2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications

# Priority and Synchronization Support for ROS

Yukihiro Saito*, Takuya Azumi†, Shinpei Kato‡, and Nobuhiko Nishio*

* Graduate School of Information Science and Engineering, Ritsumeikan University
† Graduate School of Engineering Science, Osaka University
‡ Graduate School of Information Science and Technology, The University of Tokyo

***Abstract*—Robot Operating System (ROS) is a software component framework developed for robot applications. It provides a rich set of software libraries and tools to construct robot components, including implementation of many popular perception, planning, and control algorithms. The state of the art in ROS, however, does not fundamentally support priority and synchronization among tasks, so-called *nodes* in ROS. Therefore, ROS may not be appropriate as a platform of real-time multi-tasking environments, despite the fact that most robot applications are multi-tasking and running under real-time constraints. In this paper, we explore real-time performance of ROS. We present a priority-based message transmission mechanism to reduce the execution time and time variance of high-priority ROS nodes. We also present a synchronization mechanism to harmonize multiple ROS nodes running at different frequencies. The presented mechanisms can be both used with legacy ROS applications without modification. Experiments using an autonomous driving system show a 62.3% reduction in a worst-case execution time and indicate that time differences between nodes are guaranteed below a constant time. In addition, the total CPU utilization is saved by up to about 10%.**

***Index Terms*—Priority-driven scheduling; Synchronization; ROS;**

## I. Introduction

In recent years, robot applications have been emerging in the market. An example of the most argued robot applications is an autonomous vehicle. Largely due to the advent of the Google Car and the DARPA Urban Challenge [1], [2], the autonomous driving technology is becoming more and more public, while automotive makers have been developing relevant safety technologies for a long time.

One of the most strong tools in developing early-stage robot applications is Robot Operating System (ROS) [3], [4]. ROS supports component-based development of robot applications, using a lot of existing software libraries, such as OpenCV [5], PCL [6], and OpenNI [7]. For example, ROS has been widely used in the cutting-edge research and development of autonomous driving systems [8]–[14].

A ROS application is composed of a number of independent programs called nodes, each of which communicates with other nodes using a topic-based Publish/Subscribe design pattern [15]. To communicate between nodes, a publisher node sends a message by publishing the corresponding data it to a given topic, while a subscriber node that is interested in the data will subscribe to appropriate topics.

Note that many robot applications have real-time constraints where some specific tasks must be completed before their given deadlines. For example, with pedestrian detection applications used in autonomous driving systems, if the process of detecting pedestrians is not performed by its deadline, the system cannot eventually recognize pedestrians. Unfortunately, ROS does not support such real-time performance issues.

Hongxing et al. [16] and Jonas [17] have studied real-time performance of ROS. Hongxing et al. improved real-time performance of ROS by running ROS nodes on a real-time OS (RTOS). Jonas evaluated real-time performance through real-time ROS tests; these evaluations were insufficient because the tests primarily involved the ROS timer and did not include internal process tests. These prior work did not focus on the inside of ROS, and the solutions were provided from the outside of ROS.

We consider Autoware [14] as a specific example of ROS applications that require real-time performance, since we have experienced several problems with regard to real-time performance. To support real-time performance in ROS, we could use an existing synchronization system [18] that matches messages for a set of topics. However, this system is not able to enforce a constant time difference between data timestamps. To the best of our knowledge, there does not exist an efficient approach to real-time support for ROS.

**Contribution:** This paper presents a priority-based message transmission mechanism and a synchronization mechanism for ROS applications. The presented priority mechanism can decrease the worst-case execution times (WCET) and reduce time variance of high-priority ROS nodes. The presented synchronization system, on the other hand, can ensure that time differences between related ROS nodes are suppressed to less than a constant time and, as a result, it can reduce the total CPU utilization. In addition, these presented mechanisms can be used with legacy ROS applications without modification.

**Organization:** The rest of this paper is organized as follows. Section II discusses the system model and assumptions made in this paper. Section III evaluates the real-time performance of native ROS. Section IV presents our real-time extension to ROS, i.e., the presented priority-based message transmission and synchronization system. Section V describes evaluations of the presented methods, and Section VI discusses related work. Concluding remarks are provided in Section VII.

## II. System Model

ROS has been specifically developed for the PR2 robot [19]. Consequently, it does not provide real-time constraints such

978-1-5090-4403-0/16 $31.00 © 2016 IEEE
DOI 10.1109/CPSNA.2016.24

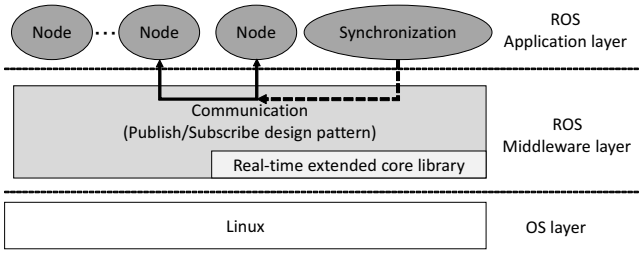


Fig. 1. Presented real-time extension of ROS.

as a priority-based mechanism provided by an OS. However, real-time tasks are required in many robot applications. Thus, we present a real-time extension to ROS, i.e., the real-time extended core library and synchronization (Fig. 1).

The real-time extended core library supports the priorities for real-time constraints. It is important that the priorities provided by the OS should be guaranteed because they affect system performance and reliability, the extended core library supports such priorities. This extended library reduces node overhead and variance caused by ROS middleware. In addition, existing node programs do not need to be modified to employ the presented library because nodes can use the same interfaces for communication.

Furthermore, the presented synchronization system provides a function to synchronize between nodes for real-time constraints. Each node runs with any period; thus, it cannot synchronize between nodes. Nodes in robots and autonomous driving systems often require synchronization. The presented system works as a node on middleware because existing node programs do not need to change interfaces to synchronize.

## III. Real-Time Performance of Native ROS

We evaluate the real-time performance of native ROS. We demonstrate problems related to real-time constraints and examine the factors that cause these problems.

### A. Preliminary Evaluation Items

The main purpose of this evaluation is two-fold, and we consider evaluation items related to ROS real-time constraints.

First, in the OS with a priority-based mechanism, it is necessary to consider priority to guarantee real-time constraints for the reason that priorities affects system performance and reliability. Although a process to be executed in ROS can have priorities provided by the OS, such processes are not always processed properly relative to the priorities of ROS functions. ROS functions affect priorities when multiple nodes subscribe to the same topic. A publisher sends topic data to each subscriber sequentially using sockets. Following this, if the publisher does not send the topic data in the order of priority, the time at which a subscriber application function is launched cannot be known by developers. Consequently, the real-time performance is reduced (we evaluate this situation in Section III-C).

Second, there are many cases where a node combines multiple topics in a system that uses ROS. For example, the car detection system in Autoware [14] detects cars using a sensor fusion method with a camera and LiDAR as shown in

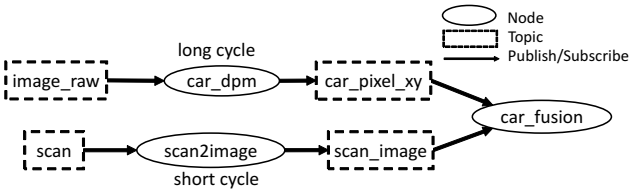


Fig. 2. Car detection system.

TABLE I
Environmental setup of evaluations

| | | |
|---|---|---|
| CPU | Model number | Intel Core i7-2600 |
| | Frequency | 3.40 GHz |
| | Cores | 4 |
| Memory | Capacity | 8GB(DDR3 SDRAM) |
| ROS version | | Indigo |
| OS | | Ubuntu 14.04 64bit |
| Kernel version | | 3.13 |

Fig. 2. car_dpm receives image data and recognizes cars after image processing. It then publishes the results of recognized cars to the car_pixel_xy topic. Following this, scan2image receives point cloud data from the scan topic and transforms the data from LiDAR coordinates to image coordinates. It then publishes the transformed result to the scan_image topic. The scan2image processing time is less than the car_dpm processing time because scan2image calculation cost is lower than the car_dpm calculation cost. car_fusion estimates the distance to each recognized car using the car_pixel_xy and scan_image topics. A gap in the timestamps (time difference) of sensors, such as the camera and LiDAR, occurs when the car_fusion node receives topic data from multiple nodes because the periods for car_dpm and scan2image differ. There should be no significant difference between the timestamps of sensors for nodes that subscribe to and combine some topics, such as the car_fusion node. If there is a large time difference in between, car_fusion cannot estimate the distance and recognize cars correctly; car_fusion may incorrectly recognize a nearby car as a distant car. Therefore, a large time difference causes traffic accidents.

### B. Evaluation Environment

The hardware and software environments for our evaluations are listed in Table I. The mainline Linux kernel implements two real-time scheduling policies [20], i.e., FIFO (*SCHED_FIFO*) and Round-Robin (*SCHED_RR*). These are preemptive and higher level than the normal scheduling queue. In our evaluations, we used *SCHED_FIFO*, and the virtual address space was locked into physical RAM at all times by the *mlockall* system call.

### C. Priority Supports in ROS

As mentioned in Section III-A, priorities must be appropriately reflected in ROS because developers cannot estimate processing times. The system shown in Fig. 3 is used to evaluate priority supports. talker publishes a topic (data size 512 KB) in a 200 msec period. listener1, listener2, and listener3 subscribe to the topic from the talker with priorities of 97, 98, and 99, respectively. These priorities are provided by the OS for real-time scheduling. Note that we limited the

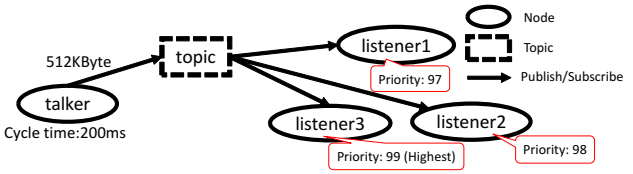


Fig. 3. Preliminary evaluation system for priorities provided by the OS.

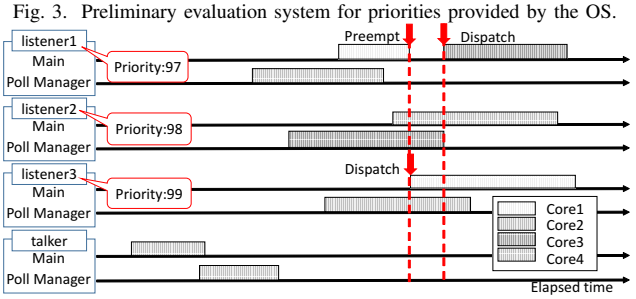


Fig. 4. System trace (ftrace) of the system shown in (Fig. 3).

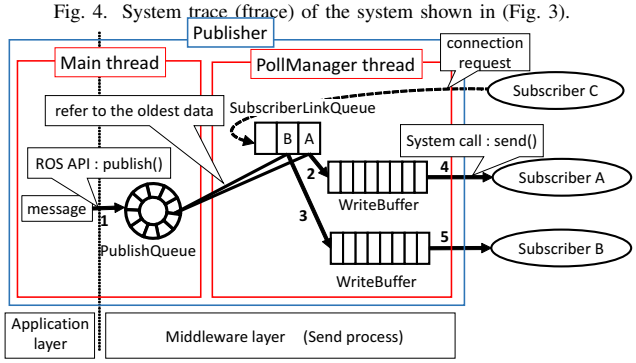


Fig. 5. ROS publish architecture.

number of cores that these nodes can use to four for this evaluation. *ftrace* [21], [22], which is a Linux kernel trace function, is used in this evaluation.

The evaluation using *ftrace* is shown in Fig. 4. Sometimes, listener1 (lowest priority) received the topic data from talker before listener3 (highest priority). This reception order differs from the priorities, i.e., listener3 should receive the topic data first based on priority because the order of reception will affect the launch time of listener3's application.

This problem is caused by a PollManager thread, which is an internal ROS thread in communication middleware (Fig. 5). This thread is related to the send process. To start, when the ROS *publish()* API is called, a publisher's main thread pushes a message to PublishQueue. A PollManager thread references the oldest data in PublishQueue, and it pushes each WriteBuffer. This thread sends a message to each subscriber using sockets in the order of elements in SubscribeLinkQueue. An element in SubscribeLinkQueue is subscriber link data, and these elements are ordered according to the order by which subscribers establish the connections. In Fig. 4, listener1 establishes the connection first. Consequently, the publisher sends a message to listener1 first.

### D. Time Difference Evaluation

Time differences using the car detection system shown in Fig. 2 (see Section III-A) were also evaluated. The results of 2,500 measurements are shown in Fig. 6. Time differences

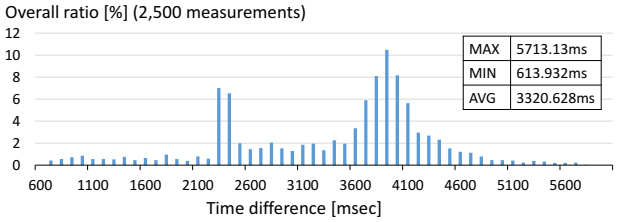


Fig. 6. Time difference between sensor timestamps.

were caused by the different processing cycles of car_dpm and scan2image. Nodes can run and publish a topic in any cycle. Therefore, even if the sensor timestamps of image_raw and scan are the same, a time difference occurs when car_fusion receives the data. These time differences cannot be guaranteed because nodes, e.g., car_dpm and scan2image, can run in any cycle.

### E. Factors that Influence Real-Time Performance

From the preliminary evaluations discussed in Sections III-C and III-D, we can define two factors that reduce the real-time performance.

A publisher does not consider the priority of the subscriber; thus, the order in which sequences are received by subscribers is not guaranteed. When a subscriber receives data from a topic, the subscriber's application is launched; thus, the sequences in which data is received affect the time at which the subscriber's application is launched. However, a publisher sends topic data to each subscriber according to the order by which subscribers establish connections. Therefore, the priorities are not always the same as those provided by the OS. If the sending order is not guaranteed, developers must consider a case in which the connection was established at the end. Therefore, the WCET of the data transfer increases as the number of nodes subscribing to the same topic increases. To address this issue, the priorities provided by OS must be supported by ROS.

When node subscribes to multiple topics, a difference between the sensor timestamps of the received data items occurs. However, a time difference cannot be guaranteed below a constant time when a subscriber combines two or more data (Section III-D). If this is not guaranteed in an autonomous driving system, traffic accidents may occur. For example, a car_fusion node may recognize a near car as the distant car incorrectly because the time difference is not guaranteed. Although the ability to run nodes and publish a topic in any cycle is an advantage of ROS, this advantage also causes the problem.

## IV. Real-Time Extension of ROS

### A. Priority-Based Message Transmission Mechanism

We present a priority-based message transmission mechanism (Fig. 7) to solve the problem described in Section III-E. This mechanism is implemented in the ROS communication core library; thus, the developers do not need to change the interfaces in existing node programs.

In the priority-based message transmission mechanism, a PollManager thread adds a priority to a subscriber link and

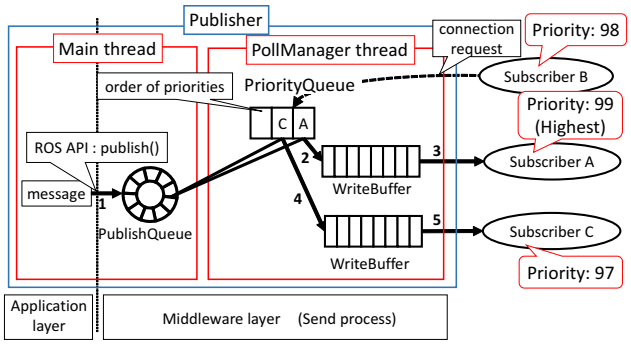


Fig. 7. Priority-based message transmission mechanism.

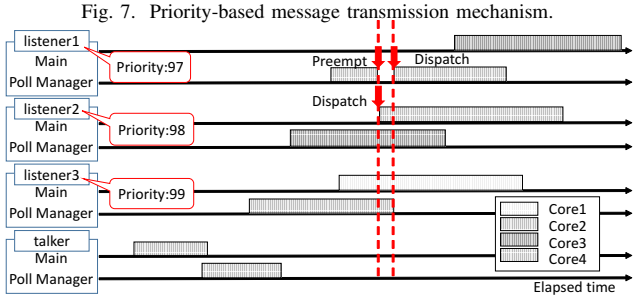


Fig. 8. System trace of the system shown in Fig. 3 after implementation of the priority-based message transmission mechanism.

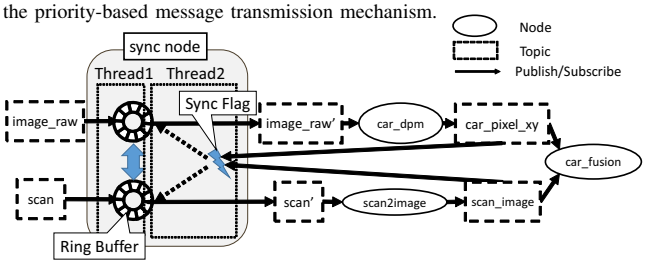


Fig. 9. Synchronization system overview.

enqueues the link in PriorityQueue when the publisher receives a connection request from a subscriber. This thread sorts elements in PriorityQueue according to priority. The thread refers to PriorityQueue rather than the existing SubscriberLinkQueue (Fig. 5). Consequently, the sending order is guaranteed. To reduce the variability of data transfer times caused by an increased number of subscribers, a PollManager thread sends a message after a PollManager thread is stored in each WriteBuffer.

Figure 8 shows an evaluation of the presented mechanism is in which the system shown in Fig. 3 was traced using *ftrace*. Subscribers receive a message in the order of priority, and we confirm that the main thread that implements the application is actually called.

### B. Synchronization System

*1) Synchronization system design:* The difference between the sensor timestamps must be guaranteed to be below a constant time. In addition, other nodes subscribe to the topics such as image_raw or scan topics (Fig. 2), and these nodes must not be affected. We present a synchronization system that uses a sync node (Fig. 9) to satisfy the requirements. This sync node is set in front of car_dpm and scan2image, which must be synchronized, and the sync node subscribes to the topics that are subscribed to by a node (car_fusion) that subscribes multiple topics. This node control to publish cycle and publish image_raw' and scan' topics similar to the image_raw and scan topics. Nodes can dynamically change the subscribing or publishing topics at runtime; thus, by introducing the sync node, nodes do not need to modify the node programs. In addition, the sync node can control the processing of scan2image according to the cycle of car_dpm, which reduces CPU utilization. Without the sync node, scan2image increases CPU utilization because it can run at any cycle period.

An existing synchronization system [18] can match messages for a set of topics; however, the existing system is run in a node that subscribes multiple topics, and it is implemented in middleware. Therefore, existing node programs must change interfaces to synchronize. In addition, the existing system cannot reduce CPU utilization, and developers find it hard to change synchronization algorithm because it is implemented in middleware. By implementing a synchronization system as a node, these problems do not occur.

*2) Synchronization algorithm:* In the synchronization system, we use one algorithm to select data in multiple topics, i.e, the Minimum Time Difference (MTD) algorithm. The MTD algorithm minimizes time differences between recent data as possible. Figure 9 gives examples of synchronized car_fusion. Thread1 of the sync node has ring buffers for the image_raw and scan topics, and it will be stored in the ring buffer for each received data. We define the timestamp of the i-th image_raw and scan data in the ring buffer as $T_i^{img}$ and $T_i^{scan}$, respectively. The most recent timestamps are denoted $T_{last}^{img}$ and $T_{last}^{scan}$, respectively. Thread2 subscribes to the car_pixel_xy and scan_image topics. It sets a Sync Flag when it receives these topics. If $T_{last}^{img} \leq T_{last}^{scan}$ is true, $T_{last}^{img}$ and $T_i^{scan}$ that satisfy $min(T_{last}^{img}, T_i^{scan})$ is published by Thread2. If $T_{last}^{img} \geq T_{last}^{scan}$ is true, $T_i^{img}$ and $T_{last}^{scan}$ that satisfy $min(T_i^{img}, T_{last}^{scan})$ is published. Consequently, the cycle occurs by the sync node.

A selecting algorithm such as MTD algorithm can be changed by developers depending on a system. In this case, the MTD algorithm can align the processing cycle of car_dpm and scan2image to guarantee that the time difference in the timestamp of the sensor is below a constant time. When the car_fusion node use two of the sensor data, we denote the cycle of the i-th sensor and maximum cycle of the sensor as $T^i$ and $T_{max}^i$, respectively. The maximum time difference of the sensor data ($T_{diff}$) is expressed by Eq. 1.

$$T_{diff} \leq min(T_{max}^1, T_{max}^2) \quad (1)$$

## V. Evaluation

The priority-based message transmission and the synchronization system are evaluated using autonomous driving applications. The environments are listed in Table I.

### A. Priority-Based Message Transmission Mechanism

*1) Evaluation method:* In an autonomous driving system, nodes can subscribe to the same topic [14]. For example,

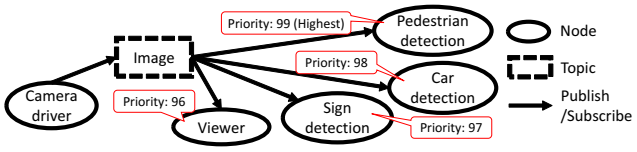


Fig. 10. Evaluation scene.

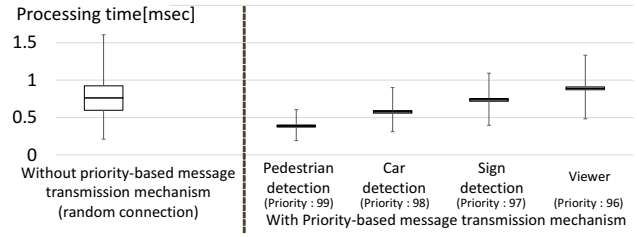


Fig. 11. Send processing time with and without priority-based message transmission mechanism.

as shown in Fig. 10, four nodes (pedestrian detection, car detection, sign detection and viewer) subscribe to the same image topic published by the camera driver node. The pedestrian detection node has the highest priority because human lives are risk if this node exceeds the recognition deadline. In the situation shown in Fig. 10, we compare the pedestrian detection node send processing time with and without presented priority-based message transmission mechanism. Note that without this mechanism, the pedestrian detection node can establish a random connection.

*2) Evaluation results:* The evaluation result is shown in Fig. 11. Without the presented mechanism, the WCET was 1.606 msec; with the mechanism, the WCET of pedestrian detection (highest priority) was 0.605 msec. Although the time variance without the mechanism was $4.599 \times 10^{-8}$, the variance of pedestrian detection with the presented mechanism was $5.702 \times 10^{-10}$. Similary, the other nodes are reduced variances. From these results, it is evident that the presented mechanism reduces the WCET and the variability.

### B. Synchronization System

*1) Evaluation items:* The synchronization system was evaluated by four items. The first is the time difference between the sensor from car_pixel_xy and scan_image received by the car_fusion node. This evaluation indicates that the time difference can be guaranteed to be below a constant time. The second item is the change in CPU utilization with the sync node. When the sync node is used in the car detection system, the CPU utilization of the scan2image node is reduced in this operation cycle. The third item is cycle latency between the send and receive times of a long cycle node request. In this case, the long latency cycle node is the car_dpm node. The latency is the time between car_dpm publishing data and the time the image_raw' topic is received.

*2) Sensors data time difference:* The sensor time difference from car_pixel_xy and scan_image in car_fusion was also measured, as shown in Fig. 12. In addition, to obtain the guaranteed time difference value, we measured the cycle period of the scan and image_raw topics.

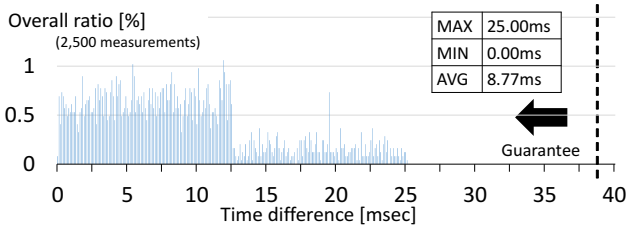


Fig. 12. Time difference of scan timestamp and image_raw timestamp.

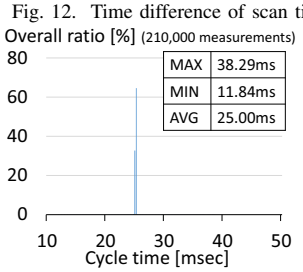


Fig. 13. scan cycle time.

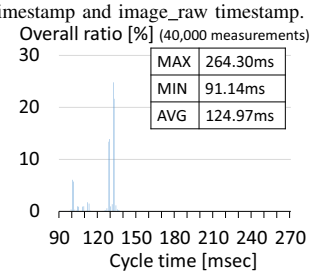


Fig. 14. image_raw cycle time.

TABLE II
CPU UTILIZATION WITH AND WITHOUT THE SYNCHRONIZATION SYSTEM

| | | CPU Utilization |
|---|---|---|
| Without synchronization | scan2image | 13.37% |
| With synchronization | scan2image | 0.70% |
| | sync | 2.67% |

Figure 13 shows the publish cycle period of the scan topic. Similarly, Fig. 14 shows the publish cycle period of the image_raw topic. The maximum period of the scan topic, $T_{max}^{scan}$, is 38.29 msec, and the maximum period of the image_raw topic, $T_{max}^{img}$, is 264.30 msec. From Eq. 1, $T_{diff} \leq 38.29$ msec. The maximum time difference is 25.00 msec, which is less than the time required to obtain the guaranteed time difference value, i.e., 38.29 msec.

*3) CPU utilization:* CPU utilization of the scan2image node and sync node with and without the synchronization system was measured for 10 min. The results are presented in Table II. Prior to the introduction of the presented system, the scan2image CPU utilization rate was 13.37%. After the sync node was introduced, the utilization rate was reduced by 0.67%. CPU utilization of this sync node was 2.67%, i.e., CPU utilization was reduced by 10.00%.

## VI. Related Work

**rosc:** rosc [23] is a dependency-free ROS client implementation in ANSI C that aims to support small embedded systems and any operating system. It is available as an alpha release, and the developers had planned real-time extensions to rosc. However, the development of rosc has ended.

**$\mu$ROS:** $\mu$ROS [24] is a middleware for embedded systems that need to communicate with ROS in a lightweight and quick manner. However, the ROS node code does not work with the $\mu$ROSnode. In addition, at present, $\mu$ROS is an alpha release and has not been in development for very long.

**rosrt:** rosrt [25] is a package that contains real-time safe tools for interacting with ROS. When used together with Xenomai [26], which is RTOS, it guarantees that the CPU time is not affected by other processes. It is currently experimental

TABLE III
COMPARISON OF PRESENTED REAL-TIME EXTENDED OF ROS AND PRIOR WORK

| | Libraries (Open CV, PCL, etc.) | Modification less | Synchronization | Open source |
|---|---|---|---|---|
| rosc | | x | | x |
| $\mu$ROS | | | | x |
| rosrt | x | | | x |
| RT-ROS | △ | △ | | |
| ROS 2.0 | x | | | x |
| Real-time extension of ROS | x | x | x | x |

and not API stable because it has been implemented for the PR2 robot [19].

**RT-ROS:** RT-ROS [16] is based on the hybrid OS platform RGMP (RTOS and GPOS on Multi-Processors). Nodes that require real-time constraints can run on RTOS. However, RT-ROS is not open source and does not appear to support libraries related to ROS and necessity to modify existing ROS node programs.

**ROS 2.0:** ROS 2.0 [27], [28] is a next-generation ROS that uses DDS as a communication middleware. It is targeted at real-time embedded systems; thus, it supports real-time constraints. Although it can communicate with existing ROS nodes to enable mixed systems, the ROS node code does not work with ROS 2.0. In other words, the legacy code is not supported for real-time constraints. In addition, ROS 2.0 is an alpha release.

Table III shows a comparison of the presented real-time extension to ROS and previous work. rosc and $\mu$ROS do not support libraries related to ROS such as Open CV and PCL. Note that the presented real-time extension to ROS has all features shown in Table III. The presented real-time extension to ROS is open source [29], [30] and does not require the modification of existing ROS node programs.

## VII. CONCLUSION

We have presented a priority-based message transmission mechanism, i.e., a real-time extended ROS communication core library and a new synchronization system that does not require the modification of existing ROS systems.

The priority-based message transmission mechanism provides a communication method that supports priorities. The presented mechanism demonstrated to reduce time variance and overheads caused by ROS communication middleware. In addition, the synchronization system works on middleware and synchronizes communication between nodes. When the presented system matches messages coming from a set of topics named buses over which nodes exchange messages, some epsilon time difference occurs. This system indicated what epsilon was guaranteed below a constant value. In addition, the presented system reduced the total CPU utilization. The presented mechanism and system are open source [29], [30] and do not require the modification of existing ROS systems. The synchronization system is already in use in the autonomous driving system [14].

In the future, we plan to port the presented system and mechanism to an RTOS to improve the real-time performance and reduce the communication overhead caused by sockets.

## REFERENCES


[1] M. Buehler, K. Iagnemma, and S. Singh, *The DARPA urban challenge: autonomous vehicles in city traffic*. springer, 2009, vol. 56.

[2] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer *et al.*, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

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

[4] J. M. O'Kane, *A Gentle Introduction to ROS*. Independently published, Oct. 2013, available at http://www.cse.sc.edu/~jokane/agitr/.

[5] "OpenCV," http://opencv.org/.

[6] "Point Cloud Library (PCL)," http://pointclouds.org/.

[7] "OpenNI," http://openni.ru/.

[8] C. Ainhauser, L. Bulwahn, A. Hildisch, S. Holder, O. Lazarevych, D. Mohr, T. Ochs, M. Rudorfer, O. Scheickl, T. Schumm, and F. Sedlmeier, "Autonomous Driving needs ROS," 2013.

[9] M. Aeberhard, T. Kuhbeck, B. Seidl, and et al., "Automated Driving with ROS at BMW," 2015.

[10] S. Kumar, L. Shi, N. Ahmed, S. Gil, D. Katabi, and D. Rus, "Carspeak: A content-centric network for autonomous driving," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 259–270, Aug. 2012.

[11] A. Hernandes, A. Brito, H. Roncancio, D. Magalhaes, M. Becker, R. Sampaio, and B. Jensen, "GISA: A Brazilian platform for autonomous cars trials," in *Industrial Technology (ICIT), 2013 IEEE International Conference on*, Feb 2013, pp. 82–87.

[12] S. Kumar, S. Gollakota, and D. Katabi, "A cloud-assisted design for autonomous driving," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. ACM, 2012, pp. 41–46.

[13] S. Kagami, T. Hamada, and S. Kato, "Autonomous vehicle navigation by building 3d map and by detecting human trajectory using lidar," in *CPSNA*, 2013.

[14] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," vol. 35, no. 6, pp. 60–68, 2015.

[15] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

[16] 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 Generation Computer Systems*, pp. –, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X15001831

[17] J. Sticha, "Validating the Real-Time Capabilities of the ROS Communication Middleware," July 2014.

[18] "message filters for synchronization," http://wiki.ros.org/message_filters.

[19] J. Barry, M. Bollini, A. Holladay, L. Kaelbling, and T. Lozano-Pérez, "Planning and control under uncertainty for the pr2," in *IROS PR2 Workshop*, September 2011, extended abstract.

[20] A. Garg, "Real-time linux kernel scheduler," *Linux Journal*, vol. 2009, no. 184, p. 2, 2009.

[21] T. Bird, "Measuring function duration with ftrace," in *Proc. of the Japan Linux Symposium*, 2009.

[22] S. Rostedt, "Finding origins of latencies using ftrace," *Proc. RT Linux WS*, 2009.

[23] N. Ensslen, "Introducing rosc," in *ROS Developer Conference 2013*, 2013.

[24] M. Migliavacca, A. Zoppi, M. Matteucci, and A. Bonarini, "uROSnode running ROS on microcontrollers," in *ROS Developer Conference 2013*, 2013.

[25] "rosrt," http://wiki.ros.org/rosrt.

[26] L. M. Surhone, M. T. Tennoe, and S. F. Henssonow, *Xenomai*. Betascript Publishing, 2010.

[27] "ROS 2.0," http://design.ros2.org/.

[28] E. Fernandez, T. Foote, W. Woodall, and D. Thomas, "Next-generation ROS: Building on DDS," in *ROS Developer Conference 2014*, 2014.

[29] "Real-time extended to ROS," https://github.com/CPFL/RT-ROS.

[30] "Synchronization system generator," https://github.com/CPFL/Autoware/tree/master/ros/src/system/sync.