IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received August 11, 2020, accepted August 15, 2020, date of publication August 20, 2020, date of current version September 2, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3018122

Real-Time Characteristics of ROS 2.0 in Multiagent Robot Systems: An Empirical Study

JAEHO PARK, RAIMARIUS DELGADO, (Member, IEEE), AND BYOUNG WOOK CHOI, (Member, IEEE)
Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul 01811, South Korea

Corresponding author: Byoung Wook Choi (bwchoi@seoultech.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science and ICT (MSIT) of the Korean government under Grant 2019R1F1A1063547.

ABSTRACT Due to its ease of use and flexibility, the robot operating system (ROS) is increasingly becoming the most popular middleware for robot applications, even in multiagent systems. Since ROS 1.0 does not satisfy real-time requirements, ROS 2.0 was developed, and it improved the communication stack with the real-time data distribution service (DDS) protocol. However, the actual performance level to be expected is still unknown and can largely depend on the operating system and the kernel being used, the DDS distribution, and the overall software load of the system. In this article, we present an empirical study that evaluates the real-time performance of ROS 2.0 in both the system and communication software layers. In the system layer, the deterministic behavior of the ROS 2.0 nodes is thoroughly observed with regard to whether the tasks are schedulable and can function within the specified temporal deadline. In the communication layer, special attention is devoted to the rate of data loss and the overall latency of messages between nodes. Experiments are performed in various working conditions; for example, the system load is increased to define the real-time performance of the tasks. For reference, the results are compared with the those from the traditional ROS variation. Moreover, we implement a multiagent service robot system to verify the suitability of ROS 2.0 for real-world applications. Our results show that the application of ROS 2.0 is more suitable than that of ROS 1.0 in terms of real-time performance.

INDEX TERMS Multi-agent system, performance evaluation, real-time operating systems, robot operating system.

I. INTRODUCTION

With recent aging and low fertility trends found around the world, robots have emerged as an alternative to compensate for the lack of human productivity. Interest and demand for service robots are increasing due to social factors such as the pursuit of an improved quality of life [1]. Most modern robotic systems employ a distributed system architecture that is composed of sophisticated control software synchronizing tasks on multiple controllers that are connected through an underlying network. Some examples are multirobot systems in heterogeneous environments [2], cloud-based frameworks [3], and industrial robots for large-scale automation systems [4].

Robot software is developed using complex algorithms, such as navigation and simultaneous localization and mapping (SLAM), which are time consuming to develop. Nonetheless, interest in the robot operating system (ROS) is steadily increasing [5]–[7].

ROS enables easy and rapid software development with various tools, libraries, and rules to implement complex control algorithms in various robot platforms. However, since the robot shares the working environment with humans in real-time, it can cause physical damage to the user if malfunctions occur due to system latency. Therefore, real-time constraints must be satisfied for stable operation [8]. Since ROS does not satisfy real-time constraints, researchers have proposed several approaches to making ROS function in real-time.

In [9], a host-guest system was proposed in which real-time tasks were executed in the guest system and nonreal-time ROS nodes were operated in the host system. In this approach, communication overhead was introduced, which could lead to unexpected delays that affect the overall performance of the system. Moreover, the manufacturing cost is

The associate editor coordinating the review of this manuscript and approving it for publication was Laxmisha Rai.

also an issue due to the multiple hardware requirements. RT-ROS in [10] addresses the hardware issue by implementing a hybrid operating system (OS) in a single-board, multicore environment. This hybrid system allocates a real-time operating system (RTOS) and general-purpose operating system (GPOS) on each core, where the real-time and nonreal-time processes execute, respectively. However, this approach requires extensive modification of the libraries and packages of the ROS, which significantly lengthen the development time. A dual-kernel environment of Xenomai was introduced by Delgado *et al.* [11], [12], with the real-time tasks and ROS nodes connected through the cross-domain datagram protocol interface. Although the real-time tasks and ROS nodes can run seamlessly, this approach still used the standard TCP-based communication layer of the ROS, namely, TCPROS, which was not suitable for lossy networks (e.g., wireless networks) due to their trade-offs in accuracy and performance.

To support real-time communication, ROS 2.0 was developed with distributed services (DDS) as middleware for internode communication. DDS also uses the quality of service (QoS) profile to provide benefits such as real-time communication, scalability, performance enhancement, and security [13]. Data transmission uses real-time publish-subscribe (RTPS) protocol and multicasting and connectionless transmission methods, such as UDP/IP. These enable stable operation even in unstable networks where data loss could occur, such as in wireless networks, for example [14]. To develop a real-time system using ROS 2.0, an indicator to evaluate whether ROS 2.0 can meet real-time constraints must be defined.

Current research studies that evaluates the real-time performance of ROS 2.0 are as follows. Chen [15] evaluated the latency and message loss rate according to the QoS profile and communication distance of ROS 2.0 DDS in a wireless network environment. Maruyama *et al.* [16] extended this work using data size as the performance metric in various forms of communication, including rosbridge between ROS and ROS 2.0. However, the stability of ROS 2.0 is vague because the experiments were conducted on an idle environment without an actual system load. Gutiérrez *et al.* in [17] presented an extensive evaluation of ROS 2.0, evaluating the communication stack according to the data size, communication speed, system load, and network traffic. However, this study was conducted only at a communication level, which does not guarantee real-time performance of the development environment. Thus, it was exceedingly difficult to determine whether the entire system provided real-time performance and implementing an ROS 2.0-based system that can satisfy real-time constraints as a whole is a challenge.

This article aims to provide the indicators that are necessary to implement a ROS-based system that should satisfy real-time constraints by conducting an empirical study on the real-time characteristics of ROS 2.0 compared to ROS 1.0. The real-time performance was evaluated based on two items: the software stack and communication. To evaluate the deterministic behavior under various conditions, the evaluation was conducted under the conditions of an environment with no system load and an environment with a system load. A software stack performance evaluation was performed to evaluate the real-time performance of the development software architecture. The performance evaluation assessed the schedulability of the Linux kernel and ROS nodes. Linux kernel scheduling latency evaluation was performed to verify that real-time scheduling was possible, and the ROS node evaluation was executed to verify that each task was capable of deterministic behavior in a multitasking environment in which multiple ROS tasks were executed simultaneously. Since the schedulability of a real-time task has a great dependence on the timing accuracy of the task, a timing analysis of real-time tasks was performed, and the results were represented by the statistical mean, maximum, minimum, and standard deviation [18]. A system periodicity analysis was also conducted to evaluate whether the job could be run within the deadline.

A communication performance evaluation was performed to evaluate the network performance with respect to the real-time performance and stability of a ROS. Two metrics were evaluated for the message loss rates and latency according to the data size and communication frequency. The message loss rate was defined as the ratio of messages lost from the receiving node during communication between the two nodes, and the communication latency was defined as the time difference from the sending point of the message to its receiving point in a round-trip communication.

Finally, based on a real-time performance evaluation, we implemented a multiagent service robot that was able to satisfy the real-time constraints and verified the effectiveness of the research results by evaluating whether the implemented system met the real-time constraints. A multiagent service robot was executed based on ROS 2.0 and PREEMPT_RT [19] to satisfy the real-time constraints, and two mobile robots performed navigation to provide services. Two ROS 2.0 navigation stacks were used for this task. However, since the navigation stack provided by ROS 2.0 is not considered for multirobots, the following two problems must be solved before we can perform navigation of two or more robots.

The first problem is that when two navigation stacks are executed, the node names and the coordinate frame (tf) of the robot overlaps, and thus, the target robot to which the navigation should be performed cannot be specified. To solve this problem, the nodes of each robot were grouped, and a prefix was added to the tf. The second problem is that the tf of the target robot that is to perform navigation in the ROS 2.0 Global Planner is hardcoded, and thus, even if two navigation stacks are used, two or more robots cannot be driven. To solve this problem, the tf of the target robot that performed navigation in the global planner was modified and specified as a parameter. Through these experiments, we proved that the real-time performance of a ROS 2.0 based multiagent system is superior in real-time compared to the

system using ROS 1.0 in terms of the proposed performance measures.

This article is organized as follows: Section 2 provides a real-time performance evaluation around the software stack. Section 3 provides a communication performance evaluation. Section 4 describes a multiagent service robot that meets real-time constraints, and the last section consists of conclusions that were made based on our research.

II. SOFTWARE STACK EVALUATION

This section describes the performance evaluation of a ROS software stack that focuses on real-time performance in the development environment. For ROS-based systems, real-time scheduling in the OS must be possible to meet real-time constraints. Additionally, in a multitasking environment where multiple nodes are running, each node must meet deadlines.

Therefore, the experiment conducted the performance evaluation with two metrics: the real-time schedulability of a Linux kernel that constitutes the software stack, and a periodicity evaluation of the nodes in a multitasking environment. We confirmed the deterministic behavior of the software stack, which is an important indicator in determining the safety of the system, and we evaluated whether the node could be executed while satisfying the hard temporal deadlines. In addition, the stress-ng tool [20] was used to generate the CPU, memory, and I/O loads on the system. This task was performed to assess whether the system was capable of stable operation even in unstable environments.

A. SCHEDULING LATENCY

This section details the evaluation of the scheduling latency between ROS 1.0 and ROS 2.0. The performance evaluation was performed using the benchmark tool called cyclictest [21]. The implemented software stack architecture is shown in Fig. 1. Because ROS 1.0 does not officially support any RTOSs, it was implemented on top of the standard Linux. On the other hand, ROS 2.0 was developed to meet real-time constraints when implemented on an RTOS. In this article, we have utilized the Linux kernel patched with PREEMPT_RT [9].

Fig. 2 shows the results of evaluating the scheduling latency in an idle environment. The scheduling latency is defined as the time difference between the actual task activation time and the configured period. In our evaluation, we configured a single real-time task to run with a period and deadline of 1 ms. To ensure that the multicore hardware did not affect the results, we configured the CPU affinity of the task to run on CPU0. Fig. 2a shows the scheduling latency evaluation of the ROS 1.0 system.

It can be seen that the measured maximum latency is 290 us, which is significantly higher than the 11 us of ROS 2.0, as shown in Fig. 2b. Additionally, the ROS 1.0 results show a vast distribution in comparison to those of the ROS 2.0, where most of the measured data samples are concentrated below the maximum value. Therefore, ROS 2.0 satisfied the hard temporal deadlines better than ROS 1.0 and

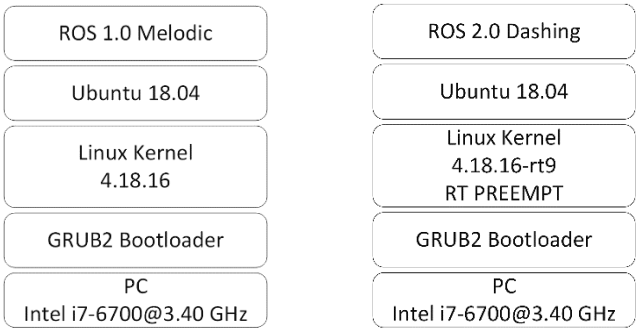

FIGURE 1. Software stack architecture of ROS 1.0 and ROS 2.0 implemented on an Intel Core i7 PC. (a) ROS 1.0 Melodic on the standard Linux kernel. (b) ROS 2.0 Dashing on top of the RT_PREEMPT patched Linux kernel.

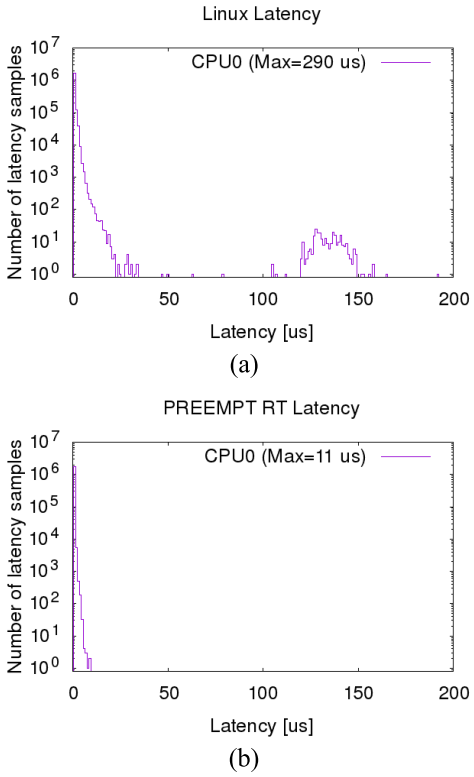

FIGURE 2. Scheduling latency in an idle environment. (a) ROS 1.0. (b) ROS 2.0.

enabled deterministic behavior. In addition, ROS 1.0 had a larger distribution than ROS 2.0 and thus did not have consistent operation. Therefore, the ROS 2.0 system provided higher real-time performance than ROS 1.0 and operated stably.

We performed the same evaluation on an environment running the stress tool, called stress-ng. In a stressed environment, the CPUs are constantly running various calculations to simulate a heavy computational load. The results of the evaluation are shown in Fig. 3.

We measured a maximum latency of 1280 us and 26 us for ROS 1.0 and ROS 2.0, respectively. Compared with the results of the performance evaluations in idle environments,

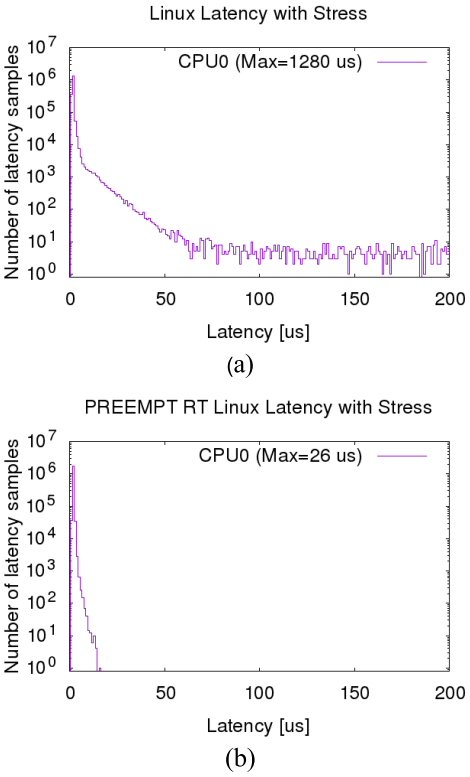

FIGURE 3. Scheduling latency in a stressed environment. (a) ROS 1.0. (b) ROS 2.0.

the maximum latency was increased for both the ROS 1.0 and ROS 2.0 systems. In this case, ROS 1.0 was not able to satisfy the deadline of 1 ms. On the other hand, even with a slight increase in the maximum latency, ROS 2.0 was able to meet the temporal constraint. With these results, we concluded that ROS-based robot systems must implement ROS 2.0 to satisfy real-time constraints.

B. TASK PERIODICITY IN A MULTITASKING ENVIRONMENT

The schedulability of nodes is highly dependent on the timing correctness of each node. The periodicity of the system was analyzed to verify that all of the nodes could run within the deadline [18]. The behavior of the execution of a node is illustrated in Fig. 4. The goal of the experiment is to verify that the nodes implemented in ROS 1.0 and ROS 2.0 can exhibit deterministic behavior based on periodicity tests.

For the experiment, we created four nodes with different priorities and periods, as follows. Node1, shown in (a) of Fig. 5, has a priority of 97 and a period of 5 ms. Node2 has a priority of 96 and a period of 10 ms. Node3 has a priority of 95 and a period of 50 ms. Node3 has a priority of 94 and a period of 100 ms. The nodes were executed to start approximately at the same time, and then, they were experimented on for 30 minutes. The node with the higher number has the higher priority.

Boxplots for each node are shown in Fig. 5 to clearly identify the difference in periodicity between ROS 1.0 and

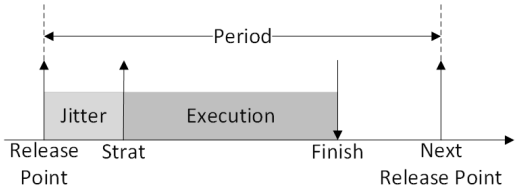

FIGURE 4. Timeline that describes the behavior of a ROS node with a given period.

TABLE 1. Periodicity of ROS 1.0 nodes in an idle multitasking Environment.

ROS 1.0	Node1		Node2	
Metric (ms)	T_{period}	T_{jitter}	T_{period}	T_{jitter}
avg.	4.999062	0.016156	9.999163	0.009272
max.	8.012010	3.012010	14.581200	4.584600
min.	1.992630	0.000000	5.415400	0.000000
st. d.	0.026875	0.021498	0.022926	0.020984
ROS 1.0	Node3		Node3	
Metric (ms)	T_{period}	T_{jitter}	T_{period}	T_{jitter}
avg.	49.99910	0.011217	99.99904	0.011097
max.	50.12970	0.147700	100.14200	0.142000
min.	49.85230	0.000000	99.86260	0.000000
st. d.	0.018505	0.014744	0.01851	0.014856

TABLE 2. Periodicity of ROS 2.0 nodes in an idle multitasking environment.

ROS 2.0	Node1		Node2	
Metric (ms)	T_{period}	T_{jitter}	T_{period}	T_{jitter}
avg.	4.999819	0.000569	9.999830	0.000719
max.	5.053560	0.055300	10.023400	0.051460
min.	4.944700	0.000000	9.948540	0.000000
st. d.	0.001726	0.001640	0.001821	0.001682
ROS 2.0	Node3		Node3	
Metric (ms)	T_{period}	T_{jitter}	T_{period}	T_{jitter}
avg.	49.99974	0.001440	99.99973	0.001296
max.	50.03450	0.050500	100.09200	0.097400
min.	49.94950	0.000000	99.90260	0.000000
st. d.	0.002194	0.001674	0.00418	0.003988

ROS 2.0. The results of the timing analysis are shown with the statistical average (avg), maximum (max), minimum (min), and standard deviation (σ) values of each timing metric. Table 1 shows the results of ROS 1.0, and Table 2 shows the results of ROS 2.0. Since all four nodes had lower jitter than ROS 1.0, ROS 2.0 satisfied the periodicity and enabled the deterministic behavior. In addition, the standard deviation was lower than that of ROS 1.0 and enabled a consistent and stable system operation. In Table 2, the task with the higher priority shows a low standard deviation, which means better deterministic behavior. In Table 1, however, it can be seen that the priority had no effect on the periodicity.

The same experiment was conducted in an environment with stress to evaluate the change in the performance in an unstable environment. Fig. 6 shows the boxplot for each node. Table 3 shows the results for ROS 1.0, and Table 4 shows the results for ROS 2.0. In an environment with stress, ROS 1.0 had significantly degraded performance compared to idle environments on all four nodes. Especially for Node1, the maximum jitter increased by more than twice the period specified.

Unlike ROS 1.0, ROS 2.0 showed no significant performance degradation and showed better performance than ROS

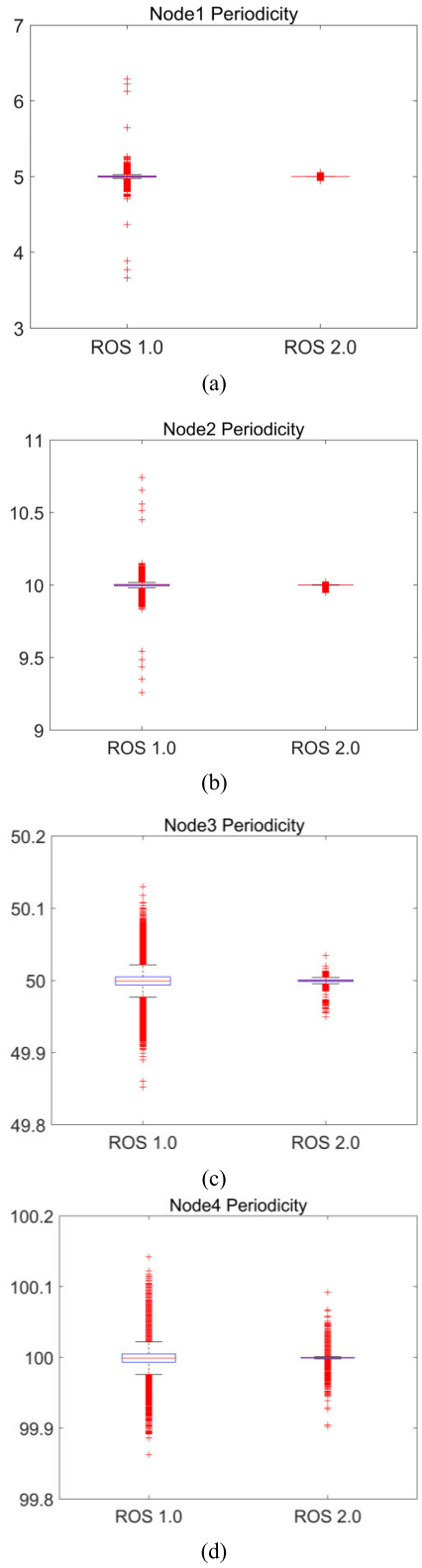

FIGURE 5. Periodicity of each node in an idle multitasking environment. (a) Node1 (b) node 2 (c) node 3 (d) node 4.

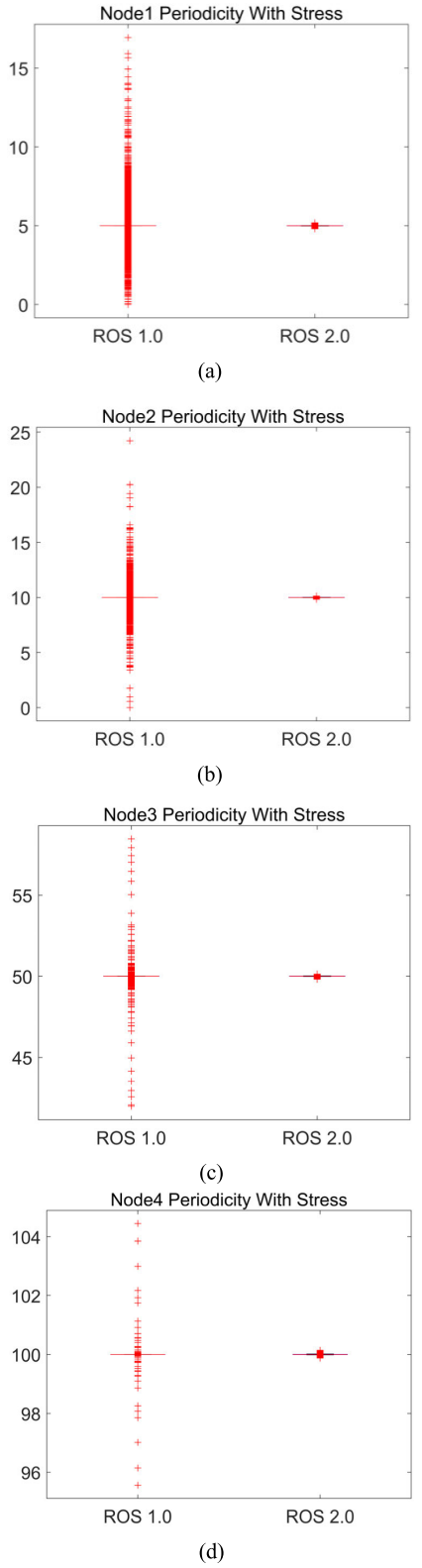

FIGURE 6. Periodicity of each node in a stressed multitasking environment. (a) Node1 (b) node 2 (c) node 3 (d) node 4.

1.0 in an idle environment. ROS 1.0 could cause problems due to malfunctions when implementing real-time applications with high CPU and memory utilization, such as navigation and SLAM. Table 3 shows detailed experimental data. However, ROS 2.0 satisfies real-time constraints and enables stable system operation even when such applications

TABLE 3. Periodicity of ROS 1.0 nodes in a stressed multitasking environment.

ROS 1.0	Node1		Node2	
Metric (ms)	T_{period}	T_{jitter}	T_{period}	T_{jitter}
avg.	4.999929	0.014473	9.99980	0.016139
max.	16.94670	11.946700	24.21960	14.219600
min.	0.000486	0.000000	0.00103	0.000000
st. d.	0.189248	0.188693	0.20538	0.204749
ROS 1.0	Node3		Node3	
Metric (ms)	T_{period}	T_{jitter}	T_{period}	T_{jitter}
avg.	49.99969	0.008427	99.99964	0.004417
max.	58.46030	8.460300	104.44700	4.447000
min.	41.98550	0.000000	95.55400	0.000000
st. d.	0.158163	0.157938	0.09101	0.090909

TABLE 4. Periodicity of ROS 2.0 nodes in a stressed multitasking environment.

ROS 2.0	Node1		Node2	
Metric (ms)	T_{period}	T_{jitter}	T_{period}	T_{jitter}
avg.	4.998924	0.006557	9.998932	0.010649
max.	5.189790	0.192360	10.133600	0.161850
min.	4.807640	0.000000	9.838150	0.000000
st. d.	0.014033	0.012453	0.019689	0.016595
ROS 2.0	Node3		Node3	
Metric (ms)	T_{period}	T_{jitter}	T_{period}	T_{jitter}
avg.	49.99905	0.008701	99.99908	0.008463
max.	50.12540	0.175900	100.13900	0.139000
min.	49.82410	0.000000	99.87470	0.000000
st. d.	0.015921	0.013367	0.01633	0.014001

TABLE 5. Attributes of the real-time tasks for response time analysis.

Task	Period	Deadline	Execution Time	Priority
Node1	10	10	3	99
Node2	20	20	5	98
Node3	40	40	10	97
Node4	80	80	15	96

are implemented. In Table 4, the task with the higher priority shows a low standard deviation, which means better deterministic behavior even in the stressed environment. These results indicate that the system based on ROS 2.0 is feasible for real-time applications.

Although ROS 1.0-based systems do not officially support RTOS, there are studies to support RTOS in ROS 1.0 [12]. However, to implement ROS 1.0 based on RTOS in these studies, a compatibility check should be performed, and the porting is complicated. In addition, although ROS 1.0 based on RTOS can be implemented to satisfy the real-time constraints of the software stack, ROS 1.0 does not satisfy the communication real-time constraints due to its communication structure using TCPROS. Therefore, the following sections perform a communication performance evaluation of ROS 1.0 and ROS 2.0, focusing on real-time performance.

C. TASK RESPONSE TIMES

The schedulability of real-time tasks is verified through the response time analysis [18] which is based on the analysis of the worst-case response time. The response time is defined as the duration in which a task starts its execution from a release point until it finishes its job as illustrated in Fig. 4. To perform the response time analysis, we consider four real-time tasks the following attributes:

TABLE 6. Response time analysis of ROS 1.0.

ROS 1.0	Node1		Node2	
Metric (ms)	T_{period}	$T_{response}$	T_{period}	$T_{response}$
avg.	10.000184	4.803624	19.999967	7.803781
max.	100.039176	81.813881	44.000271	36.760369
min.	3.009195	3.000146	5.012950	5.000270
st. d.	11.365553	5.433810	1.466066	3.451887
ROS 1.0	Node3		Node4	
Metric (ms)	T_{period}	$T_{response}$	T_{period}	$T_{response}$
avg.	39.999891	15.041838	79.999020	23.496083
max.	69.106003	68.199445	100.01593	99.134387
min.	13.669573	10.000592	50.646549	15.001000
st. d.	2.491789	5.300730	3.585778	5.950949

Note that we have selected harmonic periods for the tasks and the deadlines are equal to the period. As we are using the POSIX library for both ROS 1.0 and ROS 2.0, task priorities are configured between 1 and 99, with 99 as the highest priority. The worst-case response time (WCRT), R, of each task is calculated by the following equation [18]:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{P_j} \right\rceil \cdot C_j, \quad i = 1, 2, \ldots, n \tag{1}$$

Herein, C is the execution time. B, represents the blocking time or the instance when a low priority task forcefully owns resources that are needed by higher priority tasks. This could occur when the scheduler does not properly exhibit priority-based pre-emption, or if the task includes locking mechanisms that are not released, causing a deadlock. J, denotes the jitter of the task. The WCRT calculation in (1) should be iterated x number of times until $R_i^{x+1} = R_i^x$, or if the result exceeds the task deadline. For the real-time tasks in Table 5, we have calculated the WCRT as 3, 5, 21, and 50 respectively for Node1, Node2, Node3 and Node4. It is important that the maximum acquired response time should be approximately equal to the calculated values to ensure that the system satisfies real-time requirements. Herein, we focus on the environment under stress for the response time analysis to build confidence that the system is viable for practical applications such as multi-agent robot systems.

The experiments were performed for 30 minutes to acquire a decent number of samples for the four real-time tasks (ROS Nodes). The results of the timing analysis for the real-time tasks in ROS 1.0 are shown in Table 6 with the statistical average (avg), maximum (max), minimum (min), and standard deviation (st.d) values of each timing metric. Looking at the results, we could see that all tasks does not satisfy real-time constraints with remarkably high maximum T_{period}. The measured WCRT for all of the tasks are greater than the calculated values. We therefore conclude that ROS 1.0 is not suitable for real-time applications.

Under the same conditions, we investigate the real-time performance of ROS 2.0 and the acquired results are shown in in Table 7.

Unlike ROS 1.0, the response times acquired from the real-time tasks under ROS 2.0 show that all of the real-

TABLE 7. Response time analysis of ROS 2.0.

ROS 1.0	Node1		Node2	
Metric (ms)	T_{period}	$T_{response}$	T_{period}	$T_{response}$
avg.	10.000000	3.000282	20.000015	5.000435
max.	10.084113	3.033663	21.421125	5.038722
min.	9.917142	3.000122	19.914865	5.000197
st. d.	0.010766	0.000987	0.016650	0.001342
ROS 1.0	Node3		Node4	
Metric (ms)	T_{period}	$T_{response}$	T_{period}	$T_{response}$
avg.	40.000031	20.991573	79.999576	49.995179
max.	41.440690	21.034833	80.984228	50.044174
min.	39.912590	10.016334	70.475398	15.000722
st. d.	0.023966	0.054489	0.064237	0.233633

time tasks show periodic behavior with exceedingly small deviation from the expected task periods. Also, the response times were approximately equal to the calculated response times through equation (1). Thus, ROS 2.0 is very suitable for real-time applications at it shows deterministic behavior satisfying periodic and response time requirements of real-time tasks.

III. COMMUNICATION LEVEL EVALUATION

A ROS-based system is implemented as a distributed system in which the nodes, which are a viable process on multiple hardware platforms, run independently and send and receive data. If the real-time constraints of communication are not satisfied in such a distributed system, a malfunction may occur. This malfunction occurs because the data transmission and reception are not performed with the correct timing due to the delay time that occurs in the communication between nodes. Therefore, to provide the communication performance indicators that are necessary for realizing a ROS-based system that satisfies real-time constraints, ROS communication performance that focuses on real-time performance should be evaluated. In this article, the communication performance was evaluated with two metrics, to satisfy the real-time constraints and to verify stable communication.

First, message loss evaluation was performed to provide performance indicators for the message transmission limit that could be communicated without loss in order to prevent system malfunction. Second, latency evaluation evaluated the latency that occurred in the communication between nodes to verify whether the messages could be sent and received with the correct timing. The experiment was performed on a PC and Raspberry Pi 3 (RPI3) and ran for 30 minutes in a 5 GHz wireless network environment.

A. MESSAGE LOSS

Message loss evaluation was performed on a one-way communication method in which a message sent from a PC was received at RPI3. To verify the limits that the system could send and receive messages stably without losing messages, a performance evaluation was performed according to the communication frequency and data size. Table 8 shows the results of the message loss evaluation of ROS 1.0 and ROS 2.0. In both the ROS 1.0 and ROS 2.0 systems, there was no message loss regardless of the communication frequency

TABLE 8. Message loss with varying data size and communication frequency.

ROS 1.0	Data size (bytes)			
Frequency [Hz]	10^1	10^2	10^4	10^{10}
10	0/18000	0/18000	0/18000	0/18000
20	0/36000	0/36000	0/36000	0/36000
100	0/180000	0/180000	0/180000	46164/180000
200	0/360000	0/360000	0/360000	134259/360000
ROS 2.0	Data size (bytes)			
Frequency [Hz]	10^1	10^2	10^4	10^{10}
10	0/18000	0/18000	0/18000	0/18000
20	0/36000	0/36000	0/36000	0/36000
100	0/180000	0/180000	0/180000	179907/180000
200	0/360000	0/360000	0/360000	359859/360000

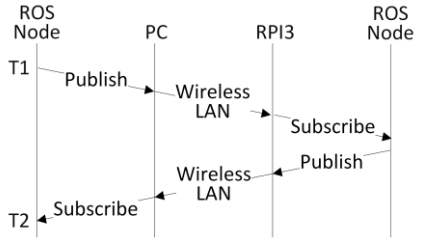

FIGURE 7. Timeline describing the behavior of a ROS node with a given period.

until the data size of the message was 104 bytes. However, the results tested at 105 bytes data size indicate that there were message losses at frequencies of more than 100 Hz, which makes it impossible for stable operation in both systems.

However, when implementing a ROS-based system, a situation can arise in which a node communicating at a frequency of 100 Hz or more must be implemented. Therefore, for the system to be able to communicate stably without losing a message, the size of the message should not exceed 10^4 bytes. In the next section, the communication frequency evaluation performed up to 10^4 bytes, which is the data size that enables stable communication.

B. LATENCY

In this section, we evaluate communication latency on varying communication frequency and data size within the range of no message loss as presented in the previous section. Communication latency was performed for 30 minutes with a round-trip time (PTT) test. Round-trip latency is measured as the time it takes for a message to travel from the PC to RPI3, and from the RPI3 back to the PC. Communication latency is measured as the difference between the time-stamp taken before sending the message (T1) in the PC node and the time-stamp taken just after the reception of the message in the callback of RPI3 node (T2), as shown in Fig. 7. To compare performance according to the environment, the experiment was conducted in an unstable network environment with an idle network environment and network traffic. Network traffic was generated on 30 Mbps using iperf3 [22].

Latency evaluation according to the data size was performed with a frequency of 100 Hz and data size of 10^1, 10^2, 10^3, and 10^4 bytes. Fig. 8 shows the results of a performance evaluation in an idle environment.

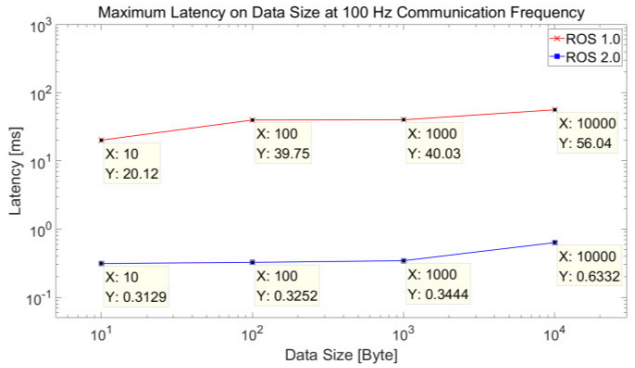

FIGURE 8. Maximum communication latency on varying data size.

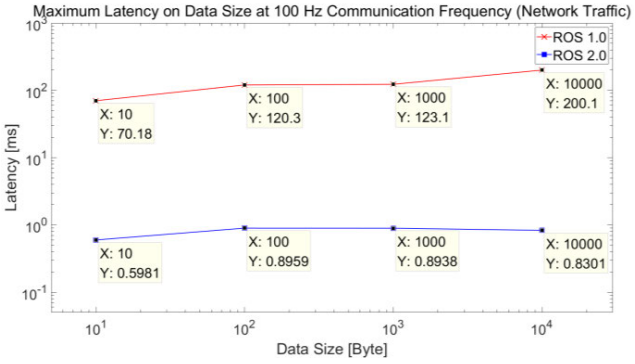

FIGURE 9. Maximum communication latency given the data size in environments with network traffic.

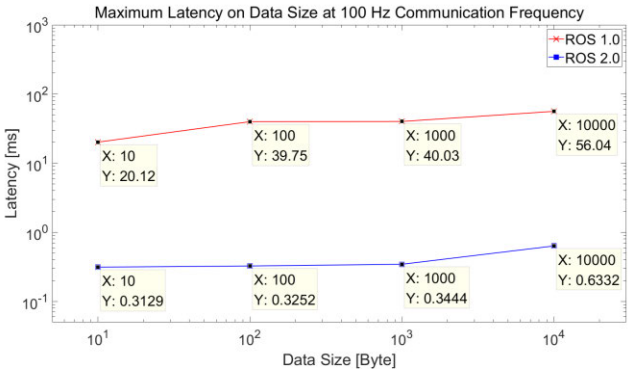

FIGURE 10. Maximum communication latency for varying data sizes.

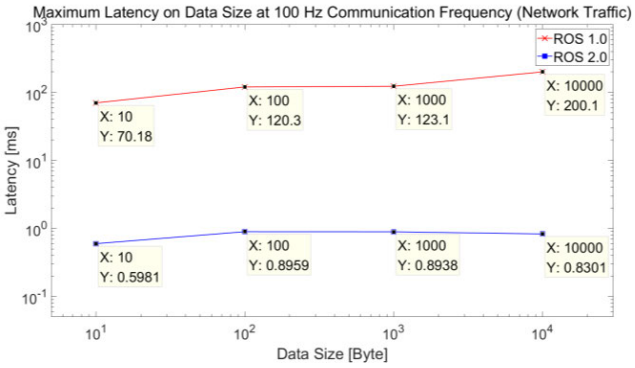

FIGURE 11. Maximum communication latency for varying data sizes in environments with network traffic.

Fig. 9 shows the results in an unstable environment with network traffic. In both the idle and unstable environments, the maximum latency was significantly lower in ROS 2.0 than in ROS 1.0. While ROS 2.0 showed a minor change in the performance in unstable environments, ROS 1.0 had difficulty sending and receiving messages with the correct timing due to latencies. In addition, it did not satisfy real-time communication constraints, which makes it difficult to operate the system in an unstable environment.

The results of the performance evaluation according to the data size showed that ROS 1.0 could not satisfy real-time constraints in the communication layer and had difficulty operating the system in an unstable environment. Since ROS 2.0 has a lower latency, it can satisfy these constraints, and stable operation of the system is possible even in an unstable environment because the network traffic has an insignificant effect on the measured performance.

On the other hand, a latency evaluation with varying communication frequency was performed with a data size of 100 Bytes. The communication frequencies are configured as 10, 20, 100, and 200 Hz. Fig. 10 shows the results of the performance evaluation in an idle environment. Fig. 11 shows the results in an unstable environment where network traffic exists. In an idle environment, ROS 1.0 showed a low latency of 0.29 ms when communicating at frequencies of 10 Hz and 20 Hz but increased to 40 ms above 100 Hz.

In ROS 2.0, the maximum latency increased with increasing frequency, but the increase was not large and maintained a low latency overall. In unstable environments, the maximum latency of ROS 1.0 was 50 ms at 10 Hz and 20 Hz and increased to over 119 ms at 100 Hz. ROS 2.0 maintained a low latency of 0.4 ms - 0.9 ms up to 100 Hz and then increased to 2.7 ms at 200 Hz. As before, both ROS 1.0 and ROS 2.0 had increased maximum latency in unstable environments.

As a result of the performance evaluation according to the communication frequency, ROS 2.0 showed better performance in both the idle and unstable environments than ROS 1.0. Unlike ROS 1.0, which is significantly degraded in an unstable environment, ROS 2.0 showed less performance degradation. In ROS 1.0, the maximum latency was low in the experimental results of 10 and 20 Hz in an idle environment, and thus, the message could be transmitted and received with accurate timing. However, in other experiments, due to the high latency, it was not possible to send and receive messages with the correct timing, which might not satisfy the real-time constraints and could cause problems in the system operation. Since ROS 2.0 had a low maximum latency in all of the experiments, it could send and receive messages with precise timing regardless of the environment, and thus, it was able to satisfy real-time constraints and enable stable system operation.

In the communication latency evaluation, ROS 2.0 showed better performance with a lower maximum latency than ROS 1.0 and showed a greater performance difference in an unstable environment with network traffic. The cause of this result

is the difference between the communication middleware used in ROS 1.0 and ROS 2.0. ROS 1.0 is not suitable for networks with loss, such as wireless networks, because it uses the TCP protocol-based TCPROS as the middleware for its communications [23].

In contrast, ROS 2.0 uses DDS as the communication middleware. DDS uses the UDP-based RTPS protocol for its data transmission and supports various transmission configurations and scalability such as Deadline, Reliability, and Durability by controlling the data flow through QoS policies. ROS 1.0 also supports UDP-based UDPROS, but since it does not support QoS policies, reliable communication is not possible. Therefore, ROS 2.0, which maintains low latency and allows messages to be sent and received with the correct timing, is suitable for implementing a stable communication system even in a lossy network and in an unstable environment with traffic.

IV. MULTIAGENT SERVICE ROBOT

To verify the effectiveness of the performance evaluations conducted in the previous sections, we implemented a ROS-based multiagent service robot that satisfied real-time constraints. A multiagent service robot scenario was written as a robot that provides serving services in a restaurant. The environment consisted of two service robots and six tables. When the user selected the table number and the robot for the destination, the backend executed the navigation stack of the selected robot and generated a movement command in such a way that the selected robot could move to the table and avoid obstacles. When the service robot arrived at the table and completed the serving, the back end navigated the robot back to the kitchen. The user was able to check the current state of the robot in the GUI environment. Since the multiagent service robot operates in a real environment, it must satisfy real-time constraints for safe operation without physical damage caused by a malfunction [8].

A system architecture that meets real-time constraints was constructed based on the results of Sections 2 and 3. In this scenario, two mobile robots utilized two different ROS navigation stacks to provide services. Since the navigation stack consumed a large amount of CPU and memory resources, we refer to the evaluation results in an unstable environment with stress when configuring the system architecture [24], [25].

In the performance evaluation of the software stack, ROS 1.0 did not satisfy the periodicity due to having a large latency in an unstable environment with a system load, and deterministic behavior was impossible. However, ROS 2.0 could satisfy the real-time constraints because it maintains deterministic behavior even in a stressed environment. In the communication performance evaluation, ROS 1.0 was unable to send and receive messages with the correct timing because of the increased latencies in the different load scenarios.

On the other hand, ROS 2.0 was less affected by the message size, communication frequency, and network traffic. It also consistently maintained a low latency, and thus, messages could be sent and received stably. For this reason, a multiagent service robot that met real-time constraints was implemented with ROS 2.0. In this section, we implemented a multiagent service robot that consisted of front-end, back-end, and service robots. It performed a real-time performance evaluation of the implemented system and verified the operation of the system in a real environment.

A. ARCHITECTURE

The architecture of a multiagent service robot consists of the front-end, back-end, and service robots. The front-end connects the user with the back-end in such a way that the user can request a service in the GUI environment. The back-end is responsible for generating control commands for the service robots using the two navigation stacks, which allows the two service robots to perform the requested services. A mobile robot receives a control command from the back-end and operates in a real environment to provide a service. Fig. 12 shows the architecture of the multiagent service robot.

B. FRONT-END SOFTWARE

The front-end software allows users to easily and simply request the functionality provided on the back-end in a GUI environment. The user can request a service through the front-end and check the current operation status of the robot. Front-end platforms that can interact with ROS have Android [26], [27] and web browsers [28], [29]. If implemented to operate on the Android platform, the use is limited because it can only operate on devices with Android OS. Implementing in a web browser is highly accessible because it does not depend on specific software platforms, such as the Android OS.

For this reason, the front-end uses a web browser to provide a GUI environment to the user. For the communication between the front-end using a web browser and the back-end based on ROS 2.0, the ros2-web-bridge [30], which can act as a bridge between the web interface and the ROS 2.0 interface, is applied to the back-end. Accordingly, the front-end includes the roslibjs library, a JavaScript library that provides an API for communicating with the ros2-web-bridge through a web interface.

Based on the roslibjs library, various libraries for interacting with ROS 2.0, such as the ros2djs library for 2D visualization and ros3djs library for 3D visualization, will operate. A user request through the browser is sent to the web interface as a JSON object and to the ros2-web-bridge using TCP/IP. Data received from ros2-web-bridge is forwarded to the ROS 2.0 nodes using ROS 2.0 messages. The communication structure using the ros2-web-bridge is shown in Fig. 13. The front-end GUI has six table numbers and two service robots selection buttons that allow the user to select the tables and robots that they want to have, and a start button to start serving after selecting the table numbers and robots. Under each robot selection button, there is a view to check the operation status of the current robot.

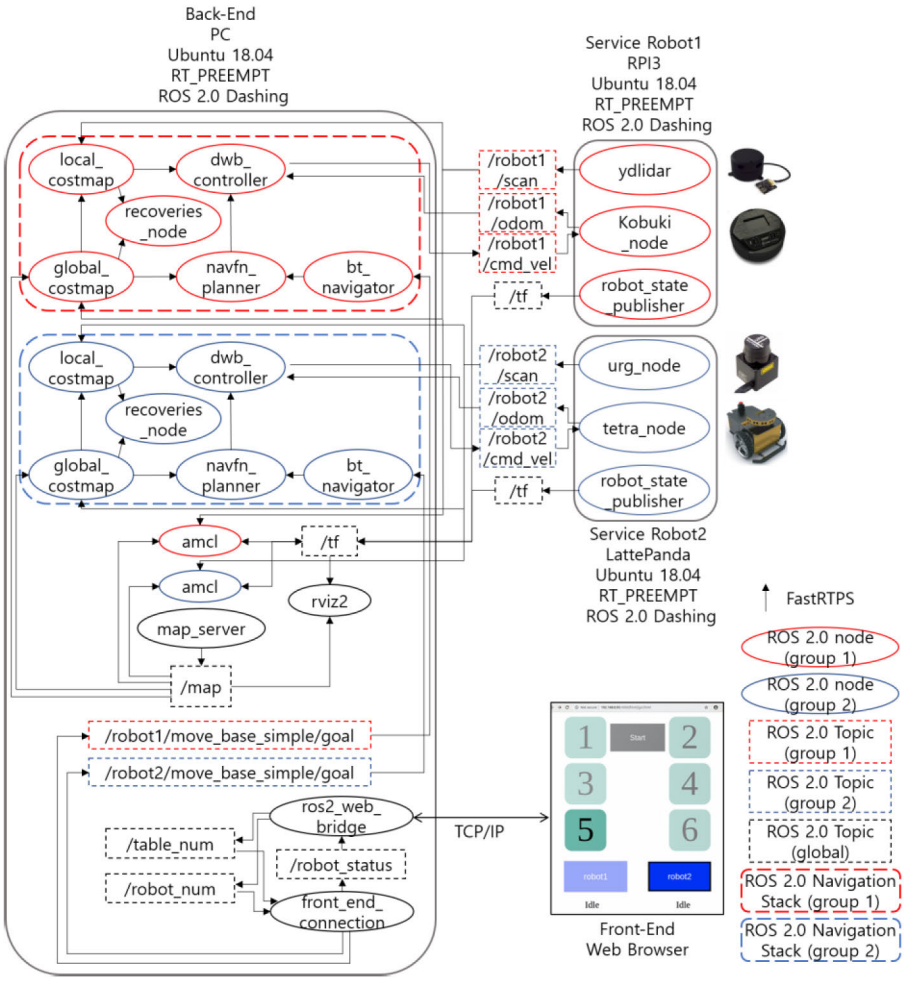

FIGURE 12. Multiagent service robot architecture.

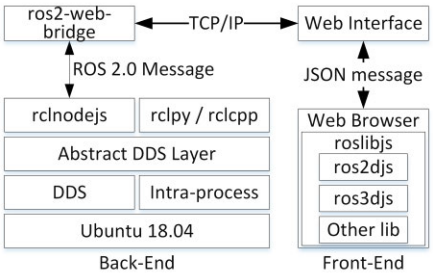

FIGURE 13. Communication interface between the front-end and back-end software.

C. BACK-END SOFTWARE

The goal of the back-end is to generate a two mobile robot command to provide the requested service from the front-end. To reliably generate control commands for the mobile robot, the real-time constraints must be satisfied in the back-end. If the real-time constraints are not satisfied, a latency occurs in the control command of the mobile robot, which must be generated periodically. Since the mobile robot operates based on a delayed control signal, this aspect could cause physical damage due to malfunction. Therefore, to satisfy the real-time constraints, the back-end software was implemented based on the real-time Linux kernel with a PREEMPT_RT patch and ROS 2.0. The kernel used 4.18.16-rt9, the file system Ubuntu 18.04, and the ROS 2.0 Dashing. Fig. 1b shows the back-end software architecture.

In the back-end, the control commands of the mobile robot were generated in the ROS 2.0 navigation stack. The following nodes were used to implement the navigation stack in ROS 2.0. The map_server published the map information to be used in the navigation using the existing map. Here, amcl was used to estimate the current position of the robot on a map issued by map_server; map_server and amcl are optional for implementing the navigation stack, but in multiagent system scenarios, there is no terrain change. Thus, in the scenario before the running navigation, we created a map of the environment and used a map created in advance.

The dwb_controller corresponds to the local_planner of ROS 1.0 and was used to regenerate a path to avoid obstacles around the robot. Here, local_costmap is a map used by dwb_controller to search for obstacles that are detected dynamically around the robot. Additionally, navfn_planner

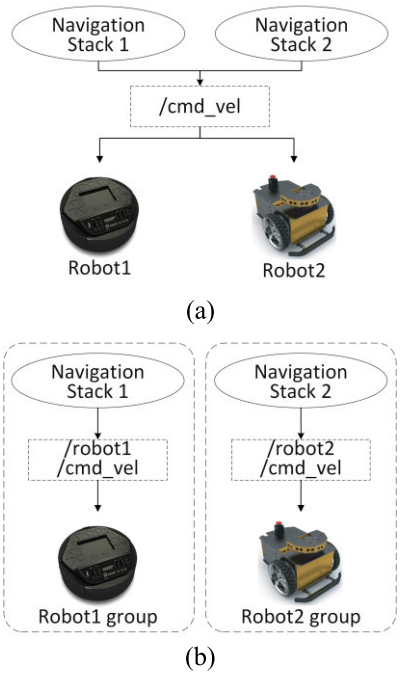

FIGURE 14. Node grouping. (a) before (b) after.

corresponds to the global_planner of ROS 1.0, and it generates a path from the start point of the robot to the destination. In addition, global_costmap is a map used by navfn_planner to create a route to a destination by avoiding static obstacles, and bt_navigator subscribes to the/move_base_simple/goal topic, which is the location information for the destination to be navigated to, and it runs navfn_planner to move the robot to the specified destination.

The back-end software is responsible for the navigation of two mobile robots. For the two mobile robots to navigate, two nodes that implement the navigation stack must be used. Due to the inability of the ROS 2.0 navigation stack to consider multiple robots, there were two problems in the implementation of two navigation stacks in this scenario. The first problem was that the topic and robot frame transformation (tf) information was duplicated when the same node was executed several times. When the topics and tf overlap, the navigation stack cannot know which robot and sensor it should receive data from, and the robot cannot know which navigation stack to receive control commands from. Therefore, the system's operation is impossible. To solve this problem, grouping by robot was performed on topics, and tf was used in the robot and navigation stack. The grouping of topics was solved by assigning namespaces to nodes, and the grouping of tf was solved by adding prefixes to tf, as shown in the comparison in Fig. 14.

D. MOBILE ROBOTS

The service robot provides services to users in a real environment, where the service robots must satisfy real-time constraints for accurate and safe navigation. If the real-time constraints are not met, a malfunction caused by the system latency could prevent the normal execution of the driving commands received from the back-end software. This circumstance can lead to situations in which the desired position is not reached, and physical damage can occur. As shown in Table 9, we have employed two service robots, namely, Kobuki2 and Tetra DS-IV. To drive the robots, we have developed ROS 2.0 driver nodes for each of the robots. These nodes are responsible for receiving the velocity commands from the back-end software and actuating the motors attached on each service robot. The mainboard for driving the Kobuki2 and Tetra DS-IV is Raspberry Pi 3 B+ and LattePanda, respectively.

TABLE 9. Specifications of the mobile robots.

Hardware		Specifications	
Mobile Robot	Model	Kobuki2	Tetra DS-IV
	External Size	L354 x W354 x H350 mm	L473 x W430 x H262 mm
	Payload	5 kg	80 kg
	Drive	Two-Wheel Differential Drive	Two-Wheel Differential Drive
	Speed	Max 0.7 m/s	Max 2.0 m/s
	Axle Track	250 mm	380 mm
	Wheel Diameter	70 mm	240 mm
Laser sensor	Model	YDLIDAR G4	URG-04LX
	Scan Angle	360°	240°
	Angular Resolution	0.26	0.35
	Scan frequency	5-12 Hz	10 Hz
Main board	Single Board Computer	Raspberry Pi 3 B+ Broadcom BCM2837(1.4 GHz)	LattePanda Intel Atom x5-Z8350(1.8 GHz)
	Ram	1 GB LPDDR2	4 GB DDR3L
	Operating System	Ubuntu 18.04, Kernel 4.19.85-rt30(RT Preempt)	Ubuntu 18.04, Kernel 4.18.16-rt9(RT Preempt)

E. EXPERIMENT

In this experiment, we implemented and drove a multiagent service robot based on ROS 2.0 in a real environment. The experimental environment consisted of a 3.2 × 8.0 m space. To be consistent with the results of the performance evaluated in Section 3, the QoS policies of the DDS were configured to keep the latest history data with a depth of 10.

The reliability and durability were set to reliable and volatile, respectively. The History and Depth organized topic publishing records were to keep records for only the last 10 topics published. Reliability ensured that the topics were transmitted without being lost, and Durability was configured to not maintain samples for nodes that participated in late communication.

F. EXPERIMENTAL ENVIRONMENT MAPPING

To perform navigation based on the generated map, SLAM was used to create a map of the experimental environment. The cartographer package was used for the SLAM [31], [32], and the mobile robot used Kobuki2 to create a map while

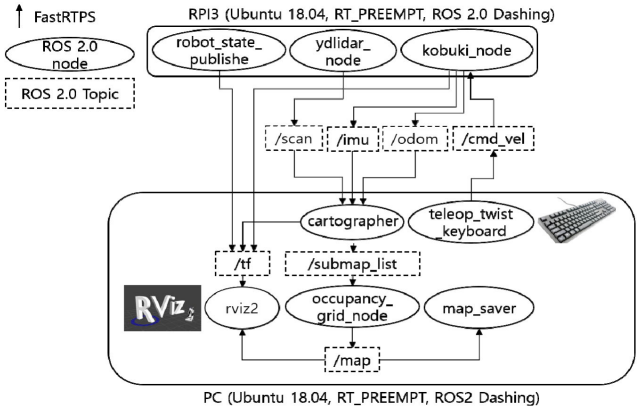

FIGURE 15. Software architecture for mapping using cartographer.

FIGURE 16. Running a multiagent service robot.

driving through keyboard input. The software architecture used to create the map is shown in Fig. 15.

G. OPERATION TEST

Based on the map generated by SLAM, we tested the multiagent service robot that provided the serving service. Fig. 16 shows a running multiagent system. Two service robots were requested to provide serving services through the front-end. Robot1 (Kobuki2) provided service to table 3, and robot2 (Tetra DS-IV) provided service to table 6.

The current driving status of each robot could be checked at the front-end. In Rviz2 on the back-end, the marker indicates the route and direction of travel to the destination of the service robot. The red marker represents robot1, and the blue marker represents robot2. Robot1 is in the process of returning to the kitchen after arriving at table 4, and robot2 shows the situation just before arriving at table 1, the destination.

H. PERFORMANCE EVALUATION

To verify that the multiagent service robot implemented based on ROS 2.0 is more stable than the existing ROS 1.0-based system, a performance evaluation on the mobile robot operating in an actual environment was performed. The periodicity was evaluated to verify that a deterministic operation was possible on the Kobuki2 node and the Tetra DS-IV node running the mobile robot. To confirm the effect of the periodicity

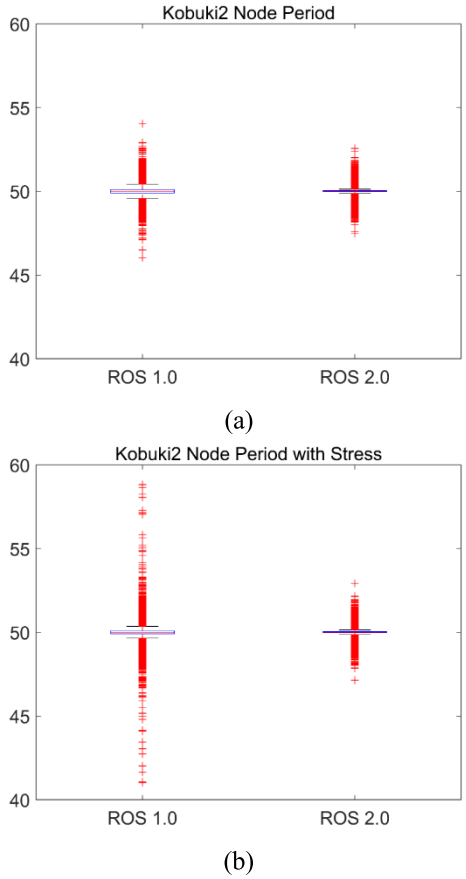

FIGURE 17. Periodicity of Kobuki2 node. (a) Idle environment (b) unstable environment with stress.

TABLE 10. Statistical analysis of periodicity of Kobuki2 node.

	Idle, Jitter[ms]		Stress, Jitter[ms]	
	ROS1.0	ROS2.0	ROS1.0	ROS2.0
avg.	0.2424	0.1425	0.2452	0.1439
max.	4.0295	2.5651	8.9721	2.9256
min.	0.0000	0.0000	0.0000	0.0000
st. d.	0.3282	0.2744	0.5124	0.016595

evaluation result on the driving of the mobile robot, an actual experiment was conducted, to follow a Bezier curve [8] in a real environment.

The real-time task responsible for controlling the robot was running periodically every 50 ms with the highest priority in both the idle and stressed environments. Table 10 shows the result of the jitter (time difference between the actual period and the expected period) and reveals that ROS 2.0 meets real-time constraints better than ROS 1.0. In ROS 1.0, the maximum jitter is 4.0295 ms in the idle environment, which is doubled when the system is under stress. Moreover, the standard deviation (st. d.) also increased, which means that the system is not deterministic. In ROS 2.0, the maximum latency is 2.5651 ms in the idle environment and 2.9256 ms in the environment with stress. The st. d. also showed superior performance in comparison to ROS 1.0. Fig. 17 displays the periodicity of the Kobuki2 node running in RPI3.

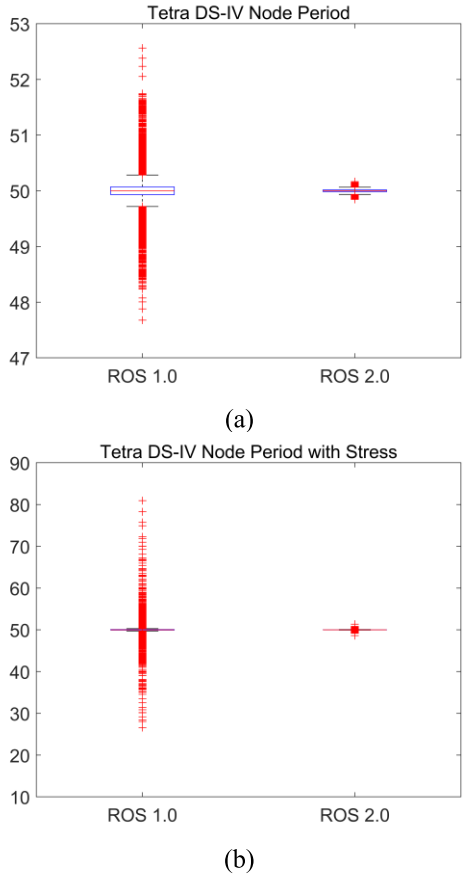

FIGURE 18. Periodicity of the Tetra DS-IV node. (a) Idle environment (b) unstable environment with stress.

TABLE 11. Statistical analysis of periodicity of tetra DS-IV node.

	Idle, Jitter[ms]		Stress, Jitter[ms]	
	ROS1.0	ROS2.0	ROS1.0	ROS2.0
avg.	0.1410	0.0199	0.3031	0.0253
max.	2.5642	0.1662	57.482	1.3873
min.	0.0000	0.0000	0.0000	0.0000
st. d.	0.1940	0.0174	1.0360	0.0343

Fig. 18 shows the results of evaluating the periodicity of the Tetra DS-IV node running on LattePanda in boxplot. Table 11 shows the results of a timing analysis. It can be seen that ROS 2.0 has better real-time performance than ROS 1.0. As a result, the maximum latency is increased to a value larger than the set period, indicating that the system is not operating stably. The maximum latency of ROS 2.0 is 0.1662 ms in the idle environment and 1.3873 ms in the environment with stress. Through this, it can be seen that the latency increase due to stress is low and is less than 1 ms.

To confirm the difference in the driving results of the mobile robots according to the real-time performance, the path planning method in reference [8] was used to create a Bezier curve path. Fig. 19 shows the difference in the driving path in an unstable environment with stress. The mobile robot traveled from the starting point (0, 0, 0°) to the target point (2, 3, 0°) at a speed of 0.5 m/s and an acceleration of 0.5 m/s2. ROS 1.0 did not meet real-time constraints and could not execute speed commands with the correct timing. As a result,

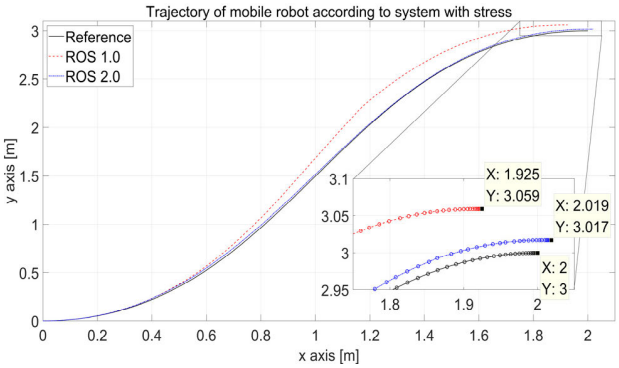

FIGURE 19. Bezier curve trajectory tracking in an unstable environment with stress.

the mobile robot moved off the reference path. However, ROS 2.0 was driven without deviating from the reference path because it could meet real-time constraints.

In the real-time performance evaluation of the Tetra DS-IV node and Kobuki2 node, there was a difference in the real-time performance according to the motherboard and software structure, but in both experiments, ROS 2.0 showed better performance than ROS 1.0. In particular, there was no change in the performance of ROS 2.0 under stress and in an unstable environment, but it was confirmed that ROS 1.0 underperformed significantly. Due to this performance difference, ROS 1.0 could not follow the exact route in the driving experiment of the mobile robot, but ROS 2.0 was able to drive the correct route. Therefore, it could be seen that a ROS 2.0-based system could be operated stably by satisfying real-time constraints without a notable change in the performance even in an unstable environment.

V. CONCLUSION AND DISCUSSION

This article provides two contributions for the development of a ROS-based real-time system. First, it furnishes the performance evaluation results necessary to implement a ROS-based system that meets real-time constraints. Second, it presents a multiagent service robot based on the ROS 2.0-based system architecture that satisfies real-time constraints. The system that shares the workspace with the user in real-time could cause physical damage to the user when a malfunction occurs due to latency, and thus, the real-time constraints must be satisfied for stable operations. To verify that the ROS-based system satisfies real-time constraints and enables stable operation, a performance evaluation was performed on software stacks and communications with a focus on real-time performance.

We evaluated whether the ROS application satisfies the real-time performance. In order to support real-time performance in ROS applications, real-time performance must be supported in the OS on which ROS is implemented. ROS 1.0 does not support RTOS, but ROS 2.0 does support RTOS. Because of these differences, applications based on ROS 2.0 can satisfy real-time performance, but those on ROS 1.0 do not. In order to verify the real-time performance, performance evaluation of general Linux and RT_PREEMPT

with ROS 1.0 and ROS 2.0 were performed. From this result, it was verified that the RT_PREEMPT-based system can operate with low latency.

In order to verify real-time performance in multi-node applications implemented with ROS 1.0 and ROS 2.0, in an environment where various nodes are running. We evaluated whether preemption and periodicity were well satisfied. Through this result, it was verified that the node satisfies real-time performance based on priority in ROS 2.0. On the other hand, in ROS 1.0, this was not satisfied.

By comparing ROS 1.0 and ROS 2.0, we verified that the software architecture was suitable for ROS-based real-time system development. To verify the effectiveness of the performance evaluation, we implemented a multiagent service robot based on ROS 2.0, which is a useful example of a real-time system. A multiagent service robot was implemented based on a scenario in which two robots provide a serving service that consisted of front-end, back-end, and two service robots. To implement the scenario, two robots must perform navigation. Currently, ROS 2.0 does not support more than two robots in multirobot navigation, and therefore, to operate the two navigation stacks independently, we grouped tf and the nodes and modified the navigation stack's global planner.

To verify whether the multiagent service robot satisfied the real-time constraints, a periodicity evaluation of the node that controls the service robot was performed in ROS 1.0 and ROS 2.0. To verify the results of the performance evaluation, we have implemented ROS 2.0 on a multiagent robot system and conducted experiments in a real environment. Through this approach, we verified that the multiagent service robot based on ROS 2.0 could satisfy the real-time constraints. This article will be promising for engineers who want to develop a stable system that meets real-time constraints based on ROS. The performance evaluation conducted in various environments and conditions as well as the implementation and development of a ROS 2.0-based system satisfying real-time constraints are both helpful in realizing this system.

REFERENCES

[1] C. Jayawardena, I. H. Kuo, U. Unger, A. Igic, R. Wong, C. I. Watson, R. Q. Stafford, E. Broadbent, P. Tiwari, J. Warren, J. Sohn, and B. A. MacDonald, "Deployment of a service robot to help older people," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Oct. 2010, pp. 5990–5995.

[2] I. Osunmakinde and R. Vikash, "Development of a survivable cloud multi-robot framework for heterogeneous environments," *Int. J. Adv. Robotic Syst.*, vol. 11, no. 10, p. 164, Oct. 2014.

[3] R. Doriya, P. Chakraborty, and G. C. Nandi, "'Robot-Cloud': A framework to assist heterogeneous low cost robots," in *Proc. Int. Conf. Commun., Inf. Comput. Technol. (ICCICT)*, Oct. 2012, pp. 1–5.

[4] 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 Manuf.*, vol. 38, pp. 349–357, 2019.

[5] S. Zaman, W. Slany, and G. Steinbauer, "ROS-based mapping, localization and autonomous navigation using a pioneer 3-DX robot and their relevant issues," in *Proc. Saudi Int. Electron., Commun. Photon. Conf. (SIECPC)*, Apr. 2011, pp. 1–5.

[6] R. Reid, A. Cann, C. Meiklejohn, L. Poli, A. Boeing, and T. Braunl, "Cooperative multi-robot navigation, exploration, mapping and object detection with ROS," in *Proc. IEEE Intell. Vehicles Symp. (IV)*, Jun. 2013, pp. 1083–1088.

[7] M. Á. Muñoz–Bañón, I. del Pino, F. A. Candelas, and F. Torres, "Framework for fast experimental testing of autonomous navigation algorithms," *Appl. Sci.*, vol. 9, no. 10, p. 1997, May 2019.

[8] G. J. Yang, R. Delgado, and B. W. Choi, "A practical joint-space trajectory generation method based on convolution in real-time control," *Int. J. Adv. Robotic Syst.*, vol. 13, no. 2, p. 56, Mar. 2016.

[9] P. Bouchier, "Embedded ROS [ROS Topics]," *IEEE Robot. Autom. Mag.*, vol. 20, no. 2, pp. 17–19, Jun. 2013.

[10] H. Wei, Z. Shao, Z. Huang, R. Chen, Y. Guan, J. Tan, and Z. Shao, "RT-ROS: A real-time ROS architecture on multi-core processors," *Future Gener. Comput. Syst.*, vol. 56, pp. 171–178, Mar. 2016.

[11] R. Delgado, B.-J. You, M. Han, and B. W. Choi, "Integration of ROS and RT tasks using message pipe mechanism on xenomai for telepresence robot," *Electron. Lett.*, vol. 55, no. 3, pp. 127–128, Feb. 2019.

[12] R. Delgado, B.-J. You, and B. W. Choi, "Real-time control architecture based on xenomai using ROS packages for a service robot," *J. Syst. Softw.*, vol. 151, pp. 8–19, May 2019.

[13] P. Bellavista, A. Corradi, L. Foschini, and A. Pernafini, "Data distribution service (DDS): A performance comparison of OpenSplice and RTI implementations," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jul. 2013, pp. 000377–000383.

[14] G. Xylomenos and G. C. Polyzos, "TCP and UDP performance over a wireless LAN," in *Proc. IEEE Conf. Comput. Commun. 18th Annu. Joint Conf. IEEE Comput. Commun. Soc. Future Now (INFOCOM)*, vol. 2, 1999, pp. 439–446.

[15] Z. Chen, "Performance analysis of ROS 2 networks using variable quality of service and security constraints for autonomous systems," M.S. thesis, Naval Postgraduate School, Monterey, CA, USA, 2019.

[16] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *Proc. 13th Int. Conf. Embedded Softw. (EMSOFT)*, Pittsburgh, PA, USA, 2016, pp. 1–10.

[17] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches, "Towards a distributed and real-time framework for robots: Evaluation of ROS 2.0 communications for real-time robotic applications," Tech. Rep.

[18] R. Delgado, J. Park, and B. Choi, "Open embedded real-time controllers for industrial distributed control systems," *Electronics*, vol. 8, no. 2, p. 223, Feb. 2019.

[19] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on PREEMPT_RT," *ACM Comput. Surv.*, vol. 52, pp. 1–36, Feb. 2019.

[20] (Nov. 11, 2019). *Stress-Ng, a Stress-Testing Swiss Army Knife*. [Online]. Available: https://eLinux.org/images/5/5c/Lyon-stress-ng-presentation-oct-2019.pdf

[21] F. Cerqueira and B. Brandenburg, "A comparison of scheduling latency in Linux, PREEMPT-RT, and LITMUS RT," in *Proc. 9th Annu. Workshop Operating Syst. Platforms Embedded Real-Time Appl.*, 2013, pp. 19–29.

[22] O. Vondrouš, P. Macejko, and Z. Kocur, "Flowping-the new tool for throughput and stress testing," *Adv. Electr. Electron. Eng.*, vol. 13, no. 5, p. 516, 2015.

[23] D. Tardioli, R. Parasuraman, and P. Ögren, "Pound: A multi-master ROS node for reducing delay and jitter in wireless multi-robot networks," *Robot. Auto. Syst.*, vol. 111, pp. 73–87, Jan. 2019.

[24] M. T. Lázaro, G. Grisetti, L. Iocchi, J. P. Fentanes, and M. Hanheide, "A lightweight navigation system for mobile robots," in *Proc. 3rd Iberian Robot. Conf.*, A. Ollero, A. Sanfeliu, L. Montano, N. Lau, and C. Cardeira, Eds. Cham, Switzerland: Springer, vol. 2018, pp. 295–306.

[25] R. L. Guimarães, A. S. de Oliveira, J. A. Fabro, T. Becker, and V. A. Brenner, "ROS navigation: Concepts and tutorial," in *Robot Operating System (ROS)*, vol. 1, A. Koubaa, Ed. Cham, Switzerland: Springer, 2016, pp. 121–160.

[26] J. P. de A. Barbosa, F. do P. de C. Lima, L. dos S. Coutinho, J. P. R. Rodrigues Leite, J. Barbosa Machado, C. Henrique Valerio, and G. Sousa Bastos, "ROS, Android and cloud robotics: How to make a powerful low cost robot," in *Proc. Int. Conf. Adv. Robot. (ICAR)*, Jul. 2015, pp. 158–163.

[27] Z. Shao, M. Li, Y. Qu, G. Song, Y. Guan, J. Tan, and H. Wei, "Reliable communication mechanism design for interaction between Android and ROS," in *Proc. IEEE 8th Annu. Int. Conf. CYBER Technol. Autom., Control, Intell. Syst. (CYBER)*, Jul. 2018, pp. 1496–1501.

[28] R. Toris, J. Kammerl, D. V. Lu, J. Lee, O. C. Jenkins, S. Osentoski, M. Wills, and S. Chernova, "Robot Web tools: Efficient messaging for cloud robotics," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sep. 2015, pp. 4530–4537.

[29] S. Osentoski, G. Jay, C. Crick, B. Pitzer, C. DuHadway, and O. C. Jenkins, "Robots as Web services: Reproducible experimentation and application development using rosjs," in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2011, pp. 6078–6083.
[30] *Node.Js* Client & Web Bridge Ready for ROS* 2.0*. Accessed: Nov. 26, 2019. [Online]. Available: https://roscon.ros.org/2018/presentations/ROSCon2018_nodejs_client.pdf
[31] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2D LIDAR SLAM," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2016, pp. 1271–1278.
[32] J. Luis Blanco-Claraco, "A modular optimization framework for localization and mapping," in *Proc. Robot., Sci. Syst. XV*, Freiburg im Breisgau, Germany, Jun. 2019, pp. 22–26.

JAEHO PARK received the B.S. degree in electronics, information, and communication engineering from Daejeon University, in 2018, and the M.S. degree in electrical and information engineering from the Seoul National University of Science and Technology, in 2020. He is currently working at Onpoom Company, Ltd. His research interests include embedded systems, distributed control, and service robotics.

RAIMARIUS DELGADO (Member, IEEE) received the B.S. and M.S. degrees in electrical and information engineering from the Seoul National University of Science and Technology, Seoul, South Korea, in 2014 and 2016, respectively, where he is currently pursuing the Ph.D. degree under the supervision of Prof. Byoung Wook Choi. His research interests include real-time systems, industrial control and automation, embedded systems, software architecture, and service robotics.

BYOUNG WOOK CHOI (Member, IEEE) received the M.S. and Ph.D. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Seoul, South Korea, in 1988 and 1992, respectively. He is currently a Professor with the Department of Electrical and Information Engineering, Seoul National University of Science and Technology. Previously, he was a Principal Research Engineer with LG Industrial Systems, from 1992–2000, and a Professor with Sun Moon University, from 2000–2005. He was the CEO of Embedded Web Company Ltd., from 2001–2003, and a Senior Fellow at Nanyang Technological University, Singapore, from 2007–2008. He has published textbooks on Embedded Linux. His current research interests include real-time systems design, embedded systems, and intelligent robot software.