Received: 8 December 2022 | Revised: 26 January 2023 | Accepted: 2 February 2023

DOI: 10.1049/csy2.12083

IET Cyber-Systems and Robotics

ORIGINAL RESEARCH

A novel distributed architecture for unmanned aircraft systems based on Robot Operating System 2

Lorenzo Bianchi | Daniele Carnevale | Fabio Del Frate | Roberto Masocco | Simone Mattogno | Fabrizio Romanelli | Alessandro Tenaglia

Department of Civil Engineering and Computer Science Engineering, University of Rome "Tor Vergata", Rome, Italy

Correspondence
Fabrizio Romanelli.
Email: fabrizio.romanelli@uniroma2.it

Funding information
Ministero dell'Università e della Ricerca, Grant/Award Numbers: 2017YKXYXJ, 2020RTWES4

Abstract

A novel distributed control architecture for unmanned aircraft system (UASs) based on the new Robot Operating System (ROS) 2 middleware is proposed, endowed with industrial-grade tools that establish a novel standard for high-reliability distributed systems. The architecture has been developed for an autonomous quadcopter to design an inclusive solution ranging from low-level sensor management and soft real-time operating system setup and tuning to perception, exploration, and navigation modules orchestrated by a finite-state machine. The architecture proposed in this study builds on ROS 2 with its scalability and soft real-time communication functionalities, while including security and safety features, optimised implementations of localisation algorithms, and integrating an innovative and flexible path planner for UASs. Finally, experimental results have been collected during tests carried out both in the laboratory and in a realistic environment, showing the effectiveness of the proposed architecture in terms of reliability, scalability, and flexibility.

KEYWORDS
navigation, robot perception, slam (robots), unmanned aerial vehicle

1 | INTRODUCTION

Robotic systems are complex interconnections of actuators and sensors on a mechanical structure with electronic devices managed by dedicated software. A robot needs both a hardware platform designed to perform tasks and a complex software architecture that embeds algorithms that allow it to act, possibly, autonomously. In the past, while designing a robot, developers often considered a single onboard computer to act as a data processing unit and supervisor, without focussing much on the software architecture driving it all. Robotics software architectures [1] strive to contain many complex systems by providing established development structures, guiding developers towards maintainable and robust software solutions that also facilitate construction starting from existing building blocks. The design of such complex tools is a crucial challenge for both developers and roboticists [2]. A key strategy to create an effective robot software architecture consists of identifying abstractions that allow us to deal with groups of components in similar ways. Along with this strategy, the need for modularity arises from the heterogeneity of the functions and algorithms running in parallel in a robot system [3]. In particular, both abstraction and modularity, together with the use of multiple processing units, lead to a distributed system within a single robot [4]. Real-time constraints, which are mandatory in some robotic platforms, are often not satisfied for distributed systems; see for example, Refs [5–7].

To address these limitations, a standard distributed framework for the development of robotic software, also called middleware [8], is necessary. Robot Operating System (ROS) [9] is a set of tools and libraries for developing modular robotic functions that interact with each other and communicate in a distributed multiprocess environment [10]. ROS is also equipped with a set of tools to manage software builds and deployment, as well as development and testing. However, ROS does not satisfy real-time constraints and does not guarantee compliance with deadlines, process synchronisation and is not natively fault-tolerant. The limitations mentioned above lead to the conclusion that ROS is not suitable for real-time embedded applications. Such limitations have motivated the robotics community to propose ROS 2 [11]. The second version considers new use cases: cross-platform, soft real-time systems, small embedded platforms, and lossy networks. To satisfy soft real-time requirements, it has been redesigned to improve Application Programing Interfaces (APIs) and incorporate new communication middleware technologies such as Data Distribution Services (DDS) [12, 13], WebSockets and ZeroMQ. In ROS 2, the DDS has superseded the ROS custom transport system [14], introducing an end-to-end middleware as an industry standard communication system. The DDS provides reliable publish/subscribe transport that has been made available at several levels, allowing the developer to use various Quality of Service (QoS) configurations (e.g., deadlines, reliability, and durability) and ensure scalability. Furthermore, depending on the implementation, DDS can provide solutions for real-time environments and meet the safety, security, scalability, and fault tolerance requirements in distributed systems.

In this article, we describe a distributed architecture for modular robotic systems based on ROS 2. According to Ref. [15], the usage of ROS 2 and DDS should be subject to extra time costs to convert application messages to meet DDS standards, which should be a huge limitation for both scalability and communication capabilities. This problem is discussed and addressed below, essentially showing how the application context of tools such as ROS 2 and the DDS does not suffer from such limitations. Furthermore, the architecture presented here supports security requirements such as authentication, access control, and cryptographic operations, intrinsically shipped with the DDS; reaching the same level of security within a ROS context is still a matter of study [16]. The architecture is designed for an autonomous quadcopter that has been developed for the 2021 Edition of the Leonardo Drone Contest, a robotics competition hosted by Leonardo S.p.A. in which custom-built unmanned aircraft system (UASs) have to perform a series of complex tasks autonomously. Such tasks range from localisation and flight control to environment mapping and autonomous navigation.

Novel contributions. There are multiple innovative contributions in this study. After the problem of designing a complex, distributed robotics architecture is discussed and new tools aimed at its solution are introduced, such tools are applied to the design and realisation of the architecture of an autonomous UAS, showing and evaluating their effectiveness and reliability in a real scenario, which also fits within a specific and active field of research. Finally, some modules that constitute such architecture represent novel contributions within their respective application scopes: the design and implementation of a flexible path planning algorithm and the application and evaluation of a Visual-SLAM system, both on an UAS platform built on a modular, distributed control framework, and embedded hardware.

The article is organised as follows. Section 2 presents the current state of the art in distributed robotic architectures, focussing on middleware such as ROS 2. Section 3 describes the implementation of this new conceptual model in an autonomous quadcopter, focussing on the design of both the architecture and the modules. Section 4 shows some experimental results, and concluding remarks are drawn in Section 5, highlighting the main features of the new framework.

2 | RELATED WORK

Autonomous systems are currently establishing an increasingly important presence. As humans, we rely on many aspects of our lives on the help offered by various kinds of such systems, and the variety of tasks for which they are designed is getting wider each day. Moreover, they are often expected to deal with tasks that require little or no input from a human operator, processing large amounts of data. The new challenges posed by their development can be overcome by implementing efficient hardware and software architectures, which would allow such systems to communicate easily when required and would assist the robotics developer during all stages of design and implementation. It also appears intuitive how such architectures should embed modules divided among different levels: lower ones, intrinsically coupled to the machine being operated, and upper ones which include supervision logics, data analysis algorithms, communication schemes, and human interaction capabilities. They would also help with the problems arising from the need to connect different systems in order to perform collaborative tasks or share information over a network, which is a scenario that may become the norm in the near future. This set of problems has gained much interest in the past 10 years, in part because of the recent availability of the computational power required to tackle them. Robotics and control theory fall under the subject of distributed control. A survey can be found in Ref. [17], and a summary focussing on mobile robots could be found in Ref. [18].

A possible set of tools for this job is introduced in this section, which are the fundamental building blocks of the application example described in Section 3. As clarified from both the following description and the practical example, such tools actually solve the problems of interest presented here.

2.1 | The DDS

A recurring choice in designing robots, drones, and autonomous systems in general is to equip them with relatively powerful on-board electronics to handle different tasks. Today, very complex embedded systems are available as full Systems-on-Chips (SoCs) ready to be integrated and easily programed. Depending on the complexity of the system itself, there could be many of them. It is usual to delegate low-level control algorithms and operations to small, real-time SoCs, while higher-level logics and computationally heavier tasks are executed on more general-purpose and powerful systems. Therefore, the problem of easily establishing a connection between them immediately arises. Communication in particular is a recurring problem in the design of robots: The situation in which one has to establish data transmissions between different hosts is often dealt with in various ways each time. This is one of many critical issues that a developer always has to face and solve rather than focus on the actual development of the core functionalities of the system. These issues can now be solved by employing the full potential of small general-purpose computers embedded in autonomous systems.

Figure 1 shows the classical organisation of the software installed on a general-purpose computer, taken from Ref. [19]. The inverted pyramid analogy exemplifies how it all relies on tools such as compilers and assemblers. Then the architecture starts with what we can identify as an operating system and a set of device drivers and system APIs, ending with common application software at the upper level. There is an intermediate level that could play a pivotal role in the aforementioned purposes: The *middleware*, a category of software that offers additional services to those made available by the operating system.

If the end goal is to easily develop and deploy a control architecture distributed among many systems connected to a network, then one kind of middleware could be very useful: DDS. A DDS is a publish-subscribe middleware that handles communications and data transmissions between real-time systems. Its definition follows an open standard, redacted by the Object Management Group consortium; the latest version can be found in Refs [20, 21]. The main services offered by a DDS are as follows:

- transparent serialisation and deserialisation of data packets;
- possibility to easily define custom data packet formats, named *interfaces*;
- enforcement of QoS policies between transmissions, carried out on channels named *topics*;
- automatic network discovery and configuration.

The DDS represents a solution to the problem of efficient interprocess communication among many software and hardware modules, giving the resulting architecture a natively distributed organisation.

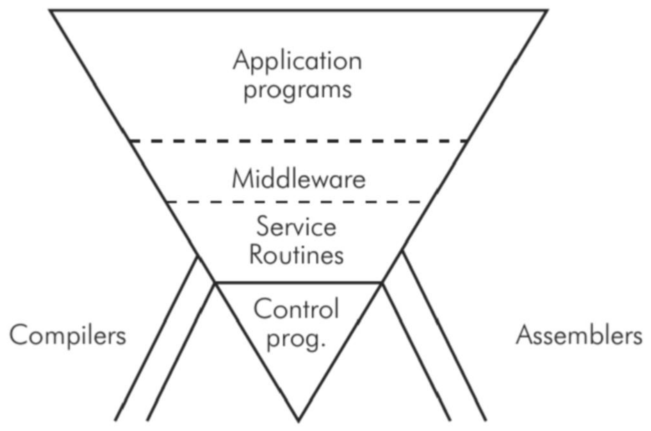

FIGURE 1 Software organisation in a general-purpose system.

2.2 | The ROS 2 robotics middleware

In 2007, Willow Garage, a spin-off of the Stanford Artificial Intelligence Laboratory, released an open-source middleware geared to robotics applications named *ROS*. Since then, ROS has become widely adopted in the robotics industry, with hundreds of software packages created by the community. From an architectural point of view, ROS enables the robotic software developer to package it in different modules, each capable of communicating with the others using specific event-driven semantics, eventually over a network.

The aim of ROS 2, as explained in Refs [22, 23], is to improve on that basis by relying almost entirely on an underlying DDS. This simplifies many aspects of the network of ROS 2-enabled applications that can be created and drastically automates configuration. Moreover, it offers additional client-server communication abstractions that are still based on the DDS layer. Inside a ROS 2 application, the entry point to the DDS layer is usually an object called *node*. It is generally described as an operating unit that executes jobs on the system in which it resides, although this is not always the case: At the very heart, a node is a data structure meant to hold other objects needed to interact with the DDS and ROS 2 services. The application code can simply access the node while doing its job, the jobs can be entirely coded inside callbacks handled by the event-based ROS 2 schedulers, or a mixture of both could be employed. Equally important features of ROS 2 range from the management of software configuration parameters to that of startup and shutdown configurations.

The best feature of ROS 2 is how it can ease the development and deployment of distributed control architectures by default, automating much of the setup and making communications satisfy QoS requirements. All that is needed is an Internet Protocol network.

2.3 | Overall architecture performance

In this section, a survey on the performance evaluation of robotic communication middleware is presented (see the work in Refs [15, 24] for further references).

ROS implementations do not support priority and synchronisation among nodes. Therefore, ROS cannot be considered an appropriate platform for multi-tasking environments in real-time. A priority-based message transmission mechanism has been presented to reduce the execution time and the time variance of high-priority ROS nodes and a synchronisation mechanism to harmonise multiple ROS nodes running at different frequencies [25]. ROS 2 overcomes several limitations of ROS, but as communication middleware, it introduces communication latency caused by message (de)serialisation, (de)conversion, and transport, as illustrated in extensive work [26]. To satisfy the requirements of the underlying DDS, (de)conversion is a required process that is used to transform messages from the application layer to the DDS layer. This process has been identified as responsible for most of the middleware time cost (89.1% when handling multidimensional arrays with different data types). Furthermore, the time cost of (de)conversion is linearly dependent on the complexity of the message data structure.

The performances of ROS and ROS 2 have been compared in terms of throughput, end-to-end latency, memory consumption, and number of threads in Ref. [14]. In this work, intra-process and interprocess communication in a single-machine and two-machine Ethernet network has been evaluated. In particular, different DDS implementations (such as FastRTPS, Connext, and OpenSplice) have been considered together with different QoS policies.

In the proposed architecture, a set of good practices have been implemented in order to reduce the overhead of the DDS layer. Specifically:

- the usage of multidimensional arrays with different data types has been avoided in order to reduce the (de)conversion overhead, thus reducing the latency of the DDS layer;
- the message data structure has been kept as simple as possible on all topics;
- the QoS policies have been selected in order to better fit the latency needs of the various applications, according to the studies summarised in Ref. [14].

It has been observed that a ROS 2-based architecture provides more robust network connectivity and better scalability with respect to the network size. At the same time, ROS 2 allows robotic applications to be modular and extendable; however, this comes with an extra time cost because of the (de)conversion of application and DDS messages. Even with a proper setup of the DDS communication infrastructure, as stated in the previous points, the performances, in terms of time cost, of the presented architecture can be assessed as being slower than those of an implementation without a DDS layer, as expected, but this does not substantially compromise the performances needed in the presented application context. The actual systems that make up the main target of ROS 2 and the DDS, as illustrated in Section 2, usually have enough resources to make such performance hit negligible under nominal conditions, as the case study in Section 3 shows.

3 | APPLICATION TO AN AUTONOMOUS QUADCOPTER

In Section 2, the DDS and ROS 2 middleware were introduced as powerful tools enabling developers to easily craft and deploy sets of applications finalised to the execution of multiple tasks in an autonomous system. To grasp not only this ease but also the intrinsic modularity and hierarchical structuring that can be achieved, a specific use case is presented in the current section.

3.1 | Multi-level design of an autonomous quadcopter

The authors, part of the University of Rome 'Tor Vergata' team, took part in the 2021 edition of the Leonardo Drone Contest hosted by Leonardo in Turin, Italy, in September 2021. In this competition, six Italian universities deployed their drones to complete a complex mission. The drones had to fly in a delimited indoor field in which many different obstacles, 10 landing pads, and three mobile robots were placed. The resulting field configuration was made known to all participants as part of the rules, and the landing pads were delimited by visual markers. The tasks each drone had to perform were as follows:

1. Take-off from a given pad.
2. Explore the environment to locate one of the three mobile robots on the ground.
3. Take pictures of the robot and send them to an operator, who will then transmit a sequence of landings to perform on the pads.
4. Navigate from pad to pad, landing on each and then taking off again until the sequence is completed.

The operator was allowed to use a personal computer (PC), named the Ground Control Station (GCS), to start the mission and monitor the drone. The sequence of landings had to be decided in order to maximise a score, given by the sum of all the points assigned to each of the pads upon which a successful landing was performed. The distribution of points among the different pads was printed on the robot that had to be found on the ground. A series of design limitations were also enforced for all teams: It was not possible to use any type of Global Navigation Satellite System platform, and the use of Light Detection and Ranging sensors was also forbidden, together with the placement of any type of fixed sensor or motion capture system on the ground.

In order to make each of these operations possible, a proper software architecture has been designed and implemented on sophisticated hardware. According to the specifications set above, an autonomous quadcopter that solves the Leonardo Drone Contest 2021 mission qualifies as a system whose complexity requires, and represents a suitable test framework for the tools introduced in Section 2. On both the hardware and the software side, tasks have been split between multiple levels, which also accounts for the different critical issues they pose. Figure 2 shows the resulting organisation of the various software modules among different functional layers. The three rows host different kinds of packages: the lowest row is made of soft real-time modules that are fundamental for the execution of all flight operations; the middle row hosts higher-level, slower algorithms that execute each of the mission tasks; and the highest row is for the supervision logic, implemented as a Finite-State Machine (FSM) and the GCS. The aim of this organisation was to arrive at a set of ROS 2 modules that performed each task, relying on the functionality offered by the other modules.

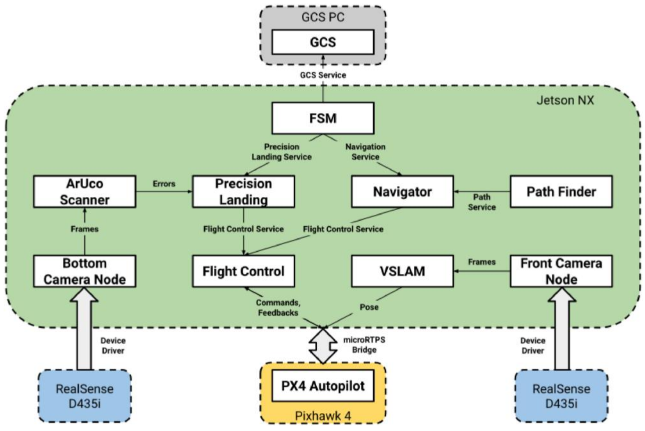

FIGURE 2 Multi-level software and hardware architecture scheme. Thin arrows indicate communications established with ROS 2 interfaces, and thick arrows represent dedicated software interfaces. White boxes represent ROS 2 nodes. ROS, Robot Operating System.

This enables higher levels to not care about how basic operations (e.g., take-off, landing, and turns) are performed, since they are fully abstracted.

3.2 | Low level design

The division in functional layers had to be reflected in the hardware because of the very different levels of criticality of the various operations: for example, relatively fast tracking algorithms and slow supervision logics had to coexist and communicate with flight control subsystems. Also, the lowest level of the aforementioned software architecture is inevitably tightly coupled with the hardware it is implemented on, in order to effectively abstract it from the higher ones. To solve these issues, a two-level hardware architecture was implemented: the lower level is entirely dedicated to real-time flight operations and drone control, while the higher level executes all the mission tasks, interfacing itself with the other to issue commands and obtain feedback. Also, the higher level is the one that actually hosts and runs a distributed software architecture, being in charge of localisation, navigation, and supervision, while flight stabilisation and control are better handled by an embedded, monolithic, and dedicated real-time system.

3.2.1 | Linux-based high-level SoC and open-source real-time flight controller

Figure 3 shows the hardware architecture that has been deployed on the final quadcopter. It is made up of the following modules:

- Holybro Pixhawk 4 real-time flight controller;
- Nvidia Jetson Xavier NX SoC.

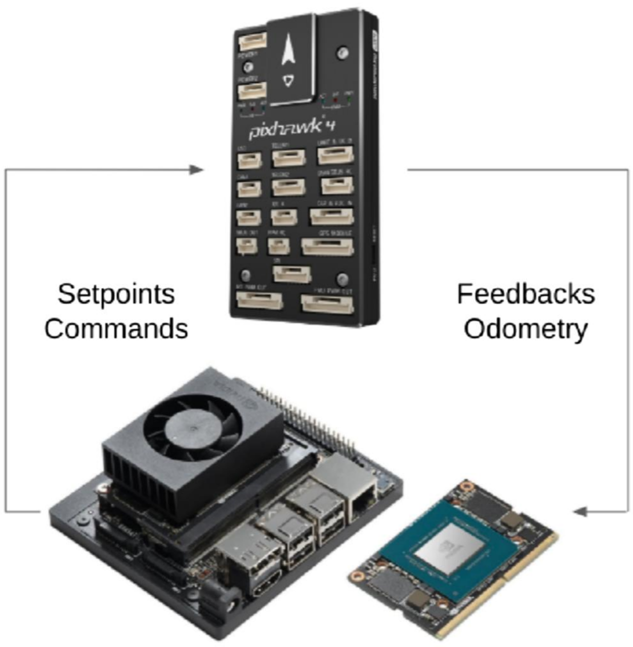

FIGURE 3 Hardware architecture scheme.

The Pixhawk 4 sits at the lowest level. It is essentially an embedded system that integrates many sensors (e.g., inertial measurement unit [IMU], gyroscope, accelerometer) in a single package, together with an STM32 ARM CPU (Advanced RISC Machine central processing unit) and an I/O coprocessor. On this system, the NuttX real-time Operating System (RTOS) is installed, in which it runs the open-source flight control firmware PX4 Autopilot. PX4 Autopilot is an open source project that provides firmware for flight controller developers and vendors and drone developers, maintained by an open community following an open standard. It is fully customisable and documented, offering a variety of features out-of-the-box like.

- angular and vertical stabilisation;
- various control modes, for example, manual, automatic, and remote.
- advanced sensor fusion algorithm based on the Extended Kalman Filter, namely *EKF2*.

A detailed description of how the PX4 firmware multicopter stabilisation and control modules work can be found in Ref. [27]. In this scenario, the EKF2 algorithm was used to integrate and merge different measurements from both integrated sensors and a visual localisation system, which had very different sampling times. The Nvidia Jetson Xavier NX SoC sits at the highest level of the architecture. It features a six-core ARM CPU, 8 GB of RAM, an Nvidia Volta graphics processing unit (GPU), Gigabit Ethernet and WiFi connectivity, and a variety of other communication interfaces. It has been chosen for this application exactly because of all these features, the compact size, and the relatively low power consumption. It natively supports the Linux operating system with a PREEMPT kernel, which made it the ideal choice for the deployment of an architecture based on the tools introduced in Section 2.

3.2.2 | DDS-based microRTPS bridge

The two systems that make up the onboard quadcopter electronics need to communicate in order to organise their operations. However, their scopes are different: The Pixhawk 4 handles drone control in real-time, while the Xavier NX runs heavier and slower algorithms to perform the various mission tasks. A proper communication infrastructure is necessary in this context to handle different kinds of data. Again, Figure 3 shows how this has been done: Modules on the NX send setpoints and control commands to the PX4 firmware, receiving execution feedback, status information, and odometry samples. To ensure that the efficiency characteristics of the tools introduced in Section 2 were not only guaranteed but also fully exploited, this communication infrastructure had to employ the same paradigms. The previous requirements have been fulfiled using a package developed by the PX4 community named *px4_ros_com*, slightly tweaked to fit our hardware platforms. The PX4 firmware is made of different modules that exchange data internally by using a very lightweight middleware named uORB; this one is still a publish-subscribe middleware, which could be easily handled by a DDS as long as an appropriate translation is performed. This is the task of the PX4 endpoint of the microRTPS Bridge: The microRTPS Client application. It runs on the Pixhawk 4 alongside PX4 in real-time, serialising and deserialising transmissions coming from the other side or going towards it. The other end is handled by the Linux endpoint of the Bridge: The microRTPS Agent application. Running on the NX, it has the same task as the Client, but this time it also has to translate transmissions for a DDS running on the system. The version used in this setup was compatible only with the eProsima FastDDS implementation at the time of writing. With that implementation, the default one for the Foxy Fitzroy ROS 2 distribution chosen for this project, topics exposed by the Agent were immediately discovered by every node in the ROS 2 network, not only those on the drone but also all peers running on the remote GCS computer. The physical link used to connect the two ends was a serial interface (Universal Asynchronous Receiver-Transmitter—UART) running at 921,600 bauds. Thus, a fully distributed control architecture was deployed, supported by a real-time communication infrastructure. The first architecture module to interact with such infrastructure was the Flight Control mentioned in Figure 2. It offers a variety of ROS 2 services that enable other modules to request basic flight operations like takeoff, landing, movement to a particular location in the (X, Y, Z) space, and so on, completely hiding the intrinsic complexities of requesting the same operations to PX4 and gathering feedback through the topics exposed by the Bridge. Since such complexities make up almost all of the Flight Control module, they are not discussed any further.

3.3 | Module design

Several approaches have been proposed to create a modular robotic architecture in the literature. Ref. [28] introduced a robotic middleware framework for the modular design of sensory modules, actuation platforms, and task descriptions. The design paradigm that has been chosen in our development process consists of creating functional layers. The modules in each layer are responsible for each specific functionality of the drone, giving it a set of skills necessary to perform complex autonomous tasks. The layers for our used case have been identified as follows: perception and high-level flight control, exploration and navigation, and supervision by an FSM. The first three implement robotic capabilities to achieve autonomy, while the last serves as a glue to join all the components of the architecture together.

3.3.1 | Perception

The main difference between simple mechanisms and robots is that the latter have the ability to adapt to changes in their subjects of operation or in their operating environment. The robot is then able to understand the surrounding environment and to derive a set of actions from the high-level goal it has been given and to implement them through actuation and control. Perception is physically implemented by means of sensors (both enteroceptors and exteroceptors) and processing algorithms applied to the data they produce. Perception has many declensions: interaction between humans and robots in industrial environments as addressed in Ref. [29], autonomous in-water inspection as studied by Ref. [30] and autonomous flight and landing as detailed by Ref. [31]. In this section, the perception modules developed and integrated into the autonomous quadcopter are shown and explained.

ArUco recognition

The term ArUco is an acronym that stands for Augmented Reality University of Cordoba, and it is a synthetic square marker composed of a wide black border and an inner binary matrix that determines its identifier (Figure 4). Its implementation is available as an open source library written in C++ and built upon the OpenCV library, whose purpose is to recognise markers in images. The ArUco project was presented in Refs [32, 33]. All markers are made up of a black background and white squares that form a pattern that is their unique identifier. There are many sets of markers, called dictionaries, that span from 4×4 to 7×7. The dictionary used in this article contains 7×7 markers that can represent numbers from 0 to 999. To recognise ArUco markers on the ground and obtain the required information, the ROS 2 node ArUco Scanner subscribes to the bottom camera topics on which frames are published at a rate of 12 Hz and publishes displacement from recognised markers on another topic. As soon as a new image arrives, it is immediately analysed. If the camera has captured at least one marker, the analysis routine will start extracting the required data. According to the current

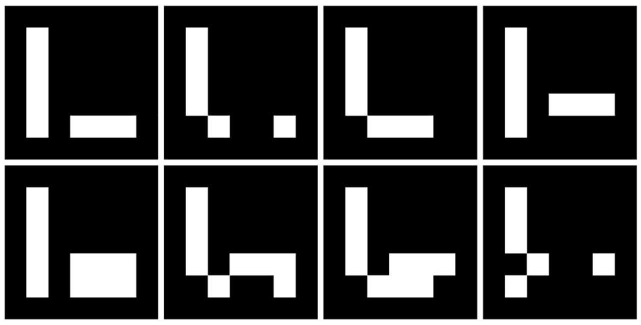

FIGURE 4 ArUco markers with IDs from 1 to 8.

mission phase, the node could be looking for two different kinds of ArUcos: markers with IDs from 12 to 26 represent mobile robots carrying alphanumeric strings, while markers from 1 to 10 are printed on landing pads. In the first case, the system sends an alert message to the Navigator node on its topic and locally saves some pictures of the robot on the ground. In the second case, the drone has to use what it sees on the ground to accurately land on a pad, which is slightly more complicated, as more computations are required in order to get to the right position. First, the four coordinates of the ArUco marker corners $P_i = (x_i, y_i)$, $i = 1, \ldots, 4$ are used to compute the pad centre coordinates (x_c, y_c) with respect to the upper left corner of the image frame, using the following closed-form expressions

$$\begin{cases} x_c = \dfrac{(x_2 - x_4)(x_1y_3 - x_3y_1) + (x_1 - x_3)(x_4y_2 - x_2y_4)}{(x_1 - x_3)(y_2 - y_4) - (x_2 - x_4)(y_1 - y_3)}, \\ y_c = \dfrac{(y_1 - y_3)(x_4y_2 - x_2y_4) + (y_2 - y_4)(x_1y_3 - x_3y_1)}{(x_1 - x_3)(y_2 - y_4) - (x_2 - x_4)(y_1 - y_3)}. \end{cases} \tag{1}$$

The previous equations arise from the intersection of two lines passing through opposite corner points (P_1, P_3) and (P_2, P_4) of the identified marker. Once the coordinates x_c and y_c have been computed, a translation is applied to express them in a reference frame fixed in the centre of the image. By doing so, the error vector can be obtained between the pad and camera centres.

The last steps of the algorithm consist of applying a transformation to the dimensional error vector to express it with respect to the North-East-Down (NED) reference frame of the drone body:

$$\begin{cases} e_x = -e_y^v, \\ e_y = e_x^v, \end{cases} \tag{2}$$

where (e_x, e_y) are the error components in the NED frame, while $\left(e_x^v, e_y^v\right)$ are the errors along x and y in the camera reference frame. Finally, these errors are packed into a message and published so that the Precision Landing node can use them.

Visual-SLAM

Visual-SLAM refers to the use of visual sensors (cameras) to address the problem of simultaneous localisation and mapping. SLAM is the computational problem of creating and updating a map of an unknown environment while simultaneously keeping track of the position and orientation of the agent on which the visual sensor is mounted. Several Visual-SLAM methodologies have been developed so far, and the most promising of them (ORB-SLAM3, OpenVSLAM, and RTABMap) have been reported by Ref. [34]. An interesting categorisation approach, dealt with by Ref. [35], summarises the VSLAM categories into the following categories: feature-based, direct, and Red Green Blue-Depth (RGB-D) camera-based. In our system, we had to focus our efforts on identifying a VSLAM algorithm capable of providing both real-time and robustness features. After a comparison of the available state-of-the-art VSLAM algorithms, ORB-SLAM2 has been chosen and implemented. ORB-SLAM2 is the second version of the ORB-SLAM algorithm introduced in Ref. [36]. All the improvements with respect to the first version have been summarised in Ref. [37]. ORB-SLAM2 uses binary feature extraction from a frame; these features help the algorithm create a local map while solving bundle adjustment problems. The same features are used throughout the system to optimise memory management and maximise efficiency. The algorithm defines the keyframes as the most important frames on which the local map points depend. The keyframes are used to compute the current local pose, creating a covisibility graph that is used both for localisation and environment mapping. Another graph is then created as soon as the map grows in size: Called the essential graph, it is used for loop detection. As soon as a loop has been detected, the system automatically performs a loop closure, so all graphs are analysed and updated, the redundant keyframes and nodes are pruned, and the local map is revised with the latest measurements in order to minimise errors. The system works in four phases, namely, tracking, local mapping, loop closing, and global adjustment. Each phase is assigned to a specific thread, which constitutes the real-time implementation of the whole system. The tracking phase is employed to track the features in the scene, while the local mapping uses the same features to create the local map referred to as the keyframes, as previously specified. The loop closing phase is required to detect any likely loop and then pass the information to the global adjustment, a phase responsible for the global bundle adjustment of the entire map, in order to increase the overall map accuracy. The video stream for the ORB-SLAM2 algorithm can be one of the following: monocular, stereo, and RGB-D. In the implementation presented in this study, a further mode was implemented using infrared and depth images (IRD mode). This choice has been made because of some technical limitations of Intel RealSense cameras; specifically, the main limitation is that the RGB frames are not aligned in time with the depth frames, while the infrared frames are. In particular, RGB versus depth frames are misaligned in time with an always different time slice, which is fatal to the ORB-SLAM2 tracking algorithm.

The ORB-SLAM2 algorithm has been heavily modified in order to increase the number of parameters controlling the algorithm, to include compilation procedures for ARM processors, and to integrate several optimisations for Nvidia CUDA GPUs. The software is provided as an open source in Ref. [38]. The ARM adaptation forced us to disable all the optimisations related to the x86 architecture, with a consequent performance drop, especially in the feature extraction and computation phases, where AVX/SSE vector instruction sets have been widely used. For that reason, the optimisation of CUDA GPUs has been taken into account as a mandatory activity in order to address performance issues. In the distributed architecture introduced in Section 2, this algorithm is implemented in a ROS 2 node that reads frames published on camera topics and publishes the current pose estimated by the tracking system and the current state of the system on two more topics. As a future development, the VSLAM algorithm will be supported by robust sensor fusion, integrating other range-only sensors (e.g., UWB and UHF-RFID), as mentioned in Ref. [39].

3.3.2 | Navigation and guidance

The problem of mobile robot navigation concerns the ability of a robot to determine its own position in the environment, while guidance relates to the ability to manage a trajectory and translate it into control setpoints, as extensively explained in Refs [40, 41]. Furthermore, as stated in Ref. [42], active SLAM is the task of actively planning robot paths while simultaneously building a map and localising within it in a fully autonomous manner, without any human interaction. In the proposed architecture, the node in charge of managing the entire drone navigation is called the Navigator. Assuming that the localisation and flight control systems work properly, the operations carried out by the navigator can be summarised as follows:

- take-off from any pad;
- exploration of the environment in search of the target mobile robot;
- navigation from any point on the map to a specific target;
- landing on any pad.

The programed paths are defined by means of points; the paths are discretised with an interpolation algorithm that is responsible for creating further intermediate setpoints that are reached in sequence by the quadcopter. Each setpoint is considered as reached when the quadcopter position falls inside a sphere with a parameterised radius. The following paragraphs deal in detail with the operations run by the navigator, so the take-off phase is not discussed, as it is managed directly by the flight controller.

Exploration

After taking off, the drone must explore the environment, looking for a specific robot on the ground. The exploration paths are predefined for each pad according to the environment, in order to maximise the probability of finding the robot. Additionally, during the exploration phase and to localise itself, the drone builds a map of the environment as a point cloud thanks to the ORB-SLAM2 algorithm. Therefore, exploration paths have also been designed to execute some loop closures in order to facilitate the construction of the map and increase its accuracy. During exploration, the ArUco Scanner is responsible for alerting the navigator that the robot has been identified. In this case, the drone hovers and requests further instructions about the landings to perform to the operator at the GCS. From then on, the subsequent displacements are managed by the Path Planner module.

Path planning

The goal of the path planner module is to find a valid path from the current position to the target one, such that it is as short as possible and collision-free at the same time. The algorithm developed solves the problem of path planning by building a graph that represents the available space in the environment. The search for the shortest routes is performed using the A* search algorithm [43], whose optimality with respect to the problem of interest is formally proven in Ref. [44]. The first step consists of building the graph directly from the point cloud of the environment obtained through the ORB-SLAM2 algorithm. The environment is then divided into cubes by a three-dimensional grid with variable resolution, and each cube is identified according to its centre. The interconnection of the free cubes constitutes the graph. All those cubes that are at the boundaries of the field are automatically marked as non-viable, so that the quadcopter will not be able to reach them. Then, from the pointcloud, the obstacles in the environment and the related cubes are identified. Initially, the pointcloud is cleaned by removing the points outside the field. Subsequently, the various points are associated with the belonging cubes, and those that contain a number of points greater than a predefined threshold are identified as not viable. Once that is done, the remaining ones define the nodes of the graph, and the algorithm proceeds with the construction of the edges. For each cube, its neighbours are determined as those along eight directions with the same altitude and the two directly above and below. Given the centres of two neighbouring cubes, a corridor is estimated according to the size of the drone, and a connecting edge is inserted between the cubes if and only if there are a number of points less than a predefined threshold within it. This procedure allows one to discretise the environment with a different resolution, but always considering the footprint of the drone in the displacements. If the corridor respects this condition, the displacement is considered valid, and therefore, an edge is inserted between the nodes with a cost equal to the Euclidean distance of the centres. The result of this iterative process is a graph that represents the space of the test field, and its computation is performed offline. This information is then used in an appropriate ROS 2 node, named Path Finder, which performs the search for the shortest paths between the various nodes upon request, using an implementation of the A* algorithm developed by the authors to ensure computational efficiency. The Path Finder node is invoked by requesting a path between the current position and the target one. Initially, the origin and target cubes are identified, and their practicability is verified. If it is satisfied, the search for the shortest path on the graph is launched and it should return the sequence of points that have to be reached in order to reach the target position. A* is a path search algorithm with properties of admissibility, completeness, and optimality, which motivated its choice. The main feature is that it is an informed search algorithm, which means that it uses knowledge for the path search process. Indeed, unlike other algorithms such as Dijkstra, the search of the shortest path towards the target is performed following a heuristic, in the form of an estimation of the traversal cost from the current position to the target one. The better the heuristic, the better the efficiency of the algorithm. When applied to navigation in a three-dimensional space, the Euclidean distance was used as a heuristic to quickly find the minimum-distance path. Figure 5 shows the result of a path planning run where the starting position is represented in blue, the target position is represented in red, and the optimal path is represented in green; the non-viable cubes are depicted in grey. The developed path planner software module has several advantages. It is applicable to any environment starting from its point cloud, allowing us to freely choose the level of discretisation and accuracy. Moreover, it also takes into account the drone footprint with a safety margin to be considered in displacements (Figure 6).

Precision landing

When the drone is led by the path planning node close to a landing pad and the camera can recognise its marker, the FSM invokes the node in charge of executing the precision landings. Landing pads are one m-side squares with a light blue background, in the centre of which an ArUco marker is positioned with a side measuring 0.5 m, as depicted in Figure 7, where several landing sequences are shown. The goal of this node is to let the drone land inside the light blue area, possibly on the marker. As soon as it is invoked by the FSM, it starts receiving position errors from the ArUco Scanner node and initially checks if the drone is stabilised (i.e., if roll and pitch angles are close to zero) so that the camera readings can be considered correct. If the drone attitude is not acceptable, the currently measured errors are discarded. We recall that position errors e_x, e_y are computed by the ArUco Scanner between the centre of the pad and the centre of the image, expressed in the NED reference frame of the drone body. To adjust the position of the drone in order to improve its alignment, the coordinates must be expressed with respect to the world-fixed NED reference frame. Therefore, a rotation around the z axis (yaw) is required to transform e_x and e_y so that they can be used to reach the proper position above the pad

$$\begin{pmatrix} e_x^w \\ e_y^w \end{pmatrix} = \begin{pmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{pmatrix} \begin{pmatrix} e_x \\ e_y \end{pmatrix}, \qquad (3)$$

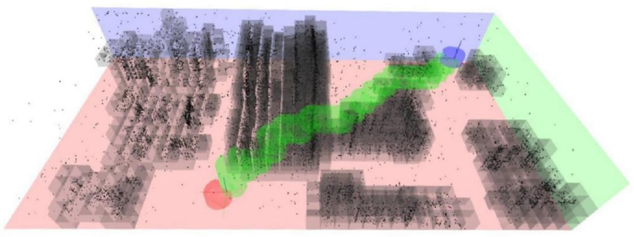

FIGURE 5 The non-viable cubes are depicted in grey, the starting position is depicted in blue, and the optimal path to reach the red target position is depicted in green.

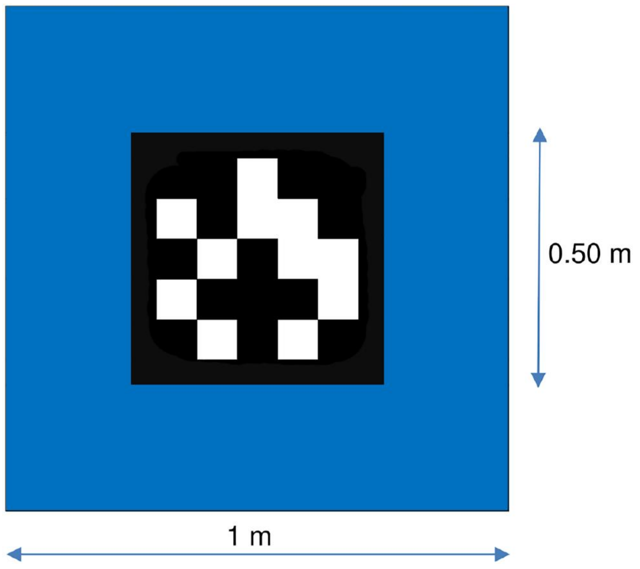

FIGURE 6 Landing pad with an ArUco marker.

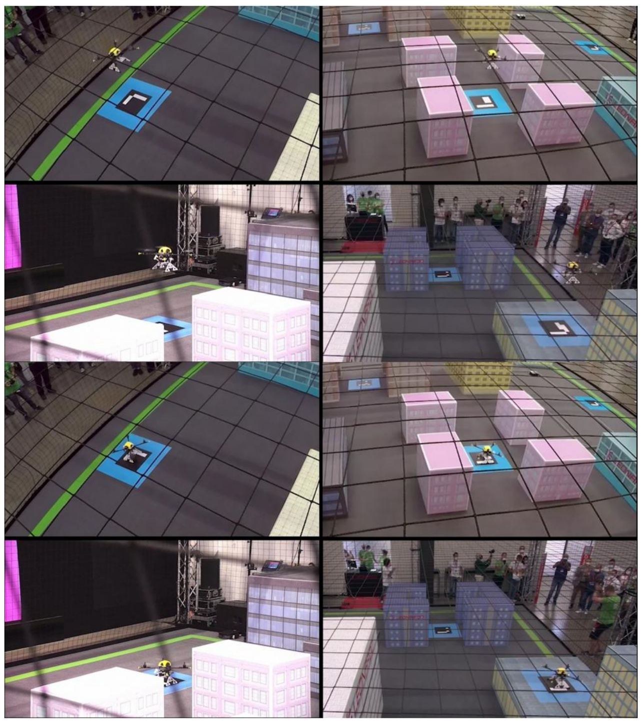

FIGURE 7 Precision landing sequences during the Leonardo Drone Contest 2021. The picture shows the successful precision landings performed on the ArUco landing pads.

where ψ is the current yaw angle and $\left(e_x^w, e_y^w\right)$ are the coordinates expressed in the fixed world frame. Consistent results are ensured by applying a simple first-order filter to the previous errors in order to reduce fast and unwanted variations due to the ArUco border miscalculations, and by truncating errors to centimetres to filter some numerical noise. Subsequently, the node checks if the pad centre lies inside a circle of parameterised radius and if the linear velocities are sufficiently small. If such conditions are met, the reference altitude at which the drone must remain is decreased by a parameterised value. At this point, two different scenarios may arise

- if the current reference altitude is less than a parameterised minimum, the node initiates the final landing procedure, invoking the PX4 firmware by means of a ROS 2 service offered by the Flight Control module;
- if the drone is still too high, the node tries to improve the alignment by reaching a new position defined as the sum of the current drone position with respect to the world frame and the two linear errors $\left(e_x^w, e_y^w\right)$ computed as explained above.

The node described so far does not embed an actual controller for positioning, as it is entirely based on the PX4 position control.

3.3.3 | Finite-state machine

The set of software modules ultimately includes an FSM, which implements the supervision logic placed at the highest level of the software architecture. From a high-level point of view, the drone appears as an automaton capable of carrying out a complete mission within the terms established by the Leonardo Drone Contest rules, managing unexpected situations as much as possible, such as breakdowns, battery depletion, or loss of tracking by the localisation system. Requests towards the various other modules are formulated during transitions between the states, which are triggered on the basis of the current state and other events that have occurred in the meantime. According to the rest of the architecture, this module integrates a ROS 2 node that is used to invoke the other modules and receive feedbacks and alerts. During the transition to the initial state, all the other modules are queried through APIs offered by the ROS 2 middleware: If all are fully operational, the mission sequence is initiated. The *Meta State Machine* library, part of the *C++ boost libraries*, was used for the implementation of the actual FSM. This library allowed us to define the following:

- *states* as structures with methods in which to code what to do when transiting in and out of each one;
- *events* as structures that could possibly hold data to be processed by the state methods during transitions;
- *transitions* and entire machines by instantiating template classes, whose code, however, resembled a real table.

The description of the machine and its states follows the dynamics of a mission as established by the Contest rules. The entire mission has therefore been coded, following the various levels of abstraction illustrated, in a single routine in which the various transitions between the states are invoked depending on the conditions in which the drone is found after each step. The management of error conditions is possible through the meta-machine concept: The main parts of a mission take place in states that constitute a separate FSM, from which one can always exit towards the termination states. It has been found experimentally that it is very risky to request an automatic emergency landing when something goes wrong, especially when the VSLAM system stops working; for this reason, it was decided that, in the case of problems, the FSM should immediately stop the execution without requiring any other operation, thus allowing the pilot to engage manual control as soon as possible. A graphical representation of the two high- and low-level machines is offered in Figures 8 and 9.

4 | EXPERIMENTAL RESULTS

The system described so far has been put through a complex set of tests to investigate the operational capabilities of each module and the robustness of the FSM. Due to the limited space available in the university facilities, it was possible to obtain an extensive verification of the harmonic functioning of all the modules only during the Contest. The single modules have also been evaluated one by one in a series of laboratory tests, to show the performance of each. In this section, both laboratory experiments and the results collected during the Contest are shown and analysed, and the Contest results are compared with the state of the art in Visual Odometry methodologies.

4.1 | Perception module evaluation

This section presents the evaluation of the perception module (both ArUco recognition and Visual-SLAM algorithms) with tests conducted in an indoor environment.

4.1.1 | ArUco recognition

The ArUco recognition module, based on the OpenCV C++ implementation, has been tested to evaluate its performance. A set of tests was performed for different marker sizes and different distances between the camera and the marker (under the same conditions). The ArUco dictionary used for the tests

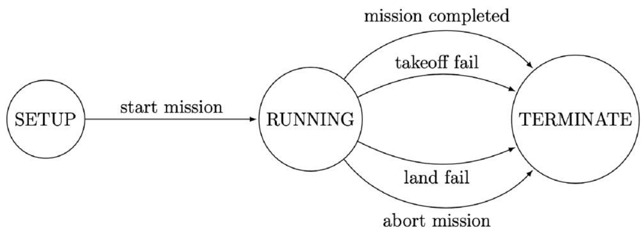

FIGURE 8 High-level finite-state machine state diagram.

is the same as that used in the Contest setup (i.e., 7 × 7 markers, representing numbers from 0 to 999), and the size of the ArUcos was 6.5 cm × 6.5 cm and 3.5 cm × 3.5 cm. The camera resolution was set to 640 × 480, and the tests were repeated under different light conditions (light and dark) at different camera to ArUco distances (80 cm and 120 cm). The test results are reported in Table 1 and show that the performances are very good when the ArUco size is 6.5 cm × 6.5 cm (both in light and dark conditions), but the performance degrades under dark conditions where the recognition module cannot detect ArUco when the distance is 80 cm, while it can still detect it at 120 cm, but sometimes it fails in recognition (i.e., it loses the ArUco). In conclusion, ArUco is a very good choice for an indoor environment with visual markers deployed on the walls or the floor when the requirements are highly flexible and good real-time performance.

4.1.2 | Visual-SLAM

The Visual-SLAM algorithm is based on a hardware-optimised implementation of the ORB-SLAM2 algorithm, as illustrated in Section "Visual-SLAM". The custom implementation has been tested in an indoor office environment with varying light conditions in order to assess its performance in terms of Root Mean Squared Error (RMSE) between the estimated trajectory and ground truth points. Since there was no availability of a complete ground truth along all trajectories, the system was tested at five ground truth points that were measured with a laser sensor with a precision of ±2 mm. The hardware setup used for the test was the same as that deployed on the UAS prototype, as described in Section 3.2.1. The results are presented in Figure 10, where the ORB-SLAM2 trajectory is depicted together with the reference trajectory and the ground truth waypoints where the RMSE between the estimated and real position has been calculated. Table 2 shows the RMSE between the ORB-SLAM2 and ground truth positions at the five waypoints. The results show that the ORB-SLAM2 algorithm works even on this new platform, intrinsically different from the one it was designed for, while there is still room for improvement.

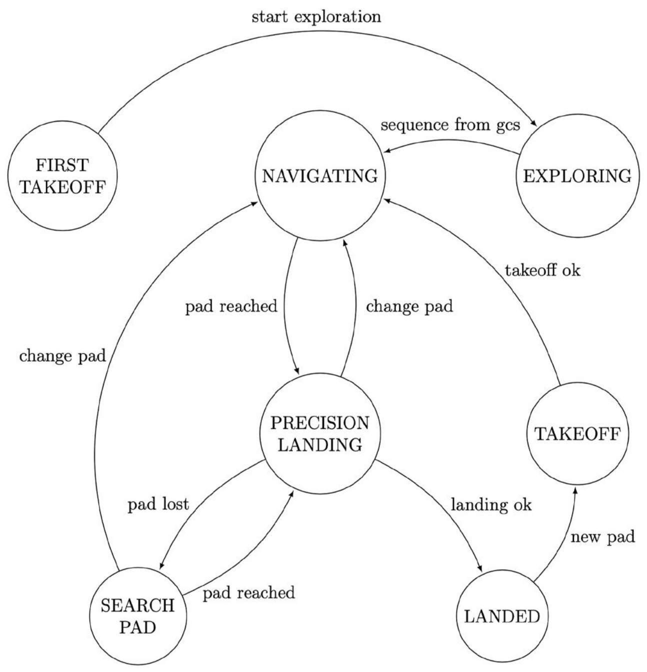

FIGURE 9 Mission finite-state machine state diagram.

4.1.3 | Precision landing evaluation

The performance and correctness of the Precision Landing module were evaluated in a series of tests performed on the Contest competition field, which Leonardo made available to participating teams for a limited time before the Contest. The complete map of the field is shown in Figure 11. The tests

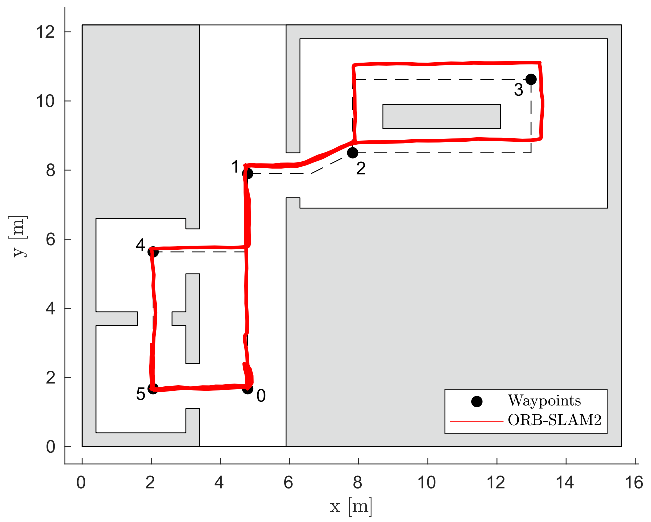

FIGURE 10 ORB-SLAM2 estimated trajectory (red, solid) over the reference path (black, dashed) with waypoints (where 0 is the home position) used as ground truth to estimate the root mean squared error.

TABLE 1 Quantitative evaluation of the ArUco recognition module. Light and Dark report, respectively, the success rate of the ArUco recognition module tested on four attempts under the conditions specified.

ArUco size (cm)	ArUco-camera distance (cm)	Light (%)	Dark (%)
6.5 × 6.5	80	100	100
6.5 × 6.5	120	100	100
3.5 × 3.5	80	100	75
3.5 × 3.5	120	100	0

TABLE 2 RMSE between the ORB-SLAM2 estimated trajectory and the ground truth waypoints.

Waypoint position (*x*, *y*) (m)	ORB-SLAM2 position (*x*, *y*) (m)
(4.788, 7.901)	(4.718, 8.158)
(7.826, 8.501)	(7.925, 8.871)
(12.986, 10.626)	(13.329, 11.146)
(2.053, 5.634)	(1.980, 5.729)
(2.053, 1.676)	(2.034, 1.665)
RMSE	0.352 m

Abbreviation: RMSE, root mean squared error.

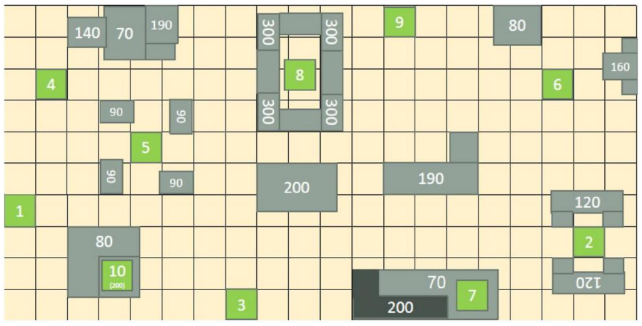

FIGURE 11 Map of the Leonardo Drone Contest 2021 competition field; landing pads are highlighted in green and numbered, and the obstacles are in grey-scale and have their height written above (cm).

consisted of reaching a specific landing pad; in case the ArUco marker was correctly identified, the Precision Landing module made position corrections to align the centre of the drone with the centre of the landing pad. The altitude was then gradually decreased in a controlled descent until the floor was reached, continuously adjusting the position of the drone. The results are presented in Table 3, where the landing pad positions are compared with the landing position of the drone after the corrections were made by the Precision Landing module and the final descent.

4.2 | Real experiments

In the laboratory test (a video is available here),[1] it is shown how the drone starts an exploration and perception phase, during which a map of the environment is created, and the drone is localised within it. During this phase, an ArUco is identified, which acts as a dummy robot. Once the exploration phase is complete, the mission begins, which in the present case involves navigation and landing on two pads. The first one that the drone should reach was removed on purpose and thus cannot be seen to test the FSM branch, which involves searching for the pad in the proximity of its known location. Since it does not find the pad, it proceeds to the next one, successfully recognised, and

TABLE 3 RMSE between the positions of the landing pads and the positions achieved after the corrections made by the Precision Landing module. The results are relative to the landings made on pads 3, 5, and 7.

Landing pads (*x*, *y*)	Landing positions (*x*, *y*)
(−10.000, −7.000)	(−10.106, −7.021)
(−13.052, −2.000)	(−13.052, −2.021)
(−2.700, −6.700)	(−2.692, −6.670)
RMSE	0.068 m

Abbreviation: RMSE, root mean squared error.

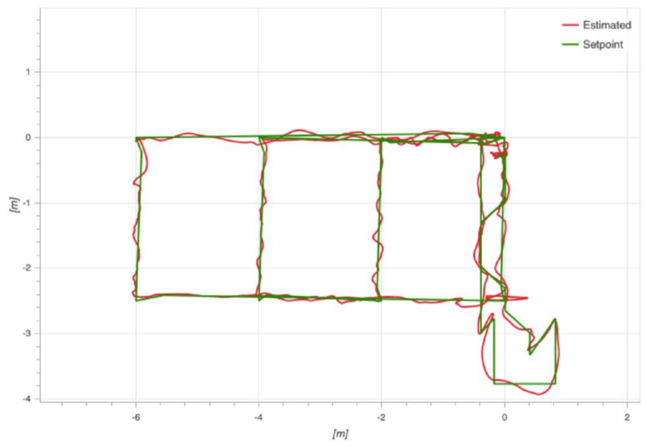

FIGURE 12 Test carried out in the laboratory—(*x*, *y*) path. The green path represents the set points, while the red path represents the estimated trajectory.

then a precision landing over it is successfully performed. Figure 12 shows the complete path covered during this run; although the effectively available space in the laboratory was limited, it allowed one to fully test the algorithms in preparation for the contest. The test lasted 7 min and 33 s, covering an overall distance of 52 m at a maximum speed of 1.7 km/h. The path is quite clean except for a large oscillation towards the end of the first exploration loop due to a computation overload of the Xavier NX companion PC identified on this run and subsequently solved. The overshoot is visible in Figure 13. To evaluate the overall performance of the navigation subsystems, the RMSE of this test was computed for each axis and is shown in Table 4. During the Contest, it was necessary to fly for longer in order to complete a full mission: The quadcopter was able to achieve 14 min of flight time, covering an overall distance of 118 m. The example shown here corresponds to the flight uploaded in the official video[2] of the race. In this case, the exploration path is shorter, and once the robot has been detected, the mission starts in order to look for the landing pads. Figure 14 shows the photo acquired by the drone when the robot was found on the ground, and Figure 15 shows the path covered during the Contest run. Five valid landings were achieved during the Contest run. The RMSE values throughout the path are given in Table 5. The error in the (*x*, *y*) plane is lower

[1]https://youtu.be/N8IV4K4qBb4

[2]https://youtu.be/ayUybULDDwg?t=8940

since the first loop has been executed immediately: in this way, the ORB-SLAM2 localisation algorithm obtained a fairly reliable map in a shorter time compared to the test in the laboratory. The RMSE on the *z* axis is higher because in this case, the variations on *z* were significantly higher.

The previous experimental results have been compared with the most relevant systems in the literature to show that the proposed method performs comparably to the other methodologies presented. To this end, the TUM VI Benchmark [45] has been used; the sequence selected in the data set has been chosen according to a comparable length and environmental complexity (*corridor4*, with a length of 114 m). In order to compare the proposed methodology with the state-of-the-art algorithms in Visual Odometry, the following algorithms have been chosen for comparison: VINS-Mono [46], OKVIS [47], ROVIO [48], BASALT [49], and ORB-SLAM3 [50]. The results are shown in Table 6. From Table 6, it is possible to see that the RMSE for the two environment sets (TUM VI *corridor4* and the Leonardo Drone Contest competition field), with similar lengths and environmental complexity, are still comparable, showing that our proposed methodology is still suitable for such a challenging environment. However, there is room for improvement, also taking into account, in future developments, IMU measurements in the Visual Odometry algorithm to prevent map drifting and thus further reduce the RMSE.

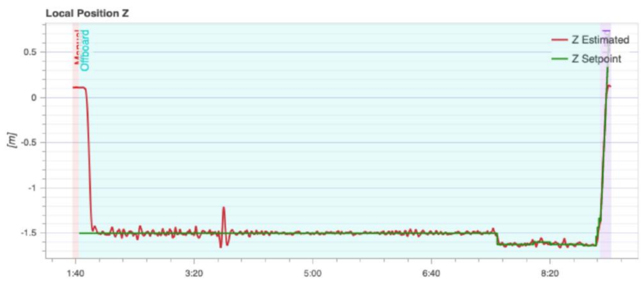

FIGURE 13 Local position for the *z* coordinate. The green curve represents the setpoints for the *z* coordinate, while the red curve represents the estimated *z* trajectory.

TABLE 4 Laboratory test RMSE (m) along each axis.

$RMSE_x$ (m)	$RMSE_y$ (m)	$RMSE_z$ (m)
0.37	0.47	0.29

Abbreviation: RMSE, root mean squared error.

FIGURE 14 Robot detection image (post-processed by the ArUco Scanner node).

5 | CONCLUSIONS

In this study, a general purpose distributed architecture for robotic systems has been presented. The whole architecture is based on ROS 2, taking advantage of the industrial grade DDS middleware that allowed the implementation of transparent communication throughout the system components, ensuring proper QoS requirements for each communication layer. The ROS 2 architecture has been exploited extensively to build all application blocks on a soft real-time Linux operating system. Particular attention was given to the design of the architecture of the whole system, respecting the paradigms of a distributed software architecture. The general results have then been applied to an autonomous quadcopter in order to build a set of solutions ranging from low-level hardware optimisations and operating system setup, to perception, exploration and many higher-level modules. The experimental results, performed both in a laboratory and in realistic Leonardo Drone Contest environments, mainly showed how strong and reliable the software architecture is, especially from the point of view of the overall level of autonomy that has been achieved. The performances of the presented architecture, in terms of both

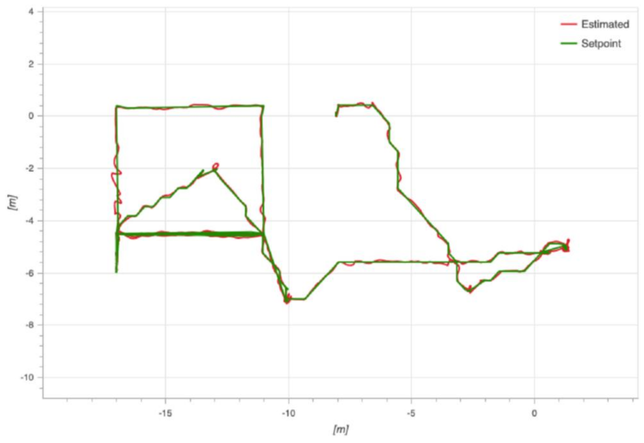

FIGURE 15 Leonardo Drone Contest—(*x*, *y*) path. The green path represents the setpoints, while the red one represents the estimated trajectory.

TABLE 5 Contest run RMSE (m) along each axis.

$RMSE_x$ (m)	$RMSE_y$ (m)	$RMSE_z$ (m)
0.21	0.24	0.38

Abbreviation: RMSE, root mean squared error.

TABLE 6 TUM VI Benchmark *corridor4*[(1)] (length 114 m) and Drone Contest environment[(2)] (length 118 m): RMSE (m) for regions with available ground-truth data.

VINS-Mono[(1)]	OKVIS[(1)]	Our method[(2)]	ROVIO[(1)]	BASALT[(1)]	ORB-SLAM3[(1)]
0.25	0.26	0.27	**0.13**	0.21	0.21

Note: The bold value represents the lowest RMSE.

Abbreviation: RMSE, root mean squared error.

autonomy and reliability, led to the achievement of the second prize at the 2021 Edition of the Leonardo Drone Contest.

The architecture proposed in this study exploits the capabilities of the ROS 2 framework and enables scalability and secure real-time communication in the context of UASs thanks to the underlying DDS middleware. Furthermore, the application of a state-of-the-art algorithm for Visual-SLAM (namely, ORB-SLAM2 with several improvements) on a real UAS is another innovation to be mentioned together with the development of a flexible path planner working in any environment. As for future developments and contributions, the case where the drone takes off from an uneven surface will also be taken into account to make the presented system even more autonomous and exploitable in more realistic contexts. Another case that will be subsequently studied is that of completely unknown environments that the drone has to explore and map in real-time with the algorithms and software tools presented here.

ACKNOWLEDGEMENTS

The authors want to thank Leonardo S.p.A. for the availability, testing scenarios, and granted participation in the Leonardo Drone Contest. Without their support in testing the system using their facilities, this work would not have been possible. Another thanks goes to Marco Passeri, Ph.D., for his fundamental support during the development and test phases of the system and for sharing his precious knowledge with the team. This work has been supported in part by the Italian Ministry of Research in the framework of the Program for Research Projects of National Interest (PRIN), under Grants No. 2017YKXYXJ and No. 2020RTWES4.

CONFLICT OF INTEREST STATEMENT

The authors do not report any conflict of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Fabrizio Romanelli https://orcid.org/0000-0002-1888-7004

REFERENCES

1. Ahmad, A., Ali Babar, M.: Software architectures for robotics systems: a systematic mapping study. J. Syst. Software 122, 16–39 (2016). https://doi.org/10.1016/j.jss.2016.08.039
2. Yang, S., et al.: Towards a hybrid software architecture and multi-agent approach for autonomous robot software. Int. J. Adv. Rob. Syst. 14(4), 1729881417716088 (2017). https://doi.org/10.1177/1729881417716088
3. Jahn, U., Wolff, C., Schulz, P.: Concepts of a modular system architecture for distributed robotic systems. Computers 8(1), 25 (2019). https://doi.org/10.3390/computers8010025. https://www.mdpi.com/2073-431X/8/1/25
4. Magee, J., et al.: Specifying distributed software architectures. In: Software Engineering—ESEC '95, pp. 137–153 (2006)
5. Knoop, S., et al.: A corba-based distributed software architecture for control of service robots. In: 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566), vol. 4, pp. 3656–3661 (2004)
6. Mishra, A., Mishra, D.: Software architecture in distributed software development: a review. In: Demey, Y.T., Panetto, H. (eds.) On the Move to Meaningful Internet Systems: OTM 2013 Workshops, pp. 284–291. Springer Berlin Heidelberg, Berlin (2013)
7. Mattmann, C., et al.: Software architecture for large-scale, distributed, data-intensive systems. In: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), pp. 255–264 (2004)
8. Lütkebohle, I., et al.: Generic middleware support for coordinating robot software components: the task-state-pattern. J. Software Eng. Robot. (2011)
9. Quigley, M., et al.: ROS: an open-source robot operating system. In: Proceedings of the IEEE International Conference on Robotics and Automation Workshop on Open Source Software, vol. 3 (2009)
10. Malavolta, I., et al.: How do you architect your robots? State of the practice and guidelines for ROS-based systems. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 31–40 (2020)
11. Open Source Robotics Foundation: ROS2. https://github.com/ros2
12. Pardo-Castellote, G.: OMG data-distribution service: architectural overview. In: In Proceedings of the IEEE International Conference on Distributed Computing Systems Workshops, pp. 200–206 (2003)
13. Schlesselman, J.M., Pardo-Castellote, G., Farabaugh, B.: OMG data-distribution service (DDS): a architectural update. In: In Proceedings of the IEEE Military Communications Conference, vol. 2, pp. 961–967 (2004)
14. Maruyama, Y., Kato, S., Azumi, T.: Exploring the performance of ROS2. In: Proceedings of the 13th International Conference on Embedded Software, pp. 1–10 (2016)
15. Shi, L., Marcano, N.J.H., Jacobsen, R.H.: A review on communication protocols for autonomous unmanned aerial vehicles for inspection application. Microprocess. Microsyst. 86, 104340 (2021). https://doi.org/10.1016/j.micpro.2021.104340. https://www.sciencedirect.com/science/article/pii/S014193312100497X
16. Lee, H., et al.: A robot operating system framework for secure UAV communications. Sensors 21(4), 1369 (2021). https://doi.org/10.3390/s21041369
17. Antonelli, G.: Interconnected dynamic systems: an overview on distributed control. IEEE Control Syst. Mag. 33(1), 76–88 (2013)
18. Ahmed, N., Cortes, J., Martinez, S.: Distributed control and estimation of robotic vehicle networks: overview of the special issue. IEEE Control Syst. Mag. 36(2), 36–40 (2016)
19. Bauer, F.L., et al.: Software Engineering. North Atlantic Treaty Organization Science Committee, Tech. Rep. (1968)
20. OMG: Omg Data Distribution Service (DDS) Version 1.4. Object Management Group, Tech. Rep. formal/2015-04-10 (2015)
21. OMG: The Real-Time Publish-Subscribe Protocol DDS Interoperability Wire Protocol (DDSI-RTPSTM) Specification Version 2.5. Object Management Group, Tech. Rep. formal/2021-03-03 (2021)

22. Macenski, S., et al.: Robot operating system 2: design, architecture, and uses in the wild. Sci. Robot. 7(66) (2022). https://www.science.org/doi/abs/10.1126/scirobotics.abm6074
23. Thomas, D., Woodall, W., Fernandez, E.: Next-generation ROS: building on DDS. In: ROSCon Chicago 2014. Open Robotics, Mountain View, CA (2014). https://vimeo.com/106992622
24. Gutiérrez, C.S.V., et al.: Towards a distributed and real-time framework for robots: evaluation of ROS 2.0 communications for real-time robotic applications. CoRR abs/1809.02595 (2018). http://arxiv.org/abs/1809.02595
25. Saito, Y., et al.: Priority and synchronization support for ROS. In: 2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA), pp. 77–82 (2016)
26. Jiang, Z., et al.: Message passing optimization in robot operating system. Int. J. Parallel Program. 48(1), 119–136 (2020). https://doi.org/10.1007/s10766-019-00647-w
27. Brescianini, D., Hehn, M., D'Andrea, R.: Nonlinear Quadrocopter Attitude Control: Technical Report. ETH Zurich, Tech. Rep. (2013)
28. Elkady, A., et al.: A structured approach for modular design in robotics and automation environments. J. Intell. Rob. Syst. 72(1), 5–19 (2012). https://doi.org/10.1007/s10846-012-9798-y
29. Bonci, A., et al.: Human-robot perception in industrial environments: a survey. Sensors 21(5), 1571 (2021). https://doi.org/10.3390/s21051571. https://www.mdpi.com/1424-8220/21/5/1571
30. Hover, F., et al.: Advanced perception, navigation and planning for autonomous in-water ship hull inspection. Int. J. Robot. Res. 31(12), 1445–1464 (2012). https://doi.org/10.1177/0278364912461059
31. Singh, S., et al.: Perception for safe autonomous helicopter flight and landing. In: 72nd Annual Forum and Technology Display, American Helicopter Society (AHS) (2016)
32. Romero-Ramirez, F.J., Muñoz-Salinas, R., Medina-Carnicer, R.: Speeded up detection of squared fiducial markers. Image Vis. Comput. 76, 38–47 (2018). https://doi.org/10.1016/j.imavis.2018.05.004
33. Garrido-Jurado, S., et al.: Generation of fiducial marker dictionaries using mixed integer linear programming. Pattern Recogn. 51, 481–491 (2016). https://doi.org/10.1016/j.patcog.2015.09.023
34. Merzlyakov, A., Macenski, S.: A comparison of modern general-purpose visual SLAM approaches. CoRR abs/2107.07589 (2021). https://arxiv.org/abs/2107.07589
35. Taketomi, T., Uchiyama, H., Ikeda, S.: Visual slam algorithms: a survey from 2010 to 2016. IPSJ Trans. Comp. Vis. Appl. 9(1), 16 (2017). https://doi.org/10.1186/s41074-017-0027-2
36. Mur-Artal, R., Montiel, J.M.M., Tardós, J.D.: Orb-slam: a versatile and accurate monocular slam system. IEEE Trans. Robot. 31(5), 1147–1163 (2015). https://doi.org/10.1109/tro.2015.2463671
37. Mur-Artal, R., Tardós, J.D.: ORB-SLAM2: an open-source slam system for monocular, stereo, and RGB-D cameras. IEEE Trans. Robot. 33(5), 1255–1262 (2017). https://doi.org/10.1109/tro.2017.2705103
38. Romanelli, F.: ORB-SLAM2—new parameters management, ARM compilation, CUDA GPU compatibility. https://github.com/fabrizioromanelli/ORBSLAM2 (2021)
39. Di Giampaolo, E., Martinelli, F., Romanelli, F.: Robust simultaneous localization and mapping using the relative pose estimation of trilateration UHF RFID tags. IEEE J. Radio Freq. Ident. 6, 1–1–592 (2022). https://doi.org/10.1109/jrfid.2022.3179045
40. Farid, K.: Survey of advances in guidance, navigation, and control of unmanned rotorcraft systems. J. Field Robot. 29(2), 315–375 (2012). https://doi.org/10.1002/rob.20414
41. Elkaim, G.H., Lie, F.A.P., Gebre-Egziabher, D.: Principles of guidance, navigation, and control of UAVs. In: Handbook of Unmanned Aerial Vehicles, pp. 347–380 (2015)
42. Lluvia, I., Lazkano, E., Ansuategi, A.: Active mapping and robot exploration: a survey. Sensors 21(7), 2445 (2021). https://doi.org/10.3390/s21072445
43. Guruji, A.K., Agarwal, H., Parsediya, D.: Time-efficient a* algorithm for robot path planning. Procedia Technol. 23, 144–149 (2016). 3rd International Conference on Innovations in Automation and Mechatronics Engineering 2016, ICIAME 2016 05–06 February, 2016. https://doi.org/10.1016/j.protcy.2016.03.010. https://www.sciencedirect.com/science/article/pii/S2212017316300111
44. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. 4(2), 100–107 (1968). https://doi.org/10.1109/tssc.1968.300136
45. Schubert, D., et al.: The TUM VI benchmark for evaluating visual-inertial odometry. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 1680–1687 (2018)
46. Qin, T., Li, P., Shen, S.: VINS-mono: a robust and versatile monocular visual-inertial state estimator. IEEE Trans. Robot. 34(4), 1004–1020 (2018). https://doi.org/10.1109/tro.2018.2853729
47. Leutenegger, S., et al.: Keyframe-based visual–inertial odometry using nonlinear optimization. Int. J. Robot Res. 34(3), 314–334 (2015). https://doi.org/10.1177/0278364914554813
48. Bloesch, M., et al.: Iterated extended Kalman filter based visual-inertial odometry using direct photometric feedback. Int. J. Robot Res. 36(10), 1053–1072 (2017). https://doi.org/10.1177/0278364917728574
49. Usenko, V., et al.: Visual-inertial mapping with non-linear factor recovery. IEEE Rob. Autom. Lett. 5(2), 422–429 (2020). https://doi.org/10.1109/lra.2019.2961227
50. Campos, C., et al.: ORB-SLAM3: an accurate open-source library for visual, visual–inertial, and multimap SLAM. IEEE Trans. Robot. 37(6), 1874–1890 (2021). https://doi.org/10.1109/tro.2021.3075644

How to cite this article: Bianchi, L., et al.: A novel distributed architecture for unmanned aircraft systems based on Robot Operating System 2. IET Cyber-Syst. Robot. e12083 (2023). https://doi.org/10.1049/csy2.12083