



Università degli Studi di Catania  
Dept. of Mathematics and Computer Science  
PhD course in Computer Science

---

Fabio D'Urso

**From the concept to the real-world implementation:  
design, simulation and deployment of multi-UAV  
autonomous control algorithms**

---

Doctoral Dissertation

---

Supervisor  
**Prof. C. Santoro**

---

# Abstract

In recent years usage of small remotely piloted aircraft (also known as drones or UAVs, Unmanned Aerial Vehicles) has been spreading, usually quadrotors or hexarotors. They are professionally employed in a multitude of fields: entertainment, art and technical applications, such as photographic surveys (aerial photogrammetry) or surveys with specialised sensors (thermographic inspections, hydrogeological monitoring, precision agriculture). In most cases, these tasks are still controlled manually by the pilot. In this work, we aimed at building a framework to simplify autonomous UAV programming, modelled after real needs, learnt through direct experience in translating research ideas into programs used on the field. The proposed framework follows the software development lifecycle from the initial prototype to the final real-world deployment, including the important steps of prototype validation and simulation of the final program. We drew special emphasis on two aspects: one is the development of multi-robot algorithms, in which a group of UAVs cooperates to achieve a common goal; the other is the need for a realistic simulation environment, in terms of modelling both physics and UAV's control software, that has to expose the same API to software being validated as a real drone, so that it can then be executed on the field with no need for adaptation. Therefore, throughout this work, three heterogeneous case studies will be presented: a decentralised and fault-tolerant multi-UAV algorithm for area coverage; a single-UAV algorithm for landing on a moving target, based on computer vision, and a multi-UAV algorithm to locate and pop balloons, also based on computer vision and with shared airspace. The software architecture, suitable for the implementation and validation of the envisioned algorithms, will be defined and, lastly, we will present the experimental results that we obtained for each of them. In particular, the last two problems were the subject of the MBZIRC (Mohamed Bin Zayed International Robotics Challenge) held in Abu Dhabi, in which the University of Catania ranked fourth in 2017 (out of 24 international teams) and ninth in 2020 (out of 22), using the framework proposed in this thesis.

---

# Sommario

Negli ultimi anni si sta diffondendo l'utilizzo di piccoli aeromobili a pilotaggio remoto (noti anche come droni o UAV, Unmanned Aerial Vehicles), solitamente quadricotteri oppure esarotteri. Essi vengono impiegati professionalmente in una moltitudine di campi: intrattenimento, arte ed applicazioni tecniche, come ad esempio rilievi fotografici (fotogrammetria aerea) o con sensori specializzati (ispezioni termografiche, monitoraggio idrogeologico, agricoltura di precisione). Nella maggior parte dei casi, queste operazioni vengono ancora gestite manualmente dal pilota. In questo lavoro ci si è posti come obiettivo la realizzazione di un framework che semplifichi la programmazione di UAV autonomi, modellato sulle basi di esigenze reali, apprese mediante esperienza diretta nella traduzione di idee di ricerca in programmi utilizzati sul campo. Il framework proposto segue il ciclo di sviluppo del software dal prototipo iniziale fino alla messa in campo finale, passando per le importanti fasi di validazione del prototipo e di test in simulazione del programma finale. Su due punti si è posta particolare enfasi: il primo è lo sviluppo di algoritmi multi-robot, in cui un gruppo di UAV coopera per la realizzazione di un obiettivo comune; il secondo è la necessità di un ambiente di simulazione realistico, sia nella modellazione del sistema fisico che nel software di controllo degli UAV, che offra al software da validare le stesse API che offrirebbe un drone reale, in modo da poterlo poi eseguire sul campo senza la necessità di adattamenti. Nel corso del lavoro, dunque, verranno presentati tre casi di studio con caratteristiche eterogenee: un algoritmo multi-UAV decentralizzato e tollerante ai guasti per la copertura di un'area; un algoritmo per singolo UAV per l'atterraggio su un obiettivo in movimento, basato su visione artificiale, ed un algoritmo multi-UAV per l'individuazione e lo scoppio di palloncini, anch'esso basato su visione artificiale e con spazio aereo condiviso. Verrà poi definita l'architettura software, adatta per l'implementazione e la validazione degli algoritmi ideati, ed, infine, verranno presentati i risultati sperimentali ottenuti per ciascuno di essi. In particolare, gli ultimi due problemi sono stati oggetto della competizione di robotica internazionale MBZIRC (Mohamed Bin Zayed International Robotics Challenge) svoltasi ad Abu Dhabi, nella quale l'Università di Catania ha conseguito il quarto posto nel 2017 (su 24 squadre internazionali) ed il nono posto nel 2020 (su 22), utilizzando il framework proposto in questa tesi.

---

# Contents

<b>Contents</b>	<b>iv</b>
<b>List of Figures</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Scope of the thesis . . . . .	2
1.2 Organisation of the thesis . . . . .	2
<b>2 Background and related work</b>	<b>4</b>
2.1 Software architectures and wireless networks for groups of robots . .	4
2.2 Multirotor simulation . . . . .	8
2.3 Area Coverage and flocking . . . . .	9
<b>3 A flocking algorithm for the Area Coverage problem</b>	<b>12</b>
3.1 Problem definition . . . . .	13
3.2 The proposed algorithm . . . . .	14
3.3 Flock formation and overlay network . . . . .	16
3.4 Distributed aggregation query . . . . .	20
3.5 Path planning and execution . . . . .	23
3.6 Acquiring and transmitting data to a Ground Control Station . . .	25

<b>4</b>	<b>Modeling a flock of quadrotors</b>	<b>27</b>
4.1	Control loop of a quadrotor . . . . .	28
4.2	A lightweight ad hoc simulator . . . . .	31
4.3	Simulation results . . . . .	33
<b>5</b>	<b>Combining heterogeneous tools for realistic UAV simulation</b>	<b>39</b>
5.1	Co-simulation of physics and networking . . . . .	40
5.2	Flight stack architecture . . . . .	41
5.3	The gzuav environment . . . . .	42
<b>6</b>	<b>A software architecture for UAV applications</b>	<b>47</b>
6.1	Onboard software . . . . .	48
6.2	UAV-to-UAV protocol . . . . .	53
6.3	Ground Control Station protocol . . . . .	55
6.4	Tuning Computer Vision Algorithms . . . . .	58
6.5	Simulation . . . . .	60
<b>7</b>	<b>MBZIRC 2017 and 2020</b>	<b>62</b>
7.1	Landing on a moving vehicle (MBZIRC 2017) . . . . .	63
7.2	Popping balloons with two cooperating UAVs (MBZIRC 2020) . . . . .	68

CONTENTS	vi
<b>8 Final remarks</b>	<b>75</b>
8.1 Limitations and open issues . . . . .	76
8.2 Conclusion . . . . .	78
<b>Bibliography</b>	<b>79</b>

---

## List of Figures

2.1	Hidden and exposed node problems . . . . .	7
3.1	Shape and axes of an area to be monitored . . . . .	13
3.2	Sensor area representation . . . . .	13
3.3	Suboptimal formations . . . . .	15
3.4	Optimal formations . . . . .	15
3.5	Example of our algorithm's R2 and R3x in action . . . . .	16
3.6	Representation of area covered by a picture . . . . .	20
3.7	Union of a set of records in a stripe . . . . .	21
3.8	Aggregation query example . . . . .	23
3.9	Path planning . . . . .	24
3.10	Sensor positioning . . . . .	26
4.1	Motor layout and coordinate system . . . . .	28
4.2	Acrobatic mode . . . . .	30
4.3	Manual mode . . . . .	30
4.4	Position mode . . . . .	31
4.5	Screenshot of a flock in the simulator . . . . .	32
4.6	Finite State Machine that implements the UAV's behavior . . . . .	33

4.7	Total mission time and average energy consumption . . . . .	35
4.8	Overcoverage Distribution . . . . .	37
4.9	Time spent in each state . . . . .	37
5.1	Connection diagram of a quadcopter . . . . .	41
5.2	Interactions among components for each simulation step . . . . .	44
5.3	Connections among components in a two-UAV simulation . . . . .	44
5.4	Interactions between high-level logic UAV processes and ns-3 . . . . .	44
5.5	Architecture of gzuavchannel in a distributed environment . . . . .	45
6.1	Software architecture of an autonomous UAV . . . . .	49
6.2	GPS-based TDMA slot assignment . . . . .	55
6.3	Structure of a UDP packet . . . . .	57
6.4	VLOG tools can replay and receive live video from UAVs . . . . .	59
6.5	The overall “VLOG” architecture . . . . .	60
7.1	Arena and target specifications . . . . .	63
7.2	3D printed landing gear . . . . .	63
7.3	Customised DJI S900 for MBZIRC 2017 . . . . .	64
7.4	Vision Module for MBZIRC 2017 . . . . .	65
7.5	Variables used in Kalman filter . . . . .	65
7.6	Finite State Machine for MBZIRC 2017 . . . . .	66
7.7	Funnel-like descent volume . . . . .	67
7.8	Successful landing at the MBZIRC 2017 event . . . . .	68



7.9	Elements of MBZIRC 2020 Challenge 1 . . . . .	69
7.10	Our system to pierce balloons . . . . .	69
7.11	MBZIRC 2020 system simulation in the <i>gz UAV</i> environment . . . . .	70
7.12	Using Circle Hough Transform to refine the result . . . . .	71
7.13	Strategy FSM for popping the balloons . . . . .	73
7.14	A balloon popped during Competition Day 1 of the MBZIRC event	74

---

# 1

## Introduction

In recent years, progress in the miniaturisation of electronics and sensors has made it very cheap to make a robot fly. Unmanned aerial vehicles (UAV), and multirotors in particular, are very easy to design and assemble using off-the-shelf components (frame, motors, propellers, battery, and control board), even without specific knowledge in mechanics and aerodynamics. Being such a powerful and cheap type of machines, they have quickly become widespread for several professional and recreational tasks.

Nonetheless, the current usage of multirotors is still “primitive” from a robotics point of view. While equipped with GPS, wireless capabilities, general-purpose on-board CPUs, and sometimes powerful GPUs, there is still a huge margin of automation because they are rarely fully autonomous: they are usually flown manually by human operators or, in the best cases, they just follow a preset path. Furthermore, several classes of problems are inherently parallelisable and could be resolved much more efficiently if multirotors were programmed to cooperate. For instance, one of the most widely known applications of multirotors is aerial photography. Apart from artistic applications, in which human supervision is inherently necessary, there are several technical applications in which the tasks to be performed are simple and repetitive. One of such applications is aerial photogrammetry, which is rapidly becoming a necessity for precision agriculture [1, 2], maintenance of large-scale plants (such as renewable energy production facilities [3, 4]), disaster response and monitoring of hydrogeological instability [5]. These different scopes have an important aspect in common: they all involve taking many pictures of a static environment, an intrinsically parallelisable task.

When we start to think about automating and parallelising this kind of tasks, we find a lot of research opportunities: to begin with, how can we control groups

of robots and multirotors in particular? How can we deal with hardware failures which, when several robots are at work at the same time, have a non-negligible probability? And what techniques and tools can we use to design, prototype, develop and validate software controlling such systems?

## 1.1 Scope of the thesis

In the first part of this work, aerial photogrammetry is used as a case study. A decentralised and fault-tolerant algorithm is proposed and validated with results in a simplified simulated environment. Starting from that, the next step would have been to put it in practice with a flock of real multicopters. Unfortunately, due to the pandemic, we could not achieve this goal in time. However, while in the process of implementing the proposed algorithm on real robots, we ended up defining a robust general-purpose architecture for UAV applications and a flexible simulation environment. Even if we could not complete the implementation of the intended algorithm in time, such architecture was extensively employed in two side projects: the two editions of an important international robotics competition, in which, also thanks to such architecture, our autonomous multicopters ranked very well. Furthermore, the autonomous tasks that they had to achieve in these competitions entailed physical interaction with a dynamic environment, a feature that had not been planned in the initial design of the architecture but, nonetheless, was added effortlessly thanks to the architecture's flexibility.

Real-world performance of the initial algorithm remained an open point, but the tools that we designed proved to be effective even in different scenarios.

## 1.2 Organisation of the thesis

This dissertation is organised as follows.

In chapter 3, we present a multi-robot algorithm to solve the Single Area Coverage problem that can be used to speed up tasks such as aerial inspection and monitoring. In chapter 4 we validate such algorithm in an ad-hoc simulation environment. In particular, we develop a virtual quadrotor model and a 3D flocking simulator in which high-level behaviours can quickly be evaluated. Such models are then used for a performance analysis of the proposed algorithm.

In chapter 5, we analyse the problem of simulating flocks of multi-robots from a more general point of view: we combine quadrotor physics with wireless networking simulation, obtaining an environment that makes it possible to write and debug full-fledged autonomous programs, implementing any algorithm and not just the previously presented one. An important feature of the new simulation environment is that, once a piece of software is tested in it, it can run, without further modifications, on a real UAV platform too.

In chapter 6, we move the focus from the simulation environment to UAV programs themselves. We present a modular, robust, and general-purpose architecture for UAV applications. A technique to implement a collision-free decentralised wireless network, suitable for the requirements of the flocking algorithm is presented too.

Chapter 7 presents the two side projects, which are actually two different UAV problems that were solved within the proposed framework: landing on a moving vehicle and cooperatively locating and popping balloons. The proposed solutions were implemented and validated at the Mohamed Bin Zayed International Robotics Challenge (MBZIRC) in Abu Dhabi, where our university's team participated in both editions, and ranked fourth (2017) and ninth (2020).

Chapter 8 summarises the scientific results that were obtained, some open points and possible future research directions concerning the topics addressed in this dissertation. Lastly, it presents the concluding remarks.

# 2

## Background and related work

### 2.1 Software architectures and wireless networks for groups of robots

Robot control software is full of implementation details: it can become very closely-coupled to the hardware it will run on, and it must implement appropriate interfaces/drivers for the many peripherals that a robot is composed of. Conversely, there are a lot of software procedures, from basic “textbook” algorithms to complex state-of-the-art computer vision pipelines, that can be seen as general-purpose building blocks.

Robot Operating System (ROS) [6] is a middleware/framework for Linux that addresses these issues by promoting the usage of loosely-coupled software components: every component (called “Node” in ROS) runs in a separate process. Nodes can communicate through a publisher/subscribe paradigm, in which ROS itself is the broker; and they can be implemented and distributed independently from each other: they do not even need to be programmed in the same language, because all programming languages supported by ROS (such as C++ and Python) can be used for any Node. Nodes do not even necessarily correspond to programs running on the same host: if a robot contains several interconnected Linux boards, ROS can transparently route messages among nodes running on them. This flexibility, of course, comes at a price:

- Increased inter-node communication latency, because the publish/subscribe paradigm is much slower than a direct call (this is especially true considering the fact that, by default, ROS transmits messages over TCP sockets);

- Loss of flexibility, because the datatypes of all messages need to be formally defined in a ROS domain-specific language: the ROS framework, at compile-time, will parse those definition files and generate appropriate serialisation/deserialisation routines. This mandatory serialisation pass (that is necessary in order to support message-passing among heterogeneous boards) will also add latency, compared to a plain *send* call with raw data in a local process;
- Loss of control: as a result of component reuse, third-party nodes behave as “black boxes”. Application-specific customization or optimization may not be possible unless the node has explicit support for it written in advance.

Pre-made ROS nodes are grouped into “ROS packages”, which are available for many purposes. For instance, interfacing with open-source multicopter controllers is possible with the *mavros* [7] package. It should be noted that, even though ROS is designed for robotics applications, it offers no real-time guarantees because it runs on top of Linux, which is not a real-time operating system.

Some researchers use ROS as the foundation for higher-level frameworks. For instance, ROSBuzz [8, 9] bridges *mavros* with Buzz [10], a programming language specifically targeted at robot swarm programming.

Communication among different robots is also an important topic. Communications involving multirotors can be categorized into three categories [11]:

- Air-to-Air (A2A): Messages transmitted by a flying robot to another flying robot;
- Air-to-Ground (A2G) and Ground-to-Air (G2A): Messages exchanged between a flying robot and a base station on the ground.

Wireless networks involving multirotors have special characteristics, such as a very dynamic topology (in other words, its nodes move very frequently) and, often, a diameter (in metres) much bigger than a single node’s transmission range. This class of networks is called Flying Ad-hoc Network (FANET). A survey on FANET technologies and performance can be found in [11] and [12].

If Internet Protocol (IP) connectivity among robots is available (as required by ROS), in the case of flying robots the underlying technology will almost inevitably be some variant of IEEE 802.11 [13], also known as “WiFi”. In its most

widespread setup (called “infrastructure” mode), this type of network is based on an access point (AP) node that periodically transmits a *beacon* packet, which client nodes (STA) listen for in order to synchronise their state. This type of network is widely supported by off-the-shelf hardware and it offers very good throughput. Its downsides are the relatively low range (in the order of a few hundreds of metres at best) and the need for a centralised AP, which represents a single point of failure: indeed, if the AP fails, or simply goes out of range, communication becomes impossible. A slightly different 802.11 variant is the Independent Basic Service Set (IBSS, also known as “ad-hoc mode”), in which there is no designated AP, but one of the peers dynamically takes the responsibility to generate beacons. Lastly, another interesting variant is 802.11s (“mesh”): similarly to IBSS, it is also a peer-to-peer network, but it adds dynamic routing in order to work even when not all nodes are within other nodes’ transmission range.

802.15.4 [14] is another interesting type of wireless network. It focuses on low power consumption and its throughput and range are smaller than 802.11. However, IEEE 802.15.4 devices are usually much simpler and customisable than those implementing 802.11. For instance, it is often possible to customise channel contention policies (e.g. tune the thresholds for clear channel assessment for CSMA-CA or disable it completely for immediate transmission). Furthermore, with the addition of Ultra-wide Band support (802.15.4-UWB), some devices also implement the “ranging” function, which can compute the distance (in metres) between the transmitter and the receiver by measuring the radio waves propagation time. Therefore, they can be used to build positioning systems too [15, 16, 17]. IEEE 802.15.4 also has provisions for selecting some devices as “coordinators” to emit beacons, assign Guaranteed Time Slots (GTS) and synchronise other devices; higher-level protocols, such as Zigbee [18], build upon this feature. However, the presence of a coordinator is not mandatory in an 802.15.4 network.

IEEE 802.11 and IEEE 802.15.4 can be used for all types of transmissions (A2A, A2G and G2A) and work on an unlicensed band (i.e. they do not require a radio license or dedicated band spectrum). Another common aspect is that they are both greatly affected by the hidden and exposed node problems:

- A hidden node problem (fig. 2.1a) occurs when a node is transmitting and another node, out of its transmission range, determines that the channel is free and starts transmitting too. As a consequence of this, a collision will occur and both messages will be corrupted<sup>1</sup>.

---

<sup>1</sup>IEEE 802.11 has a built-in mechanism to mitigate the hidden node problem, called Request-to-send/Clear-to-send (RTS/CTS) [19].

- An exposed node problem (fig. 2.1b) occurs when a node detects that the channel is in use and refrains from transmitting, even if doing that would not result in a collision. As a consequence of this, throughput will be suboptimal.

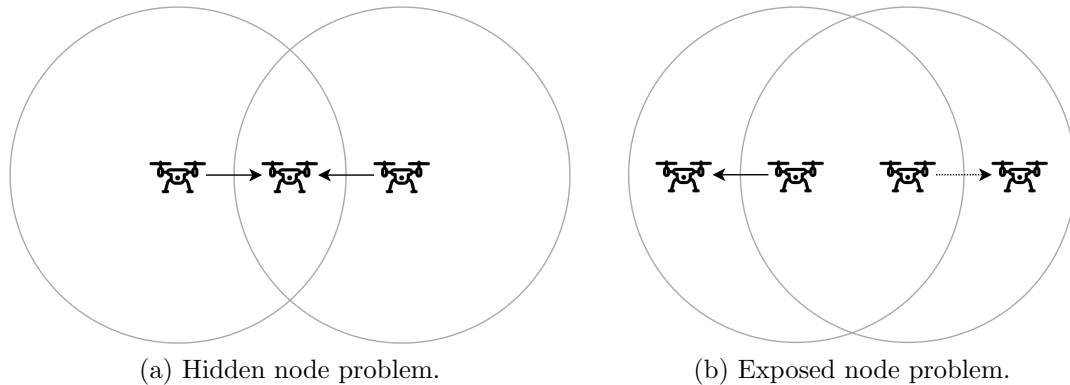


Figure 2.1: Hidden and exposed node problems.

Another possibility for FANETs is to rely on the cellular network. This type of networks operate on licensed bands and are administered by telecommunication companies. In this case, flying robots simply register to the network (as if they were cellular phones) and exchange data traffic with the company’s base station, which possibly routes it on the Internet. Depending on the network technology, the technical name of the elements of the network differs: for instance, in LTE, mobile nodes are called “User Equipment” (UE) and the base station is called “NodeB”. The mobile data network cannot be used for direct A2A messages, because all messages have to go through the NodeB. Because of this, FANETs based on the cellular network have a single inevitable point of failure: the cell’s NodeB. With the advent of 5G networks and the so-called “Tactile Internet”, the latency of end-to-end messages will be reduced to approximately one millisecond [20].

The Global Positioning System (GPS) is not strictly a communication network, but it is worth mentioning too: satellites continuously emit a signal containing the current date and time (according to their precise onboard atomic clock) and their position. GPS receivers monitor such signals and, by comparing relative arrival timestamps according to a local clock, can compute their own position. Basic GPS receivers have a 15-metre precision. Through the usage of differential GPS techniques, such as Real-Time Kinematic (RTK), it can be improved up to less than one centimetre [21]. An interesting note is that, as well as the receiver position, GPS receivers can determine the current time and data with a precision in



the order of tenths of nanoseconds. In addition to GPS, which was the first constellation of satellites for this purpose (launched by the USA), other nations have developed similar systems, such as Galileo (European Union), GLONASS (Russia) and BeiDou (China). Modern receivers can use, at the same time, satellites from different constellations.

## 2.2 Multirotor simulation

There are several software environments to simulate multirotors. Some are geared towards end users only and do not expose an API (such as DJI’s own Flight Simulator [22]), others do not emulate the API that a programmable multirotor would offer in the real world (such as ARGoS [23]). Furthermore, commercial flight control units are usually closed source. On the other end, other simulation environments are specifically designed for Software-In-The-Loop (SITL) simulation of open-source control software stacks (i.e. they can run the same control loops, in simulation, that would run in a real multirotor).

The most well-known example is probably Gazebo [24]. It is widely used to simulate several types of robotic platforms, including UAVs such as multirotors, thanks to a greatly customizable plugin-based system. Interestingly, there are ready-made plugins designed to integrate it with the two biggest open-source flight control software projects: PX4 [25] and ArduPilot’s project ArduCopter [26]. Both projects implement a SITL interface, which has the advantage of offering the same API that a real multirotor platform would offer, which is an essential requirement for the final validation of autonomous flight code before deployment in the real world.

Regarding the physics engine, one of the advantages of using a general-purpose robotics simulator is that it is easy to integrate other types of objects (e.g. balls, pliers, onboard cameras, auxiliary robots such as rovers, ...) in the simulation. Gazebo has been designed aiming at the realistic simulation of sensors and objects in the environment. At its core, it runs a physics simulation using one of the supported physics engines (ODE [27], Bullet [28], Simbody [29] and DART [30]). Models are defined in terms of *rigid bodies* (with optional attached sensors) and *joints* by means of XML files. A “world” XML file acts as the root and other XML files can be recursively included. Gazebo also offers a flexible plugin interface, which makes it possible to program custom behaviours and sensors: plugins are compiled as Linux shared libraries (.so files) from C++ source code. Plugins can interact with the simulation using Gazebo’s API (based on a *publisher-subscriber*

paradigm), but also with the host operating system using its native API (such as IPC mechanisms and sockets). As we will see in the next section, *gz* heavily uses the latter to integrate the other components. Gazebo maintains a separation between its GUI (*gzclient* program) and the simulation itself (*gzserver* program), which makes it possible to run simulations headless. Simulations can be run in real-time (at the same rate as the host's clock) or faster/slower. Lastly, thanks to its widespread adoption, Gazebo offers a wide library of open-source ready-made models and plugins: for instance, the quadrotor model that we use is the built-in *iris\_with\_standoffs* model, along with the built-in *LiftDragPlugin* and a custom *GzUavPlugin* (derived from *ArduCopterPlugin*).

As for the flight stack, PX4 and ArduCopter are the most widespread ones. They can both run in SITL mode in combination with Gazebo. ArduCopter implements control loops for several types of multicopters (such as quadrotors and hexarotors); other ArduPilot subprojects support rovers, boats, helicopters and fixed-wing planes. Furthermore, ArduPilot is easier to run at non-real-time speed, because its internal SITL architecture is less tied to the host's clock than PX4.

Other examples of SITL-capable simulators are jMAVSim [31] for PX4, and X-Plane [32], JSBSIM [33] and AirSim [34] for ArduCopter.

SITL-based simulation environments are powerful, flexible and realistic. However, developing for these environments usually takes the same programming effort as programming the final system, if not more. Furthermore, these environments are very heavyweight and slow. Therefore, they are not very useful during the initial prototyping/validation of a new flocking algorithm, when changes are frequent and the developer needs a quick response. As we will see in chapter 4, this is the reason why we developed a new lightweight simulation framework.

## 2.3 Area Coverage and flocking

Aerial photogrammetry, which we adopted as a case study, is a particular instance of the Area Coverage problem. In detail, Area Coverage problems can be divided into two subcategories [35]:

- **Single Area Coverage** aims to cover each point of interest exactly once. If a given point is not covered, it is an issue in that the result will be incomplete; redundant coverage of a point is an issue too because it is just wasted effort with no additional value.

- **Repeated Area Coverage** aims to cover each point of interest at least once, but possibly more than once, during the same mission, according to mission-dependent factors, such as the importance of each point and predetermined frequency of visits.

The algorithm that will be described in chapter 3 falls into the former category. Therefore, it will have two goals: i) to ensure that the whole target area is scanned and ii) to minimise mission time; in other words, any given point must be visited at least once, but ideally no more than that.

In an Area Coverage problem, it is important to define how the area is subdivided. A number of approaches exist, such as Exact-cell [36] and Approximate-cell [37] decompositions. In the Exact-cell decomposition, the target area is subdivided into stripes along the sweeping direction, forming a set of triangular or trapezoidal cells around obstacles. The Approximate-cell decomposition, instead, recursively subdivides the target area in a quadtree-like fashion. As we will see later, our algorithm assumes that no obstacles are present and then uses a variant of the Approximate-cell decomposition to keep track of the points that have already been visited.

Other important topics in multi-robot algorithms are fault-tolerance and network connectivity. If the number of agents is small, it is reasonable to assume that agents are reliable and direct wireless communication is always possible. However, in large multi-robot systems, the probability of failures increases dramatically. Furthermore, in large UAV flocks, which can be hundreds of meters large, full direct connectivity among agents is impossible. In these cases, an overlay network such as in [38] is necessary. Furthermore, it is necessary to avoid formation shapes that cause excessive distance between two agents, which would result in network partitions [39].

Task allocation techniques are often market-based or bio-inspired. In market-based approaches such as [40] and [41], tasks are assigned through *bids* placed by each robot, according to a cost/utility function, and administered by an *auctioneer*. This type of algorithms offer built-in robustness to agent failures: if a bidder fails, its task will be reassigned to the next best bidder. Furthermore, if the robots inform other team members about their tasks, the auctioneer too can be replaced, should it fail.

Bio-inspired works, including the one that will be presented in this dissertation, are often based on the three flocking rules invented by Craig Reynolds [42],

namely<sup>2</sup>:

**R1** *Separation*: if too close to another flock member, turn away in the opposite direction;

**R2** *Alignment*: turn towards the average orientation of known flock members;

**R3** *Cohesion*: turn towards the average position of known flock members.

An interesting work about flocking is [43], in which the authors obtained formations of different shapes in a fully decentralised way.

---

<sup>2</sup>R1 is mutually exclusive to R2 and R3, which are themselves seemingly contradictory. In fact, the “steering forces” from R2 and R3 are meant to be linearly combined.

# 3

## A flocking algorithm for the Area Coverage problem

Aerial photogrammetry is widely used for many applications, such as mapping, inspection, 3D modelling and precision agriculture. However, images are usually acquired by flying a single UAV<sup>3</sup> over a predetermined path [44]. This approach is unsuitable for large-scale inspections, which are still usually performed using aeroplanes or helicopters. Such large manned vehicles present several issues, such as high mission cost, the need for an experienced pilot and the inherent risks in flying over dangerous areas (e.g. disaster response missions).

The main obstacle preventing the use of UAVs in large-scale missions is the maximum flight time, imposed by battery capacity, which is usually somewhere between 10-20 minutes, due to physical constraints (bigger batteries imply more weight and, therefore, more energy consumption). Furthermore, even if there were no battery capacity issues at all, a single UAV would probably take too long to complete the mission.

The approach we propose in this chapter is to use *several* UAVs, each taking care of a fraction of the total area, thus completing the mission in a fraction of the time that a single UAV would have required. The proposed algorithm mimics the way birds fly and organizes the UAVs in a *flock* [45, 46].

---

<sup>3</sup>If taken literally, the term “UAV” can refer to any type of unmanned aerial vehicles (including even fixed-wing military planes operated remotely), in addition to the relatively-small toy/professional vehicles with multiple rotors that have gained traction in the last years (which are technically called “multirotors” or “multicopters”). Even if formally incorrect, for the sake of brevity, all occurrences of the term “UAV” in this dissertation are to be intended as synonyms of “multirotor”/“multicopter”.

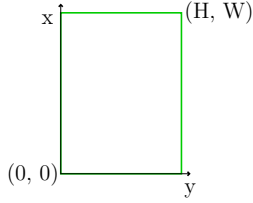


Figure 3.1: Shape and axes of an area to be monitored.

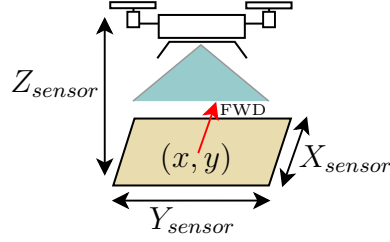


Figure 3.2: Sensor area representation.

The key aspect of the proposed solution is the decentralised control approach with built-in fault tolerance: any subset of UAVs is allowed to fail. In case of failures, the mission will simply take longer, but the acquired data at the end of the mission will still be complete.

The proposed solution can be generalised not only to aerial photogrammetry but also to all those types of inspections where a sensor must scan a predetermined area (e.g. thermal inspection of photovoltaic plants, LIDAR-based elevation mapping and so on).

### 3.1 Problem definition

Let us assume that the area to be monitored is a  $H \times W$  rectangle<sup>4</sup>, with  $W \leq H$ . Let us also assume that one of its corners is the origin of a 3D Cartesian coordinate system oriented towards the  $H$  axis (fig. 3.1).

We also assume that the sensor (e.g. camera) is capable of acquiring data corresponding to a  $X_{sensor} \times Y_{sensor}$  rectangular surface, if flown at  $Z_{sensor}$  altitude and oriented like the  $x$  axis<sup>5</sup> (fig. 3.2). Then, each acquisition (e.g. image) can be identified with the  $(x, y)$  coordinates of the UAV at the moment it was taken and its rectangular outline is identified by:

$$\begin{aligned}
 & (X1, Y1, X2, Y2) \\
 \text{with } & X1 \equiv x - \frac{X_{sensor}}{2} \quad X2 \equiv x + \frac{X_{sensor}}{2} \\
 & Y1 \equiv y - \frac{Y_{sensor}}{2} \quad Y2 \equiv y + \frac{Y_{sensor}}{2}
 \end{aligned} \tag{3.1}$$

<sup>4</sup>If it is not the case, we can always find the smallest rectangle that encloses it.

<sup>5</sup>As we will see later, the sensor will be turned off when the altitude is different from  $Z_{sensor}$  or the orientation is different from the  $H$  axis.

We assume that each UAV is equipped with a wireless transceiver, able to transmit/receive messages to/from UAVs within direct wireless range. We do not further characterise the size and range of the radiation pattern, but we do assume that if an agent is within reception range (i.e. we can receive its messages) then it is also within transmission range (i.e. it can receive our messages). We assume that no collisions occur and physical propagation delay between the transmitter and the receiver is negligible. We also assume that any UAV can fail and, in such a case, the contents of its onboard storage are irrecoverable.

The goal of the mission is to ensure that, in the end, every point within the area to be inspected has been covered by at least one acquisition (e.q. 3.2), while minimising mission time.

$$(0, 0, H, W) \subseteq \bigcup (X1, Y1, X2, Y2) \quad (3.2)$$

## 3.2 The proposed algorithm

One of the defining aspects of flocking algorithms is the shape of the formation. If we only apply Reynold’s three rules (separation, alignment and cohesion; see section 2.3), a flock will emerge, but it will not have a predictable orientation or direction, and large flocks tend to split into separate partitions [47]. Furthermore, such basic flocking rules do not take into account the characteristics of the sensor, resulting in far-from-ideal area coverage (fig. 3.3a): in this formation, each agent moves in a different direction, there is no guarantee that every point will be covered at least once and, conversely, many points will be covered more than once. Even if we could somehow make all agents agree on a common heading and direction, that would not solve under/overcoverage issues (as fig. 3.3b shows, agents that are too far from the flock would lead to “holes” in the resulting coverage, and agents that are behind other agents would basically only over-cover points already covered by the first agents). Therefore we need to obtain a *compact* formation in which agents do not stay behind other agents.

Optimal formations, satisfying the above constraints, are those in fig. 3.4a and 3.4b, but the simplest good one is a “line formation”, shown in fig. 3.4c: this type of formation keeps agents tight to each other while also imposing a fixed distance, which virtually removes the possibility of missing or overlapping coverage. It is possible to augment the basic three rules to impose a common goal and make the flock behave more coherently. Our line formation can indeed be obtained by

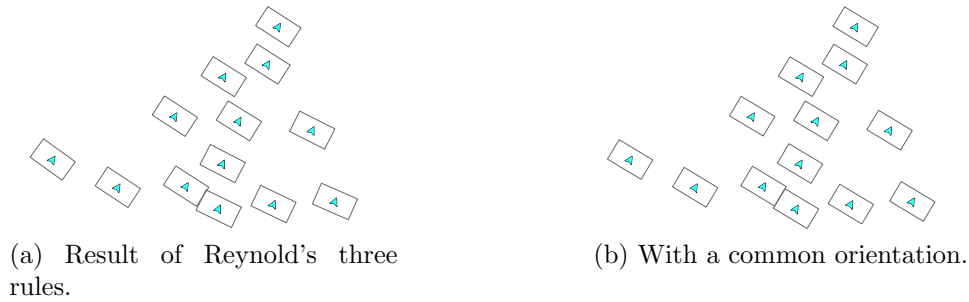


Figure 3.3: Suboptimal formations.

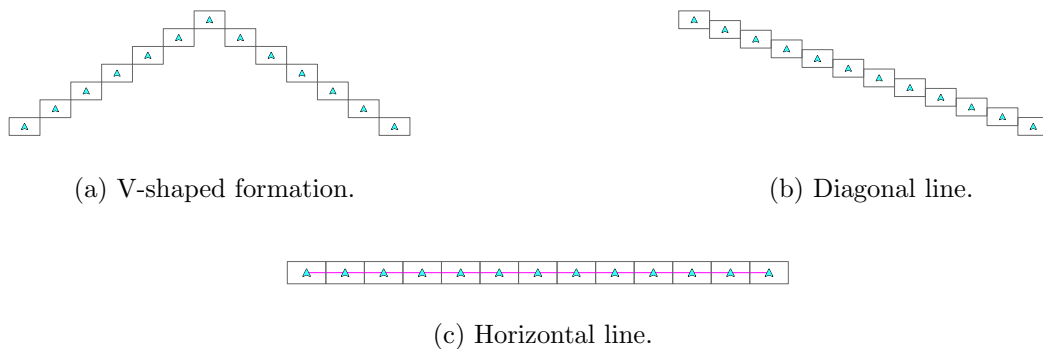


Figure 3.4: Optimal formations.

modifying the basic three rules, as will be detailed later.

To impose a common goal upon the flock we also defined a criterion to select a *leader* agent (on which all agents can quickly converge). The leader follows its own rules, different from the other agents' three flocking rules. We have slightly modified their rules so that they explicitly follow the leader's movements, instead of the generic flock average. The leader's position is known to all agents almost in real-time, and the leader's local  $y$  axis is used as the *flock line*. The new rules for non-leader agents are (fig. 3.5):

**R1** *Separation*: if we are too close<sup>6</sup> to the projection of another flock member, fly away in the opposite direction (without altering the heading and only in

---

<sup>6</sup>The threshold determines the desired distance between consecutive agents, which ideally corresponds to  $Y_{sensor}$ . In fact, it is set to a slightly smaller value (e.g. 25% smaller) to avoid introducing holes in the resulting coverage due to the inevitable control latency and oscillations and, at the same time, offer some common keypoints for post-processing (e.g. image stitching) of data acquired by adjacent agents.



parallel to the flock line, i.e. without going forward/backwards with reference to the leader);

**R2** *Alignment*: copy the leader's heading;

**R3** *Cohesion*: two independent rules:

**R3x** move towards our projection on the flock line unless that direction is obstructed by another agent;

**R3y** move towards the leader in parallel to the flock line.

Rules R2 and R3x are always applied, rule R1 and R3y are mutually exclusive.

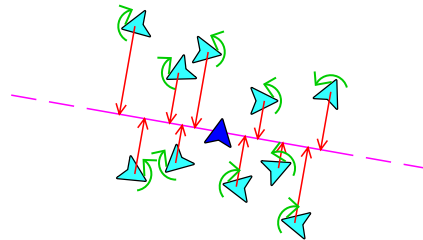


Figure 3.5: Example of our algorithm's R2 and R3x in action.

### 3.3 Flock formation and overlay network

As will be seen later, apart from take-off and landing, all phases of the mission take place at a fixed altitude  $Z_{sensor}$  above ground level. Therefore, for the rest of this chapter, the  $z$  coordinate will be omitted and assumed to be equal to  $Z_{sensor}$ .

In our approach, all agents know the state of all the other agents, albeit with a propagation delay. Each agent maintains a local database of information about the other agents (called *Agents Database*, or ADB). Each record contains:

- The other agent's ID.
- Sequence number,  $x$ ,  $y$  and heading from the latest known message from the other agent.
- Distance between this and the other agent (as the number of hops).
- Age of the previous information (in milliseconds).

Each agent periodically<sup>7</sup> broadcasts the following information:

- A sequence number, incremented for every message emitted by the agent.
- The agent's current  $x$  and  $y$  coordinates.
- The agent's target heading.
- A full copy of the agent's ADB.

When such a message is received by another agent, its contents are *merged* with the local ADB (alg. 1). In particular, for each agent mentioned in the message, the corresponding ADB's record is checked and updated only if its sequence number is older than the received one. Furthermore, every millisecond a local timer triggers the *aging* procedure (alg. 2) in each agent, which compares every record's age with its expected value (which is proportional to the distance), and evicts records about failed agents (i.e. agents that have not recently emitted a newer message). As a consequence of such algorithms, in this gossip-based scheme, an overlay network is created, in which information and failures become quickly known to all the agents.

Thanks to the fact that all agents know about all other agents' existence, a very simple criterion for selecting the leader can be used: the agent with the lowest ID acts as the leader. The leader will know that it is indeed the leader because no agents with a lower ID exist; similarly, non-leader agents will know that they are not the leader, because at least another agent with a lower ID exists. If the leader agent fails, it will disappear from the remaining agents' ADBs and they will quickly converge to a new leader.

Non-leader agents will follow the rules given in section 3.2 and implemented in algorithm 3. The leader follows different rules, that are described in section 3.5. With the algorithm described above, non-leader agents tend to maintain their relative position with respect to the leader. Therefore, with proper movements of "guidance", the leader can direct the whole flock towards uncovered portions of the target area.

---

<sup>7</sup>We found that 8 times per second is a good trade-off between latency and bandwidth requirements.

---

**Algorithm 1** Processing of incoming messages

---

```

function ON_INCOMING_MESSAGE(SenderID, SenderSeq, SenderX,
SenderY, SenderHeading, SenderADB)
  for all r in SenderADB do
    if r.ID  $\neq$  LocalID then
      MERGE(r.ID, r.Seq, r.x, r.y, r.heading, r.distance + 1, r.age)
    end if
  end for
  MERGE(SenderID, SenderSeq, SenderX, SenderY, SenderHeading, 1, 0)
end function
function MERGE(ID, Seq, X, Y, Heading, Distance, Age)
  if ID  $\notin$  LocalADB  $\parallel$  LocalADB[ID].Seq < Seq then
    LocalADB[ID]  $\leftarrow$  (Seq, X, Y, Heading, Distance, Age)
  end if
end function

```

---



---

**Algorithm 2** Aging timer handler

---

```

function ON_AGING_TIMER ▷ Runs every millisecond
  for all r in LocalADB do
    NewAge  $\leftarrow$  r.Age + 1 ▷ Increment by 1 ms
    if NewAge  $\leq$  MAX_ALLOWED_AGE(r.Distance) then
      LocalADB[r.ID].Age  $\leftarrow$  NewAge
    else
      DROP(LocalADB[r.ID])
    end if
  end for
end function

```

---

**Algorithm 3** Flocking rules pseudocode

---

```

function CONTROLLER_NONLEADER(LeaderID, LocalX, LocalY,
  LocalHeading, LocalADB,  $\Delta T$ )
  diffX  $\leftarrow$  LocalADB[LeaderID].x - LocalX
  diffY  $\leftarrow$  LocalADB[LeaderID].y - LocalY
  parallelDirX  $\leftarrow$   $\cos(\text{targetHeading})$ , parallelDirY  $\leftarrow$   $\sin(\text{targetHeading})$ 
  orthoDirX  $\leftarrow$   $\cos(\text{targetHeading} + \pi/2)$ , orthoDirY  $\leftarrow$   $\sin(\text{targetHeading} + \pi/2)$ 
  parallelDiff  $\leftarrow$  parallelDirX * diffX + parallelDirY * diffY
  orthoDiff  $\leftarrow$  orthoDirX * diffX + orthoDirY * diffY
  ▷ Apply R2 (alignment):
    yawTargetSpeed  $\leftarrow$  SATURATE(
       $k_{p\_R2}$  * ANGDIFF(LocalADB[LeaderID].heading, LocalHeading),  $s_{R2}$ )
  ▷ Apply R3x (cohesion along X axis):
    if parallelDiff  $\geq$  0 then
      R3x_dir  $\leftarrow$  targetHeading
    else
      R3x_dir  $\leftarrow$  targetHeading +  $\pi$ 
    end if
    if ISDIRECTIONOBSTRUCTED(LocalX, LocalY, R3x_dir, LocalADB) then
      parallelTargetSpeed  $\leftarrow$  0
    else
      parallelTargetSpeed  $\leftarrow$  PICONTROLLER( $k_{p\_R3x}$ ,  $k_{I\_R3x}$ , parallelDiff,  $s_{R3x}$ )
    end if
  ▷ Apply R1 (separation) or R3y (cohesion along Y axis):
    nearestAgentParallelDist, nearestAgentOrthoDist  $\leftarrow$ 
      DISTANCETONEARESTAGENT(LocalX, LocalY, LocalHeading, LocalADB)
    if  $|\text{nearestAgentOrthoDist}| < D_{R1}$  then
      sep  $\leftarrow$   $c_{p\_R1}$  * nearestAgentParallelDist2 +  $c_{o\_R1}$  *  $[1 - (\frac{\text{nearestAgentOrthoDist}}{D_{R1}})^{d_{R1}}]$ 
      orthoTargetSpeed  $\leftarrow$  SATURATE(sep,  $s_{R1}$ ) * sgn(nearestAgentOrthoDist)
    else
      orthoTargetSpeed  $\leftarrow$  SATURATE( $k_{p\_R3y}$  * ( $-orthoDiff$ ),  $s_{R3y}$ )
    end if
  end function

function ANGDIFF( $\alpha$ ,  $\beta$ ) ▷ Compute  $\alpha - \beta$  and normalise it within  $[-\pi; +\pi]$ 
   $\gamma$   $\leftarrow$   $\alpha - \beta$ 
  if  $|\gamma| > \pi$  then
     $\gamma$   $\leftarrow$   $\gamma - 2\pi * \text{sgn}(\gamma)$ 
  end if
  return  $\gamma$ 
end function

function SATURATE(v, s) ▷ Constrain v within  $[-s; +s]$ 
  return  $\min(s, \max(-s, v))$ 
end function

```

---

### 3.4 Distributed aggregation query

Before describing the leader’s rules of movement, we need to define how the leader gets to know what parts of the area have been covered by the flock, and what parts are still unexplored. Given the decentralised and fault-tolerant nature of the proposed algorithm, we assume that, whenever an agent fails, all of its acquired data is considered lost. In other words, should an agent fail, every part of the area that it had covered reverts back to the unexplored state<sup>8</sup> and must be covered again. Furthermore, should the leader fail, the new leader must be able to acquire the same information about current coverage that the old leader had. With these goals in mind, we designed a distributed algorithm to efficiently query the flock’s overall coverage map.

Let us subdivide the rectangular area to be monitored into a set of stripes. Such stripes are parallel to the  $y$  axis, and their height is approximately equal to  $X_{sensor}$ . All pictures taken by the UAVs will be axis-aligned. Furthermore, camera trigger logic ensures that only pictures fully covering a stripe along the  $x$  axis (fig. 3.6) are saved.

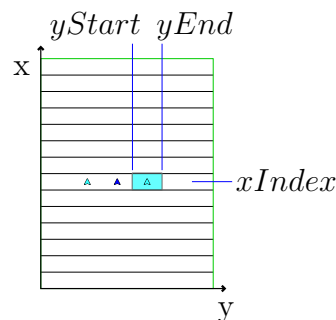


Figure 3.6: Representation of area covered by a picture.

Under the above assumptions, the area covered by a picture can be identified by the  $xIndex$  of the stripe (i.e. the index of the discretised  $x$  coordinate of the rectangular area) and the coordinates of the start and the end of the covered area ( $yStart, yEnd$ ) along the  $y$  axis (with  $yEnd - yStart = Y_{sensor}$ ).

Each agent maintains a local database of the coordinates of the pictures it has taken, called Area Parts Database (APD). Let us see how, in the proposed

<sup>8</sup>Unless another agent happens to have redundantly covered the same area. Note that redundancy is not a desired effect, because the algorithm tries to minimise overall mission time and, thus, repeated coverage.

approach, the leader can efficiently query and aggregate the APDs of all agents in the flock.

An APD is stored as an array of  $N$  lists (with  $N =$  number of stripes, known in advance to all agents). An APD is said to be in *normal form* if each list is such that every pair of consecutive entries has  $yEnd_i < yStart_{i+1}$ . An APD can be normalised by *merging* overlapping pictures within the same stripe as if they were a single wider one and then sorting the resulting elements in increasing  $yStart_i$  order.

It is also possible to aggregate several APDs into a single one, which will represent the union of the covered areas, by concatenating each stripe's data and then applying alg. 4 (whose working at stripe level is shown in fig. 3.7).

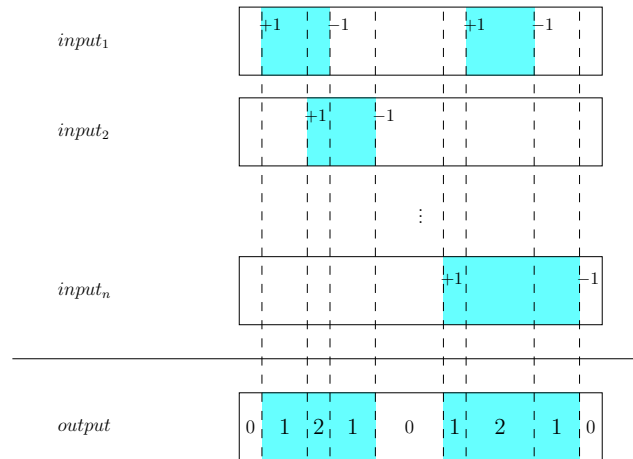


Figure 3.7: Union of a set of records in a stripe.

The leader starts an APD query operation by sending the first message. All the agents work as retransmitters and partial aggregators. Every message includes:

- the ID of the node that started the query (messages with a value different from the current leader's ID are ignored upon reception);
- a unique monotonic query ID;
- an APD containing the current partial result;
- an array of IDs of agents that have responded so far.

---

**Algorithm 4** APD normalisation and union

---

```

function MERGE_APDS(APD1, APD2)
  result  $\leftarrow \emptyset$ 
  for all xIndex do ▷ For each stripe
    tmp  $\leftarrow$  CONCATENATE_LISTS(APD1[xIndex], APD2[xIndex])
    result[xIndex]  $\leftarrow$  NORMALIZE_STRIPE(tmp)
  end for

  return result
end function

function NORMALIZE_STRIPE(listOfPatches)
  discontMap  $\leftarrow \emptyset$  ▷ Initialise to an empty associative array
  for all xStart, xEnd in listOfPatches do ▷ For each patch of covered area
    if xStart  $\in$  discontMap then
      discontMap[xStart]  $\leftarrow$  discontMap[xStart] + 1
    else
      discontMap[xStart]  $\leftarrow$  +1
    end if
    if xEnd  $\in$  discontMap then
      discontMap[xEnd]  $\leftarrow$  discontMap[xEnd] - 1
    else
      discontMap[xEnd]  $\leftarrow$  -1
    end if
  end for

  result  $\leftarrow$  MAKE_EMPTY_LIST()
  sum  $\leftarrow$  0
  for all x in discontMap do
    prevSum  $\leftarrow$  sum
    sum  $\leftarrow$  sum + discontMap[x]
    if prevSum = 0  $\wedge$  sum > 0 then
      xStart  $\leftarrow$  x
    else if prevSum > 0  $\wedge$  sum = 0 then
      xEnd  $\leftarrow$  x
      APPEND_TO_LIST(result, (xStart, xEnd))
    end if
  end for

  return result
end function

```

---

The initial message, sent by the leader, contains the leader's own APD and an array with a single entry (the ID of the leader itself). The agents that receive a message merge the received APD with their own and produce a new aggregated message. Further agents will do the same until, in the end, a message citing all agents in the flock bounces back to the leader, which will find the result of the query in the APD field.

In order to avoid flooding the network and be tolerant to lost messages, each agent keeps retransmitting, at a fixed rate, its last message. As a further optimisation, an extra list of acknowledged agent IDs is appended to all transmitted messages. If all neighbours within wireless range (which are known from the ADB) have acknowledged the latest sent message, the retransmissions stop. Fig. 3.8 shows the messages transmitted at each step for an example graph (edges represent wireless connectivity in an example graph).

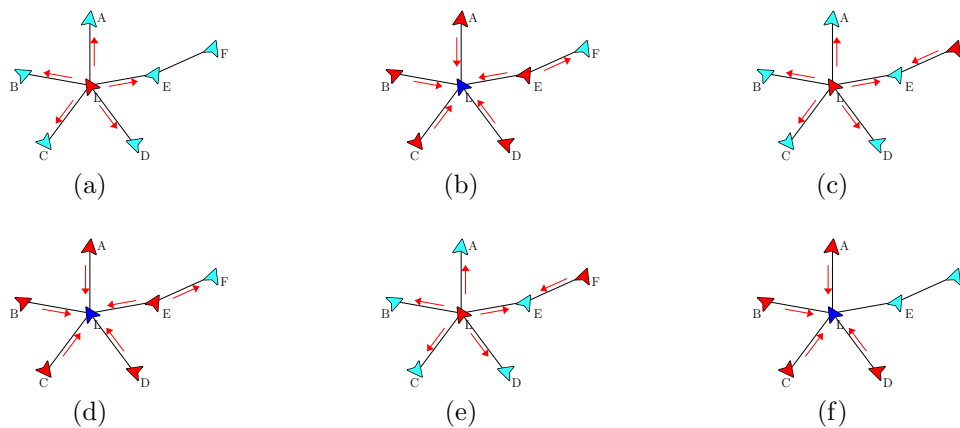


Figure 3.8: Aggregation query example.

### 3.5 Path planning and execution

As soon as the aggregation query returns its result, the leader can proceed with the computation of an optimal path over areas that have not been covered yet.

Given the distributed nature of the flocking algorithm, we decided to avoid partitioning the flock (which would imply loss of connectivity among the partitions) at all costs. Therefore, the flock always flies in tight formation. It is the leader's responsibility to guide it so that area coverage is maximised under these



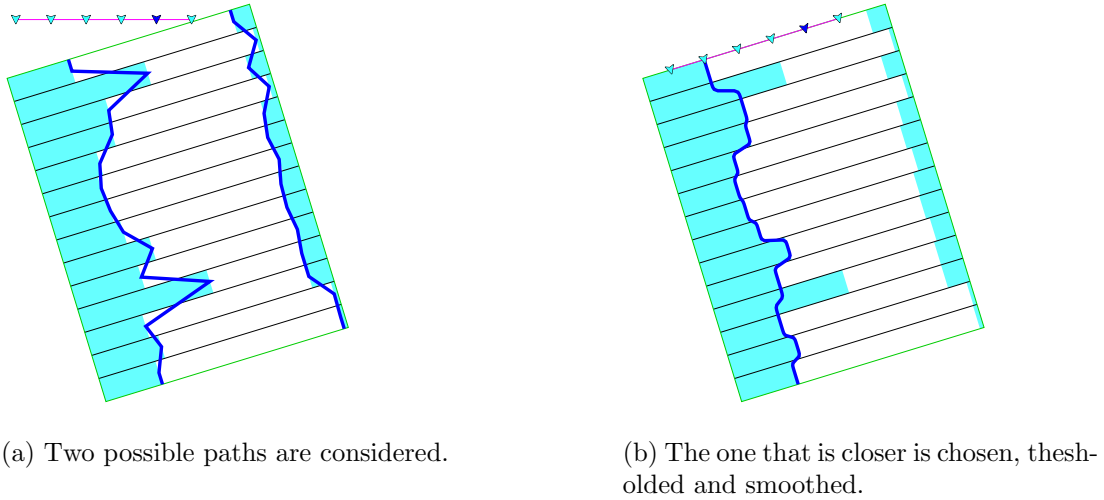


Figure 3.9: Path planning.

constraints.

Using the coverage map obtained from the result of the query, the leader computes two possible paths, respectively on the frontier of the left and the right sides of the explored area (fig. 3.9a). It then discards the one that is farther from the current position of the flock; the remaining one is thresholded (by removing control points that would cause excessive lateral movement) and smoothed (fig. 3.9b).

To drive the flock over the unexplored areas, the leader estimates the width of the flock (using only its own local ADB) as well as its own relative position in it. Using this knowledge, the leader translates so that the edge of the flock lies over the beginning of the planned path (with a 20% margin to compensate for width oscillations due to control and network latency) and then starts translating according to the planned path until the overall flock reaches the other end of the rectangle.

After waiting for the whole flock to complete, a new query is started and, unless the whole rectangle has been covered, the process repeats. Otherwise, if the whole rectangle has been covered, the mission is regarded as *completed*.

### 3.6 Acquiring and transmitting data to a Ground Control Station

When at  $Z_{sensor}$  altitude, each agent continuously checks whether its sensor (e.g. camera) is currently framing an area such that:

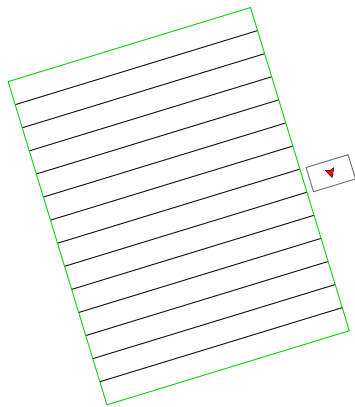
- intersects with the rectangle to be covered (fig. 3.10a);
- the sensor's orientation is aligned to the rectangle  $x$  axis (fig. 3.10b);
- does not cross a stripe boundary (fig. 3.10c);
- it has not been already acquired yet.

If all the above conditions are met (fig. 3.10d), data is acquired and the coordinates of the sensor area's projection are added to the local APD. The actual data is stored onboard for later retrieval, and not shared with the other UAVs.

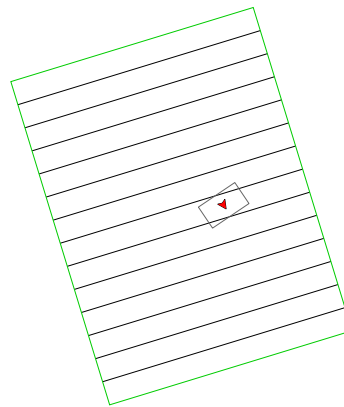
As a further optimisation, if we assume that a fast (but possibly unreliable) cellular network is available, acquired data can be transmitted to a Ground Control Station (GCS) on a best-effort basis. In this case, the onboard storage will work as an outgoing data buffer. Apart from the obvious advantage of making acquired data available even before the end of the mission, this optimisation can improve recovery times: if the UAV that fails had managed to send acquired data to the GCS before failing, there is no need to go with the whole flock back to re-acquire its data.

In this optimised variant, the GCS periodically sends all UAVs the map of the sensor data that it has received so far (in the same format as an APD). Before planning a new path, the leader merges the result of the aggregation query with the latest APD received from the GCS and then proceeds as before. An interesting note is that, despite the presence of a centralised GCS, the algorithm gracefully degrades to non-optimised operation if the GCS (or the communication link) fails: in other words, the algorithm is still tolerant to the failure of any agent (GCS and UAVs). It will simply take longer to complete the overall mission.

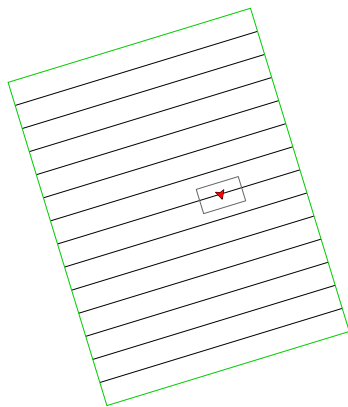
Simulation results of the algorithm presented in this chapter will be shown in section 4.3.



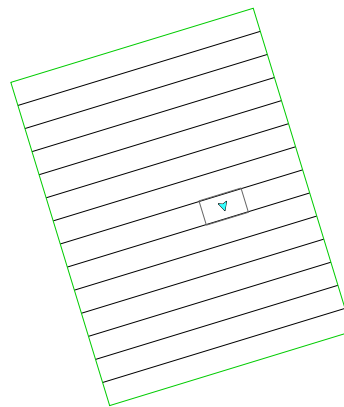
(a) Not intersecting with target area.



(b) Misaligned sensor.



(c) Crossing stripe boundaries.



(d) Good sensor position.

Figure 3.10: Sensor positioning.

# 4

## Modeling a flock of quadrotors

Flocking algorithms, such as the one described in the previous chapter, describe the high-level behaviour of a set of UAVs. To actually run (or simulate) such algorithms, we need to define a model for the UAVs and the wireless network. Depending on the chosen model, the UAVs and the network will be more or less realistic in terms of:

- physical dimensions (e.g. is time measured in generic “simulation epochs” or real-world seconds? are lengths expressed in pixels or meters?);
- primitives offered to the high-level algorithms (in other words, is the “Application Programming Interface”, or API, offered by the simulated UAV the same as the real UAV?);
- latency of UAV commands and wireless messages;
- reliability of UAVs (real-world UAVs sometimes fail; is this aspect modelled?) and wireless network (e.g. percentage of lost packets).

The simplest physical UAV model is a pointwise approximation, capable of any instantaneous movement (i.e. with neither inertia nor maximum speed). This approximation can be useful to validate the early design phases of a new algorithm, but it quickly becomes inadequate if real-world quantities are to be measured (such as total mission time, energy consumption and so on).

In the initial part of this chapter, we describe the basic control laws of a UAV and how they can be simulated.

Wireless networks can, similarly, be simulated with varying levels of detail. Starting from basic wireless range modelling (i.e. do not deliver messages to agents more than a preset number of meters away) and predetermined loss rate, there are much more advanced models. For instance, they can take into account collisions among multiple transmitters, the hidden and exposed terminal problems, asymmetry and physical obstacles to the signal propagation.

In the second section of this chapter, a custom realistic lightweight simulator (with basic wireless networking) is presented. In the third section, the simulator is used to validate the algorithm described in the previous chapter with flocks of up to 40 UAVs.

## 4.1 Control loop of a quadrotor

At its core, a multirotor is made of a frame (the physical structure), a battery, a set of motors with propellers and a *Flight Control Unit* (a microcontroller with some sensors). Multirotors with four motors are called quadrotors. In this section, the control loop of a quadrotor is described. As it is often the case in robotics, complex behaviour is obtained by cascading different controllers, starting from the basic rate controller to a full-fledged GPS-based position controller [48].

There are several layouts for quadrotors. The most common one is probably the “x” layout (fig. 4.1). As can be seen from the picture, motors are identified by a number: the odd ones rotate in clockwise direction, the even ones in counterclockwise direction.

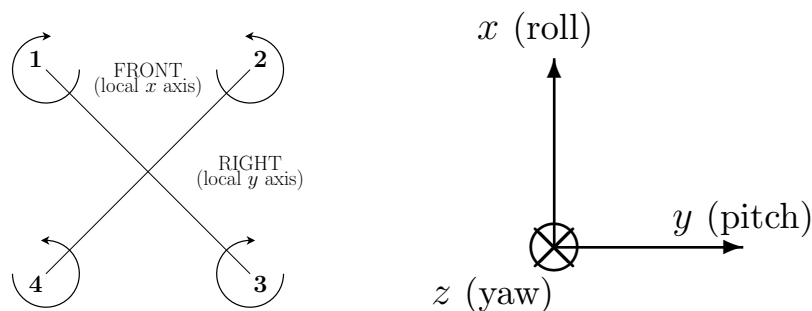


Figure 4.1: Motor layout and coordinate system<sup>9</sup>.

<sup>9</sup>The  $z$  axis enters the paper plane.

Let us consider how each motor contributes to the “unbalancing” and thrust of the quadrotor using a simplified model<sup>10</sup>. Let  $M_k$  be the normalised duty cycle applied to the  $k$ -th motor ( $0 \leq M_k \leq 1$ ). The actual units of measure and the lever arms can be ignored because we are only interested in finding relations of proportionality. The total torque along the  $x$  axis ( $T_x$ ) is affected positively by motors on the left and negatively by motors on the right (eq. 4.1); the total torque along the  $y$  axis ( $T_y$ ) is affected positively by motors on the front and negatively by motors on the back (eq. 4.2; note that the  $x$  and  $y$  axes share the same proportionality factor); the total torque along the  $z$  axis ( $T_z$ ) is affected positively by motors that spin clockwise and negatively by motors that spin counterclockwise (eq. 4.3); the total thrust ( $F$ ) is resultant force on the quadrotor, i.e. the sum of the individual forces by each motor (eq. 4.4).

$$T_x = \alpha ( +M_1 - M_2 - M_3 + M_4 ) \quad (4.1)$$

$$T_y = \alpha ( +M_1 + M_2 - M_3 - M_4 ) \quad (4.2)$$

$$T_z = \beta ( +M_1 - M_2 + M_3 - M_4 ) \quad (4.3)$$

$$F = \gamma ( +M_1 + M_2 + M_3 + M_4 ) \quad (4.4)$$

$$\begin{bmatrix} M_1 \\ M_2 \\ M_3 \\ M_4 \end{bmatrix} = \begin{bmatrix} +1 & -1 & -1 & +1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & +1 & +1 \end{bmatrix}^{-1} \begin{bmatrix} T_x/\alpha \\ T_y/\alpha \\ T_z/\beta \\ F/\gamma \end{bmatrix} \quad (4.5)$$

The four equations above can be seen as  $4 \times 4$  linear system, that can be inverted as shown in eq. 4.5. This function, which transforms the target torques and thrust into the individual duty cycles to be applied to each motor, constitutes the *mixer* block, which sits at the bottom of a flight control stack<sup>11</sup>. Immediately above the mixer block, there is always a *rate controller* block, which drives the mixer’s torque inputs according to target angular speeds (by means of a PID controller). The mixer’s thrust input can be either similarly driven by a *Z-axis velocity controller* or, in some flight modes, directly taken from the pilot.

This is the reason why proportionality factors do not need to be known explicitly at all: they are either absorbed in the gains of an upstream controller or, only

<sup>10</sup>This model is the one that is actually used in some flight stacks, such as CleanFlight [49].

<sup>11</sup>By keeping the *mixer* block separated, supporting a different multirotor layout becomes a matter of simply replacing the mixer, without affecting the rest of the flight stack.

in some flight modes and for the thrust only, known by design because the pilot’s input is naturally normalised over the range of his/her radio-controller throttle stick, therefore  $\gamma = 1$ .

In detail, multirotors can be controlled by the user in several “flight modes”: the simplest one is called “acrobatic” mode, in which the input from the radio is directly passed to the three rate controllers, except for the thrust, which is simply scaled and used as  $F$  (fig. 4.2).

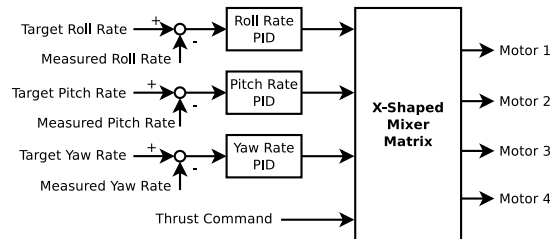


Figure 4.2: Acrobatic mode.

Flying a multirotor in acrobatic mode is extremely difficult because the pilot has to actively keep the multirotor horizontal; furthermore, releasing all sticks (i.e. letting the springs move them to the neutral position) leaves the UAV in the current orientation, possibly tilted. Even simple toy drones, therefore, offer an easier mode, called “manual”, in which the pilot controls the target attitude along the  $x$  and  $y$  axis instead of the rate (fig. 4.3). In manual mode, the multirotor stays horizontal if all sticks are released.

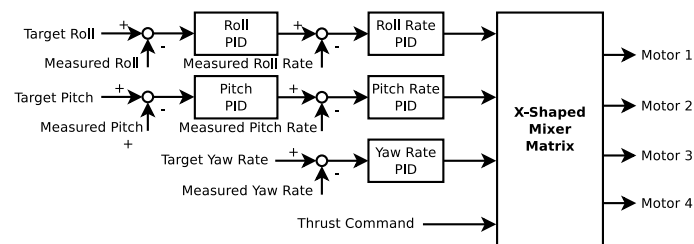


Figure 4.3: Manual mode.

Professional drones (which are equipped with a GPS receiver) are usually flown in “position” mode. From the point of view of the pilot, this mode is even simpler because the user simply moves a virtual target point, which is chased by a position+velocity control loop (fig. 4.4). In this mode, if all sticks are released, the multirotor stays still (hovering).

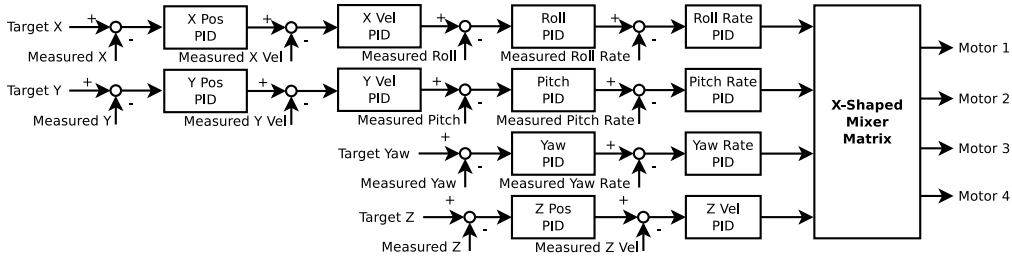


Figure 4.4: Position mode.

“Position” mode is also the most common mode in which autonomous UAVs are commanded by high-level algorithms (that, therefore, feed the control loop with a sequence of *position setpoints*, also called *waypoints*). Another widespread control mode for autonomous UAVs is the one that bypasses the outer controllers and directly applies *velocity setpoints*, usually computed in real-time by some upstream application-specific controller.

## 4.2 A lightweight ad hoc simulator

In section 2.2, several realistic UAV simulators were presented. We developed a simpler and much more lightweight simulator, specifically targeted at quickly prototyping algorithms for medium-scale flocks of UAVs [50, 51].

The proposed simulator [50, 51] is built in C++ on top of the Bullet physics library [28]. Quadrotors are implemented as a single rigid body, with each motor applying a perpendicular force and a torque. Real-time visualisation is implemented in OpenGL [52] for the 3D Views and Qt 5 [53] for the GUI controls (fig. 4.5). It is possible to run the simulations “headless”, i.e. in batch mode without the GUI, to collect numeric data and store statistics as CSV files.

The simulator has built-in support for the notion of *agents*, *physical* agents and inter-agent communication: a ground station can be implemented as an agent; UAVs are *physical* agents, a specialisation of the agents category with a *position* property. Two communication channels are implemented:

- the long-range channel can be used by all types of agents, to send unicast messages;



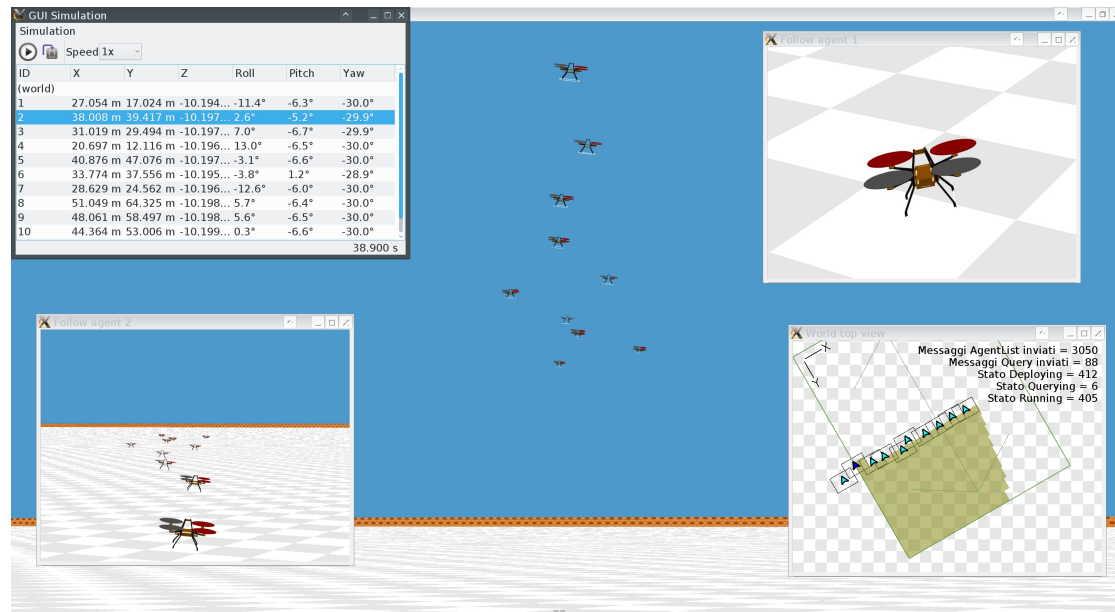


Figure 4.5: Screenshot of a flock in the simulator.

- the short-range channel can be used by physical agents only, to send broadcast messages to neighbouring agents.

Thanks to the simplicity of the simulator, in which all agents run in the same address space, messages do not need to be serialised (thus simplifying the algorithm prototyping phase). Instead, the message dispatcher module of the simulator simply shares reference-counted read-only references of the original message to the intended recipient(s). The dispatcher module can also optionally discard random messages (according to a configurable loss rate) and, for short-range channel messages, it can restrict the delivery to only physical agents closer than a preset distance threshold from the sender.

In addition to the configurable message loss rate, it is possible to simulate entire UAVs failing with a preset failure rate: a periodic simulation task will randomly “kill” (i.e. immediately disable) one or more UAVs accordingly. The simulator will also kill UAVs that collide with either the ground or other UAVs (both UAVs will be killed).

The core of the simulator is separated from the code that instances the simulation: it is trivially possible to reuse the core for a different simulation without even recompiling it. Simulation-specific code goes into a simulation-specific executable, which loads the core as a dynamic library. Simulations can be configured through

text files in `.ini` format (to set, e.g. the initial position and the number of UAVs, the target area size and the simulated failure probability).

### 4.3 Simulation results

The algorithm described in chapter 3 has been implemented and tested in the simulator. A Finite State Machine (FSM) was implemented (fig. 4.6) to clearly identify the different states:

- *Taking Off*: This is the initial state, in which agents turn on the motors and reach  $Z_{sensor}$  altitude, while also starting to build the overlay network to discover other flock members. Depending on the presence of another agent with a lower ID, the next state is either *Querying* or *Non-Leader*;
- *Querying*: A new query starts when this state is entered and this is the leader's state while an aggregation query is in progress. When the query is completed, if its result is that the area has not been fully covered yet, the next state becomes *Deploying*; otherwise the flocking logic is disabled and the agent enters the *Going Home* state;
- *Deploying*: This is the leader's state while it waits for the rest of the flock to reach the planned path's initial position;
- *Running*: This is the leader's state while it is executing a planned path;
- *Waiting*: This is the leader's state while it waits for the rest of the flock to complete the planned path;

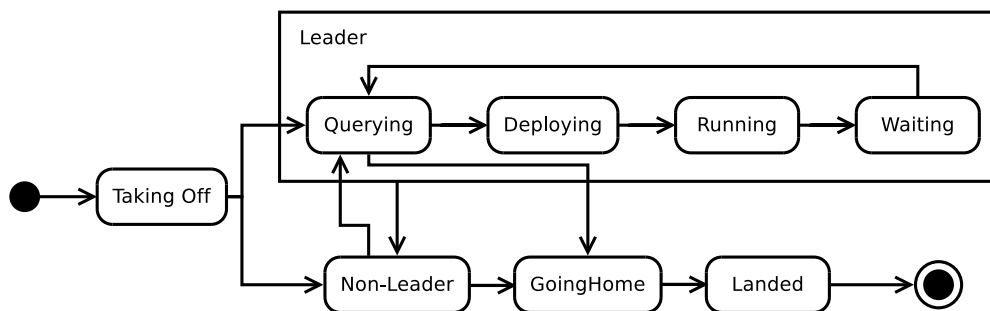


Figure 4.6: Finite State Machine that implements the UAV's behavior.

- *Non-Leader*: This is the state of non-leader agents, obeying to flocking rules. It can be entered at any time from any of the leader’s states if a new agent with a lower ID is detected (i.e. the current agent loses the leader role); similarly, it can be left if the current leader fails and the local agent has the lowest ID among the remaining ones (i.e. the current agent gains the leader role).
- *Going Home*: This state consists in going back to the take-off position (with simple rules to steer away from other agents on the way) and then slowly descending to the ground, while continuously broadcasting a “going home” message that makes other agents immediately enter this state too (this is the mechanism to propagate the information that the mission is completed).
- *Landed*: This is the final state, in which motors are turned off.

The simulated environment consists of a square target area ( $H = W = 600$  m). The flight altitude was chosen so that the sensor’s projection on the ground is  $X_{sensor} = 7$  m and  $Y_{sensor} = 10$  m. The constants referenced by alg. 3 have been set to the values in table 4.1.

Symbol	Description	Value
$D_{R1}$	Threshold for R1 (separation)	7.5
$c_{p\_R1}$	Multiplier for parallel distance component	0.5
$c_{o\_R1}$	Multiplier for orthogonal distance component	0.6
$d_{R1}$	Exponent for orthogonal distance component	0.4
$s_{R1}$	Saturation for R1 (separation)	0.8
$k_{p\_R2}$	Proportional gain for R2 (alignment)	5
$s_{R2}$	Saturation for R2 (alignment)	180
$k_{p\_R3x}$	Proportional gain for R3x (cohesion along X axis)	0.2
$k_{I\_R3x}$	Integral gain for R3x (cohesion along X axis)	0.0005
$s_{R3x}$	Saturation for R3x (cohesion along X axis)	2
$k_{p\_R3y}$	Proportional gain for R3y (cohesion along Y axis)	0.1
$s_{R3y}$	Saturation for R3y (cohesion along Y axis)	2

Table 4.1: Constants used in the simulator.

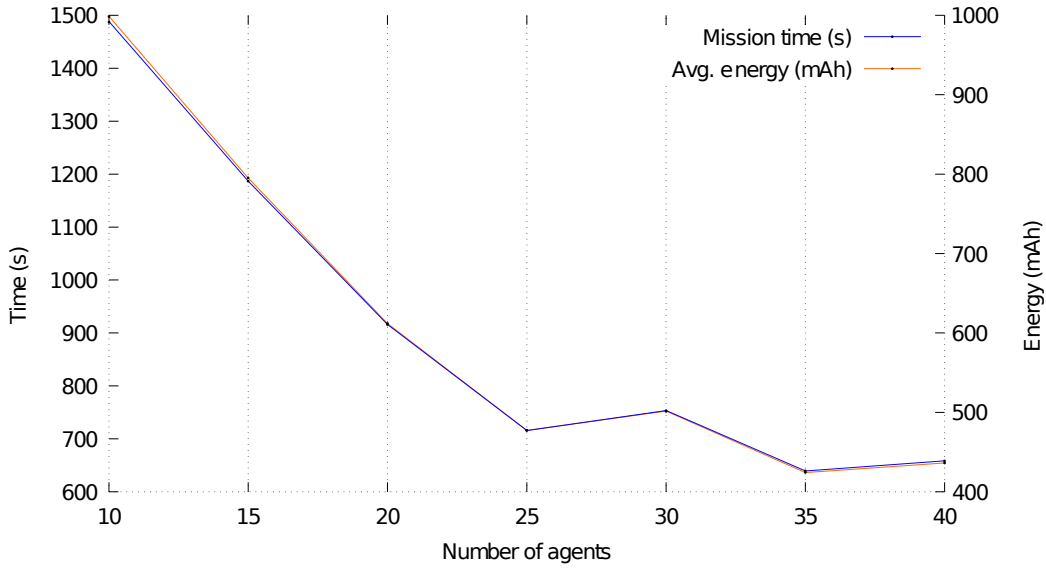


Figure 4.7: Total mission time and average energy consumption.

We analysed the total mission time (from take-off to landing) with a varying number of agents. Fig. 4.7 shows the total mission time and average energy consumption<sup>12</sup> without injected artificial faults (i.e. randomly “killed” agents). As can be observed, if we add more agents (until 25), both the mission time and the

<sup>12</sup>Assuming a 3-cell Li-Po battery (nominal voltage: 11.1 V). In detail, power consumption was computed using a simplified model as follows:

For each motor  $k$ , let  $T_k$  be its thrust. If we assume that its vertical speed is zero, we can compute its mechanical power  $P_k$  using eq. 4.6 [54] (where  $A$  is the propeller area,  $D$  is its diameter and  $\rho = 1.225 \text{ kg/m}^3$  is the air density).

$$P_k = \frac{T_k^{\frac{3}{2}}}{\sqrt{2\rho A}} \text{ with } A = \frac{\pi}{4}D^2 \quad (4.6)$$

If we also take into account the average propeller efficiency ( $\eta_1 = 0.7$ ) and average motor efficiency ( $\eta_2 = 0.75$ ), the electrical power is given by eq. 4.7.

$$P'_k = \frac{P_k}{\eta_1\eta_2} \quad (4.7)$$

Let  $P_{control}$  be the power consumption of the onboard control boards. The total power consumption is given by eq. 4.8 and the current is given by eq. 4.9.

$$P_{tot} = P'_1 + P'_2 + P'_3 + P'_4 + P_{control} \quad (4.8)$$

$$i_{tot} = \frac{P_{tot}}{11.1 \text{ V}} \quad (4.9)$$

average energy consumption decrease. However, starting from 25 agents, there is almost no advantage in adding further agents.

We also ran the same experiment with an artificial agent failure rate (between 0.01% and 0.9%, applied every 10 s). The results are aggregated in table 4.2 into two categories: trials where the number of failed agents was between 0 and 10, and trials where it was between 10 and 30. As can be seen, the latter group presents high variability, whereas the former group is more consistent, proving that, in the presence of a realistic number (i.e. less than 10) of failed agents, the algorithm stays effective.

In detail, we analysed the ratio of failed agents (and the resulting repeated coverage) with a varying artificial failure probability (table 4.3). Details about the resulting overcoverage distribution are shown in fig. 4.8. The experimental results prove that, when agents fail, some overcoverage is present, but it is at negligible levels and the algorithm stays efficient.

N. of Failed Agents	Mission Time (s)		Energy consumption (mAh)	
	Avg	Std Dev	Avg	Std Dev
0-10	719	63	479	44
10-30	1360	418	913	282

Table 4.2: Effect of failed agents on mission time and energy consumption.

Failure prob.	Failed agents	Distance flown to recover from failed agents
0.0005	2.50 %	0.0000 %
0.0007	3.75 %	0.1493 %
0.0009	5.00 %	0.2095 %
0.001	3.75 %	0.1924 %
0.003	12.16 %	0.5015 %
0.005	10.94 %	0.7475 %
0.007	38.33 %	0.6517 %
0.009	28.57 %	1.3916 %

Table 4.3: Ratio of failed agents vs extra distance for recovery.

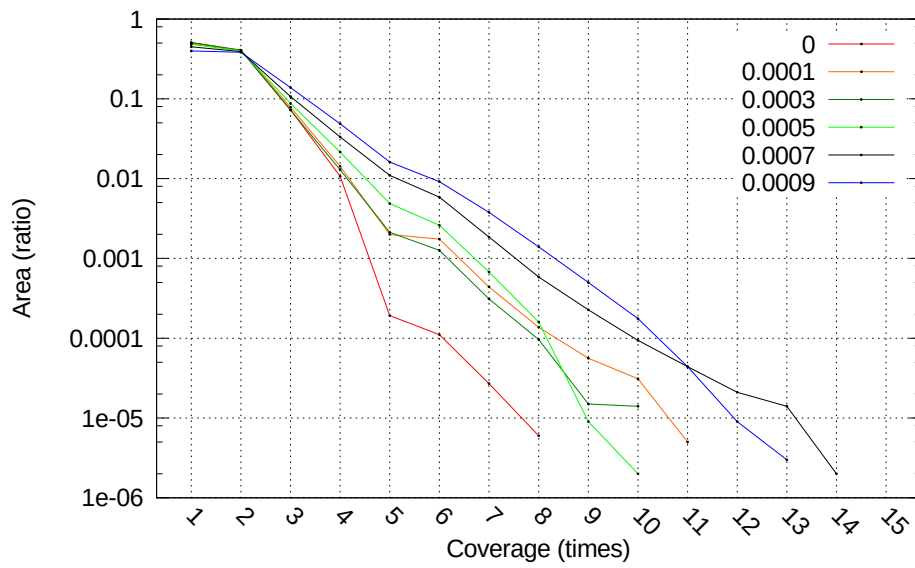


Figure 4.8: Overcoverage Distribution.

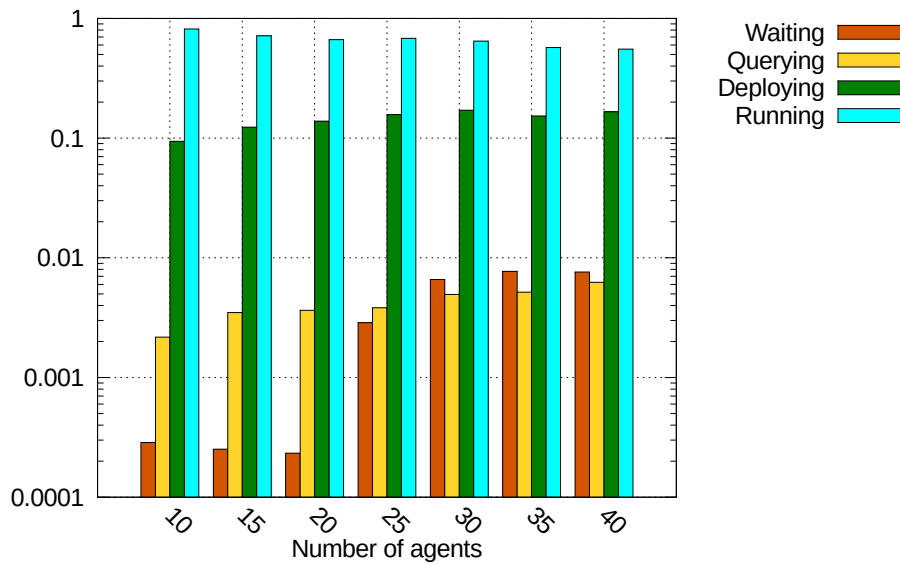


Figure 4.9: Time spent in each state.

We also analysed the relative time spent in each state of the FSM. The results show that time spent in “service” states is always below 2% (fig. 4.9, note: *Going Home* and *Landed* states have been excluded from the statistics) and, unsurprisingly, the percentage of time spent in this state increases when we add more agents.

In conclusion, if we analyse all the experimental results that we obtained, 25 seems to be the optimal number of agents. It is important to note that this is not an *absolute* outcome: this number is likely to change if we consider a target area of different size (parameters  $W$  and  $H$ ), a sensor with different  $X_{sensor}$  and/or  $Y_{sensor}$ , or change any of the parameters listed in table 4.1.

More experimental results are presented in [51].

# 5

## Combining heterogeneous tools for realistic UAV simulation

As presented in section 2.2, several options to simulate autonomous UAVs exist. Some are more realistic than others (e.g. accurate physics, API, scenery). Similarly, several tools to simulate wireless networks exist, with different levels of detail and flexibility.

In the work presented in this chapter [55, 56] we combined two heterogeneous simulation tools:

- A UAV simulator, Gazebo with ArduCopter<sup>13</sup> SITL, to provide accurate physics and API;
- A wireless network simulator, by integrating the well-known ns-3 network simulator.

The proposed architecture also lets you run several UAVs in the same simulation (e.g. to simulate flocks) and makes it possible to spread part of the simulation workload over a LAN. Unlike the simulator presented in the previous chapter, this simulation environment is meant to be used for the final development and validation, running the same code that can be run on the real UAV.

---

<sup>13</sup>We chose ArduCopter over PX4, because of its simpler yet flexible codebase (PX4 is based on a publisher-subscribe framework called *uORB*, whereas ArduCopter is based on a plain object-oriented design).



The resulting work, called *gzuav*, has been made available for download at <https://gzuav.dmi.unict.it> (with full source code and precompiled Ubuntu `.deb` packages). It includes a Gazebo plugin, an ns-3 plugin, a customised ArduCopter SITL backend and source code for the *gzuav* infrastructure, as well as launch scripts and usage examples in Python.

## 5.1 Co-simulation of physics and networking

Network simulation is a widely studied topic with a wide range of available tools, such as ns-2, ns-3 [57] (successor to ns-2) and OMNeT++ [58]. However, network simulation tools are only focused on the networking aspect and nodes' behaviour are usually idealised models (e.g. “simulate a node that transmits random 1000 bytes payload ten times a second”). In our approach, we cannot use such a simple model, because the behaviour of each node depends on the rest of the simulation. More precisely, we wanted to combine Gazebo's physics simulation with a realistic network simulator (this approach of mixing two or more heterogeneous tools is called “co-simulation”). We selected ns-3 because it has a plugin-like architecture that makes it possible to define a custom event loop without affecting the rest of the system; this aspect is peculiar to our co-simulation needs: “plain” network simulations would run on ns-3's standard event loop but, in our case, we needed to synchronise network events with the rest of the simulation components, namely Gazebo, the High-Level logic and ArduCopter, as we will see in the next section.

Other works, similar to our own, that combine different simulations tools are [59], in which the authors combine a custom physics engine (written in MATLAB) with OMNeT++, and [60], in which X-Plane [32] produces flight paths that are consumed in real-time by an OMNeT++ simulation (however, the communication flow between the two simulators is not bidirectional: network events cannot be used to make decisions about subsequent flight paths). Another interesting example is [61]: it simulates a group of robots (with the ARGoS [23] robotics simulator) interacting over a wireless channel (simulated in ns-3). This approach is similar to our own; however, the modifications made by the authors to the two simulators are much more invasive than ours (making integration more difficult for the end-user); furthermore, the behaviour of the robots must be programmed with APIs that are different from those of the real robot.

## 5.2 Flight stack architecture

Before presenting the simulation architecture, let us analyse the components of the flight stack on a real drone.

The propellers are attached to brushless motors, mounted on each leg of the frame (whose shape can be  $\times$ ,  $+$ , hexarotor, ...). Each brushless motor is connected to an Electronic Speed Control (ESC) module, which drives the three phases of the motor so that it reaches a reference speed, controllable by means of a digital PWM signal.

Motors are controlled (through the ESC's PWM signal) by the Flight Control Unit (FCU), usually a microcontroller running a real-time control firmware (such as ArduCopter or PX4). Some sensors (usually the accelerometer, the gyroscope and the barometer) are soldered on the FCU board too. The FCU must be as close as possible the UAV's centre of mass for optimal performance. The GPS and magnetometer sensors are usually placed externally, above the motor plane, to have a better sky view and less electromagnetic interference from the motors.

Manually controlled UAVs need an RC receiver unit<sup>14</sup>, which is also connected to the FCU. Depending on the RC receiver type, several protocols exist to encode the input signal, but they all transfer the same information: the position of the sticks in the pilot's RC transmitter.

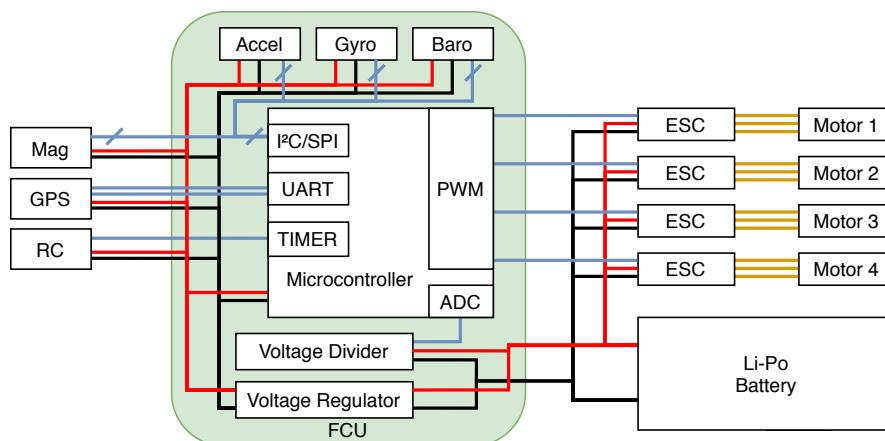


Figure 5.1: Connection diagram of a quadcopter.

<sup>14</sup>Autonomous UAVs are usually equipped with an RC receiver too, for safety or emergency takeover.

Lastly, the ESCs and the FCU need to be powered<sup>15</sup>. Li-Po batteries are usually employed because of their good energy density and discharge rate. An Analog-Digital Converter (ADC) on the FCU can monitor the battery voltage to detect the remaining charge.

Fig. 5.1 summarizes the hardware connections explained above.

The FCU continuously receives measurements from the sensors and updates the estimation of the current pose in real-time, usually with a Kalman filter to perform *sensor fusion*. The estimated pose  $(x, y, z, roll, pitch, yaw)$  and its derivative (linear and angular velocities) are consumed by the algorithms described in section 4.1.

If UAV is autonomous, an external “companion computer” (e.g. a System-on-a-chip, or SoC, such as a Raspberry Pi) can be connected to the FCU to i) receive pose updates in real-time; ii) execute its own autonomous control logic; iii) send position/velocity commands. In PX4 and ArduCopter-based FCUs, the connection happens over a UART with an open protocol called MAVLink [62]. In a multi-UAV application, the SoC is also connected to an RF antenna (e.g. IEEE 802.11 and/or IEEE 802.15.4-UWB) to communicate with the other agents.

It is worth noting that, from the point of view of the companion computer, the “MAVLink” protocol is the API of the FCU. Therefore, as we will see in the next section, our simulation environment lets simulated companion computers interact with the associated UAV through a MAVLink channel.

### 5.3 The gzuav environment

In our proposed simulation architecture [55, 56], each UAV is split into six interconnected software modules:

- Gazebo Visual Model: 3D graphical representation of the UAV, only used if the simulation is running with a GUI (i.e. not in batch/headless mode);
- Gazebo Physical Model: 3D description of the rigid bodies that compose the UAV (propellers, motors+frame) and joints that connect them;

---

<sup>15</sup>The RC receiver and the sensors are usually powered in cascade by the FCU.

- GzUavPlugin instance: an instance of our plugin for Gazebo, which exposes the physical model to the external components;
- ArduCopter process: an ArduCopter instance, running in SITL mode;
- UAV Node: module that acts as a proxy for the UAV within the network simulation process;
- High-Level Logic: an instance of the software that would run on the companion computer, provided by the user.

A middleware called “gzuavchannel” lies at the core of the gzuav architecture, whose purpose is to synchronise the execution of the other components. We defined a protocol, comprising two phases called *Phase 0* and *Phase 1* (fig. 5.2). Synchronisation begins when, at the beginning of each time step, each instance of the Gazebo plugin emits a **Begin-Tick-AC** message<sup>16</sup>. Such messages are forwarded as-is to the corresponding ArduCopter processes. At the same time, an additional **Begin-Tick-0**<sup>17</sup> is generated and sent by gzuavchannel to the UAV’s High-Level Logic process. Therefore, ArduCopter and the High-Level Logic run in parallel. When they complete the processing of the time step, they both send a signal (ArduCopter sends **End-Tick-AC**<sup>18</sup> and ArduCopter sends **End-Tick-0**<sup>19</sup>). When all **End-Tick-AC** and **End-Tick-0** messages have been collected from all the UAVs, gzuavchannel generates and sends a **Begin-Tick-1**<sup>20</sup> to the ns-3 process. Upon reception of such message, our ns-3 module updates the position of the simulated nodes, runs the events associated to the current time step and, in the end, sends **End-Tick-1**<sup>21</sup>. When gzuavchannel sees this message, it forwards the **End-Tick-AC** messages it had received to the respective Gazebo plugin instances, thus completing one cycle of the simulation protocol.

---

<sup>16</sup>The **Begin-Tick-AC** message contains the current simulation timestamp (according to the simulated clock), vehicle’s pose, linear/angular velocity and acceleration.

<sup>17</sup>The **Begin-Tick-0** message contains the current simulation timestamp.

<sup>18</sup>The **End-Tick-AC** message contains the simulated PWM duty cycle for each motor.

<sup>19</sup>The **End-Tick-0** message contains no payload.

<sup>20</sup>The **Begin-Tick-1** message contains the current simulation timestamp and each vehicle’s position.

<sup>21</sup>The **End-Tick-1** message contains no payload.

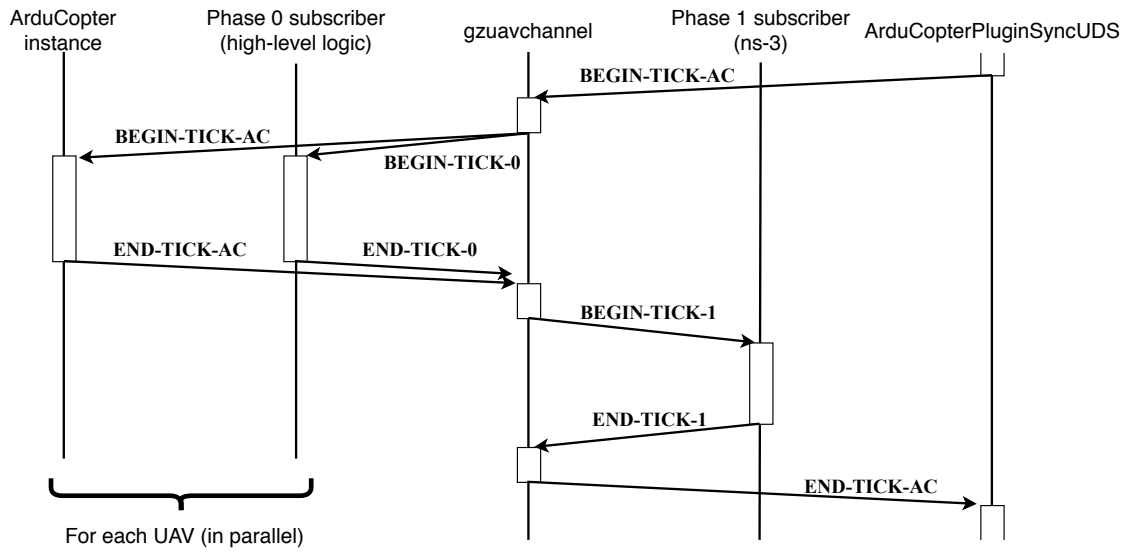


Figure 5.2: Interactions among components for each simulation step.

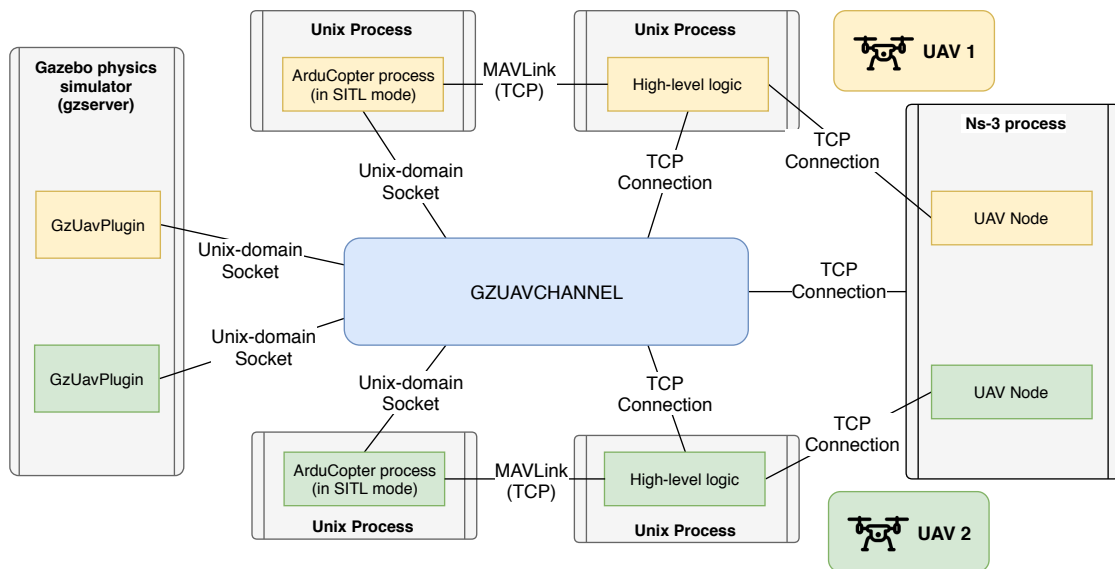
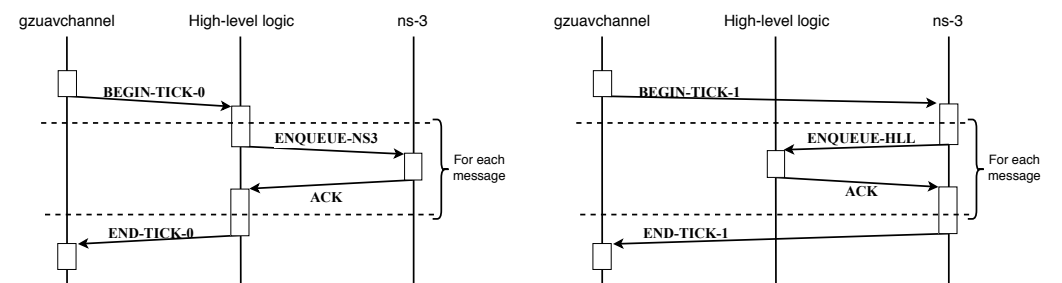


Figure 5.3: Connections among components in a two-UAV simulation.



(a) Messages sent by UAVs are enqueued in ns-3 during Phase 0. (b) Ns-3 runs the network simulation, including the delivery of messages to UAVs, during Phase 1.

Figure 5.4: Interactions between high-level logic UAV processes and ns-3.

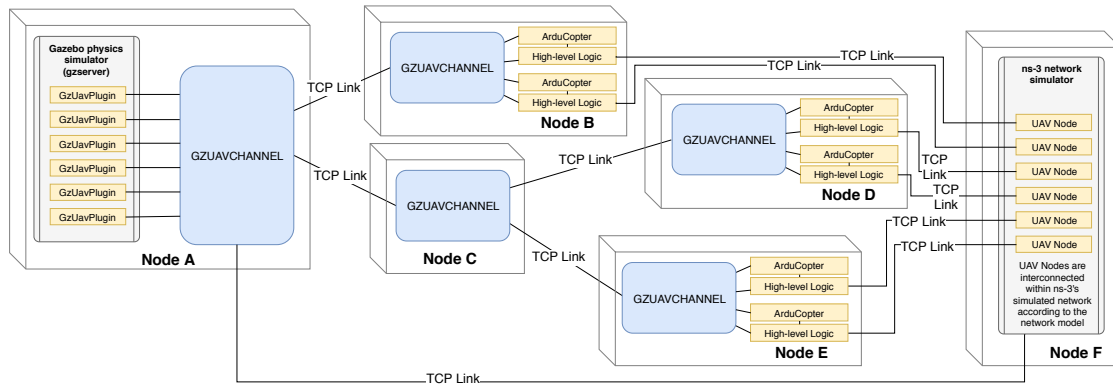


Figure 5.5: Architecture of gzuavchannel in a distributed environment.

Fig. 5.3 shows all the connections that exist for a two-UAV simulation. Most components are connected through gzuavchannel, which also transfers basic data along with timing messages, as outlined above. Two communication channels are direct between the involved components: one of them, the MAVLink channel between ArduCopter and the High-Level Logic, is simulated with a direct TCP connection.

The other direct channel is the one between ns-3 and the High-Level logic. It is also a TCP connection, and we defined a protocol to send and receive messages over it. A peculiar aspect is that messages can only be sent by the High-Level Logic during Phase 0 (while it is running) and dispatched by ns-3 during Phase 1 (fig. 5.4). As a consequence of this, all messages have a minimum latency, introduced by the simulation architecture, equal to the simulation step duration. The user can configure the step duration: typical values are in the 1-5 ms range. Smaller values result in more accurate physics in Gazebo and lower artificial latency in ns-3; however, they also result in increased CPU usage and, if the CPU saturates, slower-than-real-time simulations<sup>22</sup>.

A desirable side effect of the modular architecture is that different processes can be run on different nodes. This makes it possible to run distributed simulations. The gzuavchannel program supports being run in a master-slave fashion, where the master node runs Gazebo, and the slave nodes run ArduCopter, ns-3 or the High-Level Logic processes. Fig. 5.5 shows this kind of setup in practice.

<sup>22</sup>In case of excessive CPU usage, the simulation slows down. However, its correctness is not affected, provided the High-Level Logic synchronizes its timers to the timestamps received in **Begin-Tick-0** and does not use any timekeeping system call offered by the host operating system.

Thanks to the usage of the ArduPilot project, another interesting side effect is that different supported vehicle types (e.g. fixed-wing planes, rovers) can be integrated into the same framework. However, we have not investigated this possibility in detail.

Lastly, it should be noted that, although the next chapter will present an ad-hoc C++ framework for MAVLink programming, any existing MAVLink library can be employed by user programs. In fact, gzuav itself, in its downloadable package, includes some High-Level Logic program examples written in Python using the DroneKit [63] library.

---

# 6

## A software architecture for UAV applications

In the previous chapter we defined an environment to realistically simulate UAVs that can be controlled through the MAVLink [62] API. We also said that, in real UAVs, the MAVLink channel is usually connected to an onboard “companion computer”.

In detail, the “companion computer” is usually a Linux SoC board such as a Raspberry Pi, NVIDIA Tegra TX1/TX2 or Odroid XU4. There are many libraries for writing Linux applications that communicate with a MAVLink FCU, such as DroneKit [63] (for Python) and MAVROS [7] (for the ROS framework). A plain “mavlink” library exists too [62]; however, it only provides the implementation of the message serialisation/deserialisation routines. In fact, all of the previously mentioned libraries are actually wrappers libraries around it.

In this chapter we propose a software architecture, implemented as a C++ framework that uses the “mavlink” library too, while also providing a range of useful features for UAVs applications in a uniform and integrated fashion. In particular, the framework has support for:

- Controlling a UAV over a MAVLink channel;
- Tuning and monitoring Proportional-Integral-Derivative (PID) controllers;
- Acquiring live images from a connected camera (such as a RaspiCam or a USB camera);



- Delivering such images to instances of computer vision algorithms;
- Routing the output of such algorithms to the control loops that need it;
- Logging and transmitting live telemetry data, such as vehicle pose/state and (annotated) video feed to a Ground Control Station over IEEE 802.11;
- Exchanging application-specific direct messages with other robots over IEEE 802.15.4-UWB (A2A, in a decentralised way);
- Exchanging application-specific messages and synchronizing clocks with a specialised Ground Control Station (A2G and G2A, e.g. to coordinate groups of robots in a centralised way) over IEEE 802.11.

Furthermore, when an application based on this framework is executed in the GzUav simulation environment, its clock is transparently synchronised to the simulation clock. This is especially useful if the application contains time-dependent routines, such as timers or PID controllers.

While the architecture currently only supports MAVLink FCUs, it would be very straightforward to repurpose it for different types of FCUs or even different types of robots (e.g. rovers).

## 6.1 Onboard software

In robotic systems, many external inputs can be processed as *events*: an event might be the reception of a message from another robot, a byte over a serial port, a grabbed frame from a connected camera or the expiration of a timer (e.g. to trigger an iteration of the control loop). Even in the case of continuous data sources, such as a digital or analog input pin, a fixed sampling rate is usually imposed by software; therefore, continuous data can be exposed as *sample events* too. The proposed architecture runs on Linux and uses its *epollfd* interface to efficiently manage event handlers.

The *epollfd* interface is similar in purpose to the *select* and *poll* system calls, which enable processes to wait for events from any file descriptor in a given set. But, unlike those system calls, that require the caller to pass the list of file descriptors to each call, with *epollfd* they can be registered only once. An interesting aspect is that *epollfd* instances are themselves valid file descriptors, that can be

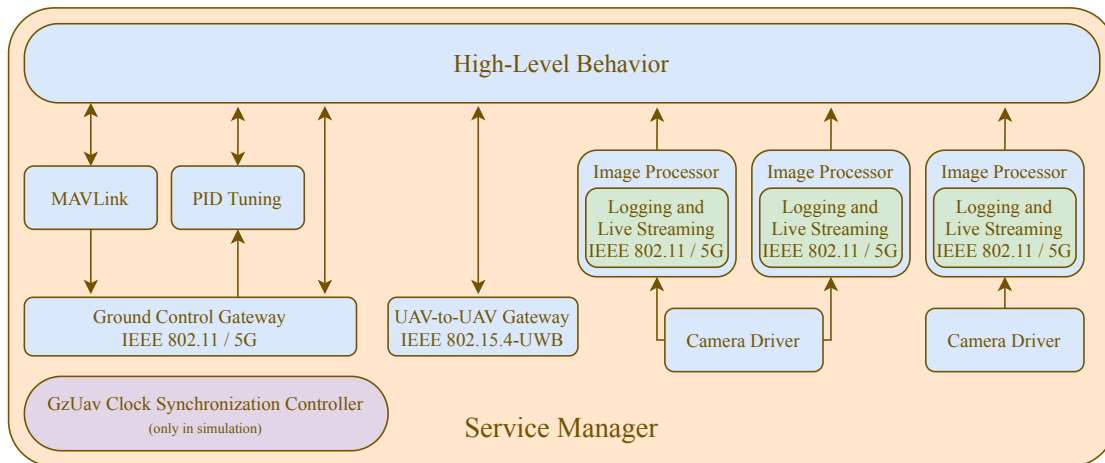


Figure 6.1: Software architecture of an autonomous UAV.

added to other *epollfd* instances. This aspect, in particular, makes them particularly useful to encapsulate and combine several low-level components (e.g. a serial port and a set of timers) into a higher-level one (e.g. a robust abstraction of a serial protocol with timeouts).

Another goal of this architecture is the isolation of faults. Control software tends to be multi-threaded, e.g. to decouple real-time activities such as control loops from background tasks such as image processing. Faults may happen for various reasons, such as hardware failures (e.g. loose cables) or programming errors; in monolithic software, they usually result in a *segmentation fault* or similar types of crashes, that cause immediate process termination by the operating system, with all of its threads. While this behaviour can be tolerable if only one UAV is involved<sup>23</sup>, in the case of an architecture designed to control *groups* of UAVs, this behaviour would be very undesirable: if several UAVs stopped in-flight, there might not be enough time for individual manual recovery actions, before the batteries run out.

The proposed architecture isolates modules by executing them in different processes and only allowing inter-process communications over well-defined channels. Within each process, one or more threads run, each having its own event loop. Fig. 6.1 shows the diagram of the processes in a typical autonomous UAV system with onboard computer vision algorithms. In this scheme, if a process crashes, only that module's functions are affected (and, possibly, the ones that depend on it).

<sup>23</sup>Provided watchdogs to stop further movements are in place (i.e. automatically start hovering until the human pilot takes control), and manual recovery is possible.

For instance, let us suppose that the “High-Level Behavior” process crashes: then, the “MAVLink” process will still be there allowing a new, respawned, “High-Level Behavior” process (or a different one, such as a process that runs an automatic emergency landing procedure) to take control.

The following modules are present in a typical autonomous UAV system:

- “High-level Behaviour”: implements the autonomous behaviour;
- “MAVLink”: connects to the microcontroller over a serial port and abstracts the low-level MAVLink protocol;
- “Ground Control Gateway”: is the single entry-/exit-point of all messages from/to the Ground Control Station (A2G and G2A) via WiFi or mobile data network;
- “UAV-to-UAV Gateway”: is the single entry-/exit-point of all messages from/to other robots (A2A) via 802.15.4-UWB;
- “PID Tuning”: lets user alter PID gains and monitor related control variables live;
- “Camera Driver”: acquires frames from connected cameras (one instance for each camera);
- “Image Processor”: processes acquired frames (one instance for each processing pipeline);
- “Logging and Live Streaming”: compresses and optionally sends an annotated video stream to the Ground Control Station in real-time via WiFi or mobile data network;
- “GzUav Clock Synchronisation Controller” (only in simulation): if running within the simulation environment described in chapter 5, it offers an alternative timer implementation that is synchronised to the simulation clock instead of the host computer’s.

Each thread in each process runs its own event loop (one for each thread), monitoring only its own file descriptors and responding to their events. Most processes are single-threaded by design: in general, the proposed architecture prefers message-passing rather than shared memory, in order to avoid the possibility of race conditions. For instance, the “MAVLink” process consists of a single thread

that monitors the serial port for incoming data from the UAV’s microcontroller, a *timerfd*-based timer for heartbeat generation and set of sockets to receive and reply to messages from other processes. “MAVLink” is, in turn, a client to the “Ground Control Gate” process using a different socket for message-passing. In general, in such an architecture, which is essentially a distributed one, processes use direct internal sockets (more specifically, *Unix domain sockets*) to communicate with each other.

This type of message-passing happens among local processes only. Keeping this in mind, in order to simplify development and maintain efficiency, a simple serialisation library (based on C++ templates) was developed, that directly passes binary memory dumps of messages, whenever possible<sup>24</sup>. In order to further reduce development time, we save the developer from having to explicitly write code to dispatch different types of received messages to different routines. Instead, a *Remote Procedure Call (RPC)* layer is adopted, where each *request* message is prefixed by the library with the address of the routine that should be called in the receiving process, with the rest of the data as an argument; *response* messages (i.e. messages sent in reply, by the receiving process, that contain the serialised return value of the called routine) do not have this field. This mechanism was wrapped in a set of object-oriented C++ template classes, so that using it is as simple as defining one abstract and two concrete classes (see alg. 5): the *MyInterface* class defines the interface of the services offered by a module, *MyServer* is the concrete implementation of such services (which runs in the isolated process) and *MyProxy* is a thin layer that runs in client processes and enables them to call the offered services. A direct socket connection exists between server and client: such connections are established through the “Service Manager” process, which is also the one that starts all the other processes at initialisation time (by forking<sup>25</sup>), and then works as a broker for connection establishment.

In the proposed architecture, all inter-process messages are implemented using the above scheme, except for:

- Output from the “Image Processor” processes to the “High-level Behavior”,

---

<sup>24</sup>Unlike other serialisation libraries that, in order to ensure binary-compatibility among different processors/languages/compilers, require messages to be explicitly defined in some domain-specific language, we aim at local messages only; therefore, our simple library serializes C++ Plain Old Datatypes (POD) as plain memory dumps and STL containers as length-prefixed streams of contained objects (which are recursively serialised using the library itself).

<sup>25</sup>All involved processes **must** fork from a common parent; otherwise, the addresses of called functions would not match due to the operating system’s Address-Space Layout Randomisation (ASLR) mechanism.

---

**Algorithm 5** Minimal Remote Procedure Call example

---

```
class MyInterface { // abstract class
    public:
        virtual int accumulate(int val) = 0;
}

class MyServer : public MyInterface {
    public:
        MyServer(UnixSeqPacketServer *unixDomainServer)
            : rpcServer(this, unixDomainServer) {
            total = 0;
        }

        int accumulate(int val) {
            return total += val;
        }

    private:
        RPCServer<MyInterface> rpcServer;
        int total;
}

class MyProxy : public MyInterface, public RPCClient<MyInterface> {
    public:
        MyProxy(UnixSeqPacketConnection *unixDomainSocket)
            : RPCClient(unixDomainSocket) {
        }

        int accumulate(int val) {
            return rpcInvoke(&MyInterface::accumulate, val);
        }
}
```

---

which employs serialised messages over *Unix domain sockets* like the rest of the system, but in one direction only and without the *RPC* layer.

- Frames from the “Camera Driver” to connected “Image Processor”, that are propagated in a zero-copy fashion over a triple-buffered shared memory area, protected by cross-process mutexes (`PTHREAD_PROCESS_SHARED`).
- Incoming UAV-to-UAV and Ground Control messages, which are transmitted by the “UAV-to-UAV Gateway” or “Ground Control Gateway” to the intended recipient over raw *Unix domain sockets* sockets.

Since the position comes from the MAVLink input channel in geographic coordinates (i.e. as a  $lat, lon, alt$  triplet), in order to make it simpler to manipulate it, the “MAVLink” module converts, with negligible approximation (provided the UAV does not go further than a few kilometres around the origin), to a local 3D Cartesian reference system centred in  $(lat_0, lon_0, alt_0)$  and rotated by  $hdg_0$  radians, using eq. 6.1<sup>26</sup>.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = lla2xyz \begin{pmatrix} lat \\ lon \\ alt \end{pmatrix} = \begin{bmatrix} \cos hdg_0 & \sin hdg_0 & 0 \\ -\sin hdg_0 & \cos hdg_0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R(lat - lat_0) \\ R(lon - lon_0) \cos lat_0 \\ alt_0 - alt \end{bmatrix} \quad (6.1)$$

## 6.2 UAV-to-UAV protocol

In a multi-UAV scenario, UAVs usually need to communicate with other UAVs. One possibility is to have the Ground Control Station relay UAV-to-UAV messages, i.e. a message is first sent to the ground station (A2G) and then retransmitted to the final recipient (G2A). There is no explicit support for this schema by the framework, because it can easily be implemented at the application level, with a few lines of Ground Control Station code.

A more interesting possibility is to send direct A2A messages. In the proposed solution, we adopted IEEE 802.15.4 as the medium for direct UAV-to-UAV mes-

---

<sup>26</sup>Angles are expressed in radians.  $R$  is the equatorial radius of the Earth (6378100 m). Note that the  $lla2xyz$  function is invertible; therefore  $xyz2lla$  can be defined too and used for the inverse transform.

sages<sup>27</sup>. An interesting aspect of this choice is that it also enables *local broadcasting* of messages, i.e. sending a message that will be received only by agents within the transmission range. Indeed, this type of broadcasting is very useful to implement bio-inspired flocking algorithms (because they often propagate information to/from neighbours only), such as the one presented in chapter 3.

As mentioned in section 2.1, IEEE 802.15.4’s CSMA/CA mechanism does not work well in presence of hidden/exposed node conditions, which are frequent in our type of applications. In the proposed architecture, we replace it with a custom TDMA mechanism synchronised to the GPS clock. In particular, we assume that the transceiver’s built-in CSMA/CA mechanism is disabled and that an external real-time system (such as a microcontroller) can trigger the immediate transmission of a frame. We also assume that all distinct GPS modules emit a synchronised pulse once a second (therefore, the period of the pulse is  $T = 1$ ). Then, let  $N$  be the number of robots and let  $D$  (with  $D < T/N$ ) be the maximum time between the generation of a “send frame” command (on the microcontroller) and the completion of its transmission (on the transceiver). Then, we subdivide the GPS’s clock signal period  $T$  into slots of duration  $D$ . Each slot is statically allocated to a single UAV which, at a given time, will be the only one transmitting on the channel. As fig. 6.2 shows, slots are assigned sequentially and the sequence restarts after period  $T$ . If  $D$  is not a divisor of  $T$ , the last slot goes wasted; if  $ND$  is not a divisor of  $T$ , the last subsequence will be truncated.<sup>28</sup>

In such a system, no collisions can occur by design and a UAV, during its slot, can transmit (unicast or broadcast) to its neighbours. The maximum latency in such a system is bounded: a UAV has to wait for its next slot before being able to transmit a message. In the worst case<sup>29</sup>, the latency will be less than  $2ND$ .

---

<sup>27</sup>Therefore, UAVs must be equipped with an IEEE 802.15.4 transceiver. If the application does not require A2A connectivity, the 802.15.4 transceiver can be omitted and this part of the framework can be excluded.

<sup>28</sup>We validated the effectiveness of the proposed schema with three Microchip MRF24J40 transceivers, each connected to a u-blox C94-M8P RTK GPS receiver and a Microchip dsPIC33FJ128GP802 microcontroller. We experimentally verified that all messages were delivered correctly without collisions.

<sup>29</sup>The worst case occurs with  $\frac{D}{T} \notin \mathbb{N}$  and  $\frac{T}{ND} \notin \mathbb{N}$ , when the last UAV is already transmitting and it needs to send another message, which queues up, while in the second-to-last subsequence.

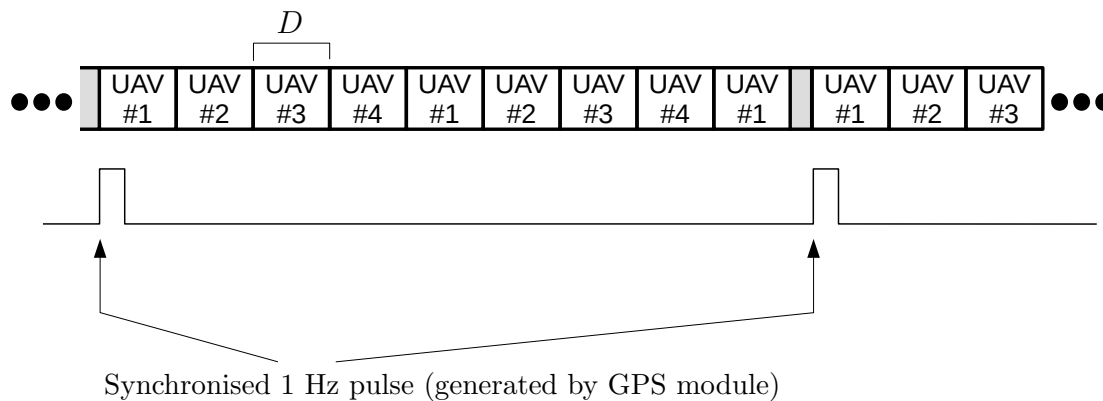


Figure 6.2: GPS-based TDMA slot assignment.

### 6.3 Ground Control Station protocol

In addition to UAV-to-UAV communication, UAVs usually need to communicate with a central Ground Control Station. A Ground Control Station can be useful for safety purposes (e.g. monitoring and emergency recovery), but also during normal operation, as a central coordination node (e.g. to collect/relay information to/from UAVs).

In the proposed architecture, we assume that IP connectivity is available between each UAV and the Ground Control Station (for A2G and G2A communication). We do not require IP connectivity among UAVs (A2A, if needed, can be served with the method presented in the previous section). Examples of compatible wireless technologies are Wi-Fi and mobile data network<sup>30</sup>.

This type of wireless links is usually unreliable, with tight bandwidth constraints. The TCP protocol adds reliability, but its automatic retransmission can be problematic: some types of messages are not worth retransmitting if more recent data is already available (e.g. it would be better for a UAV to transmit the current position instead of retransmitting an unacknowledged “current position” message emitted one second ago). On the other hand, plain UDP would not offer any retransmission at all.

With the above considerations in mind, we developed a simple protocol (whose frames are encapsulated in UDP packets) that offers built-in reliability and more

<sup>30</sup>In Wi-Fi networks, full connectivity among UAVs (mediated by the Access Point) would actually be possible. In cellular data networks, nodes can usually only connect to public addresses on the Internet; therefore, the Ground Control Station must be on a publicly-reachable network.



fine-grained control of retransmission policies. In this protocol, the programmer must associate a “topic ID” (a 16-bit integer) to each message type. The highest bit in the topic ID determines the retransmission policy:

- If the highest bit is one (i.e. the topic ID is between 0x8000 and 0xffff), only the most recent message with that topic ID can be retransmitted - if a new one is pushed to the outgoing queue, the old one is immediately dropped;
- If the highest bit is zero (i.e. the topic ID is between 0 and 0x7fff), older messages in the outgoing queue are not dropped and can be retransmitted if necessary.

Rate-limiting is achieved by periodically gathering all outgoing messages (both first-time transmissions and retransmission, from all topic IDs) into a single UDP packet. This rate is fixed (e.g. 10 Hz by default) and can be tuned to find a trade-off between packet rate and maximum acceptable latency.

In addition to outgoing messages, UDP packets also contain acknowledgements for messages received in the opposite direction. Until a message is acknowledged, it is retransmitted in every UDP packet (according to the policy associated with the topic ID’s highest bit).

Fig. 6.3 shows the packet structure: it is made of a fixed-size header, a variable number of ACK records and a variable number of message records. In addition to message-related data, the Ground Control Station also writes its transmission timestamp in each packet. Upon reception, each UAV estimates the difference between its local clock and the Ground Control’s clock using alg. 6, whose error corresponds to the shortest of the propagation delays of the observed received packets<sup>31</sup>. Timestamps are useful in many aspects of robotic applications (e.g. logging, rendez-vous, correlating data acquired by different robots at the same time): using the estimated time difference, UAVs can convert timestamps from local time to reference time (i.e. Ground Control’s clock) and the other way round.

---

<sup>31</sup>Note that the algorithm only compensates for clock offset; clock drift is not taken into account because we consider it negligible for this type of short-term algorithms.

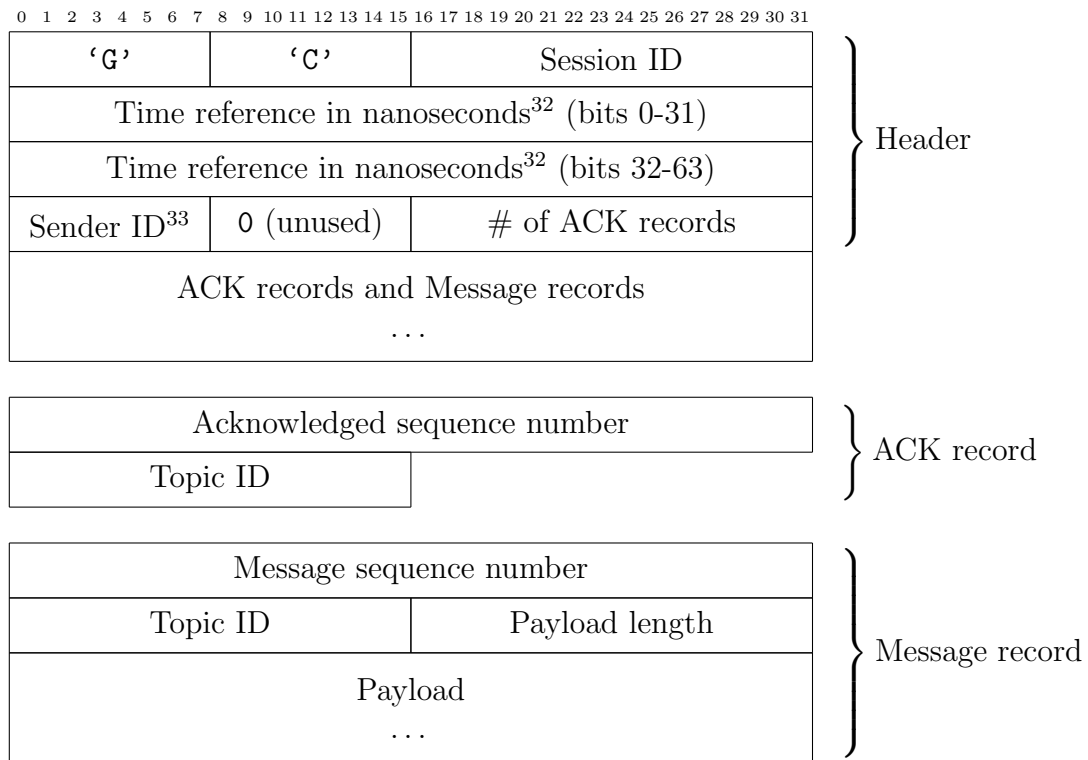


Figure 6.3: Structure of a UDP packet.

**Algorithm 6** UAVs’ Time Difference Update Algorithm

---

▷ Initialisation (only once):  
 $time\_difference \leftarrow nil$

▷ On UDP packet received:  
 $received\_time \leftarrow READ\_TIME\_REFERENCE\_FROM\_RECEIVED\_PACKET()$   
 $local\_time \leftarrow GET\_CURRENT\_LOCAL\_TIME()$   
**if**  $time\_difference = nil$  **or**  $time\_difference > local\_time - received\_time$   
**then**  
 $time\_difference \leftarrow local\_time - received\_time$   
**end if**

---

<sup>32</sup>The Ground Control’s current time, if sent by the Ground Control; or zero, if sent by a UAV.<sup>33</sup>UAVs = 0 ... 254; Ground Control Station = 255 (reserved ID).

## 6.4 Tuning Computer Vision Algorithms

Autonomous UAVs systems usually need to sense the environment in real-time. One of the most complex types of sensors are cameras, whose frames must be processed onboard with computer vision algorithms.

This class of algorithms usually requires a lot of fine-tuning and offline analysis. Furthermore, it is desirable to be able to watch the video stream in real-time, while debugging the High-Level Behaviour and/or vision algorithms themselves, in order to better understand *why* the robot is behaving in a certain way. Frames processed by computer vision algorithms can be annotated for debugging in two ways:

- overlay drawings: these include coloured rectangles, circles and text painted over the original frame;
- printf-like output: textual output describing non-visual data, such as numbers, lists and counters.

We developed a video container format (that we called “Video Log”, `vlog` file extension) capable of storing the compressed video stream, with such debugging information attached, as well as a set of classes to write VLOG files in a multi-threaded way (so that, for instance, a slow video encoder cannot block the image processing loop). A VLOG file is made of a *VLOG header* (which only describes the codec, e.g. MJPEG, H264, ...) followed by a number of variable-length *frame records*, one for each stored frame. Each *frame record* contains the following sub-fields:

- timestamp, in microseconds;
- “keyframe” flag (always set for MJPEG, set only in I-frames for H264);
- encoded frame: original image, as captured from the camera, compressed using the selected codec<sup>34</sup>;
- overlay drawings: serialised representation of paint commands;
- printf-like output: textual output associated with the frame.

---

<sup>34</sup>On the Raspberry Pi and NVIDIA TX1/TX2 platforms, the VLOG writer can use the platform’s hardware-accelerated H264 encoder.

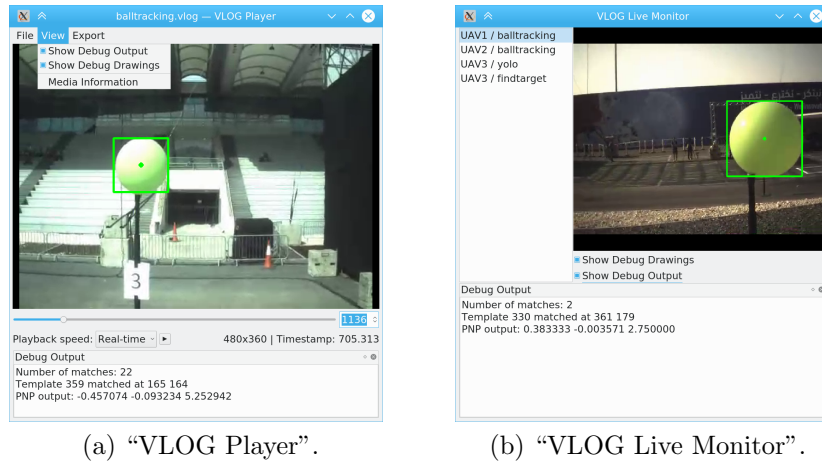


Figure 6.4: VLOG tools can replay (a) and receive live video from UAVs (b).

VLOG files are stored on board and can be extracted from the UAV’s storage (e.g. SD card, internal eMMC) and replayed offline with a custom tool named “VLOG Player” (fig. 6.4a). In addition to regular in-order playback, thanks to the keyframe flag, it can quickly seek to any frame<sup>35</sup>; once a frame is decoded, the tool can optionally overlay the drawings (by deserializing and executing the attached paint commands) and also show the associated printf-like output. It should be noted that, thanks to the fact that the video stream is encoded *without* the drawings, it can easily be exported to a regular video file (without even being re-encoded) and be used as input data for offline training/tuning.

In addition to being stored for later playback, the same data can also be streamed live by the UAV. In this architecture (fig. 6.5), each UAV (more specifically, each instance of the “Logging and Live Streaming” module) periodically announces the names of its stream and codec parameters to a designated receiving station (or, alternatively to the network’s broadcast address). A custom client program (called “VLOG Live Monitor”, fig. 6.4b), which runs on the receiving station, monitors the announcements and presents a list of available streams to the user. When the user selects one of the streams, the receiving station sets up two UDP channels and registers them with the UAV: the first one is used as the Real-Time Protocol (RTP) channel to receive the live video stream; the second one carries the metadata, i.e. overlay drawings and printf-like output. Packets

<sup>35</sup>If the target frame of a seek command is not a keyframe, the tool first seeks to the nearest keyframe before it, then decodes all the frames up to the requested one behind the scenes. Note that, in case of H264, the encoder is configured using the “baseline profile”, so that it never produces B-frames.

received through the metadata channel are synchronised to RTP packets using a common *timestamp*. A keep-alive timer ensures that the UAV stops transmitting in case of loss of link to the receiving station.

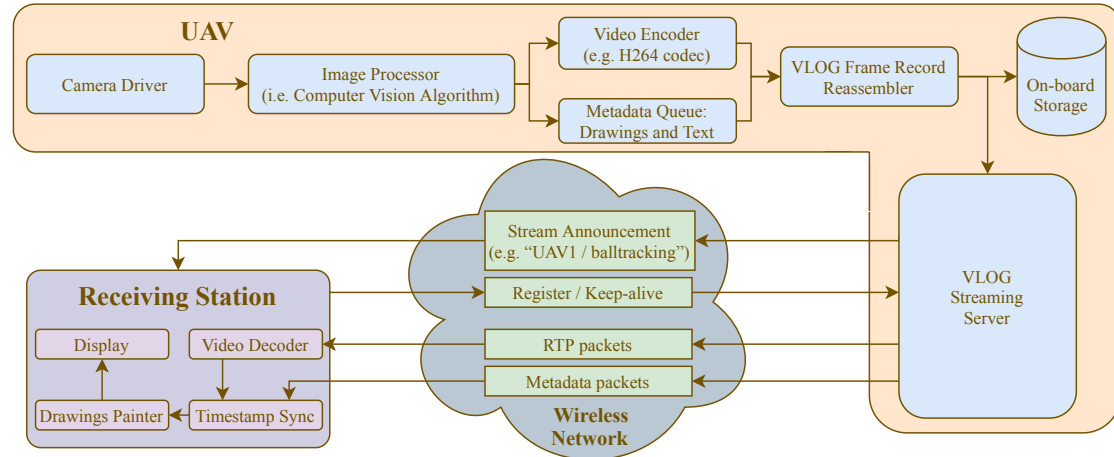


Figure 6.5: The overall “VLOG” architecture.

## 6.5 Simulation

The proposed architecture is integrated with the GzUav simulation framework in two ways:

- the “MAVLink” component can connect to ArduCopter’s TCP channel, instead of the serial port that the real MCU exposes;
- the event loop framework supports transparent replacement of the timer backend, in order to synchronise the simulated periodic tasks and timeouts to the rest of the simulation, instead of the host computer’s clock.

In particular, clock synchronisation is achieved through a process called “GzUav Clock Synchronisation Controller”, which keeps track of all active timeouts created by other processes and subscribes to GzUav’s “start of simulation step” (**Begin-Tick-0**) signal. When one or more timeouts expire (according to GzUav’s clock), the corresponding processes are woken up. However, the “GzUav Clock Synchronisation Controller” process does not send the “end of simulation step” (**End-Tick-0**) signal back to GzUav immediately: it waits until all the other processes

have completed a simulation step or, in other words, until no pending events are left. In detail, for each process, this “consent to signal the end of tick” condition is defined as follows: a process is considered as having completed the current simulation step if it is: i) waiting for a timeout whose expiration timestamp is in the future, ii) waiting for an event that does not depend on time (e.g. a message from the Ground Control Station) or iii) waiting for a combination the previous cases.

Note that, in such a scheme, a simulation step is considered complete only when all the processes are back in *waiting* state. As a consequence, if a process enters a CPU-intensive section without any I/O call, the simulation will not advance at all until that section is completed. This behavior is, in fact, quite common with computer vision (CV) algorithms, which are wakened when a new frame is generated by the simulated camera, and then take relatively long to process it. Without any special consideration, the simulation will progress correctly but very slowly because, as long as at least one CV algorithm is running, the simulation cannot advance. As a workaround for this problem, we introduced a new process state called *detached-timing*, which CV processes enter before starting a CPU-intensive section and leave immediately afterwards<sup>36</sup>. The effect of this state is that it “detaches” the CPU time of the associated process from the “GzUav Clock Synchronisation Controller”, enabling it to run uninterrupted for a preset number  $N$  of simulation steps. If the CPU-intensive section completes earlier than those  $N$  steps, the state will mutate into a regular timeout with the given expiration; conversely, if the simulation completes  $N$  steps but the CPU-intensive task has not finished yet, the simulation will be blocked until it completes. The effect is that no unnecessary latency is introduced and, in every respect, the CPU-intensive section appears to take exactly  $N$  steps. An interesting side-effect is that, if  $N$  is chosen so that it mimics the execution time of the CV algorithm on real hardware, this mechanism turns into an efficient tool to simulate the real system’s CV latency.

---

<sup>36</sup>Any I/O operation is forbidden while in *detached-timing* state.

---

# 7

## MBZIRC 2017 and 2020

The Mohamed Bin Zayed International Robotics Challenge (MBZIRC)<sup>37</sup> is a biennial international competition organised by Khalifa University in Abu Dhabi (UAE), aiming at involving universities from all over the world in the research on real-world robotics tasks.

It is currently at its second edition and we, as the University of Catania, were present in both. For each edition, three *challenges* (each involving unrelated tasks) were defined by the organizers.

In the 2017 edition, our team played only “Challenge 1”, in which the UAV had to autonomously land on a moving vehicle, controlled by the organizers, following an eight-shaped path.

In the 2019 edition (postponed to early 2020 by the organizers), our team played “Challenge 1” and “Challenge 2”. The first challenge involved popping five randomly-placed balloons (anchored to poles at known height) and stealing a ball from an “enemy” UAV controlled by the organizers, following an eight-shaped path; the second challenge consisted in picking bricks and building walls using UAVs and a rover. We also played the “Grand Challenge” (i.e. all challenges at the same time) in cooperation with University of New South Wales, Sidney.

A prototype of the architecture presented in the previous chapter was used for the 2017 edition; the full architecture, as described in the previous chapter, was used in 2020. This chapter presents software developed for both editions, which can be seen as two “case studies” of the aforementioned architecture.

---

<sup>37</sup><http://www.mbzirc.com/>

## 7.1 Landing on a moving vehicle (MBZIRC 2017)

For MBZIRC 2017 Challenge 1, the teams had to design a UAV capable of landing on a moving ground vehicle, driven by a human operator, following an eight-shaped path in an outdoor arena ( $90 \times 60$  m, fig. 7.1a). The approximate shape of the path was known, as well as the speed of the vehicle (15 km/h for the first 8 minutes, then 5 km/h). A visual marker ( $1.5 \times 1.5$  m, fig. 7.1b) was present on the target vehicle, placed on a flat ferromagnetic surface.

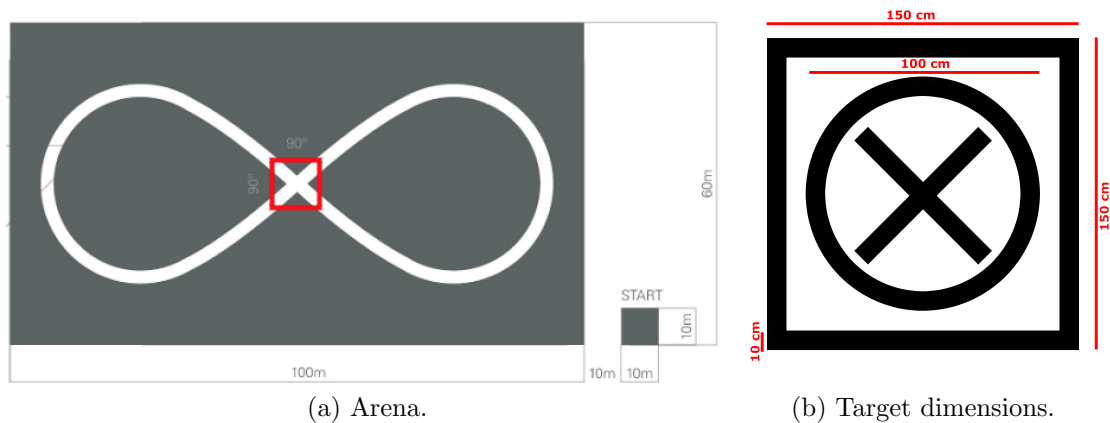


Figure 7.1: Arena and target specifications.

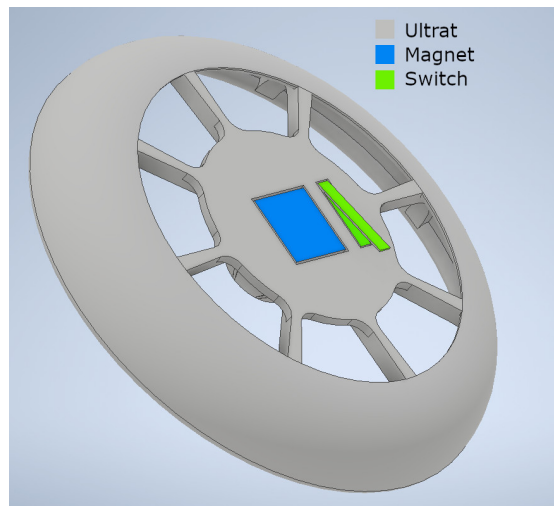


Figure 7.2: 3D printed landing gear.



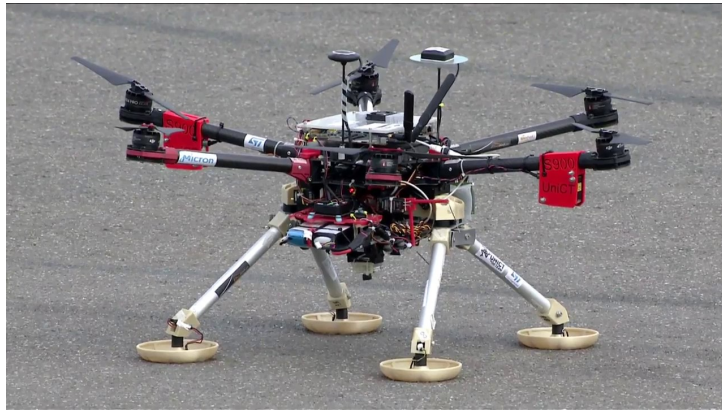


Figure 7.3: Customised DJI S900 for MBZIRC 2017.

We customised a DJI S900 hexacopter, replacing the landing gear with a 3D printed one, with a damper, and magnets and microswitches on the pads (fig. 7.2). A oCam-1MGN-U camera (grayscale, with  $111^\circ \times 65^\circ$  FOV), mounted on a two-axis gimbal programmed to always point down, was connected (to locate and track the visual marker) and an NVIDIA TX1 SoC (quad-core ARM A57 CPU; NVIDIA Maxwell GPU; 4 GB RAM) ran our software, detailed below. A u-blox C94-M8P RTK GPS was connected to the FCU, in order to obtain centimetric precision. The resulting UAV is depicted in fig. 7.3.

The 2017 software architecture was not formally defined, but many pieces that would later become part of the framework described in the previous chapter were already present: “MAVLink” interface, support for in-flight “PID Tuning”, “Image Processor” (decoupled from “Camera Driver”) and “Logging and Live Streaming”.

A vision algorithm, running at around 7 fps, constantly searched the visual marker. In detail, the algorithm was made by combining OpenTLD [64, 65] to detect and track the marker, with the Circle Hough Transform to improve the precision of the resulting coordinates (fig. 7.4). Output 2D coordinates (in image space) are converted to 3D coordinates (in arena space) using camera calibration data, the pinhole model formula and the fact that the distance between the UAV and the marker could be inferred from the GPS altitude. The 3D coordinates are transformed from the camera reference system to the arena reference system with a rototranslation (using knowledge of the current UAV position and heading), and then used to update a Kalman filter estimating the pose of the truck.

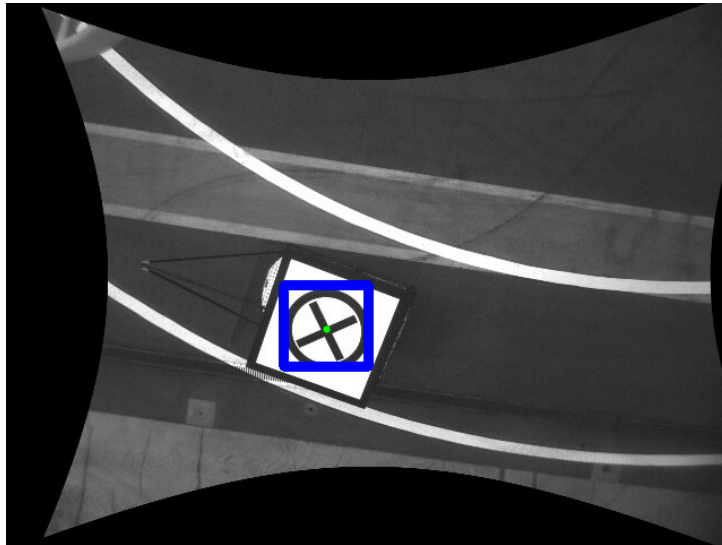
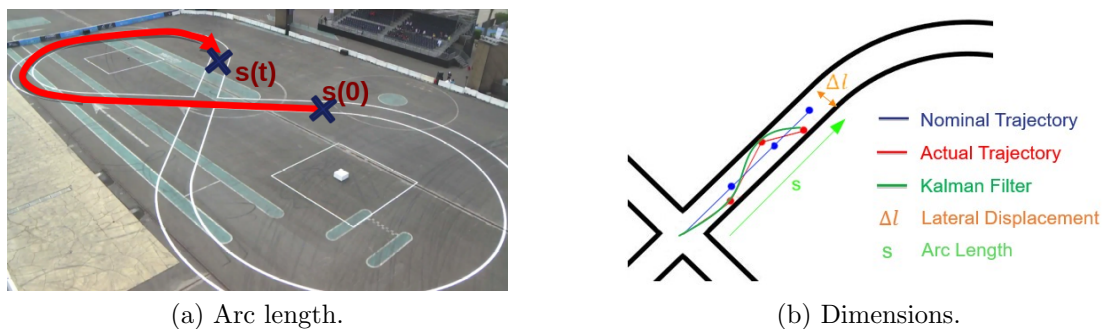


Figure 7.4: Vision Module for MBZIRC 2017: the bounding rectangle produced by OpenTLD (blue) is refined with a Circle Hough Transform (green).



(a) Arc length.

(b) Dimensions.

Figure 7.5: Variables used in Kalman filter.

In particular, the truck position is tracked in terms of arc length (over the expected trajectory, see fig. 7.5a) and lateral displacement (fig. 7.5b). The Kalman Filter has the following state variables:

- $t$  (m): Arc length corresponding to the truck position (more precisely, corresponding to the projection of the truck on the “eight” shape);
- $v_t$  (m/s): Forward/backward speed of the truck (i.e. derivative of  $t$ );

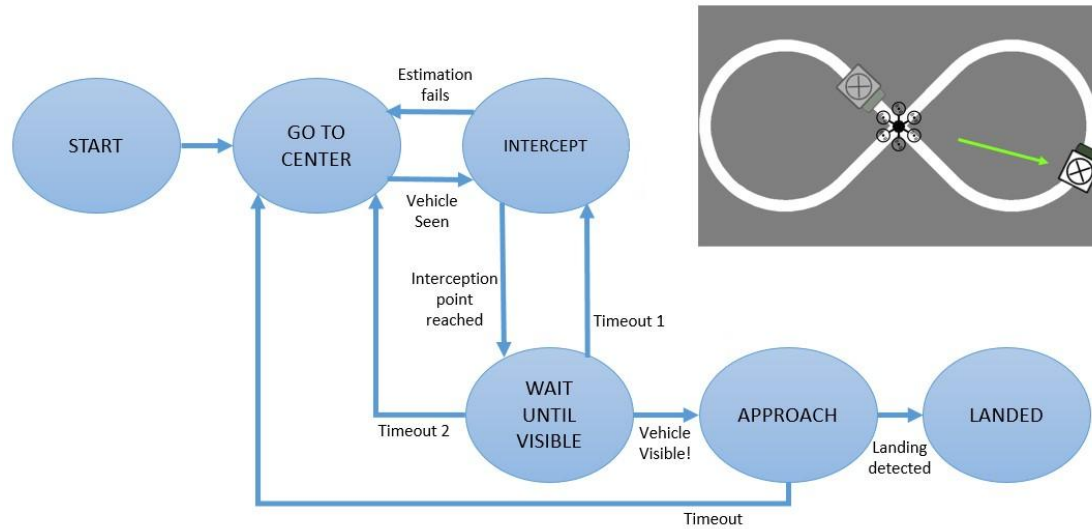


Figure 7.6: Finite State Machine for MBZIRC 2017.

- $l$  (m): Lateral displacement (0 corresponds to the truck being perfectly on the ideal path; for other values, the side is identified by the sign);
- $v_l$  (m/s): Lateral speed (derivative of  $l$ ), usually a vary small value.

The Kalman filter state is initialised when two points are available (we need two points so that the initial speed and direction can be estimated). It is then updated every time a new measured truck position is available, taking into account that the measurement is subject to image processing latency. In particular, the truck coordinates detected by the computer vision are tied to the original image acquisition timestamp: when a new measurement is available, such timestamp is compared to the previous one and the Kalman state is repeatedly *predicted* until it matches the new timestamp. Then, one final *correction* using measured data is performed. However, because of the image processing latency, such timestamps are in the past: the current (and future) truck positions can be extrapolated using basic kinematics formulas from the stored state. This approach is similar to the “Delayed Time Horizon” approach [66] used by PX4 and ArduPilot for the UAV pose.

The high-level behaviour was implemented as a Finite State Machine with the following states (fig. 7.6):

- “Start”: This is the state during the initial take-off;

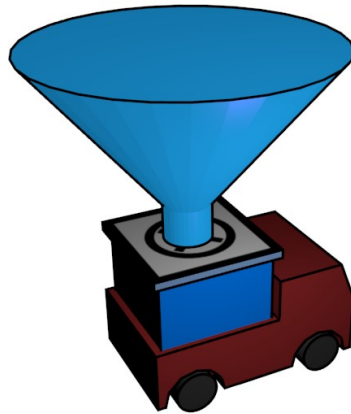


Figure 7.7: Funnel-like descent volume.

- “Go To Centre”: Go to the centre of the area (i.e. the most likely truck location) at 8 m altitude. As soon as the truck is seen and the Kalman filter initialized, transition to the “Intercept” state;
- “Intercept”: Using the extrapolated future positions of the truck and the speed profile of the UAV, go to the most convenient point that is suitable to intercept the truck. When on the interception point, transition to “Wait Until Visible”;
- “Wait Until Visible”: Start to move at half the truck speed (to minimise the speed difference) and search the truck with the camera for up to 12 seconds. When it is seen, proceed to the “Approach” phase; otherwise, go back to “Intercept” (first miss) or “Go To Centre” (second miss);
- “Approach”: Slowly descend until the microswitches detect contact with the truck. In this phase, a PID control loop tries to keep UAV above the truck (in particular, within the volume in fig. 7.7, which is regarded as safe; when outside of that volume the UAV climbs up for safety). During the last centimetres, the target marker is too close and the camera cannot see it in its entirety: for this reason, we allow up to 8 seconds of “open-loop” descent; past such timeout we assume that we lost the tracking and go back to “Go To Center”.
- “Landed”: When at least one of the microswitches touches the truck, this state is entered and all motors are instantaneously shut off.

At the MBZIRC event, our team landed in 140 seconds (fig. 7.8) and ranked 4th out of 24. A detailed description of our solution can be found in [67, 68].



Figure 7.8: Successful landing at the MBZIRC 2017 event.

## 7.2 Popping balloons with two cooperating UAVs (MBZIRC 2020)

For the MBZIRC 2019 edition (actually held in early 2020), our team played “Challenge 1”, “Challenge 2” and the “Grand Challenge”. However, “Challenge 2” was not carried out autonomously; therefore, only “Challenge 1” and the “Grand Challenge” are reported in this document.

For MBZIRC 2020 Challenge 1, the teams could deploy up to three UAVs to carry out two tasks:

1. Pop five randomly-placed green balloons (anchored to poles at a fixed height, around 3.0 m above ground level – fig. 7.9a);
2. Steal a yellow ball from an “enemy” UAV, controlled by the organizers and following an eight-shaped path (fig. 7.9b).

We only achieved the first task, using two fully autonomous UAVs, coordinated by software running on the ground control station<sup>38</sup>.

---

<sup>38</sup>We did not employ direct A2A communication due to technology constraints imposed by the competition’s rules.

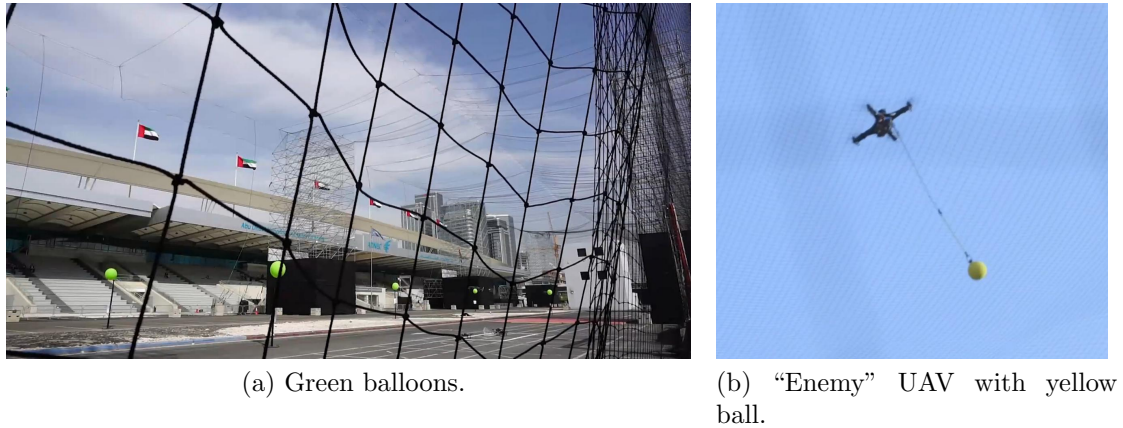


Figure 7.9: Elements of MBZIRC 2020 Challenge 1.

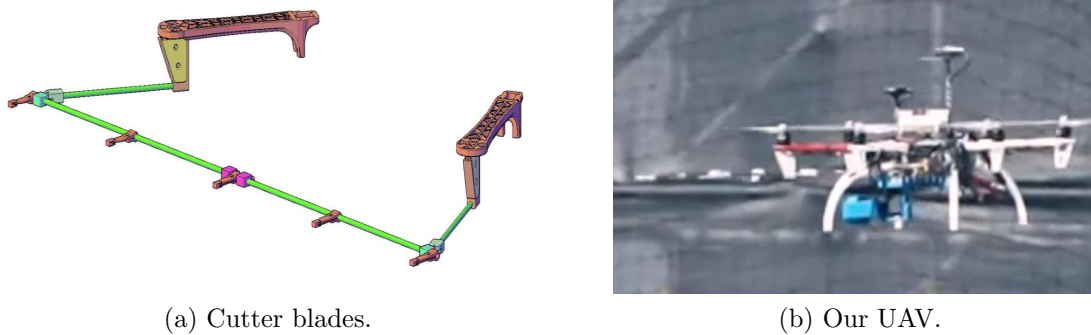


Figure 7.10: Our system to pierce balloons.

We used two DJI F550 quadcopters, equipped with an array of cutters blades on the front side, to pierce the balloons (fig. 7.10a). An oCam-1CGN-U camera ( $39^\circ \times 21^\circ$  FOV), mounted on a two-axis gimbal programmed to always look forward, was mounted along with an NVIDIA TX2 SoC (six-core ARM Denver2+A57 CPU; NVIDIA Pascal GPU; 8 GB RAM), mounted on an Orbitty carrier board (by Connect Tech). The same type of RTK GPS as the one used for MBZIRC 2017 (u-blox C94-M8P) was employed in order to reach centimetric precision. The resulting UAV can be seen in fig. 7.10b.

The system ran on the software architecture described in chapter 6. A custom ground control station software was developed: it displayed markers for the UAVs as well as other application-specific markers overlaid on top of a rectangle representing the MBZIRC arena. The ground control software also takes part in the

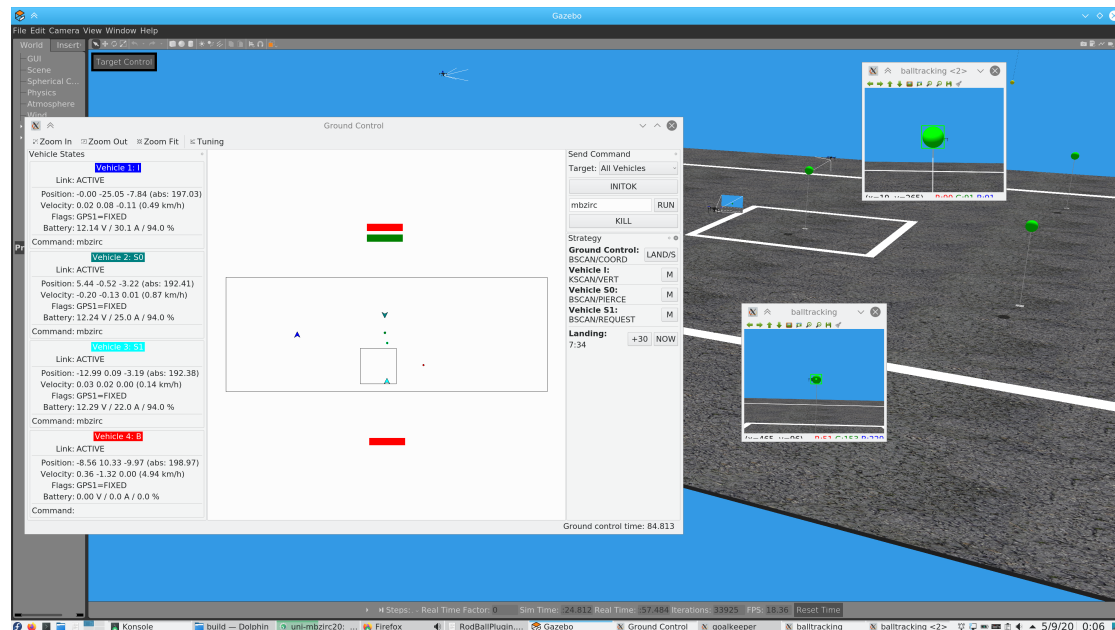


Figure 7.11: MBZIRC 2020 system simulation in the *gzuav* environment.

airspace allocation protocol described below (to avoid mid-air collisions among our UAVs) and offered a GUI to monitor, start and land all the UAVs at the same time. All simulations, which included our three UAVs and the ground control software, were carried out in the *gzuav* environment described in chapter 5 (fig. 7.11).

Three vision algorithms were developed: an algorithm to detect green balloons, another algorithm to detect the yellow ball and a further algorithm to detect the enemy UAV. Since we only succeeded in popping the green balloons, only the first one was actually used in the final solution. The algorithm was based on the LINE [69] algorithm, followed by a colour filter and a Circle Hough Transform to refine the coordinates (fig. 7.12). The 3D position of the balloon was estimated using the 2D size and shape of the ball with the PnP method [70]. Lastly, a temporal filter discarded false positives and a low-pass filter reduced the noise.

The UAV strategy was developed as a Finite State Machine (FSM)<sup>39</sup>, comprising the following states:

<sup>39</sup>Only the strategy for the drones in charge of popping the balloons (that we called “S0” and “S1”) is reported in this document. The third UAV (called “I”) was in charge of stealing the yellow ball and its software was also developed as a different FSM; however, the “I” UAV could not fly at the MBZIRC competition due to technical issues.

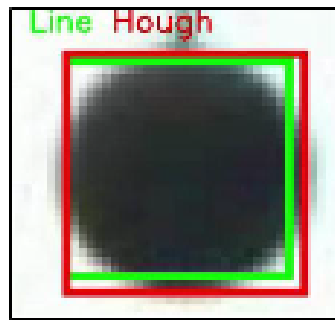


Figure 7.12: Using Circle Hough Transform to refine the result.

- “Preflight”: This is the initial state, in which the UAV stays on the ground with the motors off while waiting for the reception of the start command over the `STATE` topic. When the start command is received from the ground control station, the “Take-off” state is entered;
- “Take-off”: This is the state during the initial take-off. When altitude 5 m is reached, a transition to “Go Side Center” is executed;
- “Go Side Center”: In this state the UAV reaches the midpoint of one of the long sides of the arena rectangle (“S0” goes to a side and “S1” to the opposite one), maintaining the current altitude. The heading is adjusted so that the camera points towards the inner part of the arena;
- “Go Scan Altitude”: The UAV keeps the position on the midpoint but descends to 3 m (“S0”) or 3.5 m (“S1”). When the target altitude is reached, “Blind Scan” is entered;
- “Blind Scan”: In this state, the UAV slowly translates sideways while looking for green balloons and maintaining the current altitude. When a corner of the arena rectangle is reached, the direction of the translation is inverted and the operating altitude is swapped (3 m becomes 3.5 m and vice versa). Whenever a green balloon is detected, a transition to the “Area Allocation Request” is immediately executed. After two full scans (one at 3 m and another one at 3.5 m) the UAV, which is back at the midpoint, transitions to the “Land” state;
- “Area Allocation Request”: In this state, the UAV is on its side of the arena with a green balloon in front of it. Before entering the “Pierce” state, it has to obtain permission from the ground control software in order to enter the arena. This is necessary in order to avoid having both UAVs trying to pierce the same balloon, thus colliding. A simple *airspace allocation* mechanism is



implemented: the UAV sends its position of the `AIRSPACE` topic and stops until it receives the same value in reply<sup>40</sup>. At any time, should the balloon be no longer visible (e.g. if popped by the other UAV while waiting), the request can be cancelled by sending a `NaN` position over the `AIRSPACE` topic;

- “Pierce”: In this state, the UAV moves forward, while adjusting its lateral and vertical position so that the balloon stays horizontally centred and at the same height as the strip of blades. During the last meters, the balloon does not fully fit in the camera’s field of view; for this reason, this state is maintained for 10 seconds after the balloon was last detected. After such timeout (i.e. the tracked balloon was either popped or lost), the UAV moves backwards in order to resume the “Blind Scan” operation and deallocates the area with a `NaN` message over the `AIRSPACE` topic;
- “Land”: In this state, the UAV reaches a pre-designated landing spot, starts a slow descent and, eventually, turns off the motors.

Fig. 7.13 shows the transitions described above. In addition to those, at any time and from any state, a transition to the “Land” state can be commanded by the ground control software with a further message over the `STATE` topic, e.g. in case of emergency.

At the MBZIRC event, we played four times (once for the rehearsal, twice for the competition and one more time for the “Grand Challenge”). We were always able to pop all five balloons, a result that highlights the repeatability of the proposed solution (see table 7.1). Fig. 7.14 shows a popped balloon, as seen by the onboard camera in the “Pierce” state.

We played the “Grand Challenge” (i.e. Challenges 1, 2 and 3 at the same time) in partnership with University of New South Wales, Sidney. We provided the UAV for Challenge 1; they provided the robots for the other challenges.

We ranked 9th (out of 22) at “Challenge 1” and 8th out of 17 at the “Grand Challenge”.

---

<sup>40</sup>The ground control software receives `AIRSPACE` messages from both UAVs and, therefore, knows if they are trying to get too close. If they are sufficiently apart, it replies to both. Otherwise, it only replies to the first one and the other one has to wait.

	N. of burst balloons	Time [s]
Rehearsal	5/5	162
Competition Day 1	5/5	126
Competition Day 2 <sup>41</sup>	5/5	306
Grand Challenge <sup>42</sup>	5/5	262

Table 7.1: Burst of the balloons: task execution time.

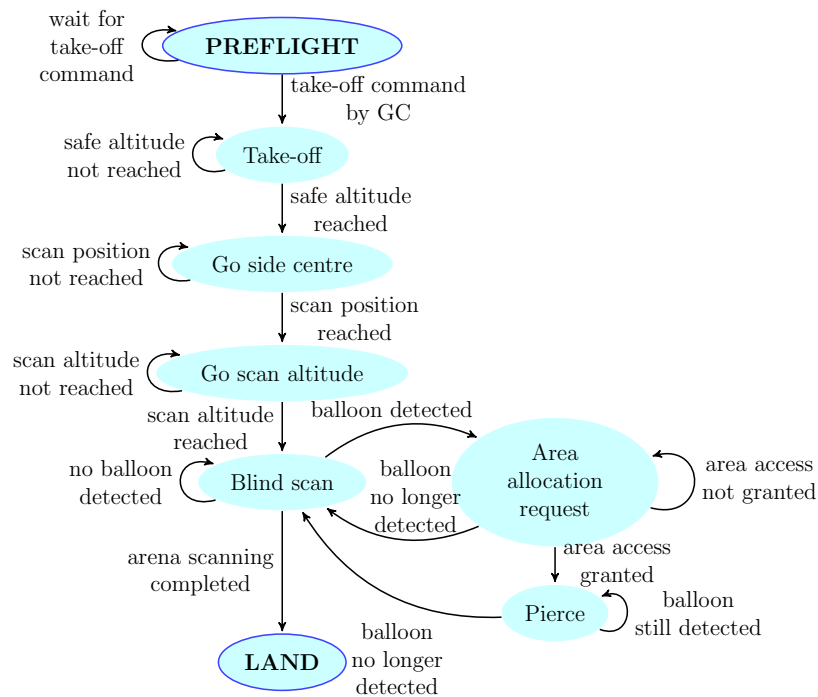


Figure 7.13: Strategy FSM for popping the balloons.

<sup>41</sup>Due to a malfunction in one UAV, only the other UAV was able to pop the balloons. Hence, it took twice the regular time to pop all five balloons. This event also highlighted the robustness of the proposed solution.

<sup>42</sup>Due to additional constraints imposed by the “Grand Challenge”, we could only deploy one UAV to pop the balloons. Hence, in this case too, it took twice the regular time to pop all five balloons.

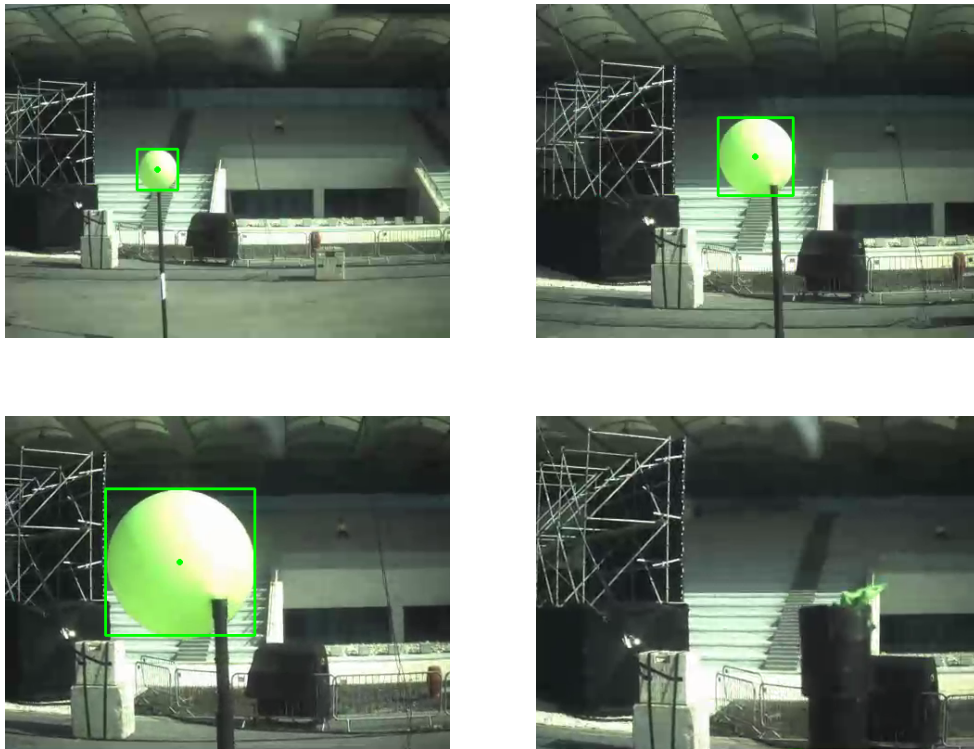
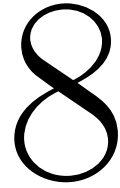


Figure 7.14: A balloon popped during Competition Day 1 of the MBZIRC event.



## Final remarks

In this dissertation, a methodology to develop, validate and implement multi-UAV control algorithms has been presented.

Chapters 3 and 4 presented and validated a distributed fault-tolerant flocking algorithm that solves the area coverage problem, which can be applied in several scenarios. The proposed solution is based on two modules: the first module controls the flocking behaviour, using a bio-inspired approach, and achieves the goal of maintaining a compact “line formation” (which is the ideal type of formation for the intended applications); the second module defines an online distributed protocol, executed repeatedly during a mission, to collectively plan the next useful path (considering that data lost due to failed agents must be re-acquired), which is covered by the flock as a whole immediately afterwards. Furthermore, a new lightweight simulation environment is described in chapter 4, which lets one quickly prototype flocking algorithms in C++. In this environment, we validated the proposed algorithm under a number of conditions, including artificially induced failures. The algorithm proved to be effective and always able to guaranteed successful area coverage, with minimal overhead. The results also highlighted that adding more agents results in shorter mission time until a “steady-state” is reached (in our test environment, it is reached when 25 or more agents are employed).

Chapters 5 and 6 focused on the “toolchain” for the implementers of control algorithms meant to be run on real-world multicopters. We presented two interoperating pieces of software: one to simulate UAVs with wireless networking capabilities and another one, which is actually a framework, to develop autonomous control programs. Programs developed in the proposed environment can then be run, without further adaptations, on real UAVs. In addition to basic control primitives (such as the ability to set waypoints, aim the camera gimbal, operate a gripper),

the framework offers support for many recurring needs in UAV programming, such as fault isolation, inter-agent communication (A2A), customizable ground control station software and A2G/G2A communication, live tuning of PID gains, logging of flight data, integration of computer vision/image processing pipelines for visual servoing (with built-in support for hardware-accelerated live video streaming and logging). Great care has been put in ensuring that all such aspects work consistently in simulation and in the real world (not only API-wise, but also by making the simulated timing of events match the real one). The final result is, therefore, a framework in which a program can be thoroughly and safely tested in simulation, and then deployed to real UAVs only when the developer is confident that it will work as expected.

Lastly, chapter 7 presented two real-world case studies: two autonomous UAV competitions in which the University of Catania was present as a team, and whose programs were developed using the proposed software architecture. In the 2017 edition, we had to land on a moving vehicle that followed an eight-shaped path. The implemented control program processed images in real-time to locate the landing pad. Its 2D coordinates were then transformed into 3D space, filtered and chased by a proper controller overlooked by a finite state machine. For the 2020 edition, the goal was to pop some randomly-located balloons using multiple UAVs. We implemented a solution comprising two UAVs equipped with cameras and blades. A centralised ground station allocated access to common areas in order to prevent UAVs from colliding. In both editions, we recreated all the competition elements in the simulation environment and everything was tested in simulation in advance. As proof of the validity of the proposed approach, we never had any UAV crash due to software bugs, neither at the competition nor during the countless hours of development and field tests. The resulting robots proved competitive, with a 4th place (out of 24 teams) in 2017 and a 9th place (out of 22) in 2020.

The distributed flocking algorithm was also planned to be implemented and validated, as a third case study, with the proposed architecture and its UAV-to-UAV communication capability. Unfortunately, due to the coronavirus pandemic, this activity could not be completed.

## 8.1 Limitations and open issues

In the *gzuv* environment, if the simulated world has many elements (e.g. UAVs, sensors, passive objects), simulations can become very slow. While there are mechanisms in place to keep the simulated clock coherent, it is frustrating for the user to

have to wait and see everything happening in “slow motion”. The design of *gzuv* makes it possible to spread the workload, by running Gazebo’s core, Gazebo’s GUI, ns-3, ArduCopter SITL processes and the user-provided application each, potentially, on a different computational node. We have investigated neither which of those processes requires most CPU time, nor if they block other processes, nor what is the optimal way to distribute them. Future work might try to run *gzuv* in a cluster and benchmark it, in order to address such issues.

Other possible directions regarding *gzuv* could be the integration with alternative simulators, possibly through the definition of an abstract “pluggable” interface, and support for different types of vehicles (e.g. rovers for UGV-UAV cooperation). In this regard, it is interesting to note that the ArduPilot project supports rovers, boats, helicopters and fixed-wing planes too, while exposing the same SITL backend and MAVLink frontend as multirotors: therefore, it would not probably be too hard to add support for such vehicles.

Ns-3 supports many network types. However, each network type has different characteristics that cannot be abstracted (e.g. different internal API within ns-3, different network address format, different data layout, ...). For instance, in order to implement support for simulating our GPS-synchronised TDMA, we had to implement the “firmware” of the microcontroller (i.e. the TDMA mechanism) within ns-3, and then defined a set of commands exposed over *gzuv*’s send/receive primitives, which a module in the user program can call. In other words, for each different network technology, a “proxy” has to be developed within ns-3 and a corresponding “driver” has to be developed within the user program. It would be interesting to add support for other technologies, in addition to our GPS-synchronised 802.15.4, and make them built-in: in this way, *gzuv* could also be used as a tool to evaluate the suitability of a range of network technologies for a given UAV application. Another limitation of ns-3 is that it does not take into account the presence of obstacles, that would affect signal propagation in the real world.

As already mentioned, testing the flocking algorithm with real UAVs (using the proposed architecture) remained an open point. Apart from that, another future expansion might be to try to employ inter-UAV distances (measured using 802.15.4-UWB ranging), to make the algorithm more robust to GPS positioning errors and glitches.

## 8.2 Conclusion

In conclusion, we designed and validated a way to enable the implementation of smarter algorithms for applications with one or more UAVs. Throughout the thesis, we tried to answer the questions raised in the introduction, regarding flocking, fault-tolerance and development methodology.

The contribution of this work extends beyond the presented control algorithms. We tried to prove that it is possible to make programming complex autonomous UAVs easier, thanks to the scalability and modularity of the proposed framework: software written in it runs isolated from other components, benefits from ready-made components offered by the framework and can easily be tested in simulation.

In other words, the proposed approach aimed at enabling the transition to smarter, autonomous and coordinated UAVs, and doing so in a robust, convenient and safe way.

---

## Bibliography

- [1] Habib Ayman, Youkyung Han, Weifeng Xiong, Fangning He, Zhou Zhang, and Melba Crawford. Automated Ortho-Rectification of UAV-Based Hyperspectral Data over an Agricultural Field Using Frame RGB Imagery. *Remote Sensing*, 8(10), 796, 2016.
- [2] Philipp Lottes, Raghav Khanna, Johannes Pfeifer, Roland Siegwart, and Cyrill Stachniss. UAV-based crop and weed classification for smart farming. *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3024–3031, 2017.
- [3] John Tsanakas, Long Duy Ha, and Franck Al Shakarchi. Advanced inspection of photovoltaic installations by aerial triangulation and terrestrial georeferencing of thermal/visual imagery. *Renewable Energy*, 102, pages 224–233, 2017.
- [4] Fabio Leonardi, Fabrizio Messina, and Corrado Santoro. A Risk-Based Approach to Automate Preventive Maintenance Tasks Generation by Exploiting Autonomous Robot Inspections in Wind Farms. *IEEE Access*, 7, pages 49568–49579, 2019.
- [5] Abdulla Al-Rawabdeh, Fangning He, Adel Moussa, Naser El-Sheimy, and Habib Ayman. Using an Unmanned Aerial Vehicle-Based Digital Imaging System to Derive a 3D Point Cloud for Landslide Scarp Recognition. *Remote Sensing*, 8(2), 95, 2016.
- [6] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. *IEEE International Conference on Robotics and Automation (ICRA), Workshop on Open Source Software*, 2009.
- [7] MAVLink extendable communication node for ROS. <https://github.com/mavlink/mavros>.
- [8] Vivek Varadharajan, David St-Onge, Ivan Svogor, and Giovanni Beltrame. A Software Ecosystem for Autonomous UAV Swarms. *International Symposium on Aerial Robotics (ISAR)*, 2017.



- [9] David St-Onge, Vivek Varadharajan, Ivan Svogor, and Giovanni Beltrame. From Design to Deployment: Decentralized Coordination of Heterogeneous Robotic Teams. 7, 51, 2020.
- [10] Carlo Pinciroli and Giovanni Beltrame. Buzz: An extensible programming language for heterogeneous swarm robotics. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3794–3800, 2016.
- [11] Samira Hayat, Evşen Yanmaz, and Raheeb Muzaffar. Survey on Unmanned Aerial Vehicle Networks for Civil Applications: A Communications Viewpoint. *IEEE Communications Surveys and Tutorials*, 18(4), pages 2624–2661, 2016.
- [12] İlker Bekmezci, Ozgur Koray Sahingoz, and Şamil Temel. Flying Ad-Hoc Networks (FANETs): A survey. *Ad Hoc Networks*, 11(3), pages 1254–1270, 2013.
- [13] IEEE Standard for Information technology – Telecommunications and information exchange between systems Local and metropolitan area networks – Specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE Std 802.11-2016.
- [14] IEEE Standard for Local and metropolitan area networks – Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). IEEE Std 802.15.4-2011.
- [15] Antonio Jiménez and Fernando Seco. Comparing Ubisense, BeSpoon, and DecaWave UWB Location Systems: Indoor Performance Analysis. *IEEE Transactions on Instrumentation and Measurement*, 66(8), pages 2106–2117, 2017.
- [16] Yanjun Cao, Chenhao Yang, Rui Li, Alois Knoll, and Giovanni Beltrame. Accurate position tracking with a single UWB anchor. *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2344–2350, 2020.
- [17] Yanjun Cao, Meng Li, Ivan Švogor, Shaoming Wei, and Giovanni Beltrame. Dynamic Range-Only Localization for Multi-Robot Systems. *IEEE Access*, 6, pages 46527–46537, 2018.
- [18] Zigbee Alliance. <https://zigbeealliance.org/>.
- [19] Periklis Chatzimisios, Anthony Boucouvalas, and Vasileios Vitsas. Effectiveness of RTS/CTS handshake in IEEE 802.11a Wireless LANs. *Electronics Letters*, 40(14), pages 915–916, 2004.

- [20] Jens Pilz, Matthias Mehlhose, Thomas Wirth, Dennis Wieruch, Bernd Holfeld, and Thomas Haustein. A Tactile Internet demonstration: 1ms ultra low delay for wireless communications towards 5G. *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 862–863, 2016.
- [21] Yiming Quan, Xiaolin Meng, Lei Yang, and Scott Stephenson. Network RTK GNSS Quality Assessment. *European Navigation Conference (ENC)*, 2013.
- [22] DJI Flight Simulator. <https://www.dji.com/simulator>. 2018.
- [23] Carlo Pinciroli, Vito Trianni, Rehan O’Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Timothy Stirling, Alvaro Gutiérrez, Luca Maria Gambardella, and Marco Dorigo. ARGoS: A modular, multi-engine simulator for heterogeneous swarm robotics. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5027–5034, 2011.
- [24] Nathan Koenig and Andrew Howard. Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2149–2154, 2004.
- [25] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. PX4: A Node-Based Multithreaded Open Source Robotics Framework for Deeply Embedded Platforms. *IEEE International Conference on Robotics and Automation (ICRA)*, pages 6235–6240, 2015.
- [26] ArduPilot: Open Source Autopilot. <http://ardupilot.org/>.
- [27] Open Dynamic Engine. <http://www.ode.org/>.
- [28] Erwin Coumans. Bullet physics library. <http://bulletphysics.org>. 2013.
- [29] Michael A. Sherman, Ajay Seth, and Scott L. Delp. Simbody: multibody dynamics for biomedical research. *IUTAM Symposium on Human Body Dynamics*, 2, pages 241–261, 2011.
- [30] Jeongseok Lee, Michael X. Grey, Sehoon Ha, Tobias Kunz, Sumit Jain, Yuting Ye, Siddhartha S. Srinivasa, Mike Stilman, and C. Karen Liu. DART: Dynamic Animation and Robotics Toolkit. *Journal of Open Source Software*, 3(22), 500, 2018.
- [31] Anton Babushkin. jMAVSim - Simple multirotor simulator with MAVLink protocol support. <https://github.com/PX4/jMAVSim>. 2014.

- [32] X-Plane Flight Simulator. <https://www.x-plane.com/>.
- [33] Jon Berndt. JSBSim: An Open Source Flight Dynamics Model. *AIAA Modeling and Simulation Technologies Conference and Exhibit*, 2004.
- [34] Shital Shah, Debadepta Dey, Chris Lovett, and Ashish Kapoor. AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. *Field and Service Robotics (FSR)*, pages 621–635, 2017.
- [35] Pooyan Fazli, Alireza Davoodi, and Alan K. Mackworth. Multi-robot repeated area coverage. *Autonomous robots*, 34(4), pages 251–276, 2013.
- [36] Jean-Claude Latombe. Exact Cell Decomposition. *Robot Motion Planning*, pages 200–247. Springer, 1991.
- [37] Jean-Claude Latombe. Approximate Cell Decomposition. *Robot Motion Planning*, pages 248–294. Springer, 1991.
- [38] Julien Schleich, Athithyaa Panchapakesan, Grégoire Danoy, and Pascal Bouvry. UAV fleet area coverage with network connectivity constraint. *ACM international symposium on Mobility management and wireless access (MobiWac)*, pages 131–138, 2013.
- [39] Vivek Shankar Varadharajan, Bram Adams, and Giovanni Beltrame. Failure-Tolerant Connectivity Maintenance for Robot Swarms. *Computing Research Repository (CoRR)*, 2019.
- [40] M. Bernardine Dias and Anthony Stentz. A market approach to multirobot coordination. 2000.
- [41] M. Bernardine Dias, Robert Zlot, Nidhi Kalra, and Anthony Stentz. Market-based multirobot coordination: A survey and analysis. *Proceedings of the IEEE*, 94(7), pages 1257–1270, 2006.
- [42] Craig Reynolds. Flocks, Herds and Schools: A Distributed Behavioral Model. *International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 25–34, 1987.
- [43] Gábor Vásárhelyi, Csaba Virágh, Gergő Somorjai, Norbert Tarcai, Tamás Szörényi, Tamás Nepusz and Tamás Vicsek. Outdoor flocking and formation flight with autonomous aerial robots. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2014.

- [44] I-Kuai Hung, Daniel Unger, David Kulhavy, and Yanli Zhang. Positional Precision Analysis of Orthomosaics Derived from Drone Captured Aerial Imagery. *Drones*, 3(2), 46, 2019.
- [45] Massimiliano De Benedetti, Fabio D’Urso, Fabrizio Messina, Giuseppe Pappalardo, and Corrado Santoro. Self-Organising UAVs for Wide Area Fault-tolerant Aerial Monitoring. *Workshop “From Objects to Agents” (WOA)*, pages 135–141, 2015.
- [46] Massimiliano De Benedetti, Fabio D’Urso, Fabrizio Messina, Giuseppe Pappalardo, and Corrado Santoro. UAV-based Aerial Monitoring: A Performance Evaluation of a Self-Organising Flocking Algorithm. *IEEE International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pages 248–255, 2015.
- [47] Noury Bouraqadi and Arnaud Doniec. Flocking-Based Multi-Robot Exploration. *National Conference on Control Architectures of Robots*, volume 4, pages 78–86, 2009.
- [48] Corrado Santoro. How does a Quadrotor fly? A journey from physics, mathematics, control systems and computer science towards a “Controllable Flying Object”. 2014.
- [49] Cleanflight: Open-Source flight control system. <https://github.com/cleanflight>.
- [50] Massimiliano De Benedetti, Fabio D’Urso, Fabrizio Messina, Giuseppe Pappalardo, and Corrado Santoro. 3D Simulation of Unmanned Aerial Vehicles. *Workshop “From Objects to Agents” (WOA)*, pages 7–12, 2017.
- [51] Massimiliano De Benedetti, Fabio D’Urso, Giancarlo Fortino, Fabrizio Messina, Giuseppe Pappalardo, and Corrado Santoro. A Fault-tolerant Self-organizing Flocking Approach for UAV Aerial Survey. *Journal of Network and Computer Applications (JNCA)*, 96, 2017.
- [52] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill Licea-Kane. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013.
- [53] Qt. <https://www.qt.io/>.
- [54] Moses Bangura, Marco Melega, Roberto Naldi, and Robert Mahony. Aerodynamics of Rotor Blades for Quadrotors. *Physics.flu-dyn*, 2016.

- [55] Fabio D’Urso, Corrado Santoro, and Federico Fausto Santoro. Integrating Heterogeneous Tools for Physical Simulation of multi-Unmanned Aerial Vehicles. *Workshop “From Objects to Agents” (WOA)*, pages 10–15, 2018.
- [56] Fabio D’Urso, Corrado Santoro, and Federico Fausto Santoro. An integrated framework for the realistic simulation of multi-UAV applications. *Computers and Electrical Engineering (CEE)*, 74, pages 196–209, 2019.
- [57] George F. Riley and Thomas R. Henderson. The ns-3 Network Simulator. *Modeling and Tools for Network Simulation*, pages 15–34. Springer, 2010.
- [58] András Varga and Rudolf Hornig. An overview of the OMNeT++ simulation environment. *International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SimuTools)*, 60, 2008.
- [59] Sabine Hauert, Severin Leven, Maja Varga, Fabio Ruini, Angelo Cangelosi, Jean-Christophe Zufferey, and Dario Floreano. Reynolds flocking in reality with fixed-wing robots: Communication range vs. maximum turning rate. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5015–5020, 2011.
- [60] Laurent Ciarletta, Adrien Guenard, Yannick Presse, Virgine Galtier, Ye-Qiong Song, Jean-Christophe Ponsart, Samir Aberkane, and Didier Theil-liol. Simulation and platform tools to develop safe flock of UAVs: a CPS application-driven research. *International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 95–102, 2014.
- [61] Michal Kudelski, Luca M. Gambardella, and Gianni A. Di Caro. RoboNetSim: An Integrated Framework for Multi-robot and Network Simulation. *Robotics and Autonomous Systems (ROAS)*, 61(5), 14, pages 483–496, 2013.
- [62] MAVLink Micro Air Vehicle Communication Protocol. <http://mavlink.org/>.
- [63] DroneKit: Developer Tools for Drones. <http://python.dronekit.io>.
- [64] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Tracking-learning-detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(7), pages 1409–1422, 2012.
- [65] OpenTLD. <https://www.gnebehay.com/tld/>.
- [66] Alireza Khosravian, Jochen Trumpf, Robert Mahony, and Tarek Hamel. Recursive Attitude Estimation in the Presence of Multi-rate and Multi-delay Vector Measurements. *American Control Conference (ACC)*, 2015.

- [67] Sebastiano Battiato, Luciano Cantelli, Fabio D’Urso, Giovanni Maria Farinella, Luca Guarnera, Dario Guastella, Carmelo Donato Melita, Giovanni Muscato, Alessandro Ortis, Francesco Ragusa, and Corrado Santoro. A System for Autonomous Landing of a UAV on a Moving Vehicle. *Image Analysis and Processing (ICIAP)*, pages 129–139, 2017.
- [68] Luciano Cantelli, Dario Guastella, Carmelo Donato Melita, Giovanni Muscato, Sebastiano Battiato, Fabio D’Urso, Giovanni Maria Farinella, Alessandro Ortis, and Corrado Santoro. Autonomous landing of a UAV on a moving vehicle for the MBZIRC. *International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines (CLAWAR)*, pages 197–204, 2017.
- [69] Stefan Hinterstoisser, Stefan Holzer, Cedric Cagniart, Slobodan Ilic, Kurt Konolige, Nassir Navab, and Vincent Lepetit. Multimodal templates for real-time detection of texture-less objects in heavily cluttered scenes. *International Conference on Computer Vision (ICCV)*, pages 858–865, 2011.
- [70] Martin A. Fischler and Robert C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Communications of the ACM*, 24(6), pages 381–395, 1981.