A. Testa, A. Camisa and G. Notarstefano, “ChoiRbot: A ROS 2 Toolbox for Cooperative Robotics”, in IEEE Robotics and Automation Letters, vol. 6, no. 2, pp. 2714-2720, April 2021

The final published version is available online at:

http://dx.doi.org/10.1109/LRA.2021.3061366

ChoiRbot: A ROS 2 Toolbox for Cooperative Robotics*

Andrea Testa†, Andrea Camisa†, Giuseppe Notarstefano

***Abstract*— In this paper, we introduce CHOIRBOT, a toolbox for distributed cooperative robotics based on the novel Robot Operating System (ROS) 2. CHOIRBOT provides a fully-functional toolset to execute complex distributed multi-robot tasks, either in simulation or experimentally, with a particular focus on networks of heterogeneous robots without a central coordinator. Thanks to its modular structure, CHOIRBOT allows for a highly straight implementation of optimization-based distributed control schemes, such as distributed optimal control, model predictive control, task assignment, in which local computation and communication with neighboring robots are alternated. To this end, the toolbox provides functionalities for the solution of distributed optimization problems. The package can be also used to implement distributed feedback laws that do not need optimization features but do require the exchange of information among robots. The potential of the toolbox is illustrated with simulations and experiments on distributed robotics scenarios with mobile ground robots. The CHOIRBOT toolbox is available at `https://github.com/OPT4SMART/choirbot`.**

I. INTRODUCTION

There is an increasing interest towards several applications in cooperative robotics, such as task assignment/allocation, model predictive control, formation control. In peer-to-peer robotic networks it is often desirable to solve these problems without a coordinating unit. Robots in the network, relying on their limited knowledge of the problem data, have to exploit local computation and communication capabilities to solve the complex task, which often requires also the solution of optimization problems. Since its introduction, the Robot Operating System (ROS), [1], has gained popularity among robotics researchers as an open source framework for the development of robotics applications. Nowadays, ROS 2 is extending ROS capabilities, paving the way to real-time control systems and large-scale distributed architectures, [2]. While several theoretical frameworks have been proposed to solve optimization problems over networks of cooperating robots, see, e.g. the survey [3] and references therein, few architectures have been proposed to simulate and run experiments on teams of heterogeneous robots communicating according to arbitrary graphs and aiming at the cooperative solution of complex tasks.

A. Testa, A. Camisa and G. Notarstefano are with the Department of Electrical, Electronic and Information Engineering, University of Bologna, Bologna, Italy. {`a.testa`, `a.camisa`, `giuseppe.notarstefano`}`@unibo.it`.

† These authors contributed equally to this work.

*This result is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 638992 - OPT4SMART).

A. Related Work

The ROS framework has been used as a building brick for a plethora of robotics applications, and several frameworks have been proposed to simulate and implement control and planning tasks. Authors in [4] propose a constraint-based task specification for robot controllers, defining a task specification language and the related controller. The framework in [5] allows users to create task plans for collaborative robots. Papers [6] and [7] instead propose architectures to simulate and control UAVs, while [8] allows users to write ROS code on a browser and run it on remote robots. On this purpose, it is worth mentioning the Robotarium, [9], a proprietary platform that allows to test and run control algorithms on robotic teams. These frameworks often leverage optimization routines to perform specific tasks. To name a few, the framework in [10] deals with trajectory optimization in dynamic environments, the architecture in [11] approaches search-and-rescue tasks by means of particle swarm optimization while [12] is suited for task allocation scenarios. Finally, in the recent years, robotics researchers started to develop robotic architectures based on the novel ROS 2 framework. Authors in [13] and [14] consider a framework for collaborative robotics, while the paper in [15] discusses an architecture for self-driving cars. The aforementioned frameworks are typically optimized for a specific task, and most of them focus on single-robot systems. Moreover, the communication in multi-robot networks is often neglected or simulated by means of the resource-demanding all-to-all communication.

B. Contributions

In this paper, we introduce a novel ROS 2 toolbox for cooperative robotics named CHOIRBOT. This toolbox, written in Python, exploits the new functionalities of ROS 2 and provides a comprehensive set of libraries to facilitate multi-robot simulations and experiments. The main focus of CHOIRBOT is on peer-to-peer network of robots, where each robot has its own processor and is able to communicate with the neighboring units according to a user-defined graph, possibly time-varying or with unreliable communication links. Importantly, CHOIRBOT does not require a central coordinating unit and, as such, allows for fully distributed control schemes. In order to maximize extendability and ensure a broad applicability, the toolbox is designed according to a modular structure. In such a way, we demonstrate that several applications of interest are easy to implement using CHOIRBOT, and we discuss in detail a few use cases that have been tested either in simulation or experimentally, namely dynamic task assignment, formation control and containment in leader-follower networks. A distinctive feature of CHOIRBOT is the ability to run general-purpose distributed optimization algorithms. To this end, the DISROPT package [16] has been fully integrated, allowing both for the semantic modeling and solution of optimization problems (locally at a robot) and for the execution of distributed optimization algorithms (cooperatively across all the robots).

The paper is organized as follows. In Section II, we provide a high-level description of the software architecture, while in Section III, additional software details are provided. Section V discusses two complex scenarios implemented in CHOIRBOT. A basic use case together with implementation details is discussed in IV. Finally, simulation and experimental results are provided in Section VI.

II. ARCHITECTURE DESCRIPTION

In this section, we describe the high-level architecture of CHOIRBOT. As already mentioned, the toolbox is modular and its blocks are intended to be combined as needed. We first focus on an overall description of the software and then we describe each block separately.

A. Overview of the Software

CHOIRBOT is written in Python and is based on ROS 2. An important feature of the new ROS 2 architecture is that no master entity is present [2]. Therefore, CHOIRBOT processes are truly distributed since no message broker is required. This fits perfectly with the goal of CHOIRBOT of providing a platform for cooperative robotics over peer-to-peer networks without a central coordinating unit.

The toolbox is structured in a three-layer architecture. Specifically, there is a *Team Guidance* layer, a *RoboPlanning* layer and a *RoboControl* layer. The Team Guidance layer is responsible for taking high-level decisions and for managing the robot lifecycle. The Team Guidance layer uses communication with neighbors in order to perform its tasks. The Roboplanning and Robocontrol layers are responsible for lower-level control actions as driven by the upper layer. Since our main goal is to ease the design and implementation of optimization-based distributed control schemes, the central focus of CHOIRBOT is the Team Guidance layer. In particular, the toolbox provides boilerplate code for distributed computation, i.e. communication among robots over a graph, coordination and optimization algorithms. Apart from a few specific scenarios, we do not provide comprehensive RoboPlanning and RoboControl features, which are robot specific and can be already implemented with existing tools.

In CHOIRBOT, each robotic agent in the network executes three ROS 2 processes, one for each layer. Each process is also associated to a separate ROS 2 node. To guarantee flexibility and code reusability, layers are implemented as Python classes. In the remainder of this section, we provide details about the three layers. A graphical illustration of the software architecture is represented in Figure 1.

B. Team Guidance Layer

The Team Guidance layer is the main entry point of the package. Here we provide an introductory description, while a more detailed analysis of this part of software is delayed to Section III.

The Team Guidance layer is modeled with a `Guidance` class, whose purpose is only to retrieve the basic information regarding the robotic agent (passed as ROS 2 parameters), to create an instance of the `Communicator` class to be described in Section III-A and to subscribe to the topic where the robot pose is published. In the current version of CHOIRBOT, the robot position can be either communicated by a Vicon motion tracking system or by a simulator (such as Gazebo). This basic version of the `Guidance` class is abstract in that it does not implement any guidance logic. However, the package currently provides two possible usable extensions. The first one (implemented as the `OptimizationGuidance` class) also provides optimization-related functionalities and is the starting point for any optimization-based distributed control scheme such as task assignment / allocation, optimal control, model predictive control. The toolbox can also be used to implement simpler distributed feedback laws that do not require the solution of optimization problems. To this end, we also provide a second extension of the `Guidance` class (implemented as the `DistributedControlGuidance` class), where robots repeatedly exchange their current position with their neighbors and compute a control input based on their position and the received positions. The actual form of the control input depends on the specific scenario and is left as an unimplemented method of the class. Currently implemented algorithms are rendezvous, containment, formation control.

C. RoboPlanning, RoboControl and RoboIntegration

Within the CHOIRBOT toolbox, the trajectory planning layer and the control layer do not communicate with the neighboring robotic agents. Instead, each robot has its own planning/control stack, which depends on the dynamics of the single robot and on the chosen control strategy. The base class for trajectory planners is `Planner`. We provide a point-to-point planner (in 2D or 3D), to be used in conjunction with controllers that are able to steer the robot towards a specified point in the space (e.g. for mobile ground robots). We also provide trajectory planners for aerial robots, which generate quad-rotor trajectories either to reach a desired constant speed or to reach a target point. As for controllers, the base class is `Controller`. For mobile ground robots, we provide two unicycle controllers that can be used either to reach a target point or to reach a desired velocity. For quadrotors, we provide a controller to stabilize the trajectories generated by the planners.

The CHOIRBOT toolbox is well integrated with simulation environments. In this regard, the `Gazebo` simulator [17] can be used. In case the user does not want to use external tools, we also provide a dynamics integration layer, named *RoboIntegration*, which can e.g. be used in conjunction with `Rviz` for visualization. Currently, there are numerical integrators for point-masses, unicycle robots and quadrotors. New integrators with custom dynamics can be written by extending the `Integrator` class.

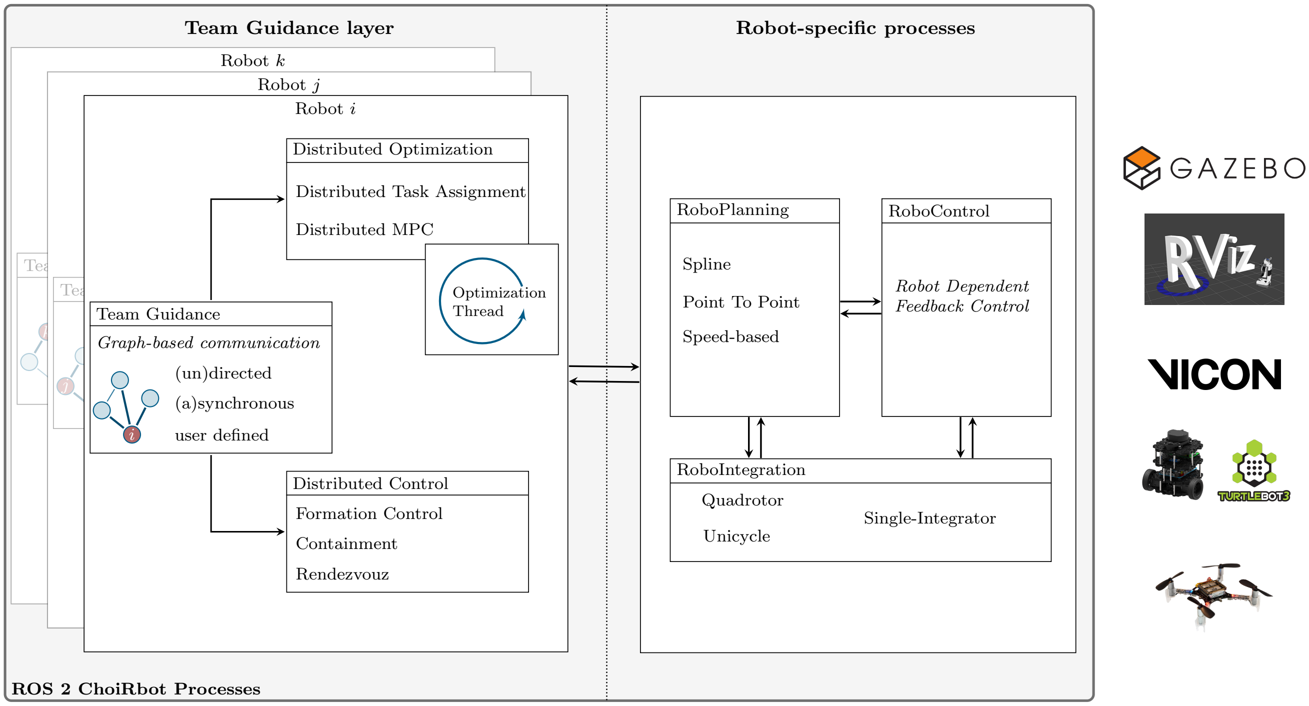

Fig. 1. CHOIRBOT architecture. Each robot communicates with neighboring robotic agents thanks to the Team Guidance Layer. The Team Guidance class handles the communication with planning and control utilities, which are specific for each robot.

III. EXPLORING THE TEAM GUIDANCE LAYER

As already mentioned, almost all functionalities of CHOIRBOT involve the Team Guidance layer. In this section, we explore in more detail this part of software. We begin by analyzing the first important feature of the Team Guidance class, i.e. graph-based communication. Then, we analyze the two essential usages of the Team Guidance layer, represented by the `OptimizationGuidance` and the `DistributedControlGuidance` classes.

A. Graph-based Communication

At the core of every distributed control scheme is the graph-based communication among the robotic agents. This feature is provided in CHOIRBOT at the team guidance layer by three different `Communicator` classes, which model static, time-varying and unreliable ("best-effort") communication, respectively. By exploiting the novel Quality of Service introduced in ROS 2, these classes can handle synchronous/asynchronous and undirected/directed communication among the robots, as summarized in Table I. To achieve this, the `Communicator` class only requires specification of the in- and out-neighbors of the robots and takes care of managing the necessary ROS 2 topics and subscriptions. To maintain the class interface of the class semantically clear, the method names correspond to the specific actions that can be performed: `send`, `receive`, `asynchronous_receive`, `neighbors_exchange` (i.e. simultaneous `send`/`receive`). Depending on the specific application, the `Communicator` classes can be used to handle scenarios in which the communication links change on the fly due to, e.g., limited communication range and/or energy consumption constraints.

Class name	Directed Undirected	Time Varying	Synch.	Asynch.	Reliable
`StaticComm.`	✓		✓	✓	✓
`TimeVaryingComm.`	✓	✓	✓	✓	✓
`BestEffortComm.`	✓	✓		✓	

TABLE I

FEATURES AND LIMITATIONS OF THE IMPLEMENTED COMMUNICATORS.

An important feature is that it is not necessary to declare the message types to be exchanged among the robots, which is typically needed for all ROS applications. Instead, we fixed the type of message to `std_msgs/ByteMultiArray` and the `Communicator` class handles (de)serialization of the exchanged message to (from) a byte sequence through the Python `dill` package. This allows the user to exchange nearly each type of message (vector, matrices, text, dictionaries, images, etc.) and even to change it at runtime without declaring what type of message must be sent.

B. Distributed Optimization

A consistent part of the toolbox is devoted to providing optimization-related functions. The main entry point is the `OptimizationGuidance` class. Since numerical optimization algorithms may require a certain number of iterations to converge, these computations are delegated to a separate thread running in the `OptimizationThread` class. This allows the ROS 2 guidance process to continue elaborating callbacks even though an optimization is in progress. This separate thread class, which is started by the `OptimizationGuidance` class, allows the user to start/stop the optimization processes on demand. At the end of an optimization process, the method `optimization_ended` is called. This method is supposed to be overridden by the user with the specific guidance logic, e.g. by retrieving the optimization results and by using them as needed. Depending on the scenario at hand, it may be required that the problem data is re-evaluated in order to avoid outdated information. The actual body of the optimization algorithm is left unimplemented so as to allow the user to implement the desired method. Currently, we provide implementations for distributed task assignment and distributed Model Predictive Control scenarios, which are analyzed in detail in Section V.

The framework allows one to model and to solve both local optimization problems (at a robot) and distributed optimization problems (involving the whole network). To this end, the Team Guidance layer has been integrated with the DISROPT package [16], which provides a large number of already implemented distributed optimization schemes and allows for the semantic modeling of (distributed) optimization problems. Since the `Communicator` classes of CHOIRBOT are fully compatible with DISROPT, the distributed algorithms implemented in DISROPT can be used seamlessly, i.e., with no further modifications.

C. Distributed Feedback Control

The toolbox is designed to support optimization-based distributed control algorithms, however simpler distributed feedback laws can also be implemented by using a subset of the toolbox features. The `DistributedControlGuidance` class implements a general communication and control structure allowing for the implementation of such feedback laws. To give an idea, let us report an excerpt of the main routine of the class, which is executed with a user-chosen frequency:

```
# exchange current position with neighbors
data = self.communicator.neighbors_exchange(
         self.current_pose.position,
         self.in_neighbors,
         self.out_neighbors, False)

# compute input
u = self.evaluate_velocity(data)

# send input to planner/controller
self.send_input(u)
```

In words, the class first exchanges the current position with the neighbors, then computes a velocity profile according to the exchanged data and to the considered feedback law, and finally publishes the control input on a topic. Despite its simplicity, this basic structure can be used for several distributed control algorithms such as containment and formation control. Simulation results can be found in Section VI.

IV. BASIC USAGE EXAMPLE

In this section, we consider a toy example that allows us to show the implementation of a basic distributed cooperative robotics scenario in CHOIRBOT. Specifically, we consider containment in leader-follower networks as described in [18]. The mathematical formulation is as follows. Robots communicate according to a time-varying undirected graph $\mathcal{G}^t = (V, \mathcal{E}^t)$, where $V = \{1, \dots, N\}$ is the set of vertices and $\mathcal{E}^t \subseteq V \times V$ is the set of edges at time t. We denote by $\mathcal{N}_i^t$ the set of neighbors of each robot i at time t. Robots are partitioned in two groups, namely leaders and followers. The goal for the followers is to converge to the convex hull of the leaders' positions. To this end, the robots implement the dynamics

$$\dot{x}_i(t) = 0 \qquad \text{(leaders)}, \tag{1a}$$

$$\dot{x}_i(t) = \sum_{j \in \mathcal{N}_i^t} \big(x_j(t) - x_i(t)\big) \qquad \text{(followers)}. \tag{1b}$$

A. Main Software Components

This scenario can be easily implemented in the proposed architecture by means of two classes in the Team Guidance Layer and in the RoboIntegration layer. The first one is an extension of the `DistributedControlGuidance` class discussed in Section III-C and inherits the distributed feedback control structure. We simply override the `evaluate_velocity` method (cf. Section III-C) in order to encode the control law in (1) as follows:

```
u = np.zeros(3)
if not self.is_leader:
   for pos_ii in neigh_data.values():
      u += self.containment_gain*(pos_ii -
           self.current_pose.position)
return u
```

The second class instead extends the functionalities of the base class `Integrator`. In particular, it is sufficient to implement the classical forward Euler method for single-integrator dynamics:

```
self.current_pos += self.samp_time * self.u
```

B. Visualizing the Evolution

Since we are not using an external simulation tool, we need a means to visualize the results graphically. To achieve this, it is possible to rely on the ROS 2 toolbox RVIZ, in which we associate each robot with a circle moving on the (x, y) plane. To achieve this, the toolbox provides a class that receives the pose from the integration layer and forwards it to the RVIZ visualizer with a user-defined frequency.

C. Running the Simulation

The final simulation can be run by writing a ROS 2 launch file with all the required nodes. Note that these three nodes must be executed by each of the robotic agents in the network. The launcher file is also responsible for declaring the initial positions of the robots and the communication graph, specified as a binary adjacency matrix.

In order to differentiate the ROS 2 nodes of each robot, robot-specific parameters must be passed as ROS 2 parameters at the time of spawning. For instance, the Team Guidance classes are spawned in the launch file as follows:

```python
def generate_launch_description():
  # .. initialization of adjacency matrix and
       initial positions

  list_description = []
  for i in range(N):
    # .. initialize in_neighbors, out_neighbors,
         is_leader

    list_description.append(Node(
      package='choirbot_examples',
          node_executable='choirbot_containment',
      node_namespace='agent_{}'.format(i),
      parameters=[{'agent_id': i, 'N': N,
          'in_neigh': in_neighbors, 'out_neigh':
          out_neighbors, 'is_leader':
          is_leader}]))

  return LaunchDescription(list_description)
```

A simulation with e.g. $N = 6$ robots can be run by executing the following command:

```
ros2 launch choirbot_examples
    containment.launch.py -n 6
```

We consider a scenario with three followers and three leaders spanning a triangle. In order to save energy, at each time instant robots decide whether to communicate their position to neighbors according to a certain probability, thus the neighbors change at each communication round. The evolution of the robot positions is reported in Figure 2.

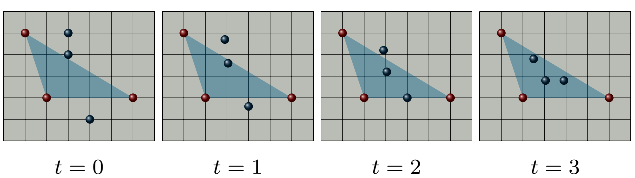

Fig. 2. Sequence of images from the RVIZ toolbox at different subsequent time instants. Red spheres represent leader robots, while the blue ones represent the followers. As time progresses, followers enter the convex hull of the leaders' positions, depicted with the cerulean triangle.

V. Implemented Complex Scenarios

In this section, we discuss in detail two complex cooperative robotic scenarios that have been implemented in ChoiRbot, i.e. distributed dynamic task assignment and distributed model predictive control. In both scenarios we employ the `OptimizationGuidance` class to leverage optimization capabilities.

A. Distributed Dynamic Task Assignment

Problem formulation: In this scenario, we assume there is a set of tasks that must be performed by a team of robots. In order to choose the final assignment, robots must self-coordinate by using their communication capabilities. The assignment problem can be formulated as an integer optimization problem with unimodularity properties, which is eventually reformulated into a distributed linear program [19]. Formally, assume there are N robots (indexed by i) and N tasks (indexed by k). A scalar c_{ik} represents the cost incurred by robot i when servicing task k. The goal is to find the optimal assignment, i.e. to assign each robot i to exactly one task j such that the total incurred cost is minimized. We focus on the challenging scenario in which tasks are not known a-priori but arrive dynamically. While robots perform the previously assigned tasks, a new task can arrive and robots re-execute the distributed optimization algorithm to compute the new optimal assignment.

Implementation: In ChoiRbot, there is a number of classes to handle the dynamic task assignment scenario. We assume there is a cloud accepting task requests and forwarding them to the robots such as, e.g. in online order and delivery services, which is also responsible for maintaining an up-to-date task list and to mark tasks as completed. An instance of the `OptimizationGuidance` class (cf. Section III-B) communicates with the cloud, triggers the distributed optimization algorithm in the separate computation thread whenever a new task request is received and maintains a local task queue. While the queue is not empty, tasks are executed in order with the specific task execution logic (e.g. move to a given point and to perform loading/unloading operations). A specific `OptimizationThread` is responsible for formulating the actual linear programming task assignment problem and to run the distributed simplex algorithm [19] in DISROPT. The `OptimizationThread` class also handle communication among neighboring robots.

Remarkably, from the point of view of the final user, all that is needed is to implement the task execution logic and to integrate the cloud with the task request service of the specific application scenario. The whole dynamic task assignment mechanism is taken care of by ChoiRbot.

B. Distributed Model Predictive Control (MPC)

Let us now describe the second complex scenario implemented in ChoiRbot.

Problem formulation: In this scenario, we assume there are N robots with linear dynamics of the type $x_i(t+1) = A_i x_i(t) + B_i u_i(t)$, where, for all $i \in \{1, \ldots, N\}$, $x_i(t) \in \mathbb{R}^{n_i}$ and $u_i(t) \in \mathbb{R}^{m_i}$ are the i-th robot state/input. We assume the robots must satisfy local state/input constraints $x_i(t) \in \mathcal{X}_i$ and $u_i(t) \in \mathcal{U}_i$. Each robot is associated to an output $z_i(t) = C_i x_i(t) + D_i u_i(t)$, with $z_i \in \mathbb{R}^{r_i}$. The robot outputs are assumed to be coupled with a constraint $\sum_{i=1}^{N} z_i(t) \in \mathcal{S}$. Finally, we assume each robot i is equipped with a local cost function $\ell_i(x_i, u_i)$ that must be minimized. The conceptual idea of MPC is to solve a finite-horizon optimal control problem at each time step t, with trajectories spanning the time horizon $[t, t+T]$. The first control input is applied and the process is repeated. In distributed model predictive control schemes, each robot solves a local version of the overall optimal control problem and leverages communication with neighbors to achieve a feasible solution.

Implementation: We implemented the classical distributed MPC algorithm in [20]. The departing point is a targeted extension of the `OptimizationGuidance` class, which implements the actual steps of the distributed MPC algorithm. The problem data (i.e. system matrices A_i, B_i,

C_i, D_i, local constraints $\mathcal{X}_i, \mathcal{U}_i$, coupling constraints $\mathcal{S}$, prediction horizon T) are provided by the user as class parameters. This class interacts with an `OptimizationThread` instance to formulate and solve the local optimal control problem at each control iteration. Note that, differently from the dynamic task assignment, in this scenario the solution of optimization problems is completely local. Here, communication among neighbors occurs within the `OptimizationGuidance` class, as required by the MPC algorithm (see [20] for details).

VI. Simulations and Experiments

In this section, we provide simulation and experimental results for two different scenarios that can be handled by ChoiRbot. We begin by showing experimental results of the dynamic task assignment scenario described in Section V-A. Then, we show how to the proposed package can be easily interfaced with Gazebo and show simulation results for a team of mobile wheeled robots.

A. Dynamic Task Assignment on Turtlebot3 Mobile Robots

In this experiment, we consider a team of four Turtlebot 3 Burger mobile robots that have to accomplish a set of task scattered in the environment. A task is considered accomplished if the designed robot reaches its position on the $\{x, y\}$ plane. As in real applications, problem data are not completely known a-priori, we consider a dynamic assignment problem where new data arrive during the execution. Thus, robots have to re-optimize and adjust their local planning whenever new information is available. Inspired by the approach in [21], we assume that a new task is revealed as soon as one has been completed. The pose of each robot is retrieved by communicating with a Vicon Motion capture system. In order to interface ChoiRbot with the Vicon system, we developed an ad-hoc ROS 2 package, which is provided in the toolbox repository. As for the control layer to steer robots to the desired positions, we implemented a dedicated controller node executing the control law in [22]. Each robot is assigned a fixed label in $\{1, \ldots, 4\}$. For the sake of presentation, we assume robots communicate according to a fixed Erdős-Rényi graph with edge probability 0.2, depicted in Figure 3. However, the distributed simplex works under time-varying, asynchronous networks and is robust to packet losses [19]. Thus, it can be also implemented with the `BestEffortCommunicator` (cf. Table I).

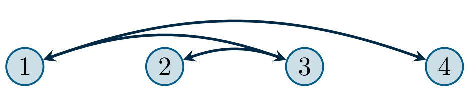

Fig. 3. Communication graph for the dynamic task assignment scenario.

When the cloud communicates to robots the pending tasks, robots start the distributed simplex as described in Section V-A. At each communication round of the distributed algorithm, the generic robot exchanges with its neighbors a matrix representing possible robot-to-task assignments until they converge to a consensual solution. In Figure 4, we report a snapshot from the experiment in which the robots are servicing a set of tasks, while Figure 5 reports a Gantt chart of the actual task execution. A video of the experiment is available as supplementary material to the manuscript.

Fig. 4. Snapshot from the dynamic task assignment experiment. Robots move in order to reach their designed tasks (red markers).

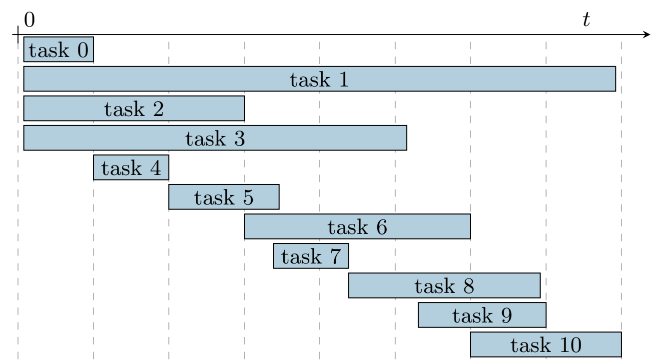

Fig. 5. Gantt chart of the task execution flow in the dynamic task assignment. The horizontal axis represents time with the actual scale. The left side of each rectangle indicates the beginning of a task, while the right indicates that the task has been serviced by a robot. As described above, when a task is serviced (e.g. task 0) a new one begins (e.g. task 4).

B. Formation Control for Unicycle-Like Robots

In this scenario, the goal is to drive robots to a translationally independent formation in the $\{x, y\}$ plane that satisfies a set of given constraints. These constraints are specified by a set of desired inter-robot distances $d_{ij} \geq 0$ for certain couples (i, j) of robots. In order to achieve this formation in a distributed way, robots are assumed to communicate according to a fixed undirected graph. Specifically, all the robot couples (i, j) for which there is a desired distance d_{ij} are assumed to communicate with each other. In this scenario, at each control iteration the i-th robot communicates to neighboring robots a vector in $\mathbb{R}^2$ representing its position in the plane. We refer the reader to, e.g. [23] for a detailed description. Let the robots apply the distributed control law

$$u_i(t) = \sum_{j \in \mathcal{N}_i} (\|x_i(t) - x_j(t)\|^2 - d_{ij}^2)(x_j(t) - x_i(t)), \quad (2)$$

where $\mathcal{N}_i$ denotes the neighbor set of robot i.

We extend the `DistributedControlGuidance` and override the `evaluate_velocity` method (cf. Section III-C) in order to implement the control law in (2). However, the single integrator control input in (2) is not directly implementable on unicycle-like robots. Thus, we developed a specific controller that maps the input provided by (2) to a suitable set of inputs for wheeled robots using the approach described in [9]. The interfacing with Gazebo is straightforward. Robot poses are retrieved directly by the odometry topic maintained by Gazebo, while robot inputs are sent on suitable topics read by Gazebo plugins. In Figure 6 a snapshot from a Gazebo simulation in which six Turtlebot 3 Burger robots have to draw an hexagon. A video of the simulation is available as supplementary material.

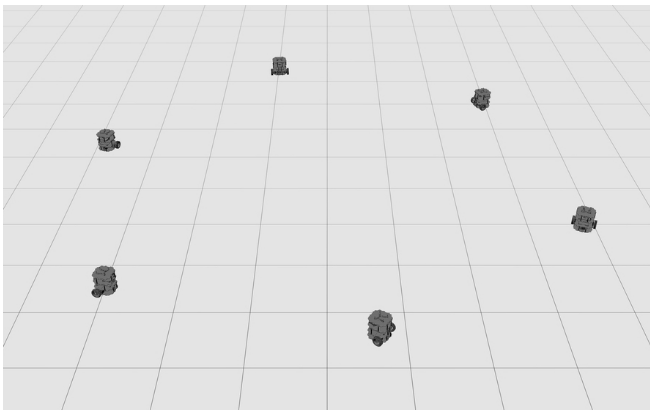

Fig. 6. Formation control simulation via Gazebo. By leveraging on ChoiRbot functionalities, robots reach the desired hexagonal formation.

VII. Conclusions

In this paper, we presented ChoiRbot, a novel toolbox for distributed cooperative robotics written in Python and based on the ROS 2 platform. The toolbox is designed with a three-layer structure and provides a comprehensive set of functionalities for communication and distributed optimization. Thanks to this toolbox, complex optimization-based cooperative robotic scenarios can be implemented in a straightforward manner, thus allowing programmers to focus on the most important part of the production code. Several scenarios have been described and simulations and experimental results have been presented. Future directions include the implementation of additional features for the RoboPlanning/RoboControl layers in order to support additional types of robots and distributed algorithms.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[2] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *Proceedings of the 13th International Conference on Embedded Software*, 2016, pp. 1–10.

[3] G. Notarstefano, I. Notarnicola, and A. Camisa, "Distributed optimization for smart cyber-physical networks," *Foundations and Trends® in Systems and Control*, vol. 7, no. 3, pp. 253–383, 2019. [Online]. Available: http://dx.doi.org/10.1561/2600000020

[4] E. Aertbeliën and J. De Schutter, "eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 1540–1546.

[5] C. Paxton, A. Hundt, F. Jonathan, K. Guerin, and G. D. Hager, "CoSTAR: Instructing collaborative robots with behavior trees and vision," in *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2017, pp. 564–571.

[6] 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 *2013 European Conference on Mobile Robots*. IEEE, 2013, pp. 19–25.

[7] J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, and O. Von Stryk, "Comprehensive simulation of quadrotor uavs using ROS and gazebo," in *International conference on simulation, modeling, and programming for autonomous robots*. Springer, 2012, pp. 400–411.

[8] G. A. Casan, E. Cervera, A. A. Moughlbay, J. Alemany, and P. Martinet, "ROS-based online robot programming for remote education and training," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 6101–6106.

[9] S. Wilson, P. Glotfelter, L. Wang, S. Mayya, G. Notomista, M. Mote, and M. Egerstedt, "The robotarium: Globally impactful opportunities, challenges, and lessons learned in remote-access, distributed control of multirobot systems," *IEEE Control Systems Magazine*, vol. 40, no. 1, pp. 26–44, 2020.

[10] F. Albers, C. Rösmann, F. Hoffmann, and T. Bertram, "Online trajectory optimization and navigation in dynamic environments in ros," in *Robot Operating System (ROS)*. Springer, 2019, pp. 241–274.

[11] A. S. Kumar, G. Manikutty, R. R. Bhavani, and M. S. Couceiro, "Search and rescue operations using robotic darwinian particle swarm optimization," in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 2017, pp. 1839–1843.

[12] W. P. N. dos Reis and G. S. Bastos, "Implementing and simulating an alliance-based multi-robot task allocation architecture using ros," in *Robotics*. Springer, 2016, pp. 210–227.

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

[14] E. Erős, M. Dahl, A. Hanna, A. Albo, P. Falkman, and K. Bengtsson, "Integrated virtual commissioning of a ROS2-based collaborative and intelligent automation system," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2019, pp. 407–413.

[15] M. Reke, D. Peter, J. Schulte-Tigges, S. Schiffer, A. Ferrein, T. Walter, and D. Matheis, "A self-driving car architecture in ROS2," in *2020 International SAUPEC/RobMech/PRASA Conference*. IEEE, 2020, pp. 1–6.

[16] F. Farina, A. Camisa, A. Testa, I. Notarnicola, and G. Notarstefano, "Disropt: a python framework for distributed optimization," *arXiv preprint arXiv:1911.02410*, 2019.

[17] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, 2004, pp. 2149–2154.

[18] G. Notarstefano, M. Egerstedt, and M. Haque, "Containment in leader–follower networks with switching communication topologies," *Automatica*, vol. 47, no. 5, pp. 1035–1040, 2011.

[19] M. Bürger, G. Notarstefano, F. Bullo, and F. Allgöwer, "A distributed simplex algorithm for degenerate linear programs and multi-agent assignments," *Automatica*, vol. 48, no. 9, pp. 2298–2304, 2012.

[20] A. Richards and J. P. How, "Robust distributed model predictive control," *International Journal of control*, vol. 80, no. 9, pp. 1517–1531, 2007.

[21] S. Chopra, G. Notarstefano, M. Rice, and M. Egerstedt, "A distributed version of the hungarian method for multirobot assignment," *IEEE Transactions on Robotics*, vol. 33, no. 4, pp. 932–947, 2017.

[22] J. J. Park and B. Kuipers, "A smooth control law for graceful motion of differential wheeled mobile robots in 2d environment," in *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 4896–4902.

[23] M. Mesbahi and M. Egerstedt, *Graph theoretic methods in multiagent networks*. Princeton University Press, 2010, vol. 33.