ROS^{RT}: Enabling Flexible Scheduling in ROS 2*

Sizhe Liu†, Rohan Wagle†, Shareef Ahmed‡†, Zelin Tong†, and James H. Anderson†
†*University of North Carolina at Chapel Hill, USA*
‡*University of South Florida, Tampa, USA*
Email: {sizheliu, wagle, ztong, anderson}@cs.unc.edu, shareefahmed@usf.edu

***Abstract*—The Robot Operating System 2 (ROS) is heavily used in autonomous systems due to its large ecosystem and modular design. However, ROS remains problematic for real-time applications despite prior efforts to improve its real-time capabilities. Many of these problems are rooted in the implementation of the ROS executor, which does not support preemption or user-specified priorities. These properties are fundamental to real-time scheduling and desirable for general systems to reduce latency. ROS variants in prior work have separately supported the preemption and prioritization of callbacks but place restrictions on the application, preventing their adoption in a real-world workload. This paper addresses these deficiencies with a novel executor framework, ROS^{RT}, which is compatible with any type of ROS application while supporting preemptive, priority-driven scheduling. Additionally, to support flexible EDF scheduling in ROS^{RT}, a custom EDF scheduler implementation is proposed using the new Linux scheduling class `SCHED_EXT`. ROS^{RT}'s flexibility and real-time compatibility do not come at the cost of overheads: it achieves a significant decrease in publisher-to-subscriber overhead from the native ROS executor. Finally, this paper concludes with a case study performed on the Autoware Reference System, which simulates the execution of the LiDAR module in an autonomous driving application, demonstrating the ability of ROS^{RT} at real-time scheduling in real-world scenarios.**

***Index Terms*—Real-time systems, ROS 2**

I. Introduction

The Robot Operating System 2 (ROS) is increasingly being adopted in autonomous systems because of the modularity it affords. Modularity facilitates the efficient development of complex features (*e.g.*, autonomous driving) by reducing the cost of integration: each module (*e.g.*, object detection) is kept functionally independent and communicates only through pre-defined interfaces. The ROS *executor*, responsible for scheduling ROS module functions, furthers ROS's modularity, as it is module agnostic and thus obviates the need for custom scheduling solutions for each ROS application.

The ROS executor was not designed with real-time correctness as a first-class concern, so not surprisingly, it has been one of the most heavily studied and modified ROS components by real-time researchers [1]–[6]. Specifically, ROS executors do not support preemption: each module function—a *callback*—is always executed to completion before another callback can be executed. Furthermore, ROS maintains an internal queue—the *wait set*—to dispatch callbacks. Because the wait set is internal to ROS, it cannot be configured to reflect user-preferred callback priorities without modifying the ROS library.

*Work supported by NSF grants CPS 2038960, CPS 2038855, CNS 2151829, and CPS 2333120.

Unfortunately, previously proposed mechanisms for easing these real-time-related limitations have proven to be partial fixes at best. For example, most proposed methods for providing callback prioritization do not support callback preemption [1]–[4], [7], [8]. A notable exception is a scheduling method proposed by Wilson *et al.* that supports both prioritization and preemption [5]. However, their method only supports processing chains, rather than graphs, as occur, for example, in commonly used software such as Autoware [9]. With this state of affairs, ROS developers face the dilemma of having to sacrifice at least one of preemptivity, prioritization, or usability. Motivated by this dilemma, this paper is directed at supporting preemptive real-time scheduling in ROS, without imposing restrictions on user applications.

Contributions. The ability to prioritize and preempt threads is a common and mature feature in most operating systems (OSes). Given this, richer real-time capabilities can be enabled in ROS callback scheduling by simply leveraging these OS-provided features. However, both ROS and most OSes are complex software systems, so various non-trivial issues arise when attempting to realize a new ROS variant in this way—especially if the existing ROS API is to be maintained. This paper is directed at addressing this challenge with the objective of re-designing the ROS executor so that it offloads callback scheduling entirely to the OS, thereby granting ROS greater real-time capabilities and scheduling flexibility. Our contributions are threefold:

First, we present *ROS^{RT}*, a new ROS execution model that supports callback preemption and prioritization by offloading callback scheduling to the OS.[1] ROS^{RT}'s design can be implemented for any ROS-supporting OS such as Linux and QNX.[2] Here, we focus exclusively on Linux, utilizing its `SCHED_FIFO` and `SCHED_EXT` (detailed later) scheduling classes to realize fixed-priority (FP) and earliest-deadline first (EDF) scheduling, respectively. Our implementation maintains the following properties:

1) Preemptive FP/EDF callback multicore scheduling fundamental to real-time applications.
2) Callback priorities exposed to the entire OS, which allows the ROS application to co-exist with other user processes on the same multicore hardware.

[1]The ROS^{RT} implementation can be accessed at: https://github.com/sizheliu-unc/rclcpp

[2]`SCHED_EXT` and our EDF implementation, which are independent from ROS^{RT}, are compatible only with Linux 6.12+.

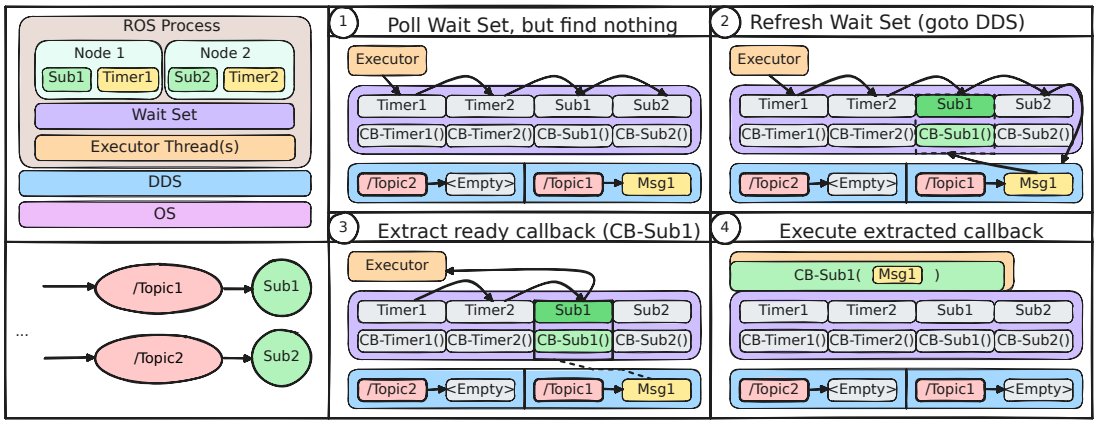

Fig. 1: Overview of a ROS process and the steps taken by the executor to find and execute a pending callback.

3) Independent priority/deadline assignment of callbacks, which provides users maximal control over scheduling.
4) Support for arbitrary ROS workloads (*e.g.*, graphs and arbitrary deadlines), which enables real-time scheduling for complex ROS applications.

Second, the lack of flexibility in Linux's EDF scheduler, `SCHED_DEADLINE`, has likely been a major stumbling block in introducing EDF scheduling to many existing ROS variants. To achieve a more flexible EDF scheduling solution, we use the new Linux scheduling class `SCHED_EXT` to implement a custom EDF scheduler that avoids limitations in `SCHED_DEADLINE` by allowing for direct user management of absolute deadlines without the restriction of sporadic callback releases, making it compatible with ROS^{RT}'s design.

Third, we show via experiments that ROS^{RT} achieves improved response times for applications while maintaining a low overhead. ROS^{RT} is shown to reduce the publisher-to-subscriber overhead (69 μs) by up to 60% compared to the multi-threaded executor in ROS (173 μs), demonstrating the superiority of ROS^{RT}'s design over the native ROS executor. ROS^{RT}'s support for real-time scheduling in real-world workloads is then tested via a case study on the Autoware Reference System, which simulates the LiDAR module on the Autoware autonomous driving system. Benefiting from its support for preemptive FP and EDF scheduling, ROS^{RT} achieved better response times compared to the native ROS and prior work [6].

Organization. In the rest of this paper, we provide relevant background (Sec. II), survey prior work (Sec. III), discuss the ROS^{RT} implementation (Sec. IV), evaluate ROS^{RT} (Sec. V), and conclude (Sec. VI).

II. Background

In this section, we introduce key concepts in ROS and discuss scheduling problems introduced by its executor.

A. The ROS Framework

The ROS framework, illustrated in Fig. 1, is used by autonomous systems to define precedence constraints for event-driven data-processing. Despite its name, ROS is not an OS, but rather a publish-subscribe framework that supports communication through shared message queues, referred to as *topics*. ROS utilizes the Data Distribution Service (DDS) [10] to facilitate message passing and maintain the message queues. Atop the DDS, ROS creates high-level abstractions for its user library, which will be discussed in this section.

ROS nodes. ROS *nodes* are independent user modules that store user-defined data structures and can *publish* (*i.e.*, transmit) messages to or *subscribe* to (*i.e.*, receive messages from) topics. Within a node, user-defined *callback* functions can process received messages and publish their output to trigger subsequent callbacks. Because each node is functionally independent, a ROS application can execute its ROS nodes in separate processes or on different machines across a network.

Callbacks and callback groups. Broadly speaking, callbacks are triggered by events in ROS: timer expiration, topic publication, service request, and service response. Each event source defines one type of callback: timer, subscription, service, and client callbacks, respectively. Callbacks are assigned to *callback groups*, which can be either *mutually exclusive* (the default) or *reentrant*. Callbacks within the same mutually exclusive callback group cannot execute parallel to each other or to themselves, whereas reentrant callback groups do not have this limitation. We say a callback is *ready* if it is triggered but yet to be executed. Ready callbacks are invoked by a ROS *executor*, which we discuss next.

Executors. A ROS executor is a component in a ROS process that identifies and invokes ready callbacks. There are primarily two types of ROS executors in the existing ROS framework: *single-threaded* and *multi-threaded*, represented by one and multiple threads in the OS, respectively. The executor also maintains a data structure called the *wait set*[3], which stores registered callbacks, in the order of timers, subscriptions, services, and clients; callbacks within the same category are ordered by their registration order in ROS [11]. Note that the wait set only stores callbacks, not messages, which are instead stored in the DDS.

[3]This data structure is defined as `wait_set` in ROS but referred to as *readySet* in some prior work [2], [11].

Steps 1–4 in Fig. 1 demonstrate the workflow of an executor thread. When an executor thread is idle (*i.e.*, not executing a callback), it iterates through the wait set to acquire the next-to-be-executed callback (step 1). If the wait set contains no ready callbacks, the executor thread will update the wait set by checking the system clock for expired timers, poll the DDS for new messages (step 2), and return to step 1. The point in time when this refresh occurs is called a *polling point*, and each pair of consecutive polling points defines a *processing window*. If the wait set contains a ready callback, the executor will extract the first ready callback found (step 3). The executor will then retrieve the pending message from the DDS (for non-timer callbacks), execute the acquired callback, and remain busy until the callback completes (step 4).

A multi-threaded executor contains multiple, independent executor threads, each performing the aforementioned steps, but shares a single wait set. The number of executor threads is statically configured by the user. A mutual exclusion lock is used to arbitrate executor threads' access to the wait set. The lock must be acquired before accessing the wait set (at step 1), and then released when a ready callback is found and extracted from the wait set for execution (at step 3). Multi-threaded executors are commonly used to support modules with complex functionalities, owing to the greater parallelism they afford compared to single-threaded executors.

The DDS and ROS middleware. ROS's underlying publish and subscribe functionalities are realized by the DDS, a standard that defines a publish-subscribe message passing interface upon concrete communication protocols (*e.g.*, network, shared memory, etc.). Several implementations of the DDS standard are supported by ROS, *e.g.*, FastDDS and CycloneDDS, each implementing the DDS model with its own API. To accommodate this, ROS implements the ROS Middleware (RMW), a set of libraries that provide a unified interface across multiple DDS implementations. Consequently, when a callback invokes the ROS publish function, the message is passed to the DDS via the RMW interface.

Many DDS implementations maintain a set of threads to handle message passing. When a message is received, it is stored in a message buffer until a ROS process retrieves it. If multiple messages are received on the same topic, ROS will retrieve them in FIFO order.

The DDS is configurable via *Quality of Services* (QoS) policies [12], which can be used to control the message buffer size, reliability of delivery, etc. Additionally, for each subscription, users can register an *event handler* function to be invoked within the DDS threads immediately when new messages arrive.

Alternatively, ROS applications contained within a single process can be configured to use *intra-process communication*. This mechanism, implemented in the RMW, bypasses the DDS to pass messages directly within the process's private address space. This reduces communication overhead by avoiding costly OS operations.

Processing graphs and chains. Processing *graphs* and *chains* are not entities in the ROS library but are abstractions commonly used for the representation and analysis of ROS applications. Analytically, a directed graph can be constructed to represent a ROS workload where each vertex represents a callback and each edge a precedence constraint. Timer callbacks have event sources external to ROS (*i.e.*, system clocks), and therefore are represented as source vertices (vertices with an in-degree of zero) in the processing graph. If the processing graph is acyclic,[4] processing chains can be derived from the graph, where each source-to-sink path represents a chain.

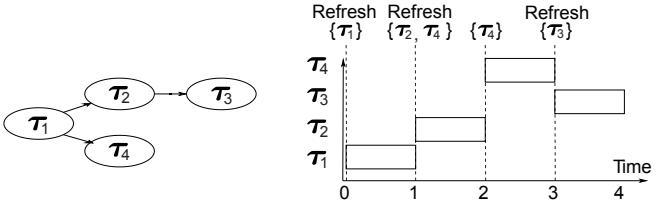

Fig. 2: A schedule demonstrating the polling points problem in ROS. The callbacks in the wait set at each time unit are shown above the schedule.

We call a workload *chain-based* if the vertices have an in-degree and out-degree of at most one. Otherwise, the workload is defined as *graph-based*. A chain-based workload in ROS effectively means that each callback publishes to a single topic, and each topic is subscribed to by a single callback. Both chain- [1]–[4], [11] and graph-based schedulability analyses [13]–[18] are well studied in the real-time systems literature but will not be the primary focus of this paper.

B. Problems in the ROS executor

As discussed in Sec. I, ROS executors do not support callback preemption or prioritization. Callbacks in ROS are non-preemptive with respect to other callbacks in that they are always executed to completion by the executor. Priority support is also lacking in ROS, mainly because the user cannot explicitly control the invocation order of callbacks.

The polling points problem. Besides problems directly related to scheduling, polling points present another challenge in ROS scheduling: the wait set contains only a subset of all ready callbacks in the system within each processing window [11]. This breaks the common assumption in the publisher-subscriber execution model that a callback is ready whenever its predecessor completes, as the callback must also wait for the wait set to be refreshed before it becomes visible to the executor. Since refresh occurs when all callbacks from the previous refresh have been executed or are being executed, the durations of processing windows depend on the execution times of each callback and thus can be unpredictable in timing.

Example. Fig. 2 illustrates the polling points problem with four callbacks, τ_1, τ_2, τ_3 and τ_4, on a single-threaded executor (multi-threaded executors are similar), with all callbacks having an execution time of 1 time unit. In this example, τ_1, a timer callback, publishes to a topic subscribed to by τ_2 and

[4]ROS does not prohibit cycles or self-loops in the processing graph.

TABLE I: Comparison of prior work with ROS^{RT} (Ours).

Work	Multi-Threaded	Preemptive	OS-Enforced Priority	Arbitrary Workload	Support FP	Support EDF
⇒ **ROS^{RT} (Ours)**	Yes	Yes	Yes	Yes	Yes	Yes
ROS 2 (native)	Yes	No	No	Yes	No	No
CallbackIsolated[1][19]	Yes	Yes	Yes	No	Yes	No[2]
Micro-ROS [20]	Yes	Yes	Yes	No	Yes	No
Teper *et al.* [6]	No	No	No[3]	Yes	Yes	Yes
Wilson *et al.* [5]	Yes	Yes	Yes	No	No	No[2]
Arafat *et al.* [4]	Yes	No	No	Yes	Yes	Yes
PiCAS [1], [3]	Yes	No[4]	Yes	Yes	Yes	No

[1] This approach's exact mechanism is unclear.
[2] Supports `SCHED_DEADLINE`, but callbacks must release sporadically.
[3] EDF priority is not OS-enforced.
[4] Limited preemption via multiple single-threaded executors.

τ_4. After its completion, τ_2 publishes to a topic subscribed to by τ_3. We assume that the callbacks are registered in their index order. Thus, τ_3 should be prioritized in ROS over τ_4 if both are ready in the wait set. However, as shown in the figure, at time unit 2, even though both τ_4 and τ_3 are ready (as τ_1 and τ_2 have completed), τ_4 is executed before τ_3. This is because at time unit 2, the wait set, which only contains τ_4, is non-empty and thus not refreshed.

Imprecise Timers. Timers in ROS are inherently imprecise, due to the wait set's polling-based implementation. Armed timers are scanned by an executor thread as it refreshes the wait set. If a timer has expired, the executor invokes the timer's registered callback. Thus, each timer callback's release can be delayed by up to one processing window.

III. Prior Work

Prior work has improved on the ROS multi-threaded executor in two main directions. Some methods maintained the original ROS executor design but replaced the wait set with a prioritized queuing mechanism [1], [3], [4], [6]. Some have studied the assignment of executor threads to a group of callbacks to achieve preemptivity [1], [5], [19], [20]. In this section, we compare these methods in several aspects and point out their limitations. Table I provides an overview of existing ROS executor designs and their properties.[5] Our work will be discussed later in Sec. IV.

Designs. Fig. 3 provides a simplified overview of existing ROS executor designs, which can be grouped into four categories. As shown in Fig. 3 (top-left), a ROS executor thread obtains ready callbacks from the wait set (steps 2–3) and invokes them directly (step 4). Most existing work retained this design [1], [3], [4], but replaced the wait set with a priority queue to enable FP or EDF scheduling. To overcome the polling points problem (the wait set contains only a subset of all ready callbacks), these methods refresh the wait set every time before obtaining the next callback.

[5]The exact mechanism in the CallbackIsolated [19] is unclear to us. The properties included are inferred from their high-level discussion and may not be accurate. For this reason, we do not consider this work in later discussions.

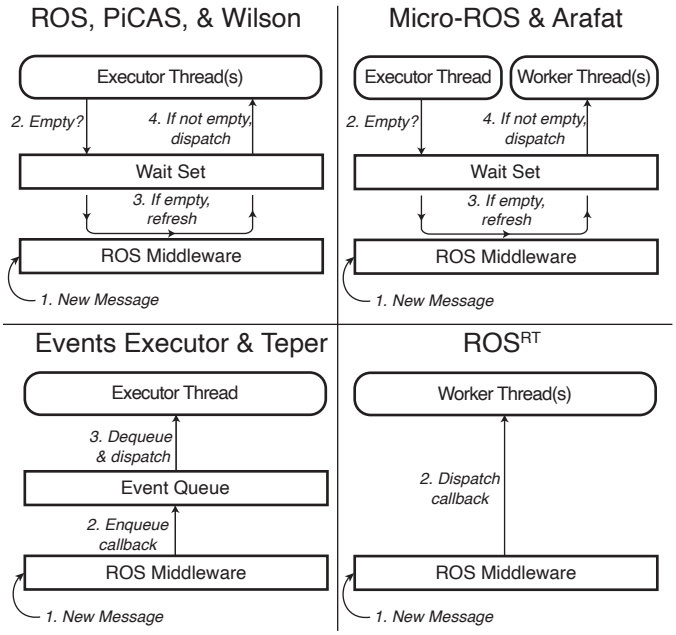

Fig. 3: Four ROS executor designs handling a new message. The order of operations are numbered.

Instead of invoking the callbacks directly, Micro-ROS [20] and Wilson *et al.* [5] (Fig. 3, top-right) assign each subscription a dedicated worker thread to execute its callback but still obtain incoming callbacks from the wait set in an executor thread (steps 2–3). The executor thread works in the same manner as the native ROS executor, except that it dispatches the incoming callbacks to their pre-allocated threads (step 4).

The *events executor* (Fig. 3, bottom-left) is an experimental feature in ROS that avoids polling from the wait set. To achieve this, the wait set is replaced by a queue data structure called the *event queue*. The RMW registers DDS event handler functions that enqueue new callbacks into the event queue as messages arrive (step 2). The executor thread thus directly dequeues from the event queue to obtain the next callback (step 3) without involving the RMW. This method addresses the polling points problem as the event queue contains all ready callbacks. It also reduces overhead by avoiding unnecessary polling. Since timers do not have event handlers, the events

executor manages them in a separate thread called the *timer management thread*. This thread is responsible for periodically enqueuing timer callbacks into the event queue by using the `sleep` function in Linux, which ultimately utilizes timer interrupts to self-suspend for a fixed amount of time.

Teper *et al.* further improved the event queue by using a priority queue [6]. However, only a single thread is responsible for executing callbacks within the events executor, and the current implementation forbids sharing a single event queue with multiple events executors, making it compatible only with single-core or partitioned systems.

Preemptive scheduling. Preemptive scheduling of callbacks improves schedulability and response times of ROS applications, by eliminating blocking and priority inversion caused by non-preemptivity. Among the works we have surveyed, only Micro-ROS [20] and Wilson *et al.* [5] considered preemptive callback scheduling. These two variants pre-allocate a single thread per callback function, thereby enabling preemption. Wilson *et al.* combined this design with Linux's `SCHED_DEADLINE` scheduler policy to implement preemptive EDF scheduling for ROS callbacks. However, as a consequence, the design inherits the sporadic release model assumed by `SCHED_DEADLINE`. This can prevent a callback from executing, even if its predecessors have completed and sufficient processors are available.

OS-enforced priority. Callbacks are executed in the context of a thread. Therefore, once dispatched, the scheduling of a callback's execution is ultimately determined by how the OS schedules the thread. In practice, a ROS thread rarely executes in complete isolation. Competing workloads, such as kernel threads, DDS threads, or other user-space applications (including other ROS applications) may access the processors simultaneously. If callback priorities are not exposed to the OS by the thread, then callbacks may experience priority inversion or blocking. For example, some ROS variants use system timer interrupts to implement ROS timers. If a ROS thread were given sufficiently high priority, it could block the kernel thread responsible for handling system timer interrupts, delaying the ROS timers. Such problems may compromise the performance of ROS applications and even the entire system. It is therefore necessary that callback priorities are enforced system-wide (*i.e.*, at the OS level).

ROS executor designs that support preemptive scheduling by executing callbacks in dedicated threads [5], [20] inherently support OS-enforced priorities, because these methods rely on OS scheduling instead of user-space scheduling: each callback's priority is reflected directly in its respective thread's OS priority. Furthermore, the methods proposed by Teper *et al.* [6] and PiCAS [1], [3] both support OS-enforced priorities for FP scheduling by dynamically assigning the executor thread the priority of a callback before executing it.

However, enforcing callback priorities at the OS level under EDF scheduling is less straightforward (discussed later). For this reason, existing ROS variants with EDF support [4], [6] perform scheduling in user space and assign fixed OS-level priorities to the underlying executor threads, leaving the absolute deadlines hidden from the OS.

To minimize the chance of priority inversion or blocking, these ROS variants rely on CPU isolation to grant each ROS process exclusive access to a set of processors. While this prohibits other user-space applications from contending for the isolated processors, the ROS application will still share the processors with some kernel threads and DDS threads.

Arbitrary workloads. Ultimately, the goal of improving the ROS executors is to directly benefit a diverse group of real-time ROS applications such as Autoware [9]. Thus, it is important that the proposed design does not exclude workload types that are already supported by the original ROS framework. The design from Wilson *et al.* [5] and Micro-ROS [20] allocates only one thread for each callback, and thus each callback cannot run in parallel to itself (*i.e.*, callbacks must be mutually exclusive). In both implementations, the executor will be forced to terminate if such a scenario is encountered. Wilson *et al.* further constrain the workload such that each callback can only publish to a single topic.

Most of the existing ROS modifications are highly dependent on the software architecture of the ROS version that their development is based upon. These modifications may become incompatible with later ROS versions ([4], [5]), which often break backward compatibility, or rely on experimental features of ROS ([6]). Micro-ROS [20], on the other hand, is forked from ROS, and the preemptive callback executor design is not featured in the main release of Micro-ROS [21].

Support for EDF and FP scheduling. FP and EDF scheduling policies are staples in real-time applications. Both policies have been studied in prior work and are straightforward to implement in user-space callback schedulers. Enforcing callback priorities at the OS level for FP scheduling is straightforward using `SCHED_FIFO`, and has been covered in prior works [1], [3], [4], [6]. However, no prior work has implemented EDF callback scheduling with OS-enforced priorities. This can be attributed to the lack of user control over absolute deadlines and the enforced sporadic release model assumed by Linux's deadline-based scheduler, `SCHED_DEADLINE`, which is incompatible with ROS. This is because non-timer callbacks do not have a minimum inter-release spacing; they release as soon as their predecessor completes. For this reason, although Wilson *et al.* [5] utilized `SCHED_DEADLINE` to implement deadline scheduling for callbacks at the OS level, any non-timer callback must wait until its next period begins before it can be released, leading to unnecessary delays.

Summary. As demonstrated in this section, all existing ROS callback scheduling methods have limitations. The least supported feature among existing ROS variants is preemptive callback scheduling. This is because most existing ROS variants follow a user-space scheduling design, making preemption difficult to realize. Besides problems within the ROS framework, `SCHED_DEADLINE` has been an obstacle for existing methods to implement EDF callback scheduling due to its inflexibility.

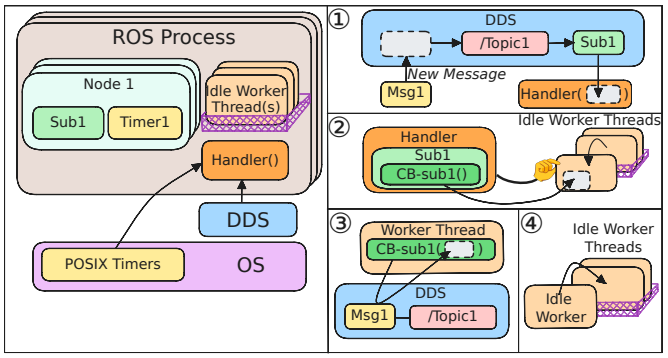

Fig. 4: ROS^{RT} and its callback scheduling mechanism.

We next present the design of ROS^{RT} as a unifying solution to these problems.

IV. Proposed Design

We propose ROS^{RT}, a fully event-based, preemptive ROS scheduling framework that can support all aspects discussed in Sec. III. In this section, we describe how this is achieved in the design and implementation of ROS^{RT}.

A. The ROS^{RT} Execution Framework

Fig. 4 illustrates ROS^{RT}'s design. Instead of relying on the user-space scheduling mechanisms of the ROS executors, where preemption is difficult to implement, ROS^{RT} offloads the execution of callback functions to OS threads. When callbacks become ready, ROS^{RT} assigns idle threads (or creates new ones) to execute them, delegating their scheduling to the OS.

Distinctions from prior work. Unlike most prior ROS variants [1], [3]–[5] that rely on polling the wait set to be notified of ready callbacks, ROS^{RT} takes an event-driven approach. Like the events executor, ROS^{RT} registers an event handler function with the DDS that is immediately invoked when a new message is received. However, ROS^{RT}'s event handler does not enqueue ready callbacks in an event queue structure, instead dispatching them directly to OS threads. This solution supports preemptive, priority-driven scheduling with minimal overheads by avoiding the unnecessary polling of the wait set and non-preemptivity from user-space scheduling.

Programming interface. ROS^{RT} retains all existing ROS interfaces, making its integration into existing applications seamless. ROS^{RT} introduces two new functions in the ROS `Subscription` class to assign callback priorities, and the new `NoExecutor` class, a drop-in replacement for the standard executors that enables OS-level scheduling. These new interfaces are listed in Table II. All other ROS interfaces remain unchanged.

The two optional functions added to the `Subscription` class, `set_sched_attr` (if using FP) and `set_edf_attr` (if using EDF), allow users to assign OS-level and EDF scheduling attributes, respectively, to a subscription's associated callback. If no scheduling attribute is defined for a callback, then the callback is scheduled with `SCHED_FIFO` priority level 1.

Listing 1 Example ROS^{RT} usage.

```
1  class UserNode: public Node {
2    UserNode() {
3      sub = new Subscription(...);
4      // Assign priorities.
5      sub->set_sched_attr(user_sched_attr);
6    }
7    Subscription* sub;
8    ...
9  };
10
11 int main() {
12   ros_init();
13   UserNode* user_node = new UserNode();
14   // In ROS: MultiThreadedExecutor executor(...)
15   NoExecutor executor;
16   executor.add_node(user_node);
17   executor.spin();
18   ros_deinit();
19 }
```

Listing 1 provides a simple example of ROS^{RT} usage, with lines that differ from original ROS usage highlighted: the assignment of priorities and the initialization of the `NoExecutor` object. The `NoExecutor` class implements the same interface as the standard single-threaded and multi-threaded ROS executors. Users create an instance of the `NoExecutor` (line 15) and add nodes to it with `add_node` (line 16). The user then invokes the `spin` function (line 17) to begin executing the ROS workload. In the native ROS executors, this function invokes the executor in the current thread (and spawns additional threads if using the multi-threaded executor). In ROS^{RT}, we have repurposed this function to enable the message event handlers and then self-suspend the current thread. This is because ROS^{RT} does not rely on the wait set or event queue, obviating the need for an executor thread.

B. ROS^{RT} Implementation

We now present the implementation of three primary components of ROS^{RT}: event handlers, worker threads, and timers. We also discuss how ROS^{RT} interacts with RMW and the DDS.

Event handlers. ROS^{RT} is fully event-based. This is made possible with the event handlers invoked directly from the DDS[6] when new messages arrive (Fig. 4, step 1). Listing 2 is a simplified implementation of the subscription event handler in ROS^{RT} (handlers for other ROS event types are similar), which is shared by all subscriptions.

The event handler takes two arguments: the incoming subscription and the number of messages for the subscription. The number of messages is almost always one, as the handler is immediately notified when each message is received.[7] For each message, the handler simply obtains an idle thread or creates a new one if none is available (lines 3–6, Fig. 4, step 2). Then, the handler assigns the callback's scheduling parameters to

[6]More accurately, the DDS invokes the encapsulated event handler in the RMW, which in turn invokes the user-defined event handlers.

[7]In our experience, we have not encountered any case where the number of messages was greater than one.

TABLE II: Interfaces created by ROS^{RT}

Class	Function	Purpose
NoExecutor	add_node(Node* node)[1]	Add ROS Node object for execution.
	start()	Start receiving events.
	stop()	Stop receiving events.
	spin()[1]	Start and suspend the current thread.
Subscription[2]	set_sched_attr(SchedAttr* attr)	Set the (OS) scheduling attribute.
	set_edf_attr(EDFAttr* attr))	Set the EDF scheduling attribute.

[1] Override of original ROS functions.

[2] Also applies to other event sources, such as timers.

Listing 2 The implementation of a subscription handler.

```
1  void SubscriptionHandler(Subscription sub, size_t
   ↪  num_messages) {
2    while (num_messages--) {
3      ThreadData* thread_data = idle_threads.pop();
4      if (thread_data == nullptr) {
5        thread_data = new ThreadData;
6      }
7      /* set either FP or EDF priority */
8      thread_data->SetPriority(sub.priority);
9      /* dispatch callback and wake up worker */
10     thread_data->Dispatch(sub.callback);
11     thread_data->is_idle = false;
12   }
13 }
14 /* below is for illustration, the handler is
   ↪  automatically set in add_node()*/
15 Subscription my_subscription;
16 my_subscription.set_handler(subscription_handler);
```

Listing 3 The implementation of a worker thread.

```
1  struct ThreadData {
2      bool is_idle, is_mutually_exclusive;
3      mutex callback_group_mutex;
4      void (*callback)(Message m);
5      ...
6  };
7
8  void WorkerThreadFunc(ThreadData* thread_data) {
9    while (1) {
10     /* Thread is idle; push onto a stack */
11     thread_data->is_idle = true;
12     idle_threads.push(thread_data);
13     /* Suspend until being dispatched */
14     wait_till_equal(thread_data->is_idle, false);
15     if (thread_data->is_mutually_exclusive) {
16       lock(thread_data->callback_group_mutex);
17     }
18     Message new_msg = take_message();
19     thread_data->callback(new_msg);
20     if (thread_data->is_mutually_exclusive) {
21       unlock(thread_data->callback_group_mutex);
22     }
23   }
24 }
```

the chosen thread (line 8), before dispatching it to execute the callback (line 9).

To enable this handler, it must be registered for each existing subscription object via the `Subscription` member function `set_on_new_message_callback` (simplified as `set_handler` in Listing 2), which exists in standard ROS. This function passes the handler as a `DataReaderListener` to the DDS, which is then invoked directly by the DDS when new messages for the subscription arrive. Handlers for all subscriptions in a `Node` object are automatically registered when the user calls the `add_node` function as a common ROS programming practice.

Worker threads. In ROS^{RT}, callbacks are executed by worker threads (Fig. 4, step 3) that are scheduled by the OS. The implementation of worker threads is shown as the `WorkerThreadFunc` function in Listing 3, with each thread represented by a unique `ThreadData` object. As illustrated, the worker thread executes in an infinite loop. At the beginning of each iteration, the worker thread makes itself available to the event handler by adding itself to the pool[8] of idle threads (Fig. 4) and self-suspends (lines 10–11). After a callback is dispatched by the event handler (shown in Listing 2, line 9) to the worker thread, the worker thread will begin to execute the callback.

[8]A stack data structure is used to store idle threads to maintain cache affinity—the most recently executed thread is most likely to retain its cache lines on the CPU. The benefit of doing so varies depending on the hardware platform and architecture.

Before executing a mutually exclusive callback, the thread acquires the callback group's mutex lock (lines 14–16), which will be released upon the callback's completion. The callback execution in ROS^{RT} is identical to the native ROS: the message is retrieved from the DDS through a function called `take_message`, which is then used as an argument when invoking the callback (lines 17–18).

With this design, ROS^{RT} is able to handle reentrant callbacks, as each worker thread is not bound to a specific subscription, and new threads will be created on the fly if more worker threads are needed to handle new messages. One potential concern may be that thread creation would incur a high overhead on the system.

To test the validity of this concern, we performed two experiments on the thread creation overhead. In the first experiment shown in Fig. 5, we ran a process that creates 10,000 threads sequentially via the `pthread_create` function. The duration of each `pthread_create` invocation is measured. After each thread is created, the main thread waits until the created thread completes before creating another. This result represents the overhead for thread creation in the average case, as ROS applications rarely receive a large number of new messages simultaneously. As shown in the result, the creation of threads can be completed within 13 μs in most cases.

In the second experiment shown in Fig. 5, we test the thread creation times under extreme demand by creating 10,000 threads sequentially and measuring the duration of the `pthread_create` invocation for each individual thread created without waiting for the completion of each thread (as was the case in the previous experiment). This scenario is almost impossible to occur in a meaningful ROS workload, as it requires 10,000 new messages to arrive at the same time. Thus, this data can serve as an upper bound on the thread creation times. Despite involving an extreme number of threads, the overhead for creating each individual thread still remains relatively low (34 μs). From the results of these two experiments, we can conclude that thread creation does not incur a high overhead, making it unnecessary to pre-allocate the worker threads or restrict the number of threads created.

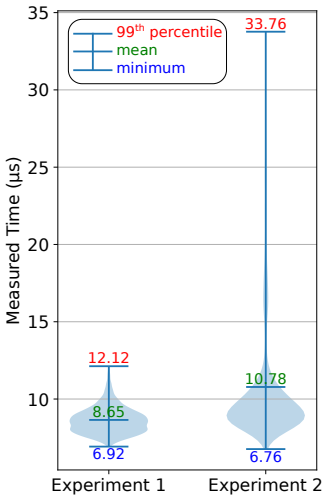

Fig. 5: Thread creation times in two experiments.

Timers. As ROS timers do not involve the DDS, they do not have a mechanism for registering event handlers and thus must be handled ad hoc. For each ROS timer defined by the user, ROS^{RT} registers a POSIX interval timer, which signals the main thread (the starting thread of the application) when the timer expires (Fig. 4). Upon receiving these signals, the main thread's signal handler (similar to subscription event handlers in steps 2 and 3 of Fig. 4) will assign or create worker threads to execute the timer callbacks. This timer design is similar to that of the events executor (used by Teper *et al.* [6]), as both offload timers to the OS, whereas the conventional ROS executors manage timers in user space.

Interactions with the DDS. As demonstrated in Listing 2, the subscription event handlers, which are invoked within the DDS as new ROS messages arrive, do not directly retrieve the incoming messages. Instead, ROS messages are retrieved only when their respective callbacks are invoked (Listing 3, line 6). This mechanism is identical to the native executors: the wait set only stores ready callbacks, which will retrieve ROS messages as they begin execution. Thus, messages from the same topic will always be retrieved and (begin to be) processed in the FIFO order, regardless of the scheduling algorithm. Furthermore, since ROS^{RT} does not involve any changes to the DDS, the QoS policies in the native ROS are compatible with ROS^{RT}. If a message is dropped by the DDS due to QoS policies (*e.g.*, message buffer overflow), then the callback will not execute at line 6 of Listing 3.[9]

Compatibility with intra-process communication. Recall from Sec. II that ROS applications can use intra-process communication to bypass the DDS. In this case, the event handlers are directly invoked by the RMW within the user application when the publish function is called. Therefore, ROS^{RT} is compatible with intra-process communication even though it does not involve the DDS.

Summary. At this point, we have discussed how we enabled preemptive callback scheduling in ROS. The remaining task is to allow the worker threads to execute with user-defined priorities in the OS. In the case of FP scheduling, this can be easily achieved by assigning worker threads to their desired `SCHED_FIFO` priorities before invoking them (Listing 2, line 8). However, to enable EDF scheduling, where each callback is assigned a deadline relative to its release (or relative to its originating source timer), is not as straightforward. Here, the main obstacle is `SCHED_DEADLINE`'s lack of flexibility, which we address next.

C. EDF scheduling in Linux.

Although Linux implements an EDF constant-bandwidth server scheduler (EDF-CBS, *i.e.*, `SCHED_DEADLINE`), using it to schedule the worker threads in ROS^{RT} would be problematic mainly for two reasons.

First, `SCHED_DEADLINE` is the highest priority scheduling class in Linux. Therefore, if the worker threads were using `SCHED_DEADLINE` in ROS^{RT}, then the DDS threads, which receive and publish new messages, would also need to use `SCHED_DEADLINE`. This is because the DDS can otherwise be starved by worker threads, preventing the ROS application from receiving or publishing new messages. As DDS threads do not follow a sporadic release model as assumed by `SCHED_DEADLINE`, it is unlikely that new messages will always be received (or published) on time.

Second, if ROS^{RT} were to use `SCHED_DEADLINE`, it would need to dynamically configure a worker thread's scheduling parameters (runtime budget, deadline, and period) when the thread is assigned a new callback, since each worker thread is not bound to a particular callback. However, when a thread becomes inactive (*i.e.*, idle), `SCHED_DEADLINE` still considers the thread to be contributing to system utilization, and only removes it from the system after a "cool-down" period [22]. During this period, the thread's scheduling parameters cannot be modified, and thus preventing ROS^{RT} from re-using it. Instead, ROS^{RT} must create redundant threads to service ready callbacks promptly.

Unfortunately, by doing so, another problem arises: both the newly created threads and the inactive threads contribute to system utilization. This over-accounting can cause `SCHED_DEADLINE` to reject the newly created threads unnecessarily, believing the system to be over-utilized. In such a scenario, ready callbacks cannot be immediately dispatched into a thread, delaying their execution even if they have enough priority to execute and the system has idle capacity.

A custom EDF scheduler on Linux. To achieve EDF scheduling without these problems in `SCHED_DEADLINE`, we

[9]This is handled automatically by the callback invocation logic in ROS.

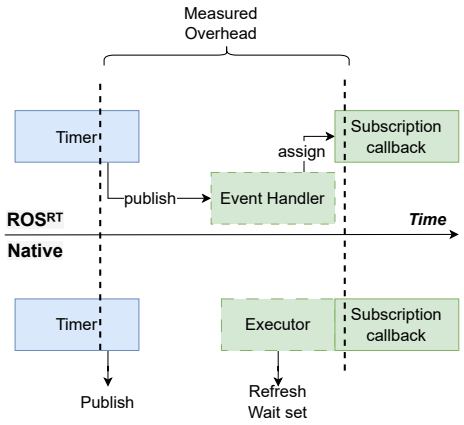

Fig. 6: An illustration of the subscription overhead. The top (resp. bottom) figure illustrates the overhead composition for ROS^{RT} (resp. native ROS). Boxes in blue (resp. green) represents the publisher (resp. blue) process. Boxes in dashed lines represents functionalities not controlled by the user.

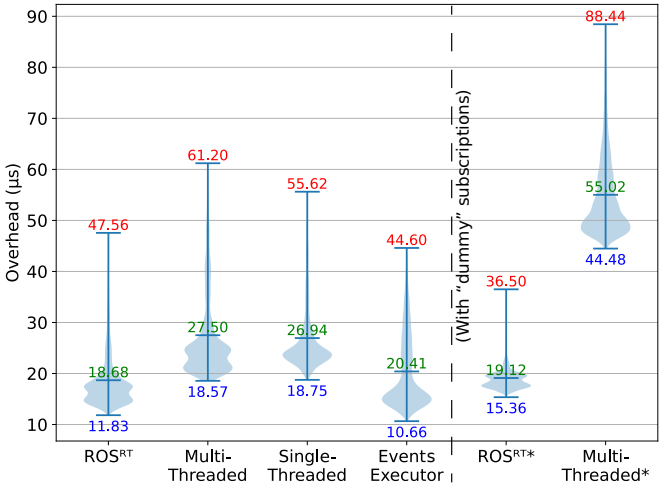

Fig. 7: The measured subscription overheads over a total of 50,000 invocations, marked with the 99th percentiles (red), the means (green), and the minimums (blue).

leverage the Linux kernel's new extensible scheduling class, `SCHED_EXT` [23], to implement a lightweight EDF scheduler. Our custom EDF scheduler prioritizes threads only by their absolute deadline, which can be updated by ROS^{RT} through a single system call. This simple design agrees with the event-driven execution model of ROS: non-timer callbacks do not follow a sporadic release model.

The `SCHED_EXT` scheduling class has a lower priority than `SCHED_FIFO`. This prevents worker threads from preempting or starving DDS threads, which can be configured with `SCHED_FIFO` priorities. It also supports the dynamic configuration of a thread's absolute deadline without restrictions. Our `SCHED_EXT` scheduler can be dynamically loaded and unloaded from user space at runtime within or outside of the ROS^{RT} process without any modification of the kernel.

V. Evaluation

In this section, we evaluate ROS^{RT}. We first evaluate overheads incurred by ROS^{RT} in comparison to the native ROS executors, and prior work [6]. Then, we present a case study of our framework on the Autoware Reference Systems [24].

We performed our evaluations on an AMD Ryzen 9 7950X3D 16-core processor with 64GB of RAM. The machine was configured with Ubuntu 22.04 and the 6.14.5 release of the Linux kernel. Our ROS^{RT} executor was implemented in the "Humble Hawksbill" LTS release of ROS's rclcpp library, and used eProsima's Fast-DDS version 2.6 for the experiments.

A. Overhead and Jitter

In ROS and ROS^{RT}, there are non-negligible overhead costs incurred by publishing and receiving messages, and handling timer signals. In this section, we study the overheads and jitters in ROS^{RT}, the native ROS executors (single- and multi-threaded), and the events executor used by Teper *et al.* [6].

The results presented in this section can also serve as a meaningful reference to the advantage of ROS^{RT} over other existing ROS variants. This is because most variants [1], [3]–[5], [20] did not modify ROS timers or ROS's wait set refresh mechanism, the primary contributors of overheads in ROS. They would perform similar to the native ROS executors, if not worse (cf. [3, page 116, Fig. 8], [4, page 3741, Fig. 6]).

Subscriptions. In ROS and ROS^{RT}, there is a non-negligible delay between publishing a message and a subscriber invoking its respective callback. We define this delay as the *subscription overhead*. We performed three experiments to compare the subscription overhead incurred by ROS^{RT}, the native single- and multi-threaded executors, and the events executor.

In the first experiment, we constructed a pair of publisher and subscriber node, running in separate processes on the same machine, as shown in Fig. 6. To accurately capture the overhead, we ensured that the subscription callback could always execute immediately when it was ready by assigning it a high priority and providing it with a sufficient number of processor cores. We configured the publisher to publish to a topic via a timer callback every 10 ms. Each message included a timestamp denoting the time just before the publish function was called. When the subscriber received the message, it would execute a callback that recorded another timestamp, denoting when the callback was invoked. Then, it would calculate the difference between the timestamps as the subscription overhead.

We collected data from five trials, each consisting of 10,000 invocations of the subscription callback. As shown in Fig. 7, the 99th percentile overhead in ROS^{RT} (47 μs) is significantly lower than in the native ROS executors (61 μs and 55 μs). This overhead reduction comes directly from ROS^{RT}'s design: ROS^{RT} does not rely on polling from the wait set, which incurs unnecessary overhead. The events executor similarly avoids the wait set and thus has almost identical 99th percentile overhead (45 μs) to that of ROS^{RT}.

Methods using the ROS wait set [1], [3]–[5], [20] data structure have an additional disadvantage: at each wait-set

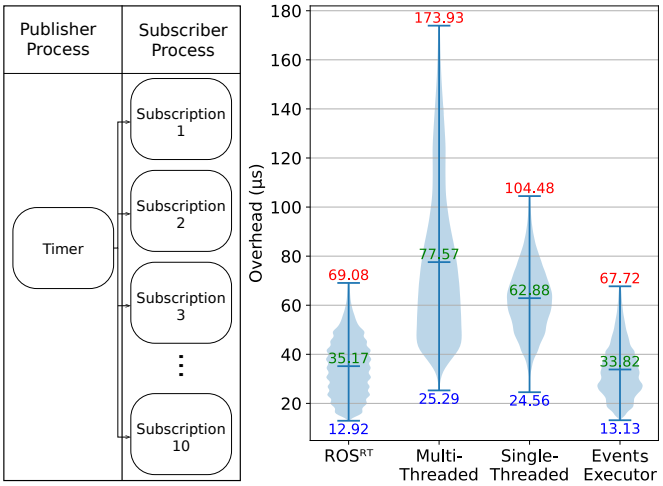

Fig. 8: The subscription overheads of 10 subscriptions subscribing to the same topic processed simultaneously.

refresh, the executor must check each of the registered subscriptions by polling from the DDS, even if they are not ready. This can incur significant latencies in ROS workloads with many subscribers. On the other hand, both ROS^{RT} and the events executor avoid polling from the DDS and thus remain unaffected by the existence of unready callbacks.

To demonstrate this disadvantage in using the wait set, we additionally constructed 100 "dummy" subscriptions that subscribe to a non-existent topic and repeated the experiment for the native ROS multi-threaded executor and ROS^{RT}. As these subscriptions would never receive any message, ideally, they should not impact performance. However, as can be observed in Fig. 7 (right, marked with "*"), with the native ROS executor, the 99th percentile overhead increased significantly compared to the previous test ($61 \rightarrow 88\ \mu s$), while the subscription overhead remained almost unchanged on ROS^{RT} as expected, with the minimum overhead slightly higher.

Finally, to further demonstrate the overhead advantages of ROS^{RT} under stress, we created 10 subscriptions that subscribe to the same topic (illustrated in Fig. 8, left). In this case, all 10 subscriptions will be triggered and processed simultaneously. Because the single-threaded executor and the events executor contain only one thread for callback execution, executing the workload outright in one executor would unfairly disadvantage these two variants. Thus, for these two variants, we created 10 threads, each given its own executor and a single subscription, enabling them to process the subscriptions in parallel.

We executed this workload with the four executor variants. Fig. 8 (right) shows the experiment result. Here, all executors have higher overheads than in the scenario of a single subscription. However, the increases in the 99th percentile overhead experienced by ROS^{RT} ($47 \rightarrow 69\ \mu s$) and the events executor ($45 \rightarrow 67\ \mu s$) are much smaller than the additional overhead experienced by the native ROS multi- and single-threaded executors ($61 \rightarrow 174\ \mu s$ and $55 \rightarrow 104\ \mu s$).

The increased overhead in this experiment is due to contention. In the native ROS multi-threaded executor, locking is used to ensure atomic access to the wait set. This invariably incurs delays when multiple subscriptions are triggered at the same time. ROS^{RT} and the events executor are free from lock contention, but the event handlers are all processed in a single (DDS) thread,[10] leading to the increased overhead.

Timers. Ideally, a timer callback would be released precisely at the beginning of each period. Even if there are overheads involved in triggering the callback function, as long as these overheads remain constant, the timer callback releases would remain one period apart. In practice, however, these overheads can vary between iterations. These deviations from the desired timer triggering times are called *jitters*. Jitters can cause unwanted delays in the system (if late) or introduce higher interference than assumed (if early).

As ROS and ROS^{RT} employ different methods to handle timers, we measured the jitters experienced by timers in this experiment to demonstrate the soundness of our design. Here, we define jitter as $(this\ callback\ release - last\ callback\ release - period)$. Since the actual release times of callbacks are difficult to obtain, we measure the user-perceived release times instead (*i.e.*, the start time of the timer callbacks, given sufficient priorities and processors).

Table III shows the measured jitters of a single timer in ROS^{RT}, native ROS, and the events executor as the period varies. Here, ROS^{RT} and the events executor performed similarly, as both ultimately rely on clock interrupts. In contrast, the native ROS executor performed significantly worse than ROS^{RT} in almost all cases owing to the polling mechanism in the wait set.

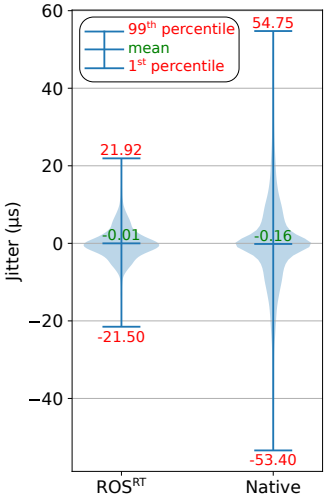

Fig. 9: Jitters of ten timers in ROS^{RT} and the native executor.

To demonstrate ROS^{RT}'s performance under stress, we increased the number of timers to ten and fixed the timer periods to 10 ms on ROS^{RT} and the native multi-threaded executor with ten threads. Fig. 9 shows the distribution of jitters under this scenario. It can be observed that ROS^{RT} is able to achieve a significant improvement from the native executor. This advantage in ROS^{RT} is again because of the contention introduced by locking in the native ROS's wait set, which is not present in ROS^{RT}.

B. Case Study: the Autoware Reference System

Since ROS^{RT} retains the original ROS interfaces, it is straightforward to integrate it into existing ROS applications. Integration only requires assigning user-defined priorities to

[10]DDS threads are created per ROS context (*i.e.*, process). Thus, creating multiple executors does not increase the number of DDS threads.

TABLE III: Timer jitters with different periods over five trials, each measured 1000 times. Data includes the 1st and the 99th percentiles and the average of the measured jitters.

	ROS^{RT} (μs)			Native (μs)			Events Executor (μs)		
Period (μs)	1%	mean	99%	1%	mean	99%	1%	mean	99%
100	-1.8	0.0	1.8	-1.6	0.0	1.6	-1.2	0.0	1.5
500	-2.2	0.0	2.1	-7.5	-0.1	7.3	-1.3	0.0	1.4
1,000	-10.4	0.0	10.4	-10.7	0.0	10.6	-10.4	0.0	10.4
10,000	-10.4	-0.1	10.2	-18.9	-0.1	19.9	-8.3	0.0	8.5
50,000	-11.1	-0.1	11.0	-22.7	-0.8	22.1	-12.6	0.0	12.8
100,000	-11.5	-0.1	10.7	-27.0	-1.3	24.1	-13.7	0.0	13.5

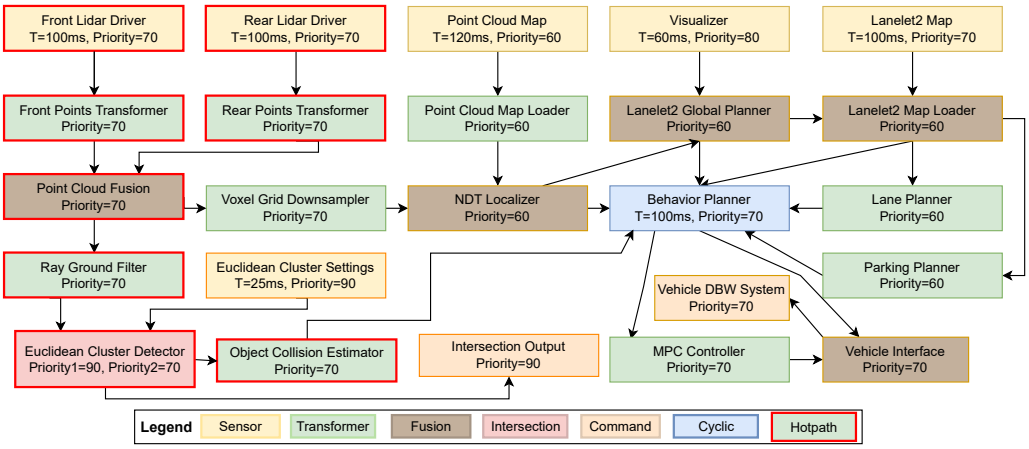

Fig. 10: The Autoware Reference System. The hot-path nodes are outlined in red.

callbacks. This can greatly improve workload schedulability and reduce computational requirements with only a handful of minor modifications.

We demonstrate ROS^{RT}'s ease of integration into an existing ROS workload to enable flexible, preemptive scheduling with user-defined priorities, by retrofitting it into the Autoware Reference System (ARS) [24].

The ARS is a performance benchmark workload for custom ROS executors and middleware that simulates the LiDAR processing graph of the popular Autoware autonomous driving system [9]. The ARS simulates LiDAR sensors with ROS nodes that send "3D pointcloud" data to a designated topic at a fixed 100ms period via timer callbacks. This leads to the invocations of downstream nodes that simulate data processing with callbacks that busy-wait for fixed durations. For instance, the Object Collision Estimator node simulates safety-critical pedestrian and vehicle detection tasks. The processed data leads to the Behavior Planner node that represents the vehicle's path planning logic. It periodically (100ms period) scans the processed LiDAR data, takes evasive action if obstacles lie in the vehicle's current trajectory, and then sends a message to its downstream nodes, including an MPC controller.

Fig. 10 visualizes the graph structure of the ARS workload, including the periods of the source nodes. In the ARS, the path of processing nodes between the LiDAR sensors and the Object Collision Estimator node is known as the "hot-path". The duration of this path dictates the latency at which the Object Collision Estimator can detect obstacles should one enter the vehicle's path. It is therefore crucial to minimize the hot-path latency to ensure safety. Custom ROS executors can accomplish this by prioritizing the hot-path nodes' subscriptions and allowing preemption of lower-priority callbacks. Thus, the hot-path latency serves as a natural performance metric to benchmark custom executors.

The nodes forming the hot-path are outlined in red in Fig. 10. We measure the hot-path latency starting at the time data has been received from either the Front and Rear LiDAR Drivers, and stopping once the Object Collision Estimator callback begins executing.

We executed the ARS with three ROS executor variants: the native multi-threaded executor, the events executor (with augmentations from Teper *et al.* [6]), and ROS^{RT} (under FP and EDF). The ARS was configured with a 4KB message size and a 1.93 ms execution time (to simulate the ARS's real-life data processing modules). For each experiment, we collected data from five trials, each running for 2 minutes. The experiments were performed as follows.

The multi-threaded executor. The number of executor threads was set to match the number of CPU cores, and the ARS was executed in its base form.

ROS^{RT} (RM). Using `SCHED_FIFO`, we set the priority of each source callback inversely proportional to its period (lower period $\rightarrow$ higher priority) to implement a rate-monotonic (RM)

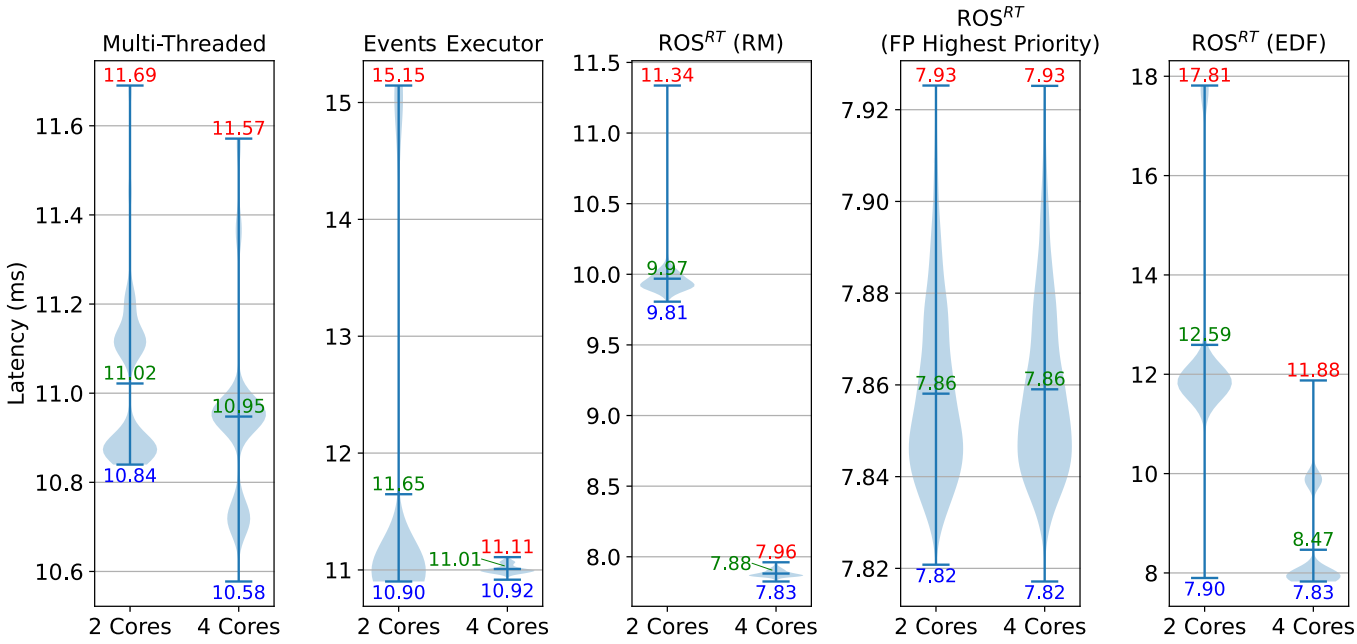

Fig. 11: The ARS hot-path latencies under three ROS variants: the multi-threaded executor, the events executor (with augmentations from Teper *et al.* [6]), and ROS^{RT}. **The y-axis ranges are not normalized for better visibility.**

scheduler. Fig. 10 lists the priority of the nodes' callbacks.

ROS^{RT} (with the highest priority). To demonstrate the flexibility of ROS^{RT}, we assigned the hot-path nodes the highest `SCHED_FIFO` priority in the system (99), while the remaining callbacks retain their priority assignment under RM.

ROS^{RT} (EDF) Using our `SCHED_EXT` EDF scheduler, we assigned implicit deadlines (deadline = period) to source callbacks. The absolute deadlines for non-source callbacks are inherited from the callbacks that invoke them.

The events executor (Teper). Because the events executor is single-threaded, we created an executor for each node shown in Fig. 10 and dispatched them in independent threads. Each thread's priority follows the RM priority assignment mentioned above. Results obtained from this method effectively represent the performance of the events executor were it to support multi-threading and preemption and thus can serve as a meaningful benchmark.

Results. We report the hot-path latency statistics for each configuration in Fig. 11. As can be observed from the results, due to its lack of ability at prioritization, the multi-threaded executor's performance did not benefit much from two additional processors. From our observation, it can also suffer from timer misalignment: the two LiDAR driver timers would not be released simultaneously, thus causing delays to the "Point Cloud Fusion" node, which requires two inputs.

By contrast, ROS^{RT} with the highest priority for the hot path performed the best as expected, as the FP scheduling prevents other nodes from interfering with the hot path execution. For this reason, the results are similar under two and four cores. With RM priority assignment, ROS^{RT} performed better than the events executor. Since both workloads are preemptive, this result demonstrates that ROS^{RT} effectively achieved preemptive scheduling without losing performance. On the other hand, ROS^{RT} with EDF scheduling performed worst. This is expected, as the hot path under EDF scheduling can be interfered with by more non-hot-path nodes than under RM scheduling. For example, the "Point Cloud Map" (and its descendants) would always have lower priority than the nodes on the hot path under RM scheduling, but it can have equal or higher priority when scheduled under EDF scheduling (every 600 ms), thus explaining the much higher latency observed, particularly when scheduling with two cores.

VI. Conclusion

In this paper, we presented ROS^{RT}, a novel ROS execution framework that enables the preemptive EDF/FP scheduling of ROS callbacks, without sacrificing ROS's flexibility. Our framework addresses many limitations in existing ROS variants and achieves lower overhead than the native ROS executor. To provide users with maximum flexibility, we implemented an EDF scheduler with `SCHED_EXT`, which does not impose any limitations on user's workload as is the case in `SCHED_DEADLINE`. Finally, our case study on the Autoware Reference System demonstrated the superiority of ROS^{RT} in a realistic scenario.

Currently, the DDS lacks mechanisms to prioritize messages (it can only prioritize topics), and therefore ROS^{RT} is unable to police the dispatch order of callback worker threads based on the scheduling policy. For future work, we would like to unify the scheduling of DDS functionalities and the OS scheduling, combining the scheduling of DDS and ROS^{RT}.

REFERENCES

[1] H. Choi, Y. Xiang, and H. Kim, "PiCAS: New design of priority-driven chain-aware scheduling for ROS2," in *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 251–263.

[2] A. A. Arafat, S. Vaidhun, K. M. Wilson, J. Sun, and Z. Guo, "Response time analysis for dynamic priority scheduling in ROS2," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, p. 301–306.

[3] H. Sobhani, H. Choi, and H. Kim, "Timing Analysis and Priority-driven Enhancements of ROS 2 Multi-threaded Executors," in *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2023, pp. 106–118.

[4] A. A. Arafat, K. M. Wilson, K. Yang, and Z. Guo, "Dynamic priority scheduling of multithreaded ROS 2 executor with shared resources," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3732–3743, 2024.

[5] K. Wilson, A. A. Arafat, J. Baugh, R. Yu, and Z. Guo, "Physics-informed mixed-criticality scheduling for f1tenth cars with preemptable ros 2 executors," in *Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2025, pp. 215–227.

[6] H. Teper, O. Bell, M. Günzel, C. Gill, and J.-J. Chen, "Reconciling ros 2 with classical real-time scheduling of periodic tasks," in *Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2025, pp. 177–189.

[7] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio, "ROSCH:Real-Time Scheduling Framework for ROS," in *Proceedings of the 24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2018, pp. 52–58.

[8] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors," in *Proceedings of the 41st IEEE Real-Time Systems Symposium*, 2020, pp. 231–243.

[9] "Home page - autoware," 2025, accessed: 2025-02-28. [Online]. Available: https://autoware.org/

[10] Object Management Group, "About the data distribution service specification version 1.4." [Online]. Available: https://www.omg.org/spec/DDS/1.4/About-DDS

[11] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling," in *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, vol. 133, 2019, pp. 6:1–6:23.

[12] Open Robotics, "Quality of service settings." [Online]. Available: https://docs.ros.org/en/humble/Concepts/Intermediate/About-Quality-of-Service-Settings.html

[13] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional DAG tasks in multiprocessor systems," in *ECRTS'15*, 2015, pp. 211–221.

[14] S. Baruah, "The federated scheduling of constrained-deadline sporadic DAG task systems," in *DATE'15*, 2015, pp. 1323–1328.

[15] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet, "Global EDF scheduling of directed acyclic graphs on multiprocessor systems," in *RTNS'13*, 2013, pp. 287–296.

[16] S. Baruah, "Improved multiprocessor global schedulability analysis of sporadic DAG task systems," in *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, 2014, pp. 97–105.

[17] Q. He, N. Guan, M. Lv, X. Jiang, and W. Chang, "Bounding the Response Time of DAG Tasks Using Long Paths," in *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, 2022, pp. 474–486.

[18] S. Ahmed and J. Anderson, "Exact Response-Time Bounds of Periodic DAG Tasks under Server-Based Global Scheduling," in *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, 2022, pp. 447–459.

[19] T. Ishikawa-Aso, A. Yano, T. Azumi, and S. Kato, "Work in progress: Middleware-transparent callback enforcement in commoditized component-oriented real-time systems," in *Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2025, pp. 426–429.

[20] J. Staschulat, I. Lütkebohle, and R. Lange, "The rclc executor: Domain-specific deterministic scheduling mechanisms for ros applications on microcontrollers: work-in-progress," in *2020 International Conference on Embedded Software (EMSOFT)*, 2020, pp. 18–19.

[21] J. Staschulat, "Micro-ROS Dispatcher Executor Pull Request," 2021. [Online]. Available: https://github.com/ros2/rclc/pull/87

[22] "Deadline task scheduling," 2025, accessed: 2025-02-28. [Online]. Available: https://docs.kernel.org/scheduler/sched-deadline.html#bandwidth-reclaiming

[23] "Extensible scheduler class," 2025, accessed: 2025-02-28. [Online]. Available: https://docs.kernel.org/scheduler/sched-ext.html

[24] Ros-Realtime, "Ros-realtime/reference-system: A reference system that simulates real-world systems in order to more fairly compare various configurations of executors and other settings." [Online]. Available: https://github.com/ros-realtime/reference-system