ThreadedCallback: Improving Real-time Performance of ROS 2

Bo Peng[1,a)] Yuqing Yang[1] Yoshikazu Okumura[2] Atsushi Hasegawa[2] Takuya Azumi[1]

Abstract: Autonomous robot systems have been appearing rapidly to meet the increasing demands. The Robot Operating System (ROS) has been used widely in connection with such systems. ROS has been upgraded to ROS 2 using the Data Distribution Service (DDS) to accommodate high communication latency. ROS 2 contains an Executor module to support execution management in real-time performance. However, improving real-time performance is difficult in ROS 2. This paper proposes ThreadedCallback which enables to run callback as a thread with a specific central processing unit (CPU) core and scheduling policy. We use a ping-pong test to examine and evaluate the performance of the proposed approach. Experimental results show that our approach can achieve significant performance improvement over the standard Executor in ROS 2.

Keywords: Real-Time, ROS 2, Performence Test

1. Introduction

In recent years, autonomous robotic systems have been a popular topic. These systems create and maintain a map of its surrounding environment based on various sensors situated in different parts of the vehicle. They process the sensory input, plot a path, and send instructions to the vehicle's actuators, which control acceleration, braking, and steering. A principal feature of these systems is their sensitivity to communication delays. Moreover, Robot Operating System (ROS [30]) has been useful in the development of many such systems (e.g., Autoware [2,13]). ROS provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and other functions. These can be helpful to software developers creating robot applications.

ROS is not suitable for real-time embedded systems, because it can only run on a few operating systems and does not meet real-time runtime requirements. ROS 2 has been proposed to satisfy the requirements of the expanding ROS community [19]. ROS 2 supports different programming languages. Each programming language has a separate programming language interface, and all programming languages share a common layer, ROS Client Library (RCL) layer. Consequently, if general optimizations are made in RCL layer, all applications can be optimized regardless of which language is used. In ROS 2, the ROS transport system is replaced by the Data Distribution Service (DDS) [21], an industry-standard real-time communication system and end-to-end middleware.

ROS 2 has other features in addition to the DDS. Developers can bundle multiple nodes in one OS process using ROS 2. An Executor has been introduced in rclcpp and rclpy to coordinate the execution of process callbacks. However, the standard ROS 2 Executor has several limitations in the C++ application programming interface (API: rclcpp), such as the precedence of timers and non-preemptive round-robin (RR) scheduling for non-timer handles [4].

Real-time Executor can be used to solve these problems. Real-time Executor not only ensures deterministic execution, but also guarantees real-time performance. In previous research [32], we reported that thread priority could affect the jitter of sleep or timing of callback. However, if threads are in different central processing unit (CPU) cores, they have no effect on each other. Hence, it would appear to be natural to run callback as a thread with a specific CPU core and scheduling policy. Moreover, we attempt to detect deadline misses. The existing timer mechanism can be applied to an overrun handler (the timer can be triggered even if callback is running by splitting the callback thread).

The contributions of this work are as follows:

- This paper proposes ThreadedCallback which enables to run callback as a thread with a specific CPU core and scheduling policy.
- This paper explores the real-time performance of ThreadedCallback.
- This paper clarifies the real-time Executor and explains the difference between the proposed approach and the existing real-time Executor.

Organization: The remainder of this paper is organized as follows. Section 2 describes ROS 2 and the ROS 2 standard Executor. Section 3 presents our proposed approach. Section 4 evaluates real-time performance of the proposed approach on ROS 2 and shows a use case of it. Section 5 discusses related work. Section 6 concludes the paper and offers suggestions.

[1] Saitama University, sakura-ku, Saitama-shi, Saitama-ken 338–8570, Japan
[2] Research Institute of Systems Planning, Inc, Nihonkaikan, 18-6, Sakuragaoka, Shibuya, Tokyo 150-0031, Japan
[a)] peng.b.203@ms.saitama-u.ac.jp

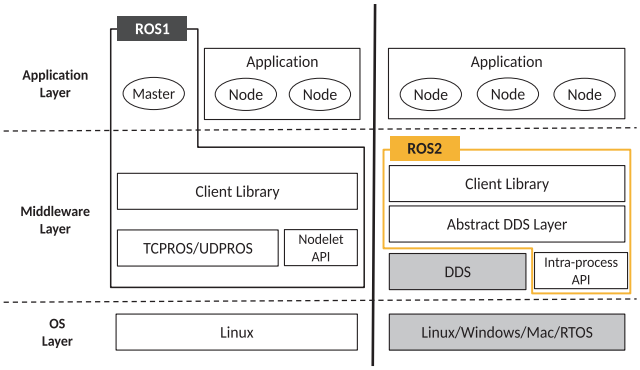

Fig. 1 The difference in architecture between Robot Operating System (ROS) and Robot Operating System 2 (ROS 2)

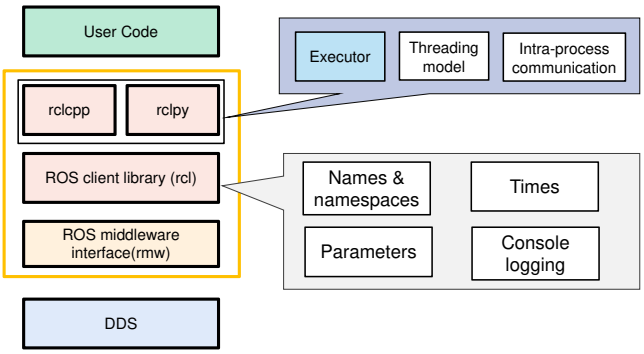

Fig. 2 The architecture of Robot Operating System 2 (ROS 2)

2. System Model

In this section, we provide background knowledge and the system model. First, this section discusses the ROS 2 system model, focusing on the communication system, and we compare it to ROS. Secondly, we describe and analyze the ROS 2 standard Executor. In addition, we provide a detailed description of the ROS 2 scheduling.

2.1 ROS 2

The difference between ROS and ROS 2 is shown in **Fig. 1**. ROS 2 is better adapted than ROS to newer requirements of robot system development, such as real-time control [14] and increasing distributed processing. Moreover, ROS 2 can create multiple nodes in a single process.

ROS 2 is designed as a three-layer stack. The upper layer is equivalent to a user application. There are several language-specific wrappers, to extend ROS 2 to multiple programming languages, such as Python with rclpy [25] or C++ with rclcpp [24]. The implementation of the upper layer must wrap a C interface, ROS Client Library (RCL). The middle layer consists of a library connecting RCL and the ROS Middleware Interface (RMW). The ROS 2 architecture is shown in **Fig. 2**. The lower layer is the implementation of ROS middleware, which provides communication between ROS nodes [9].

In **Fig. 3**, we show the communication-related components of ROS 2 as follows [5]:

- Node: a process that promotes computation independently;
- Topic: a communication channel used to transmit messages between publishers and subscribers;

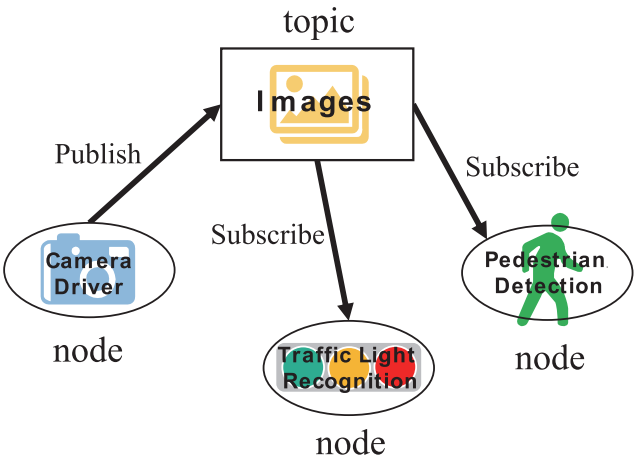

Fig. 3 Robot Operating System 2 (ROS 2) publish/subscribe model

- Message: a simple data structure which is defined by .msg files.

The basic communication model in ROS 2 is called publish/subscribe model. This model can be understood through the example in Fig. 3. The “Camera Driver” node publishes messages to the “Images” topic. The “Car Detection” node and the “Pedestrian Detection” node subscribe to the “Images” topic and utilize the messages. The publish/subscribe model can be suitable for most distributed systems.

ROS 2 is based on the DDS, which is used to discover *nodes*, to serialize, and to transmit information. The DDS provides some functions required by the ROS system, such as distributed discovery nodes (not as centralized as ROS 1). New nodes can be discovered by other nodes in the same DDS network by using the DDS publishing and subscription/reading mechanism. Using a third-party mature DDS (such as RTI [31] and eProsima [6]) as a framework for ROS underlying communication and related core functions can substantially reduce the workload of ROS 2 development.

The Data-Centric Publish/Subscribe (DCPS) model is an important component of DDS. This model can perform data transfer between processes more efficiently even on distributed heterogeneous platforms. The pub/sub method of distributed communication is a common mechanism that can be used in many types of applications. The DDS standard defines a language-independent model for pub/sub communications and standardizes mappings to various implementation languages. The “data-centric” part of the term DCPS refers to the basic concepts supported in API design. In object-centric communication, in contrast, the focus is on the interface between applications. Data and object-centric communication provide a complementary paradigm for distributed systems. Applications can require both at the same time. However, real-time communication typically works more effectively with data-centric models. A data-centric system consists of a data publisher and a data subscriber. Communication is based on a known type of data passed from the publisher to the subscriber in named streams. Each data transfer between processes is performed according to the Quality of Service policy, which represents the data transport behavior [26].

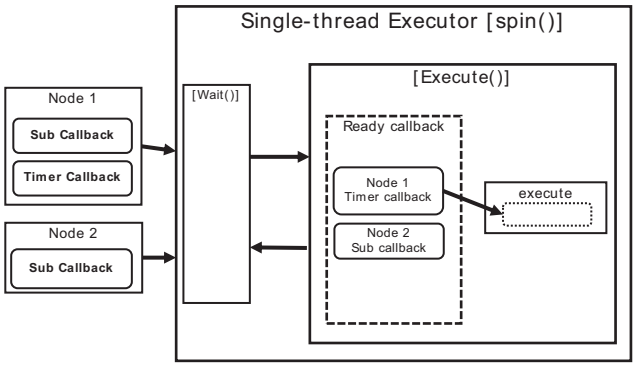

Fig. 4 ROS 2 standard Execution model

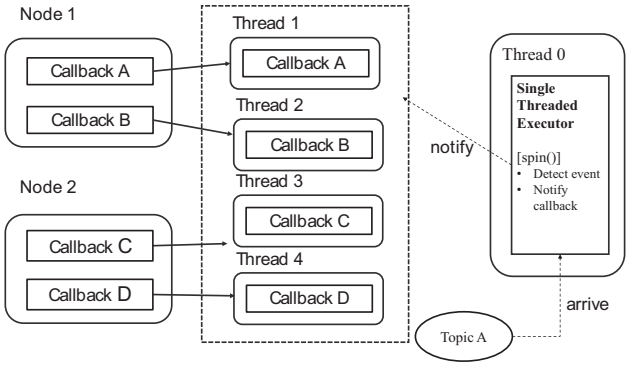

Fig. 5 The basic principle of ThreadedCallback

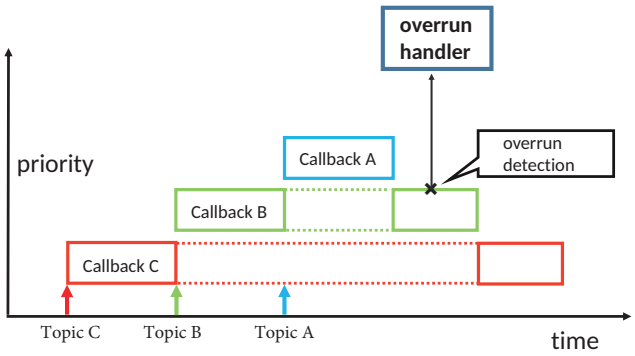

Fig. 6 The sample scenario of ThreadedCallback

2.2 ROS 2 Standard Executor

The Executor module is used to coordinate the order and timing of available communication tasks. The Executor module is implemented in rclcpp and rclpy. ROS 2 standard Execution model of ROS 2 is shown in **Fig. 4**.

ROS 2 typically defines one Executor for each process in the custom main function or the function created by the launch system. The Executor coordinates the execution of all callbacks by checking for available work (timers, services, and messages) from the DDS queue and dispatching to one or more threads, using *SingleThreadedExecutor* [28] and *MultiThreadedExecutor* [23].

The Executor has two spin functions: spin_node_once and spin_some. According to the thread or concurrency scheme provided by the subclass implementation, it coordinates the nodes and callback groups by finding available work and completing the work.

The Executor is responsible for scheduling callbacks. However, the Executor does not provide methods for prioritizing or classifying incoming callbacks. Moreover, it does not fully utilize the real-time characteristics of the underlying operating system scheduler. Its first-in first-out (FIFO) mechanism also limits the range of worst-case latencies that can be caused by the execution of each callback.

The ROS 2 scheduling mechanism is not complicated and exhibits a behavior similar to the ROS spin thread: the Executor looks up the wait sets, which notifies it of any pending callback in the DDS queue. If multiple pending callbacks are in the DDS queue, the ROS 2 Executor executes them in the order in which they are registered with the Executor.

The Executor has four available categories of callbacks: timers, subscribers, services, and clients. Timers are triggered by system-level timers and have the highest priority. The Executor always processes timers first. However, the order of priority of the remaining three items is not obvious. Non-timer handles follow non-preemptive RR scheduling. A snapshot called readySet is updated when the Executor is idle, and in this step, it interacts with the DDS layer updating the set of ready tasks. Messages arriving during the processing of the readySet are not considered until the next update, which depends on the execution time of all remaining callbacks. This leads to priority inversion, as lower-priority callbacks can implicitly block higher-priority callbacks by prolonging the current processing of the readySet. Furthermore, the readySet contains only one task instance. For example, even if multiple messages on the same topic are available, only one instance is processed until the Executor is idle again and the readySet is updated from the DDS layer. This intensifies priority inversion, as a backlogged callback might have to wait for multiple processing of readySets until it is considered for scheduling. This can result in non-timer callback instances being blocked by multiple instances of the same lower-priority callback.

3. Design and Implimentation

This section clarifies the basic principle of ThreadedCallback. Moreover, this section compares it with existing research (Callback-group-level Executor) to point out differences and improvements.

3.1 ThreadedCallback

ThreadedCallback allows running callbacks in a specific thread with its own settings. A DDS child process affects sleep jitter if only one priority is used. Even if callbacks are performed in a thread other than the main thread, the scheduler falls into RR or FIFO if only one priority is used. Therefore, it fits the programmer to specify a priority or scheduling policy.

Because the thread priority could affect the jitter of ROS 2 callbacks timing, we propose to create threads per callback to improve the real-time performance when configuring the threads real-time settings. Nevertheless, only use ThreadedCallback cannot guarantee the real-time performance, we need more mechanisms to guarantee it. The basic principle of ThreadedCallback is shown in **Fig. 5**. As shown in Fig. 5, the task thread is created for each callback in nodes. Threads are created internally when

calling rclcpp::init(), add_node(). And we use pthread [17] to assign priority and cpu to each callback thread. When the topic receives the message from pubNode, the Executor will notify the corresponding callback thread to execute.

A sample scenario as shown in **Fig. 6** illustrates the effect of priority setting. Only one CPU core is used in this scenario. The scenario has three callbacks with different priorities, CallbackA, CallbackB, and CallbackC. CallbackA has the highest priority and the shortest task. CallbackB has the middle priority and a middle-length task. CallbackC has the lowest priority and the longest task. CallbackA must run even if CallbackB or CallbackC is running. If the TopicB comes while CallbackC is executing, CallbackC processing is interrupted and CallbackB processing is started. Next, when the TopicB comes, the same processing is performed. After the processing of CallbackA is completed, the suspended processing of CallbackB is resumed. After the processing of CallbackB is completed, the same processing is performed for CallbackC.

Using such a thread, we could get the following.

- ROS 2 Executor thread interruption even if callbacks are running.
- Easily implemented preemption.
- Prevention of scheduling procedure duplication in the Executor and OS, making it easy to verify.
- Possible combination with a Logical Execution Time [15] (LET) scheduler.

In addition, we want to detect deadline miss. Therefore, we add an overrun handler function to ThreadedCallback. The realization of this function is mainly through the existing timer mechanism. The timer can be triggered at any time, even by splitting the callback thread to run the callback. The overrun handler function mainly includes two parts, overrun detection and overrun handler. The overrun handler is worth mentioning that in ThreadedCallback, this function corresponds to a non-DDS event. The QoS policy in DDS can configure the deadline period, but this is only for DDS events. Although ROS 2 uses DDS as its communication system, ROS 2 also includes intra-process communication, and DDS will not be used in this process. Once the overtime situation is detected, follow-up processing will be performed in accordance with the set method. According to the specific needs, you can set the overrun callback to be executed again from the beginning or terminate the current callback directly.

3.2 Comparison with Callback-group-level Executor

Callback-group-level Executor (Cbg-Executor) [16] is an open source rclcpp Executor API that was developed by micro-ROS [20]. Callback-group-level Executor is derived from the default rclcpp Executor. An Executor instance can be classified as particular callback groups, and the threads of the Executor can be prioritized based on the real-time requirements of these groups. We have analyzed the performance of Cbg-Executor in our previous study [32].

Real-time profiles, such as RT-CRITICAL and BEST-EFFORT introduced in the callback-group API, take advantage of the callback group concept in rclcpp. Therefore, each callback that requires specific real-time guarantees can be associated with a dedicated callback group when it is created. This allows a single node to assign callbacks with different real-time profiles to different Executor instances in one process. Cbg-Executor introduced an enum that distinguishes up to three real-time classes per node, and changed association with an Executor instance from nodes to callback groups. Moreover, Cbg-Executor can add and remove individual callback groups in addition to entire nodes.

Table 1 Evaluation environment

CPU	Model number	AMD Ryzen 5 3600
	Frequency	3.59 GHz
	Core	6
	Thread	12
Memory		16 GB
ROS 2		Foxy
DDS implementation		Fast DDS
OS	Distribution	Ubuntu 20.04
	Kernel	Fully Preemptible Kernel (RT)

The most obvious difference between ThreadedCallback and Cbg-Executor is the number of Executors. In Cbg-Executor, each thread has one corresponding Executor. However, in ThreadedCallback, all threads correspond to only one Executor.

The proposal is better than Cbg-Executor at fixing priority inversion issues. The Executor is not protected in Cbg-Executor. The Executor overhead when many priorities are required is also an unsolved problem. The task of callback groups is to partition relevant callbacks. An irrelevant or lower-priority callback does not prevent higher-priority callbacks. In the same callback group, callbacks are executed in FIFO like order. This is similar to using multiple same-priority FIFO threads in terms of scheduling. However, callback groups look to be more than that, because it handles locks and Mutually-Executable. Hence, the callback thread and the Executor are separated. In addition, ThreadedCallback uses one Executor to cause the Executor to focus on event detections and event triggers, which can avoid priority inversion due to Executor blocking.

4. Evaluation

In this section, we demonstrate the effectiveness of ThreadedCallback through ping-pong experiments. Furthermore, we propose a use case to verify the practicality of ThreadedCallback.

4.1 Experimental Scenarios and Methods

The hardware and software environment is outlined in **Table 1**. In this work, we used FastDDS [6], the default DDS for this evaluation. Furthermore, we used the Linux kernel with the PREEMPT RT patch [7] in this experiment. PREEMPT RT patch by using the interrupt threads, preemption of the critical area to improve real-time performance. The interrupt thread technology is to turn the interrupt service program into a thread that the operating system can schedule, and assign priority to the interrupt service program, thus reducing the operation of turning off the interrupt. The standard Linux kernel does not support preemption due to the use of spinlock. In the PREEMPT RT patch, spinlocks are converted to RT Mutexes to achieve critical area preemption.

We demonstrated the effectiveness of our proposal by using

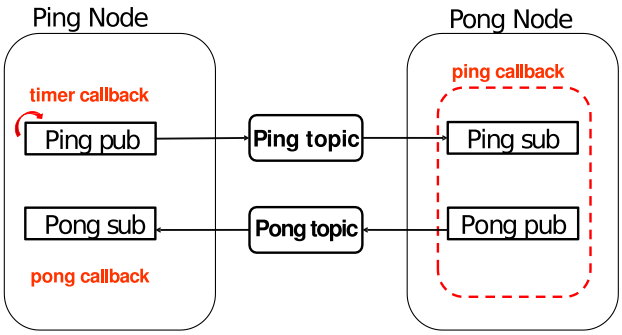

Fig. 7 Ping-pong test

ping-pong experiments, as shown in **Fig. 7**. The ping-pong test is implemented using two topics with the following publisher and subscriber. The ping-publisher (ping-pub) sends ping periodically using ping-topic. The ping-subscriber (ping-sub) and pong-publisher(pong-pub) subscribes to ping-topic and optionally pong-pub sends a pong by pong-topic. The pong-subscriber (pong-sub) subscribes to the pong-topic. We provide bash scripts to test inter-process communication scenarios, where nodes are located in one or more processes.

This test scenario has three callbacks: timer-callback, ping-callback and pong-callback. Timer-callback includes ping-pub. Ping-callback includes ping-sub and pong-pub. Pong-callback includes pong-sub. There are some variations for the number of Executors and Nodes, in this test, we chose one Executor and two Nodes. The output of the experiment is the latency of the three callbacks and the overall ping-pong.

To verify the improvement of real-time performance by ThreadedCallback, we measured the overall delay of ping-pong, and the latency of ping callback and timer callback with different timer period. We then compared the measured value with the real-time performance of SingleThreadedExecutor.

As a whole, we set the number of loops to 1,000. Firstly, we measured the overall delay of ping-pong, and the latency of ping callback and timer callback of the SingleThreadedExecutor and MultiThreadedExecutor. Secondly, we set different priorities for different threads. This experiment has two kinds of threads: main thread, child thread. Their corresponding priorities are RR98 and RR97. Thirdly, we tested the latency of ping-pong, ping callback, and timer callback of ThreadedCallback in the same core. For example, let the cpuid of all threads equal to one. In ThreadedCallback, each callback corresponds to a thread, and the priority setting of callback thread is added in this experiment. Moreover, the priority of the callback thread is set to RR96. That means the main thread has the highest priority and the callback thread has the lowest priority. Finally, we put the threads in different cores. We set the cpuid of the callback thread equal to three.

4.2 Comparison with the SingleThreadedExecutor

We discuss the real-time performance for timer callback period of 500 μs to 100,000 μs.

The experimental results of single core are shown in **Figs. 8**, **10**, and **12**. In the single core case, it can be seen overall that the performance of ThreadedCallback is worse than SingleThreadedExecutor. As can be seen in Figs.10 and 12, the latency of ping callback and timer callback of ThreadedCallback is much higher than SingleThreadedExecutor. We discuss this phenomenon for the following two reasons. Firstly, since SingleThreadedExecutor has no thread switching or interrupts, timer callback and ping callback are executed promptly. Secondly, in the single core case, the main thread, child threads, and callback threads all run on the same CPU, and the callback thread has the lowest priority, it is thus more likely to be preempted, especially when the timer callback period is extremely small.

The experimental results of multiple cores are shown in

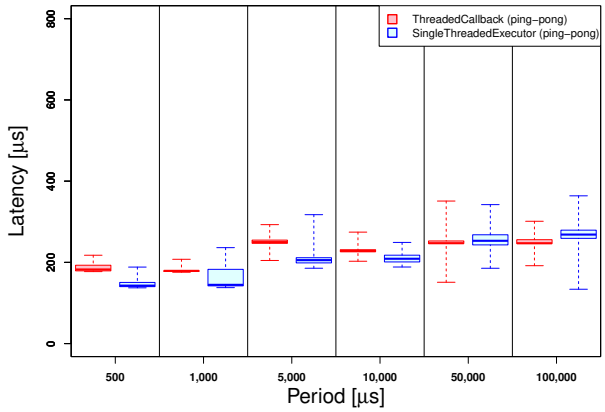

Fig. 8 The latency of pingpong in single core

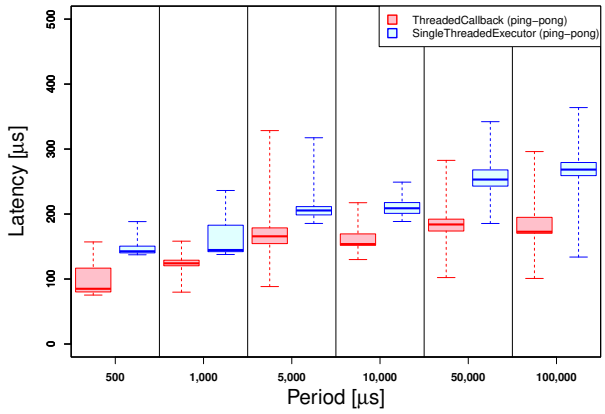

Fig. 9 The latency of pingpong in multiple cores

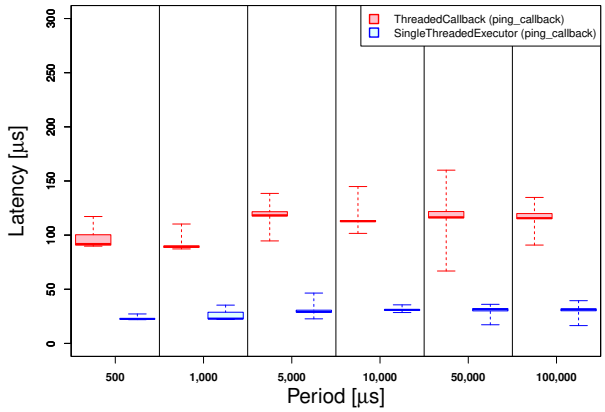

Fig. 10 The latency of ping callback in single core

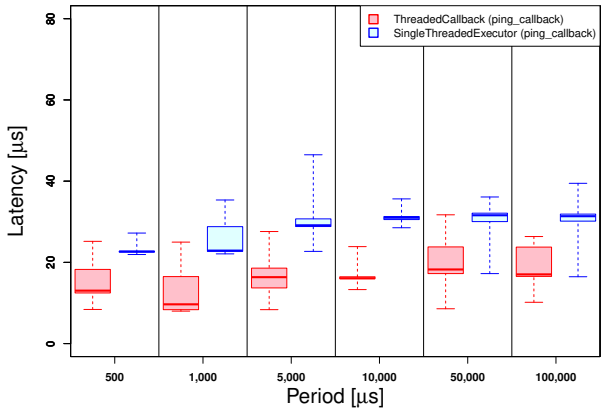

Fig. 11 The latency of ping callback in multiple cores

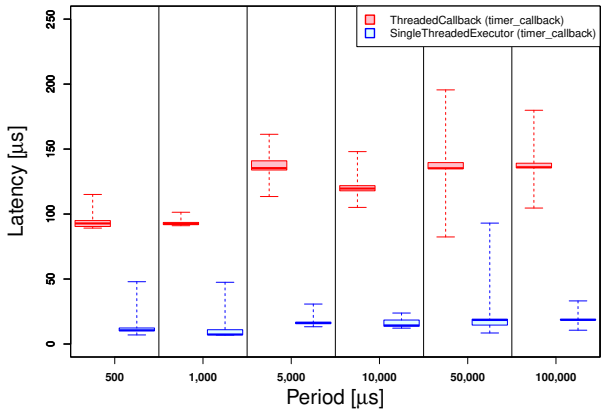

Fig. 12 The latency of timer callback in single core

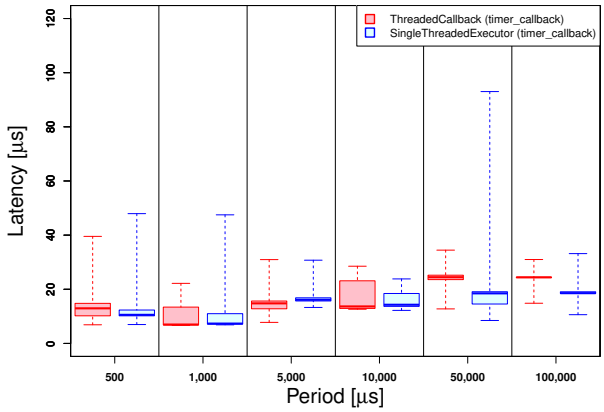

Fig. 13 The latency of timer callback in multiple cores

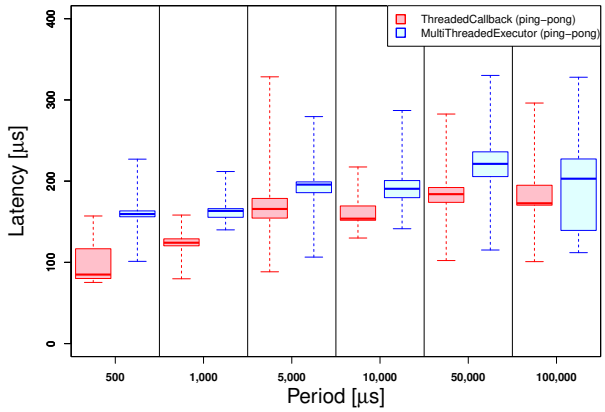

Fig. 14 The latency of pingpong in multiple cores

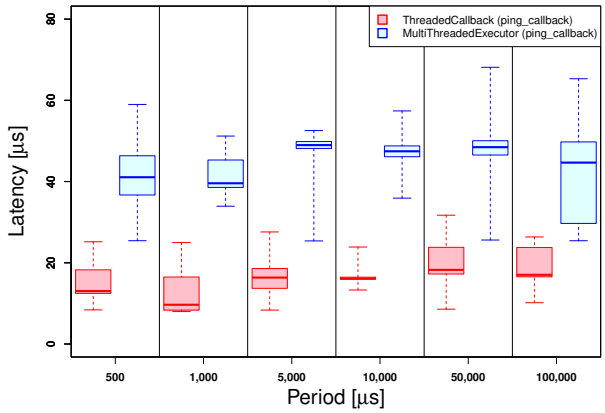

Fig. 15 The latency of ping callback in multiple cores

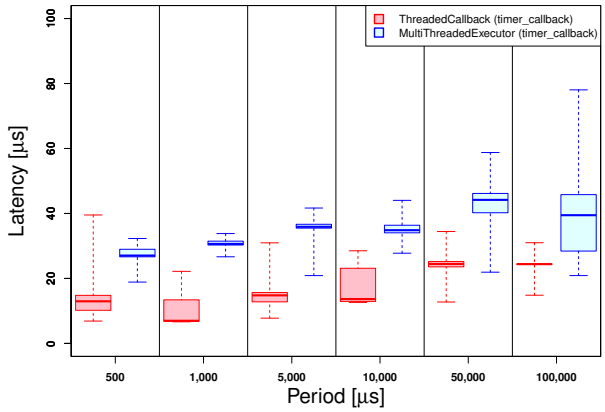

Fig. 16 The latency of timer callback in multiple cores

Figs. 9, **11**, and **13**. As can be seen in Fig. 9, the performance of ThreadedCallback significantly outperforms SingleThreadedExecutor. This is because in ThreadedCallback, even if the callback is running, the child thread can be interrupted (i.e., can read topic), and SingleThreadedExecutor can only wait for the callback to complete because it cannot preempt. Moreover, as we can see in Fig. 13, the latency of the timer callback of ThreadedCallback is basically the same as the latency of the timer callback of SingleThreadedExecutor.

4.3 Comparison with the MultiThreadedExecutor

We also measure the real-time performance of ThreadedCallback and MultiThreadedExecutor in multiple cores. The experimental results of multiple cores are shown in **Figs. 14**, **15**, and **16**. In timer period from 500 μs to 100,000 μs, we can observe that the overall latency of ThreadedCallback is better than MultiThreadedExecutor from Fig.14. We discuss this phenomenon for the following two reasons. Firstly, MultiThreadedExecutor only consider one message per handle. This means that non-timer callback instances might be blocked by multiple instances of the

Table 2 Comparison of ways to explore ROS

	Many-Core	LET Paradigm	Response Time	Performance Test	RTOS	ROS 2	RCLCPP (C++)
Maruyama et al. [19]				✓		✓	✓
Processing Chains [4]			✓			✓	✓
ATSA [12]			✓			✓	✓
Gutiérrez et al. [8]			✓	✓	✓	✓	✓
ROS-lite [3]	✓		✓		✓		
Pemmaiah et al. [1]				✓	✓	✓	✓
Park et al. [29]				✓	✓	✓	
Igarashi et al. [11]	✓	✓					
Yang et al. [32]				✓	✓	✓	✓
Ogawa et al. [22]		✓		✓	✓		
Choi et al. [10]			✓	✓	✓	✓	✓
iRobot [27]			✓	✓		✓	✓
micro-ROS [20]		✓	✓	✓	✓	✓	
This paper			✓	✓	✓	✓	✓

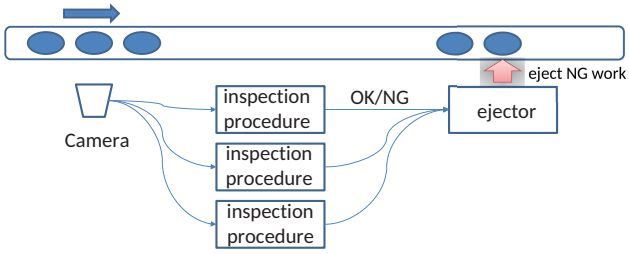

Fig. 17 Visual inspection system example

same lower-priority callback. Secondly, non-timer handles are using Non-preemptive RR scheduling in MultiThreadedExecutor. This leads to priority inversion [18], as lower-priority callbacks may implicitly block higher-priority callbacks. By analyzing the results of this experiment, we can draw a conclusions that the real-time performance of ThreadedCallback is better than MultiThreadedExecutor in multiple cores.

4.4 Use Case

A simple visual inspection system is as shown in **Fig. 17**. The work comes per 100 ms and the camera inspection procedure takes 300 ms. There is an ejector that ejects NG work.

Because the work does not necessarily appear in a fixed period, the system needs to be able to be processed aperiodically. Under these circumstances, DDS deadline cannot meet the demand because it is specified in the maximum period of the previous pub/sub. This function requires inspection program overrun detection and precise ejector control. In addition, the requirements of the system include running multiple inspection procedures and emergency stop simultaneously. ThreadedCallback can fully meet the above requirements.

5. Related Work

Table 2 briefly summarizes the characteristics of several related methods. Research on ROS concerns primarily the message passing process. Maruyama et al. [19] conducted an experimental study aiming to compare the performance of ROS 1 and ROS 2 under different DDS implementations. Casini et al. [4] proposed a scheduling model and a response-time analysis for ROS 2. They provided a practical analysis to bind the worst-case response times of their applications. Gutiérrez et al. [8] presented an experimental setup to demonstrate the suitability of ROS 2 for real-time robotic applications. They developed an evaluation of ROS 2 communications in a hardware communication case on top of Linux. The adaptive two-layer serialization algorithm [12], allows some of the serialization to be adaptively moved to the programming language interface layer instead of the ROS 2 middleware layer to reduce complexity. Pemmaiah et al. [1] discussed how an end-to-end performance testing system could effectively provide a standardized, unbiased, and reproducible evaluation. Park et al. [29] evaluated the real-time performance of ROS 2 in both the system layer and the communication software layer. In the experiments, the system load was increased to define the real-time performance of the tasks. In addition, they implemented a multi-agent service robot system to verify the suitability of ROS 2 for actual applications. From the results, they proved that the real-time performance of ROS 2 is higher than that of ROS. Yang et al. [32] explored the performance of Callback-group-level Executor. The evaluation results show that the real-time performance of Cbg-Executor is better than that of the default ROS 2 Executor.

ROS-lite [3] is a lightweight ROS development framework for the NoC-based embedded many-core platform. ROS-lite runs with low memory consumption, allowing ROS nodes to run on each core on many-core platforms and communicate with each other.

Micro-ROS [20] places ROS 2 onto microcontrollers. The overall goal is to provide ROS 2 concepts in a suitable implementation for microcontrollers. Micro-ROS and ROS 2 have two principal differences between them. Micro-ROS uses an RTOS, Nuttx, instead of Linux, as well as DDS for extremely Resource Constrained Environments (DDS-XRCE).

Many studies have been conducted on scheduling. Ogawa et al. [22] proposed an approach to realize LET that is suitable for a powertrain application to which subscheduling is applied. Addi-

tionally, they proposed approaches that distribute LET processes into several CPUs to reduce the execution time of LET processes. Igarashi et al. [11] improved the predictability of contentions by dividing tasks into the memory access phase and the execution phase using a Directed Acyclic Graph (DAG).

For ROS 2, Choi et al. [10] proposed a priority-based chain aware scheduler for ROS 2 and its end-to-end latency analysis framework. With this scheduler, callbacks are prioritized according to the given timing requirements of the corresponding chains to reduce the end-to-end latency. Furthermore, they analyze the end-to-end latency of the proposed scheduler. It is shown that the proposed scheduler outperforms the default ROS 2 scheduling in a real scenario.

The iRobot team [27] proposed a change to how ROS 2 handles incoming events. They believe that rather than using user level waitsets, an Executor event queue design will allow events to propagate faster. When the Executor thread is waiting on events to arrive, it simply blocks the CPU from performing other work. Each event contains a type enumeration and a unique handle for processing the event.

6. Conclusion

In this paper, we conducted a proof of concept for ThreadedCallback for ROS 2. ThreadedCallback allows running callbacks in a specific thread with its own settings to avoid priority inversion issues. We also compared ThreadedCallback with CbgExecutor and demonstrated the advantages of ThreadedCallback. The results demonstrated that ThreadedCallback could improve the real-time performance of ROS 2 and help developers understand the ROS framework.

In future work, we plan to combine ThreadedCallback with LET semantics. We can increase predictability by using LET semantics. This is suitable for hard real-time systems such as self-driving systems. We aim to combine the advantages of ThreadedCallbadk and LET semantics to further optimize our proposal. The code of the ThreadedCallback and the evaluation tool can be downloaded from the GitHub repository [https://github.com/bopeng-saitama/ROS2_ThreadedCallback] and [https://github.com/bopeng-saitama/TwoWaysMeasurement].

Acknowledgments

This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO). We would like to thank Takuya Aikawa.

References

[1] Apex.AI: Performance Testing in ROS 2., https://www.apex.ai/post/performance-testing-in-ros-2.

[2] Autoware Foundation: Autoware, http://github.com/autowarefoundation/autoware.

[3] Azumi, T., Maruyama, Y. and Kato, S.: ROS-lite: ROS Framework for NoC-Based Embedded Many-Core Platform, *Proc. of 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2020).

[4] Casini, D., Blaß, T., Lütkebohle, I. and Brandenburg, B. B.: Response-time analysis of ROS 2 processing chains under reservation-based scheduling, *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, Vol. 6, pp. 1–23 (2019).

[5] Dąbrowski, A., Kozik, R. and Maciaś, M.: Evaluation of ROS 2 communication layer, https://roscon.ros.org/2016/presentations/rafal.kozik-ros2evaluation.pdf.

[6] eProsima: eProsima FastDDS., https://www.eprosima.com/index.php/products-all/eprosima-fast-dds.

[7] Foundation, T. L.: PREEMPT RT patch, https://rt.wiki.kernel.org/index.php/Main_Page.

[8] Gutiérrez, C., Juan, L., Ugarte, I. and Vilches, V.: Towards a distributed and real-time framework for robotsç Evaluation of ROS 2.0 communications for real-time robotics applications, Technical report, Erle Robotics S.L. (2018).

[9] Hood, D. and Woodall, W.: ROS 2 Update, https://roscon.ros.org/2016/presentations/rafal.kozik-ros2evaluation.pdf.

[10] Hyunjong, C., Yecheng, X. and Hyoseung, K.: PiCAS: New Design of Priority-Driven Chain-Aware Scheduling for ROS 2, *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2021).

[11] Igarashi, S., Ishigooka, T., Horiguchi, T., Koike, R. and Azumi, T.: Heuristic Contention-Free Scheduling Algorithm for Multi-core Processor using LET Model, *Proc. of the 24th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT)* (2020).

[12] Jiang, Z., Gong, Y., Zhai, J., Wang, Y.-P., Liu, W., Wu, H. and Jin, J.: Message Passing Optimization in Robot Operating System, *International Journal of Parallel Programming (IJPP)* (2019).

[13] Kato, S., Tokunaga, S., Maruyama, Y., Maeda, S., Hirabayashi, M., Kitsukawa, Y., Monrroy, A., Ando, T., Fujii, Y. and Azumi, T.: Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems, *Proc. of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)* (2018).

[14] Kay, J. and Tsouroukdissia, A.: Real-time control in ROS and ROS 2.0, https://roscon.ros.org/2015/presentations/RealtimeROS2.pdf.

[15] Kirsch, C. and Sokolova, A.: The Logical Execution Time Paradigm, *Proc. of Advances in Real-Time Systems* (2012).

[16] Lange, R.: Mixed Real-Time Criticality with ROS 2 - the Callback-group-level Executor, *ROSCon 2019 Lightning Talk* (2019).

[17] Lawrence Livermore National Laboratory: pthread, https://hpc-tutorials.llnl.gov/posix/.

[18] Locke, D., Sha, L., Rajikumar, R., Lehoczky, J. and Burns, G.: Priority Inversion and Its Control: An Experimental Investigation, *ACM SIGAda Ada Letters* (1988).

[19] Maruyama, Y., Kato, S. and Azumi, T.: Exploring the performance of ROS 2, *Proc. of International Conference on Embedded Software (EMSOFT)*, pp. 5–15 (2016).

[20] micro-ROS: micro-ROS, https://github.com/micro-ROS.

[21] Object Management Group (OMG): Data Distribution Services (DDS) v1.4, http://www.omg.org/spec/DDS/1.4/.

[22] Ogawa, M., Honda, S. and Takada, H.: Efficient Approach to Ensure Temporal Determinism in Automotive Control Systems, *Proc. of 2018 8th International Symposium on Embedded Computing and System Design (ISED)* (2018).

[23] Open Robotics: MultiThreadedExecutor, https://docs.ros2.org/crystal/api/rclcpp/classrclcpp_1_1executors_1_1MultiThreadedExecutor.html.

[24] Open Robotics: rclcpp, http://docs.ros2.org/beta2/api/rclcpp/classrclcpp_1_1executor_1_1Executor.html.

[25] Open Robotics: rclpy, https://docs.ros2.org/latest/api/rclpy/index.html.

[26] Open Robotics: ROS 2 Quality of Service policies, https://design.ros2.org/articles/qos.

[27] Open Robotics: ROS2 Middleware Change Proposal, https://discourse.ros.org/t/ros2-middleware-change-proposal/15863.

[28] Open Robotics: SingleThreadExecutor, https://docs.ros2.org/crystal/api/rclcpp/classrclcpp_1_1executors_1_1SingleThreadedExecutor.html.

[29] Park, J., Delgado, R. and Choi, B. W.: Real-Time Characteristics of ROS 2.0 in Multiagent Robot Systems: An Empirical Study, *IEEE Access* (2020).

[30] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R. and Ng, A.: ROS: an open-source Robot Operating System, *Proc. of IEEE International Conference on Robotics and Automation Workshop on Open Source Software (ICRA)* (2009).

[31] Real-Time Innovations: RTI Connext DDS., https://www.rti.com/products/connext-dds-professional.

[32] Yang, Y. and Azumi, T.: Exploring Real-Time Executor on ROS 2, *Proc. of IEEE International Conference on Embedded Software and Systems (ICESS)* (2020).