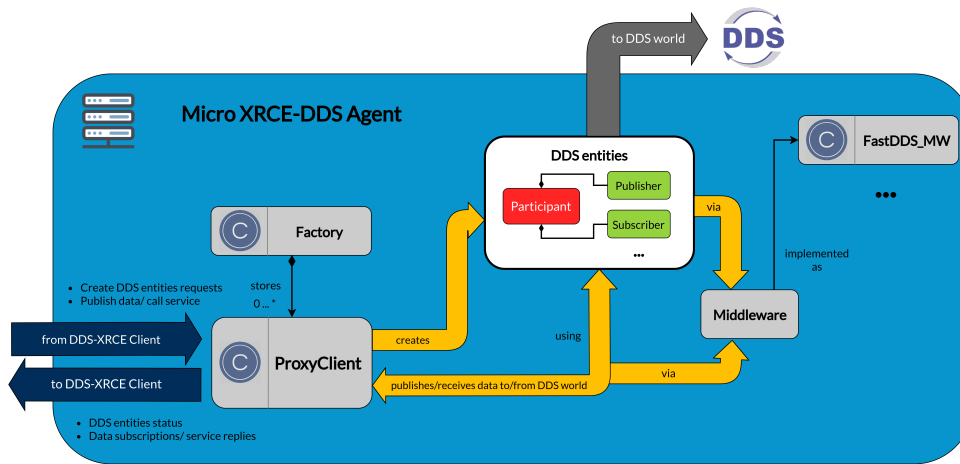


Figure 4.2: Micro XRCE-DDS Agent working principle [74]

However, as the transport layer of Micro XRCE-DDS middleware does not support CRTTP, the communication protocol of the Crazyflie, additional steps are needed to establish GS - MAV communication. Existing approaches such as eProsima's *crazyflie demo*¹ use a custom transport in the client-side to transmit XRCE-DDS messages over CRTTP and then resort to software bridges on the agent side that use the *cflib* python3 library to connect to the Crazyflie. Then, the CRTTP stream is converted to serial and relayed to a serial Micro XRCE-DDS Agent. Similarly, in this work we use the same logic on the client-side but implement a custom CRTTP agent using the micro-ROS custom transport API. This is to the authors knowledge the first agent-side implementation of a custom transport.

4.1.2 Custom CRTTP Micro XRCE-DDS Agent

As the name implies, our custom agents uses CRTTP as its transport protocol. In order to interface with the transport layer, micro-ROS provides an API that allows the implementation of custom

¹https://micro.ros.org/docs/tutorials/demos/crazyflie_demo/

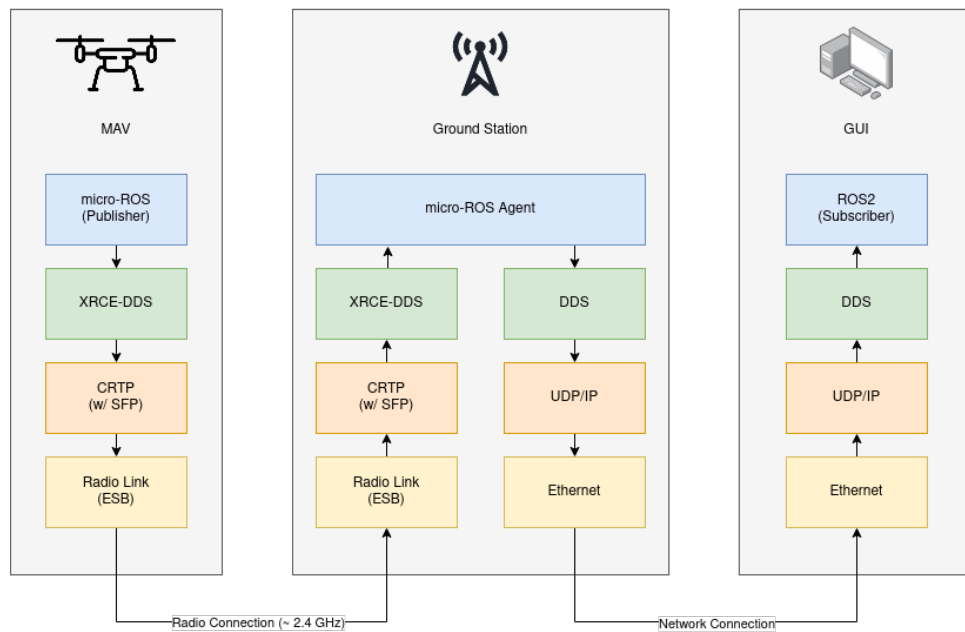


Figure 4.3: Communication stack

transports. This implementation consists of defining callbacks both in the client and agent side for opening, reading from, writing to and closing the custom transport.

Connection to the crazyflies is established through *crazyflie-link-cpp*², an open-source C++ library developed by Bitcraze which is then linked against the custom Micro XRCE-DDS agent.

Framing Protocol

An important aspect of the custom agent's working principle is the usage of the *Serial Framing Protocol* which uses High-level Data Link Control (HDLC) framing and CRC Calculation to provide integrity and security to the framing³. However, the usage of XRCE-DDS and its framing protocol adds a considerable overhead to the communication. To better understand this overhead, let us consider the custom *cf_messages/AirQuality* message that uses type composition to include the *geometry_msgs/Point32* type from the ROS2 common_interfaces package.

Listing 4.1: cf_messages/AirQuality

```
geometry_msgs/Point32 p
uint16 eco2
uint16 tvoc
```

Listing 4.2: geometry_msgs/Point32

```
float32 x
float32 y
float32 z
```

²<https://github.com/bitcraze/crazyflie-link-cpp>. At the time of writing, there are plans to change the default method for communicating with Crazyflies in *cfli* to this library through the use of Python wrappers.

³<https://micro-xrce-dds.docs.eprosima.com/en/latest/transport.html>

It is straightforward to understand that the payload to publish a message from this type is equal to $32 \cdot 3 + 16 + 16 = 128$ bits, or 16 bytes. Next, the XRCE header and submessage header (12 bytes) are added to the frame as well as the HDLC header (5 bytes) and the CRC (2 bytes) totalling 19 bytes of overhead. Note that in the case of sequences, an additional 4 bytes are used in the payload to indicate the length of the data array.

Additionally, as CRTP packets have a maximum payload of 31 Bytes (with a 1 Byte header) this message consisting of 16B payload and 19B overhead needs to be fragmented into two packets with lengths of 32B and 5B respectively accounting for the 2B of additional overhead caused by the CRTP headers. This process can be seen on figure 4.4.

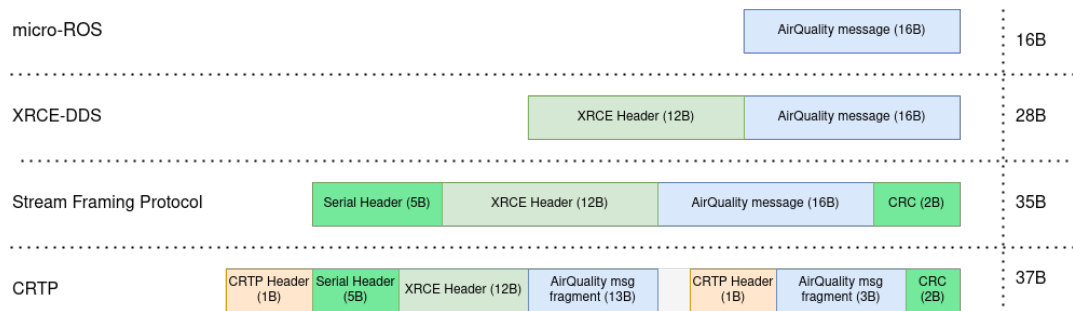


Figure 4.4: Encapsulation of micro-ROS messages in MAV - GS communication

The working principle of the SFP can be seen in figure 4.5. A relevant aspect of this protocol is that the framing buffer that has a fixed size of 42 bytes.

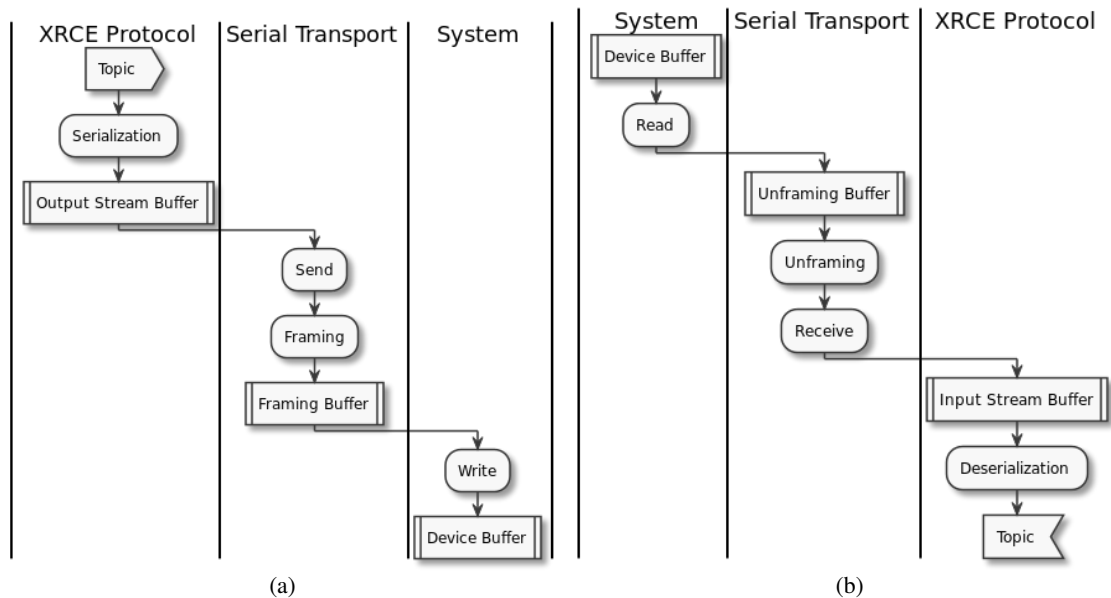


Figure 4.5: Serial Framing Protocol working principle [75]. a) Client-side (MAV) sending, b) Agent-side receiving.

According to the official documentation, the transport writing function must be implemented to always try to write up to 42 bytes, else it timeouts. An implication of this is that the number of

CRTP packets needed to send a XRCE message is increased.

Considering a payload p_L and the number of CRTP packets needed n_{pkts} , the number of packets sent before occurring the timeout is given by n'_{pkts} as:

$$n'_{pkts} = \text{floor}(p_L/40) \cdot 2 \quad (4.1)$$

Note that 40 was used as the headers of the CRTP packets are taken into account. Thus, the total number of packets for a given payload size p_L , can be modelled by the following mathematical expression:

$$n_{pkts} = \begin{cases} n'_{pkts} & p_L \bmod 40 = 0 \\ n'_{pkts} + 1 & 0 < p_L \bmod 40 \leq 30 \\ n'_{pkts} + 2 & p_L \bmod 40 > 30 \end{cases} \quad (4.2)$$

Remark regarding GS - MAV Communication Representation

In the remainder of this document, while communications that involve the agent are performed according to the sequence diagram in figure 4.6a, they are represented in the form of 4.6b, omitting the agent for simplification.

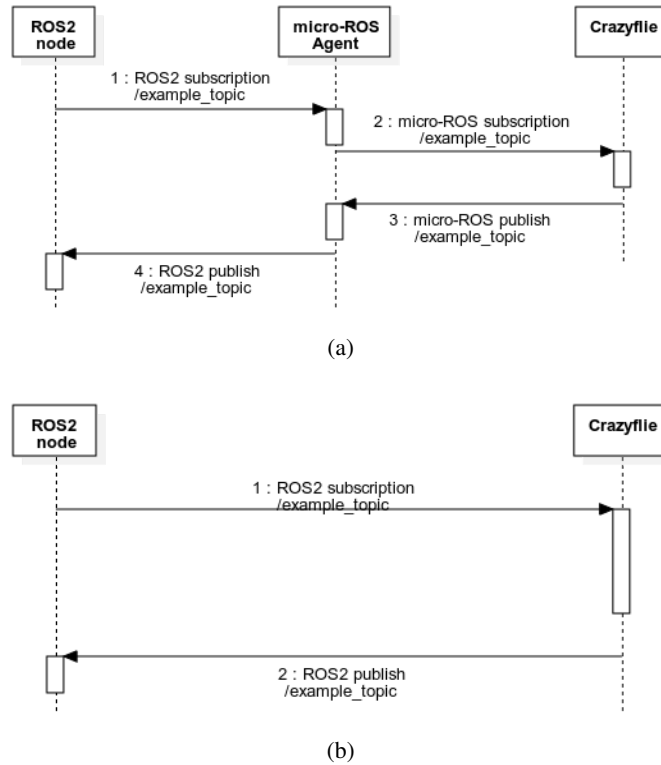


Figure 4.6: Micro-ROS communication example, a) Pre-simplification, b) Post-simplification

This representation is not how communication is actually performed but is valid from a practical standpoint as the agent is transparent in micro-ROS communication.

A complete listing of the topics/services used per software component and the definition of custom defined types is described in appendixes [A.1](#) and [A.2](#) respectively.

4.2 Motion control and plan execution

4.2.1 Waypoint-based IAQ sensing

In this use-case, a mission consists of a single MAV performing a trajectory through multiple sensing waypoints, performing IAQ measurements in each one and publishing them on the DDS data space using micro-ROS.

This is accomplished by using a Finite State Machine (FSM) represented in figure 4.8 implemented on the MAV's firmware app layer which interacts both with the HLC module and the micro-ROS logic. It is desired to both allow the MAV to execute a sensing mission autonomously and to perform simple manual motion commands for use in other Capgemini Engineering R&D projects.

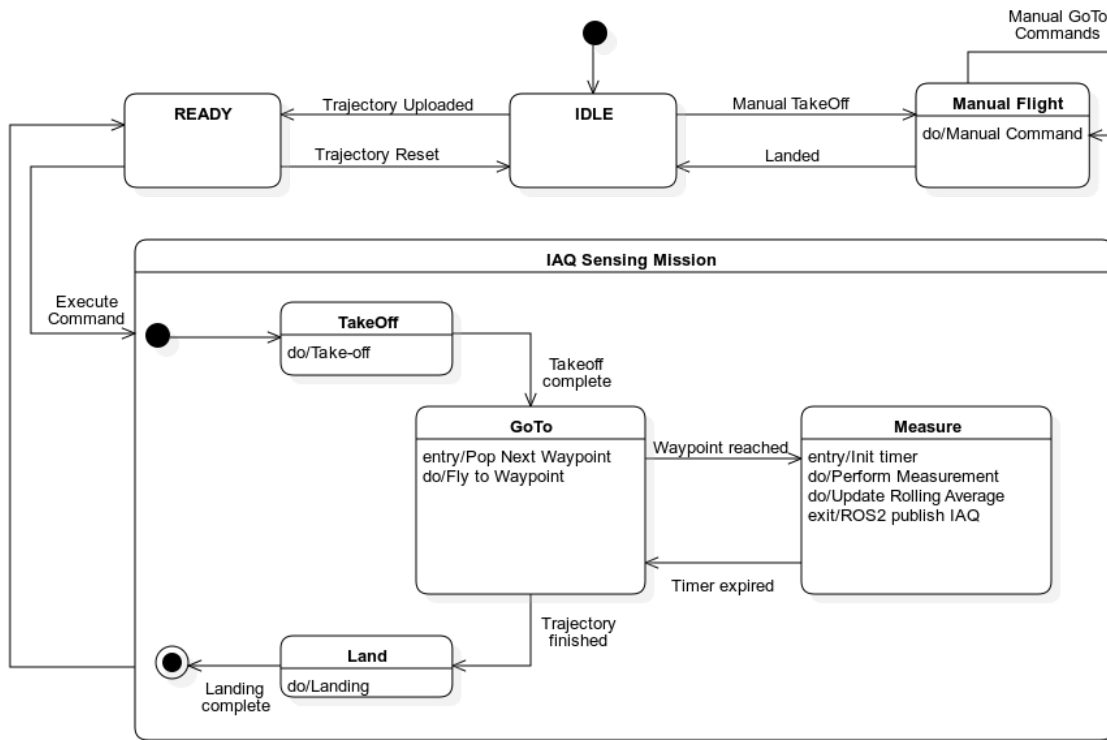


Figure 4.7: Main mission control FSM

The MAV's initial state, "Idle", represents that the drone is currently without any mission plan and can both be piloted manually or a mission can be uploaded. The "Ready" state indicates that the MAV has been uploaded with a trajectory and can execute a mission.

Mission execution consists of performing a "TakeOff" and then a simple loop that consists of flying to each waypoint of the trajectory, "GoTo", and performing measurements, "Measure",

during a pre-configured time interval, and finally "Land" after all measurements are complete. During the measurement time, a rolling average of the sensor valued and the MAV position is performed in accordance to equation 4.3, where x represents a generic variable and n the sample size.

$$x_{avg} = x_{avg} + \frac{x_i - x_{avg}}{n} \quad (4.3)$$

4.2.2 Collision Avoidance System

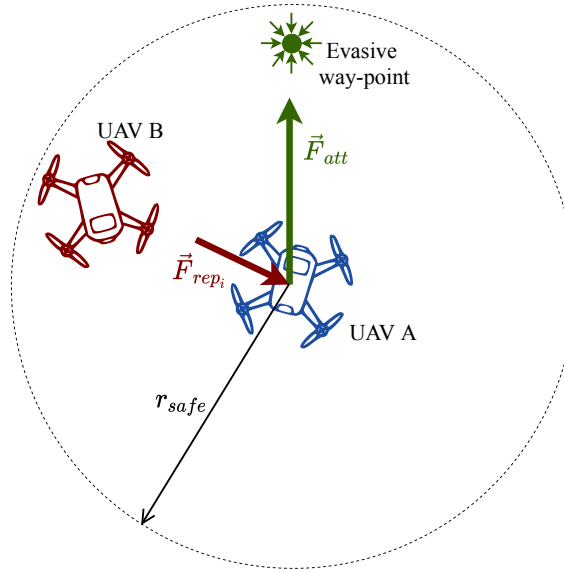


Figure 4.8: Visual representation of the CAS [73]. In this scenario, the MAV is subjected to a potential field caused by an attractive force to the goal, $\vec{F}_{att}$ and a repulsive force, $\vec{F}_{rep}$ due to other MAVs. As no geometric evasion is performed, the evasive waypoint corresponds to the current target.

Mid-flight collisions between MAVs are prevented with the integration of a Collision Avoidance System (CAS) that was previously developed in Capgemini Engineering by Sanches et. al. [73]. As the Crazyflies do not have any sensor for detecting its peers, detection of other MAVs and collision-prone situations is done through inter-drone communication. Communication between MAVs is done using CRTP broadcasting through Bitcraze's P2P API as brokerless P2P communication in micro-ROS is still in development. Each MAV broadcasts its id, position and linear speed.

A collision scenario is considered whenever an UAV enters another's safe zone which the authors consider a sphere of radius r_{safe} (in this project due to the arena's dimension constraints we use a radius of 0.5m). Avoidance maneuvers follow an approach based on artificial potential fields where the current goal (in this case the current waypoint) has an attractive potential whereas other MAVs have a repulsive one.

4.3 Trajectory and mission planning

4.3.1 Optimal sensing trajectories

Mission planning for UAVs is a prime-example of a vehicle routing problem (VRP), a class of optimization problems whose objective is to plan routes for a fleet of vehicles to visit a set of locations departing from set of depots. Additional constraints such as maximum capacity (i.e. cargo) per vehicle or time windows for visiting each location are typical in this type of problems.

The Multi-Depot Multiple Travelling Salesman Problem (MmTSP) can be viewed as a relaxation of the VRP [76] without capacity constraints. In this problem, a number of travelers (in this case MAVs) depart from a set of depots (locations where the MAVs can take-off/land safely), visit a number of nodes (sensing waypoints) and return to the same depot where they started the mission (fixed destination variant).

4.3.1.1 Mixed-Integer Linear Programming (MILP) formulation

As the MmTSP is a well studied optimization problem in the literature, we use a modified version of an existing integer linear programming formulation proposed by Kara et al. in [77] to tackle this problem.

In our use-case, the main goal is to optimize the mission time by minimizing the most time-consuming trajectory between all MAVs. Assuming that the energy consumption is directly related to the time needed for the MAVs to perform the mission this also allows for the optimization of the overall energy consumption (this assumption is asserted in section 5.2.4).

For each MAV, we can express the total time needed to complete the mission, t_{tot_k} , in two components: the time needed to move between sensing waypoints, t_{m_k} , and the time needed to perform the measurements, t_{s_k} .

$$t_{tot_k} = t_{m_k} + t_{s_k} \quad (4.4)$$

Considering a complete graph $G = (V, A)$, where V is a set of nodes that comprises both waypoints V' and depots D such that $V = D \cup V'$ and A is the set of edges (i, j) between each node. The associated cost matrix C contains the length of each arc which corresponds to the euclidean distance (assuming an open space) between nodes. We consider a binary variable x_{ijk} that equals 1 if the arc (i, j) is assigned to MAV k and 0 otherwise. The distance/cost of traversing this arc is given by c_{ij} . Finally, considering that each depot can only accommodate one MAV, $k \in D$.

The time needed by a MAV k to travel between measurement points, t_{m_k} , is given by equation 4.5. In this application, the linear speed, v , between each point is always constant. Therefore, t_{m_k} is given by the sum of the lengths of each arc in the solution dividing by the linear speed.

$$t_{m_k} = \frac{1}{v} \cdot \left(\sum_{j \in V'} (c_{kj}x_{kjk} + c_{jk}x_{jkk}) + \sum_{i \in V'} \sum_{j \in V'} c_{ij}x_{ijk} \right) \quad (4.5)$$

Before defining the time needed for a MAV k to perform all measurements t_{m_k} , we need to consider the time $t_{measure}$ needed to perform a single measurement. In a sensing application, this is given by the quotient of the number of samples, N and the sampling rate of the sensor f_{sample} that can be defined by the end user.

$$t_{measure} = \frac{N}{f_{sample}} \quad (4.6)$$

Thus,

$$t_{s_k} = t_{measure} \cdot \sum_{i \in V'} \sum_{j \in V'} x_{ijk} \quad (4.7)$$

Finally, the optimization problem results in a min-max cost function in which we minimize the lengthiest trajectory whose duration is given by t_{max} :

$$t_{max} = \max\{t_{tot_1}, t_{tot_2}, \dots, t_{tot_k}\}, \quad k \in D \quad (4.8)$$

Next, we add the constraints to the optimization problem. Similiarly, to Kara et al [77], u_i is defined as the number of waypoints visited before the i -th waypoint and we bound the number of waypoints each MAV can visit between W_{min} (minimum) and W_{max} (maximum) allowing for customization of the swarm behavior (i.e. only using multiple MAVs for a high number of waypoints):

$$\text{minimize } t_{max} \quad (4.9a)$$

$$\text{subject to } \sum_{j \in V'} x_{kjk} = 1 \quad k \in D, \quad (4.9b)$$

$$\sum_{k \in D} x_{kjk} + \sum_{k \in D} \sum_{i \in V'} x_{ijk} = 1 \quad j \in V', \quad (4.9c)$$

$$x_{kjk} + \sum_{i \in V'} x_{ijk} - x_{jkk} + \sum_{i \in V'} x_{jik} = 1 \quad k \in D, j \in V', \quad (4.9d)$$

$$\sum_{j \in V'} x_{kjk} - \sum_{j \in V'} x_{jkk} = 0 \quad k \in D, \quad (4.9e)$$

$$u_i + (W_{min} - 2) \sum_{k \in D} x_{kik} - \sum_{k \in D} x_{ikk} \leq W_{min} - 1 \quad i \in V', \quad (4.9f)$$

$$u_i + \sum_{k \in D} x_{kik} + (2 - W_{min}) \sum_{k \in D} x_{ikk} \geq 2 \quad i \in V', \quad (4.9g)$$

$$\sum_{k \in D} x_{kik} + \sum_{k \in D} x_{ikk} \leq 1 \quad i \in V', \quad (4.9h)$$

$$u_i - u_j + W_{max} \sum_{k \in D} x_{ijk} + (W_{max} - 2) \sum_{k \in D} x_{jik} \leq W_{max} - 1 \quad i \neq j, (i, j) \in V', \quad (4.9i)$$

$$t_{tot_k} \leq t_{bat} \quad k \in D, \quad (4.9j)$$

$$x_{ijk} \in \{0, 1\} \quad (i, j) \in V, k \in D \quad (4.9k)$$

Constraint 4.9b enforces that no more than one MAV can take-off from each depot, constraint

4.9c makes sure that each sensing waypoint is only visited once and by a single MAV, 4.9d and 4.9e are flow conservation constraints that guarantee that the number of MAVs leaving a waypoint is the same as the ones entering, 4.9f, 4.9g and 4.9h are the bounding constraints (in case no lower bounding is used and therefore $W_{min} = 0$ then x_{ikk} is equaled to zero in 4.9g and constraint 4.9h is omitted from the formulation), 4.9i is the subtour elimination constraint (SEC)⁴. Finally, constraint 4.9j ensures that the total time of the mission does not exceed a conservative estimate, t_{bat} , of the MAV's maximum time of flight as allowed by its battery.

Another interesting aspect of this formulation is that constraint 4.9b ensures that each available MAV is used in the sensing task. As this can be non-ideal in situations where MAVs are very far apart, the feature can be disabled by turning the constraint into an inequality where the number of MAVs leaving the depot is less or equal to one. This, however, leads to an increase of the computation time.

4.3.1.2 ROS2 Integration

In order to formulate and solve the optimization problem, a python3 module named PuLP⁵ was used. This module can generate the problem formulation which is solved using a solver. In our approach, we used the default open-source C++ MILP solver CBC⁶.

Integration with ROS2 was done through the *rclpy* client library to create a *Planner* server node. This server node provides the */Planner/plan_trajectories* service that given the MAVs positions (depots) and the waypoints returns the optimal sensing trajectories.

Graphical User Interface (GUI): Although any ROS2 node can be a client of the *Planner* node, in an attempt to provide an intuitive way of a human operator to plan and execute missions a GUI was developed in the form of an RQT plugin. The decision to create a plugin instead of a standalone graphical application was made in order to easily integrate other GUIs developed by the Capgemini Engineering R&I team.

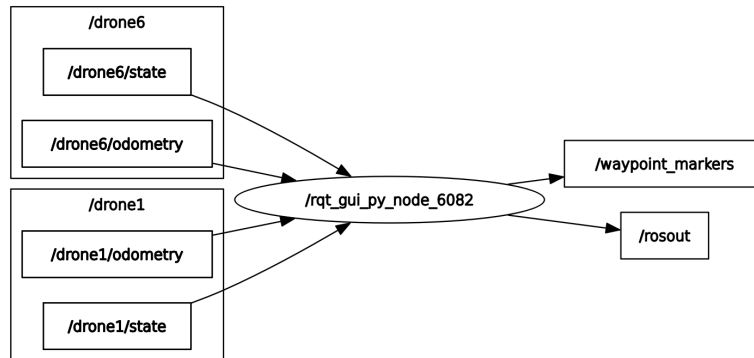


Figure 4.9: GUI representation in rqt_graph. Non-relevant nodes/topics have been omitted.

⁴ A SEC forbids solutions containing a tour or multiple tours that do not start and end in a depot. In the TSP case, this corresponds to solutions composed of multiple tours that only visit a subset of the waypoints (subtours).

⁵<https://github.com/coin-or/pulp>

⁶<https://github.com/coin-or/Cbc>

In order to perform the planning request, the node associated with the GUI needs to know the location and state of the MAVs. Therefore, the node subscribes to the state and odometry topics of all MAVs in accordance to figure 4.9. Additionally, markers representing the sensing waypoints are published for visualization purposes.

The GUI, depicted in figure 4.10, supports the following functionalities:

- Manual take-off and landings.
- Establishing communication with the MAVs.
- Simulating Alarms for testing purposes.
- Loading waypoints.
- Planning, uploading and giving the execute command to the MAVs.
- Visualization of the trajectories with the help of a custom Matplotlib-based widget.

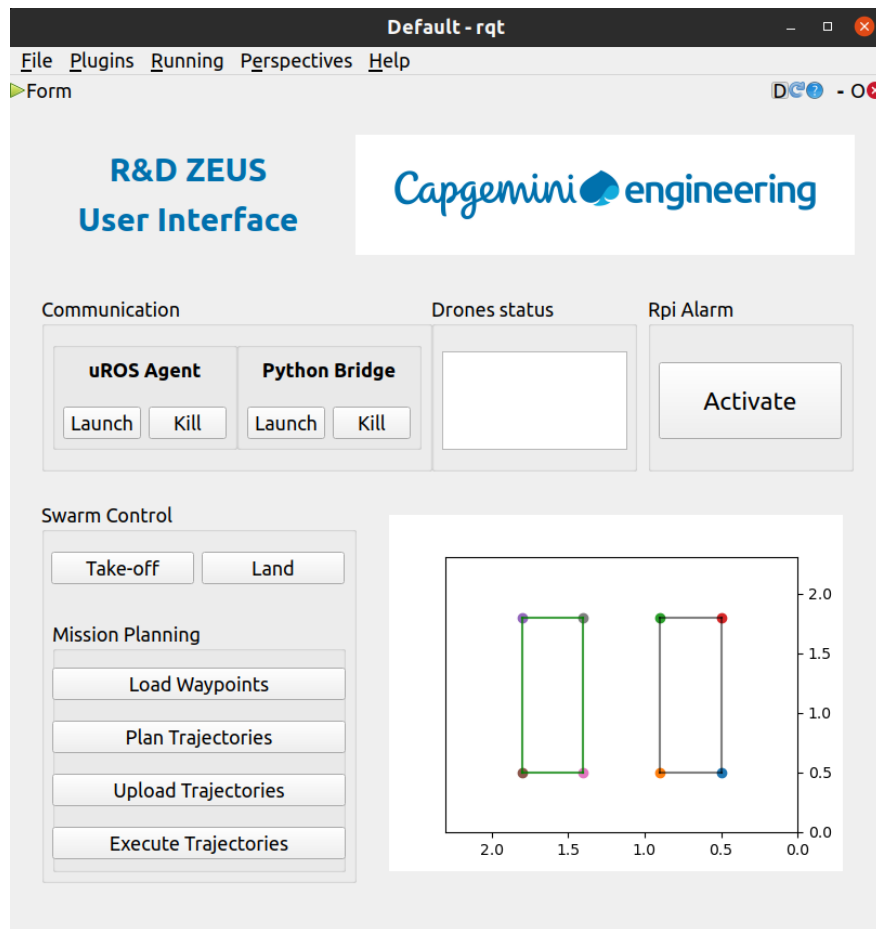


Figure 4.10: Mission planning GUI

Lastly, the complete planning procedure is represented by the sequence diagram in figure 4.11.

The `/drone<id>/state` topics are used to ensure that a trajectory upload is only done when the MAV is "idle" meaning the MAV is grounded and its flight plan is empty. Moreover, the MAV can only execute a trajectory if the MAV is "ready" meaning that the trajectory has been written in the MAV's internal memory. These "guard conditions" are also ensured in the MAV firmware by checking the state in the service callback and returning the appropriate value in the service's response.

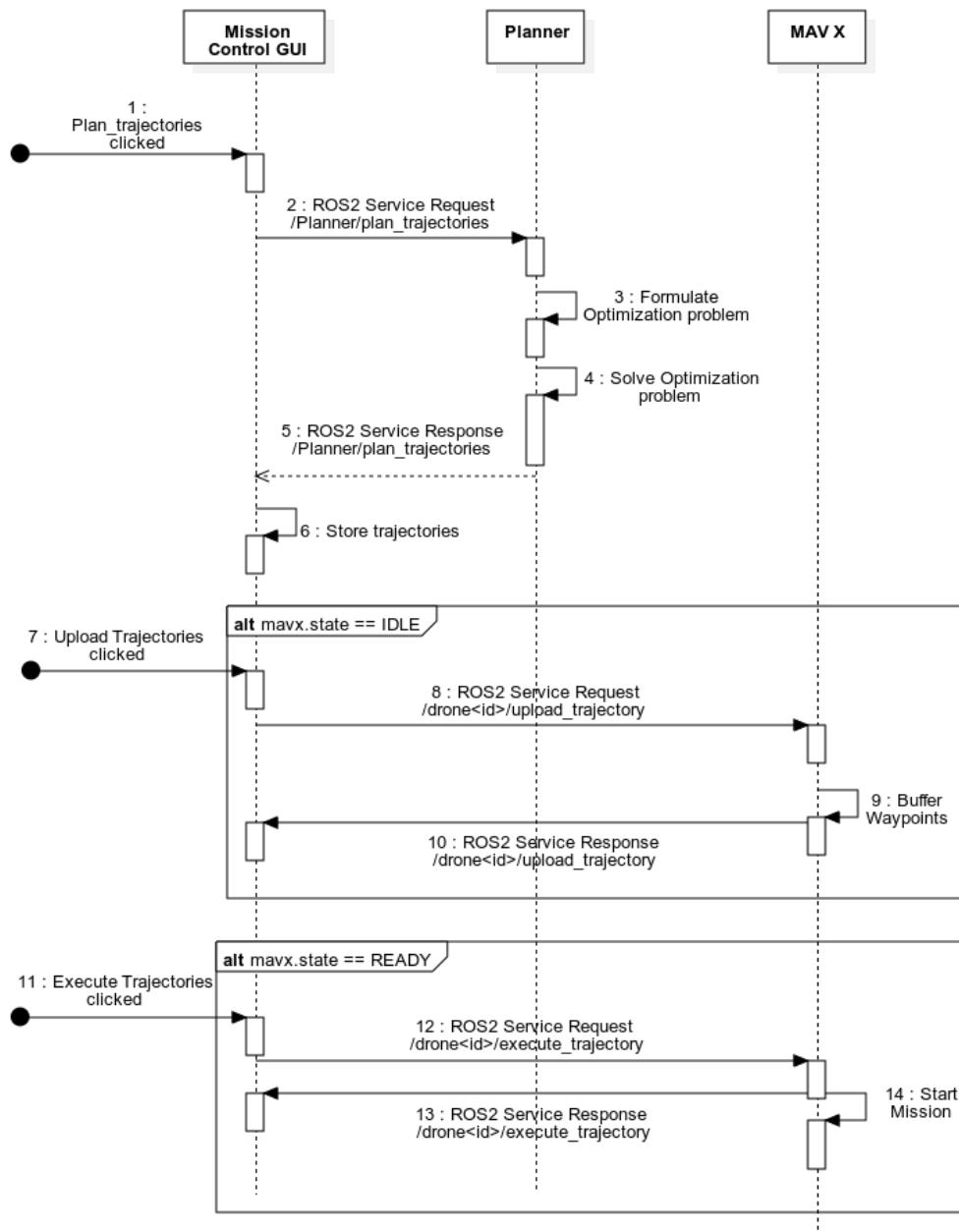


Figure 4.11: Sequence diagram of the trajectory planning triggered by a human operator using the GUI

4.3.2 Auction-based algorithm for event-triggered (alarms) sensing

4.3.2.1 Algorithm outline

In addition to pre-determined flight plans, the ability to request a MAV to perform high priority measurements (alarms) in other locations either by a human operator or another system such as IoT sensor is a common use-case. With this in mind, an algorithm to assign MAVs to these "alarms" was developed.

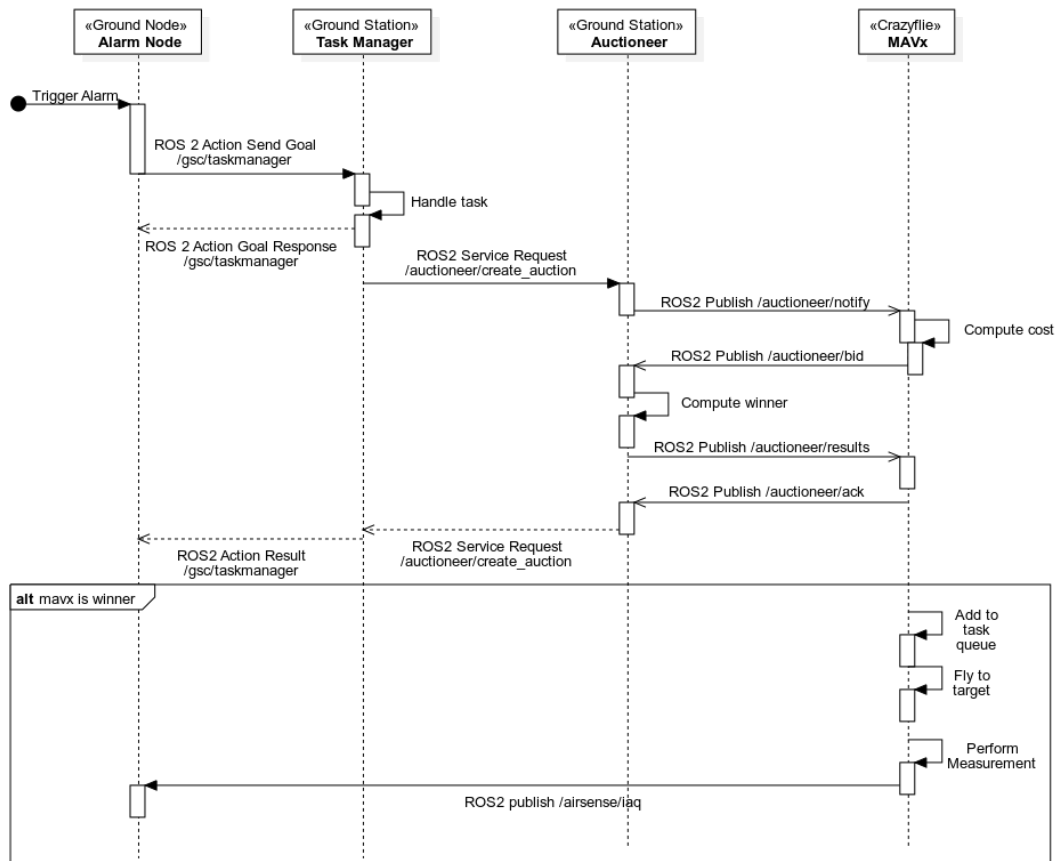


Figure 4.12: Sequence Diagram for the "auction" algorithm

The algorithm is represented in figure 4.12. The main idea behind this algorithm is that an entity called the *auctioneer* (ROS2 node on the GS) notifies all available MAVs of the alarm and each MAV places a *bid* with the amount of time it needs to perform the measurement. This computation is done in the MAV as by our design it is the only entity which has simultaneous knowledge of: a) its flight plan for both normal sensing tasks and queued alarms, b) its battery and location and c) its current measurement status - in case the *auction* started during a measurement. The MAV that can perform the task in less time is then assigned the task. The way the MAVs interact with the topics is shown on figure 4.13.

There are four main partaking entities in this procedure:

- **Alarm Node:** Triggers the alarm which starts the *auction*. Executed by the ground nodes.

- **Task manager:** Responsible for queueing incoming alarms while an *auction* is taking place. In order to process requests from multiple alarms at the same time this node uses the *MultiThreadedExecutor* in conjunction with the *ReentrantCallbackGroup* in order to execute callbacks in parallel,
- **Auctioneer:** Node that notifies all MAVs of the alarm and then selects the best bid.
- **MAVx:** Representation of a generic MAV, is responsible for computing the cost of performing the alarm measurement, bidding in the *auction* and then performing the measurement in case it is the most apt one.

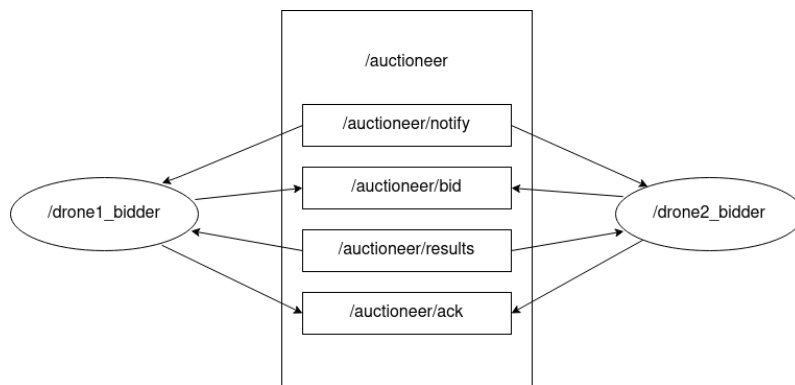


Figure 4.13: Node/topic interaction of the auction system. Based on the `rqt_graph` and edited for better visual understanding. Non-relevant nodes have been ommited.

4.3.2.2 Firmware modifications

In the MAV side, a few modifications to the firmware are needed to accommodate this new functionality. Firstly, an additional queue containing all alarms is added and is considered to have higher priority than regular sensing waypoints. Then, the FSM illustrated in figure 4.14 is added and runs concurrently to the main FSM.

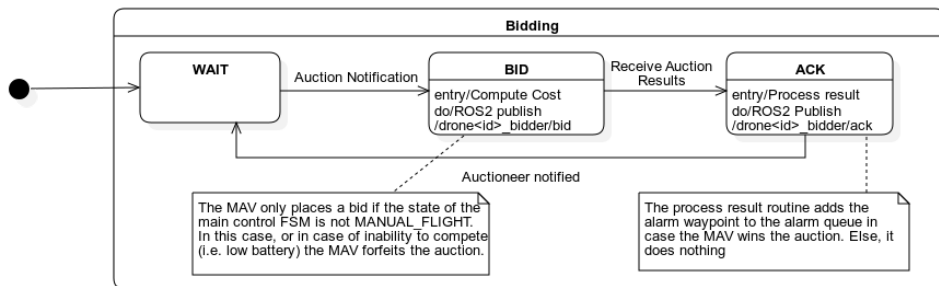


Figure 4.14: Bidding FSM

This FSM's state transitions are triggered by the interactions with the auctioneer. The initial state is "wait" in which the MAV awaits for an auction notification, after receiving the notification

Another FSM (figure 4.15) is added to monitor whether the MAV still has got alarms to visit or not. This is relevant as this *flag* is used to trigger transitions in the main FSM.

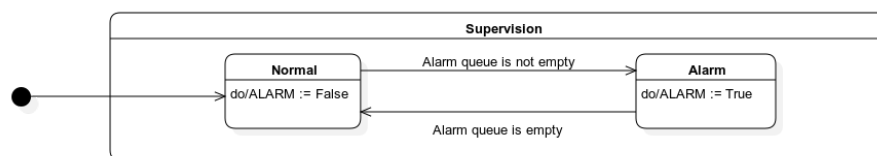


Figure 4.15: Supervision FSM

Manual GoTo Commands

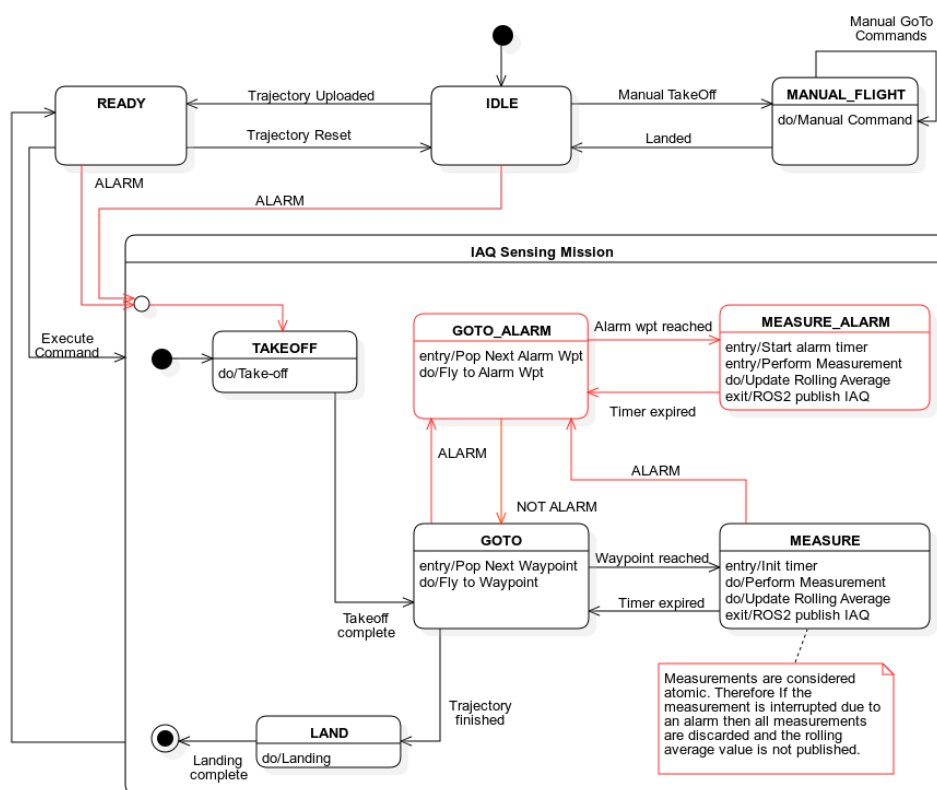


Figure 4.16: Main FSM with alarm functionality. Red lines represent changes from the non-alarm version.

Alarms are considered to be priority in regards to all other behaviors except manual control. In case an alarm is triggered during a measurement, the rolling average is reset and the MAV has to redo the measurement after attending all alarms.

4.4 On-board IAQ sensing

4.4.1 Hardware

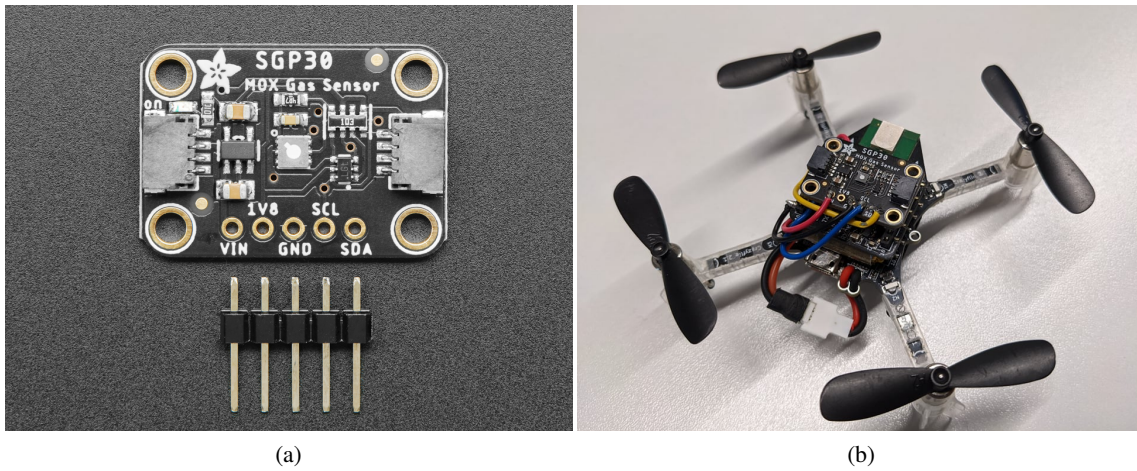


Figure 4.17: Sensing Hardware. a) Adafruit breakout board [78]; b) Sensor mounted on top of the Crazyflie

The sensor chosen for this project is the Sensirion SGP30 MOX Gas Sensor (Adafruit breakout board). The SGP30 is a COTS low-cost and low power digital I^2C sensor that outputs both Total Volatile Organic Compounds (TVOC) and equivalent⁷ Carbon Dioxide (CO_{2eq}) measurements.

Additionally, the sensor has an on-chip baseline calibration and a humidity compensation feature (requiring external absolute humidity sensor readings).

The choice of sensor had three major deciding factors between other compatible I^2C sensors:

- Low-cost and availability.
- Reduced size and weight (to mount on the MAV)
- Previous integration of the sensor on the Crazyflie [58]

Physical integration of the sensor on the MAV was done simply by soldering the sensor to the corresponding (VCC, SDA, SCL, GND) LPS deck headers (this allows removing the sensor simply by replacing the LPS deck). As the crazyflie already has 2.2k Ω pull-up resistors on the SDA and SCL lines of I^2C bus, no additional components are needed. The MAV in conjunction with the

⁷The equivalent CO_2 is a calculated measurement, meaning it is not a real CO_2 measurement but a calculation based on raw H_2 and Ethanol signals.

decks (flow deck, LPS and SGP30) weighs 40g which is still under the maximum recommended takeoff weight of 42g.

4.4.2 Firmware Integration

To integrate the sensor in the firmware, a custom deck driver that was kindly provided by Neuman et al. [58] was used. This driver creates additional tasks for both sending I^2C commands to the sensor, but also for performing humidity compensation and baseline calibration. Baseline values are stored in the EEPROM of the Crazyflie after running for a certain amount of time (in this case both sensors performed a baseline calibration of 12 hours in a clean environment).

The sensing subsystem also takes advantage of the publisher/subscriber paradigm of ROS2, where the MAVs publish their measurements to a common `/airsense/iaq` topic and subscribe to `/rpi/weather_data` temperature and humidity measurements for updating the compensation.

Another modification was to update the mass of the crazyflie in the EKF module's source code to improve the accuracy of the filter's previsions.

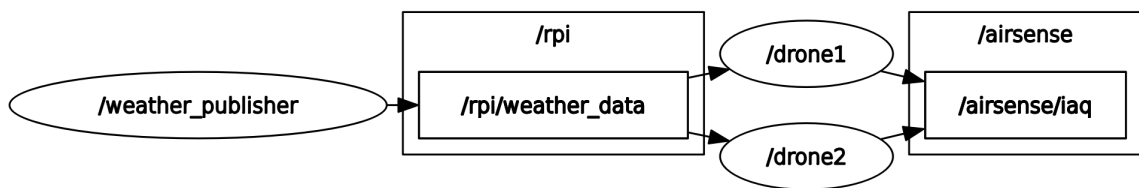


Figure 4.18: rqt_graph of sensor publishing and humidity correction.

4.5 Cloud Services

4.5.1 Integration Service and cloud-based planning

In order to reduce the workload of local nodes/machines and allow more complex algorithms to be developed, a cloud-based ROS2 system was set up using AWS Elastic Cloud Computing (EC2). AWS EC2 is an Infrastructure as a Service (IaaS) application that allows users to deploy applications on a virtual machine running on the AWS infrastructure. In this case, a general purpose *t2.xlarge* instance running Ubuntu Server 20.04 was used.

The decision for the instance's region was done based on an online tool⁸ that performs RTT measurements to multiple regions worldwide by "calculating the time needed to load a resource from a health page of a AWS service". Based on this, the EC2 instance was deployed in London (eu-west-2).

Running the planning node in the EC2 instance is just a matter of installing ROS2 and then relocating the node to the cloud VM. However, the complexity arises when connecting the local

⁸<https://ping.psa.fun/>

DDS data space with the cloud one as DDS multicast-based discovery will not work without additional configurations. In this regard, approaches such as setting up a Virtual Private Network (VPN) and/or replacing the default discovery mechanism with a unicast-based discovery server⁹ were considered. However, in order to separate the cloud and local data spaces, an alternative method called *Integration Service*¹⁰, developed by eProsima, was used.

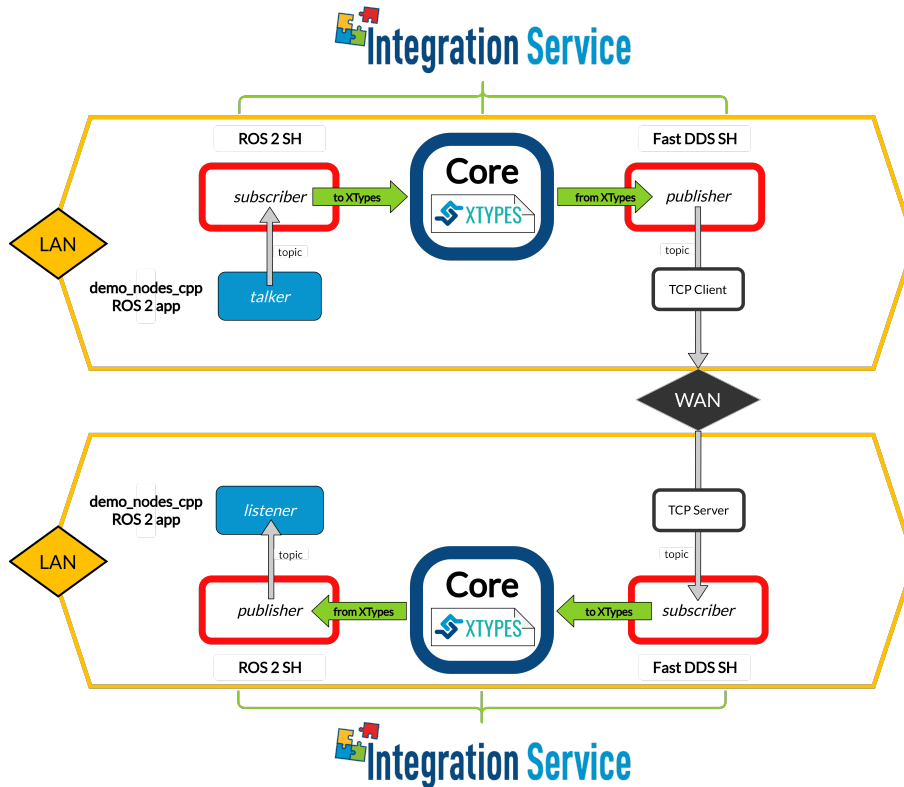


Figure 4.19: WAN communication example using integration service [79]

The *Integration Service* in WAN communication works by "translating the types and topics of ROS2 to those of Fast DDS, and then use the WAN-TCP communication capabilities of the latter to operate the tunneling"¹¹. This functionality is represented in figure 4.19 where multiple middlewares or communication protocols have a specific system handle (SH). In this case, the ROS2 SH and Fast DDS SH can communicate through the use of Extensible and Dynamic Topic Types for DDS (xTypes). Nonetheless, additional configuration files are necessary to define transport settings such as the remote machine's IP address and mapping between ROS2 and Fast DDS topics/services - these configuration files can be consulted in appendix A.3.1. Furthermore, it was necessary to configure port forwarding of the EC2 instance in the AWS console.

⁹https://fast-dds.docs.eprosima.com/en/latest/fastdds/use_cases/wifi/disable_multicast.html

¹⁰<https://github.com/eProsima/Integration-Service>

¹¹https://integration-service.docs.eprosima.com/en/v3.0.0/examples/wan_communication/wan_tcp_dds.html

4.5.2 AWS Cloudwatch monitoring

Aside from ROS2-enabled cloud nodes, integration with an online tool for application and infrastructure monitoring with dashboard visualization, AWS Cloudwatch, was also established. Integration with the service is achieved through the *AWS RoboMaker cloud extensions*, a set of ROS1/ROS2 packages to ease access to AWS by robotic systems.

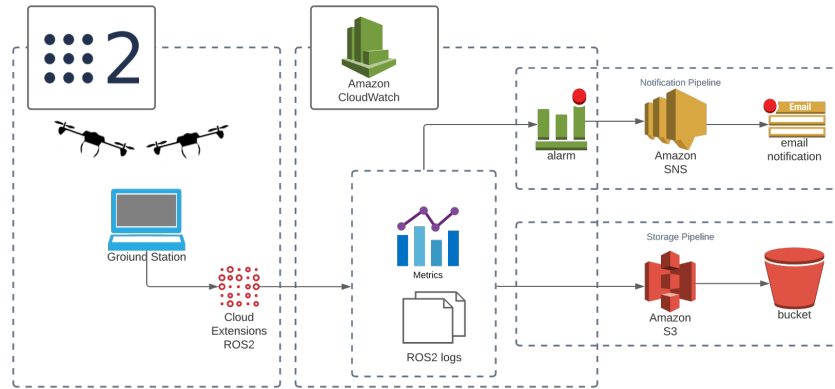


Figure 4.20: Cloudwatch diagram.

As represented in figure 4.20, the "gateway" between the ROS2/DDS world and the AWS cloud are the *cloud extensions* which subscribe to ROS2 topics or monitor information (i.e. CPU load) and publish it to AWS Cloudwatch in JSON format using standard HTTP requests. The *cloudwatch extensions* and their purpose can be listed as:

- **cloudwatchmetrics-ros2:** Subscribes to metrics (as ROS2 messages) such as the environment measurements (humidity, temperature and IAQ).
- **cloudwatchlogs-ros2:** Subscribes to ROS2 logs (in this case only rosout but can be configured to subscribe to other logging sources).
- **health-metrics-collector-ros2:** Collects system metrics (CPU load, available RAM, etc.).

The interaction between ROS2 nodes can be seen in the figure 4.21.

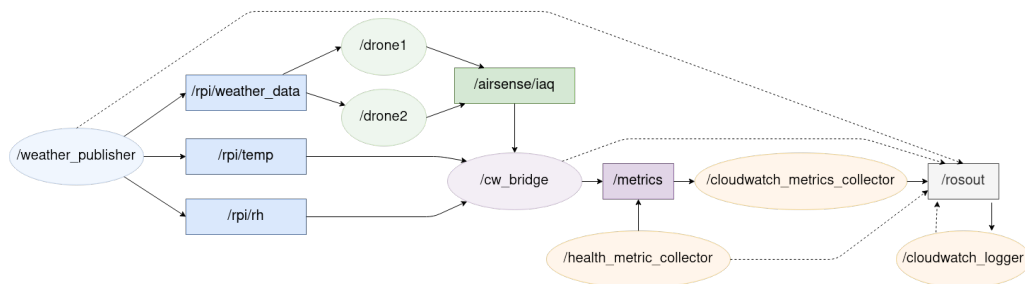


Figure 4.21: rqt_graph-based representation of the AWS Cloudwatch related nodes. Color code: Blue - Raspberry Pi (weather sensor), Green - MAVs, Purple - Ground station additional nodes/-topics, Orange - AWS cloud extensions. Dashed lines represent the logging of all nodes.

An additional node, *cw_bridge*, was added in order to subscribe the topics related to the IAQ sensing and temperature/humidity measurements, convert the ROS2 messages to the appropriate type (*ros_monitoring_msgs/msg/MetricList*) and then publish it to the metrics topic. The metrics topic is then subscribed by the *cloudwatch_metrics_collector* node that relays the data to the cloud.

The dashboard for this application can be seen in figure 4.22.

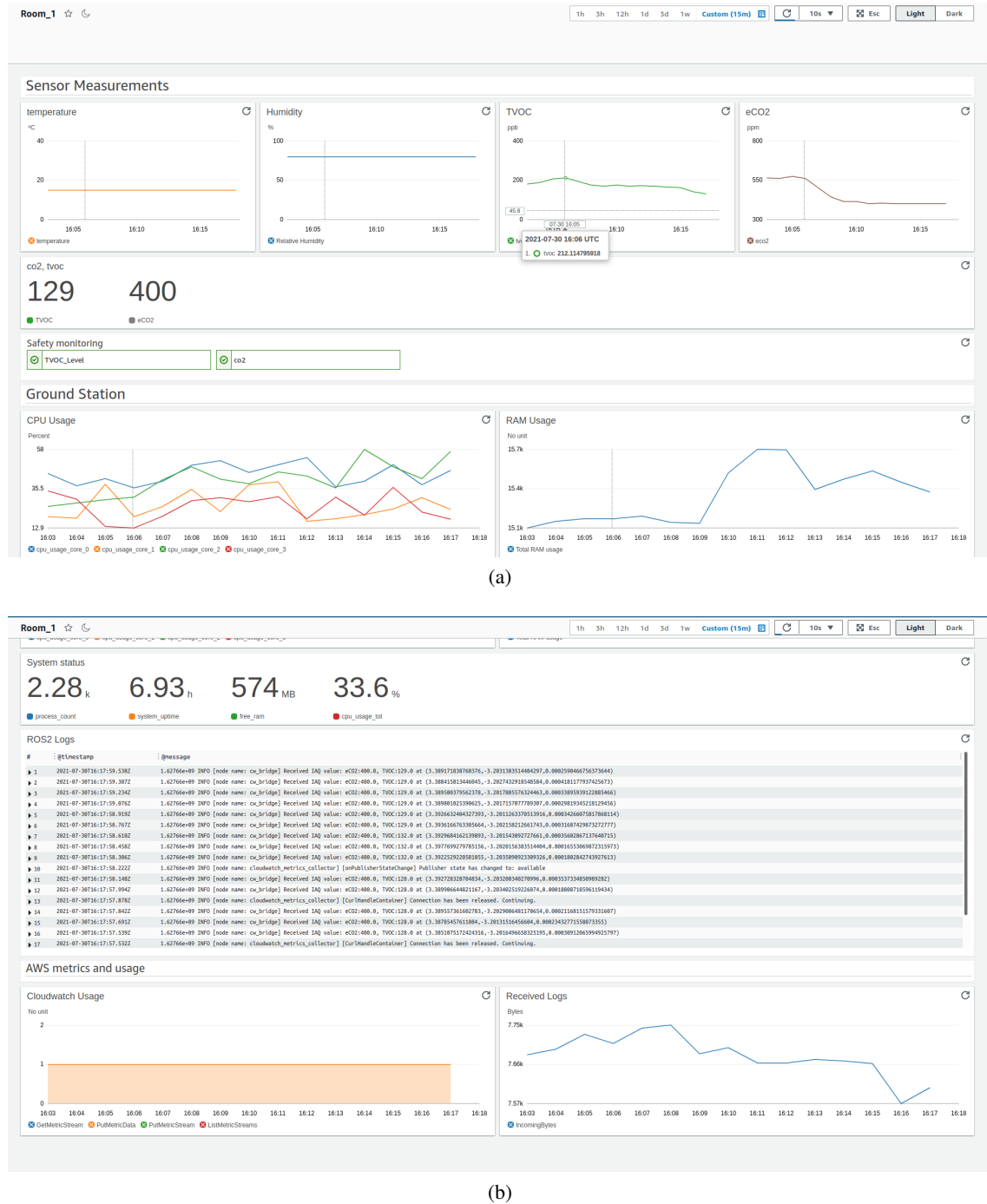


Figure 4.22: AWS Dashboard. a) Sensor measurements and ground station monitoring; b) ROS2 logs and cloud service metrics and usage.

4.6 Visualization

Aside from the GUI and AWS cloudwatch dashboard, real-time visualization of the MAVs movement and sensing waypoints/alarms can also be useful for monitoring the mission execution. For this we have used an open-source RVIZ visualization module previously developed for the crazyflie/turtlebot micro-ros demo¹². Additionally, we have modified this package to support more than one Crazyflie, added a map representation of the arena using the map_server package which is part of NAV2¹³ (in this case only serving for aesthetic purposes but can be useful in other robotic applications such as obstacle detection and mapping) and markers as a means of representing waypoints (during mission execution).

As mentioned previously in subsection 3.2.3, the bottom right anchor is used as the reference frame for the MAVs (*odom*). In order to visualize the MAVs in RVIZ, each MAV has an individual coordinate frame¹⁴ obtained from the odometry and attitude topics (using a translation node from the aforementioned demo). This frame is obtained w.r.t. the *odom* frame which in turn is obtained w.r.t. the *map* frame. The RVIZ module (with two idle MAVs) can be seen in figure 4.23.

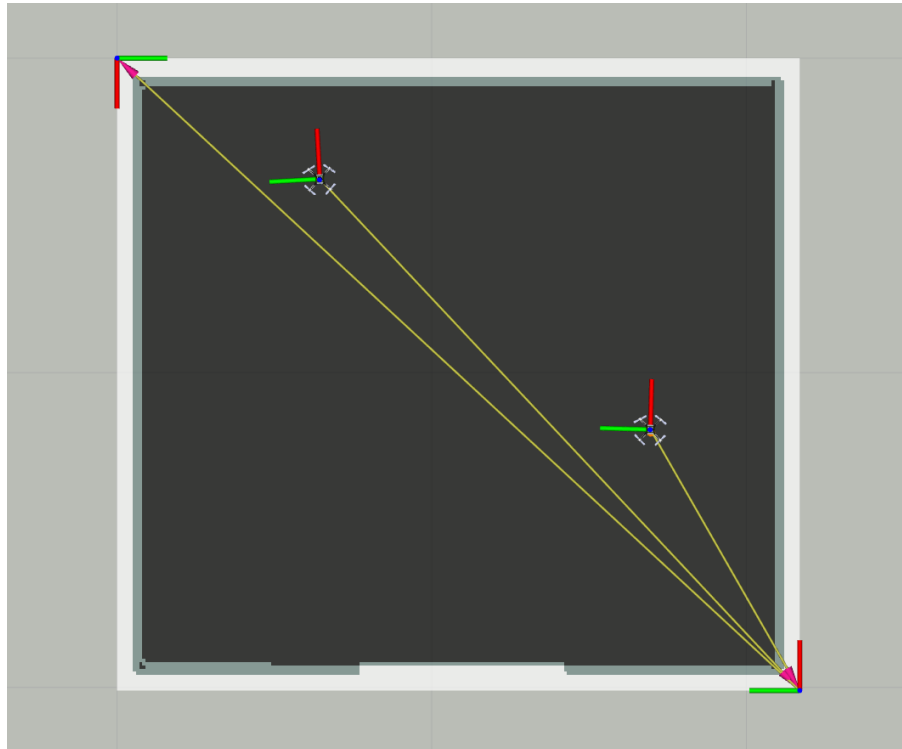


Figure 4.23: RVIZ visualization. Top left frame is the *map frame* and bottom right is the *odom* frame.

¹²<https://eprosima.com/index.php/company-all/news/126-micro-ros-turtlebot-meets-crazyflie>

¹³<https://navigation.ros.org/>

¹⁴Coordinate frames and relationships between them is managed by the ROS2 package tf2: <http://wiki.ros.org/tf2>

4.7 Summary

In this chapter, a description of the implemented software/hardware components was provided. From this description we can summarize the design choices and limitations as:

- Communication with the MAVs requires a computer or embedded device with a Crazyradio, so all information must go through Ground Station to reach the MAVs and vice versa. However, the use of XRCE-DDS/DDS allows the end user to be agnostic to this process. In addition, a custom Micro XRCE-DDS agent has been developed that uses the CRTP transport protocol.
- A simple FSM for following sensing waypoints was implemented within the firmware's app layer allowing for two separate queues with low (normal behavior) and high priority (alarms). The firmware also provides micro-ROS/ROS2 services to manually control the MAVs, allowing them to be used in other applications.
- For mission planning, a static algorithm based on the MmTSP was developed and solved with an open-source branch-and-cut MILP solver that guarantees an optimal solution for the proposed cost function (minimising the most time-consuming trajectory of the MAVs). However, since the MmTSP is proven to be NP-Hard this approach is expected to become exponentially slower as the number of MAVs and waypoints increases.
- Since multiple MAVs share the same flight space, a collision avoidance algorithm is required. For this purpose, we integrated a collision avoidance firmware module previously developed at Capgemini Engineering. This algorithm is based on artificial potential fields and uses P2P communication to exchange positions between MAVs. This module does not use micro-ROS, as the framework does not support brokerless P2P at the time of writing.
- The addition of high priority waypoints (alarms) is done by a simple auction-based algorithm, where the MAVs calculate the cost of performing the measurement and assign the task to the one that can perform the measurement in less time. This design choice is due to the MAVs being the only participant in this architecture who has access to its mission plan, battery life and measurement status.
- As an example use case, we have mounted an SGP30 sensor on the MAVs to measure IAQ and integrated it into the micro-ROS-enabled firmware. A baseline calibration was performed. However, in order for the sensor to provide accurate readings, an additional calibration against known sources and a comprehensive investigation of the effects of rotor downwash on the readings is required.
- Cloud services were integrated both for cloud-based planning in a AWS EC2 instance using eProxima's Integration Service for WAN communication and with AWS Cloudwatch for metric storage, monitoring and dashboard visualization through the AWS Robomaker Cloud Extensions.

- To provide visual feedback of the mission planning and execution, both a *ros_rqt* GUI and an RVIZ visualization module were developed.

Chapter 5

Results and Discussion

In this chapter, an experimental validation/evaluation of the proposed system is presented as well as the methodology to obtain these results. Firstly, an evaluation of the planner component is carried out in Section 5.1. Afterwards, a validation of the execution of a example mission is performed on Section 5.2, and an evaluation of the latency of the communication architecture is presented in Section 5.3. Lastly, Section 5.4 offers an overview and critical discussion of the results.

5.1 Planning

In this section, a visual validation of the routes produced by the planner module described in Section 4.3 is performed as well as a quantitative evaluation of the computing time needed for the algorithm to find an optimal solution for a variable number of MAVs and waypoints.

5.1.1 Produced Mission Trajectories

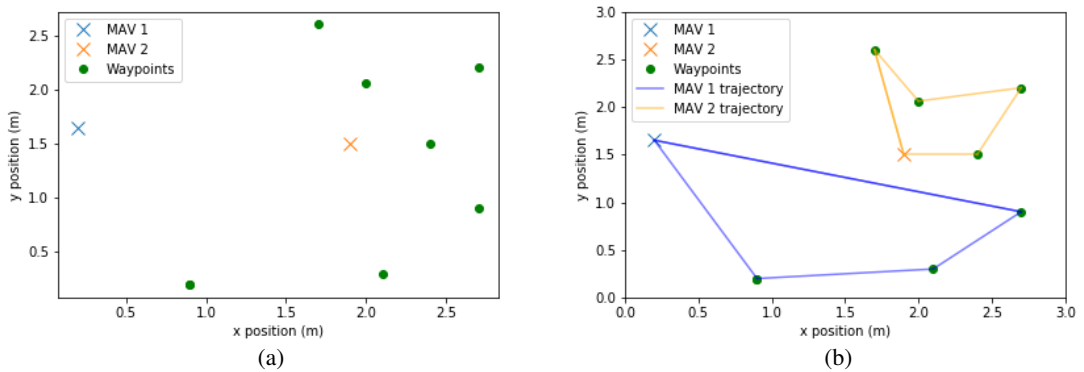


Figure 5.1: Example planning. a) Pre-plan; b) Post-plan

In Figure 5.1, a visual representation of the planner node's input and output (projected on the XY plane) is provided. The MAVs' initial positions are represented with a cross with the sensing

waypoints as green circles. For this experiment, both the MAVs initial positions and waypoints are randomly generated using a uniform distribution within the limits of the arena. MAVs perform measurements for 10s in each waypoint and move between them with a linear velocity of 0.25 m/s with no limitation on the maximum and minimum number of waypoints visited per MAV.

An important consideration of the proposed algorithm is that the time spent in each waypoint is considered in the optimization problem's cost function as well as the time spent in-route to each location. Figure 5.2 shows the output of the planner node, for the same input of Figure 5.1, when the measurement time is omitted.

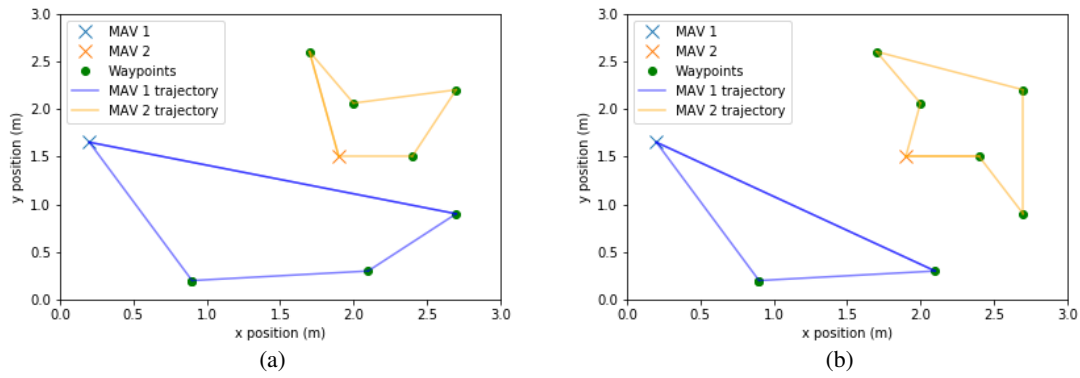


Figure 5.2: Example planning with different criteria. a) Planning considering measurement time; b) Planning not considering measurement time

When only the distance between waypoints is considered, the waypoint at (2.7, 0.9, 0.5) previously visited by MAV1 becomes part of the route of MAV2. A comparison of the duration and length of the resulting trajectories can be found on Table 5.1. With the addition of this waypoint to the MAV2's route the largest route is of 5.15m which is 17.8% shorter than before and the difference between the distance traversed by the MAVs is reduced. However, in this situation MAV2 has to visit 5 waypoints and MAV1 just 2, causing the maximum mission time (and therefore the time needed to perform all measurements) to increase by 13.10s (14.7%). In situations where the waypoints are close to each other but the time spent performing measurements is considerable, which is somewhat common in indoor applications, this can be highly suboptimal.

Table 5.1: Comparison of the route length and total mission time considering or not the measurement time

	w/ measurement time		w/o measurement itme	
	MAV1	MAV2	MAV1	MAV2
Route length (m)	6.27	3.71	5.15	4.73
Total time (s)	75.09	75.84	60.58	88.94

5.1.2 Planner performance

5.1.2.1 Test Setup

In order to test the computing time needed for the planner to find an optimal solution, a bash script was prepared to deploy a test ROS2 node that performs a service request to the planner node. The script passes the test conditions to the node using ROS2 parameters to modify the number of MAVs and waypoints of the request. Again, the MAVs initial positions and waypoints are randomly generated using a uniform distribution within the arena's physical boundaries.

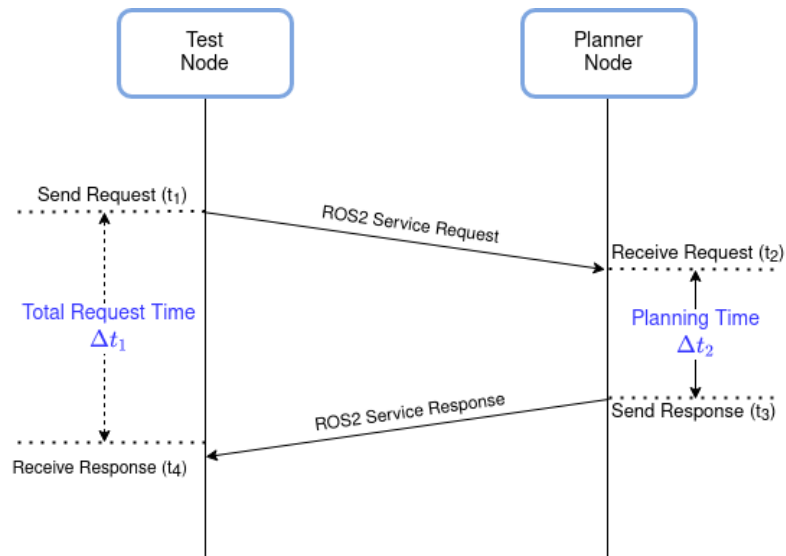


Figure 5.3: Planner RTT Test

The evaluation setup is represented in Figure 5.3 and consists of registering the timestamps of when the request is performed and when the response is retrieved: the difference between these timestamps equates to the total time needed for the planning Δt_1 . This time is not only the time needed to solve the optimization problem Δt_2 , which we measure as the time between receiving the request and sending the response, but also the latency of the communication $\Delta t_1 - \Delta t_2$. The latter is especially relevant when using cloud-based planning.

This evaluation was performed both on a Desktop-grade PC, a Raspberry Pi (as the possibility of using an embedded device for communicating with the MAVs was a requirement for this project) and the cloud platform. The specifications of all devices can be found on Table 5.2

Table 5.2: Device specifications

	Desktop	Embedded device (Raspberry Pi)	Cloud (AWS EC2)
Model	Thinkpad T14 Gen1	Raspberry Pi 4	t2.xlarge
CPU	AMD Ryzen Pro 5 4650U	Quad core Cortex-A72 (ARM v8)	Intel Xeon E5-2686
RAM	16GB	1GB	16 GB
OS	Ubuntu 20.04	Ubuntu Server 20.04	Ubuntu Server 20.04
Kernel	5.11.0-25-generic	5.4.0-1041-raspi	5..8.0-1041-aws

5.1.2.2 Local Execution

As a first evaluation, both the test script and the planner node were executed on desktop-grade hardware. The script performed 100 measurements for each (number of MAVs, number of waypoints) tuple, then the median of the measurements for each tuple was taken in order to understand how the processing time (Δt_2) changes with the number of MAVs and number of waypoints. The results can be seen on Figure 5.4 and Table 5.3.

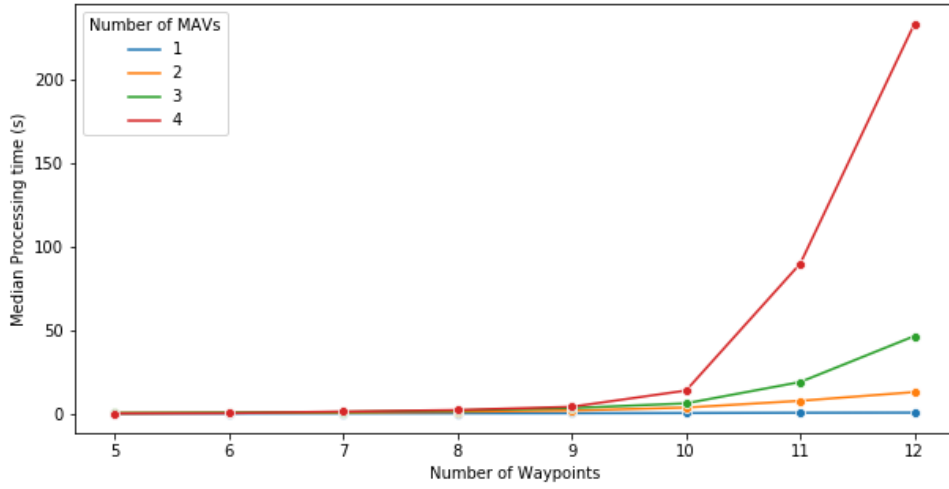


Figure 5.4: Graphical representation of the growth of the median planning time w.r.t. MAVs and waypoints (data from Table 5.3)

The results show that, as expected from an optimal optimization approach, the algorithm escalates poorly with the number of MAVs and waypoints. In the range of the proposed use-case (2/3 MAVs for up to 12 waypoints) the planning module can still find a solution in sub-minute time. However, in use-cases with more MAVs/waypoints another approach should be taken unless optimality is a must and/or the mission can be planned before-hand.

Table 5.3: Median planning time (in seconds) w.r.t. the number of waypoints and MAVs - both nodes are running on a Desktop PC. Values in bold represent decreases in latency w.r.t. number of MAVs

		Number of Waypoints							
		5	6	7	8	9	10	11	12
Number of MAVs	1	0.0495	0.0697	0.1018	0.1533	0.2395	0.3500	0.4946	0.5690
	2	0.2911	0.4011	0.6083	1.0622	1.8254	3.602	7.655	12.890
	3	0.250	0.560	0.960	1.826	3.318	6.129	18.909	46.443
	4	0.105	0.256	1.200	2.130	4.089	13.798	89.809	233.455

Another observation, highlighted in the results of Table 5.3 is that the planning time actually decreases when the number of MAVs is superior to half of the number of the waypoints. This is most likely due to the constraint that imposes that every MAV must visit at least one waypoint which can result in a decrease of the branch-and-cut tree.

5.1.2.3 Cloud-based planning

In this experiment, the same procedure from the previous section was adopted but with different hardware targets. For simplicity instead of computing results for a wide range of waypoints only two cases were considered: a relatively small number (5 waypoints) and a larger sample of 10 waypoints. Similarly, the tests were made for up to 4 MAVs.

The test consisted of two trials for testing the performance gain of using cloud computing. The hardware used to run the test node (ground station) was the embedded device in both trials. Planning was performed locally for the first trial while in the second it was done on the cloud using integration-service to establish the WAN connection.

Table 5.4: Median planning and communication time (in seconds) w.r.t. the number of waypoints and MAVs - comparison of using local and cloud computing.

		Number of waypoints							
		5				10			
		RPI		RPI + Cloud		RPI		RPI + Cloud	
		Δt_1	$\Delta t_1 - \Delta t_2$	Δt_1	$\Delta t_1 - \Delta t_2$	Δt_1	$\Delta t_1 - \Delta t_2$	Δt_1	$\Delta t_1 - \Delta t_2$
Number of MAVs	1	0.3430	0.0147	0.1334	0.0778	2.293	0.0154	0.6526	0.0637
	2	2.672	0.0150	0.4625	0.0512	35.052	0.0161	5.4221	0.0685
	3	1.216	0.0152	0.2969	0.0647	90.014	0.0167	11.4337	0.0724
	4	0.671	0.0159	0.17629	0.0671	169.743	0.0174	21.5715	0.0753

The results (Table 5.4) show that the usage of cloud computing can greatly improve the performance of the planning module, in the worst-case of this experiment (4 MAVs for 10 waypoints) the trial that used cloud computing achieved an 8.5x faster planning. Evidently, this computation time can be further reduced by increasing the computational resources of the EC2 instance.

Additionally, as expected due to the nature of cloud computing, communication latency between nodes is larger and less constant in the cloud trial. However its impact on planning performance can be considered negligible. Furthermore, all cloud-enabled measurements show a median communication latency below 100ms indicating that this approach may also be suitable for performing non-critical planning tasks "on-the-fly".

5.2 Flight Tests

Regarding flight tests with the actual MAVs, four tests were performed to validate the implementation on a real-world scenario:

1. A simple static mission with trajectories obtained from the planning module.
2. A test where an alarm is triggered in an area patrolled by two MAV.
3. A mapping experiment using a sweeping motion.
4. A test regarding the impact of the sensor on the MAVs battery life and consequently time of flight.

5.2.1 Test 1: Static Mission Execution

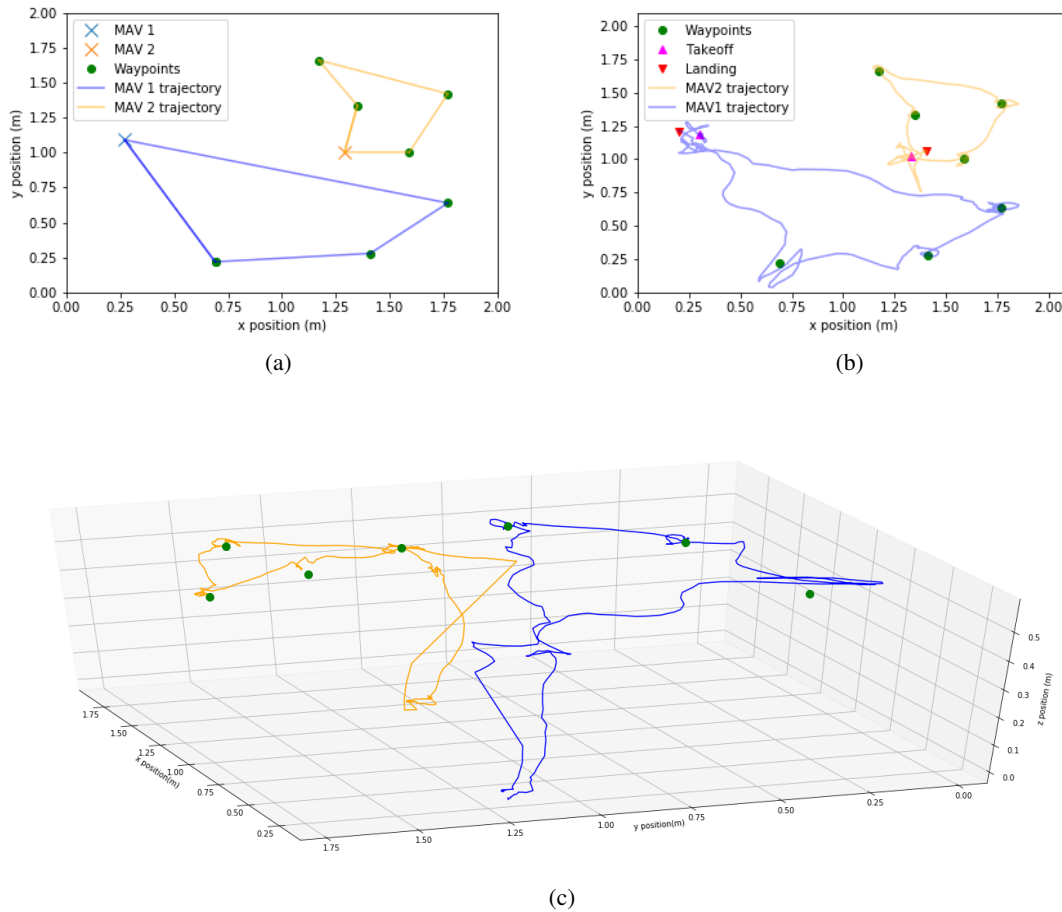


Figure 5.5: Execution of test 1. a) Mission plan, b) top-down view, c) 3D representation of MAV trajectories.

In this test, the MAVs have to complete the mission planned in the previous section (scaled to the dimensions of the test arena) and represented in Figure 5.5a. The MAVs tri-dimensional flight

can be seen in Figure 5.5c and a top-down view at $z=0.5\text{m}$ (the altitude of all sensing waypoints) in Figure 5.5b.

This test was successful as the MAVs performed the trajectories created by the planning module. A systematic occurrence during flight tests is the loss of stability near the limits of the arena which can be seen in this mission when the MAV is in the vicinity of the waypoint at $(0.69, 0.22, 0.50)$. Take-off and landing locations are slightly different this is due to slippage in the landing (that can be seen on Figure 5.5c). The planning algorithm considers the desired landing location as the "depot" of the MmTSP algorithm therefore the MAV always attempts to reach this position.

5.2.2 Test 2: Mission with alarms

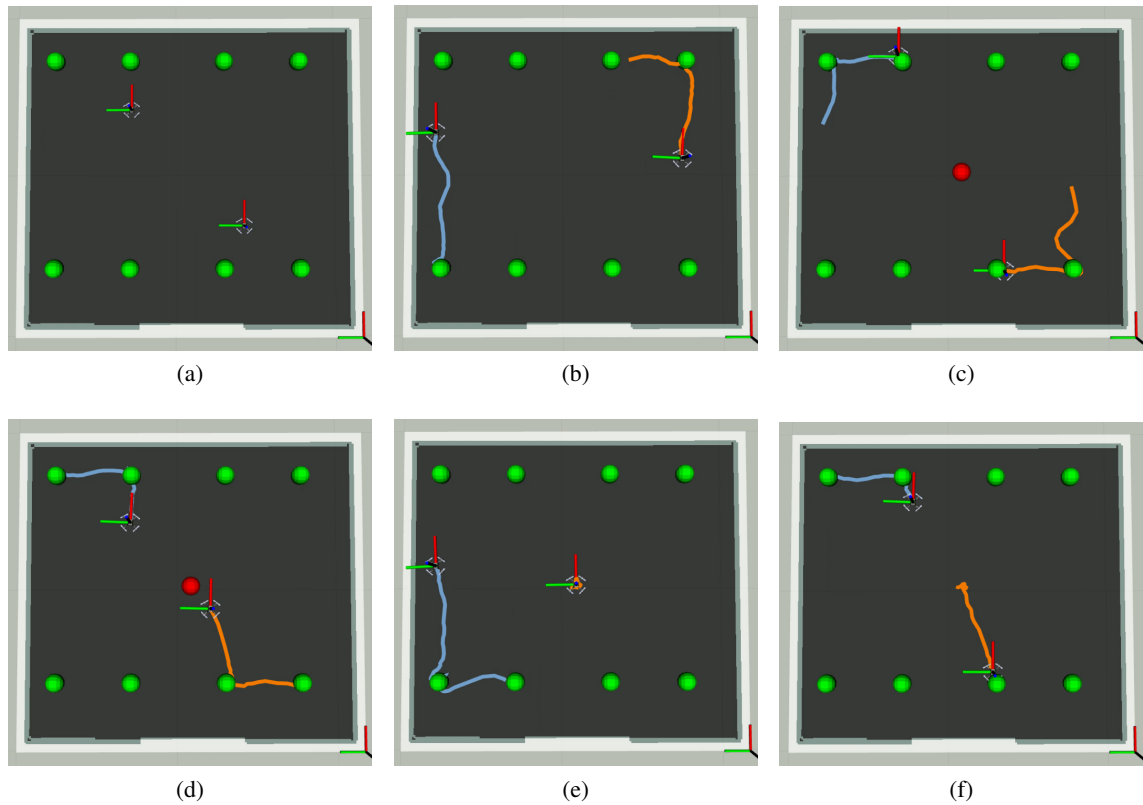


Figure 5.6: Example mission with RVIZ visualization. (a) Planning and mission upload, green markers represent sensing waypoints; (b) Mission execution, orange and blue lines represent the trajectory of MAV1 and MAV2 respectively; (c) An alarm is triggered in the middle of the arena (represented by the red marker); (d) Auction is won by MAV1 (orange) who goes to measure the alarm point; (e) MAV1 performs the measurement while MAV2 continues its mission; (f) MAV1 returns to the its normal route.

As an example mission with the alarm functionality, we have divided the arena in two halves. Each half of the arena has four waypoints in a rectangle disposition and a MAV to perform a "patrol" in a loop. During the mission, an alarm was requested in the middle of the arena (between

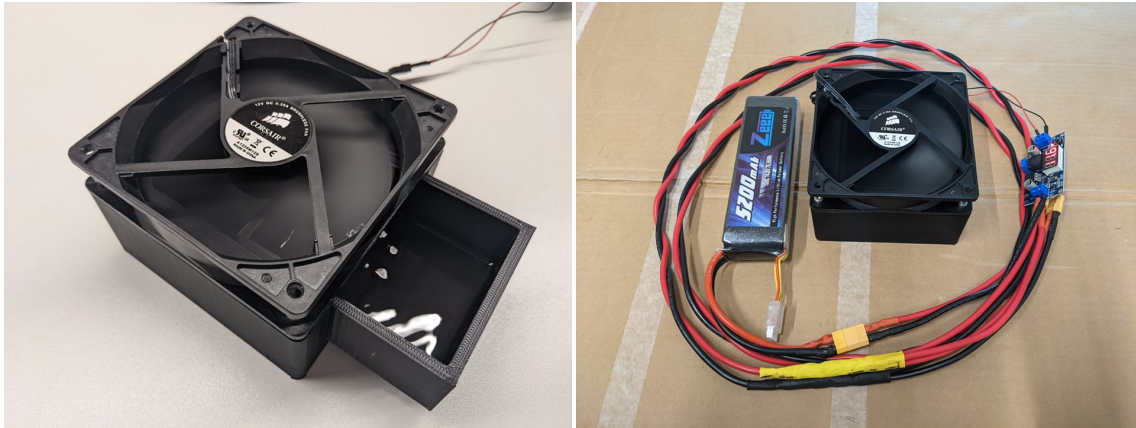
the two halves). Two separate requests were made to the planner module for each one of the arena's halves. The mission execution can be seen on Figure 5.6.

The test was completed with success as the MAVs were, first and foremost, able to perform the static patrol and then react to the alarm published by a ground node by deciding which MAV to go to its location. The MAV closer to the alarm was chosen (as it can complete the task quicker), performed the measurement correctly and then returned to its patrol.

5.2.3 Test 3: IAQ Sensing

In this experiment, we aim to validate that the sensor is working correctly and publishing measurements. For this purpose, we have adopted a strategy similar to Neumann et. al [58] and Duisterhoff et al. [59] where a contamination source is placed in the arena and the MAV performs a boustrophedon¹ sweep of the area performing measurements in a grid-like manner.

The contamination source was simulated by placing liquid ethanol 96% (v/v) in a 3D printed container visible in Figure 5.7, coupled with a common 120mm 12V Desktop PC fan powered by a DC battery using a buck converter. The fan is placed face-up in the upper-right corner of the arena and disperses the ethanol in the air which can then be sensed by the SGP30 on-board the MAV.



(a) 3D printed container with deposit open.

(b) Artificial contamination source in the arena.

Figure 5.7: Artificial contamination source setup.

The results of the IAQ mapping in Figure 5.8 show that the MAV was able to follow the pre-defined trajectory (which was uploaded using the `/drone<id>/upload_trajectory` service) and publish sensor measurements using the rolling average method on the desired waypoints. This experiment is not meant to test the accuracy of the measurements, only to validate that the MAV reacts to a contaminant in the area.

The interpolation of the sensor measurements in the plot of Figure 5.8b was obtained with help of the `scipy.interpolate` python3 module applied to a rosbag² containing recordings of both

¹Boustrophedon means the "way of the ox" and consists of simple back and forth motions across a given axis.

²A rosbag is a format used to record and playback messages from ROS2 topics.

the `/drone<id>/odometry` and the `/airsense/iaq` topics. The ability to detect airborne contaminants can be confirmed in the mapping plot of Figure 5.8b as the values of TVOC increase abruptly near the contamination source.

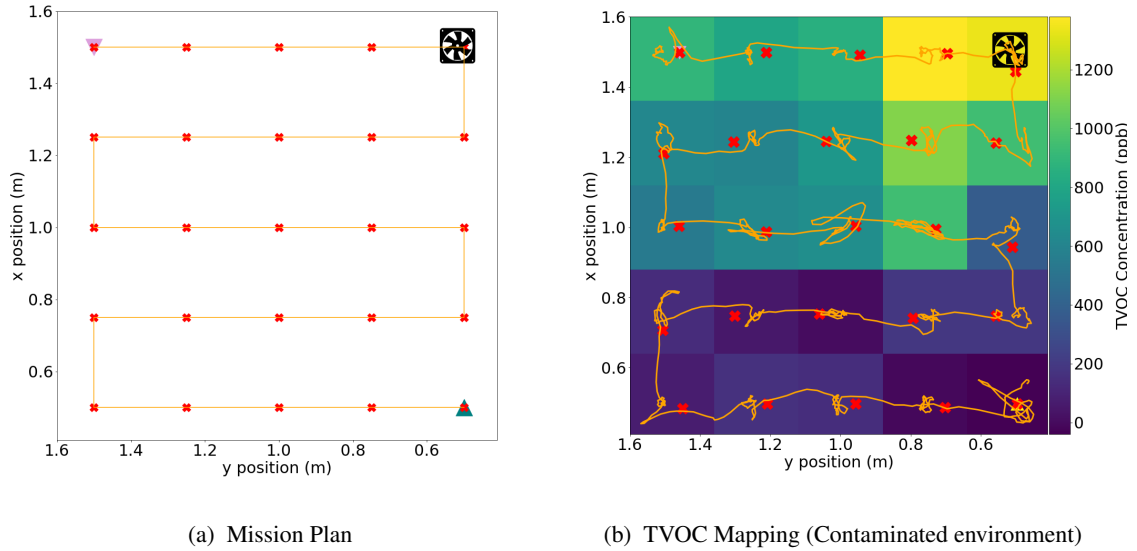


Figure 5.8: IAQ Mapping Experiments. Orange line represents the trajectory of the MAV, red crosses the measurement points and the fan icon the contaminant source. Upward and downward arrows represent the take-off and landing locations respectively. Note that the x-axis is inverted to match the reference frame of the LPS.

5.2.4 Test 4: Battery consumption

In this test, the crazyflie was made to hover in the middle of the arena at an altitude of 0.5m in two different situations: with the SGP30 sensor and without. For the two trials, the quadcopter used was the same. As the gas sensor is soldered onto the LPS deck, the former can be easily removed by replacing the latter.

The procedure is simple: an additional publisher was created in the micro-ROS application to publish the analog battery voltage along with a timestamp. Then, the MAV is commanded to take-off and hover until it no longer can fly (resulting in a crash). The results can be seen on the graph of Figure 5.9

The results show that the SGP30 reduces the flight time of the MAV by 50.60s which is 12.63% less than without the sensor. The voltage drop in the beginning of the trials is due to starting the motors for take-off, and the initial Y-gap is due to the voltage drop caused by the IAQ sensor (which consumes 48mA@1.8V). During flight, the voltage decreases in a relatively linear fashion before dropping abruptly causing the MAV to fail to stabilize itself and plummet to the ground. Additionally, we obtain a conservative estimate of the MAV's safe time of flight (before the voltage drop) of around 240s which can then be used as a constraint of the planning module.

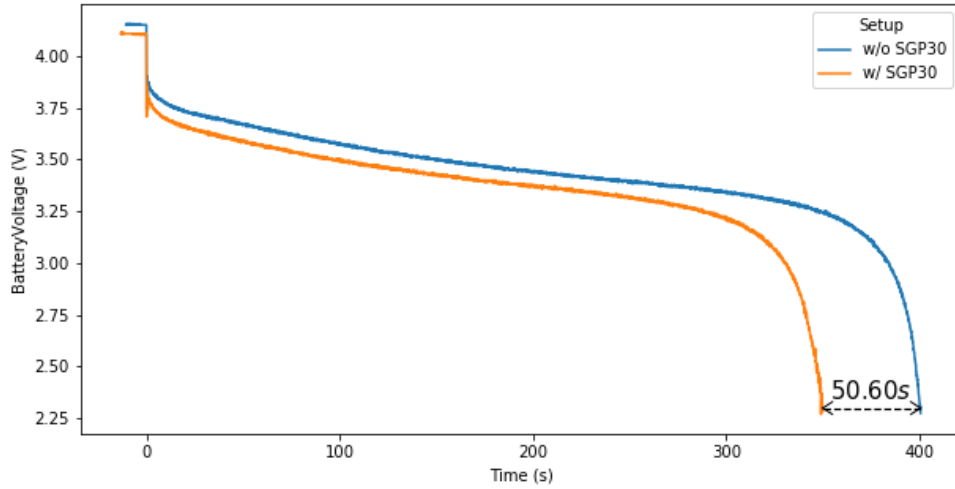


Figure 5.9: Impact of IAQ sensor on time of flight

5.3 Communications

In order to evaluate communications, a similar methodology to section 5.1.2 was used. A primary node publishes a custom ROS2 message type made up of a sequence of bytes with variable size and a secondary node subscribes this topic and publishes a return message which is then subscribed by the primary node. The round-trip time (RTT) latency is obtained as the time difference between publishing the message and receiving the response. Unless stated otherwise, the number of samples is equal to 1024 for each test condition.

5.3.1 GS-MAV communication

For evaluating the latency between GS and the MAVs, two tests were performed. The first test does not use micro-ROS or ROS2 at all but only the native transport C++ library that the custom XRCE-DDS agent uses and serves as a baseline for accessing the overhead of using micro-ROS/ROS2. The second uses micro-ROS/ROS2 with different QoS Profiles (cf. Section 2.2.1.3).

Evaluation Setup

Evaluation of the native library was carried out using the *benchmark example*³. The example consists of sending packets to the echo port of the MAV and then measuring the time interval between sending the packet and receiving its echo. The payload is a simple `uint64_t` (8B) with a sequence number for evaluation purposes.

Concerning the micro-ROS/ROS2 evaluation, five different QoS Profiles were defined and can be seen on Table 5.5. This selection consists in two of the default ROS2 profiles as well as three custom-defined ones to test other QoS policies. Similarly to Fernandez et al. [44], the selected

³<https://github.com/bitcraze/crazyflie-link-cpp/blob/master/examples/benchmark.cpp>

profiles aim to compare three policies: RELIABILITY, DURABILITY and HISTORY. These tests follow the methodology for ROS2-enabled evaluations described previously.

Table 5.5: QoS Profiles used in the evaluation

QoS Profile	Reliability	Durability	History	Depth
Sensor Data	Best Effort	Volatile	Keep Last	5
Default	Reliable	Volatile	Keep Last	10
Custom A	Reliable	Transient Local	Keep All	N/A
Custom B	Reliable	Volatile	Keep All	N/A
Custom C	Best Effort	Volatile	Keep All	N/A

Comparison over the specified QoS Profiles

First a basic comparison of the native and the various QoS profiles was performed. In this interest, the payload size used in both cases was of 8B to be able to transmit the test data in a single CRTP packet. In this test, only a single static MAV was used. The results can be seen on Figure 5.10

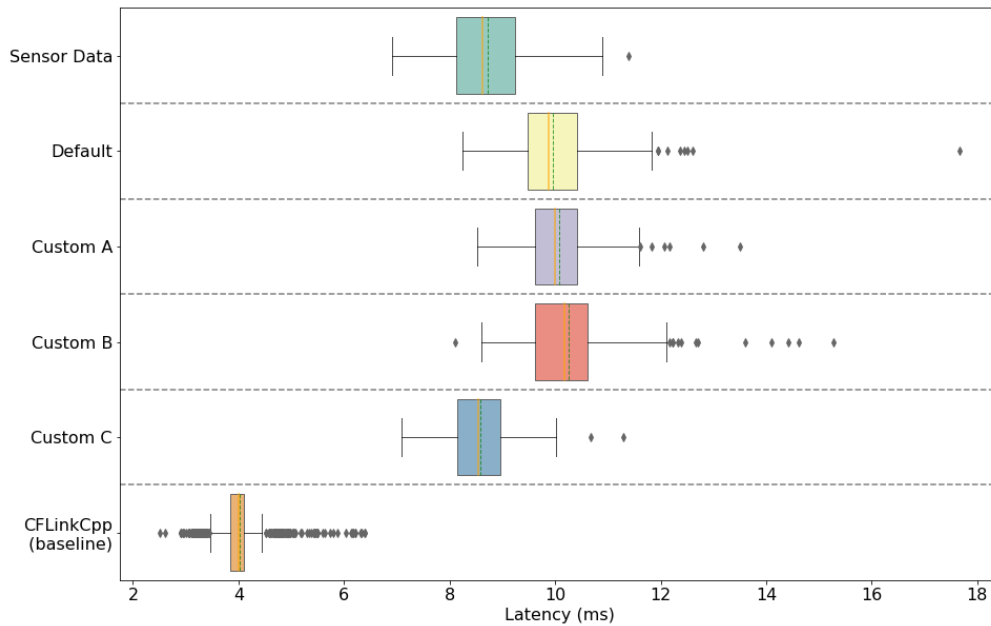


Figure 5.10: QoS profile comparison and CRTP communication baseline ($p_L = 8B$)

The results show that the usage of the micro-ROS/ROS2 and our custom agent has a considerable overhead when compared to the native library baseline. Furthermore, the *Sensor Data* and *Custom C* QoS profiles showed a lower latency compared to the rest of the profiles indicating that the RELIABILITY policy has the most significant impact on the GS-MAV which is to be expected due to the need for transmitting additional confirmation messages and heartbeats.

Next, the same test was performed but with an increasing message size. The results can be seen in Figure 5.11 and show that: the latency increases quickly with the payload size (most likely due to the packet fragmentation resulting of the SFP and the multiple function calls needed to handle the data on the agent-side) and that the impact of the QoS profile on the median latency is similar despite the message size. In this test, the frequency was set to 10Hz and 512 samples were taken for each (payload, QoS profile) tuple.

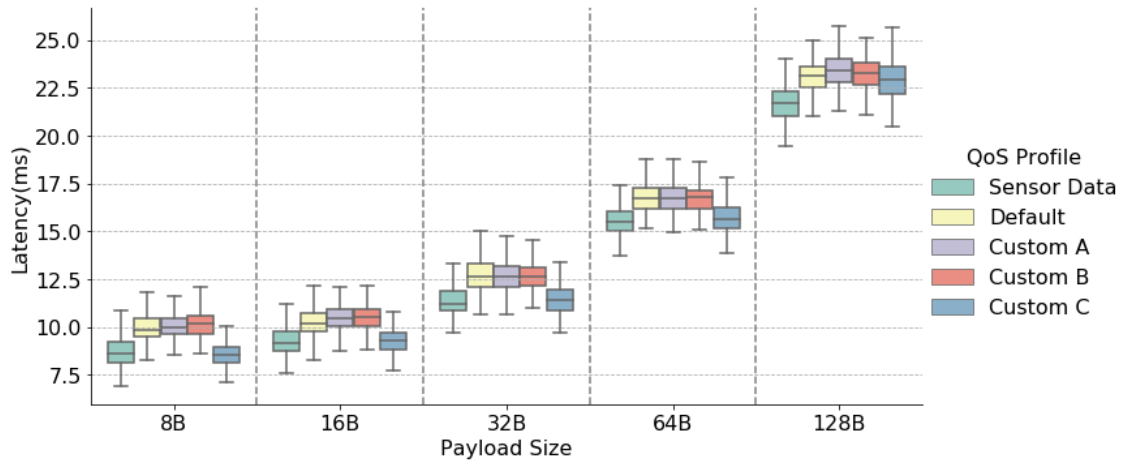


Figure 5.11: Latency comparison for the different QoS profiles

Impact of Reliability on Latency

As the RELIABILITY policy was shown to be the most impactful, the remainder of evaluations will only compare the *Sensor Data* and *Default* profiles. Figures 5.12 and 5.13 show the variation of the latency with message size and publishing frequency respectively.

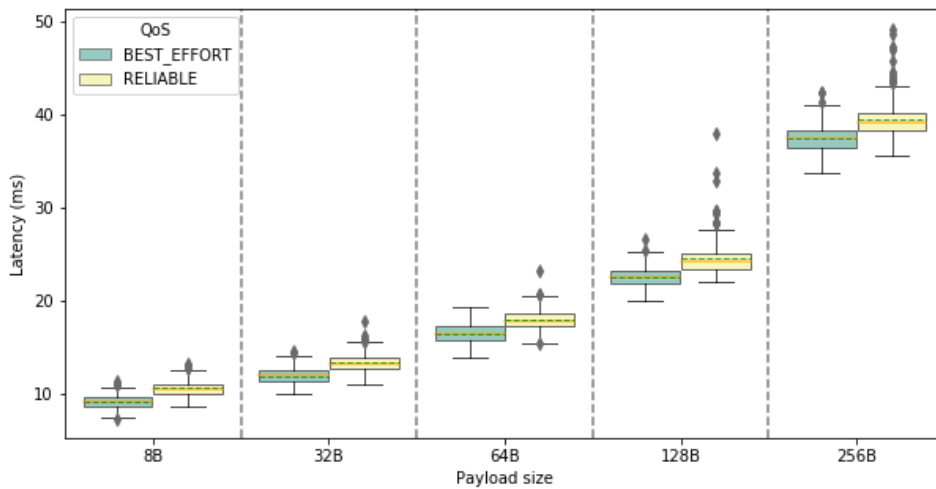


Figure 5.12: Latency w.r.t. payload size for different QoS RELIABILITY policies ($f = 1Hz$)

The results in Figure 5.12 show that the experiments using the *Sensor Data* profile have slightly lower latency than the ones using the *Default*. The spread of the outliers is also larger when using the *Default* profile as the middleware performs multiple re-transmissions to ensure the delivery of the message.

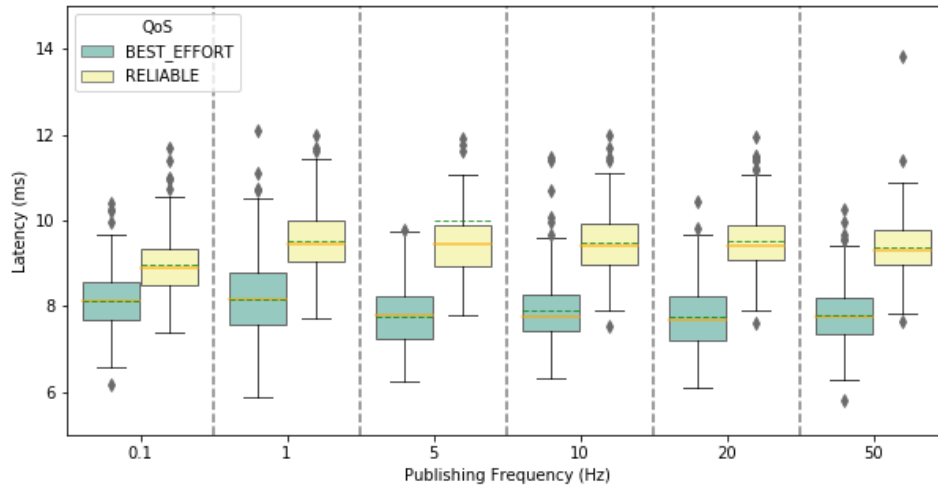


Figure 5.13: Latency w.r.t. publishing frequency for different QoS RELIABILITY profiles ($p_L = 8B$).

Concerning the publishing frequency experiments in Figure 5.13, no significant change was observed between all trials.

Impact of flight on latency

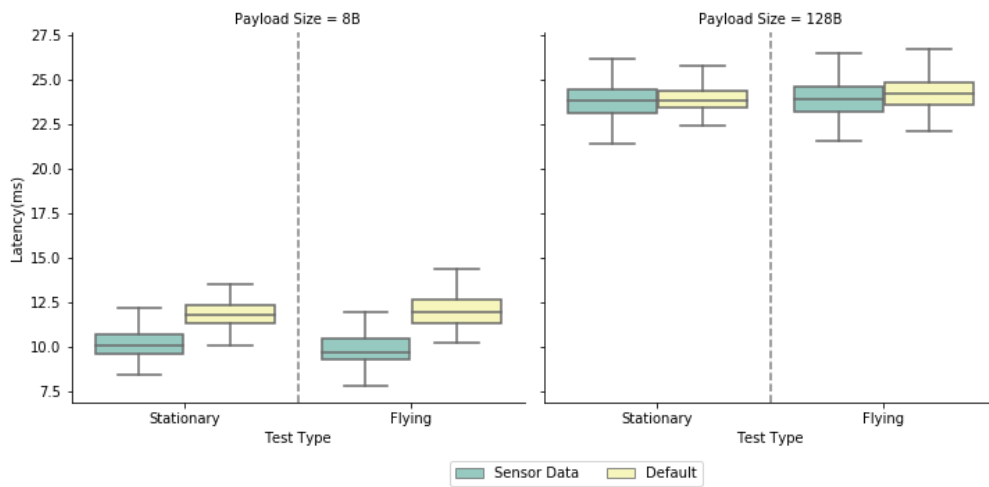


Figure 5.14: Latency comparison for a stationary and airborne MAV.

Additionally, a comparison of the latency with stationary MAVs and during flight (following a simple square-shaped trajectory) was performed. The results can be seen in Figure 5.14 which show that there is no significant latency change between having the MAV stationary or flying.

Multiple MAVs

To test the performance for more than one MAVs, a simple RTT test was made using both the Custom CRTP for multiple MAVs and a single one. Test conditions are similar to the previous tests: 4B messages published at 1Hz. Only one of the MAVs (the same one for both trials) is sending/receiving test messages. The comparison of the two trials is represented in Figure 5.15

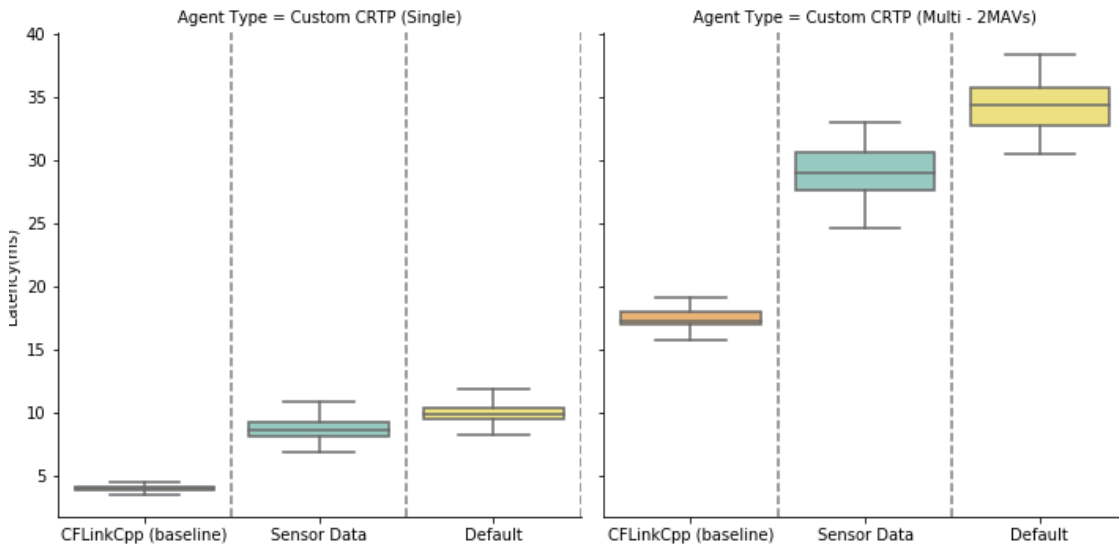


Figure 5.15: Latency comparison for a single and two MAVs.

The results show a clear performance loss when using two MAVs instead of a single one. However, the baseline results show that this increase in latency is not only due to the agent implementation. An implication of this result is that, in order to use large swarms of MAVs, additional tests are needed to understand sources of latency in the agent implementation as well as improve the concurrent access to the Crazyradio PA.

5.3.2 End-to-end latency

In this test, we extend our setup to not run the secondary node on the GS but on the cloud AWS EC2 instance which also requires the addition of the integration-service. The message type is the same as before: the *Default* QoS profile, a fixed publishing rate of 1Hz was used and 512 samples for each payload size.

This setup, represented in Figure 5.16, consists of adding an intermediary node in the GS. The Cloud-MAV latency, Δt_1 , is obtained in the cloud node and the GS-MAV latency, Δt_2 , in the GS node. The Cloud-MAV latency can be obtained as the difference, $\Delta t_1 - \Delta t_2$, between these two.

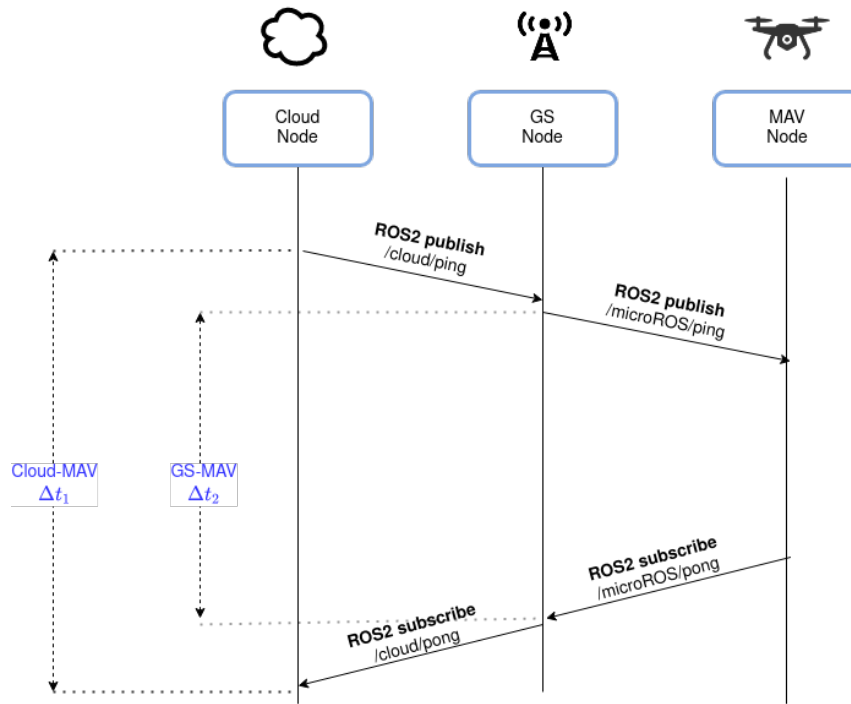


Figure 5.16: End-to-end latency measurement setup.

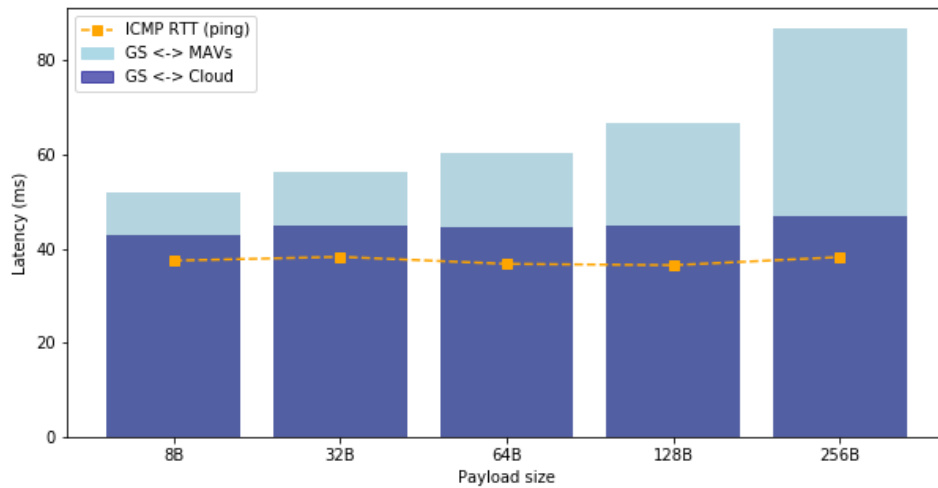


Figure 5.17: Median end-to-end latency profiling

The results of the experiment are shown in the Figure 5.17. It can be seen that there is a constant latency in all measurements due to the WAN communication with the cloud. This latency corresponds to the "ping" of the EC2 instance with a minimal overhead due to the use of the integration-service. No significant changes in the latency caused by the WAN communication were observed for the payload sizes used. However, there is a significant increase in latency with respect to payload caused by the GS -MAV communication as described in the previous subsection.

5.4 Summary

- The MAVs were able to successfully complete the mission plans generated through the planning module as well as requests from alarms. During these missions, the MAVs have shown to be capable of stable flight.
- Integration of all modules was successful and facilitated by the usage of micro-ROS/ROS2 frameworks. Moreover, evaluation of the latency in the various stages of the proposed system validates the viability of the frameworks in MAV sensing systems. However, the results indicate a bottleneck caused by the GS-MAV communication not only because of the overhead of the XRCE-DDS middleware but due to the micro-ROS agent. This indicates that there is still room for improvement on the custom CRTP agent implementation.
- MAVs were able to perform IAQ measurements using the on-board sensor and publish them in a common topic. As stated in the implementation chapter, additional calibration is required to validate the accuracy of the measurements. However, in the mapping experiment, the MAV was able to identify the contaminant source in the arena indicating that the sensor is working and the integration of the latter in the firmware and the overall communication architecture was successful.
- The planning module was able to optimize the mission duration and consequently the battery life of the MAVs. The addition of the measurement time to the traditional MmTSP formulation was shown to result in a more accurate cost function for this kind of application. Practical battery consumption tests show that the battery consumption is approximately linear with time and that the used sensor reduces the battery life by a considerable amount.
- As the planner uses a branch-and-cut based solver, its performance escalates poorly with the number of MAVs and waypoints used. For the proposed use-case, the performance is acceptable but other approaches, such as heuristic-based algorithms, should be considered if reducing the planning time is preferred over optimality. The planner was able to run on a resource-constrained device (Raspberry Pi) and leverage cloud computing effort to offload computation and reduce planning time.
- Our evaluation of the MAV-to- GS has shown that the latency increases proportionally to the payload of the message, since the micro- ROS agent forces fragmentation of the message and thus the number of packets increases proportionally. It has been shown that the MAV-to-cloud latency is well described by an additive model that includes the MAV-GS latency and the (tendentiously) constant GS-Cloud latency.

Chapter 6

Conclusions and Future Work

As the final chapter of this dissertation, this chapter attempts to summarize the conclusions gathered during the development of this project in Section 6.1, and provide a critical reflection of the limitations and future work in Section 6.2.

6.1 Conclusions

The main goal of this thesis to develop an end-to-end MAV sensing system based on the ROS2/micro-ROS frameworks was completed with success as well as the various demos done in the scope of the partnership with Capgemini Engineering. The usage of the aforementioned frameworks allowed the embedded systems to easily access and share data with other platforms and thus solving the "interoperability challenge" associated with integrating micro-controller based robots in IIoT systems.

During the course of this work, many other modules needed to be implemented to achieve a "complete use-case" namely the integration of the IAQ sensor in the MAV firmware, visualization tools, cloud monitoring and the creation of a mission planning ROS2 package. In typical ROS fashion, most packages leverage previously developed tools created by the open-source community and are modular allowing them to easily be used in other Capgemini Engineering projects.

Validation of the developed system was done through real-world experiments using the crazyflie platform. These experiments validate the ability of the MAVs to act as airborne sensing systems, execute missions planned on a GS using a custom-developed ROS2 planning node based on the MILP formulation of the MmTSP whilst considering measurement times and to seamlessly communicate with the IoT devices, GS and cloud nodes through the usage of DDS.

The planning node was shown to be able to create trajectories that optimize the mission time at the cost of having an extensive computation time. The planning time and workload can be reduced through the usage of cloud computing allowing the GS to be executed on embedded devices such as a Raspberry Pi.

An evaluation of the communication systems was also performed, indicating a latency bottleneck in the communication between the GS and the MAVs due to the overhead of using micro-ROS which was found to increase rapidly with the payload size limiting the usage of large message types. The QoS functionalities of DDS were also taken advantage of, namely the DURABILITY, HISTORY and RELIABILITY policies where the former was shown to have the most significant impact on the latency. A custom micro XRCE-DDS agent for the CRTP transport was developed in order to improve the overall latency of the system but needs further work to improve the performance when communicating with multiple MAVs.

Nonetheless, we consider that despite the increased latency, the interoperability and modularity granted by the usage of ROS2/micro-ROS can facilitate the integration of resource-constrained robots in sensing systems.

6.2 Limitations and Future Work

During the course of this project, some limitations of the proposed architecture/system were noted which require future work in order to overcome:

- A major limitation to the practical tests was the size of the test arena available for testing, as future work testing in a larger arena and then in a real indoor environment (i.e. in an industrial setting). In this work, an open-space environment is considered but this scenario is rare in the real-world and as such the planning would need to accommodate additional aspects of the sensing area such as obstacles and no-flight zones.
- Micro-ROS/ROS2 integration was accomplished successfully but the communication performance can be improved especially in the case of the multi-CRTP agent by additional programming effort in the concurrency of the custom transport (for each connection to a MAV). Furthermore, a more in-depth study and profiling of the GS-MAV communications could be useful for characterizing, model and possibly reduce the latency introduced by the agent.
- In order to escalate the proposed architecture to large areas (with a high number of waypoints and/or MAVs), the planning approach needs to be changed for another heuristic-based strategy such as a Genetic Algorithm if reducing planning time is relevant. Due to the modularity of ROS2, this implies only modifying the planner node.
- The algorithm for task assignment (alarms) is simplistic and needs to be improved for situations where multiple alarms occurring at the same time is expected or giving different priorities to them is of interest.
- As mentioned previously, in order to use the sensor in a real application where accuracy is relevant, an additional calibration using known sources and expert knowledge is needed. Additionally, as the architecture is generic, other sensors can be used for different use-cases such as a camera for visual inspection.

Appendix A

Custom micro-ROS/ROS2 Interfaces and configuration files

This appendix contains the full description of the topics and services used and their publisher-s/subscriber or client/servers in section A.1. The definition of the custom message types is also provided in A.2. Lastly, the relevant configuration files are listed in section A.3.

A.1 Full Service/Topic Listing

A.1.1 MAV

Table A.1: MAV Publishers

Topic	Type	QoS	Description
/drone<id>/odometry	geometry_msgs/Point32	BE, VT	X, Y, Z values from internal EKF
/drone<id>/attitude	geometry_msgs/Point32	BE, VT	Roll, pitch, yaw angles from EKF
/drone<id>/battery	cf_messages/Batt	BE, VT	Battery voltage and epoch
/drone<id>/state	cf_messages/State	RE, VT	State of FSM
/airsense/iaq	cf_messages/Airquality	RE, VT	eCO2 and TVOC readings from SGP30
/drone<id>/pong	cf_messages/Ping	NA*	Returns messages for rtt measurement
/auctioneer/bid	auction_interfaces/AuctionBid	RE, VT	Bids the calculated cost for performing a given task.
/auctioneer/ack	auction_interfaces/AuctionAck	RE, VT	Acknowledgement of the auction result.

Table A.2: MAV Subscribers

Topic	Type	QoS	Description
/rpi/weather_data/	cf_messages/Weather_data	RE, TL	Humidity and temperature values
/drone<id>/ping	cf_messages/Ping	NA*	Receive messages for rtt measurement
/auctioneer/notify	auction_interfaces/AuctionNotify	RE, VT	Receive auction notification
/auctioneer/result	auction_interfaces/AuctionResult	RE, VT	Receive auction result

Table A.3: Services used by the GUI (as a server)

Service	Type	Description
/drone<id>/takeoff	cf_messages/TakeOff	Take-off with a given height and duration.
/drone<id>/land	cf_messages/Land	Land with a given height for shutting down the motors and duration.
/drone<id>/gotoabs	cf_messages/GoTo	Movement w.r.t. to the odom frame.
/drone<id>/gotorel	cf_messages/GoTo	Movement w.r.t. to the MAV frame.
/drone<id>/upload_trajectory	cf_messages/Trajectory	Writes a trajectory to the MAV's internal memory
/drone<id>/execute_trajectory	std_msgs/Trigger	Executes pre-uploaded trajectory
/drone<id>/reset_trajectory	std_msgs/Trigger	Resets trajectory.

A.1.2 RQT Plugin (GUI)

Table A.4: Services used by the GUI (as a client)

Service	Type	Description
/Planner/plan_trajectories	cf_messages/VrpServer	Send planning parameters. Returns optimal trajectories.
/drone<id>/upload_trajectory	cf_messages/Trajectory	Writes a trajectory to the MAV's internal memory
/drone<id>/execute_trajectory	std_msgs/Trigger	Executes pre-uploaded trajectory
/drone<id>/reset_trajectory	std_msgs/Trigger	Resets trajectory.

Table A.5: Subscribers

Topic	Type	QoS	Description
/drone<id>/odometry	geometry_msgs/Point32	BE, VT	X, Y, Z values from all MAVs
/drone<id>state	cf_messages/State	RE, VT	State of all MAVs

Table A.6: RQT Plugin (GUI) Subscribers

Topic	Type	QoS	Description
/waypoint_markers	visualization_msgs/Marker	RE, TL	Waypoint markers for visualization in RVIZ

A.1.3 Planner

Table A.7: RQT Plugin (GUI) Services used by the Planner (as a server)

Service	Type	Description
/Planner/plan_trajectories	cf_messages/VrpServer	Plan trajectories given MAVs and waypoints.

A.1.4 RVIZ

Table A.8: RVIZ Subscribers

Topic	Type	QoS	Description
/drone<id>/path	nav_msgs/Path	RE, VT	Real-time trajectory of the MAV (obtained from the tfs)
/tf	tf2_msgs/TFMessage	RE, VT	Transformation frames of the MAVs
/tf_static	tf2_msgs/TFMessage	RE, VT	Transformation frame of the odom in relation to the map
/map	nav_msgs/OccupancyGrid	RE, VT	Representation of the arena (only for visualization in this project)
/waypoint_markers	visualization_msgs/Marker	RE, VT	Regular waypoint markers (green spheres)
/alarm_markers	visualization_msgs/Marker	RE, VT	Alarm waypoint markers (red spheres)

Table A.9: Actions used by the alarm node (as a client)

Action	Type	Description
/gsc/task_manager	alarm_interfaces/SendTask	Used to issue an alarm.

A.1.5 Alarm node

A.1.6 Task Manager

Table A.10: Actions used by the task manager (as a server)

Action	Type	Description
/gsc/task_manager	alarm_interfaces/SendTask	Receive alarms and enqueue them to be "auctioned".

Table A.11: Services used by the task manager (as a client)

Action	Type	Description
/auctioneer/create_auction	auction_interfaces/Auction	Send task to be "auctioned".

A.1.7 Auctioneer

Table A.12: Services used by the task auctioneer (as a server)

Action	Type	Description
/auctioneer/create_auction	auction_interfaces/Auction	Create auction for given task.

Table A.13: Auctioneer subscribers

Topic	Type	QoS	Description
/auctioneer/bid	auction_interfaces/AuctionBid	RE, VT	Receive bids of the calculated cost of performing a given task for each MAV.
/auctioneer/ack	auction_interfaces/AuctionAck	RE, VT	Receive acknowledgements of the auction result.

Table A.14: Auctioneer publishers

Topic	Type	QoS	Description
/auctioneer/notify	auction_interfaces/AuctionNotify	RE, VT	Notify MAVs of an auction
/auctioneer/result	auction_interfaces/AuctionResult	RE, VT	Send auction result to MAVs

A.1.8 Weather Publisher

Table A.15: Weather publisher subscribers

Topic	Type	QoS	Description
/rpi/weather_data	cf_messages/weather_data	RE, TL	Combined relative humidity and temperature topic.
/rpi/rh	std_msgs/float32	RE, TL	Relative Humidity (percentage)
/rpi/temperature	std_msgs/float32	RE, TL	Temperature (in Celsius)

A.1.9 AWS Cloudwatch

Table A.16: Cloudwatch-related subscribers

Topic	Type	QoS	Description
/metrics	ros_monitoring_msgs/MetricList	RE, VT	Listen to metrics, namely weather and IAQ data converted using the cw_bridge.
/rosout	rcl_interfaces/Log	RE, VT	Listen and record logs from ROS2 applications.

A.2 Custom Service/Topic definition

A.2.1 cf_messages

A.2.1.1 Services

Listing A.1: cf_messages/TakeOff

```
float32 height
float32 duration
```

```
---
int8 ret
```

Listing A.2: cf_messages/Land

```
float32 height
float32 duration
---
int8 ret
```

Listing A.3: cf_messages/GoTo

```
float32 x
float32 y
float32 z
float32 yaw
float32 duration
---
int8 ret
```

Listing A.4: cf_messages/SimpleTraj

```
Trajectory traj
---
int8      ret
```

Listing A.5: cf_messages/VrpServer

```
Waypoint[] targets           # <x,y,z,yaw> location of the sensing waypoints
Waypoint[] drone_pos         # <x,y,z,yaw> location of the depots
int8[] drone_ids              # IDs of the MAVS
float32 linear_speed          # Desired linear speed between waypoints
---
string message                # Solver result message
float64 t2
float64 t3
int8[] drone_ids              # IDs of the MAVS
Trajectory[] trajectories     # Optimal trajectories ordered by MAV id
```

A.2.1.2 Messages

Listing A.6: cf_messages/AirQuality

```
geometry_msgs/Point32 p
uint16 eco2
uint16 tvoc
```

Listing A.7: cf_messages/Waypoint

```
float32 x
float32 y
float32 z
float32 yaw
```

Listing A.8: cf_messages/Trajectory

```
Waypoint[] waypoints
float32 speed
int8 len
```

Listing A.9: cf_messages/Batt

```
int64 epoch
float32 batt_level
```

Listing A.10: cf_messages/WeatherData

```
float32 temperature
float32 rh
```

Listing A.11: cf_messages/Ping

```
uint8[] data
```

A.2.2 alarm_interfaces

A.2.2.1 Actions

Listing A.12: alarm_interfaces/SendTask

```
Task task
---
bool rv
cf_messages/AirQuality measurement
---
uint8 assigned_id
float32 value
```

A.2.2.2 Messages

Listing A.13: alarm_interfaces/Task

```
cf_messages/Waypoint p
uint8 sender_id
uint8 seq_num
uint8 task_type
uint8 robot_type
```

A.2.3 auction_interfaces

A.2.3.1 Messages

Listing A.14: auction_interfaces/AuctionNotify

```
alarm_interfaces/Task task
```

Listing A.15: auction_interfaces/AuctionBid

```
uint8 bidder_id
float32 value
```

Listing A.16: auction_interfaces/AuctionResult

```
uint8 bidder_id
```

Listing A.17: auction_interfaces/AuctionAck

```
bool ret
```

A.3 Configuration Files

A.3.1 Integration Service

Listing A.18: Fast DDS profiles for integration service (WAN communication)

```
<?xml version="1.0"?>
<profiles>
  <transport_descriptors>
    <transport_descriptor>
      <transport_id>tcp_transport_server</transport_id>
      <type>TCPv4</type>
      <listening_ports>
        <port>5100</port>
      </listening_ports>
      <wan_addr>AWS_EC2_PUBLIC_IP</wan_addr>
    </transport_descriptor>
  </transport_descriptors>

  <transport_descriptors>
    <transport_descriptor>
      <transport_id>tcp_transport_client</transport_id>
      <type>TCPv4</type>
    </transport_descriptor>
  </transport_descriptors>

  <participant profile_name="is_profile_server">
    <rtps>
      <userTransports>
        <transport_id>tcp_transport_server</transport_id>
      </userTransports>
      <builtin>
        <discovery_config>
          <leaseAnnouncement>
            <sec>1</sec>
          </leaseAnnouncement>
        </discovery_config>
      </builtin>
    </rtps>
  </participant>
</profiles>
```

```

        <useBuiltinTransports>false</useBuiltinTransports>
    </rtsp>
</participant>

<participant profile_name="is_profile_client">
    <rtsp>
        <userTransports>
            <transport_id>tcp_transport_client</transport_id>
        </userTransports>
        <builtin>
            <initialPeersList>
                <locator>
                    <tcpv4>
                        <physical_port>5100</physical_port>
                        <address>AWS_EC2_PUBLIC_IP</address>
                    </tcpv4>
                </locator>
            </initialPeersList>
            <discovery_config>
                <leaseAnnouncement>
                    <sec>1</sec>
                </leaseAnnouncement>
            </discovery_config>
        </builtin>
        <useBuiltinTransports>false</useBuiltinTransports>
    </rtsp>
</participant>

</profiles>

```

Note that for this configuration to work, port forwarding must be configured for port 5100. This file has been based on the WAN communication example ¹.

Listing A.19: Client-side ROS2 <-> DDS mapping

```

systems:
  ros2: { type: ros2 }
  dds:
    type: fastdds
    types-from: ros2
    participant:
      file_path: "wan_config.xml"
      profile_name: is_profile_client

routes:
  ros2_to_dds: { from: ros2, to: dds }

```

¹https://integration-service.docs.eprosima.com/en/latest/examples/wan_communication/wan_tcp_dds.html

```

dds_to_ros2: { from: dds, to: ros2}
dds_server:
  server: dds
  clients: [ros2]

services:
  cloud_plan_routes: {
    request_type: "cf_messages/VrpServer:request",
    reply_type: "cf_messages/VrpServer:response",
    route: dds_server,
    remap: {
      ros2: {
        request_type: "cf_messages/VrpServer:request",
        reply_type: "cf_messages/VrpServer:response"
      }
    }
  }

topics:
  airquality: {type: "cf_messages/AirQuality", route: ros2_to_dds, remap: {dds: { type: "
    cf_messages/AirQuality", topic: AirQualityTopicC}}}
  microROS/pong: {type: "cf_messages/Ping", route: ros2_to_dds, remap: {dds: { type: "
    cf_messages/Ping", topic: MicroRosPongC}}}
  MicroRosPingC: {type: "cf_messages/Ping", route: dds_to_ros2, remap: {ros2: { type: "
    cf_messages/Ping", topic: microROS/ping}}}

```

Listing A.20: Server-side ROS2 <-> DDS mapping

```

systems:
  ros2: { type: ros2 }

  dds:
    type: fastdds
    types-from: ros2
    participant:
      file_path: "wan_config.xml"
      profile_name: is_profile_server

routes:
  dds_to_ros2: { from: dds, to: ros2 }
  ros2_to_dds: { from: ros2, to: dds }

```

```
ros2_server: {server: ros2, clients: dds }

services:
  cloud_plan_routes: {
    request_type: "cf_messages/VrpServer:request",
    reply_type: "cf_messages/VrpServer:response",
    route: ros2_server,
    remap: {
      ros2: {
        request_type: "cf_messages/VrpServer:request",
        reply_type: "cf_messages/VrpServer:response"
      }
    }
  }
}

topics:
  airquality: {type: "cf_messages/AirQuality", route: dds_to_ros2, remap: { dds: { type: "
    cf_messages/AirQuality", topic: AirQualityTopicC }}}
  microROS/pong: {type: "cf_messages/Ping", route: dds_to_ros2, remap: { dds: { type: "
    cf_messages/Ping", topic: MicroRosPongC }}}
  MicroRosPingC: {type: "cf_messages/Ping", route: ros2_to_dds, remap: { ros2: { type: "
    cf_messages/Ping", topic: microROS/ping }}}}
```


References

- [1] Vijay Kumar and Nathan Michael. Opportunities and challenges with autonomous micro aerial vehicles. *The International Journal of Robotics Research*, 31(11):1279–1291, September 2012. URL: <http://journals.sagepub.com/doi/10.1177/0278364912455954> [last accessed 2020-12-27], doi:10.1177/0278364912455954.
- [2] Juan Jesús Roldán, Guillaume Joossen, David Sanz, Jaime Del Cerro, and Antonio Barrientos. Mini-UAV Based Sensory System for Measuring Environmental Variables in Greenhouses. *Sensors*, 15(2):3334–3350, February 2015. Number: 2 Publisher: Multidisciplinary Digital Publishing Institute. URL: <https://www.mdpi.com/1424-8220/15/2/3334> [last accessed 2020-12-11], doi:10.3390/s150203334.
- [3] Martin Saska, Vojtěch Vonásek, Jan Chudoba, Justin Thomas, Giuseppe Loianno, and Vijay Kumar. Swarm Distribution and Deployment for Cooperative Surveillance by Micro-Aerial Vehicles. *Journal of Intelligent & Robotic Systems*, 84(1-4):469–492, December 2016. URL: <http://link.springer.com/10.1007/s10846-016-0338-z> [last accessed 2020-12-09], doi:10.1007/s10846-016-0338-z.
- [4] M. Beul, D. Droeschel, M. Nieuwenhuisen, J. Quenzel, S. Houben, and S. Behnke. Fast Autonomous Flight in Warehouses for Inventory Applications. *IEEE Robotics and Automation Letters*, 3(4):3121–3128, October 2018. Conference Name: IEEE Robotics and Automation Letters. doi:10.1109/LRA.2018.2849833.
- [5] Georgy Skorobogatov, Cristina Barrado, and Esther Salamí. Multiple UAV Systems: A Survey. *Unmanned Systems*, 08(02):149–169, April 2020. URL: <https://www.worldscientific.com/doi/abs/10.1142/S2301385020500090> [last accessed 2020-12-30], doi:10.1142/S2301385020500090.
- [6] Huiru Cao, Yongxin Liu, Xuejun Yue, and Wenjian Zhu. Cloud-Assisted UAV Data Collection for Multiple Emerging Events in Distributed WSNs. *Sensors*, 17(8):1818, August 2017. Number: 8 Publisher: Multidisciplinary Digital Publishing Institute. URL: <https://www.mdpi.com/1424-8220/17/8/1818> [last accessed 2020-12-31], doi:10.3390/s17081818.
- [7] E. Tuyishimire, A. Bagula, S. Rekhis, and N. Boudriga. Cooperative data muling from ground sensors to base stations using UAVs. In *2017 IEEE Symposium on Computers and Communications (ISCC)*, pages 35–41, July 2017. doi:10.1109/ISCC.2017.8024501.
- [8] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio. ROSCH:Real-Time Scheduling Framework for ROS. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 52–58, August 2018. ISSN: 2325-1301. doi:10.1109/RTCSA.2018.00015.

- [9] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. volume 3, page 5. Kobe, Japan, 2009. Issue: 3.2.
- [10] Sophia Kolak, Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. It takes a village to build a robot: An empirical study of the ros ecosystem. pages 430–440, 2020. doi:10.1109/ICSME46990.2020.00048.
- [11] Víctor Mayoral-Vilches, Martin Pinzger, Stefan Rass, Bernhard Dieber, and Endika Gil-Uriarte. Can ROS be used securely in industry? Red teaming ROS-Industrial. *arXiv:2009.08211 [cs]*, September 2020. arXiv: 2009.08211. URL: <http://arxiv.org/abs/2009.08211> [last accessed 2020-12-30].
- [12] Brian Gerkey. Why ROS 2? URL: https://design.ros2.org/articles/why_ros2.html [last accessed 2021-01-10].
- [13] Dirk Thomas. Changes between ROS 1 and ROS 2. URL: <http://design.ros2.org/articles/changes.html> [last accessed 2021-01-10].
- [14] Jens Vankeirsbilck. Introduction of Robot Operating Systems 2: ROS2 – SAFER AUTONOMOUS SYSTEMS. URL: <https://etn-sas.eu/2020/03/23/introduction-of-robot-operating-systems-2-ros2/> [last accessed 2021-01-10].
- [15] Gene Jain. DMTCP: Fixing the Single Point of Failure of the ROS Master. 2017. doi:10.36288/roscon2017-900267.
- [16] roserial - ROS Wiki. URL: <http://wiki.ros.org/roserial> [last accessed 2021-01-29].
- [17] Danilo Tardioli, Ramviyas Parasuraman, and Petter Ögren. Pound: A multi-master ROS node for reducing delay and jitter in wireless multi-robot networks. *Robotics and Autonomous Systems*, 111:73–87, January 2019. URL: <http://www.sciencedirect.com/science/article/pii/S0921889017309144> [last accessed 2021-01-02], doi:10.1016/j.robot.2018.10.009.
- [18] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ROS2. In *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, pages 1–10, New York, NY, USA, October 2016. Association for Computing Machinery. URL: <https://doi.org/10.1145/2968478.2968502> [last accessed 2020-11-28], doi:10.1145/2968478.2968502.
- [19] J. M. Schlesselman, G. Pardo-Castellote, and B. Farabaugh. OMG data-distribution service (DDS): architectural update. In *IEEE MILCOM 2004. Military Communications Conference, 2004.*, volume 2, pages 961–967 Vol. 2, October 2004. doi:10.1109/MILCOM.2004.1494965.
- [20] Y. Liu, Y. Guan, X. Li, R. Wang, and J. Zhang. Formal Analysis and Verification of DDS in ROS2. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–5, October 2018. doi:10.1109/MEMCOD.2018.8556970.

- [21] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B Brandenburg. Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling. page 23, 2019.
- [22] ROS 2 Foxy Fitzroy: Setting a new standard for production robot development, June 2020. Section: Announcements. URL: <https://aws.amazon.com/blogs/robotics/ros2-foxy-fitzroy-robot-development/> [last accessed 2021-01-30].
- [23] About the DDS Interoperability Wire Protocol Specification Version 2.2. URL: <https://www.omg.org/spec/DDSI-RTPS/2.2/> [last accessed 2021-01-31].
- [24] About Quality of Service settings. URL: <https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/> [last accessed 2021-01-31].
- [25] Ruffin White, Dr Henrik I. Christensen, and Dr Morgan Quigley. SROS: Securing ROS over the wire, in the graph, and through the kernel. *arXiv:1611.07060 [cs]*, November 2016. arXiv: 1611.07060. URL: <http://arxiv.org/abs/1611.07060> [last accessed 2021-01-31].
- [26] Robotics security: What is SROS 2? URL: <https://ubuntu.com/blog/what-is-sros-2> [last accessed 2021-01-31].
- [27] About the DDS Security Specification Version 1.1. URL: <https://www.omg.org/spec/DDS-SECURITY/About-DDS-SECURITY/> [last accessed 2021-01-31].
- [28] Fast DDS Documentation. page 700.
- [29] Features and Architecture. URL: <https://micro-ros.github.io//docs/overview/features/> [last accessed 2021-01-31].
- [30] micro-ROS. URL: <https://micro-ros.github.io//> [last accessed 2021-01-31].
- [31] Muhammad Asghar Khan, Ijaz Mansoor Qureshi, and Fahimullah Khanzada. A Hybrid Communication Scheme for Efficient and Low-Cost Deployment of Future Flying Ad-Hoc Network (FANET). *Drones*, 3(1):16, March 2019. Number: 1 Publisher: Multidisciplinary Digital Publishing Institute. URL: <https://www.mdpi.com/2504-446X/3/1/16> [last accessed 2021-02-04], doi:10.3390/drones3010016.
- [32] Open Robotics. ROS World 2020: Dronecode Parallel Session, November 2020. URL: <https://vimeo.com/480477960> [last accessed 2021-02-04].
- [33] Azza Allouch, Omar Cheikhrouhou, Anis Koubaa, Mohamed Khargui, and Tarek Abbes. MAVSec: Securing the MAVLink Protocol for Ardupilot/PX4 Unmanned Aerial Systems. In *2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC)*, pages 621–628, Tangier, Morocco, June 2019. IEEE. URL: <https://ieeexplore.ieee.org/document/8766667/> [last accessed 2021-02-04], doi:10.1109/IWCMC.2019.8766667.
- [34] Emad Samuel Malki Ebeid, Martin Skriver, Kristian Terkildsen, Kjeld Jensen, and Ulrik Schultz. A Survey of Open-Source UAV Flight Controllers and Flight Simulators. *Micro-processors and Microsystems*, 61, May 2018. doi:10.1016/j.micpro.2018.05.002.
- [35] Olimpiya Saha and Raj Dasgupta. A Comprehensive Survey of Recent Trends in Cloud Robotics Architectures and Applications. *Robotics*, 7, August 2018. doi:10.3390/robotics7030047.

- [36] Manal Aissam, Mohammed Benbrahim, and Mohammed Nabil Kabbaj. Cloud Robotic: Opening a New Road to the Industry 4.0. In Nabil Derbel, Jawhar Ghommam, and Quanmin Zhu, editors, *New Developments and Advances in Robot Control*, volume 175, pages 1–20. Springer Singapore, Singapore, 2019. Series Title: Studies in Systems, Decision and Control. URL: http://link.springer.com/10.1007/978-981-13-2212-9_1 [last accessed 2021-02-08], doi:10.1007/978-981-13-2212-9_1.
- [37] V. Dawarka and G. Bekaroo. Cloud Robotics Platforms: Review and Comparative Analysis. In *2018 International Conference on Intelligent and Innovative Computing Applications (ICONIC)*, pages 1–6, December 2018. doi:10.1109/ICONIC.2018.8601220.
- [38] Ali Yahya, Adrian Li, Mrinal Kalakrishnan, Yevgen Chebotar, and Sergey Levine. Collective Robot Reinforcement Learning with Distributed Asynchronous Guided Policy Search. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 79–86, September 2017. arXiv: 1610.00673. URL: <http://arxiv.org/abs/1610.00673> [last accessed 2021-02-08], doi:10.1109/IROS.2017.8202141.
- [39] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg. A Survey of Research on Cloud Robotics and Automation. *IEEE Transactions on Automation Science and Engineering*, 12(2):398–409, April 2015. Conference Name: IEEE Transactions on Automation Science and Engineering. doi:10.1109/TASE.2014.2376492.
- [40] What is PaaS? URL: <https://www.redhat.com/en/topics/cloud-computing/what-is-paas> [last accessed 2021-02-10].
- [41] K.G. Shin and P. Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, January 1994. URL: <http://ieeexplore.ieee.org/document/259423/> [last accessed 2021-02-11], doi:10.1109/5.259423.
- [42] Rajib Mall. *Real-Time Systems: Theory and Practice*. January 2009.
- [43] Zhaolin Chen. PERFORMANCE ANALYSIS OF ROS 2 NETWORKS USING VARIABLE QUALITY OF SERVICE AND SECURITY CONSTRAINTS FOR AUTONOMOUS SYSTEMS. page 88.
- [44] J. Fernandez, B. Allen, P. Thulasiraman, and B. Bingham. Performance Study of the Robot Operating System 2 with QoS and Cyber Security Settings. In *2020 IEEE International Systems Conference (SysCon)*, pages 1–6, August 2020. ISSN: 2472-9647. doi:10.1109/SysCon47679.2020.9275872.
- [45] Tobias Kronauer, Joshua Pohlmann, Maximilian Matthe, Till Smejkal, and Gerhard Fetsch. Latency Overhead of ROS2 for Modular Time-Critical Systems. *arXiv:2101.02074 [cs]*, January 2021. arXiv: 2101.02074. URL: <http://arxiv.org/abs/2101.02074> [last accessed 2021-01-29].
- [46] Carlos San Vicente Gutiérrez, Lander Usategui San Juan, Irati Zamalloa Ugarte, and Víctor Mayoral Vilches. Real-time Linux communications: an evaluation of the Linux communication stack for real-time robotic applications. *arXiv:1808.10821 [cs]*, August 2018. arXiv: 1808.10821. URL: <http://arxiv.org/abs/1808.10821> [last accessed 2021-01-04].

- [47] M. Reke, D. Peter, J. Schulte-Tigges, S. Schiffer, A. Ferrein, T. Walter, and D. Mathis. A Self-Driving Car Architecture in ROS2. In *2020 International SAUPEC/RobMech/PRASA Conference*, pages 1–6, January 2020. doi:[10.1109/SAUPEC/RobMech/PRASA48453.2020.9041020](https://doi.org/10.1109/SAUPEC/RobMech/PRASA48453.2020.9041020).
- [48] J. Park, R. Delgado, and B. W. Choi. Real-Time Characteristics of ROS 2.0 in Multiagent Robot Systems: An Empirical Study. *IEEE Access*, 8:154637–154651, 2020. Conference Name: IEEE Access. doi:[10.1109/ACCESS.2020.3018122](https://doi.org/10.1109/ACCESS.2020.3018122).
- [49] Ziyue Jiang, Yifan Gong, Jidong Zhai, Yu-Ping Wang, Wei Liu, Hao Wu, and Jiangming Jin. Message Passing Optimization in Robot Operating System. *International Journal of Parallel Programming*, 48(1):119–136, February 2020. URL: <https://doi.org/10.1007/s10766-019-00647-w> [last accessed 2020-12-31], doi:[10.1007/s10766-019-00647-w](https://doi.org/10.1007/s10766-019-00647-w).
- [50] Bander Alzahrani, Omar Sami Oubbati, Ahmed Barnawi, Mohammed Atiquzzaman, and Daniyal Alghazzawi. UAV assistance paradigm: State-of-the-art in applications and challenges. *Journal of Network and Computer Applications*, 166:102706, September 2020. URL: <https://linkinghub.elsevier.com/retrieve/pii/S1084804520301806> [last accessed 2021-02-05], doi:[10.1016/j.jnca.2020.102706](https://doi.org/10.1016/j.jnca.2020.102706).
- [51] Shidrokh Goudarzi, Nazri Kama, Mohammad Hossein Anisi, Sherali Zeadally, and Shahid Mumtaz. Data collection using unmanned aerial vehicles for Internet of Things platforms. *Computers & Electrical Engineering*, 75:1–15, May 2019. URL: <https://www.sciencedirect.com/science/article/pii/S0045790618322250> [last accessed 2021-02-06], doi:[10.1016/j.compeleceng.2019.01.028](https://doi.org/10.1016/j.compeleceng.2019.01.028).
- [52] S. Arabi, H. Elbiaze, E. Sabir, and M. Sadik. Tradeoffs for Data Collection and Wireless Energy Transfer Dilemma in IoT Environments. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2018. ISSN: 1938-1883. doi:[10.1109/ICC.2018.8422712](https://doi.org/10.1109/ICC.2018.8422712).
- [53] C. Y. Tazibt, M. Bekhti, T. Djamah, N. Achir, and K. Boussetta. Wireless sensor network clustering for UAV-based data gathering. In *2017 Wireless Days*, pages 245–247, March 2017. doi:[10.1109/WD.2017.7918154](https://doi.org/10.1109/WD.2017.7918154).
- [54] Qiyue Wu, Peng Sun, and Azzedine Boukerche. An Energy-efficient UAV-based Data Aggregation Protocol in Wireless Sensor Networks. In *Proceedings of the 8th ACM Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications - DIVANet’18*, pages 34–40, Montreal, QC, Canada, 2018. ACM Press. URL: <http://dl.acm.org/citation.cfm?doid=3272036.3272047> [last accessed 2021-02-06], doi:[10.1145/3272036.3272047](https://doi.org/10.1145/3272036.3272047).
- [55] Julian Bermudez Ortega. Drone use-case - Initial. page 15.
- [56] Maurizio Rossi and Davide Brunelli. Gas Sensing on Unmanned Vehicles: Challenges and Opportunities. In *2017 New Generation of CAS (NGCAS)*, pages 117–120, September 2017. doi:[10.1109/NGCAS.2017.58](https://doi.org/10.1109/NGCAS.2017.58).
- [57] Javier Burgués, Victor Hernández, Achim J. Lilienthal, and Santiago Marco. Smelling Nano Aerial Vehicle for Gas Source Localization and Mapping. *Sensors*, 19(3):478, January 2019. Number: 3 Publisher: Multidisciplinary Digital Publishing Institute. URL:

- <https://www.mdpi.com/1424-8220/19/3/478> [last accessed 2021-07-26], doi: 10.3390/s19030478.
- [58] Patrick P. Neumann, Paul Hirschberger, Zhandos Baurzhan, Carlo Tiebe, Michael Hofmann, Dino Hüllmann, and Matthias Bartholmai. Indoor Air Quality Monitoring using flying Nanobots: Design and Experimental Study. In *2019 IEEE International Symposium on Olfaction and Electronic Nose (ISOEN)*, pages 1–3, May 2019. doi:10.1109/ISOEN.2019.8823496.
- [59] Bardienus P. Duisterhof, Shushuai Li, Javier Burgués, Vijay Janapa Reddi, and Guido C. H. E. de Croon. Sniffy Bug: A Fully Autonomous Swarm of Gas-Seeking Nano Quadcopters in Cluttered Environments. *arXiv:2107.05490 [cs]*, July 2021. arXiv: 2107.05490. URL: <http://arxiv.org/abs/2107.05490> [last accessed 2021-07-26].
- [60] M. Gharibi, R. Boutaba, and S. L. Waslander. Internet of Drones. *IEEE Access*, 4:1148–1162, 2016. Conference Name: IEEE Access. doi:10.1109/ACCESS.2016.2537208.
- [61] A. Koubâa, B. Qureshi, M. Sriti, Y. Javed, and E. Tovar. A service-oriented Cloud-based management system for the Internet-of-Drones. In *2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 329–335, April 2017. doi:10.1109/ICARSC.2017.7964096.
- [62] Chen Hong and Dianxi Shi. A Cloud-based Control System Architecture for Multi-UAV. In *Proceedings of the 3rd International Conference on Robotics, Control and Automation - ICRCA '18*, pages 25–30, Chengdu, China, 2018. ACM Press. URL: <http://dl.acm.org/citation.cfm?doid=3265639.3265652> [last accessed 2021-01-03], doi:10.1145/3265639.3265652.
- [63] G. Mehrooz, E. Ebeid, and P. Schneider-Kamp. System Design of an Open-Source Cloud-Based Framework for Internet of Drones Application. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 572–579, August 2019. doi:10.1109/DSD.2019.00087.
- [64] Marouane Salhaoui, Antonio Guerrero-González, Mounir Arioua, Francisco J. Ortiz, Ahmed El Oualkadi, and Carlos Luis Torregrosa. Smart Industrial IoT Monitoring and Control System Based on UAV and Cloud Computing Applied to a Concrete Plant. *Sensors*, 19(15):3316, January 2019. Number: 15 Publisher: Multidisciplinary Digital Publishing Institute. URL: <https://www.mdpi.com/1424-8220/19/15/3316> [last accessed 2021-02-09], doi:10.3390/s19153316.
- [65] W. Chen, B. Liu, H. Huang, S. Guo, and Z. Zheng. When UAV Swarm Meets Edge-Cloud Computing: The QoS Perspective. *IEEE Network*, 33(2):36–43, March 2019. Conference Name: IEEE Network. doi:10.1109/MNET.2019.1800222.
- [66] Home | Bitcraze. URL: <https://www.bitcraze.io/> [last accessed 2021-02-10].
- [67] Crazyflie 2.0: System Architecture | Bitcraze. URL: <https://www.bitcraze.io/2014/07/crazyflie-2-0-system-architecture/> [last accessed 2021-02-10].
- [68] Loco Positioning system | Bitcraze. URL: <https://www.bitcraze.io/documentation/system/positioning/loco-positioning-system/> [last accessed 2021-08-23].

- [69] Loco Positioning node | Bitcraze. URL: <https://www.bitcraze.io/products/loco-positioning-node/> [last accessed 2021-08-23].
- [70] Loco Positioning deck | Bitcraze. URL: <https://www.bitcraze.io/products/loco-positioning-deck/> [last accessed 2021-08-23].
- [71] Flow deck v2 | Bitcraze. URL: <https://www.bitcraze.io/products/flow-deck-v2/> [last accessed 2021-08-23].
- [72] The Commander Framework | Bitcraze. URL: https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/sensor-to-control/commanders_setpoints/ [last accessed 2021-07-29].
- [73] Matheus Pedroso Sanches. Visual Flight Rules-based Collision Avoidance System for VTOL UAV. Master's thesis, FEUP, Universidade do Porto, 2020.
- [74] Micro XRCE-DDS Agent, August 2021. original-date: 2017-12-05T13:41:14Z. URL: <https://github.com/eProsima/Micro-XRCE-DDS-Agent> [last accessed 2021-08-11].
- [75] Transport — eProsima Micro XRCE-DDS 2.0.0 documentation. URL: <https://micro-xrce-dds.docs.eprosima.com/en/latest/transport.html> [last accessed 2021-08-25].
- [76] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209–219, June 2006. URL: <https://www.sciencedirect.com/science/article/pii/S0305048304001550> [last accessed 2021-07-26], doi:10.1016/j.omega.2004.10.004.
- [77] Imdat Kara and Tolga Bektas. Integer linear programming formulations of multiple salesman problems and its variations. *European Journal of Operational Research*, 174(3):1449–1458, November 2006. URL: <https://www.sciencedirect.com/science/article/pii/S0377221705003188> [last accessed 2021-07-26], doi:10.1016/j.ejor.2005.03.008.
- [78] Adafruit Industries. Adafruit SGP30 Air Quality Sensor Breakout - VOC and eCO2. URL: <https://www.adafruit.com/product/3709> [last accessed 2021-07-27].
- [79] 3.1. WAN-TCP tunneling over DDS — Integration Service 3.1.0 documentation. URL: https://integration-service.docs.eprosima.com/en/latest/examples/wan_communication/wan_tcp_dds.html [last accessed 2021-08-23].