Reconciling ROS 2 with Classical Real-Time Scheduling of Periodic Tasks

Harun Teper*, Oren Bell†, Mario Günzel*, Chris Gill† and Jian-Jia Chen*‡
*TU Dortmund University, Germany, †Washington University at St. Louis, USA ‡Lamarr Institute, Germany
{harun.teper, mario.guenzel, jian-jia.chen}@tu-dortmund.de, {oren.bell, cdgill}@wustl.edu

Abstract—The Robot Operating System 2 (ROS 2) is a widely used middleware that provides software libraries and tools for developing robotic systems. In these systems, tasks are scheduled by ROS 2 executors. Since the scheduling behavior of the default ROS 2 executor is inherently different from classical real-time scheduling theory, dedicated analyses or alternative executors requiring substantial changes to ROS 2 have been developed.

In 2023, the events executor was introduced into ROS 2. It features an events queue and allows the possibility to make scheduling decisions immediately after a job is completed. In this paper, we show that with minor modifications of the events executor, a large body of research results from classical real-time scheduling theory becomes directly applicable to ROS 2. This enables analytical bounds on the worst-case response time and the end-to-end latency, outperforming bounds for the default ROS 2 executor in many scenarios. Our solution is easy to integrate into existing ROS 2 systems since it requires only minor modifications of the events executor, which is natively included in ROS 2. The evaluation results show that our ROS 2 events executor with minor modifications can have significant improvement in terms of dropped jobs, worst-case response time, end-to-end latency, and performance compared to the default ROS 2 executor.

Index Terms—Robot Operating System 2 (ROS 2), Priority-Based Executor Scheduling, Compatibility with Classical Real-Time Scheduling Theory

I. Introduction

The Robot Operating System 2 (ROS 2) is a widely used middleware for creating robotics applications. It provides tools and libraries to build modular systems with many interacting components. In comparison to the original Robot Operating System (ROS), it offers opportunities to configure real-time properties through its use of the Data Distribution Services (DDS) for real-time communication and its custom scheduling abstraction, called an executor, that manages the execution of time-triggered and event-triggered tasks.

The default scheduling mechanism in ROS 2 relies on polling points and processing windows. That is, at each polling point, jobs that are eligible for execution are moved to a wait set, followed by a processing window in which the jobs in the wait set are scheduled non-preemptively by a predefined priority order. Due to this round-robin-like approach, even tasks with high priority may experience long blocking times. That is, a high-priority task may be blocked for the length of a full processing window—including the execution time of every lower-priority task. Moreover, the implementation of ROS 2 only releases one job of a task, even if the task period has been reached several times during one processing window.

There have been several dedicated analyses for the default ROS 2 executor [6], [8], [34], [36], [38]. To mitigate the potentially long response times of the ROS 2 default executor, alternative executor designs have been proposed for both fixed-priority schedulers [10] and dynamic-priority schedulers [1].

In 2023, the events executor [23] was introduced in ROS 2, replacing the wait set with an events queue and passing only one job to the processing window at a time. This paper addresses the following fundamental question:

> Is it possible to use the events executor in ROS 2 in a manner that is compatible with the classical real-time scheduling theory of periodic task systems, in which every high-priority task can only be blocked by at most one lower-priority task, and a job is released every time the task period is reached?

Our answer is yes, but only under certain conditions.

In this paper, we uncover these conditions and show that any priority-based, non-preemptive scheduling strategy, with slight modifications of the events executor, can be realized, including Fixed-Priority (FP) and Earliest-Deadline-First (EDF) scheduling. Our solution is easy to apply since it requires only minor backend modifications of the events executor, which is natively included in ROS 2. With our solution, classical results from the real-time systems literature for priority-based, non-preemptive scheduling of periodic tasks can be applied.

Our Contributions: The contributions of this paper are:

- We present modifications to allow priority-based scheduling for the events executor in Section V.
- We state the conditions that make our proposed executor compatible with priority-based, non-preemptive scheduling theory for periodic tasks in Section VI. Furthermore, we demonstrate how to apply analytical results in the literature to compute the worst-case response time and the end-to-end latency for the proposed ROS 2 scheduler.
- While the previous sections focus on timer tasks (i.e., tasks with periods), we extend to subscription tasks (i.e., tasks triggered by other tasks) in Section VII and

discuss the need for dedicated interfaces enabling the prioritization of subscriptions in ROS 2.

- In Section VIII, we evaluate our findings, showing our compatibility with the classical real-time scheduling theory and our proposal's benefits. We show the applicability of response-time bounds from the literature and significant improvements to end-to-end latencies in many cases.

In Section II, we outline the different characteristics of typical priority-based schedulers and the default ROS 2 executor. The events executor is introduced in Section III. We define the problem this paper investigates in Section IV.

II. Classical Schedulers and Executors in ROS 2

In this section, we introduce the scheduling mechanisms of periodic tasks in classical real-time scheduling and the default ROS 2 executor and highlight their differences. To that end, we use a running example of three tasks τ_1, τ_2 and τ_3. Task τ_1 should release a job every 10 time units, with an execution time of 3 time units. Tasks τ_2 and τ_3 should release a job every 30 time units, with an execution time of 10 time units.

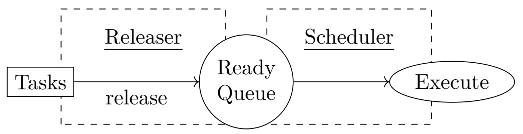
A. Classical Priority-Based Scheduler

The scheduling mechanism of the classical real-time scheduler is depicted in Figure 1a. Specifically, it consists of a releaser[1] and a scheduler, where the releaser inserts jobs into the ready queue and the scheduler decides which job in the ready queue should be executed.

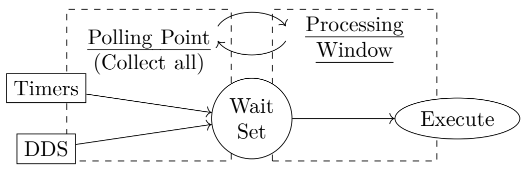
The releaser can be implemented as a timer interrupt service routine. Specifically, a timer interrupt is triggered for each system tick, and the releaser decides whether to release a job of task $\tau_i \in \mathbb{T}$, given a set $\mathbb{T}$ of tasks. A periodic task $\tau_i \in \mathbb{T}$ is specified by the tuple $\tau_i = (C_i, T_i, D_i, \phi_i) \in \mathbb{R}^4$, where $C_i \geq 0$ represents the worst-case execution time (WCET), $T_i > 0$ defines the period, $D_i > 0$ is the relative deadline, and ϕ_i is the phase. The periodic task τ_i releases its first job at time ϕ_i and subsequent jobs are released every T_i time units. It is usually assumed that T_i is an integer multiple of the system tick for every task $\tau_i \in \mathbb{T}$. Jobs must finish within their relative deadline D_i after their release time.

A scheduler determines which job in the ready queue should be executed. Priority-based schedulers, in which scheduling decisions are made by assigning priorities to jobs, have been widely studied in the literature. At any point in time when a scheduling decision has to be made, the highest-priority job among the jobs in the ready queue is allocated to the processor for execution. When a job completes its execution, it is removed from the ready

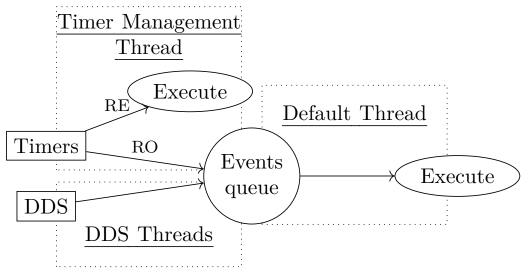
[1]We use the term releaser to indicate an independent component of a system that is responsible for determining if and when a job should be released. Its role is distinct from that of the scheduler, which decides which of the released jobs should be running at any point in time.



(a) Classical real-time scheduler.



(b) ROS 2 default executor.



(c) ROS 2 events executor.

Fig. 1: Scheduling mechanisms.

queue. Every job of τ_i executes for at most C_i time units and has an absolute deadline specified as its release time plus the relative deadline. A task set is an implicit-deadline system if $D_i = T_i$ for all tasks $\tau_i \in \mathbb{T}$, and a constrained-deadline system if $D_i \leq T_i$ for all tasks $\tau_i \in \mathbb{T}$. If the deadline is not constrained by the period, i.e., $D_i > T_i$ is allowed, then we call the task set an arbitrary-deadline system. We consider arbitrary-deadline task systems in this paper, covering all possible cases.

In the literature, priority-based schedulers (on the task level) are classified into fixed-priority and dynamic-priority schedulers. A scheduler is a fixed-priority scheduler if, for any two tasks τ_i and τ_j, either all jobs of τ_i have higher priority than all jobs of τ_j or all jobs of τ_j have higher priority than all jobs of τ_i. In contrast, for dynamic-priority scheduling, a job of τ_i may have a higher priority than some jobs of τ_j but a lower priority than other jobs of τ_j. Specifically, the rate-monotonic (RM) scheduler (where a task with a shorter period has a higher priority) and deadline-monotonic (DM) scheduler (where a task with a smaller relative deadline has a higher priority) are well-known fixed-priority scheduling policies, while the earliest-deadline-first (EDF) scheduler (where a job with the earliest absolute deadline has the highest priority) is a well-known dynamic-priority scheduling policy [30].

Furthermore, we differentiate between preemptive and non-preemptive scheduling algorithms. That is, while for preemptive scheduling a scheduling decision is made at every timer interrupt, for non-preemptive scheduling, decisions are only made at a few specific checkpoints of job execution. More specifically, in preemptive scheduling, at every timer interrupt, it is checked whether the currently running job is still the highest-priority job in the ready queue, and if so, it continues executing. Otherwise, the system performs a context switch, preempts the currently executing job, and executes the new highest-priority job in the ready queue instead. On the other hand, for non-preemptive scheduling, a scheduling decision is only made when (i) a running job finishes its execution or (ii) a job is inserted into an empty ready queue. Since the scheduler can make a scheduling decision to start the execution of the highest-priority job in the ready queue only at those time points, a running job is never preempted and continues to be executed until it finishes. Worst-case analysis of non-preemptive schedulers for real-time systems has been widely studied, e.g. [12], [13], [15], [24], [31].

Example 1. We consider the running example from the beginning of Section II with three tasks. Specifically, we model them as periodic tasks with implicit deadlines, i.e., $\tau_1 = (3, 10, 10, 0)$, $\tau_2 = (10, 30, 30, 0)$, and $\tau_3 = (10, 30, 30, 0)$. Under RM scheduling, task τ_1 has the highest priority. The schedule for RM non-preemptive scheduling is depicted in Figure 2a. In particular, we observe that at time 10, task τ_1 is placed in the ready queue directly by the releaser. At time 13, the scheduler makes a new scheduling decision, identifies the second job of task τ_1 as the highest-priority job, and schedules it. Most importantly, all tasks are released periodically, and the task prioritization and job selection of the scheduler lead to a schedule where all tasks meet their specified deadline. □

B. Default Executor in ROS 2

The default single-threaded ROS 2 executor is depicted in Figure 1b. It schedules tasks' eligible jobs using a wait set. Its scheduling mechanism is split up into two alternating phases, the polling point and the processing window, which are performed sequentially in one thread.

The polling point functions like the releaser in classical real-time scheduling. At each polling point, the executor updates the wait set by sampling one job from each activated task. In ROS 2, tasks are typically either timer tasks or subscription tasks. Timer tasks are time-triggered and activated every Period T_i, while subscription tasks are event-triggered and activated indirectly through message reception. Tasks can publish messages through the communication middleware of ROS 2, the Data Distribution Service (DDS).

During each processing window, which is equivalent to the scheduler in classical real-time scheduling, the jobs are executed using a fixed-priority non-preemptive scheduling policy. The executor iterates over the wait set and selects the highest-priority job according to the fixed-priority order of their respective tasks. The executor then executes the job non-preemptively by calling a function (callback) associated with the task. After finishing the execution of all jobs in the wait set, the wait set is empty, and the executor starts a new polling point. The default ROS 2 executor prioritizes timers over subscriptions, and tasks of the same type are prioritized according to their order of creation. Specifically, ROS 2 does not provide any direct interfaces to configure task priorities.

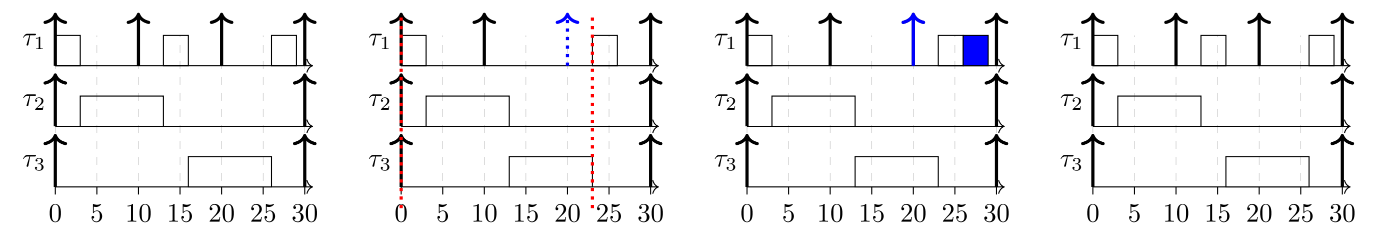
In general, the design of the ROS 2 scheduling mechanism ensures that no matter how many tasks are part of the system, all of them will be executed at some point. This approach simplifies development, as developers do not need to analyze whether all tasks are guaranteed to execute. However, this design comes at the cost of strict timing requirements, such as ensuring the periodic release of timer tasks.

For the release of tasks in ROS 2, there is no timer interrupt that checks the eligibility of tasks at every system tick. Instead, jobs are released only at polling points, regardless of the time elapsed since the last polling point during the preceding processing window. This reduces the flexibility of the schedule since fewer scheduling decisions are made. Furthermore, to guarantee the periodic release of timers, the processing window length must be less than the period of all timer tasks.

However, the ROS 2 executor is designed to execute all tasks at some point for any system configuration, even when the processing window may be longer than the period of a timer task. For this, it provides a flexible mechanism to track and perform the release of timers. Each timer has a timestamp[2], which is used to determine the eligibility of the timer for release. At each polling point, the executor checks if the current time is greater than or equal to the timestamp of the timer. If so, a job of that timer is sampled and added to the wait set. Then, during the following processing window, when the timer job is selected for execution from the wait set (the start time of the timer job), the timer's timestamp is updated. To update the timestamp, it is increased by the minimal multiple of the period such that it is greater than the current time. As a result, the executor may increase the timestamp of the timer by a multiple of the period, skipping a timer job if the time between the previous timestamp and the start time of the next timer job is longer than the period of the timer. This design, while being flexible, may skip timer jobs, contrary to the expectation of a periodic release of timers.

Example 2. Again, we consider the running example from the beginning of Section II with three tasks. Since the tasks are time-triggered, we implement them as timer tasks with periods 10 for τ_1 and 30 for τ_2 and τ_3. The schedule obtained using the default ROS 2 executor is depicted in Figure 2b. In particular, the first polling point at time 0 samples a job of each timer task and puts them

[2]The timestamp refers to the activation time variable of the ROS 2 codebase in the C++ client library rclcpp.



(a) Under classical non-preemptive FP scheduler. (b) Under ROS 2 default executor. (c) Under proposed scheduler without priority-ordering. (d) Under proposed scheduler with priority ordering.

Fig. 2: Schedules for the running example of Section II.

to the wait set. During execution of the first job of τ_1, the timestamp of τ_1 is updated to time 10. Only when the processing window finishes (at time 23), during the polling point, the executor collects the second job of τ_1 and moves it to the wait set. In the subsequent processing window, the timestamp of τ_1 is updated to 30—skipping the timestamp 20. While all jobs are executed at some point after activation, the timer jobs are not released at their specified periods. □

Dedicated response-time analyses have been developed for the default ROS 2 executor [6], [8], [34], along with suggestions to optimize the ROS 2 application code to mitigate long response times [34]. Casini et al. [8] also introduced the concept of processing chains, in which tasks may be triggered due to the arrival of data and output a result, triggering other tasks. This propagation of data creates a natural structure of chained tasks that can be used to measure the end-to-end latency of the task set. Specifically, Teper et al. have analyzed [35], [38] and optimized [36] such end-to-end latencies in the ROS 2 single-threaded executor. Multi-threaded executors have been developed and analyzed [25], [29], [33], but these are still constrained by the wait set behavior of the default executor. Their implementation either shares the wait set among threads in a thread pool or runs multiple unmodified executors in different threads. The latter approach also requires the manual assignment of nodes to executors. Furthermore, the multi-threaded variant of the default executor is shown to be affected by starvation [39], i.e., it does not even ensure that all tasks will be executed at some point after activation.

C. Comparison of Schedulers

We now compare the classical non-preemptive priority-based scheduler from Section II-A with the default executor in ROS 2 from Section II-B.

First, the release mechanism in ROS 2 is substantially different from the typical release mechanism in classical real-time systems. That is, in classical real-time systems, jobs are released (almost) immediately by the timer interrupt. Contrarily, in ROS 2, activated tasks are passively sampled by the executor at polling points. Specifically, timers are not released every time their period elapses, but only when the executor samples them at the polling point, potentially causing delays in the release of jobs. Combined with the timestamp update mechanism, this can lead to the skipping of timer jobs for ROS 2, as jobs may not be sampled every period. This is depicted in Figure 2b, where a job of τ_1 for timestamp 20 is skipped. In contrast, the same task set under the classical scheduler with implicit deadlines would release all jobs at their specified periods, and no task would miss its deadline.

Secondly, the scheduling mechanism and the resulting delays in ROS 2 differ from the classical non-preemptive fixed-priority scheduler. In a classical non-preemptive FP scheduler, high-priority jobs can only be blocked by one lower-priority job, as a new scheduling decision is made every time a job finishes. We can see this in Figure 2a, where the second job of task τ_1 is blocked by only one job, the job of τ_2, and the response time of τ_1 is 6. On the other hand, in ROS 2, scheduling decisions can only be made at the polling points. That is, only once all the remaining jobs in the current processing window have finished execution, including potentially all lower-priority jobs, new jobs are added to the wait set. This case is depicted in Figure 2b where the second job of τ_1 is blocked by both the jobs of τ_2 and τ_3 during $[10, 23]$, leading to a response time of 16. This may lead to higher response times for tasks in ROS 2 compared to the classical non-preemptive FP scheduler.

Thirdly, ROS 2 does not allow for the explicit setting of timing properties, such as deadlines, or the direct control of the priority of tasks to influence the scheduling order. Instead, ROS 2 enables developers to focus on the application logic and the communication between nodes, while the underlying scheduling mechanism ensures that all tasks are executed at some point after activation, even if the timing requirements, such as the timer periods, are not met.

Due to these substantial differences between the classical non-preemptive scheduler and the ROS 2 default scheduler, the rich literature of real-time scheduling theory is not directly applicable to ROS 2 systems. Therefore, alternative executors for fixed-priority [10] or dynamic-priority [1] schedulers have been developed. However, using executors provided directly by ROS 2 for periodic task systems is still desirable to reduce design and implementation costs for developers.

III. The Events Executor

In 2023, ROS 2 added the events executor to its distribution [23], providing an alternative to the default executor. It does not use a wait set but instead uses a FIFO queue, called an events queue, that stores the jobs of tasks that are eligible for execution. Furthermore, it uses separate threads for releasing jobs to the events queue and for scheduling jobs from the events queue. Specifically, we have a timer management thread, DDS threads, and the default thread. The mechanism of the events executor is depicted in Figure 1c.

The default thread is responsible for scheduling the jobs in the events queue. It does not release jobs, but only executes the jobs in the events queue in a FIFO order. Non-timer tasks are released by the DDS threads into the events queue. The release of non-timer tasks occurs when the DDS threads receive events, such as the arrival of a message. Specifically, the DDS threads enqueue the events in the events queue as part of the operation that receives messages. For timer tasks, the timer management thread manages a timer release queue, and there are two options:

- Option 1: Release-Only (denoted as RO). The timer management thread releases the timer jobs from the timer release queue to the events queue (where non-timer events also reside), and all jobs are scheduled by the default thread in FIFO order.
- Option 2: Release-and-Execute (denoted as RE). The timer management thread holds both released and unreleased timer jobs in its timer release queue and schedules them based on their timestamp.

Therefore, for the RO option, the timer management thread only releases the timer jobs to the events queue, while for the RE option, the timer management thread both releases and schedules the timer jobs.

Algorithm 1 summarizes the implementation of the timer management thread in ROS 2. First, in Lines 2-3, the timer management thread determines the time to sleep until the next timer release time and then waits until that time elapses. For each release, in Lines 4–5, the timer management thread first determines if the timestamp of the head timer has been reached. When the timer is released, its timestamp is updated to the next timestamp in Line 6, which is the smallest multiple of the period that is greater than the current time. Then, for the RO option, the timer job is released into the events queue in Line 8. Likewise, for the RE option, the timer job is immediately scheduled in Line 10. Afterward, in Line 12, the queue is reordered according to the new timestamps of the timers that remain in the queue.

The events executor has two properties that make it a potentially suitable option to bridge the gap between ROS 2 and classical real-time scheduling:

1) Separation: Different threads can be used to separate the execution and release of tasks. Specifically, as depicted in Figure 1c, the timer management thread and the DDS threads can be used to release jobs to the events queue, while the default thread handles the job execution. This is similar to the mechanism of the classical real-time scheduler, depicted in Figure 1a, where the releaser and the scheduler are separated as the releaser inserts jobs into the ready queue when they are ready and the scheduler takes ready jobs out of the ready queue for execution.
2) Granularity: Compared to the polling point in the default ROS 2 scheduler, depicted in Figure 1b, where all jobs are collected during the polling point for execution, the events executor always collects only one job for execution (either from the events queue or from the set of active timers). Therefore, this leads to a higher granularity of scheduling decisions, potentially allowing to reduce the blocking from lower-priority jobs.

Algorithm 1 Timer Management Thread in Events Executor

```
1: while Running do
2:     $time\_to\_sleep$ ← next timer release time - now
3:     $wait\_for(time\_to\_sleep)$
4:     head_timer ← timers.front()
5:     while head_timer is eligible do
6:         head_timer.update_timestamp()
7:         if Option RO is configured then
8:             $events\_queue.enqueue(head\_timer, data)$
9:         else if Option RE is configured then
10:            $execute(head\_timer, data)$
11:        end if
12:        Reorder timers by release time
13:        head_timer ← timers.front()
14:    end while
15: end while
```

IV. Problem Definition

As we have discussed in Section II-B, the default executor of ROS 2 is not compatible with typical literature results on real-time systems. That is, the default executor shows behavior that is typically not considered when looking at typical priority-based schedulers. More specifically, classical priority-based schedulers have the following properties:

P1) Each job is inserted into the ready queue (almost) immediately by the releaser.
P2) Each scheduling decision determines only one job to be executed next.

Contrarily, for the default scheduler of ROS 2 we have:

1) A job is inserted into a wait set only at polling points and only if the corresponding task has been activated since its preceding execution.
2) At the polling point, the scheduler makes the scheduling decision to run all jobs in the wait set in a deterministic order and does not make another scheduling decision until the next polling point.

As a direct consequence, the default ROS 2 scheduler shows behavior that usually cannot occur in typical priority-based schedulers. For example, a job release can be skipped, and a high-priority task can be blocked by all lower-priority tasks. Therefore, the literature on classical real-time systems theory is not directly applicable to ROS 2.

In this paper, we address the problem:

> How can the results from classical real-time systems be made applicable to ROS 2?

To investigate this, we consider the events executor of ROS 2, which exhibits properties that are closer to traditional real-time scheduling mechanisms, making it a promising candidate for bridging the gap between these two models. To that end, we focus on timer tasks first, present our proposed scheduler based on the events executor in Section V, and discuss the compatibility with the literature on non-preemptive schedulers for periodic tasks in Section VI. Afterward, we discuss extensions of our results to ROS 2 applications with subscription tasks in Section VII.

V. Proposed Scheduler

In this section, we present our proposed scheduler. In particular, we discuss the necessary configuration and modification of the events executor to ensure the same behavior as a classical priority-based, non-preemptive real-time scheduler.

As discussed in Section III, the events executor has a finer scheduling granularity than the default ROS 2 executor. In particular, a new scheduling decision is made when a job finishes execution. Therefore, the events executor inherently fulfills property P2) from the problem definition. The remainder of this section is divided into two parts: First, we discuss the configuration to ensure immediate job releases, i.e., P1). Second, we detail the modifications to model different priority-based scheduling algorithms like RM, DM, or EDF.

A. Ensure Immediate Releases

To ensure immediate releases, we need to ensure that the release functionality is decoupled from the execution functionality and that the execution cannot block job releases. We achieve this by proper configuration of the RO/RE option and by proper prioritization of the threads.

a) Choice of RO/RE Option: As discussed in Section III, there are two options for the events executor: the Release-Only (RO) and the Release-and-Execute (RE) option.

The RE option does not separate the releaser and scheduler for timers. In this case, job releases are only performed between job executions. This can lead to timer jobs being lost if the execution time of a timer job exceeds the period of other timers. Due to these delays, the RE option is not suitable to ensure immediate job releases.

However, the RO option separates the releaser and scheduler for timers. Just like the classical real-time scheduler, this allows job releases to be handled without being blocked by job execution. Specifically, timer jobs can be released into the events queue while the default thread executes jobs. Furthermore, job releases and timestamp updates are independent of the scheduling decisions made by the default thread. Therefore, to achieve P1), we choose the RO option.

b) Thread Management: To achieve P1), we need to ensure that the release of jobs is guaranteed even while a job is executed. For this, we need to rely on the assumption that the operating system allows scheduling threads preemptively, meaning that the operating system can interrupt the execution of a thread at any time. Furthermore, we need to assume that threads can be prioritized if they run on the same processing core, i.e., if a lower-priority thread is executing, the operating system can preempt it in favor of a higher-priority thread.

If the threads run on different processing cores, the operating system can schedule them independently, and the threads can run concurrently. However, the threads share a common data structure in the events queue, and access to the data structure needs to be synchronized. Therefore, the only interference when releasing jobs occurs during access to the events queue. We consider such blocking times as part of the job release overhead and can thus be ignored. Therefore, the timer management thread can release jobs almost immediately, even if the default thread is executing jobs.

If the threads share the same processing core, the priority of the threads is crucial. To ensure jobs are released (almost) immediately, the timer management thread and the DDS threads are configured to have a higher priority than the default thread. In this case, the timer management thread and the DDS threads can preempt the default thread when it is executing a job, releasing the timer jobs (almost) immediately. Otherwise, if, for example, the default thread has a higher priority than the timer management thread, a long job execution by the default thread may delay the release of a timer job, potentially skipping it. Again, overheads caused by the operating system and for accessing the events queue are considered part of the job release overhead and are ignored.

B. Job Prioritization

Although the configuration of the previous subsection achieves P1) and P2), the job execution order in the events queue remains FIFO. Specifically, we consider the running example of three tasks, as specified in Example 2 for the ROS 2 architecture. The schedule is depicted in Figure 2c. While no job releases are skipped and scheduling decisions occur after every job, the schedule differs from the schedule under classical non-preemptive FP scheduling, depicted in Figure 2a. Specifically, while the second job of τ_1 is added to the events queue at time 10, the job is only started at time 23 because the jobs are executed in a FIFO order. Assuming implicit deadlines, this would lead to a missed deadline for τ_1. Under the classical non-preemptive FP scheduling (Figure 2a), the second job of τ_1 is added to the ready queue at time 10 as well, but started already at time 13, because it has higher priority than the job of τ_3. To achieve the same schedule, we need to handle the priority ordering of the events queue.

For this, we change the events queue to a priority queue instead of a FIFO queue. The highest-priority job can then be selected by extracting the first job from the priority queue. The priority queue only adds an overhead of $O(\log n)$ for each priority insert, where n is the number of tasks in the system. For priority-based reordering, the overhead is $O(n \log n)$. This overhead is inherent to priority-based management, unless special assumptions or mechanisms are introduced to avoid it. However, the overall overhead is negligible compared to that of the other ROS 2 management routines.

For sorting the events queue in ROS 2, we additionally require information about the priority of the jobs. By default, ROS 2 does not provide a priority field for tasks. In the following, we discuss two options to provide the priority of the timer jobs, with different levels of user interaction required. We focus on RM and EDF scheduling of timer tasks. Possible extensions for subscription tasks can be found in Section VII.

The first option uses each timer's period as its priority. Each timer task already provides a period, required when creating the timer. Thus, this information can be used during scheduling to prioritize the timer jobs in the events queue. For RM scheduling, the timer job with the smallest period has the highest priority. Therefore, when inserting a timer job into the events queue, the queue can access the period of the timer and insert the job based on the period. For implicit-deadline EDF scheduling, we can use the next timestamp of the timer as the priority of the timer job. When inserting a timer job into the events queue, the queue can access the next timestamp of the timer and insert the job based on the timestamp.

As a second option, we propose to add a user-defined priority field to the timer structure. It can either hold static priorities for static-priority scheduling or relative deadlines for dynamic-priority scheduling. The approach theoretically supports any job-level fixed-priority non-preemptive scheduler if the priority can be decided when the job arrives, e.g., the non-preemptive rate-monotonic (RM) and earliest-deadline-first (EDF) scheduling policies for uniprocessor systems. While this option is less user-friendly, as it requires users to change the current code of their system to provide the priority of the timer jobs, it is a more versatile approach and supports any job-level priority ordering. We provide both options to enable priority-based scheduling in ROS 2.

After prioritization of the events queue using one of the two options above, the corresponding schedule for the running example is depicted in Figure 2d. Specifically, at time 10, the second job of task τ_1 is inserted into the events queue with the highest priority and is chosen to be scheduled at time 13. The corresponding schedule coincides with the schedule under classical non-preemptive FP scheduling from Figure 2a. To conclude, our changes enable the events executor to be used for arbitrary fixed-priority or dynamic-priority scheduling policies for non-preemptive scheduling. Thus, our proposed design is compatible with the classical real-time scheduling theory. In Section VI, we discuss the compatibility of our proposed scheduler with the classical real-time scheduling theory.

VI. Compatibility with Non-Preemptive Schedulers for Periodic Tasks

In this section, we explain how our proposed scheduler from Section V bridges the gap between the literature on priority-based scheduling and ROS 2. To that end, we show that our scheduler behaves analytically like a non-preemptive work-conserving priority-based scheduler in Section VI-A. In Section VI-B, we discuss the overhead that results from our scheduler. We specify which analytical results apply to our scheduler in Section VI-C, focusing on the worst-case response time and end-to-end latency (which is one of the predominantly studied metrics for ROS 2 systems).

For the discussion in this section, we assume that the ROS 2 application has only timers and the system is exclusively used to execute the ROS 2 application on one processor. An extension to incorporate subscriptions can be found in Section VII. We assume that tasks have different priorities and ties are broken arbitrarily but deterministically.

A. Analytical Behavior of Our Scheduler

Our solution uses the events executor with a modification of the events queue for priority-based job ordering. The release of timer jobs into the events queue is done in the timer management thread, which is given a higher priority than the default thread. Therefore, whenever the timestamp of the timer task is reached, a job of that task is inserted into the events queue immediately[3]. Hence, jobs are released (i.e., inserted into the events queue) periodically. Moreover, scheduling decisions are made whenever a job finishes by pulling the first job (i.e., the highest priority job) from the events queue. In addition, when the executor idles and a new job is released, it makes a scheduling decision immediately and starts executing the job. Therefore, our scheduler behaves like a typical non-preemptive work-conserving scheduler.

[3]Please note that our solution only works under the assumption that the timer management thread is not blocked.

B. Bounding the Releaser Overhead

Let $\delta > 0$ be the maximal time for releasing a job. Then, if a job is released into the events queue, this prolongs the execution time of the job that is currently running. Hence, by counting the maximal number of releases during the execution of a job, we obtain a bound on the overhead. We denote by Δ_i the overhead that jobs of τ_i can experience.

With our solution, a job is released whenever a timestamp is reached. The number of timestamps that can occur for a task τ_j during an interval of length t is $\left\lceil \frac{t}{T_j} \right\rceil$. Let $t_0 \in \mathbb{R}_{>0}$ be the lowest positive real number such that $t_0 \geq C_i + \sum_{j=1}^{n} \left\lceil \frac{t_0}{T_j} \right\rceil \cdot \delta$. Then t_0 is the amount of time that a job of τ_i can execute, including the overhead from releasing other tasks. Hence, the release overhead for task τ_i is bounded by

$$\Delta_i \leq \sum_{j=1}^{n} \left\lceil \frac{t_0}{T_j} \right\rceil \cdot \delta. \tag{1}$$

This overhead has to be accounted for when applying the results in the literature for non-preemptive schedulers by prolonging the worst-case execution time value C_i by Δ_i.

This bound on the overhead Δ_i can be tightened further if the following two conditions hold:

- The task set has constrained deadlines.
- The task set is already shown to be schedulable (e.g., by using the bound on the overhead from Equation (1) together with a schedulability test that is referenced in Section VI-C).

Under these conditions, every other task can only release one job during the execution of a job of τ_i. Otherwise, assuming a task τ_j releases two jobs at timestamps t_1 and $t_2 = t_1 + T_J$ during the execution of a job of τ_i, then the job of τ_j cannot be executed until time t_2. Hence, it would miss its constrained deadline, which violates the schedulability of the task set. Therefore, under the two conditions, the overhead for the releaser is upper bounded by

$$\Delta_i \leq n \cdot \delta, \tag{2}$$

where n is the number of tasks in the system.

We note that it is unsafe to apply the tightened bound from Equation (2) directly for calculating a response time bound and checking the schedulability based on that. However, when the response time is already shown to be less than the deadline D_i, we can use this formula to tighten the response time bound further. This can be beneficial when applying analyses that utilize the worst-case response time (cf. Section VI-C).

C. Analytical Results

As detailed in Section VI-A, our proposed scheduler behaves analytically like a typical non-preemptive scheduler. Although this pertains to any scheduling algorithm that can be modeled by reordering the events queue, as discussed in Section V-B, we focus specifically on FP and EDF scheduling. For this section, we assume that the overhead is accounted for by prolonging the WCET, i.e., by redefining $C_i := C_i + \Delta_i$.

Worst-Case Response Time: The worst-case response time of a task is the maximal time between the release and finish of any job of that task. With our proposed scheduler, the rich literature on non-preemptive scheduling for periodic tasks becomes applicable (cf. [12], [15], [24], [31], [42]). Note that this also encompasses the literature on non-preemptive sporadic tasks by setting the minimum inter-arrival time to the task period. Especially, results for non-preemptive FP scheduling can be found in [12], [15], [31], [42] and results for non-preemptive EDF scheduling can be found in [15], [24], [31]. With our scheduler, any of these analyses can be applied to derive a bound on the worst-case response time. We note that when a task set is shown to be schedulable (for example by any of the above analyses), then we can also claim an upper bound on the worst-case response time, namely $R_i \leq D_i$.

End-To-End Latency: One of the predominantly studied metrics for ROS 2 applications is end-to-end latency. That is, given a so-called cause-effect chain, which is a sequence of tasks $E = (\tau_{i_1} \rightarrow \dots \tau_{i_N})$, with $i_j \in \{1, \dots, n\}$ being a task index for all $j = 1, \dots, n$, then we are interested in the amount of time that data needs to traverse the cause-effect chain. Specifically, the Maximum Reaction Time (MRT), i.e., the amount of time until an external cause is fully processed, and the Maximum Data Age (MDA), i.e., the age of data utilized in an actuation, have been analyzed extensively in the literature. Recently, it has been shown that MRT and MDA are equivalent [19]. Hence, we use *Lat* to denote the end-to-end latency of a cause-effect chain in general.

While there are only a few analytical results for the standard ROS 2 application [35], [37], there is a large body of literature results for typical periodic tasks [2], [3], [5], [11], [14], [16]–[20], [22], [26]. Our scheduler enables such results and insights from end-to-end analysis for periodic task systems.

A classical bound, proposed by Davare et al. [11], is to bound the end-to-end latency by summing up the task periods T_{i_j} and worst-case response times R_{i_j}:

$$Lat \leq \sum_{j=1}^{N} \left(T_{i_j} + R_{i_j} \right) \tag{3}$$

Here, T_{i_j} is the period of the j-th task in the cause-effect chain, and R_{i_j} is the worst-case response time of the j-th task in the cause-effect chain, i.e., of task τ_{i_j}. The bound assumes that tasks communicate via the implicit communication principle [22], i.e., jobs read data when they start execution and write data when they finish execution. While the original bound is proven only for preemptive fixed-priority scheduling in [11], this result is universally applicable when utilizing a correct bound on the worst-case response time. This is proven for example as a byproduct of the probabilistic analysis by Günzel et al. [21], i.e., their reduction to the case with deterministic worst-case response time bound in Note 6.3 of [21] proves Equation (3) without assumptions on the scheduling algorithm.

To summarize, given a cause-effect chain E on the task set $\mathbb{T}$ scheduled by our scheduler, then the following steps result in a bound on the end-to-end latency:

1) Calculating the overhead Δ_i from Section VI-B and incorporating it in the WCET by $C_i := C_i + \Delta_i$.
2) Providing an upper bound on the worst-case response time R_i using literature results for non-preemptive periodic tasks.
3) Calculating a bound for the end-to-end latency using Equation (3).

In Section VIII, we compare this bound on the end-to-end latency with the latency that can be analytically guaranteed for the typical ROS 2 scheduler [37].

VII. Subscription Tasks

The previous sections focused on the behavior of timer tasks. However, besides timers, ROS 2 allows tasks to be triggered through the built-in Data Distribution Service (DDS). Since such subscription tasks are widely used in practice, this section explains how our findings can be extended to systems involving subscription tasks.

First, we observe that our proposed scheduler from Section V still fulfills P1) and P2) even for subscription tasks. That is, by giving the DDS threads higher priority than the default thread or by running them on a separate core, the release of subscription tasks is not blocked by job execution, and a new scheduling decision is made after every job.

We are left with the problem that subscription tasks cannot be modeled and analyzed as periodic tasks because they are not inherently time-triggered (unlike timer tasks). Instead, the timing behavior of subscriptions is dependent on the tasks that publish to the subscription's topic. Furthermore, there is no direct parameter to set the priority of a subscription task. In the following, we present two approaches to model and analyze subscription tasks. First, a sporadic task model that applies to any system structure. Second, a limited-preemptive scheduling approach can be used for systems with a specific structure to obtain potentially tighter bounds. Afterward, we discuss solutions for the prioritization of subscription tasks.

A. Modeling as Sporadic Tasks

To analyze subscriptions as sporadic tasks, we need to assume that subscription tasks have a minimum inter-arrival time > 0, meaning that no two jobs of the same subscription task are released within this time frame. If such a minimum inter-arrival time can be identified, then we can analyze them as sporadic tasks. For example, if tasks are assumed to only publish once at the end of their execution, the minimum inter-arrival time can be bounded by the minimum best-case execution time among all tasks that publish to the subscription topic, for single-executor systems. While this general approach to deriving a minimum inter-arrival time can be quite pessimistic, more dedicated analyses for specific task sets can potentially derive much tighter bounds on the minimum inter-arrival time.

Please note that, as in Section VI-B, we also need to account for the overhead from the releaser. For this, we prolong the WCET of the subscription tasks. Given a minimum inter-arrival time T_i for a subscription task τ_i, the overhead Δ_i is bounded by Equation (1).

B. Modeling as Limited-Preemptive Tasks

For a special case, where tasks are arranged in sequences, the analysis for limited-preemptive tasks is applicable. That is, we consider the case that each task invokes the activation of at most one subscription task, and each subscription task can be activated by exactly one task. Furthermore, we assume that the first task of a sequence is a timer task, to achieve time-triggered behavior, and all subsequent tasks are subscription tasks since they subscribe to a topic by definition. An example of such sequences can look as follows:

- $\text{sequence}_1 = \text{timer}_1 \rightarrow \text{subscription}_1 \rightarrow \text{subscription}_2$
- $\text{sequence}_2 = \text{timer}_2 \rightarrow \text{subscription}_3 \rightarrow \text{subscription}_4$

Each job of the timer task in a sequence releases one job of the subsequent subscription tasks. Furthermore, we assume no delay is introduced by the DDS between releasing a subscription task and activating the downstream task, preventing lower-priority jobs from being started before the subscription task's release. For this, ROS 2 provides the option to use synchronous DDS communication or zero-copy communication.

With these assumptions, the literature on limited preemptive scheduling, e.g., [4], [7], [43], is applicable, where each sequence is modeled as one 'task', with each timer task and subscription task being a 'subtask'. Each subtask is scheduled non-preemptively, and between subtasks, the task can be preempted. The benefit of considering sequences is that there is no need to determine (potentially pessimistic) minimum inter-arrival times of subscription tasks.

We further note that, in case the DDS introduces a certain delay in releasing the subsequent subscription [28], another lower-priority job may be started in between the execution of two non-preemptive subtasks. In that case, the additional delay can be modeled as self-suspension [9]. More specifically, the set of sequences behaves like a set of segmented self-suspending tasks scheduled non-preemptively on a single core.

C. Prioritization of Subscription Tasks

For both models, a prioritization of the subscription tasks is necessary. Currently, there is no direct way to set the priority of a subscription task in ROS 2. However, we propose two potential solutions to prioritize subscription tasks.

- The first one is to utilize the ROS 2 DDS and its Quality-of-Service (QoS) settings to configure parameters about the communication behavior. Specifically, each subscription has one topic that it subscribes to, and each topic has a QoS setting. Currently, one QoS parameter for topics is the so-called deadline [32], which for topics describes the expected maximum amount of time between subsequent messages being published to the topic. To integrate the minimum inter-arrival time, we propose to introduce a QoS parameter that reflects the expected minimum amount of time between subsequent messages being published to the topic. For the final integration into the scheduling decisions, changes to the executor are necessary that would consider the DDS QoS parameters, such as the minimum inter-arrival time.
- For the second one, we propose to add a priority field to subscriptions, which can be set by the user. This approach is more direct and flexible but requires changes to the ROS 2 codebase and the application code. Given this priority field, the same procedure as for the timer jobs mentioned in Section V-B can be applied for both static and dynamic priority assignments.

In both cases, it would be beneficial to have interfaces to dynamically set the priority comparison function of an executor. This would maximize compatibility, ensuring that existing executors can still be used, while other scheduling policies can be implemented just as easily.

In any case, the priority queue that we introduced in Section V to replace the events queue can be used to handle both timer jobs and subscription jobs. Moreover, there are no further modifications required for handling subscriptions.

VIII. Evaluation

In this section, we evaluate our proposed modifications of the ROS 2 events executor and the corresponding analyses. The evaluation consists of three contributions:

- We confirm compatibility with scheduling theory on non-preemptive schedulers for periodic tasks in Section VIII-A. More specifically, we demonstrate for a timer-only task set that the analytical results of Section VI are applicable. That is, the modified ROS 2 events executor does not drop jobs, and it respects the worst-case response time bounds for non-preemptive scheduling presented in Section VI-C.
- In Section VIII-B, we show that the bounds on end-to-end latency for cause-effect chains, enabled by the theory of non-preemptive scheduling bounds in Section VI-C, are tighter than the state-of-the-art analytical end-to-end latency bounds for the default executor. To that end, we apply the analyses to synthetic task sets using the WATERS [27] benchmark.
- We compare the performance of our modified executor using the RM scheduling policy with the default executor in the Autoware reference system in Section VIII-C. The Autoware reference system includes both timers and subscriptions, and demonstrates that we can achieve lower end-to-end latency than the other executor options that are provided natively by ROS 2.

Our experiments feature the following executor configurations:

- default executor (Default),
- static default single-threaded Executor (Static),
- unmodified default events executor (Events),
- our rate-monotonic events executor (RM), and
- our earliest-deadline-first events executor (EDF).

We include the static default executor for completeness. It behaves like the default executor, but with less overhead, as it does not have to handle dynamic system configurations. We also provide an artifact[4] of our evaluation.

A. Compatibility with Scheduling Theory

In this subsection, we demonstrate that existing results on non-preemptive scheduling apply to our scheduler, and analytical results correctly predict the behavior of the system.

To that end, we examine a timer-only task set that consists of three types of components, and seven nodes in total. The system has four cameras, two LiDARs, and an IMU. Each node has one timer task, and each component is independent of the others, meaning that they do not pass data between each other and have no precedence constraints. By varying the maximum execution time of the tasks, we configure the system to have a utilization of 60%, 80%, or 90%. Our three configurations are as follows:

60% utilization

- 4 Camera Nodes: 84 ms period, 10 ms execution each
- 2 LiDAR Nodes: 200 ms period, 10 ms execution each
- 1 IMU Node, 30 ms period, 1 ms execution

80% utilization

- 4 Camera Nodes: 84 ms period, 14 ms execution each
- 2 LiDAR Nodes: 200 ms period, 10 ms execution each
- 1 IMU Node, 30 ms period, 1 ms execution

90% utilization

- 4 Camera Nodes: 84 ms period, 16 ms execution each
- 2 LiDAR Nodes: 200 ms period, 10 ms execution each
- 1 IMU Node, 30 ms period, 1 ms execution

Task periods stay consistent across all utilization levels, so all task sets have the same hyperperiod of 4.2 s. Each task set is run for 5 min, elapsing over 70 hyperperiods. We repeat this setting for each executor design.

Each of these task sets is pinned to a single core of a Raspberry Pi Model 4B with 4GB of RAM running

[4]https://github.com/tu-dortmund-ls12-rt/ros2_executor_evaluations

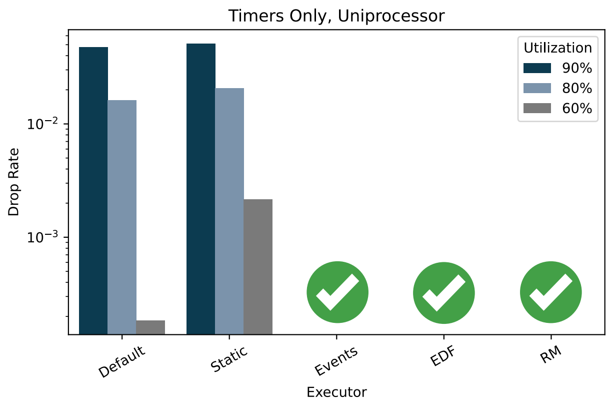


Fig. 3: Dropped jobs on timer-only task sets: A checkmark indicates no dropped jobs across all utilization levels.

Ubuntu 22.04 and ROS 2 Humble. VIII-A uses one core, while VIII-C uses two cores with no thread pinning. VIII-B assumes that the executor runs on one processing core.

Number of Dropped Jobs: The rate of dropped jobs for different configurations is shown in Figure 3. We can see that the default and static executors repeatedly drop jobs. In some cases, a drop rate of 5% is observed. Following our analytical results of Section VI-A, which indicate no job drops, our proposed events executors do not have any job drops for any configuration of the RM and EDF events executors.

Response Time Bounds: In the following, we analytically derive upper bounds on the worst-case response time, following Section VI, and then confirm that the response time bounds are not violated by our RM events executor.

For the analysis, we measured that a job release takes up to $\delta = 0.12\,\mu\text{s}$. The total overhead Δ_i for a timer task τ_i can be determined by Equation (1). Doing the calculations for the task sets of this section, every timer task has an overhead of $\Delta_i = 0.84\,\text{ms}$. For the analysis, we add this overhead to the WCET of each task, i.e., $C_i := C_i + \Delta_i$.

Afterward, we apply Equation 6 from von der Brüggen et al. [42] to determine the worst-case response time of each task.

Theorem 1 (Non-preemptive FP, reformulated from [42]). *Assume that the tasks $\mathbb{T} = \{\tau_1, \cdots, \tau_n\}$ are ordered by their priority, i.e., τ_1 has the highest priority and τ_n has the lowest priority. If there exists a $t \geq 0 \in \mathbb{R}$ such that $t \leq D_k$ and*

$$t \geq C_k + \max_{i>k} C_i + \sum_{i<k} \left\lceil \frac{t}{T_i} \right\rceil C_i, \tag{4}$$

then the worst-case response time R_k of $\tau_k \in \mathcal{T}$ under non-preemptive FP scheduling is upper bounded by t.

We consider implicit-deadline task systems, i.e., the relative deadline of a timer task is equal to its timer period. The analyzed worst-case response times for our RM events executor are reported in Table I, indicating no deadline misses. Thus, we should not observe any job drops, as verified in Figure 3, and all tasks are guaranteed to meet their deadlines.

We then measured the response times during the experiment. The aggregated results of the Camera nodes are shown in Figure 4, the LiDAR nodes in Figure 5, and the IMU node in Figure 6. As shown in the figures, the response times of our work are consistent with the analytical results, and no deadline overruns are observed. Furthermore, in Figure 6, we see that the IMU tasks have deadline overruns on the Default, Static, and Events executors. We note that Figures 4, 5, and 6 only account for the response times of jobs that are not dropped. Hence, for a significant amount of timestamps, the Default executor and Static executor do not respond within the deadline, as can be observed in Figure 3.

In conclusion, our analysis not only correctly indicates the absence of dropped jobs in our RM events executor, but also a correct upper bound on the response time of each task.

TABLE I: Analytical worst-case response times (ms) of our RM events executor.

Utilization	60%	80%	90%	Period
IMU	12.67	16.67	18.67	30
Camera	57.83	75.66	83.66	84
LiDAR	70.50	149.50	167.33	200

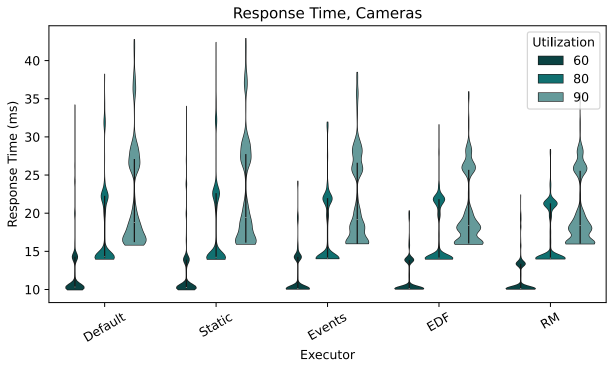


Fig. 4: Response Time of Camera Tasks (ms), D=84 ms

B. Tighter End-to-End Bounds for ROS 2

In this section, we compare the analytically derived upper bounds of the end-to-end latencies between the default ROS 2 executor and our RM events executor. For this experiment, we generate synthetic task sets using the WATERS benchmark [27] for automotive systems.

We evaluate utilization levels of 60%, 80%, and 90%, generating one thousand task sets per configuration with 10 to 200 tasks. To compute end-to-end latencies, we generate 5 to 60 task chains, each with 2 to 15 tasks. We assume all tasks are assigned to one events executor running on a single core.

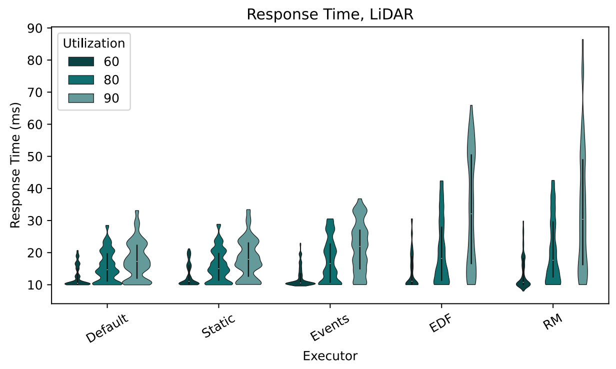


Fig. 5: Response Time of LiDAR Tasks (ms), D=200 ms

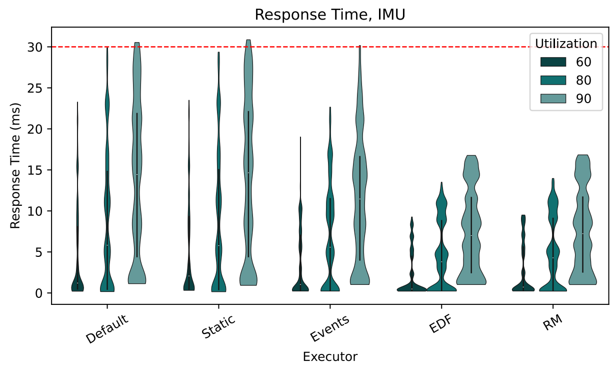


Fig. 6: Response Time of IMU Tasks (ms), D=30 ms

For each task set, we calculate the end-to-end latency for the default ROS 2 executor and our RM (including both RO- and RE-options, as the analysis) events executor, as presented in Section VI-C. Specifically, we use Equation (3) in Section VI to calculate the end-to-end latency of each chain.

The end-to-end latency of the default ROS 2 executor has been analyzed by Teper et al. [35]. We use Equation (15) of Lemma VI.1 and Equation (17) of Lemma VI.2 from [35] to get an upper bound on the latency of each timer. For each chain, we sum up the latency upper bounds of all chain tasks to get the end-to-end latency of the chain.

For each chain, we calculate the normalized reduction of the end-to-end latency by using $\frac{E_{default} - E_{RM}}{E_{default}}$, where $E_{default}$ (E_{RM}, respectively) is the end-to-end latency of the chain under the default executor (our RM events executor, respectively). The results of the evaluation are shown in Figure 7, where the x-axis is the normalized reduction and the y-axis is the number of chains within the binned normalized reduction.

The end-to-end latencies of the chains are typically much lower when using our RM events executor. As the utilization level increases, the number of chains with higher end-to-end latencies increases as well. This is because the default ROS 2 executor treats all callbacks fairly regarding their priorities. Meanwhile, static-priority

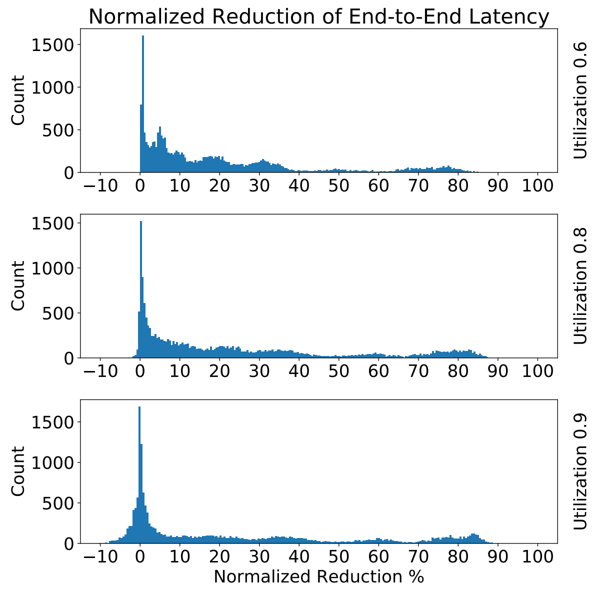


Fig. 7: Reduction of the end-to-end latency between the default ROS 2 executor and our RM events executor.

scheduling assigns some callbacks lower priorities, increasing their response time (see Figure 5), and resulting in a negative reduction of end-to-end latencies for chains that include such tasks. However, the majority of chains have lower end-to-end latency when using the RM events executor. In some cases, we observe a normalized reduction of the end-to-end latency bound of almost 90%. Our experiment shows that our design can be applied to different kinds of task sets and potentially leads to latency improvements.

C. Performance of Autoware Reference System

In this section, we evaluate end-to-end latencies for the Autoware reference system, detailed in [41], which includes interconnected timers and subscriptions [40]. The Autoware reference benchmark runs a simulated version of the Autoware application on a Raspberry Pi Model 4B running Ubuntu 22.04 and ROS 2 Humble. We run the ROS 2 Autoware benchmark on two dedicated cores, while the other two cores are handling the remaining services on the operating system. Specifically, we measure the end-to-end latency of the hot path defined in the benchmark from the front and rear LiDAR sensors to the object collision estimator (c.f. the Autoware reference system), as this latency is a key metric for the responsiveness to crash avoidance. The data at the start of the hot path that is generated by the LiDARs is produced at a frequency of 10 Hz.

For our RM events executor, the subscription tasks inherit the highest priority from their corresponding upstream publishers. Non-preemptive EDF in this scenario

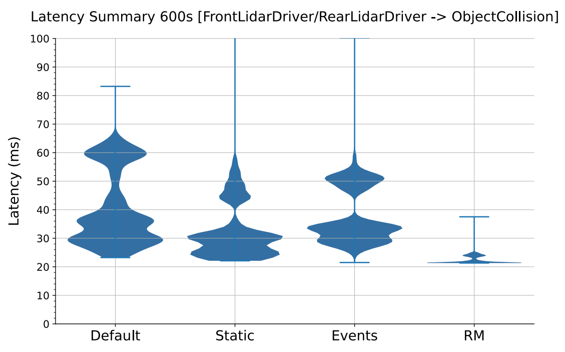


Fig. 8: End-to-end latency of the hot path in the Autoware reference system under different executors.

is not well specified, as subscription tasks do not have any assigned deadlines, and hence is not part of this evaluation.

Figure 8 shows the violin plot for the latency of the hot path in the Autoware reference system from a test run of 600 seconds. Of the executors measured, our RM events executor has the best worst-case latency, significantly outperforming the other executors. Compared to our work, the default ROS 2 executor has a 2.6x slowdown along the hot path. Our evaluation shows that our executor design using existing scheduling policies from classical real-time systems can be applied to practical ROS 2 applications and can provide major benefits in terms of end-to-end latency.

IX. Conclusion

This paper provides deep investigations of the recently introduced events executor (in 2023) in ROS 2 to provide compatibility with the classical real-time scheduling theory of periodic task systems. Specifically, our solution is easy to integrate into existing ROS 2 systems since it requires only minor modifications of the events executor, already natively included in ROS 2. Thus, it can coexist with available ROS 2 executors, such as the default executor.

Our study enables the rich literature of the real-time scheduling theory for non-preemptive schedules to apply to ROS 2, under the assumption that an individual core is exclusively assigned to our executor. Furthermore, we intensively validate the feasibility of our modifications for non-preemptive RM and non-preemptive EDF in the ROS 2 events executor for several scenarios. The evaluation results show that our ROS 2 events executor with minor modifications can have significant improvement in terms of dropped jobs, worst-case response time, end-to-end latency, and performance in comparison to the default ROS 2 executors that are available.

We note that the emphasis of our paper is the compatibility of the ROS 2 events executor with the classical scheduling theory of periodic tasks. As a result, we mainly focus on the transformation of a timer into a periodic task. We highlight that event-triggered tasks require dedicated analyses, but also provide methods to analyze some specific system configurations using existing analyses on limited preemptive scheduling. Therefore, in future work, we will further investigate the interaction of the built-in DDS and the events executor to provide more insights when both timers and subscriptions are present and interact with each other.

Finally, to be compatible with event-triggered tasks in ROS 2, such as subscriptions, we propose to add dedicated interfaces to ROS 2 to specify timing properties, which are currently missing. Towards this, we discuss two potential solutions, either involving DDS QoS settings or introducing a dedicated priority field. These additions would enable the integration of many scheduling policies into ROS 2 and provide compatibility for event-triggered tasks.

X. Acknowledgments

This work has received funding by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038.

This result is part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 865170).

This material is based upon work supported by the National Science Foundation under Grant No. 2229290.

References

[1] 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, 2022.

[2] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. Synthesizing job-level dependencies for automotive multi-rate effect chains. In International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2016.

[3] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. End-to-end timing analysis of cause-effect chains in automotive embedded systems. Journal of Systems Architecture - Embedded Systems Design, 2017.

[4] Marko Bertogna and Sanjoy Baruah. Limited preemption edf scheduling of sporadic task systems. IEEE Transactions on Industrial Informatics, 2010.

[5] Ran Bi, Xinbin Liu, Jiankang Ren, Pengfei Wang, Huawei Lv, and Guozhen Tan. Efficient maximum data age analysis for cause-effect chains in automotive systems. In DAC. ACM, 2022.

[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 Proceedings of the 42nd Real-Time Systems Symposium (RTSS), 2021.

[7] Giorgio C. Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems. A survey. IEEE Trans. Ind. Informatics, 2013.

[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. Schloss Dagstuhl, 2019.

[9] Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn B. Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil C. Audsley, Raj Rajkumar, Dionisio de Niz, and Georg von der Brüggen. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. Real Time Syst., 2019.

[10] Hyunjong Choi, Yecheng Xiang, and Hyoseung Kim. Picas: New design of priority-driven chain-aware scheduling for ROS2. In 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2021.

[11] Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto L. Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In Design Automation Conference, DAC, 2007.

[12] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. Real Time Syst., 2007.

[13] Robert I. Davis, Abhilash Thekkilakattil, Oliver Gettings, Radu Dobrin, Sasikumar Punnekkat, and Jian-Jia Chen. Exact speedup factors and sub-optimality for non-preemptive scheduling. Real Time Syst., 2018.

[14] Marco Dürr, Georg von der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. End-to-end timing analysis of sporadic cause-effect chains in distributed systems. ACM Trans. Embedded Comput. Syst. (Special Issue for CASES), 2019.

[15] Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and non-preemptive real-time uni-processor scheduling. Research Report RR-2966, Project REFLECS, 1996.

[16] Pourya Gohari, Mitra Nasri, and Jeroen Voeten. Data-age analysis for multi-rate task chains under timing uncertainty. In RTNS. ACM, 2022.

[17] Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. Timing analysis of asynchronized distributed cause-effect chains. In RTAS. IEEE, 2021.

[18] Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. Compositional timing analysis of asynchronized distributed cause-effect chains. ACM Trans. Embed. Comput. Syst., 2023.

[19] Mario Günzel, Harun Teper, Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. On the equivalence of maximum reaction time and maximum data age for cause-effect chains. In ECRTS, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[20] Mario Günzel, Niklas Ueter, Kuan-Hsun Chen, and Jian-Jia Chen. Timing analysis of cause-effect chains with heterogeneous communication mechanisms. In RTNS. ACM, 2023.

[21] Mario Günzel, Niklas Ueter, Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. Probabilistic reaction time analysis. ACM Trans. Embed. Comput. Syst., 2023.

[22] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication centric design in complex automotive embedded systems. In Euromicro Conference on Real-Time Systems, ECRTS, 2017.

[23] iRobot. Events executor. https://github.com/irobot-ros/events-executor, 2022.

[24] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of period and sporadic tasks. In RTSS. IEEE Computer Society, 1991.

[25] 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), 2022.

[26] Tomasz Kloda, Antoine Bertout, and Yves Sorel. Latency analysis for data chains of real-time periodic tasks. In IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2018.

[27] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), 2015.

[28] Tobias Kronauer, Joshwa Pohlmann, Maximilian Matthé, Till Smejkal, and Gerhard Fettweis. Latency analysis of ROS 2 multi-node systems. In 2021 IEEE international conference on multisensor fusion and integration for intelligent systems (MFI). IEEE, 2021.

[29] Ralph Lange. Mixed real-time criticality with ROS2 - the callback-group-level executor. ROSCon Lightning Talk, 2018.

[30] C. L. Liu and James W. Layland. Journal of the ACM, 1973.

[31] Mitra Nasri and Björn B. Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In RTSS. IEEE Computer Society, 2017.

[32] Open Robotics. ROS 2: Quality of Service Settings, May 2023. https://docs.ros.org/en/humble/Concepts/Intermediate/About-Quality-of-Service-Settings.html.

[33] Hoora Sobhani, Hyunjong Choi, and Hyoseung Kim. Timing Analysis and Priority-driven Enhancements of ROS 2 Multi-threaded Executors. In 2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS), 2023.

[34] 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 2020 IEEE Real-Time Systems Symposium (RTSS). IEEE, 2020.

[35] Harun Teper, Tobias Betz, Mario Günzel, Dominic Ebner, Georg von der Brüggen, Johannes Betz, and Jian-Jia Chen. End-to-end timing analysis and optimization of multi-executor ROS 2 systems. In RTAS, 2024.

[36] Harun Teper, Tobias Betz, Georg von der Brüggen, Kuan-Hsun Chen, Johannes Betz, and Jian-Jia Chen. Timing-aware ROS 2 architecture and system optimization. In 2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2023.

[37] Harun Teper, Mario Günzel, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen. End-to-end timing analysis in ROS2. In RTSS. IEEE, 2022.

[38] 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), 2022.

[39] Harun Teper, Daniel Kuhse, Mario Günzel, Georg von der Brüggen, Falk Howar, and Jian-Jia Chen. Thread carefully: Preventing starvation in the ROS 2 multithreaded executor. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2024.

[40] ROS Real time Working Group. ROS 2 reference system. https://github.com/ros-realtime/reference-system, 2023.

[41] ROS2 Real time Working Group. Autoware reference system. https://github.com/ros-realtime/reference-system, 2022.

[42] Georg von der Brüggen, Jian-Jia Chen, and Wen-Hung Huang. Schedulability and optimization analysis for non-preemptive static priority scheduling based on task utilization and blocking factors. In 27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015, 2015.

[43] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with fixed preemption points. In 2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications, 2010.