Experimental Evaluation of Callback Behavior in ROS 2 Executors

Lukas Johannes Dust
Mälardalen University
Västerås, Sweden
lukas.dust@mdu.se

Emil Persson
Mälardalen University
Västerås, Sweden
emil.persson@mdu.se

Mikael Ekström
Mälardalen University
Västerås, Sweden
mikael.ekström@mdu.se

Saad Mubeen
Mälardalen University
Västerås, Sweden
saad.mubeen@mdu.se

Cristina Seceleanu
Mälardalen University
Västerås, Sweden
cristina.seceleanu@mdu.se

Rong Gu
Mälardalen University
Västerås, Sweden
rong.gu@mdu.se

***Abstract*—Robot operating system 2 (ROS 2) is increasingly popular both in research and commercial robotic systems. ROS 2 is designed to allow real-time execution and data communication, enabling rapid prototyping and deployment of robotic systems. In order to predict and calculate execution times in ROS 2, one needs to analyze its internal scheduler, called executor. The executor has been updated in various distributions of ROS 2, which is shown to impact significantly the periodic execution invoked by the underlying operating system's timers, potentially causing unexpected latencies. To expose the mentioned impact due to executor differences, in this paper, we present an experimental evaluation of the execution behavior of ROS 2's schedulable entities, namely callbacks, among the existing versions of the executor. We visualize the differences of callback execution order via simulation, and we create design-level scenarios that impact the execution of periodically scheduled callbacks, negatively. Moreover, we show how such negative impact can be mitigated by using multi-threaded executors. Finally, we illustrate the observed behavior on a real-world centralized multi-agent robot system. Our work aims to raise awareness within the ROS 2 developer community, regarding possible problems of timer blocking, and propose a mitigation solution of the latter.**

I. Introduction

Robot Operating System 2 (ROS 2) is a middleware that supports the development and deployment of software for robotic systems, ranging from fully decentralized to centralized robotic systems [4, 17, 11]. With the rising need for real-time capabilities in robotic systems, ROS 2 has been developed to replace ROS 1, and meet real-time demands by using Data Distribution Service (DDS) communication [7]. The execution of ROS 2 relies on an underlying operating system (OS). Furthermore, ROS 2 includes an internal scheduler, called *executor*, which schedules ROS 2's smallest schedulable entities, namely *callbacks*. While a system can consist of several instances of executors, knowledge of its execution is essential to meet real-time requirements in ROS 2. Consequently, fellow researchers have conducted work on analyzing ROS 2 execution, including response-time analysis [8, 6]. To improve overall system performance, the executor has changed across different ROS 2 distributions. The change allows schedulable entities released by timers, that is, *timer callbacks*, to be included in the so-called *ready set*. This is important, as the ready set is only updated at specific time points after the executor of ROS 2 becomes idle. The executor schedules all callbacks (of any type) that are included in the ready set only. Prior to this change, timer-triggered callbacks are excluded from the set, and considered instead for scheduling after each execution of any callback, once the respective timer event arises. This change in execution causes issues, especially in periodic systems, as callbacks released by system timers can get blocked by callbacks available in the previously polled ready set. Some effects of the change have also been studied via timing analysis methods, in related work [6, 13, 21]. Assuming systems that rely on periodic execution, the executor change and its consequences need to be identified. This can be done by investigating the differences in the executor versions and the respective scheduling of callbacks, via an experimental evaluation, such that design choices can be made by developers, without assuming worst-case execution time only, as in related work.

Problem Statement and Paper Contributions

A functional model of the first version of the ROS 2 executor is developed by Casini et al. [8], and the change is noted in the literature [6]. Nevertheless, no existing work compares the order of execution of callbacks in different executors in detail. In this paper, we present the differences in the internal scheduling of callbacks over the native ROS 2 versions, by an experimental evaluation. The experiments give insights into the semantics of ROS 2 executors, helping practitioners to decide for an appropriate implementation of real-time robotic systems, being aware of the implications of the different executor semantics. In the experimental evaluation, we consider the following research questions. **RQ1**: How does the scheduling of callbacks in a node differ in the single-threaded executor among the existing ROS 2 distributions? **RQ2**: What type of system configurations lead to timer blocking in the existing single-threaded executor versions, and how does the blocking affect periodic execution? **RQ3**: How do approaches using multi-threaded executors affect the timer blocking while preserving mutually exclusive execution?

The main contributions of this paper are as follows:

1. We identify and also visually show the differences in the execution of callbacks on the native single-threaded executor over different distributions in ROS 2, by conducting various experiments. Furthermore, we show the

critical blocking of timer callbacks, which is possible in the updated version of the executor included in ROS 2 distributions starting from Eloquent on.
2. We provide a solution to the issue exposed above, by employing mutual-exclusive multi-threaded executors in experiments and showing again the differences in scheduling, as well as callbacks grouping in ROS 2 Humble.
3. We illustrate the impact of the system design on the periodic execution of callbacks in the single-threaded executor, on an abstracted real-world centralized multi-agent robot system.

II. Background

This section introduces ROS 2 and its components, followed by an overview of the available ROS 2 distributions. The Robot Operating System (ROS 2) is an open-source meta-operating system for developing robotic applications. Since ROS 2 is not a standalone operating system (OS), it must be installed on top of an existing OS. ROS 2 has been introduced to meet the industrial requirements of guaranteeing fault tolerance, process synchronization, and meeting real-time constraints. Compared to its predecessor ROS 1, ROS 2 adopts the communication middleware called Data Distribution Service (DDS) developed by the Object Management Group (OMG) [2]. Middleware is software that enables connection and communication between two or more applications or application parts, by linking the communication between the applications that are not built to communicate with each other.

A. Robot Operating System 2 Distributions

ROS 2 is constantly improving in versioned sets of ROS 2 packages or so-called *distributions*. Table I lists the distributions used in this paper, the respective default DDS vendor, as well as the version of the executor, respectively.

TABLE I
ROS 2 DISTRIBUTIONS OF INTEREST.

Distro	Release	EOL	Default DDS vendor	Executor
Humble Hawksbill	2022	2027	eProsima Fast RTPS	E2
Galactic Geochelone	2021	2022	Eclipse Cyclone DDS	E2
Foxy Fitzroy	2020	2023	eProsima Fast RTPS	E2
Eloquent Elusor	2019	2020	eProsima Fast RTPS	E2
Dashing Diademata	2019	2021	eProsima Fast RTPS	E1
Crystal Clemmys	2018	2019	eProsima Fast RTPS	E1

B. ROS 2 Systems and Communication

ROS 2 systems are composed of executable *nodes*, where each node consists of one or several callbacks that represent ROS 2's smallest schedulable entity, see Section II-C. Communication between nodes in the included DDS can be performed using *publish-subscribe* or *service-client* communication patterns, see Figure 1. In the *publish-subscribe* pattern, publishers generate data, while subscribers consume the data [5]. The publisher and subscriber discover each other at run-time using the Real-time Publish-Subscribe Protocol (RTPS) [1]. A *topic* describes the data that publishers and subscribers share; the publishers and subscribers only send and receive data on topics of interest. In ROS 2, several publishers may publish on the same topic, and many subscribers can subscribe to the same topic. Subscribers receive data from all

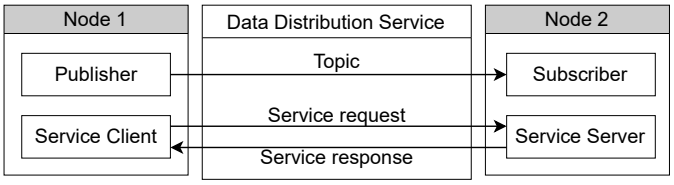

Fig. 1. ROS 2 communication scheme of nodes, topic, publisher, subscriber, service, and client.

publishers related to a specific topic, see Figure 1. Compared to asynchronous publisher-subscriber communication, ROS 2 provides synchronous *service-client* communication. This type of communication uses request-and-response communication among nodes. A *service server*, commonly used to compute a request, responds only when a service client node sends an initial request.

C. Executors and Callbacks

The smallest schedulable entity in ROS 2 is a *callback*, which is scheduled by the *executor*. The executor utilizes one or more threads of the underlying operating system to invoke *callbacks* that belong to one of the four callback types: timers, subscribers, services, and clients. *Timer* callbacks are triggered by events invoked by the operating system's timer, whereas *subscribers* are invoked by new messages on subscribed topics. A *client callback* is invoked when a client makes a service request, while the *service callback* is invoked when a service response to the clients' request is received.

The first analytical model of the executor, which is valid up to the ROS 2 distribution Eloquent, was proposed by Blass and Casini et al. [8]. In their work, the authors explain the execution behavior of ROS 2 Crystal 2018. The second executor version was introduced in ROS 2 Eloquent 2019, and the changes were first identified by Blaß et al. [6]. It is important to note the differences between the two versions of executors, illustrated in Figure 2. In both versions, all callbacks in ROS 2 are non-preemptive, which means that once the execution of a callback starts, it will be completed before the next callback is executed. However, the second executor version brought improvements and changes that warrant further discussion and analysis in this paper. The execution priorities of callbacks are determined based on their type. Callbacks belonging to the group of timers have the highest priority, followed by subscription callbacks and service callbacks. The client callbacks have the lowest priority. Inside each group, the priorities of callbacks are determined by the order of registration in the executor, meaning that, e.g., the first-registered timers are executed before the later-registered timer when both are available simultaneously [8].

Decisions of scheduling in the executor are based on the so-called *ready set*. The *ready set* contains one instance of all topics, services, and client callbacks where data is available in the DDS, i.e., the so-called ready callbacks. When the executor is idle, e.g., when all callbacks in the *ready set* are executed, a ready set is built in both versions of the executor. The point where the ready set is built is called a polling point. The difference between execution models resides in how timers are handled. In the first version of the executor, timers are excluded from the ready set and scanned for every iteration when any callback instance finishes its

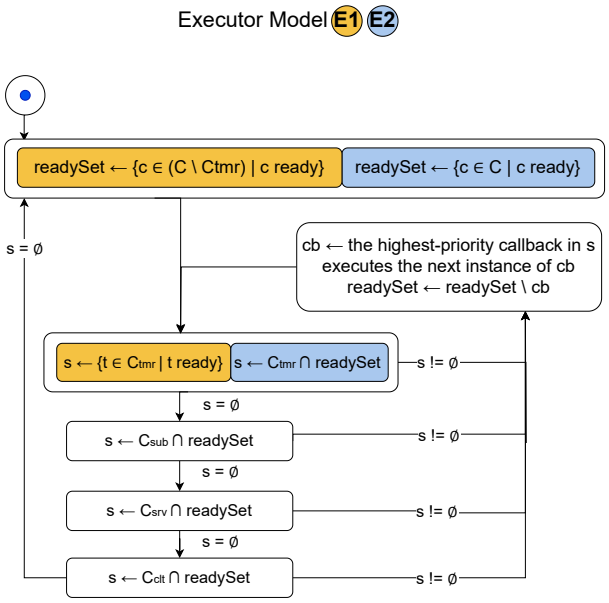

Fig. 2. Version 1 (E1) for the ROS 2 distributions earlier than Dashing Casini et al. [8] and 2 (E2) for later distributions of ROS 2 (Eloquent - Humble) of the executor model. C represents a set of callbacks with an instance c, where tmr denotes a timer, sub the subscriber, srv the server, and clt the client callbacks. By s we denote the subset of a ready set, and by cb the highest priority callback in s.

execution. In the updated version of the executor, starting with ROS 2 Eloquent, the timers are included in the ready set [6].

The prioritization and execution of the callbacks in multi-threaded executors (introduced in ROS 2 Foxy) follow the same procedure as shown in Figure 2, except that the execution of callbacks is distributed to several system threads [13]. That leaves two options for operating callbacks, that is (a) mutual exclusion, meaning that all callbacks in the ready set have to be processed after the previous callback finishes its execution, or (b) reentrant, meaning that all callbacks in the ready set can be executed in parallel. In an executor using option (a), multiple threads cannot operate callbacks from the same callback group in parallel, but operate the idle operation of updating the ready set, which increases the number of polling points.

III. Related Work

This section gives an overview of related works on timing analysis of ROS 2, alternative executor design, real-time capabilities of ROS 2, and systems and communication design. The first work mentioning and stating the ROS 2 execution model was conducted by Casini et al. [8] in 2019. By analyzing the execution of the ROS 2 distribution Christal Clemmys, the executor model was identified and described. A response-time analysis, calculating the worst-case response time was proposed under reservation-based scheduling. Another work regarding response-time analysis, considering the priorities of the ROS 2 callbacks, based on the first executor version was conducted by Tang et al. [20]. In the paper by Blaß et al. [6] the actual timer as a part of the ready set was considered. An expanded analysis under real-time scheduling for multi-threaded executors has been conducted by Jiang et al. [13]. This work proposes response-time analysis for processing chains using multi-threaded executors. The latest paper in the field by Teper et al. [21] proposes an end-to-end timing analysis on a single-threaded executor, focusing on cause-effect chains. The named works are significant for the research we present in this paper. However, actual execution differences were not compared, which are addressed in our work. Furthermore, our work expands the knowledge about timer blocking in periodic systems and how the system design influences execution.

In addition to research on explaining and analyzing the execution of ROS 2 in the native distributions, work was conducted on implementing new executor algorithms. Choi et al. [10] propose a new priority-driven executor and provide analysis in terms of end-to-end latency, which is improved compared to the native ROS 2 executor. Arafat et al. [3] propose a new executor scheduling algorithm based on deadline-based scheduling and carry out a response time analysis. Yang and Azumi [22] replace the executor with the real-time executor from a scaled-down version of ROS 2 built for microcontrollers. While the previously stated works focus on creating new executors, our work is related to explaining the native ROS 2 scheduling and executor instead. Analyzing the real-time characteristics and performance of ROS 2, Choi et al. [10] focuses more on the system and communication levels including the underlying OS, that is neglected in our work. Measuring performance in terms of transport latencies in the transportation layer of ROS 2 is conducted by Maruyama et al. [16], C. S. V. Gutiéerrez [7], and Kronauer et al. [14]. Regarding communication architectures and performance using DDS, several works consider these within the context of ROS 2, for instance, Puck et al. [18] [19] for real-time communication in distributed systems. Additional analysis of system performance, assuming different quality-of-service settings in the DDS, is proposed by Chen [9].

Further analysis on DDS vendors and the performance in centralized multi-agent robot systems was carried out by Dust et al. [11]. Z. Li, A. Hasegawa, T. Azumi [23] have proposed a framework for measuring system performance and callback execution time. The work from Erős et al. [12] investigates and states different possible communication architecture approaches for multi-agent robot systems using ROS 2. However, neither performance analysis nor implications to systems' design are drawn. A time disparity analysis for message synchronization has been proposed by Li et al. [15], which helps evaluate data consistency in systems using sensor fusion. Above stated performance measures, tools, and advanced analysis methods are essential to provide a more holistic analysis of existing systems in the future.

IV. Experimental Evaluation of Execution Behavior

In this section, we present four experiments and their results to evaluate and visualize the execution differences in the existing executors. An overview of the experiments can be found in Table II. In Exp1, a simple scenario is created on a single-threaded executor, using multiple subscribers and two single-shot timers, each triggered once. Throughout the experiment, the same scenario is repeated across the ROS 2 distributions Dashing, Eloquent, and Humble, respectively. In Exp2, experiments are carried out for the three distributions under consideration on a single-threaded executor creating a

new scenario by using periodic timers. This scenario demonstrates the blocking in the execution of periodic systems. In Exp3 and Exp4, we conduct experiments in the same scenario as Exp2 utilizing periodic timers, but using multi-threaded executors. The difference between Exp3 and Exp4 is the usage of callback grouping in Exp4, where only callbacks assigned to the same group share the same instance of the executor. As the multi-threaded executors and callback groups have been introduced in more recent versions of ROS 2, our evaluation specifically targets ROS 2 Humble. By focusing on this latest distribution, we can better assess the performance and characteristics of these features in the most up-to-date ROS 2 distribution.

TABLE II
OVERVIEW OF THE PERFORMED EXPERIMENTS

Experiment	Executor settings	ROS 2 Distributions	Timer mode
Exp1	single-threaded	Eloquent, Dashing, Humble	single shot
Exp2	single-threaded	Eloquent, Dashing, Humble	periodic
Exp3	multi-threaded mutually-exclusive	Humble	periodic
Exp4	multi-threaded mutually-exclusive callback-grouping	Humble	periodic

A. Experimental Setup

All the experiments in this paper are conducted on a single computer using docker, to create a controlled environment and enable simple switching between the different ROS 2 distributions. The ROS 2 distributions under consideration and their respective native DDS are as follows:

1) ROS 2 Humble, latest distribution of ROS 2, E2
2) ROS 2 Eloquent, first distribution to introduce E2
3) ROS 2 Dashing, last distribution that contains E1

We create a simple ROS 2 system consisting of two nodes, four topics, and three services, which are used throughout all four experiments. An overview of the evaluation system can be found in Figure 3. The left-side node in Fig. 3 is defined as the control node and is used to control the experiment, by sending the start message and requesting service sequences at defined times. The right-side node in Fig. 3 is the experiment node, where the execution order of the callbacks is tracked. Topics and services in the system are called after their priority, which is determined by the order of registration. The topics are called H, M, and L (High, Medium, Low), and services SH, SM, and SL (Service High, Service Medium, Service Low). Furthermore, one *control* topic is created to start and control the experiment. A message sent over the control topic triggers the registration of the needed timers in the experiment node.

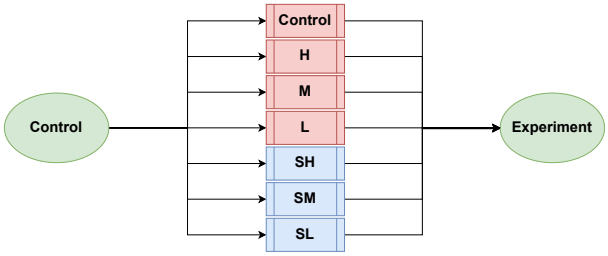

Fig. 3. Simple ROS 2 system to verify the executor models. Green ellipses represent nodes, red boxes represent topics and blue boxes depict service communication. The Control node provides the publishers and service clients, while the experiment node provides the subscribers and the services service.

Each subscriber and each client in the experiment node are connected to a unique callback that publishes its execution information in the ROS 2 log. For simplicity and better execution analysis, the callback execution time in the experiment node (except for the control topic callback) is artificially programmed to be 500 ms. However, the lengths of periods do not influence our conclusion of experimental evaluation. Each experiment consists of a different timer, message and service sequences to be released/requested at specified time points from the control node and executed in the experiment node. The same source code runs in all three different ROS 2 distributions under consideration, without any changes. All experiments are performed on the same computer, using Ubuntu 20.04.4 LTS with an Intel Xeon E5-1660 CPU, NVIDIA Quadro K2000 GPU and 32GiB RAM.

B. Timers on Single-threaded Executor

The significant difference between the two versions of the single-threaded executor (E1 and E2 in Fig. 4 and Fig. 5) is the handling of the timers. The differences in the execution behavior in the ROS 2 distributions are visualized by conducting the experiments described in the following.

1) Experimental Scenarios: We set up two scenarios for the experiments. The *first* scenario (SC1) used in Exp1 utilizes single-shot timers that only execute once, intending to introduce differences in execution in a simple manner. Furthermore, the first experiment only contains two timers and utilizes a message sequence using only the three topics, H, M, and L, released at the scenario's beginning.

In the *second* scenario (SC2) used in Exp2,3 and 4, a periodic timer with a given period is initialized, the scenario containing four message sequences released at different times. The scenario utilizes the topics H, M, and L and the services SH, SM, and SL. The scenarios (SC1 and SC2) and the corresponding experiments are specified in Table III, where timers are denoted by T0 and T1, and sequences by S0, S1, S2, and S3. In Exp1 and Exp2 respectively, both scenarios are executed using the ROS 2 distributions Dashing, Eloquent and Humble, using the single-threaded executor (E1 and E2).

TABLE III
EXPERIMENT SETUP SHOWING THE MESSAGE SEQUENCES (S) AND TIMERS (T) AND THEIR RESPECTIVE RELEASE TIME AND PERIOD. RT: RELEASE TIME. SEQ: MESSAGE SEQUENCE.

			Event					
			S0	S1	S2	S3	T0	T1
Exp1	SC1	RT	0	-	-	-	0.2	2.3
		Seq	H;M;L;H;M;L	-	-	-	-	-
Exp2, 3, 4	SC2	RT	0	3.2	4.5	6.3	1.3	-
		Seq	H;M;L;SH;SL	H;M;L	SH;SM	H	-	-

2) Results and Discussion: The execution behavior shown in Figure 4 is observed when executing Exp1. The figure includes the execution of the corresponding callbacks to each topic and timer, respectively. Furthermore, it visually presents the release time of the timers and the polling point for the update of the ready set, respectively. The polling points and the corresponding ready set of Exp1 can be found in Table IV. Note that the execution behavior differs for ROS 2 Dashing using E1, as compared to Eloquent and Humble, using E2.

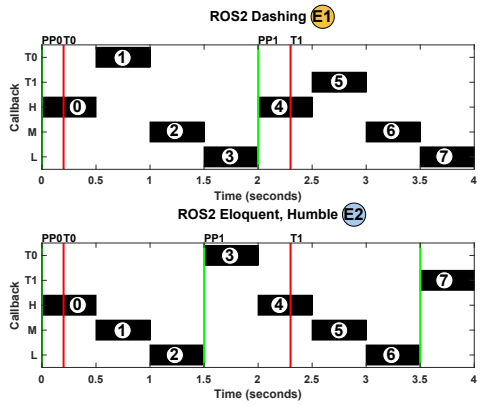

Fig. 4. Exp1: execution diagram of the given callbacks for subscribers (H, M, L) and timers (T0, T1). Red line: activation time of the timer callbacks. Green line: the polling points (PP) in the system.

TABLE IV
POLLING POINTS AND READY SET IN EXPERIMENT 1

	Polling Instance	PP0	PP1	PP2
E1	Time	0	2	-
	Ready Set	H;M;L	H;M;L	-
E2	Time	0	1.5	3.5
	Ready Set	H;M;L	T0;H;M;L	T0

In both E1 and E2, the first ready set is created at the start of the experiment. The highest-priority task in the ready set, i.e., the H topic callback, is executed first (⓪). During the execution of this callback, the first timer is released (first red lines). Now the first difference between E1 and E2 can be observed. After finishing the execution of the first callback, E1 immediately schedules the timer callback to execution, whereas, in E2, the next element of the ready set (M-callback) is executed (①). E2 executes all remaining callbacks in the ready set (①, ②) before polling the system to create a new ready set (③ - ⑦ after the green line), which contains the released timer during the execution of the first callback (③), whereas in E1, the executor finishes the execution of the timer callback (①) before it schedules the rest of the callbacks in the ready set (②, ③) and then polls the system to create the second ready set (green line). During the execution of the first callback of the second ready set (④), Timer 2 is expiring (second red lines). The difference in scheduling shown in the first ready set can be seen here too, that is, E1 schedules T1 for execution immediately after finishing the execution of the running callback (⑤ right after ④), and then E1 continues executing the callbacks of the ready set. In E2, T1 is executed after emptying the ready set (⑦ right after ⑥). These phenomena confirm the timer to be part of the ready set in E2.

Answer to **RQ1**: When a timer is released during an execution of a callback and its ready set has remaining callbacks, ROS 2 Dashing, which uses the E1 version of executors, immediately schedules the timer callback after the current callback, whereas ROS 2 Eloquent and Humble, which use the E2 version of executors, continue executing the remaining callbacks in the set before scheduling the timer callback.

Exp2 is conducted in scenario SC2 and the resulting execution diagram for Exp2 can be found in Figure 5. The resulting ready set for all polling points can be found in Table V.

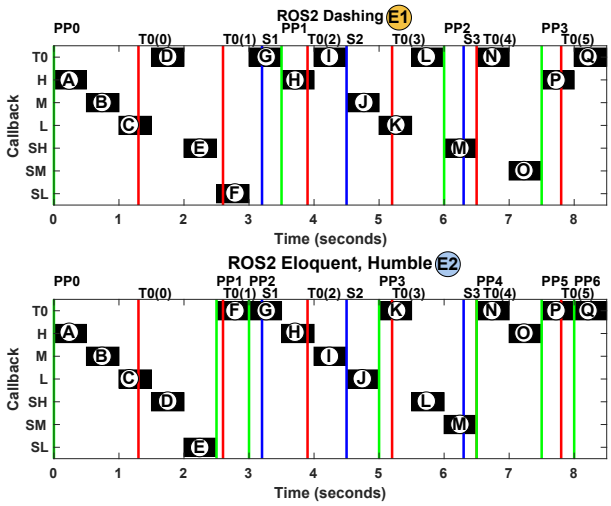

Fig. 5. Exp2: execution diagram of the given callbacks for subscribers (H, M, L) and services (SH, SM, SL), and timer T0. Red line: expiration time of the timers. Green line: polling points (PP) in the system. Blue line: release of message sequences.

TABLE V
POLLING POINTS AND READY SET IN EXP2

	Polling Point	PP0	PP1	PP2	PP3	PP4	PP5	PP6
E1	Time	0	3.5	6	7.5	-	-	-
	Ready Set	H;M;L;SH;SL	H;M;L	SH;SM	H	-	-	-
E2	Time	0	2.5	3	5	6.5	7.5	8
	Ready Set	H;M;L;SH;SL	T0	T0;H;M;L	T0;SH;SM	T0;H	T0	T0

In scenario SC2, the periodic timer T0 is released every 1.3 seconds. As depicted in Fig. 5, E1 in ROS 2 Dashing executes all Timer instances directly after the previously executing callback is finished (e.g., Ⓓ right after Ⓒ). Therefore, the maximum blocking of a timer in E1 must be the time of the longest callback. For E2 in ROS 2 Eloquent and Humble, the scheduling is different. The timers are firstly scheduled for execution after the next ready set is created after release time (e.g., Ⓕ after Ⓔ). This can cause a long blocking time of a periodic timer. In Fig. 5 E2, one can see that the first instance of timer T0 is released at 1.3 seconds (T0(0), first red line), but the execution is blocked until 2.5 seconds (Ⓕ), which is shortly before the second instance of timer (T0(1), second red line). Such a long blocking is due to the fact that callbacks Ⓒ - Ⓔ, which are in the ready set when the first instance of T0 is released, must be finished before T0 can be scheduled. T0(1) is released at 2.6 seconds after a new creation of the ready set (PP1, first green line), which means T0(1) is released during the execution of T0(0). In this case, a consecutive execution of two timer instances takes place, which could be highly undesired. Therefore, system designers need to be aware of such a phenomenon and avoid potential faults. However, as far as we know, this paper is the *first one* that reveals the differences in scheduling between E1 and E2 in experiments. Moreover, we can see that in the following scheduling of E2, new instances of the timer are blocked again, for 1.2 and 1.5 seconds, followed by two consecutive executions of the timer callback, again (Ⓠ and Ⓡ). Therefore, with a poor

system design, blocking of periodic timers appears, which causes them to run sporadically, even though they are set to run periodically. In the worst case, the blocking of timers in E2 can be the sum of the execution time of all callbacks in the system, which is likely to cause significant degradation in system performance.

> Answer to **RQ2**: Timer blocking can be significantly different in ROS 2 Dashing, Eloquent, and Humble, as it depends on the number of callbacks in the ready set. Therefore, more callbacks of any type can lead to a higher blocking time. In ROS 2 Dashing, timer blocking is slight and dependent on the execution time of the previous executing callback and higher priority timers, whereas ROS 2 Eloquent and Humble potentially have long timer blocking leading even to consecutive occurring timer callbacks, which makes periodic timer callbacks occur irregularly.

As timers do not contain buffers in ROS 2, if blocking is longer than two periods of a timer, one instance of the timer would be skipped. This issue cannot be solved on a single-threaded executor. Hence, it needs to be regarded when designing systems using ROS 2, especially if updating the system with a different ROS 2 distribution is planned. In the following, we show that a multi-threaded executor setting could be regarded as a mitigation solution for the previously exposed unpredictable behavior.

C. Periodic Timers on Multi-threaded Executors

In this section, we evaluate the multi-threaded executor via Exp3 and Exp4 to present a meaningful comparison to the single-threaded executor. The two designed experiments only focus on the mutually-exclusive executor and callback grouping, as the execution behavior is closest to the model of the single-threaded executor. Since multi-threaded executors and callback groups have been added to ROS 2 during recent releases, the experiments are only carried out on the ROS 2 distribution Humble.

1) Experiment Setup: Both Exp3 and Exp4 use scenario SC2, which can be found in Table III. While Exp3 only uses the multi-threaded executor, Exp4 uses callback grouping. By callback grouping, callbacks can be assigned to different groups, and each group has its own instance of the executor. In Exp4, the timer is assigned one callback group, while all the other callbacks are grouped in another callback group.

2) Results and Discussion: Figure 6 shows the resulting execution trace from Exp3 and Exp4, respectively. The order of execution of the callbacks in the experiment node is visualized. Furthermore, the timer and the sequence release times are marked as well.

Using the mutually-exclusive multi-threaded executor in Exp3, the timer is executing periodically, with a jitter caused by the callback that is under execution at the release time (e.g., the gap between ⓓ and T0(0), the first red line). The scheduling policy in Exp3 follows that of the single-threaded executor E2, which includes timers in the ready set, but the actual scheduling result is almost similar to E1. However, E1 in Exp2 (E1 in Figure 5) schedules the H callback (Ⓠ) at time point 7.5, *after* the SM callback (Ⓟ), whereas in Figure 6 Exp3, the H callback (ⓞ) is scheduled at time point 7.0, *before*

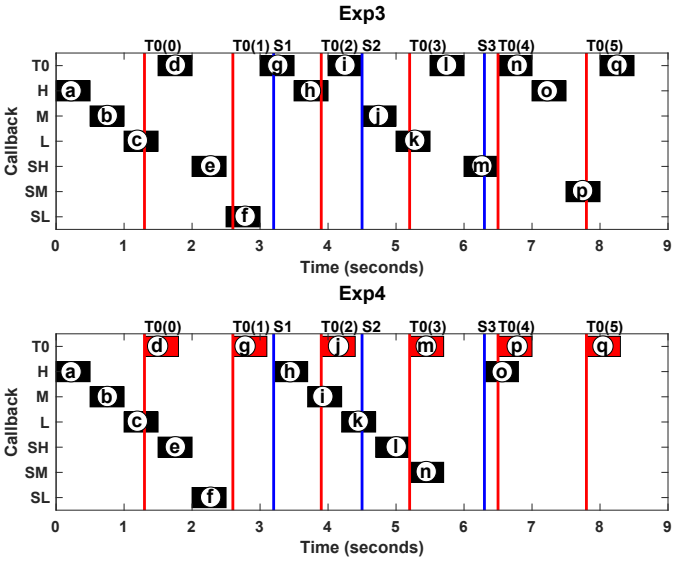

Fig. 6. Execution diagram of the given callbacks for subscribers (H, M, L) and timers (T1, T0) in Exp3 and Exp4. Red line: expiration time of the timers. Blue line: release of message sequences.

the SM callback (ⓟ). This execution difference shows that the multi-threaded executor in Exp3 polls more frequently than the single-threaded executor in Exp2, and the ready set is updated during the execution of callbacks. As the executor is set to be mutually exclusive, once a ready set is assigned to a thread, other threads cannot preempt the set and execute other callbacks, but they can update a ready set by adding later released callbacks.

In the execution trace of Exp4, Figure 6, we mark the callbacks of the timer in red, as they are assigned a different callback group than other callbacks. The mutual-exclusive property is only valid within a callback group and each callback group has its own ready set, which means the multi-threaded executor can execute the timer in parallel to other callbacks belonging to the other callback group. Additionally, as there are no other callbacks in the timer's callback group, the timer is executed as soon as it is released. As expected, in both experiments using the mutually exclusive multi-threaded executor, the blocking issue seen from the single-threaded executor is resolved.

> Answer to **RQ3**: By using multi-threaded executors in ROS 2 Humble, timer blocking is largely alleviated or even eliminated, compared to the single-threaded executor. Even without callback grouping, timer blocking is reduced to higher priority timers and the length of the previously executed callback. When using callback grouping, timer callbacks can be scheduled in parallel to the execution of other callbacks, as timers can be scheduled as soon as they are released. Nevertheless, mutually exclusive execution is no longer possible using the callback grouping option.

V. Use-Case: Centralized Multi-Agent Robot System

This section focuses on a real-world use case by building an experimental system around the centralized multi-agent robot system. In such systems, multiple robots communicate towards a central node, that is, the robot tracker, via the so-called

topics that are elements of ROS 2 communication by which data is moved between nodes. We investigate the impact of system design choices on the communication between multiple agents towards a centralized computing node that tracks robot information. We focus only on the design possibilities of the robot's communication towards the edge computer using ROS 2 and show the effects this communication has on the computation inside the tracker node.

A. Experiment Setup

The experiment system is based on a real-world legacy system. A centralized multi-robot system is built around central servers that manage the intensive processing for robots with limited computing capacity. The robots connect to the server and continuously update essential information about themselves, respectively, allowing global fleet management to control each robot. In the selected use case, the robots send their status information, including odometry and load data, to the centralized node periodically with a configurable period. The task of the central node is to collect all incoming data creating a list, that is sent out periodically, for further processing. For simplicity, this paper neglects all other parts of the system. While building such a centralized communication in ROS 2, three different communication approaches are possible, as shown in Figure 7.

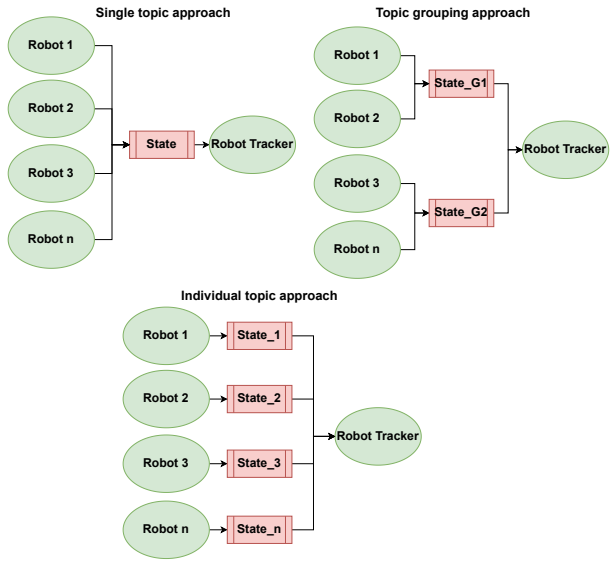

Fig. 7. Overview of the different communication configurations. Green ellipses represent nodes, and red boxes represent topics.

In the first approach (at the bottom), each robot registers its topic and, hence, receives a subscription callback in the centralized node. The second approach (on the upper left side) uses only one topic for all of the robots' state information, leading to only one subscription callback in the edge node. The third approach (on the upper right side) combines the first two approaches, where robots are grouped, with each group receiving a particular topic. Each approach uses a different amount of callbacks in the centralized node, which influences the periodicity of the centralized sending of the list of status information. Assuming the use-case and the ROS 2 system described at the beginning of this section, we perform measurements to show the impact of the system design on the execution. The experiment focuses on the tracker's periodic sending of the status list. To show how the communication approach influences the period of execution, the system is simulated with 10, 30, 50, and 70 robots, respectively. The period is manually configured to be 1 ms, being under the system's limit when the system contains 50 or 70 robots. In the first step, the system is simulated with one topic for all robots. In the second step, the robots are grouped into 10, 5, and 2 robots per topic. The last part of data collection is then carried out, with each robot having a particular topic. Each measurement consists of 500 iterations of sending the state list. Timestamps are taken and analyzed to calculate the average period. As the legacy system is implemented using ROS 2 Galactic and real system messages are considered, the experiments are performed on ROS 2 Galactic utilizing Cyclone DDS. There might be differences in the obtained periods depending on the network traffic and chosen DDS vendor. Nevertheless, the experiment is designed to keep the impact of communication minimal relative to the impact of the scheduling. The used hardware is the same as the one used during experiments one to four. For simplicity, all nodes are simulated on the same computer, which reduces other influences by a real network.

B. Results and Discussion

We present the results of the above measurements in Table VI.

TABLE VI
AVERAGE UPDATE PERIOD OF THE STATE LIST OVER 500 MEASUREMENTS.

		Total amount of Robots			
		10	30	50	70
Robots per topic	1	1	1.14	2.16	4.18
	2	1	1	1.78	3.08
	5	1	1	1.42	2.92
	10	1	1	1.32	2.78
	Max		1	1.27	2.26

In the first simulation with ten robots, we can see that the increase in topics does not influence the periodicity. Therefore, the callback execution time and the number of topics are not significant enough to block the timer and delay the execution. For 30 robots, the period is set to 1 ms, except for the approach where each robot has its topic (one robot per topic), so 30 topics and 30 reception callbacks exist in the tracker node, which is blocking the execution of the timer. In other cases, the impact is not significant enough to block the timer.

In the next simulation with 50 robots, one can see that already during the execution with only one topic (max robots per topic), the period is not set to 1, but it is 1.27 ms instead. This can be explained by the fact that the timer and receiver callback takes longer with more robots to create, sort, or send out the array. In this case, 1.27 ms is the maximum frequency, which the edge node can guarantee if the timer sends out the data array periodically. When we increase the number of topics, one can observe that the period is growing to 2.16 ms, with one robot per topic (50 topics and subscription callbacks). That clearly shows that the timer is blocked in its execution by the callbacks, and the period rises. The same effect can be seen in the simulation with 70 robots, where the maximum frequency with one topic is 2.26 ms, and the period increases to 4.18 ms for 70 topics. In brief, we show that in cases when the update period is manually set to the system's limit,

the execution gets increasingly delayed with the number of callbacks. This phenomenon can be explained with the current execution models, as blocking of the timer occurs.

VI. Discussion and Conclusion

The experimental work conducted in this paper compares the execution of callbacks in the different executors in ROS 2.

During the experiments and in the stated use-case, possible effects of the underlying operating system, network traffic and quality-of-service settings are neglected. Further analysis needs to be performed, including communication, to allow evaluation on holistic system performance.

From proposed response time analysis in related work, the timer blocking can be calculated, leading the result of this paper not to be surprising. However, the results give a good overview of the internal execution of ROS 2 with ideal underlying scheduling. Generally, the obtained scheduling semantics are hardware independent. Nevertheless, the execution of ROS 2 relies on the underlying OS as well, which might influence the obtained results in cases of limited resource availability. For ROS 2 developers and even researchers, this paper introduces the different existing executors and shows the effects of system design on execution. We show in a simple way how the different executors influence the execution order of callbacks, which can guide developers to the right system design choices without the need to perform worse-case response-time analysis for their system.

We conclude our experimental investigation as follows. In E1, included up to ROS 2 Dashing, through continuously polling for timers, a timer can only get blocked until the previous computing callback is finished. In comparison, in E2, through the inclusion of the timers into the ready set, a timer callback is blocked until the execution of all callbacks of the previous ready set is finished. With a higher amount of ready callbacks, the blocking time gets longer easily blocking a timer into its subsequent execution. This is critical, as timers have no buffer and instances can get lost. Using a mutually-exclusive multi-threaded executor reduces the blocking of the timer to the execution of the previous computing callback. Due to the mutual exclusion, a second thread is not computing any callbacks, but updating the ready set instead. The execution jitter of the timer can be reduced to a minimum, by utilizing callback grouping, only adding the timer callback to a specific group. Nevertheless, the callback grouping shows a reentrant execution from the perspective of the node.

The provided use-case in this paper demonstrates that caused by the executor, a chosen communication architecture influences the execution of a system. Especially in cases of high utilization, the choice of callback organization becomes significant. When designing systems that rely on periodic execution, one needs to keep the number of topics to a minimum to reduce the amount of callbacks in the executor. Alternatively the multi-threaded executor options need to be considered. Especially when systems are built to scale in the number of agents, design compromises need to be made with system complexity, resource utilization, and system performance.

In a subsequent step to this paper, formal verification can be employed, to help address and prevent the timer issue more straightforwardly. Furthermore, towards real-time execution, work can be conducted to improve the ROS 2 executor by implementing other design opportunities, e.g., dynamic settings for the ready set.

References

[1] The real-time publish-subscribe protocol, 2022. URL https://www.omg.org/spec/DDSI-RTPS/2.3/Beta1/. Accessed: 22-11-7.

[2] Object management group, 2022. URL https://www.omg.org/. Accessed: 2022-11-7.

[3] Abdullah Al Arafat, Sudharsan Vaidhun, Kurt M Wilson, Jinghao Sun, and Zhishan Guo. Response time analysis for dynamic priority scheduling in ros2. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 301–306, 2022.

[4] Maximilian Berndt, Dennis Krummacker, Christoph Fischer, and Hans Dieter Schotten. Centralized robotic fleet coordination and control. In *Mobile Communication - Technologies and Applications; 25th ITG-Symposium*, pages 1–8, 2021.

[5] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *Proceedings of the eleventh ACM Symposium on Operating systems principles*, 1987. doi: 10.1145/41457.37515.

[6] Tobias Blaß, Daniel Casini, Sergey Bozhko, and Björn B Brandenburg. A ros 2 response-time analysis exploiting starvation freedom and execution-time variance. In *IEEE Real-Time Systems Symposium*, pages 41–53. IEEE, 2021.

[7] C. S. V. Gutiérrez. Towards a distributed and real-time framework for robots: Evaluation of ros 2.0 communications for real-time robotic applications. *Tech. Rep*, 2018.

[8] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn Brandenburg. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems*, pages 1–23, 2019.

[9] Zhaolin Chen. Performance analysis of ros 2 networks using variable quality of service and security constraints for autonomous systems. Technical report, Naval Postgraduate School Monterey United States, 2019.

[10] Hyunjong Choi, Yecheng Xiang, and Hyoseung Kim. Picas: New design of priority-driven chain-aware scheduling for ros2. In *IEEE 27th Real-Time and Embedded Technology and Applications Symposium*, pages 251–263. IEEE, 2021.

[11] Lukas Dust, Emil Persson, Ekström Mikael, Mubeen Saad, and Dean Emmanuel. Quantitative analysis of communication handling for centralized multi-agent robot systems using ros2. In *IEEE International Conference on Industrial Informatics*, 2022.

[12] Endre Erős, Martin Dahl, Kristofer Bengtsson, Atieh Hanna, and Petter Falkman. A ros2 based communication architecture for control in collaborative and intelligent automation systems. *Procedia Manufacturing*, 38, 2019.

[13] Xu Jiang, Dong Ji, Nan Guan, Ruoxiang Li, Yue Tang, and Yi Wang. Real-time scheduling and analysis of processing chains on multi-threaded executor in ros 2. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 27–39, 2022. doi: 10.1109/RTSS55097.2022.00013.

[14] Tobias Kronauer, Joshwa Pohlmann, Maximilian Matthé, Till Smejkal, and Gerhard Fettweis. Latency analysis of ros2 multi-node systems. In *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*, 2021.

[15] Ruoxiang Li, Nan Guan, Xu Jiang, Zhishan Guo, Zheng Dong, and Mingsong Lv. Worst-case time disparity analysis of message synchronization in ros. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 40–52, 2022. doi: 10.1109/RTSS55097.2022.00014.

[16] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *2016 International Conference on Embedded Software*, pages 1–10, 2016.

[17] Samyeul Noh and Junhee Park. System design for automation in multi-agent-based manufacturing systems. In *20th International Conference on Control, Automation and Systems*, pages 986–990, 2020.

[18] L. Puck, P. Keller, T. Schnell, C. Plasberg, A. Tanev, G. Heppner, A. Roennau, and R. Dillmann. Performance evaluation of real-time ros2 robotic control in a time-synchronized distributed network. In *17th International Conference on Automation Science and Engg.*, 2021.

[19] Lennart Puck, Philip Keller, T Schnell, Carsten Plasberg, Atanas Tanev, Georg Heppner, Arne Roennau, and Rüdiger Dillmann. Distributed and synchronized setup towards real-time robotic control using ros2 on linux. 10 2020.

[20] Yue Tang, Zhiwei Feng, Nan Guan, Xu Jiang, Mingsong Lv, Qingxu Deng, and Wang Yi. Response time analysis and priority assignment of processing chains on ros2 executors. In *IEEE Real-Time Systems Symposium*, pages 231–243, 2020.

[21] Harun Teper, Mario Günzel, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen. End-to-end timing analysis in ros2. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 53–65, 2022.

[22] Yuqing Yang and Takuya Azumi. Exploring real-time executor on ros 2. In *IEEE International Conference on Embedded Software and Systems*, pages 1–8, 2020.

[23] Z. Li, A. Hasegawa, T. Azumi. Autoware-perf: A tracing and performance analysis framework for ros 2 applications. *Journal of Systems Arch.*, 123, 2022.