CrazyChoir: Flying Swarms of Crazyflie Quadrotors in ROS 2

Lorenzo Pichierri, Andrea Testa, and Giuseppe Notarstefano

***Abstract*—This letter introduces CrazyChoir, a modular Python framework based on the Robot Operating System (ROS) 2. The toolbox provides a comprehensive set of functionalities to simulate and run experiments on teams of cooperating Crazyflie nano-quadrotors. Specifically, it allows users to perform realistic simulations over robotic simulators as, e.g., Webots and includes bindings of the firmware control and planning functions. The toolbox also provides libraries to perform radio communication with Crazyflie directly inside ROS 2 scripts. The package can be thus used to design, implement and test planning strategies and control schemes for a Crazyflie nano-quadrotor. Moreover, the modular structure of CrazyChoir allows users to easily implement online distributed optimization and control schemes over multiple quadrotors. The CrazyChoir package is validated via simulations and experiments on a swarm of Crazyflies for formation control, pickup-and-delivery vehicle routing and trajectory tracking tasks.**

***Index Terms*—Distributed robot systems, software architecture for robotics and automation, cooperating robots, optimization and optimal control.**

I. Introduction

UAV swarms have gained a great interest in both research and industrial applications due to their applicability in a wide range of scenarios. In the last years, the Crazyflie platform (Fig. 1) has gained a lot of attention among researchers. Crazyflie is a cheap nano-quadrotor weighting 27 g. It is endowed with an ARM Cortex-M4 microcontroller and can receive radio messages through a so-called Crazyradio dongle. In the last years, several efforts have been put into the development of simulation and control toolboxes based on the Robot Operating System (ROS) [1]. Recently, ROS is being substituted by ROS 2, which is expanding its capabilities with novel functionalities [2]. In this paper, we introduce CrazyChoir, a ROS 2 package allowing researchers to run realistic simulations and laboratory experiments on a swarm of cooperating Crazyflie nano-quadrotors.

Manuscript received 19 January 2023; accepted 25 May 2023. Date of publication 15 June 2023; date of current version 29 June 2023. This letter was recommended for publication by Associate Editor G. Notomista and Editor M. A. Hsieh upon evaluation of the reviewers' comments. This work was supported in part by the Italian Ministry of Foreign Affairs and International Cooperation, under Grant BR22GR01. *(Corresponding author: Lorenzo Pichierri.)*

The authors are with the Department of Electrical, Electronic and Information Engineering, University of Bologna, 40136 Bologna, Italy (e-mail: lorenzo.pichierri@unibo.it; a.testa@unibo.it; giuseppe.notarstefano@unibo.it).

CrazyChoir is available at https://github.com/OPT4SMART/crazychoir.

This letter has supplementary downloadable material available at https://doi.org/10.1109/LRA.2023.3286814, provided by the authors.

Digital Object Identifier 10.1109/LRA.2023.3286814

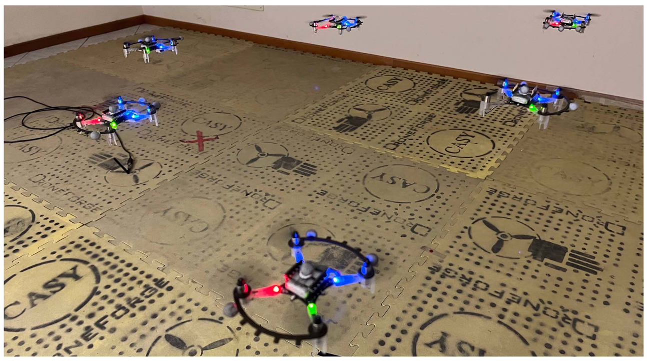
Fig. 1. Snapshot of a swarm of Crazyflie nano-quadrotors in our testbed.

A. Related Work

Control of quadrotors is currently performed with toolboxes based on the first ROS version. As for toolboxes specific for the Crazyflie, the toolboxes in [3], [4] allow users to run experiments on multiple Crazyflies. Researchers also developed tools to efficiently simulate generic, autonomous quadrotors in a realistic environment. An Unreal Engine simulator for unmanned vehicles is proposed in [5]. Authors in [6] propose a simulator based on Unreal Engine with bridges for RotorS and OpenAI-Gym to perform Reinforcement Learning tasks. Works in [7], [8] propose two toolboxes to simulate and control multiple UAVs. Finally, a ROS-Gazebo simulator for the Erle-Copter is provided in [9]. As for simulation toolboxes tailored for the Crazyflie, authors in [10] extend the well-known RotorS toolbox [11] to target the Crazyflie. A Gazebo-ROS-Simulink toolbox for the Crazyflie is proposed instead in [12]. However, in the above toolboxes, the swarm is handled by few, centralized ROS processes that interact with all the quadrotors, thus affecting the scalability and robustness of the swarm. Moreover, they do not involve distributed or decentralized design tools for swarm management and control. Recently, the novel ROS 2 [13] is substituting ROS. Indeed, ROS 2 allows multi-process communication leveraging the popular Data Distribution Service (DDS) open standard. The DDS does not require a centralized broker to dispatch messages, and nodes can implement self-discovery procedures. Moreover, the DDS allows for the implementation of different Quality of Service profiles, and provides reliable and secure communication. Also, ROS 2 has been designed for industrial settings, thus supporting real-time [14] and embedded systems [15] development. Authors in [16] propose a framework for collaborative industrial manipulators. The work in [17] proposes a ROS 2-based architecture for self-driving cars. As for works concerning software for multi-robot systems

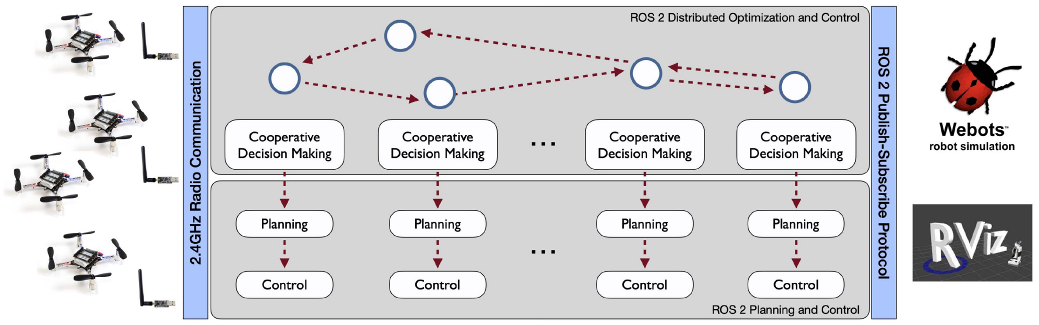
2.4GHz Radio Communication
ROS 2 Distributed Optimization and Control
Cooperative Decision Making
Cooperative Decision Making
...
Cooperative Decision Making
Cooperative Decision Making
Planning
Planning
...
Planning
Planning
Control
Control
Control
Control
ROS 2 Planning and Control
ROS 2 Publish-Subscribe Protocol
Webots robot simulation
RViz

Fig. 2. CrazyChoir architecture. Crazyflies can exchange information with neighbors to implement distributed feedback and optimization schemes. Local classes handle planning and low-level control. Control inputs can be sent to simulation environments (Webots or RViz) or to real robots via radio.

in ROS 2, papers [18], [19] address the development of toolboxes for swarm robotics on ground mobile robots. Moreover, these works neglect cooperation among robots or simulate it via (computationally demanding) all-to-all communication. None of these frameworks is however tailored for the Crazyflie. Recently, Crazyswarm2 [20] has been proposed as a ROS 2 extension to Crazyswarm [3]. Still, this package does not include routines to implement distributed optimization and control schemes on cooperating Crazyflies. Finally, the toolbox in [21] provides distributed optimization and communication functionalities for teams of generic robots. This package however does not handle Crazyflie swarms and radio communication with the quadrotors. Moreover, as we detail next, our package provides simulations by combining the Webots engine [22] and Crazyflie firmware bindings.

B. Contributions

In this paper, we introduce CrazyChoir, a ROS 2 modular framework to perform realistic simulations and experiments on swarms of cooperating Crazyflie nano-quadrotors. The toolbox provides a comprehensive set of Python modules allowing for fast prototyping of (distributed) decision-making and control schemes on Crazyflie swarms. Specifically, CrazyChoir is a scalable package in which each Crazyflie is handled by a set of independent ROS 2 processes, thus reducing single points of failure. These nodes are executed on a ROS-enabled workstation and can be allocated among multiple workstations connected on the same networks. Also, its modular structure allows the user to easily switch from simulative scenarios to real experiments by changing a few lines of code. The toolbox is integrated with the Webots engine, so that users can perform realistic simulations of multiple Crazyflie. To the best of the authors' knowledge, realistic, CAD-based simulation of large-scale swarms is often neglected. Moreover, the simulator incorporates the firmware bindings provided by Bitcraze implementing Crazyflie onboard control functions. Thereby, the user can perform parameter tuning in a safe simulation environment. The proposed framework also allows for lightweight simulations by means of an ad-hoc numerical integrator combined with the RViz visualizer. Finally, CrazyChoir supports DISROPT [23] and ChoirBot [21] toolboxes, thus allowing users to implement distributed optimization schemes and handle inter-robot communications over ROS 2 according to a user-defined graph. The paper unfolds as follows. In Section II, we detail the architecture of CrazyChoir. Section III details libraries for control schemes. Section IV discusses swarm decision-making strategies, distributed control and distributed optimization functionalities of the proposed package. In Section V, we discuss the communication bridge with Crazyflie. Section VI shows how to simulate swarms of Crazyflies in Webots and RViz. In Section VII, we provide a use-case implementation, while other experiments are discussed in Section VIII.

II. Architecture Description

CrazyChoir is written in Python and is based on the novel ROS 2 framework. In CrazyChoir, each Crazyflie is handled by a set of ROS 2 processes (also called *nodes*). These nodes handle simulation, control, planning, and radio communication for each quadrotor, as well as inter-robot communication and cooperation. In the proposed package, these functionalities are implemented in a modular fashion as a set of Python classes. At runtime, these classes are instantiated in a dedicated ROS 2 node as a standalone process. An illustrative representation of the software architecture is in Fig. 2. The software architecture of CrazyChoir is made of five main blocks: i) control layer, ii) swarm planning layer, iii) cooperative decision-making layer, iv) radio communication layer, v) realistic simulation layer. We now briefly introduce the main components of the proposed architecture. A detailed description is provided in the following sections. The control layer provides a set of classes that can be extended to implement off-board feedback-control schemes. These control classes receive reference trajectories from the planning module, in which users can also provide graphical inputs to describe complex trajectories. The trajectory planning functionalities are used to generate feasible trajectories, steering the generic quadrotor to fulfill high-level decisions. These decisions are taken in another module provided by CrazyChoir, in which users can implement cooperative, decision-making, and distributed feedback schemes on swarms of Crazyflies. This cooperative decision-making layer indeed provides functionalities to implement distributed, online optimization and control schemes. Leveraging this cooperative framework, robots can jointly fulfill complex tasks by exchanging messages with neighboring robots. The control inputs generated by the control classes, or the setpoints provided by the planning module, can either be sent to the Crazyflie via radio, or transmitted via ROS 2 topics to the simulators provided in CrazyChoir. In particular, users can choose between realistic simulations, developed in Webots, and lightweight numerical integrations, visualized in Rviz. Finally, CrazyChoir can be interfaced with motion capture systems as, e.g., Vicon so that users can retrieve quadrotor poses during the experiments. The proposed package implements derivative and low-pass filters to obtain linear and angular velocities for each robot. Indeed, motion capture systems usually provide only position and attitude data. Thanks to this modular design, users can combine classes from the different layers according to their needs. As we detail in the following, this allows for switching from simulations to experiments with a few lines of code. Users can quickly extend the proposed modules with novel functionalities and, thanks to the ROS 2 capabilities, can also deploy the software on different workstations.

III. CrazyChoir Control Library

CrazyChoir provides a `CrazyflieController` template class that can be specialized to implement the desired control schemes for the Crazyflie. This class aims at mapping desired trajectory inputs to a control setpoint to be sent to the Crazyflie or to the simulators. The `CrazyflieController` class is designed according to the *Strategy* pattern [24] so that the specific control law can be defined at runtime. Specifically, the class owns as attributes a set of classes specifying its behavior. That is, it owns `TrajectoryHandler` and `CommunicationHandler` classes specifying how desired trajectories will be published to the controller and which control input must be communicated to the Crazyflie. In the proposed package, we already provide a `HierarchicalController` class that implements classical flatness-based control schemes [25]. This class is suitable to track a desired flat-output trajectory (e.g., position and yaw profiles with their derivatives), or it can be used to track desired acceleration profiles coming from, e.g., double-integrator distributed control laws. We provide an example of this setting in Section VII, in which we use this class to perform a distributed formation control task. `HierarchicalController` extends the `CrazyflieController` template. To this end, it owns two additional classes `PositionControl` and `AttitudeControl`, implementing position and angular control, respectively. The outputs of the `PositionControl` class are the thrust and desired attitude profile. The `AttitudeControl` classes instead generate an angular velocity profile given desired attitude references. As an example, we implement the geometric control law in [26]. We refer the reader to the next sections for a detailed discussion of these modules. A sketch of the role of these classes and the related topics is provided in Fig. 3. Here, the arrows generated from the `HierarchicalController` class represent the fact that `TrajectoryHandler` and `CommunicationHandler` are attributes of `HierarchicalController`, that call their methods. The `HierarchicalController` class receives trajectory messages on `/traj_topic` and sends control messages on `/cmd_vel` topic.

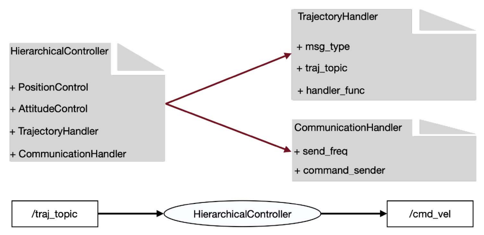

Fig. 3. `HierarchicalControl` class and involved topics.

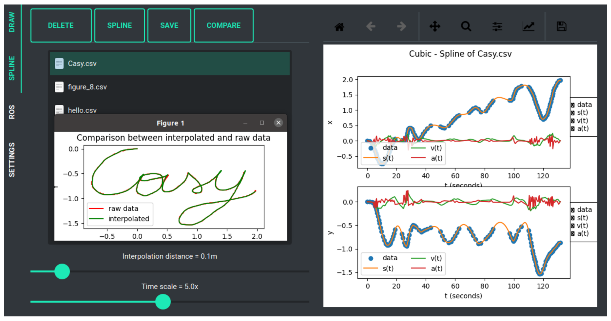

Fig. 4. Snapshot of the proposed GUI.

IV. CrazyChoir Modules for Cooperative Decision Making and Planning

CrazyChoir provides modules to perform off-board planning tasks for each quadrotor. Also, the proposed package allows users to implement cooperative decision-making and distributed feedback schemes on swarms of Crazyflies.

A. GUI-Based Swarm Trajectory Planning

CrazyChoir involves a set of functionalities to handle planning tasks. More in detail, CrazyChoir includes a graphical interface to provide trajectories to a set of quadrotors. In fact, one of these functionalities allows users to hand-draw trajectories and directly send them to the chosen quadrotor(s). The points of these hand-drawn trajectories are interpolated, and a polynomial spline is constructed in order to ensure the continuity of velocity and acceleration profiles. Then, users can tune the trajectory time in order to increase/decrease its execution and avoid high accelerations. Also, this process can be extended in order to use different planning methods and, e.g., manually impose bounds on the trajectories. A snapshot of the interface is reported in Fig. 4, where we draw the name of our laboratory (i.e., *Casy*). Further, in Section VIII we present an example in which a Crazyflie follows this trajectory.

This functionality can be exploited in scenarios in which, e.g., there is a subset of *leading* quadrotors that move in the space and a subset of *following* quadrotors that run cooperative algorithms to suitably track the leaders. In Section VII, we provide an example of formation control with moving leaders to highlight this aspect. The GUI can be also used to help the final user to manage classical operations needed during simulations/experiments (e.g., hovering, landing, starting the experiment, etc.). Then, the package provides methods to perform point-to-point trajectories and to evaluate polynomial paths passing through multiple, arbitrary points. These modules allow the user to send (e.g., to the controller node) position, velocity, and acceleration setpoints constituting a smooth trajectory at the desired communication rate. These classes also handle dynamic replanning scenarios where a new trajectory has to be evaluated while the quadrotor is already following another path. To interface planning classes with, e.g. controller classes, CrazyChoir provides a `TrajectoryHandler` Strategy class. This class details the topic on which the desired trajectory will be published. In this way, ROS 2 nodes dynamically instantiate the proper ROS 2 publisher and subscribers. The class also specifies the trajectory message type and a callback function to extract the trajectory from the message.

B. Distributed Optimization and Control

In CrazyChoir, the user can implement *distributed* optimization and control schemes in which teams of quadrotors exchange suitable data in order to solve complex tasks. In this cooperative setting, each quadrotor communicates with few, neighboring quadrotors, possibly according to time-varying communication topologies. As a result, the team of robots leverages the *local* knowledge of each quadrotor (coming, e.g. from onboard sensors) and exploits inter-robot communications to achieve a global goal. CrazyChoir implements a set of functionalities to deploy distributed optimization and control schemes in which users can implement distributed algorithms from the perspective of each robot. Embedding DISROPT [23] functionalities, users can model different optimization problems in which each robot knowns only a part of the objective function and constraints, eventually including non-convex, mixed-integer sets. An example is provided in Section VIII-B. Inter-robot communication is then handled using the ROS 2 middleware. To this end, it is compatible with ChoiRbot [21], a ROS 2 package for cooperative robotics. More in detail, it allows users to implement static or time-varying directed communication networks. This is performed according to the publisher-subscriber protocol. Exploiting the Quality of Service feature introduced in ROS 2, the package allows for the implementation of synchronous and asynchronous schemes, as well as lossy communications. This set of functionalities is provided by two main classes, `CFGuidance` and `CFDistributedFeedback`, that extend ChoiRbot functionalities in order to implement distributed schemes tailored for Crazyflie swarms. The two classes expose to users a set of pre-defined methods `send`, `receive`, `asynchronous_receive`, `neighbors_exchange` (to simultaneously perform send/receive) according to some user-defined communication graph. These functions also allow to send different kinds of data to different neighbors. An interesting feature is that the user does not need to define any ROS message. Indeed, all the data is automatically serialized and sent as a `std_msgs/ByteMultiArray` on the sender side, while it is de-serialized on the receiver side. Thus, users can extend these classes according to their needs by only implementing the desired algorithmic strategies without warning about inter-robot communications. We remind the reader to Section VII and Section VIII for usage examples of these classes for cooperative control and optimization tasks.

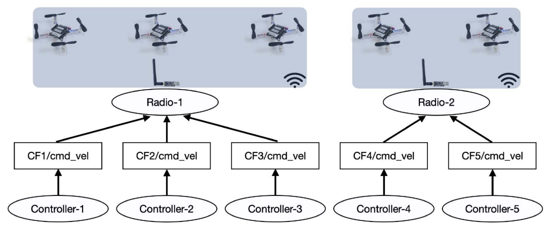

Fig. 5. Examples of nodes and topics involved in the radio communication for a swarm of 5 quadrotors and 2 radio dongles.

V. Radio Communication

To physically control the Crazyflie nano-quadrotor, angular or angular-rate set points can be sent via radio. Position or velocity set-points can be communicated as well, together with periodical ground-truth information. Crazyflies are equipped with an nRF51822 radio controller, used to communicate with a PC by means of the Crazyradio PA. The Crazyradio PA is a 2.4 GHz USB radio dongle based on the nRF24LU1+. It transmits 32-byte packets with datarate up to 2Mbits, spreading in a range of 125 channels. Each Crazyflie is able to receive data on one of these channels at a certain datarate. The choice of these communication parameters is crucial when performing experiments with a large number of quadrotors. Indeed, a wrong choice may result in a noisy communication channel and poor flight performance. To mitigate this aspect, CrazyChoir provides a script that identifies the channels that are less subject to external interferences. To handle the communication with the quadrotors, CrazyChoir implements a tailored `RadioHandler` class. Each Crazyradio PA is handled by an instantiation of this class into an independent ROS 2 node. In particular, this class allows each dongle to communicate with multiple Crazyflies by exploiting the Python methods of `Crazyflie-lib-python` developed by Bitcraze. The `RadioHandler` requires a list of `URI_address`, used to set a unique communication link with each Crazyflie associated with the radio. In an initialization procedure, the class updates a set of required parameters of the Crazyflie related to the onboard controller. After that, each `radio` node subscribes to the multiple `cmd_vel` topics published by the control nodes (cf. Section III) for each Crazyflie associated with the dongle. The subscription callback triggers the `cmd_sender`, which is implemented in subclasses of `RadioHandler`, and it is in charge of elaborating the control input and forwarding it to the nano-quadrotors. A sketch of this architecture is provided in Fig. 5.

We provide a set of subclasses that extend `RadioHandler`, namely `RadioHandlerFPQR`, `RadioHandlerFRPY` and `RadioHandlerXYZ`. More in detail, the first one sends to the nano-quadrotors thrust and angular-rate setpoints. The onboard firmware then evaluates a suitable control input using a PID controller, comparing the setpoint with gyroscope data. The second one instead sends thrust and angular setpoints. An onboard PID controller tracks these setpoints. The class leverages Bitcraze logging methods to retrieve onboard log data. These subclasses are tailored for precise control schemes and require control inputs to be sent at high communication rates. The `RadioHandlerXYZ` class instead sends position setpoints and ground truth information to the robots. These setpoints are handled onboard by the `higher_level_commander` implemented on the Crazyflie firmware. The communication of position setpoints can be performed at a low communication rate, in the order of a few Hertz. This fact reduces the number of messages on the communication links. Thus, this class is tailored for experimental scenarios with several quadrotors. Each `radio` node handles a group of Crazyflies and thus requires a list of radio URIs and a set of numerical identifiers to subscribe to the correct topics.

VI. Simulation Tools in CrazyChoir

In this section, we describe how to simulate a swarm of Crazyflies in CrazyChoir. First, we detail how to perform this task in conjunction with Webots, a realistic engine tailored for robot simulations. Then, we detail how to run a numerical simulation using a custom integrator and RViz visualization. Although we focus on Webots, users can extend the package to leverage other engines such as Gazebo.

A. Realistic Simulations Via Webots

CrazyChoir includes a set of functionalities, in the form of *plugins*, to interact with Webots. These plugins receive inputs from the higher-level classes (e.g., control inputs or waypoints coordinates) and map them to motor commands. The command mapping is evaluated using Python bindings of the Crazyflie firmware functions provided by Bitcraze. This allows the designer to have realistic feedback on the quadrotor behavior. Also, the designer could use this feature to test new firmware functions, e.g., embed control algorithm directly onboard. To properly simulate a robot in the Webots environment, also called *World*, the designer has to specify (i) the geometry of the robot, (ii) external features that enrich the model (e.g. sensors, cameras, and actuators) and (iii) plugins to control the actuators. The first requirement is achieved by importing a `.stl` file (*STereo Lithography* interface format). External features can be endowed by filling up a `.urdf` file (*Unified Robot Description Format*). Finally, to satisfy the third requirement, CrazyChoir provides a class of plugins to control the Crazyflie actuators. Hence, to endow our package with an exhaustive set of Webots plugins, we developed a parent class named `MotorCtrl`. At the beginning, the class sets out the whole Crazyflie physical equipment, making available GPS measurements and IMU detections. Moreover, the class initializes ROS 2 publishers and subscribers in order to interface Webots with CrazyChoir planners and controllers. The main class of plugins `MotorCtrl` is extended by two subclasses. The first subclass, `MotorCtrlFPQR`, takes as input references the desired thrust and angular rates. The computation of these quantities is explained in Section III. The mapping to motor torques is then performed using firmware Python bindings, thus using the same model implemented on the Crazyflie. The second subclass, `MotorCtrlXYZ`, leverages the functionalities of the trajectory planner coded in the firmware. Indeed, it receives desired position and yaw setpoints that are forwarded to the planner methods to evaluate a trajectory. As a result, the trajectory is elaborated by firmware bindings and motor commands are provided to the simulated Crazyflie.

B. Lightweight Simulations Via Numerical Integration

To allow users to perform simulations without additional, external software, we provide a dynamics simulator for the Crazyflie nano-quadrotor running an Explicit Runge-Kutta method. The integration is performed at 100 Hz, and is based on the following dynamics

$$\dot{\mathrm{p}} = \mathrm{v} \tag{1}$$

$$\dot{\mathrm{v}} = g z_W - \frac{u_1}{m} R(\eta) z_W - \frac{1}{m} R(\eta) A R(\eta)^\top \mathrm{v} \tag{2}$$

$$\dot{\eta} = [u_2 \; u_3 \; u_4]^\top \tag{3}$$

Here, $\mathrm{p} \in \mathbb{R}^3$ is the position of the quadrotor in the world frame $\mathcal{F}_W = \{x_W, y_W, z_W\}$, $\mathrm{v} \in \mathbb{R}^3$ is its linear velocity, and $m = 0.027$ Kg is the Crazyflie mass. The rotation matrix is denoted by $R \in SO(3)$, and the vector $\eta = [\varphi, \theta, \psi]^\top$ stacks the Euler's angles, namely, roll, pitch, and yaw. The Crazyflie allows the user to send thrust, roll rate, pitch rate, and yaw rate as control actions. Thus, the control inputs in the considered dynamics are collected by the vector $u = [u_1 \; u_2 \; u_3 \; u_4]^\top \in \mathbb{R}^4$, with $u_1 \in \mathbb{R}$ denoting the thrust and $u_2, u_3, u_4 \in \mathbb{R}$ the angular rates (according to the $Z - Y - X$ extrinsic Euler representation). The term $RAR^\top \mathrm{v}$ in (2) refers to the rotor drag effect as modeled in [27]. The numerical values of the drag parameters have been chosen according to the discussion in [28]. Thanks to the modular structure of the package, the user can extend or modify the proposed integrator to simulate more complex dynamics. To visualize the quadrotor motion, we also provide a visualization utility based on RViz.

VII. CrazyChoir Hands-On Example: Bearing-Based Formation Control

This section aims at introducing the user to our toolbox. To this end, we show how to simulate the bearing-based distributed formation control scheme as in [29] over a swarm of Crazyflie. We start by providing the problem set-up and the classes needed for its implementation. Then, we show how to run the simulation both in RViz and Webots. Finally, we show how CrazyChoir allows users to easily switch from simulation to experiment and run the same setting on a real swarm of Crazyflies.

A. Problem Set-up

We consider a network of N quadrotors, divided into N_l leaders and $N - N_l$ followers. The objective is to control the followers to deploy a desired formation in the space, while the leaders stand still in their positions. In the considered distributed set-up, each quadrotor can communicate with a set $\mathcal{N}_i$ of neighboring quadrotors. The formation is defined by a set of *bearings* $g_{ij}^\star$ for all the couples (i, j) with $i \in \{1, \ldots, N\}$ and $j \in \mathcal{N}_i$. Authors in [29] consider double-integrator systems in the form $\ddot{\mathrm{p}}_i^{di} = u_i^{di}$. Leaders apply $u_i^{di} = 0$, while followers implement

$$u_i^{di} = -\sum_{j \in \mathcal{N}_i} P_{g_{ij}^\star} [k_\mathrm{p}(\mathrm{p}_i - \mathrm{p}_j) + k_\mathrm{v}(\mathrm{v}_i - \mathrm{v}_j)], \tag{4}$$

where $P_{g_{ij}^\star} = I_3 - g_{ij}^\star g_{ij}^{\star\top}$ and I_3 is the 3×3 identity matrix. Notice that this input cannot be directly fed to the quadrotors. In our example, we use the control input u_i^{di} as a desired

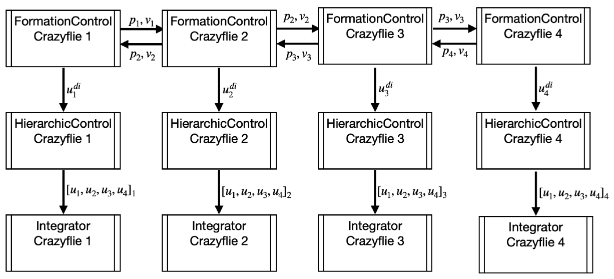

Fig. 6. Class communication diagram. Rectangles represent different classes, each one handled by a different node. Arrows represent directional communications, which are performed leveraging ROS 2 topics. On each arrow, exchanged data is reported.

acceleration profile for the i-th Crazyflie. This profile is then tracked using a flatness-based controller (cf. Section III).

B. Software Implementation

To implement the distributed bearing-based formation control scheme, we need three classes. The first one implements the distributed control law in (4). This is done in a specific method as follows

```
u = np.zeros(3)
if not self.is_leader:
   for j, neigh_pose in neigh_data.items():
      err_pos = self.current.position -
          neigh_pose.position
      err_vel = self.current_pose.velocity -
          neigh_pose.velocity
      u += - P_i[j] @ (self.kp * err_pos +
          self.kv * err_vel)
```

The rest of the CrazyChoir ecosystem will handle ROS 2 communication among nodes so that the user solely has to implement the desired control law. The second required class needs to track the acceleration profile generated by the distributed control scheme. This is developed according to Section III. For this example, we need a control law tracking the acceleration profile generated by the guidance class. Then, we generate an angular velocity input according to a geometric control law. To this end, we have to specify that the controller sends the Crazyflie angular rates and thrust as control inputs. This can be implemented as follows

```
position_ctrl = FlatnessAccelerationCtrl()
attitude_ctrl = GeometryAttitudeCtrl()
sender = FPQRSender()
desired_accel = AccelerationTraj()

controller = HierarchicalController(
               pos_strategy=position_ctrl,
               attitude_strategy=attitude_ctrl,
               command_sender=sender,
               traj_handler=desired_accel)
rclpy.spin(controller)
```

The last class instead simulates the Crazyflie dynamics. In Fig. 6, we provide a *communication diagram* highlighting the classes involved in this use-case and their interactions. For the formation control problem, the guidance node of each quadrotor sends to its neighbors only two vectors (i.e., position and velocity) at 100 Hz rate.

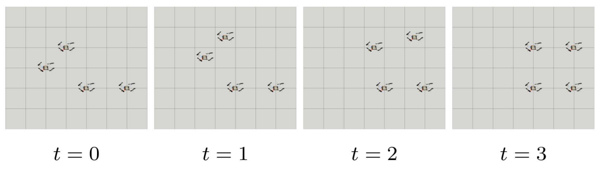

Fig. 7. Sequence of images from the RVIZ toolbox at different subsequent time instants. Static quadrotors are the leaders, while the moving ones are the followers. As time progresses, quadrotors reach the desired formation.

C. Running and Visualize the Simulation

To easily run the simulation, we leverage the ROS 2 launching system. Specifically, it is necessary to write a Python script specifying which nodes have to be run. It is necessary to set-up instances of the `DistributedControlGuidance`, `HierarchicalController`, and `CFIntegrator` classes for each quadrotor in the network. The launch file also allows the user to specify additional parameters, e.g., robot initial conditions and the communication graph. As an example, let N be the number of Crazyflie in the desired simulation. To instantiate a controller for each quadrotor, the code can be formatted as follows.

```
def generate_launch_description():
  # ... define graph and init. positions

  launch_description = [] # list overall nodes
  for i in range(N): # for each quadrotor add
       needed nodes
    # ... set-up distributed feedback node
    # ... set-up integrator node

    launch_description.append(Node(
       package=’crazychoir_examples’,
       node_executable=’crazychoir_controller’,
       namespace=’agent_{}’.format(i),
       parameters=[{’cf_id’: i}]))

  return LaunchDescription(launch_description)
```

In order to visualize the simulation, we included in our toolbox a script to visualize the Crazyflie team in RVIZ. More in detail, we provide a class that receives the pose information from the `CFIntegrator` classes and sends suitable messages to RVIZ for visualization. A set of snapshots for a simulation with $N = 4$ robots is provided in Fig. 7. To run the same setting in the Webots simulator, it is solely necessary to substitute the `CFIntegrator` classes with the Webots simulation architecture. To emphasize the potential of Webots, we simulate 30 quadrotors drawing a grid. In this setting, we leverage the distributed paradigm allowed by CrazyChoir employing three workstations to handle the swarm, each of them endowed with Ubuntu 20.04 and ROS 2 Foxy. One handles the physics engine, while each one handles a subset of the quadrotors. We implemented a keep-out zone to ensure that, as soon as a quadrotor flies beyond a certain limit area, it is immediately shut down. Snapshots from the Webots simulation are depicted in Fig. 8.

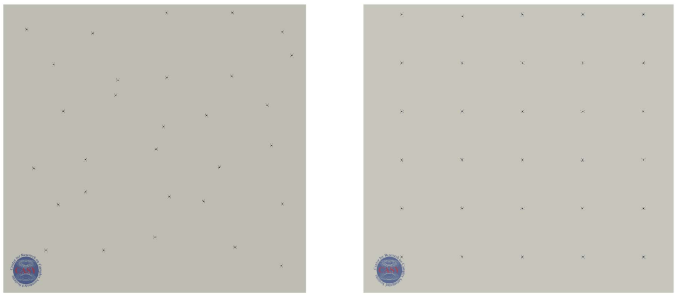

Fig. 8. Beginning of the simulation (left) and end of the simulation (right) of the bearing formation control problem in Webots.

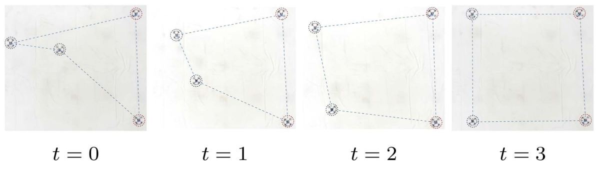

Fig. 9. Snapshots from an experiment at different time instants. Leaders are circled in red, followers in blue.

D. Running the Experiment

In this section, we show how CRAZYCHOIR meets prototyping standards, allowing the user to switch from simulation and experiment by changing only a few lines of code. Indeed, to run an experiment, it is necessary to include in the launch file the Crazyflie URIs and the radio nodes as described in Section V. These few lines of code shall substitute the ones related to Webots or RVIZ. After that, it is only required to check that nodes subscribe to the topics transporting Vicon data. To run our experiments, we used the `ros2-vicon-receiver` package[1] to receive data from our Vicon system. Snapshots from an experiment with 4 quadrotor are provided in Fig. 9. Furthermore, we provide an experiment in which leaders track a hand-written trajectory, leveraging the tools provided by the GUI (see Section IV-A). More precisely, we draw a circle with a radius of 0.4 m, and we execute this trajectory for 40˜ s. To realize this scenario, we changed only a few settings in the leader control node. We modified the control strategy to `FlatnessPositionCtrl()`, which also processes position and velocity references, and the trajectory handler to `FullStateTraj()`. In addition, it is necessary to launch a planner node for the leaders, with the goal of evaluating the splines retrieved by the drawing done in the GUI. A video is available as supplementary material.[2]

VIII. CRAZYCHOIR EXPERIMENTS ON TRAJECTORY TRACKING AND PICKUP AND DELIVERY

We now provide two additional experiments. First, a quadrotor tracks a hand-written trajectory. Then, a swarm of Crazyflies executes a pickup-and-delivery task. All the needed nodes are executed on a workstation (Intel i9, Nvidia RTX 4000, Ubuntu 20.04, ROS 2 Foxy) equipped with multiple radio transmitters, each handling two quadrotors. Moreover, a second workstation (Intel Xeon, Windows 7) handles the Vicon Tracker software.

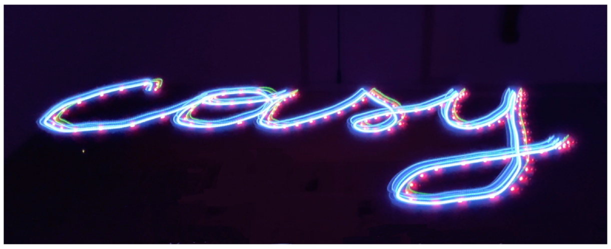

Fig. 10. Long-term exposure snapshot from an experiment in which a quadrotor draws a hand-drawn trajectory.

A. Tracking of a Hand-Written Trajectory

We provide experimental results for a Crazyflie tracking a hand-drawn trajectory. The path has been implemented using the graphical interface introduced in Section IV-A, see also Fig. 4. The chosen trajectory, i.e., the name of our laboratory "*Casy*", is interpolated by a set of Python methods provided by the proposed planning module. This trajectory is tracked by an off-board controller that elaborates the desired references and, leveraging the `RadioFPQR` class, forwards the thrust and angular-rates commands to the Crazyflie. Fig. 10 shows a long-exposure snapshot from the experiment. A video is available as supplementary material to the paper.[3]

B. Multi-Robot Pickup and Delivery

By exploiting the distributed optimization features detailed in Section IV-B, we choose an experimental set-up that deals with a distributed optimization problem.

1) Problem Set-up: We employ CRAZYCHOIR to perform experiments for a cooperative pickup-and-delivery scenario involving $N = 6$ Crazyflie. In this setting, quadrotors have to pick up goods at given locations and delivery them in other locations. As a further constraint, each quadrotor has a certain load capability and can pick up only some goods. The goal is to minimize the overall travel distance, while guaranteeing that all the goods are successfully delivered. This problem can be formulated as a large-scale mixed-integer linear program, i.e. an optimization problem with a large number of both binary and continuous variables. We refer the reader to [30] for a description of the problem.

2) Software Implementation: Similarly to Section VII, we exploit the modularity features of CRAZYCHOIR to implement the considered scenario on a real swarm of Crazyflies. Firstly, we take advantage of the `CFGuidance` class, creating a child subclass named `PDVRPGuidance` which computes the optimal pickup-and-delivery locations for each quadrotor. In this subclass, we exploit the functionalities of DISROPT [23] to model and solve the distributed optimization problem. As opposed to the simulative example provided in Section VII, in this experiment, we exploit the onboard controller, by sending position references to each Crazyflie. Thanks to the `RadioHandlerXYZ`

[1]https://github.com/OPT4SMART/ros2-vicon-receiver
[2]The video is also available at https://youtu.be/mJ1HOquR-vE

[3]The video is also available at https://youtu.be/wX2r55WdZ9g

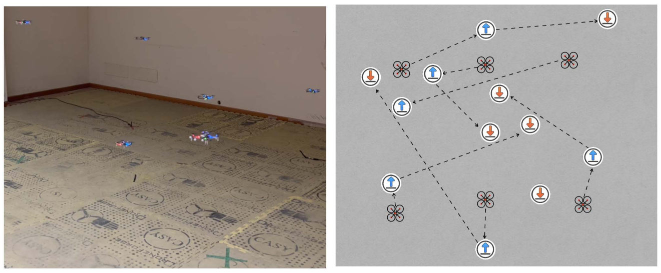

Fig. 11. The left snapshot illustrates the Crazyflies reaching the delivery locations, while the right scheme shows the pickup points (blue arrows) and the delivery points (orange arrows). Dashed lines represent optimal paths.

class, the position references given by the solution of the pickup-and-delivery problem are forwarded to the onboard controller of the Crazyflie, which is in charge to generate and track the desired trajectory. A snapshot from an experiment is in Fig. 11. A video is available as supplementary material[3]

IX. Conclusion

In this letter, we introduced CrazyChoir, a ROS 2 toolbox specifically tailored for swarms of Crazyflie nano-quadrotors. The package allows the user to perform realistic simulations of Crazyflie swarms using Webots and firmware bindings. Also, the toolbox interfaces with radio dongles to perform experiments on real nano-quadrotors. We provide several template classes to perform control, planning, and cooperative decision-making. Thanks to these functionalities, users can easily extend the package by implementing their own algorithms. Illustrative simulations and experiments have been provided to assess the potentiality of the toolbox.

References

[1] M. Quigley et al., "ROS: An open-source robot operating system," in *Proc. ICRA Workshop open Source Softw.*, 2009, vol. 3, no. 3.2, p. 5.

[2] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *Proc. 13th Int. Conf. Embedded Softw.*, 2016, pp. 1–10.

[3] J. A. Preiss, W. Honig, G. S. Sukhatme, and N. Ayanian, "Crazyswarm: A large nano-quadcopter swarm," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2017, pp. 3299–3304.

[4] W. Hönig and N. Ayanian, "Flying multiple UAVs using ROS," in *Robot Operating System*. Berlin, Germany: Springer, 2017, pp. 83–118.

[5] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics*. Berlin, Germany: Springer, 2018, pp. 621–635.

[6] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A flexible quadrotor simulator," in *Proc. Conf. Robot Learn.*, PMLR, 2021, pp. 1147–1157.

[7] V. Grabe, M. Riedel, H. H. Bülthoff, P. R. Giordano, and A. Franchi, "The TeleKyb framework for a modular and extendible ROS-based quadrotor control," in *Proc. IEEE ECMR*, 2013, pp. 19–25.

[8] J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, and O. V. Stryk, "Comprehensive simulation of quadrotor UAVs using ROS and Gazebo," in *Proc. IEEE 3rd Int. SIMPAR*, 2012, pp. 400–411.

[9] K. Kumar, S. I. Azid, A. Fagiolini, and M. Cirrincione, "Erle-copter simulation using ROS and Gazebo," in *Proc. IEEE Mediterranean Electrotechnical Conf.*, 2020, pp. 259–263.

[10] G. Silano, E. Aucone, and L. Iannelli, "Crazys: A software-in-the-loop platform for the crazyflie 2.0 nano-quadcopter," in *Proc. IEEE Mediterranean Conf. Control Automat.*, 2018, pp. 1–6.

[11] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, "RotorS–A modular Gazebo MAV simulator framework," in *Robot Operating System*. Berlin, Germany: Springer, 2016, pp. 595–625.

[12] M. Nithya and M. Rashmi, "Gazebo-ROS-simulink framework for hover control and trajectory tracking of crazyflie 2.0," in *Proc. IEEE Region 10 Conf.*, 2019, pp. 649–653.

[13] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Sci. Robot.*, vol. 7, no. 66, 2022, Art. no. eabm6074.

[14] L. Puck et al., "Distributed and synchronized setup towards real-time robotic control using ROS2 on Linux," in *Proc. IEEE Int. Conf. Automat. Sci. Eng.*, 2020, pp. 1287–1293.

[15] K. Belsare et al., "Micro-ROS," in *Robot Operating System (ROS) the Complete Reference*, vol. 7, Berlin, Germany: Springer, 2023, pp. 3–55.

[16] E. Erõs, M. Dahl, K. Bengtsson, A. Hanna, and P. Falkman, "A ROS2 based communication architecture for control in collaborative and intelligent automation systems," *Procedia Manuf.*, vol. 38, pp. 349–357, 2019.

[17] M. Reke et al., "A self-driving car architecture in ROS2," in *Proc. IEEE Int. SAUPEC/RobMech/PRASA Conf.*, 2020, pp. 1–6.

[18] T. K. Kaiser, M. J. Begemann, T. Plattenteich, L. Schilling, G. Schildbach, and H. Hamann, "ROS2SWARM-A ROS 2 package for swarm robot behaviors," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2022, pp. 6875–6881.

[19] S. Mai, N. Traichel, and S. Mostaghim, "Driving swarm: A swarm robotics framework for intelligent navigation in a self-organized world," in *Proc. IEEE Int. Conf. Robot. Automat*., 2022, pp. 01–07.

[20] "Crazyswarm2," [Online]. Available: https://github.com/IMRCLab/crazyswarm2/

[21] A. Testa, A. Camisa, and G. Notarstefano, "ChoiRbot: A. ROS 2 toolbox for cooperative robotics," *IEEE Robot. Automat. Lett.*, vol. 6, no. 2, pp. 2714–2720, Apr. 2021.

[22] O. Michel, "Cyberbotics Ltd. Webots: Professional mobile robot simulation," *Int. J. Adv. Robot. Syst.*, vol. 1, no. 1, pp. 39–42, 2004.

[23] F. Farina, A. Camisa, A. Testa, I. Notarnicola, and G. Notarstefano, "DISROPT: A python framework for distributed optimization," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 2666–2671, 2020.

[24] E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Munich, Germany: Pearson Deutschland GmbH, 1995.

[25] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2011, pp. 2520–2525.

[26] T. Lee, M. Leok, and N. H. McClamroch, "Geometric tracking control of a quadrotor UAV on SE (3)," in *Proc. IEEE Conf. Decis. Control*, 2010, pp. 5420–5425.

[27] J.-M. Kai, G. Allibert, M.-D. Hua, and T. Hamel, "Nonlinear feedback control of quadrotors exploiting first-order drag effects," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 8189–8195, 2017.

[28] J. Förster, "System identification of the crazyflie 2.0 nano quadrocopter," B.S. thesis, ETH Zurich, Zrich, Switzerland, 2015.

[29] S. Zhao and D. Zelazo, "Translational and scaling formation maneuver control via a bearing-based approach," *IEEE Trans. Control Netw. Syst.*, vol. 4, no. 3, pp. 429–438, Sep. 2017.

[30] A. Camisa, A. Testa, and G. Notarstefano, "Multi-robot pickup and delivery via distributed resource allocation," *IEEE Trans. Robot.*, vol. 39, no. 2, pp. 1106–1118, Apr. 2023.

Open Access funding provided by 'Alma Mater Studiorum - Università di Bologna' within the CRUI CARE Agreement