Towards a ROS2-based software architecture for service robots

Yong Hwan Jo, Se Yeon Cho, Byoung Wook Choi
Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul, South Korea

Article Info

Article history:

Received Dec 23, 2022
Revised Feb 3, 2023
Accepted Feb 23, 2023

Keywords:

Embedded system
Python
Robot operating system 2
Service robot
Software architecture

ABSTRACT

This paper presents a scalable software architecture based on robot operating system 2 (ROS2) for service robots. ROS2 supports the data distribution service (DDS) protocol that provides benefits such as real-time operation and security and performance enhancements. However, ROS2 still lacks task management capabilities, essential for practical robotic applications consisting of multiple threads and processes. Moreover, integrating new devices into ROS2 requires additional development effort to create specific drivers for specific devices. The proposed software architecture addresses these drawbacks and provides a simple and user-friendly programming interface for easier integrating of various devices and existing ROS2 applications. Moreover, it is designed using python with multi-processing to avoid issues related to the python global interrupt lock (GIL). To verify the developed software architecture, an application for a custom-made service robot called the SeoulTech service robot (SSR) is implemented on a Jetson Xavier NX board with various features, such as ROS2 navigation and SLAM, text-to-speech (TTS), speech recognition, and face recognition.

Corresponding Author:

Byoung Wook Choi
Department of Electrical and Information Engineering
Seoul National University of Science and Technology
Seoul 01811, South Korea
Email: bwchoi@seoultech.ac.kr

1. INTRODUCTION

Recently, robots made by integrating complex and diverse mechanical devices and sensors have been used in many applications. Robots are finding active use in various fields, from the existing manufacturing industry to the service sector [1]. In these areas, use cases are increasing in applications that involve working with humans or sharing tasks closely with them, such as serving and delivery [2], [3].

Service robots must work with humans or interact with customers to provide services [4], [5]. The development cost and duration of a service robot have increased considerably because it is difficult to apply the existing service robot architecture to these types due to the direction of design and development efforts depending on the purpose of the service. To this end, the demand for robot software frameworks such as robot operating system (ROS) [6] and OROCOS [7] has been constantly increasing. In particular, ROS1 has become the new norm for robotic developers, and existing robotic platforms offer integration to ROS1. However, ROS1 does not support real-time systems and communication protocols, which are the requirements for a reliable and deterministic robotic system. Thus, ROS2 [8] has been developed to solve these issues. ROS2 uses data distribution service (DDS) [9], [10] with the quality of service (QoS) concept to provide improve the real-time performance of inter-node communications and its security. However, robotic applications must perform multiple tasks that often involve multi-threading and multi-processing applications. ROS2 still lacks task management, which is essential for practical applications. Moreover, the integration of new devices into ROS2 requires additional development effort to create specific drivers for specific devices.

To address this issue, this paper proposes a scalable, flexible software architecture based on ROS2 for service robots. The proposed software architecture is designed to be applicable to various robots. It uses a multiprocessing architecture and python and is designed to facilitate hardware deployment, software service expansion, and device integration. It is also designed to enable easier integration by inheriting the class when implementing the services of other functions through its use of multi-process classes.

The SeoulTech service robot (SSR), a service robot with various sensors and actuators, was developed to apply the proposed software and verify the flexibility and scalability of the proposed software architecture. In addition, two Jetson Xavier NX boards were used as the control board and application board on which to deploy the applications and user interfaces so as to expand the artificial intelligence services offered by the robot. In addition, commercialized Kobuki2 [11] and StellarB2 [12] devices were utilized to verify the scalability of the application services. The application was developed to emulate a delivery scenario, on which the robot is expected to avoid three-dimensional obstacles while implementing various artificial intelligence features, such as text-to-speech (TTS) [13].

The contribution of the paper is the proposal of a scalable and flexible software architecture for service robots based on ROS2. The architecture is designed to ease hardware deployment, software service expansion, and device integration, making it applicable to a variety of robots. The authors also demonstrate the feasibility and scalability of the proposed architecture through the development of the SSR and its applications in a delivery scenario. In the future, the proposed architecture will implement floor obstacle avoidance and various artificial intelligence applications [14]–[16] as part of the continuing research on service robots applications that support more diverse scenarios. In addition, we will extend the software architecture to real-time systems by applying various forms of RTOS, such as RT-preempt [17] and Xenomai [18], [19], which are real-time systems, and by applying RT-AIDE [20] to interlock with non-real-time tasks.

2. SOFTWARE DESIGN

This section describes the design of the proposed software architecture. The design of a software architecture is the most crucial phase in the software development process. Therefore, the following considerations are taken into account when designing software architectures: i) scalable to and accessible by other software; ii) each device runs in its own context; iii) synchronization between processes/threads; iv) easy integration of multiple devices; and v) monitor process/thread activity.

2.1. Scalable to and accessible by other software

Service robots use various devices, such as cameras, microphones, and sensors, depending on the service purpose of the robot. For simple integration of these devices, each is modularized so that the software architecture can be designed in a platform format. In such cases, when developing a robot intended to provide a new service, it is easy to apply an existing robot software platform, and doing so has the advantage of reducing the development cost and duration.

Here, we design the software architecture using python to simplify the issues of scaling and access to other software. Development in python is fast and it offers concise and easy grammar with high scalability and portability [21], [22]. The software architecture in this case is difficult to implement in a language such as C/C++, which is more commonly used, as complex operations and algorithms must be applied to implement the artificial intelligence required for service robots. In general, python make development easy due to the existence of numerous artificial intelligence libraries, including PyTorch [23] and TensorFlow [24].

2.2. Each device runs in its own context

The features of the software are designed to run modules for each device in their own context. In general types of software architecture, multiple tasks are designed to be performed via multi-threading. However, python uses what is termed the global interrupt lock (GIL) [25] concept to avoid a deadlock between multiple threads. With GIL, only one thread runs per process. For this reason, in python, a single thread is capable of higher performance than a multi-thread setup, especially in CPU-bound threads. The software architecture developed to solve these problems is designed using the multi-processing technique.

2.3. Synchronization between processes/threads

Figure 1 shows the designed software architecture. The software architecture is designed with two main components: ControlCore and MainWindow. ControlCore is the main controller that deals with devices, interfaces, and the control software. MainWindow is a graphic user interface (GUI) that deals with signals and slots in the user interface. All objects and methods related to the robot control system are separated from the GUI. Data exchange capabilities between MainWindow and ControlCore are implemented using a callback function. The architecture is also applicable to a range of service robots, such as StellaB2, Kobuki2, and SSR. Multiple threads do not require special techniques for communication because each thread shares data with the others. However, for multiprocessing, a communication IPC between processors is required for data exchanges among the processes.

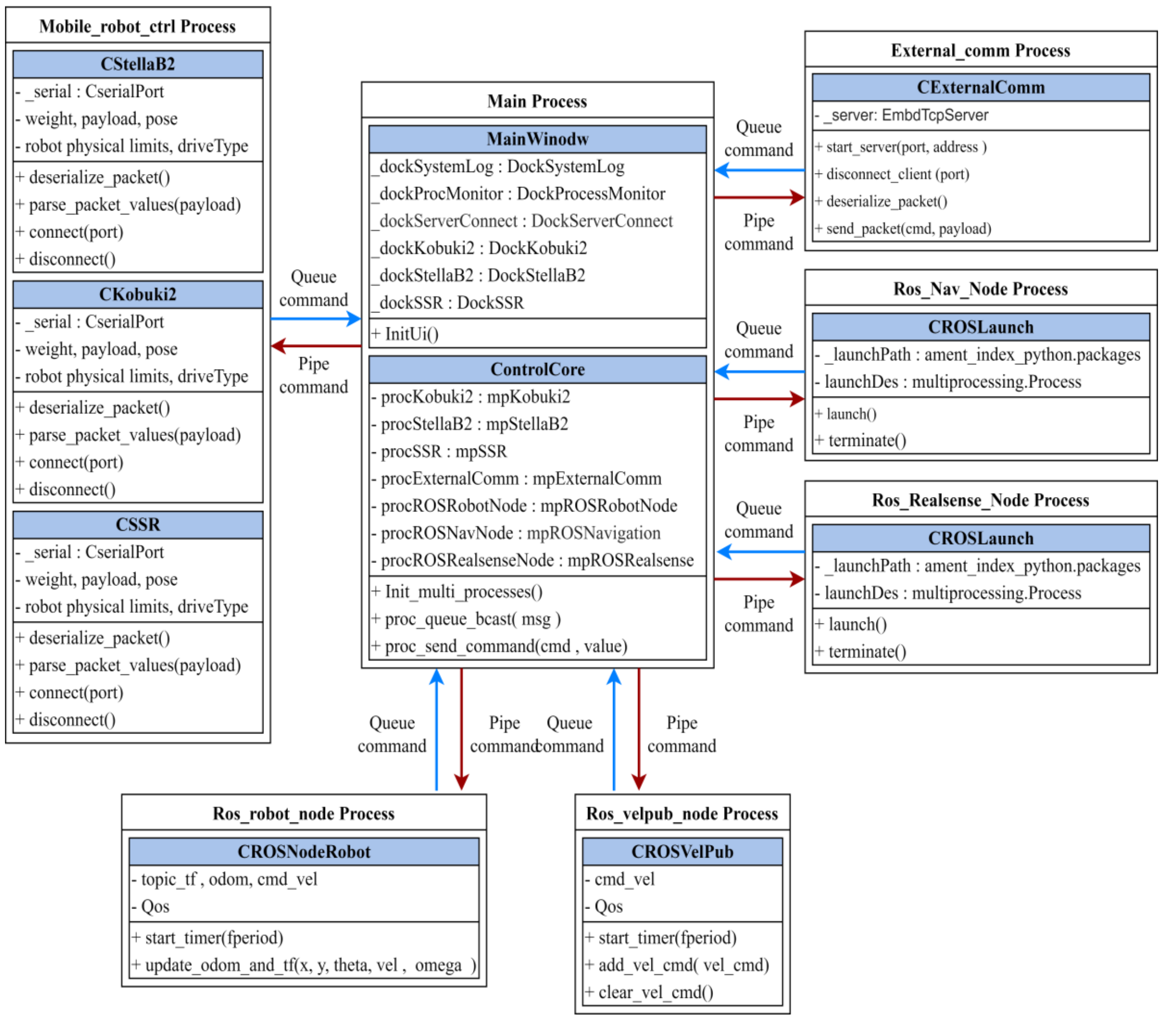

Mobile_robot_ctrl Process
CStellaB2
- _serial : CserialPort
- weight, payload, pose
- robot physical limits, driveType
+ deserialize_packet()
+ parse_packet_values(payload)
+ connect(port)
+ disconnect()
CKobuki2
- _serial : CserialPort
- weight, payload, pose
- robot physical limits, driveType
+ deserialize_packet()
+ parse_packet_values(payload)
+ connect(port)
+ disconnect()
CSSR
- _serial : CserialPort
- weight, payload, pose
- robot physical limits, driveType
+ deserialize_packet()
+ parse_packet_values(payload)
+ connect(port)
+ disconnect()
Main Process
MainWinodw
_dockSystemLog : DockSystemLog
_dockProcMonitor : DockProcessMonitor
_dockServerConnect : DockServerConnect
_dockKobuki2 : DockKobuki2
_dockStellaB2 : DockStellaB2
_dockSSR : DockSSR
+ InitUi()
ControlCore
- procKobuki2 : mpKobuki2
- procStellaB2 : mpStellaB2
- procSSR : mpSSR
- procExternalComm : mpExternalComm
- procROSRobotNode : mpROSRobotNode
- procROSNavNode : mpROSNavigation
- procROSRealsenseNode : mpROSRealsense
+ Init_multi_processes()
+ proc_queue_bcast(msg)
+ proc_send_command(cmd , value)
External_comm Process
CExternalComm
- _server: EmbdTcpServer
+ start_server(port, address)
+ disconnect_client (port)
+ deserialize_packet()
+ send_packet(cmd, payload)
Ros_Nav_Node Process
CROSLaunch
- _launchPath : ament_index_python.packages
- launchDes : multiprocessing.Process
+ launch()
+ terminate()
Ros_Realsense_Node Process
CROSLaunch
- _launchPath : ament_index_python.packages
- launchDes : multiprocessing.Process
+ launch()
+ terminate()
Ros_robot_node Process
CROSNodeRobot
- topic_tf , odom, cmd_vel
- Qos
+ start_timer(fperiod)
+ update_odom_and_tf(x, y, theta, vel , omega)
Ros_velpub_node Process
CROSVelPub
- cmd_vel
- Qos
+ start_timer(fperiod)
+ add_vel_cmd(vel_cmd)
+ clear_vel_cmd()
Queue command
Pipe command

Figure 1. Process diagram for software architecture

Therefore, we design the communication architecture in the manner shown in Figure 2. In the design, interprocess communication is facilitated using pipes and queues. As shown in the figure, communication between the main process and another process is designed to communicate on a 1:1 basis using pipes. Reliable data delivery is possible by using pipes in this way. For communication from a process to the main process, N:N communication using a queue is utilized. The queue is designed to be scalable, allowing data to be exchanged and processed among multiple processes. The block option in Queue.get() is set to block until a message from a queue arrives, at which point it is set to pipe.poll(none) to prevent unnecessary CPU usage. The next step is to wait for a pipe without a time limit and no immediate return. In this way, unnecessary CPU usage is prevented.

2.4. Easy integration of multiple devices

This software architecture also provides classes of features for various devices. These classes consist of the functions used by each device, in which only protocol changes for the device can be used for other devices. Providing these classes will help developers create applications that are easier to integrate with various devices and will facilitate various services. Below the method used to implement these classes is described. ControlCore in Figure 1 essentially implements the MultiProcessBase class, which is the basis for implementing multiple processes.

Figure 3 shows the configuration of the MultiProcessBase class. In MultiProcessBase, mpDesc manages instances of the class. Queue sends feedback to ControlCore, which is the main process, and pipe receives commands from the main process. Each queue is periodically read within the thread and data is transferred to the main process. Because other processes are implemented by referring to this, if there is a function to be added, this class can be inherited and easily integrated. The following chart shows the processes currently being constructed.

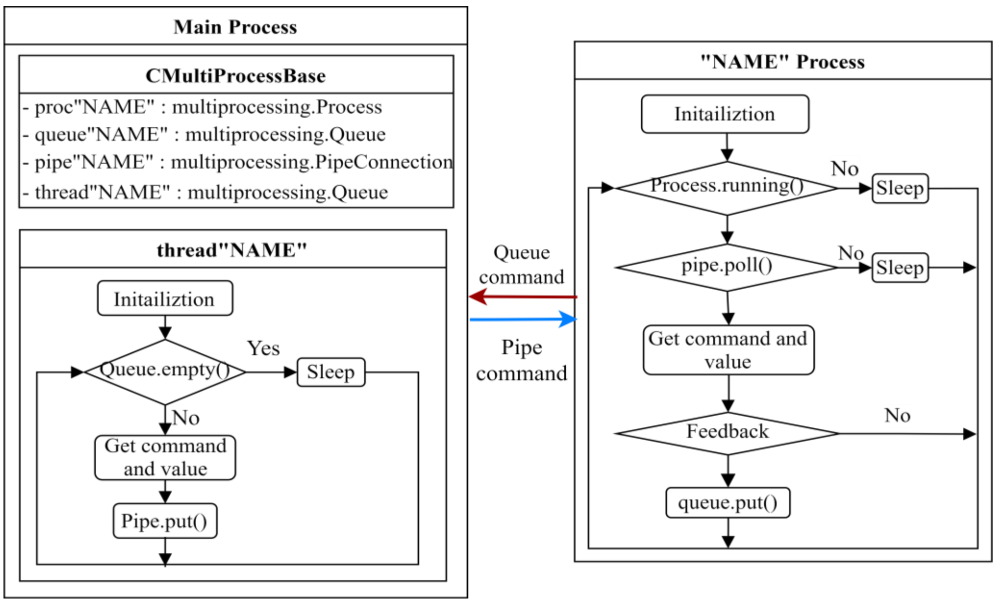


Figure 2. Software communication architecture

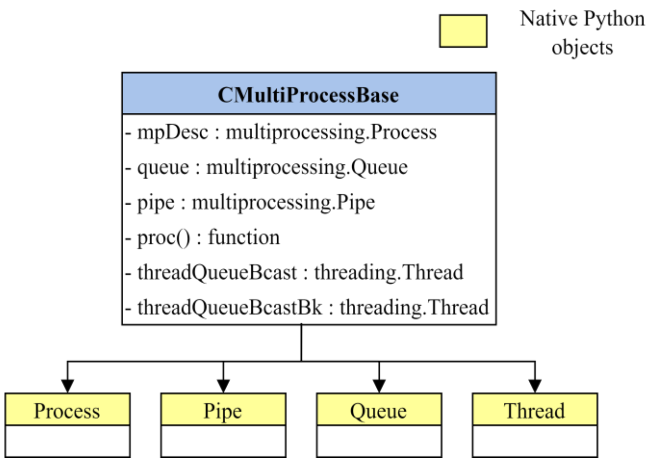

Native Python objects
CMultiProcessBase
- mpDesc : multiprocessing.Process
- queue : multiprocessing.Queue
- pipe : multiprocessing.Pipe
- proc() : function
- threadQueueBcast : threading.Thread
- threadQueueBcastBk : threading.Thread
Process
Pipe
Queue
Thread

Figure 3. MultiProcessBase class

Here, CMobileRobotCtrl refers to the class of controlling mobile robots, creating robot objects, connecting the robot to the communication interface, and controlling the robot according to the data (motor encoder). Figure 4 shows the configuration of the mobile robot class. As shown in the figure, the class CMMobileRobot is a basic class that supports various types of mobile robots. Depending on DriveType, there are common features for several types of robots. CTMRDiff is the basic class of two-wheeled differential drive mobile robots (TWR), referring to a two-wheel differential robot that inherits CMobileRobot. A method for calculating common inverse kinematics and forward kinematics for all TMRs is provided. Currently, it provides a communication interface and information for Kobuki, SSR, and StellaB2. CKobuki2, CSSR, and CStellaB2 are classes that inherit CTMRDiff and process the actual information of each mobile robot.

The serial interface of the mobile robot is provided through the SerialPort class. Figure 5 shows the configuration of CserialPort. CserialPort is a class that provides a serial interface. It uses pyserial and implements it as a separate thread for recv, send, and recvqueue. The queue is utilized between read, recv and recvqueue threads to prevent data losses.

CROSRobotNode is the class of creating a ROS2 RobotNode that publishes or subscribes to ROS topics such as odom, tf, and cmd_vel, among others. Figure 6 shows the configuration of the CROSNodeBase class. CROSNodeBase is a basic class that supports basic ROS nodes. The node name, topic, period, Qos, and other parameters can be set. CROSNodeRobot is a class that inherits CROSNodeBase to generate nodes for robots. Basically, mobile robots publish odom and tf pertaining to the robot's status and subscribe speed. CROSNodeVelPub is the class that publish cmd_vel and is used to test robots. CROSNodeVelPub is a class that inherits CROSNodeBase.

CROSLaunch is the class that executes the any package, which receives the launch descriptor and executes the launch file. Figure 7 shows the configuration of the CROSLaunch class. CROSLaunch receives the package name and path to the launch file, receives the launch descriptor, and executes the launch file as a process. CExternalComm is the class related to external communication, and the more TCP communication is used, the more client connections and servers are generated. Figure 8 shows the configuration of the CExternalComm. CExternalComm generates a server through a socket.

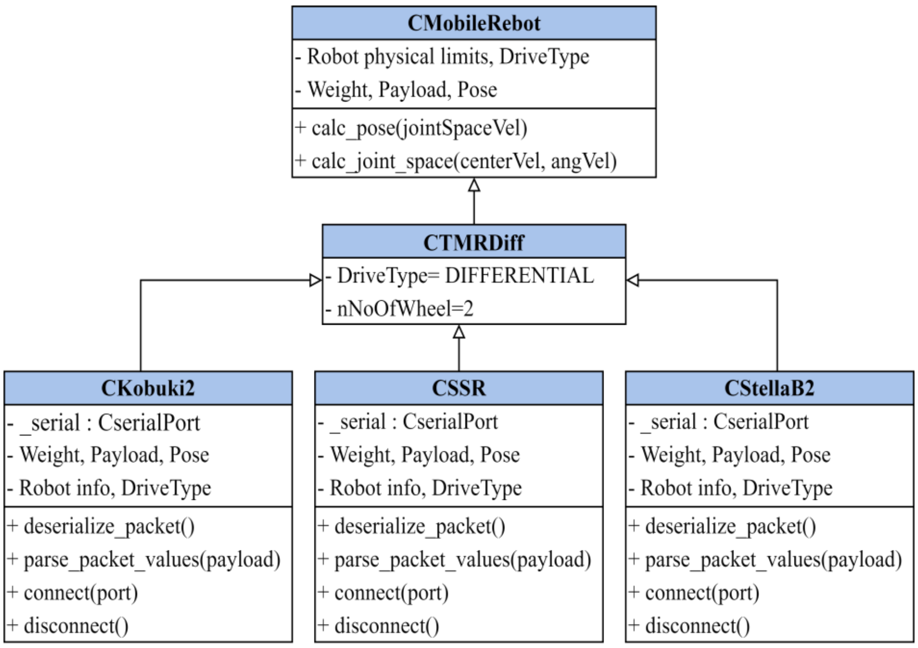


Figure 4. Mobile robot class

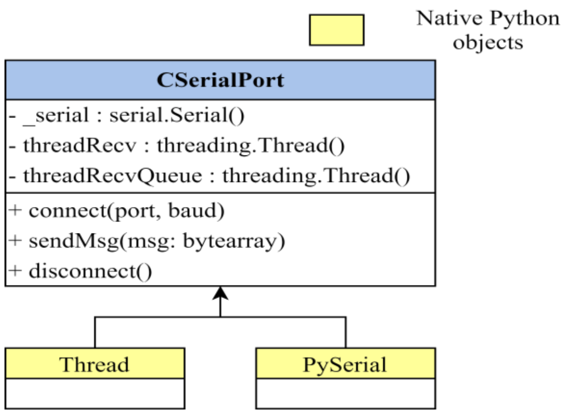

Native Python objects
CSerialPort
- _serial : serial.Serial()
- threadRecv : threading.Thread()
- threadRecvQueue : threading.Thread()
+ connect(port, baud)
+ sendMsg(msg: bytearray)
+ disconnect()
Thread
PySerial

Figure 5. CserialPort class

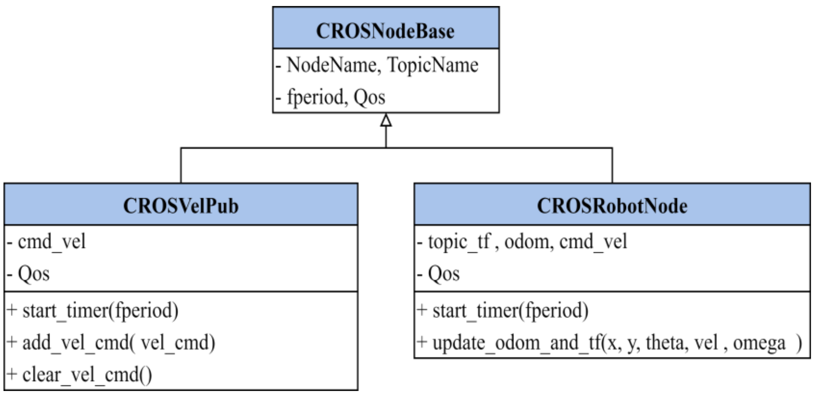


Figure 6. ROSNodeBase class

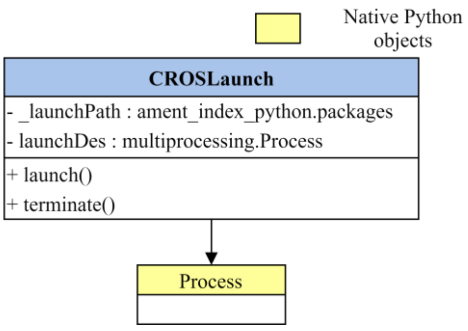

Native Python objects
CROSLaunch
- _launchPath : ament_index_python.packages
- launchDes : multiprocessing.Process
+ launch()
+ terminate()
Process

Figure 7. ROSLaunch class

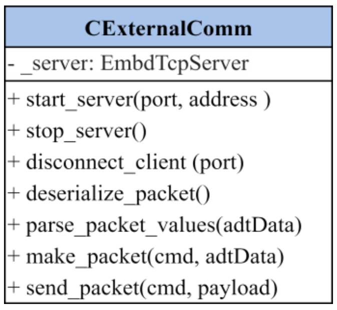

CExternalComm
- _server: EmbdTcpServer
+ start_server(port, address)
+ stop_server()
+ disconnect_client (port)
+ deserialize_packet()
+ parse_packet_values(adtData)
+ make_packet(cmd, adtData)
+ send_packet(cmd, payload)

Figure 8. ExternalComm class

2.5. Monitor process/thread activity

The main window is largely divided into three parts. The first of there is the GUI for external communication. The GUI can check the server address or the client list. The second part is the GUI for the robots. This part allows each robot to communicate, or it provides a speed command to the robot and checks the encoder or the speed. The last part is a GUI that can show a list of threads or processes currently running and can check for communication errors or system errors. It is built in the form of a GUI multiple-document interface (GUI MDI) to make it easy to integrate multiple windows.

3. IMPLEMENTATION OF A SSR ROBOT WITH ITS SOFTWARE ARCHITECTURE

The validity of the designed software was applied to the newly implemented SSR robot. The SSR's main control unit was implemented on the Jetson Xavier NX board. In addition, the most basic navigation of the service robot is implemented through ROS2 Navigation2 to avoid 3D obstacles.

3.1. SSR robot

Robots that apply software are as follows. The SSR is a service robot that developed sensor interface of drive unit and consist of a control board and an app board. For the servo controllers for driving wheels, STM32 is used. It communicates with the motor driver using RS485 communication. For the control boards and app boards, Jetson Xavier NX is used, and the LiDAR and camera required for navigation is connected to the control board. Figure 9 shows the SSR robot. The robot is constructed using the RealSense D435i as the camera, and LiDAR is operated by Hukoyo's URG-04LX-UG01. Table 1 briefly describes the information about the robot by presenting its specifications.

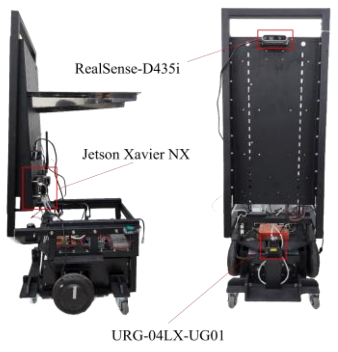


Figure 9. Image of the SSR

Table 1. SSR specifications

Item	Value
Motor driver	10
Wheel radius	15
Number of casters	4
Battery	TABOS X2
LiDAR	URG-04LX-UG01
Camera	RealSense D435i
Payload	100 kg
Velocity	0.3~2 m/s

3.2. Implementation of service application

Regarding the use of the Jetson Xavier NX boards, the system is constructed using JetPack 4.6 provided by NVIDIA. The file system uses Ubuntu 18.04, and in this study, we use the ROS 2.0 Eloquent version, which is supported by the file system and satisfies the time constraints, to use the corresponding navigation stack. Table 2 shows the architecture of the system software configured on the Jetson Xavier NX board.

Table 2. System software architecture

Item	Value
Board	Jetson Xavier NX
OS	Ubuntu 18.04
ROS2	Eloquent
Kernel	4.9.253-tegra

Figure 10 shows the software deployment diagram of experiment. On the control board, the main process, MobileRobotCtrl process, ExternalComm process, ROSNavNode process, ROSRealsenseNode process, ROSRobotNode process, and the ROS2 processes are implemented. The app board implements artificial intelligence applications, in this case TTS, demo GUI. On the app, the main process, texttospeech process are implemented. The app board provide demo GUI. The control board and the app board use the TCP protocol provided by ExternalComm process. Therefore, the deployment of processes can be flexibly implemented with respect to hardware.

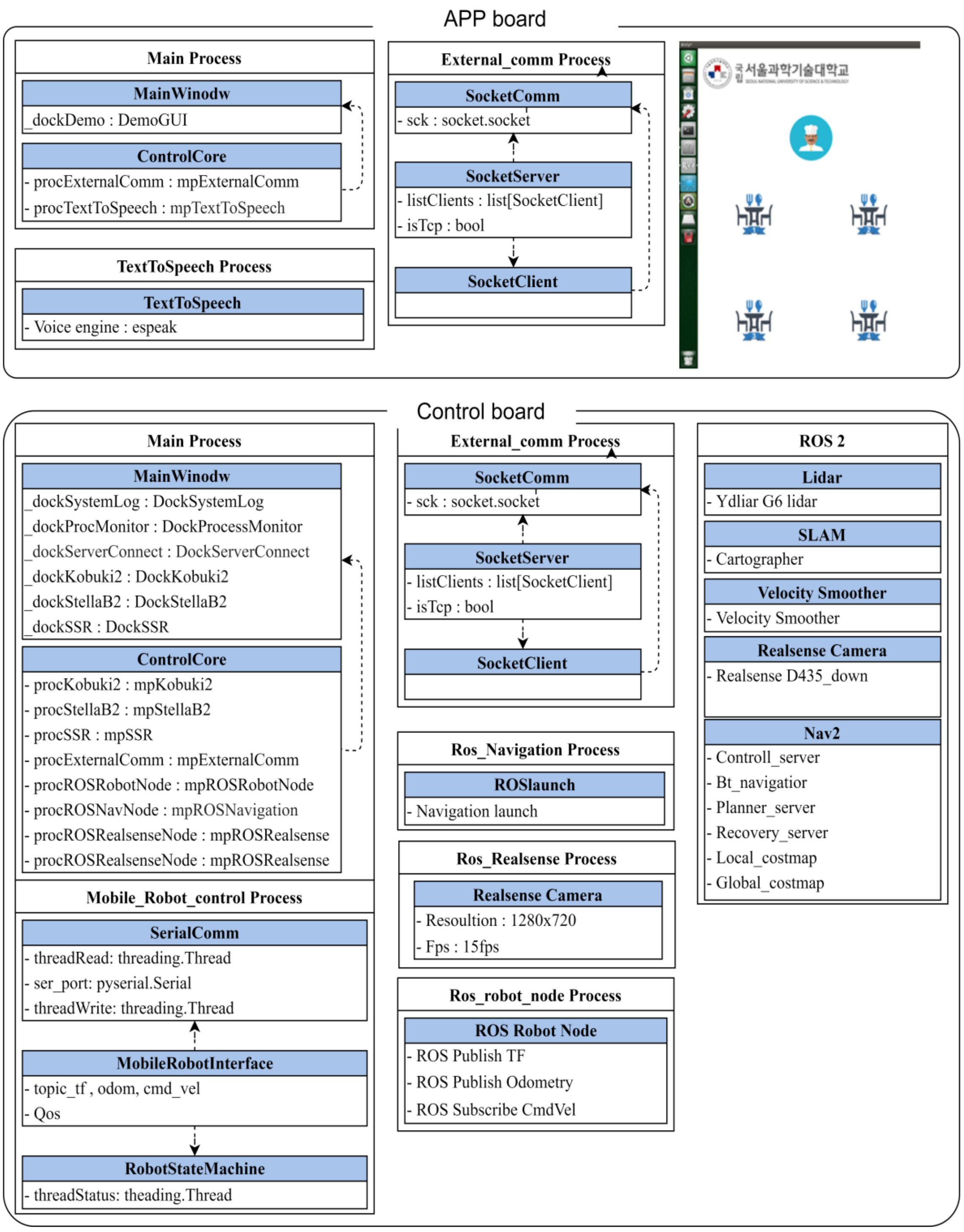
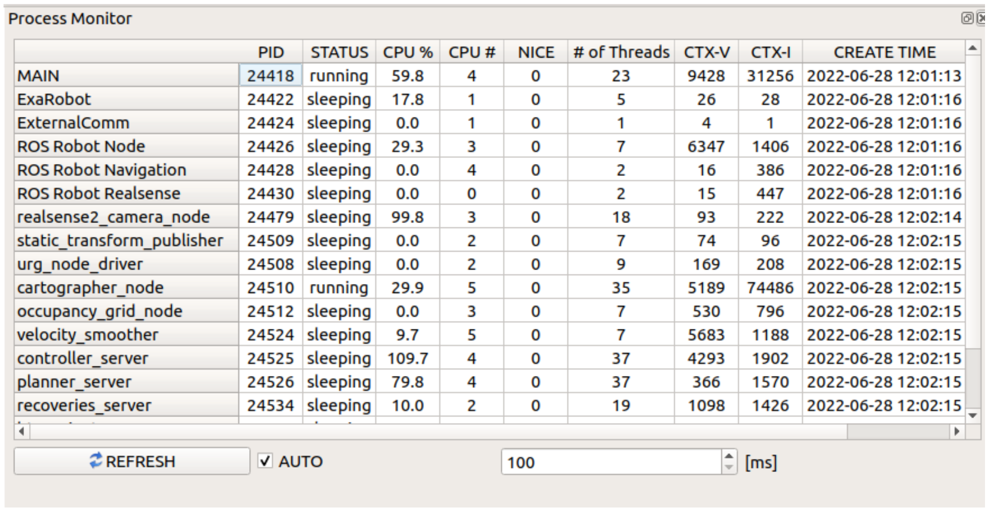


Figure 10. Deployment diagram of service application

Figure 11 shows the results of process monitoring. ROS2 does not offer these services. These services will help analyze and optimize the load of each application. In the software placement diagram shown in Figure 10, information about each process, such as the PID, status, CPU share, and behavioral CPU of the process being implemented, can be checked through the monitor. The CPU share shows the share for each core. In the figure, the ROS robot navigation node runs as a process the procROSRealsense Node, the ROS RobotRealsense node runs as the procROSNavNode, and it runs the python launch file; when the process of Nav2 or RealSense in ROS2 runs, the state enters a sleeping state and occupies 0% of the CPU.


Process Monitor

	PID	STATUS	CPU %	CPU #	NICE	# of Threads	CTX-V	CTX-I	CREATE TIME
MAIN	24418	running	59.8	4	0	23	9428	31256	2022-06-28 12:01:13
ExaRobot	24422	sleeping	17.8	1	0	5	26	28	2022-06-28 12:01:16
ExternalComm	24424	sleeping	0.0	1	0	1	4	1	2022-06-28 12:01:16
ROS Robot Node	24426	sleeping	29.3	3	0	7	6347	1406	2022-06-28 12:01:16
ROS Robot Navigation	24428	sleeping	0.0	4	0	2	16	386	2022-06-28 12:01:16
ROS Robot Realsense	24430	sleeping	0.0	0	0	2	15	447	2022-06-28 12:01:16
realsense2_camera_node	24479	sleeping	99.8	3	0	18	93	222	2022-06-28 12:02:14
static_transform_publisher	24509	sleeping	0.0	2	0	7	74	96	2022-06-28 12:02:15
urg_node_driver	24508	sleeping	0.0	2	0	9	169	208	2022-06-28 12:02:15
cartographer_node	24510	running	29.9	5	0	35	5189	74486	2022-06-28 12:02:15
occupancy_grid_node	24512	sleeping	0.0	3	0	7	530	796	2022-06-28 12:02:15
velocity_smoother	24524	sleeping	9.7	5	0	7	5683	1188	2022-06-28 12:02:15
controller_server	24525	sleeping	109.7	4	0	37	4293	1902	2022-06-28 12:02:15
planner_server	24526	sleeping	79.8	4	0	37	366	1570	2022-06-28 12:02:15
recoveries_server	24534	sleeping	10.0	2	0	19	1098	1426	2022-06-28 12:02:15

Figure 11. Process monitoring results of experiment

3.3. Autonomous navigation

A basic function of a service robot is navigation, which is an autonomous driving function that allows the robot to move safely to the desired point while avoiding obstacles. In ROS2, by providing a navigation stack as a package, navigation can be implemented with the package provided without the need to implement the algorithm directly [26]. The package used to implement the ROS2 navigation stack is as follows: i) DWB_controller; ii) navfn_planner; iii) local_costmap; iv) global_costmap; v) recoveries; and vi) bt_navigator.

DWB_controllers correspond to local_planners and are commonly used to create paths to avoid obstacles around a robot. During ROS2 navigation, dwb_controller is used as the default controller. Navfn_planner corresponds to global_planner and generates a path from the start point of the robot to the destination. Local_costmap is a map used by dwb_controller to search for obstacles that are detected dynamically around the robot. Global_costmap is a map used in navfn_planner to create a route to a destination by avoiding static obstacles. Recovery means waiting until a costmap or obstacle disappears such that if a problem occurs during the operation of the navigation stack, the problem can be resolved. In this study, the point cloud of the camera was used using spatio-temporal voxel layer (STVL) to avoid 3D obstacles, and autonomous navigation was implemented to avoid 3D obstacles [27].

4. EXPERIMENTAL RESULTS AND VALIDATION

Autonomous navigation was performed in the 3D obstacle environment of the service robot using SSR, a robot to which the developed software architecture was applied. The space for the experiment used a space containing 3D obstacles such as a table, which 2D LiDAR cannot detect. Figure 12 shows the actual experimental indoor environment used for the experiment here. The moving point has starting position Ⓢ and target points labeled as ①, ② and ③. The experimental environment was taken at the point indicated on the map.

The experiment involves transferring the location of each target point to the robot through TCP communication so that the robot can autonomously drive to that point while avoiding obstacles. When moving to the target point, TTS is used to signify the arrival of the robot at the target point. Figure 13 shows two Rviz images and the actual scenario during autonomous driving. The figure was taken when the robot drives toward points ① to ②. The left Rviz image is a costmap indicated by circles using only the LiDAR device. In the image, the table can't be detected. In the real-world image, there is a table in front of the robot. As indicated by the right Rviz image, the table recognized by the camera is expressed in voxels and the obstacle is avoided by using it in the costmap. Figure 14 shows the navigation driving path of the robot in the experiment while avoiding 2D and 3D obstacles. In a three-dimensional environment, it is possible to create voxels with a camera and apply them to a costmap to confirm that the robot will move to the desired point without a collision with the table.

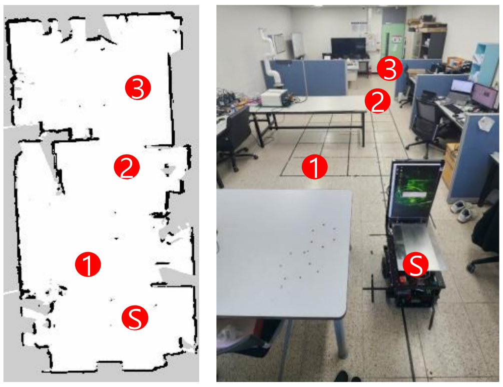


Figure 12. A 2D map of the indoor environment with 3D obstacles

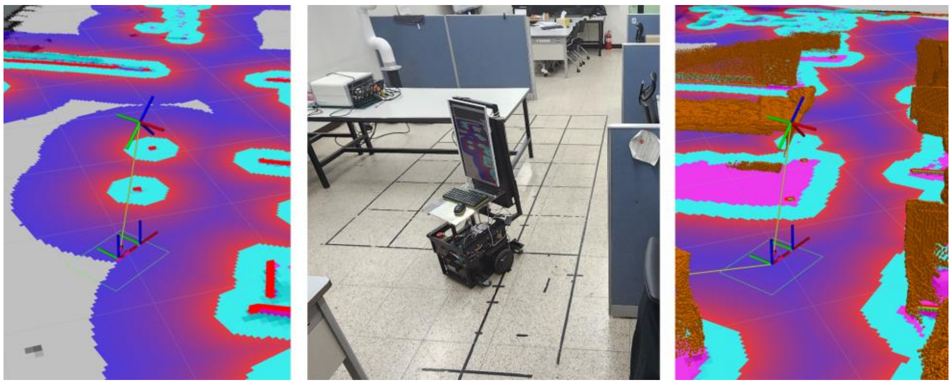


Figure 13. Navigation driving

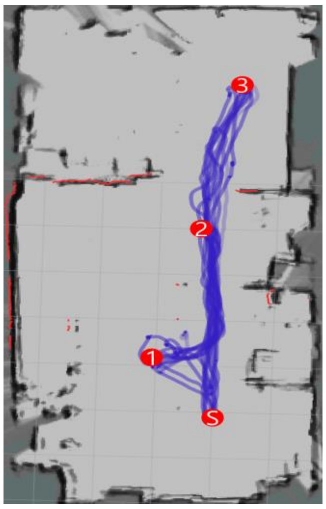


Figure 14. Autonomous navigation result while avoiding 2D and 3D obstacles

5. CONCLUSION

In this paper, we construct ROS2-based scalable software architecture for service robots and designed an inter-process communication methodology. For flexibility during development efforts, the software architecture is designed as a scalable type of architecture that can be applied to various robots, instilling the advantage of reducing the development period and development cost during the development of a new robot while also allowing easy integration of various devices. It is possible to scale easily by adding multiple other processes using the class. The proposed architecture was verified by applying it to SSR, a Jetson Xavier NX-based service robot. We implemented three-dimensional obstacle avoidance autonomous driving, which is a basic function of a service robot, and verified that the proposed architecture is useful as a form of service robot software architecture by applying it to various robots.

ACKNOWLEDGEMENTS

This work has been financially supported by SeoulTech (Seoul National University of Science and Technology).

REFERENCES

[1] B. S. Kim, J. H. Choi, C. Lee, and S. Oh, "Development and control of an autonomous user-tracking golf caddie robot," *Journal of Institute of Control, Robotics and Systems*, vol. 28, no. 2, pp. 102–108, 2022, doi: 10.5302/J.ICROS.2022.21.0216.

[2] D. Belanche, L. V. Casaló, C. Flavián, and J. Schepers, "Service robot implementation: a theoretical framework and research agenda," *Service Industries Journal*, vol. 40, no. 3–4, pp. 203–225, 2020, doi: 10.1080/02642069.2019.1672666.

[3] T. Dewi, P. Risma, and Y. Oktarina, "Fruit sorting robot based on color and size for an agricultural product packaging system," *Bulletin of Electrical Engineering and Informatics*, vol. 9, no. 4, pp. 1438–1445, 2020, doi: 10.11591/eei.v9i4.2353.

[4] C. Steppa and T. L. Holch, "HexagDLy—processing hexagonally sampled data with CNNs in PyTorch," *SoftwareX*, vol. 9, pp. 193–198, 2019, doi: 10.1016/j.softx.2019.02.010.

[5] C. Odabasi, F. Graf, J. Lindermayr, M. Patel, S. D. Baumgarten, and B. Graf, "Refilling water bottles in elderly care homes with the help of a safe service robot," in *2022 17th ACM/IEEE International Conference on Human-Robot Interaction*, 2022, pp. 101–109, doi: 10.1109/HRI53351.2022.9889391.

[6] M. Quigley *et al.*, "ROS: an open-source robot operating system," *ICRA workshop on open source software*, vol. 3, pp. 4754–4759, 2009.

[7] S. Barut, M. Boneberger, P. Mohammadi, and J. J. Steil, "Benchmarking real-time capabilities of ROS 2 and OROCOS for robotics applications," in *2021 IEEE International Conference on Robotics and Automation*, 2021, pp. 708–714, doi: 10.1109/ICRA48506.2021.9561026.

[8] 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, doi: 10.1145/2968478.2968502.

[9] G. P. -Castellote, "OMG data-distribution service: architectural overview," in *Proceedings - 23rd International Conference on Distributed Computing Systems Workshops, ICDCSW 2003*, 2003, pp. 200–206, doi: 10.1109/ICDCSW.2003.1203555.

[10] A. Corsaro and D. C. Schmidt, "The data distribution service–the communication middleware fabric for scalable and extensible systems-of-systems," in *System of Systems*, Rijeka, Croatia: InTech, 2012, pp. 13–30, doi: 10.5772/30322.

[11] J. Park, R. Delgado, and B. W. Choi, "Real-time characteristics of ROS 2.0 in multiagent robot systems: an empirical study," *IEEE Access*, vol. 8, pp. 154637–154651, 2020, doi: 10.1109/ACCESS.2020.3018122.

[12] M. S. Kim, R. Delgado, and B. W. Choi, "Comparative study of ROS on embedded system for a mobile robot," *Journal of Automation, Mobile Robotics and Intelligent Systems*, vol. 12, no. 3, pp. 61–67, 2018, doi: 10.14313/JAMRIS_3-2018/19.

[13] Z. Faiz, V. Srivastava, and S. Khoje, "Virtual voice assistant for smart devices," *ECS Transactions*, vol. 107, no. 1, pp. 4315–4326, 2022, doi: 10.1149/10701.4315ecst.

[14] M. T. H. Fuad *et al.*, "Recent advances in deep learning techniques for face recognition," *IEEE Access*, vol. 9, pp. 99112–99142, 2021, doi: 10.1109/ACCESS.2021.3096136.

[15] A. A. Arriany and M. S. Musbah, "Applying voice recognition technology for smart home networks," in *2016 International Conference on Engineering & MIS (ICEMIS)*, 2016, pp. 1–6, doi: 10.1109/ICEMIS.2016.7745292.

[16] R. Padilla, S. L. Netto, and E. A. B. d. Silva, "A survey on performance metrics for object-detection algorithms," in *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*, 2020, pp. 237–242, doi: 10.1109/IWSSIP48289.2020.9145130.

[17] G. K. Adam, N. Petrellis, and L. T. Doulos, "Performance assessment of linux kernels with PREEMPT_RT on ARM-based embedded devices," *Electronics*, vol. 10, no. 11, pp. 1–28, 2021, doi: 10.3390/electronics10111331.

[18] R. Delgado, J. Park, and B. W. Choi, "Open embedded real-time controllers for industrial distributed control systems," *Electronics*, vol. 8, no. 2, pp. 1–18, 2019, doi: 10.3390/electronics8020223.

[19] U. C. Shin and B. W. Choi, "Performance evaluation of real-time mechanisms on open embedded hardware platforms," *Journal of Institute of Control, Robotics and Systems*, vol. 23, no. 1, pp. 60–66, 2017, doi: 10.5302/J.ICROS.2017.16.0181.

[20] R. Delgado, Y. H. Jo, and B. W. Choi, "RT-AIDE: a RTOS-agnostic and interoperable development environment for real-time systems," *IEEE Transactions on Industrial Informatics*, vol. 19, no. 3, pp. 2772–2781, 2023, doi: 10.1109/TII.2022.3182790.

[21] G. v. Rossum and F. L. Drake, *An introduction to python*. Bristol, UK: Network Theory Limited, 2003.

[22] S. Cass, "Top programming languages to learn in 2021: python dominates as the de facto platform for new technologies," *IEEE Spectrum*, Aug. 24, 2021, [Online]. Available: https://spectrum.ieee.org/top-programming-languages-2021 (accessed Mar. 16, 2023).

[23] A. Paszke *et al.*, "PyTorch: an imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, pp. 1–12, 2019.

[24] C. Liu *et al.*, "Detecting TensorFlow program bugs in real-world industrial environment," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 55–66, doi: 10.1109/ASE51524.2021.9678891.

[25] D. Beazley, "Understanding the python GIL," in *PyCON Python Conference. Atlanta, Georgia*, 2010, pp. 1–62.

[26] S. Macenski, F. Martin, R. White, and J. G. Clavero, "The marathon 2: a navigation system," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 2718–2725, doi: 10.1109/IROS45743.2020.9341207.

[27] S. Macenski, D. Tsai, and M. Feinberg, "Spatio-temporal voxel layer: a view on robot perception for the dynamic world," *International Journal of Advanced Robotic Systems*, vol. 17, no. 2, pp. 1–14, 2020, doi: 10.1177/1729881420910530.

BIOGRAPHIES OF AUTHORS

Yong Hwan Jo received the B.S. degree in Electrical and Information Engineering from Seoul National University of Science and Technology, Seoul, South Korea, in 2021, where he is currently working toward the M.S degree in Electrical and Information Engineering. His research interests include real-time systems, industrial control and automation and embedded systems. He can be contacted at email: jyh159@seoultech.ac.kr.

Se Yeon Cho received the B.S. degree in Electronic Engineering from Shinhan University, Uijeongbu, South Korea, in 2020, where he is currently working toward the M.S degree in Electrical and Information Engineering. His research interests include real-time systems, industrial control and automation, embedded systems. He can be contacted at email: seyeon@seoultech.ac.kr.

Byoung Wook Choi received the M.S. and Ph.D. degrees in Electrical Engineering from Korea Advanced Institute of Science and Technology (KAIST), Seoul, South Korea in 1988 and 1992, respectively. He is currently the dean of the College of Creativity and Convergence Studies and a professor in the Department of Electrical and Information Engineering at Seoul National University of Science and Technology. Previously, he was a principal research engineer in LG Industrial Systems from 1992-2000 and a professor in Sun Moon University from 2000-2005. He was the CEO of Embedded Web Co., Ltd. from 2001-2003. Also, he was a senior fellow at Nanyang Technological University in Singapore from 2007-2008. Prof. Choi has published textbooks on embedded linux. His current research interests include real-time systems design, embedded systems, and intelligent robot software. He can be contacted at email: bwchoi@seoultech.ac.kr.