Configuration and Control of KMR iiwa Mobile Robots using ROS2

Charlotte Heggem*, Nina Marie Wahl* and Lars Tingelstad
Department of Mechanical and Industrial Engineering
NTNU, Norwegian University of Science and Technology
Trondheim, Norway

***Abstract*—In this paper, we present a novel system for controlling a KMR iiwa mobile robot using ROS2. The KMR iiwa is a mobile robot with a manipulator mounted on the base that is developed by the robot manufacturer KUKA. The developed system integrates with the Sunrise.OS operating system of the mobile robot and exposes sensor and control interfaces over UDP and TCP sockets. The controllability of the mobile robot from ROS2 is verified using the Cartographer and Navigation2 projects.**

***Index Terms*—KMR iiwa, ROS2, KUKA, Mobile Robot, Industry 4.0**

I. Introduction

Industry 4.0 is about digitalization, automation, machine learning, and real-time data. A company that works extensively with industry 4.0 is the German company KUKA, Keller und Knappich Augsburg. The company produces robot systems for the industry and automated production solutions, and focus on networked, intelligent production. Such a production robot is the KMR iiwa, KUKA Mobile Robot Intelligent Industrial Work Assistant, shown in Figure 1.

The KMR iiwa is composed of a robot arm, the LBR iiwa 14 R820, and a mobile platform, the KMP 200 omniMove [1]. LBR is a German abbreviation for lightweight robot, while KMP is short for KUKA Mobile Platform. The intended use of the mobile robot is to handle automated manufacturing tasks and transport components, and it is characterized by its high degree of mobility and flexibility.

The operating system for the KMR iiwa mobile robots is KUKA Sunrise.OS. Programming the software requires knowledge of the Java programming language and the Sunrise software, including robot specific functions. This could be a blockade for many developers. Another drawback is that data and information from the robot are only available locally in the Sunrise system during executions of applications. In a decade when everything should be online, compatibility between networked devices is desired, and information should be available. Tools that support system integration based on common platforms are essential for the continuous evolution of Industry 4.0.

This work was supported by the Norwegian Research Council project MANULAB: Norwegian Manufacturing Research Laboratory under grant 269898.

*Author names in alphabetical order. Charlotte Heggem and Nina M. Wahl contributed equally to this work.

Fig. 1. The KMR iiwa is composed of a mobile platform and a lightweight manipulator [2]. Image courtesy of KUKA Nordic AB.

ROS2, Robot Operating System 2, is the second generation of the open source framework ROS for developing robot applications with support for several programming languages and platforms [3]. Integration between Sunrise.OS and ROS2 is desirable as it would limit the required preknowledge required to operate the robot and make it more available for users. The ROS2 packages Cartographer and Navigation2 perform real-time localization, mapping and navigation of mobile vehicles and are highly relevant to use with the KMR iiwa.

Two projects with focus on interaction between the LBR iiwa and ROS have been inspirational for the work presented in this paper. The LBR iiwa and the KMP is operated by the same controller, and hence a similar approach for programming the robot can be applied to the KMP.

Virga and Esposito [4], propose an architecture with native ROSJava nodes launched on the robot controller. Hence, the two operating systems can exchange data and commands in the form of ROS messages over the ROS framework. The proposed solution requires installation of third-party libraries on the robot controller to run ROS nodes.

The work by Mokaram et al. [5] provides a simple standalone application that requires minimal installation on the robot controller. The architecture of the API consists of two main components, a single Sunrise.OS application on the robot controller and a ROS node launched on the external computer. The ROS node establishes a connection to the controller over TCP, Transmission Control Protocol. Data and command

messages are transmitted between the two components as strings.

The two APIs of [4] and [5] are developed for different purposes, but the implementations contain the same fundamental functionality. The main difference between the two architectures is whether to publish ROS messages directly from the robot controller or to utilize a middle-layer to handle the communication. The API of [5] includes simple methods for directly controlling the robot, while [4] present a more advanced system, including functionality for simulation in Gazebo and using the robot with ROS packages as Moveit!. In this paper, we present a communication interface between the KMR iiwa and ROS2 to control the mobile platform. The developed system integrates with the Sunrise.OS operating system of the mobile robot and exposes sensor and control interfaces over UDP, User Datagram Protocol, and TCP sockets. The interface is available online [6], and is the first free and open approach to controlling a KMR iiwa using ROS2.

The main functionality is autonomous navigation in an unknown, dynamic environment without collision. The goal is that a user should easily be able to connect to the robot and control it without any preknowledge about KUKA robotic systems.

The paper is structured as follows. A description of the KMR iiwa is presented in Section II. Further, in Section III, an introduction to ROS2 and the packages Cartographer and Navigation2 is given. A challenge with the implementation was to find the correct methods to command and retrieve data from the robot. The approach for obtaining the correct methods is described in Section IV. Next, in Section V, follows a system description with the implemented functionality of the interface. Section VI presents a verification of the functionality of developed system. Finally, Section VII concludes the paper.

II. KMR IIWA

A. Hardware

The KMP mobile platform has four mecanum wheels, which allow the mobile platform to move omnidirectionally. It is equipped with two SICK S300 safety laser scanners mounted diagonally opposite of each other.

The S300 is a compact laser measurement system that scans the environment in two dimensions in the height of 150 mm above the ground in means of infrared laser beams. Each laser has a scanning range of 270°, covering one long side and one short side of the vehicle, and a resolution of 0.5°.

The controller of the robot system, Sunrise Cabinet, is contained inside the KMP. The Sunrise Cabinet contains two PCs: one control PC and one navigation PC.

B. Software

KUKA Sunrise.OS is the system software for robots that are operated by the Sunrise Cabinet. It offers functionality for programming and configuration of robot applications. Commands, sensor data, and information related to the ongoing operation are only available locally on the robot control system, or the supplied teach pendant, the SmartPAD.

C. Operation

There are three different approaches for controlling the KMP: manually, autonomously, or by an application.

The most interesting option in this context is to control the KMP by a Java-based Sunrise application on the Sunrise Cabinet. The application can be implemented in the programming environment Sunrise.Workbench. The software packages KUKA RoboticsAPI and KUKA Sunrise.Mobility contain methods for obtaining information from the robot system and executing motions, which are the basis for developing a program for controlling the KMP.

KUKA.NavigationSolution is an optional software package with functionality for autonomous navigation of mobile platforms. The navigation software is based on sensor data from the S300 laser scanners and the odometry of the mobile platform.

The KMP can be moved manually by jogging the robot from the SmartPAD or the Radio Control Unit, an optional device with joysticks for controlling the platform.

The robot system has three different operation modes. T1 and T2 are manual operation modes used for testing and verification of programs. AUT is autonomous mode, which is the operating mode for program execution.

D. Safety

The primary function of the S300 sensors is to operate as the safety equipment of the system by monitoring predefined areas around the vehicle. By default, the S300 sensors are configured with a protective field and a warning field. The size of the monitored fields depends on the velocity of the vehicle. The consequence of a violation of the two fields varies with the operation mode. Generally, a breach of a field causes a reduction of maximum travel speed or triggers a safety stop of the vehicle. The laser scanners are not active for velocities below $0.13\ \mathrm{m/s}$ in the manual operation modes.

III. ROS2

ROS [3] is a robot operating system that can be used with multiple programming languages and has implemented open source functionality. It includes tools and libraries to handle the programming of robots without having to deal with hardware. The main goal of ROS is to provide a standard that can be used by any robot. ROS2, the second generation of ROS, is state-of-the-art software and is currently under massive deployment.

In general, ROS2 is cleaner and faster than the prior version, in addition to more flexible and universal. One of the main differences between the two versions is that ROS2 is built on top of DDS, Data Distribution Service, which provides a distributed discovery feature.

Two ROS2 packages that are relevant for controlling the KMP is Cartographer [7] and Navigation2 [8].

Cartographer is a package for real-time SLAM, simultaneous localization and mapping, and is part of Google's open source projects [9].

A map can be created based on the robot's odometry, transformation information, and sensor information when the robot moves. The map can further be provided to Navigation2 and be used for navigation in the environment. The requirement for the cartographer node is sensor data measuring the distance to obstacles in the environment. Data from IMU sensors and odometry sensors can be included to improve the result.

Navigation2 is a package that can be used to control mobile robots and is based on a velocity controller. The main goal of applying the package is to navigate the robot from a start pose to a goal pose. This task can be broken down into subtasks like handling maps, localization of the robot, obstacle avoidance, and path following. When an obstacle-free path is calculated, velocity commands are produced to describe how the robot should move to follow the path. The navigation package requires information about the environment and how the robot moves, which can be provided in the form of a map, sensor data, and odometry data.

IV. Approach

The information required by Cartographer and Navigation2 defines the functional requirements of the system: It must be possible to retrieve odometry from the wheel encoders and laser data from the SICK scanners on the KMP, and the vehicle must be able to be controlled by velocity messages.

The motion commands and sensor retrieval methods from KUKA RoboticsAPI are limited and are chosen based on KUKA's definition of what is the intended use for programming the robot. A challenge that was faced during the development was to find the appropriate methods to retrieve data and move the KMP for the desired outcome.

As mentioned in Section II-D, the main functionality of the laser scanners is to monitor predefined areas around the vehicle. Boolean signals from the lasers indicate whether the monitored fields are violated, and can be extracted through defined methods from the KUKA RoboticsAPI. Range data from the laser scanners are only available through KUKA.NavigationSolution, and there are no direct methods to retrieve sensor data through KUKA RoboticsAPI.

The correct method for retrieving the sensor data was found by investigating the underlying functionality of KUKA.NavigationSolution. This software package introduces several views that can be opened in Sunrise Workbench, among them are the LaserView that visualizes range readings in real-time. By exploring the source code of the view, it was found that a FDI, Fast Data Interface, connection is established to the Navigation PC. The FDI connection utilizes a UDP socket to transmit data and enables functionality for subscribing to sensor data. As KUKA.NavigationSolution relies on the odometry data from the KMP for navigation, this data can also be subscribed to through the FDI connection.

As for the motion commands, the predefined methods available from KUKA RoboticsAPI provide functionality for moving the KMP to an absolute or relative pose. This is not applicable for the intended use, as Navigation2 sends velocity commands.

The approach leading to the proposed solution was to investigate the underlying code of the devices used to jog the KMP manually. When a jogging button is held on the smartPAD or the joysticks are used to move the robot with the Radio Control Unit, the robot is continuously jogged. In a similar manner, a jog method can be executed continuously from a Java program to make the robot move with the specified velocity. The method found to jog the robot enabled access to the internal functionality of the robot, which is mainly marked as private and not intended to use for programming by external users.

By investigation of the source code, it was found that for each time the jog method is executed, a new thread is established. As the method is intended for internal use, the threads are marked as private, and there is no way to handle all the threads established when continuously calling the method. Over time this lead to an accumulation of threads, which is a problem when the number of threads get too high. A solution to this problem was found in the code of the Radio Control Unit, which revealed that a support class had to be implemented to handle the threads. The support class creates a thread pool for each new jogging execution, which can be shut down when the velocity motion is finished, and hence, kill all the threads established.

With the jogging motion it is possible to execute motion based on velocity commands, which further makes it possible to move the KMP by the commands from Navigation2.

V. System Description

The implemented interface has the following main functionalities:

- Retrieve laser data from the KMP
- Retrieve odometry data from the KMP
- Retrieve status information from the KMP
- Move the KMP by giving velocity commands in the terminal
- Move the KMP by setting a goal pose in the terminal
- Use Cartographer to create a map of the environment
- Use Navigation2 to move the KMP

The interface consists of multiple ROS2 nodes running on a remote PC communicating with a Java program over TCP. The remote PC does not need to be in the immediate vicinity of the KMR, but must be connected to the same network. The Java program, *KMRiiwaSunriseApplication*, is installed on the Control PC in the Sunrise Cabinet, which handles further communication with other internal devices on the robot. The physical architecture is shown in Figure 2.

Figure 3 shows the architecture of the implemented communication interface between the Java application and the ROS2 nodes. Each node is responsible for one area and is communicating with a corresponding Java class on the Sunrise Cabinet over UDP or TCP. Both protocols are implemented, and the desired communication type can be set in the system parameters. By creating separate nodes and connections for all tasks, it is possible to launch only the nodes needed for a

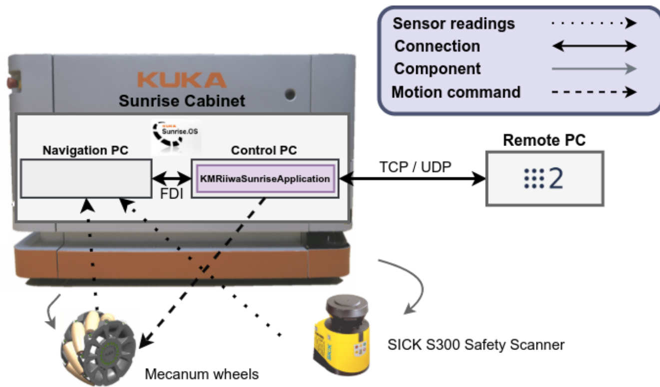

Fig. 2. Graphical representation of the physical system. Component images courtesy of KUKA Global.

specific use case. This limits the transferred data and enables a faster system.

The information is transmitted with the selected protocol as strings on the following format:

$$\underbrace{Length >}_{\text{Additional Information}} \quad \underbrace{Type \quad (LaserID) \quad Timestamp \quad Data}_{\text{Message}}$$

The additional information, *Length*, is only included if the message is sent over TCP. As TCP is a buffer protocol, this is necessary to ensure that the whole message is read.

The *Type* field describes what kind of data this message includes, and is necessary to know how the data should be processed. If the message is coming from the robot, an example of a type could be *Odometry*. An odometry message contains data describing the pose and velocity of the robot.

The pose of the robot is represented by a position vector $x \in \mathbb{R}^3$ and a unit quaternion $q \in S^3$, while the velocity is represented by a twist $\mathcal{V} = (\omega, v) \in \mathbb{R}^6$ where $\omega \in \mathbb{R}^3$ and $v \in \mathbb{R}^3$ are the angular and linear velocities, respectively.

The *LaserID* only applies for laser scan messages to denote from which sensor the data is read. The data, in this case, consist of 540 range readings from the specified laser.

Similarly, a message of type *setTwist* can be sent, commanding a change in the velocity of the robot.

A. Sunrise Application

The *KMRiiwaSunriseApplication* is the main application running on the Sunrise Cabinet. The Java classes communicating with ROS2 are initiated from the application as threads and executed in parallel. The Java classes are responsible for sending sensor data and other relevant information, and for carrying out the pose or velocity commands received from the ROS2 nodes. Support classes are implemented for both the protocol options TCP and UDP for handling the socket objects and transmitting data. Each of the Java communication classes establishes a socket class to transmit the data to the corresponding ROS2 node.

The *kmp_sensor_reader* class is handling both the data from the S300 laser sensors and the odometry data. An FDI connection, which is used to transmit odometry and laser data, is established between an instance of the class and the Navigation PC. The FDI connection is based on subscriptions, meaning a data type is only sent if a subscription to the data type is created. This makes it possible to subscribe to only laser or odometry data. A data listener class is implemented that, by subscribing to the data of interest, retrieves odometry data and laser data through the FDI connection. Further, the data is transmitted to the two corresponding ROS2 nodes.

The *kmp_status_reader* use functionality from KUKA RoboticsAPI to retrieve information from the robot. Examples of information that can be retrieved are whether an emergency stop has been triggered and if there are any obstacles in the monitored fields.

The *kmp_commander* receives velocity or pose commands from the corresponding ROS2 node. Pose commands are executed by the motion type MobilePlatformRelativeMotion from KUKA RoboticsAPI, while velocity commands are being carried out by jogging the robot. More specific this is done by a KMPJogger object, which is implemented to handle the execution of motion and the accumulation of threads.

B. Remote PC

On the remote PC, there are four different nodes available for communication with the KMP: *kmp_odometry*, *kmp_laser*, *kmp_command* and *kmp_status*.

All the nodes have data process methods and ROS publishers and subscribers to correctly handle the data to and from the KMP. All the publishers and subscribers, and the associated ROS topic are listed in Table I and II.

The *kmp_odometry* and *kmp_laser* nodes handle the sensor information retrieved from the KMP, while the *kmp_command* subscribes to multiple ROS topics and forwards the commands from each topic. The currently supported commands are: move with a certain velocity, move to a given pose, and shutdown. The shutdown command is essential to be able to shut down all connections and threads in the program correctly.

The *kmp_status* retrieves information data from the KMP and saves the information to a ROS message, *KmpStatusdata*. This message type is custom made for the interface and includes information that is useful when operating the robot. The message could be extended with more information if necessary. The *KmpStatusdata* includes the information shown in Table III.

VI. System Verification using Cartographer and Navigation2

The ROS packages Cartographer and Navigation2 are used to verify if the communication and control work as expected. Both packages include parameter files with multiple parameters that can be tuned to improve the algorithms used. These parameters are minimally tuned for this experiment, and only the parameters necessary for the packages to work with our data have been changed. All available data sources are used, which includes sensor data from both S300 sensors as well as odometry data.

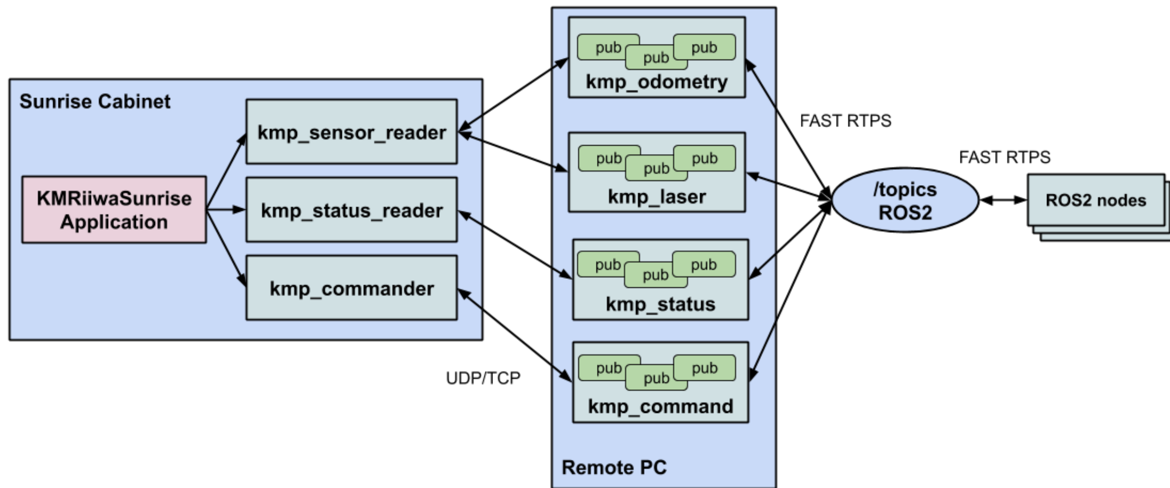

Fig. 3. Architecture of the implemented solution.

TABLE I
PUBLISHERS FOR PUBLISHING DATA FROM KMP TO ROS

Name	Message type	Topic	Description
pub_odometry	Odometry	/odom	Odometry information.
pub_laserscan1	LaserScan	/scan_1	Data from B1 S300 laser (front).
pub_laserscan2	LaserScan	/scan_2	Data from B4 S300 laser (back).
pub_kmp_statusdata	KmpStatusdata	/kmp_statusdata	Statusdata retrieved in the *kmp_status* node.

TABLE II
SUBSCRIBERS FOR SUBSCRIBING TO DATA FROM ROS TO KMP

Name	Message type	Topic	Description
sub_twist	Twist	/cmd_vel	Make KMR move at a certain velocity.
sub_pose	Pose	/pose	Make KMR move to a certain pose.
sub_shutdown	String	/shutdown	Make the application on the Sunrise controller shut-down. Any string sent to this topic do the same purpose.

TABLE III
FIELDS INCLUDED IN A *KmpStatusdata* MESSAGE

Name	Message type	Description
header	std_msgs/Header	Regular header for all ROS messages.
operation_mode	String	The KMR iiwa has three different operation modes. This field states the current mode.
ready_to_move	Boolean	True if the robot is ready to move, and no safety rules is violated.
warning_field_clear	Boolean	False if either of the warning fields of the S300 sensors are violated.
protection_field_clear	Boolean	False if either of the protection fields of the S300 sensors are violated.
is_kmp_moving	Boolean	True if the KMP is moving.
kmp_safetystop	Boolean	True if the KMP performs a safety stop. This happens if any of the internal safety monitoring functions of Sunrise software are violated.

By driving the robot around in the laboratory and using Cartographer, the map in Figure 4 was created. Tuning the parameters to a more significant extent would likely improve the result, but was not relevant for testing the communication interface. As mentioned in section II, the SICK scanners perform a planar scan 150 mm above the ground. This affects the map, as overhanging obstacles are not detected. Hence, additional sensors should be included in the system to be able to perform autonomous navigation safely.

The compatibility with Navigation2 was verified by entering a goal pose in Rviz to which the KMP was to navigate.

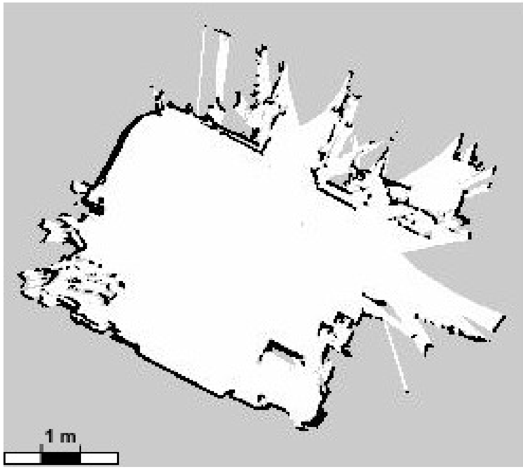

Fig. 4. Map created by Cartographer

The goal pose is sent to Navigation2 which returns velocity commands. The use of Navigation2 works as expected for more straightforward scenarios. When commanded to navigate to poses in open areas, the robot moves with holonomic movement and follows the planned path until the requested goal pose. When the goal pose is set too close to obstacles, the navigation is not completed.

The maximum velocity specified in the parameter file of Navigation2 are above $0.13\ \mathrm{m/s}$, which is the velocity where the sensors are activated for T1 mode. If the sensors detect an object inside the protective field when driving at a speed higher than this, the safety restriction of the robot turns in and stops the movement. This causes the navigation to fail, as the vehicle is not following the given commands and not showing enough progress within a time limit. The vehicle is not able to drive out of this area as all the commands from Navigation2 are at a higher speed than allowed by the robot.

When the KUKA Navigation Solution controls the vehicle, the velocity is automatically reduced when objects are inside the protective field. Navigation2 does not implement this behavior, and this causes a conflict between the built-in safety restrictions and the navigation controller. To make Navigation2 work optimally, this must be taken into account. Newly implemented behavior in Navigation2 makes it possible to update the parameters which specify the maximum velocities dynamically.

Our suggested solution to the navigation problem is to monitor the status of the warning fields and protective fields, which both are included in the *KmpStatusdata* message, and use this information to control the velocity. For both operation modes, the robot stops if an obstacle is detected in the protective field. The velocity should be reduced when an obstacle is detected in the warning field to reduce the size of the protective field. If necessary, a second warning field can be configured by the use of SICK software, to be notified about obstacles earlier. This could solve the problem, but would lead to a less generic system.

VII. Conclusion

This paper describes a control interface for operating the mobile robot KMR iiwa with ROS2. The proposed architecture is verified by a proof-of-concept implementation that enables control of the robot from an external computer. The architecture is based on multiple ROS2 nodes with corresponding Java classes that handle separate tasks. The architecture is created in a scaleable manner, where more nodes can be added for additional functionality.

It was desired to navigate the robot by utilizing the ROS2 packages Cartographer and Navigation2. For this purpose, functionality was implemented to retrieve odometry and laser data from the sensors, and for the ability to control the model by velocity commands. The specified ROS2 packages were used to verify the controllability of the mobile vehicle. The Cartographer package was able to create a fully recognizable map of the environment, and simple navigation in the environment was performed. The verification revealed issues related to navigation closer to obstacles, and a possible solution for this was proposed.

Further work include manipulation of the LBR iiwa, improved navigation, and the addition of camera sensors to handle issues related to 2D laser scans.

The work is open source and available online at https://github.com/ninamwa/kmriiwa_ws.

References

[1] KUKA. *KMR iiwa omniMove*, 2019. https://xpert.kuka.com/app/portal.

[2] Direct Industry. KMR IIWA. https://www.directindustry.com/prod/kuka-ag/product-17587-1714901.html, 2016. The image is used with permission from KUKA Nordic AB. Accessed: 2020-03-13.

[3] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[4] Salvatore Virga and Marco Esposito. IFL-CAMP/Iiwa Stack. https://github.com/IFL-CAMP/iiwa_stack, 2019. Accessed: 2019-11-07.

[5] Saeid Mokaram, Jonathan M. Aitken, Uriel Martinez-Hernandez, Iveta Eimontaite, David Cameron, Joe Rolph, Ian Gwilt, Owen McAree, and James Law. A ROS-integrated API for the KUKA LBR iiwa collaborative robot. *IFAC-PapersOnLine*, 50(1):15859 – 15864, 2017. 20th IFAC World Congress.

[6] Charlotte Heggem and Nina Marie Wahl. Project repository: kuka_ws. https://github.com/ninamwa/kmriiwa_ws, 2019.

[7] W. Hess, D. Kohler, H. Rapp, and D. Andor. Real-time loop closure in 2D LIDAR SLAM. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1271–1278, 2016.

[8] ROS2 Navigation. https://ros-planning.github.io/navigation2/. Accessed: 2020-03-27.

[9] Cartographer. https://opensource.google/projects/cartographer. Accessed: 2020-03-27.